November 1988

Workshop on Database Programming Languages

François Bancilhon
Altaïr

Peter Buneman
University of Pennsylvania

Workshop on Database Programming Languages

Abstract

These are the revised proceedings of the Workshop on Database Programming Languages held at Roscoff, Finistère, France in September of 1987. The last few years have seen an enormous activity in the development of new programming languages and new programming environments for databases. The purpose of the workshop was to bring together researchers from both databases and programming languages to discuss recent developments in the two areas in the hope of overcoming some of the obstacles that appear to prevent the construction of a uniform database programming environment. The workshop, which follows a previous workshop held in Appin, Scotland in 1985, was extremely successful. The organizers were delighted with both the quality and volume of the submissions for this meeting, and it was regrettable that more papers could not be accepted. Both the stimulating discussions and the excellent food and scenery of the Brittany coast made the meeting thoroughly enjoyable.

There were three main foci for this workshop: the type systems suitable for databases (especially object-oriented and complex-object databases,) the representation and manipulation of persistent structures, and extensions to deductive databases that allow for more general and flexible programming. Many of the papers describe recent results, or work in progress, and are indicative of the latest research trends in database programming languages.

The organizers are extremely grateful for the financial support given by CRAI (Italy), Altaïr (France) and AT&T (USA). We would also like to acknowledge the organizational help provided by Florence Deshors, Hélène Gans and Pauline Turcaud of Altaïr, and by Karen Carter of the University of Pennsylvania.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-93.

WORKSHOP ON DATABASE PROGRAMMING LANGUAGES

Organizers:
Peter Buneman, University of Pennsylvania
Francois Bancilhon, Altair

MS-CIS-88-93

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

November 1988

WORKSHOP ON DATABASE PROGRAMMING LANGUAGES

Organizers:	François Bancilhon (Altaïr)
	Peter Buneman (University of Pennsylvania)
aided by:	Serge Abiteboul (INRIA)
	Rishiyur Nikhil (MIT)
	Atsushi Ohori (University of Pennsylvania)
	Domenico Saccà (CRAI)
	Michel Scholl (INRIA)
Sponsors:	Altaïr
	CRAI

Copies of these proceedings are available from

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

(cost $26.00, postage inc., within USA) and from

Altaïr
BP 105 - Roquencourt
78153 Le Chesnay cédex
France

FOREWORD

These are the revised proceedings of the Workshop on Database Programming Languages held at Roscoff, Finistère, France in September of 1987. The last few years have seen an enormous activity in the development of new programming languages and new programming environments for databases. The purpose of the workshop was to bring together researchers from both databases and programming languages to discuss recent developments in the two areas in the hope of overcoming some of the obstacles that appear to prevent the construction of a uniform database programming environment. The workshop, which follows a previous workshop held in Appin, Scotand in 1985, was extremely successful. The organizers were delighted with both the quality and volume of the submissions for this meeting, and it was regrettable that more papers could not be accepted. Both the stimulating discussions and the excellent food and scenery of the Brittany coast made the meeting thoroughly enjoyable.

There were three main foci for this workshop: the type systems suitable for databases (especially object-oriented and complex-object databases,) the representation and manipulation of persistent structures, and extensions to deductive databases that allow for more general and flexible programming. Many of the papers describe recent results, or work in progress, and are indicative of the latest research trends in database programming languages.

The organizers are extremely grateful for the financial support given by CRAI (Italy), Altaïr (France) and AT&T (USA). We would also like to acknowledge the organizational help provided by Florence Deshors, Hélène Gans and Pauline Turcaud of Altaïr, and by Karen Carter of the University of Pennsylvania.

François Bancilhon and Peter Buneman

Construction and Calculus of Types for Database Systems

David Stemple
Tim Sheard
Department of Computer and Information Science
University of Massachusetts, Amherst, MA. 01003

October 22, 1987

Abstract

Database systems should allow the construction of types for the kinds of complex objects used in modern applications such as design systems and artificial intelligence applications. In addition to complex structures, the type system should incorporate encapsulation and inheritance features appropriate to such applications. Furthermore, arbitrary constraint specification should be a feature of such a type system in order to bind the systems to the semantics of the occasion. Incorporating these features in a database system specification language must be done very carefully in order to produce a facility that

1. can be used effectively by database system designers
2. can be implemented efficiently
3. supports the kind of mechanical reasoning required to satisfy 1. and 2.

The ADABTPL system under development at the University of Massachusetts represents an attempt to provide the features and meet the requirements listed above. The ADABTPL type system is a crucial part of this effort and contains the following features:

- A type construction approach with embedded constraints
- Parametric polymorphic types = user-defined type constructor functions
- Encapsulated abstract data types
- Multiple inheritance
- Constraints specifiable on function input and checked at compile time (verified) on all calls
- Type conditions on type parameters

In this paper we present the ADABTPL type features and concentrate on the motivations for choosing these features and for limiting certain capabilities such as recursive types and inheritance.

1 Introduction

Database systems should allow the construction of types for the kinds of complex objects used in modern applications such as design systems and artificial intelligence applications. In addition to complex structures, the type system should incorporate encapsulation and inheritance features appropriate to such applications. Furthermore, arbitrary constraint specification should be a feature of such a type system in order to bind the systems to the semantics of the occasion. Incorporating these features in a database system specification language must be done very carefully in order to produce a facility that

1. can be used effectively by database system designers
2. can be implemented efficiently
3. supports the kind of mechanical reasoning required to satisfy 1. and 2.

The ADABTPL system under development at the University of Massachusetts represents an attempt to provide the features and meet the requirements listed above. The following aspects of ADABTPL are designed to make the system usable by database designers:

1. Schema and transaction program model of system specification
2. Database in the name space of transaction programs (no I/O)
3. Relational model a subset of the data model
4. Robust feedback on design of transactions in the presence of constraints
5. Rapid prototype capability

The mechanical reasoning required to verify that transactions obey all integrity constraints and to provide robust feedback to designers is facilitated by

1. basing the formal semantics of the schema structures on a few abstract data types – tuples, lists, finite sets, and natural numbers – that are predefined axiomatically
2. using computational logic along with the recursive function semantics of the the ADABTPL language to build a usable theory of constraints and updates of complex objects
3. using higher order theory and polymorphic types to make theorem proving more efficient.

The ADABTPL type system is an essential element in the support of both mechanical reasoning and usability, and contains the following features:

- A type construction approach with embedded constraints
- Parametric polymorphic types = user-defined type constructor functions
- Encapsulated abstract data types
- Multiple inheritance
- Constraints specifiable on function input and checked at compile time (verified) on all calls
- Type conditions specifiable on type parameters

In this paper we present features of the ADABTPL type specification language and discuss the criteria used to choose and form those features. We will take care to motivate the limitations we have placed on certain sophisticated features such as recursive types and inheritance.

A database system is specified in ADABTPL by defining the type of the database object and writing transactions to define the operations allowable on the database object. Transactions are written in the ADABTPL procedural language which is a high level set-oriented language whose name space comprises the components of the database object and the transaction input variables. The database type is specified in the ADABTPL schema language which is a type definition language that includes a predicate language for defining constraints on any type, including the types of all constituents of the database as well as the database type itself. Both procedural and schema languages include a function definition language for defining predicate and object functions. In the rest of the paper we will describe the salient features of the type definition.

2 Construction of structural types

The basic type constructors of ADABTPL can used to specify types for "simple" objects such as tuples, finite sets and lists. The first two of these constructors allow the specification of first normal form relation types. For example, the following defines a simple employee relation.

```
EmpTuple = [EmpNo: Integer, EmpName: String, EmpDept: Integer];

EmpRel = Set(EmpTuple)
```

The definition of EmpTuple uses the brackets to form a tuple type and then that type is used as input to the finite set type constructor written as a prefix function. Of course, the tuple type could have been left anonymous as in

```
EmpRel = Set([EmpNo: Integer, EmpName: String, EmpDept: Integer])
```

A tuple type may not contain a component that is either of the tuple type itself or depend in any way on it, except in the recursive union type described below.

Constraints are specified in where clauses of type defining equations. They may be specified in any definition. For example, to constrain a range for employee numbers (EmpNo) and to constrain the employee relation to be keyed on EmpNo, we write

```
EmpTuple = [EmpNo: Integer, EmpName: String, EmpDept: Integer]
   where EmpNo < 10000;

EmpRel = Set(EmpTuple) where Key(EmpRel, EmpNo)
```

These definitions illustrate two features of constraints. The first is that component names in tuple types can be used as variables in where clauses, for example, in the EmpTuple definition. Our semantic capture of component names is as axiomatized functions on the elements of the tuple type. A tuple type definition also creates an axiomatized constructor function for elements of the type. This function can have its name supplied by the user, but has been left as the default, MakeEmpTuple, in the example. The main axioms specifying the behavior of the constructor and selector functions are similar to the following for MakeEmpTuple and EmpNo.

```
   EmpNo(MakeEmpTuple(e, n, d)) = n
```

where e, n and d are variables universally quantified over their appropriate types. Thus, the constraint on the EmpTuple type corresponds to the axiom

```
   EmpNo(t) < 10000
```

for t universally quantified over the EmpTuple type.

The Key constraint on EmpRel uses another naming convention that allows the type name to stand for an element of the type in a where clause. Key is a predicate function that takes a set of tuples and a list of component names (selector functions) and returns true if the component names determine unique values over the set. Other functions, including user-defined functions, may be used in where clauses. Thus, the constraint language is open-ended. (It must be noted that the ability to reason effectively about constraints, though open, is at any time limited by the theory that has been developed by that time. The system reasons from lemmas that are kept in its knowledge base and is limited by this extendable resource (see [4]).

In order to specify interrelational constraints in a relational database, a where clause is added to the database type definition that must end any ADABTPL schema. For example, to define referential integrity for the department number in EmpRel, the following would be written.

```
EmpTuple = [EmpNo: Integer, EmpName: String, EmpDept: Integer]
   where EmpNo < 10000;

EmpRel = Set(EmpTuple) where Key(EmpRel, EmpNo)

DeptRel = Set([DeptNum: Integer, DeptName: String, NumberOfEmps: Integer]);

Database EmployeeDB: [Emps: EmpRel, Depts: DeptRel]
   where Contains(Depts.DeptNum, Emps.EmpDept) and
         For all d in Depts:
             d.NumberOfEmps = Count(All e in Emps
                                    where e.EmpDept = d.DeptNum)
```

The second constraint requires the NumberOfEmps component of all Depts tuples to be the count of Emps tuples matching in the department number components. In this we see an example of the ADABTPL predicate language including universal quantification over a set (For all), projection (denoted by the dot folowing a variable that ranges over a relation), and selection (denoted by the All phrase). Note that EmployeeDB constitutes the only identifier that plays the role of a programming language variable. In the transaction specifications that complete the database system definition, the component names of the database tuple are used as variables much as in the where clauses of tuple type definitions.

The discussion so far has given a brief view of how a simple database schema can be written in ADABTPL. The following should be observed. Declaring a tuple type does not declare a type for a collection of tuple instances. Even a declaration of a relation type does not declare that one relation of that type will be maintained in the database. The constitution of the database is declared in the database declaration that completes a schema. It is only at this point that relations and their tuples are declared to be maintained as instances. There are a number of reasons for this. The main two are a desire to maintain independence of particular semantic data models (ADABTPL is a generic data model in that it can model a large number of different semantic data models) and the desire to keep everything explicit and directly translatable into axiomatic, functional semantics. One result of this is that non-first normal form relations as well as non-relational database components are simple to specify. The following example demonstrates the ease with which non-first normal form relations and non-relational data is accommodated.

```
Task = [RequestDate: date, RequestTime: time, Requester: String,
        TaskDescr: String]

TaskQueue = Set([Priority: Integer, TaskList: List(Task)])
            where Key(TaskQueue, Priority) and
                  not (TaskList = NIL)

EmpTuple = [EmpNo: Integer, EmpName: String, EmpDept: Integer, Tasks: TaskQueue]
   where EmpNo < 10000;

EmpRel = Set(EmpTuple) where Key(EmpRel, EmpNo)

Database EmpTaskDB: [Emps: EmpsRel, TotalTasks: Integer]
```

Note that Emps is no longer a first normal form relation and that TotalTasks is not even a relation. TaskQueue defines the structure for a priority queue object containing non-empty lists of task descriptions paired with unique priorities. The design could be refined further to constrain operations on TaskQueue objects to obey queue protocol and to guarantee that the TotalTasks component of the database always reflects the total number of tasks queued for all employees(see [5]). These examples, though limited, give the essential flavor of the basic ADABTPL features for specifying the structure of databases along with integrity constraints. Advanced features of the language include refined, parametric, union, recursive and encapsulated types, most of which are used to achieve and control inheritance. We now turn to the ADABTPL means of dealing with inheritance.

3 Inheritance

Inheritance is one type's acquisition of a property by virtue of its being a subtype of another type. The fundamental property involved in inheritance is the eligibility of instances of types to be passed as arguments to functions. Other uses of inheritance are extant, e.g., as an implementation aid (allowing reusable generic code) and as part of a logic programming computational paradigm [1]. The subtype relationship among types can be based on the inclusion relationship among the types' value sets or among the operations allowed by the types. Each of these bases for subtyping has its use, and both are supplied by ADABTPL type constructors.

The simplest subtyping in ADABTPL is based on subsets of value sets and is accomplished by using the where constructor. For example,

```
  Person = [Name: String; Age: Number; Gender: (male, female)];
```

```
OldPerson = Person where Age > 80;
```

creates a subtype relationship making OldPerson a subtype of Person.

Subranges create the same kind of subtype relationship that the where clause does. For example,

```
SmallNumber = 1..9
```

makes SmallNumber a subtype of Number. Note that any SmallNumber can be used as input to any function requiring a Number, but the closure properties of functions may not be preserved. For example, although SmallNumbers can be added, the results may not be SmallNumbers.

While value set subsetting is a convenient and useful method of subtyping, it is not sufficient for building robust well controlled systems. For this we need to control inheritance in ways that speak more to the behavior of types than to the set of legitimate instances. In order to illustrate the means for controlling inheritance in ADABTPL, we now turn to a lattice capture of the subtyping achievable in ADABTPL and enumerate the type constructors and their effects on the type lattice.

4 The type lattice and its construction

It is useful to place types in a lattice based on the subtype relation, where the LUB of the structure is called UNIVERSE (the type on which almost no functions operate, but which when thought of as a set, contains all objects); and where the GLB of the structure is called EMPTY, (the type on which all functions operate, but when thought of as a set contains no objects). If x is a subtype of y then x is "lower" in the structure than y. For example, the OldPerson and Person types as defined above yield the following lattice.

```
UNIVERSE
    *
    *
 Person
    *
    *
OldPerson
    *
    *
  EMPTY
```

Thus, in general, as one moves down the structure the types have more and more functions defined on them, but the sets defined by the types have fewer and fewer elements.

Equivalent types appear as types with horizontal arcs in the lattice. For example,

```
Age = Number
```

causes

```
UNIVERSE
   * *
   *   *
   *     *
Number *** Age
   *     *
   *   *
   * *
  EMPTY
```

We will now go through the type defining constructs of ADABTPL and show their effects on the type lattice.

4.1 With

The With clause explicitly creates subtypes by adding new components to preexisting types. Thus the value set of a subtype created by a With has no overlap with its supertype. (Though the obvious projection on the subtype value set is equal to the supertype.) However, the semantics of the With construct is to allow all functions defined on the base type to be defined on the new type in addition to the new component names (which are selector functions). As an example of a subtype created using With, a Student type can be constucted from Person as follows:

```
Person = [Name: String; Age: Number; Gender: (male, female)];

Student = Person With [GPA: Number];
```

The type following the With keyword must be a tuple type. In this example it specifies a new function, GPA from Student to Number. It also declares that Student is a subtype of Person. The following type, though structurally equivalent to Student, is not considered a subtype of Person in ADABTPL.

```
Student2 =  [Name: String; Age: Number; Gender: (male, female); GPA: Number];
```

We reason that if the user wants two tuple types with nested component structure to be related by the subtype relation he will use the With clause, otherwise two similar types are not subtypes.

```
          UNIVERSE
           * * *
         *   *   *
       *     *     *
     *       *       *
   *                   *
Person     . . .    Student2
  *                      *
  *          *          *
  *          *         *
Student      *        *
   *         *       *
     *       *      *
       *     *     *
         *   *   *
           * *  *
            EMPTY
```

4.2 Parametric types

A parametric type is a new type constructor which takes types as input and returns a new type. A parametric type is to types, what functions are to objects [3]. In ADABTPL, parametric types are defined using a parenthesized parameter notation. We can identify a parametric type with the union of all types that can be produced by all substitutions for the type parameters, and place the parametric type above any type produced by supplying a concrete type for any of the type parameters. For example,

```
WeightedObject(Alpha) = [Object: Alpha; Weight: Number]
```

defines a parametric type with parameter Alpha. Alpha is a type variable and stands for any type. When it is instantiated then the expression stands for a concrete type. For example,

```
WeightedBoolean = WeightedObject(Boolean)
```

stands for

```
WeightedBoolean = [Object: Boolean; Weight: Number]
```

In addition to user-defined parametric types there are some system defined parametric types as well. The List, Set and Array types are in this class. The List and Set types take the element type as input, while the Array type takes two types as input, an index type, and an element type. List and Set types were illustrated in the section on structural types above. Array types are unremarkable in ADABTPL and are declared as in

```
Percentile = Array [0..99] of Number;
```

In the type lattice an instantiated type is a subtype of its parametric parent. For example,

```
                UNIVERSE
                  *  *
                *      *
              *          *
            *              *
WeightedObject(Alpha)  Array[0.99] OF Alpha
           *                 *
           *                 *
           *                 *
   WeightedBoolean     Array[0..99] OF Number
            *              *
              *          *
                *      *
                  *  *
                 EMPTY
```

4.3 Union

The discriminated union constructor is a case of disjunctive aggregation and creates a new supertype. All the components of the union become subtypes of the newly created type. In ADABTPL the union type is written much like the Tuple type, except that the colon is replaced with a right arrow. The colon stands for conjunctive aggregation, and the right arrow for disjunctive aggregation.

```
Atom = Union [ n -> Number; s -> String; b -> Boolean ];
```

```
        UNIVERSE
           *
           *
          Atom
         * * *
       *   *   *
     *     *     *
   *       *       *
Number   String   Boolean
   *       *       *
     *     *     *
       *   *   *
         * * *
         EMPTY
```

Of course, any subtype of Number is also a subtype of Atom. The labels n, s and b may only appear in case expressions where they are used to determine the type of an instance of a union type. For example, the following expression evaluates to a character string reflecting the base type of variable x of type Atom.

```
Case x of
     n -> "x is a number";
     s -> "x is a string";
     b -> "x is a boolean"
  end;
```

4.4 Inherits

The Inherits type forms a conjunctive aggregation with inheritance (unlike tuple types which don't support inheritance). An Inherits type is the same as a tuple type except that it is also a subtype of its components's types. This means that an Inherits tuple can be used to stand for one of its components whenever that is unambiguous. As in tupling we use the colon (:) syntax to indicate conjunction.

```
GradStudent = Inherits [t: Teacher; s: Student];
```

GradStudent now inherits all the functionality of both teachers and students. GradStudent also becomes a subtype of both Teacher and Student.

```
        UNIVERSE
         *   *
       *       *
     *           *
   *               *
Teacher          Student
   *               *
     *           *
       *       *
         *   *
       GradStudent
           *
           *
         EMPTY
```

The component labels can be used to disambiguate expressions. Suppose both Teacher and Student types have a function called F, and that X is of type GradStudent. The compiler could not disambiguate the expression "F(x)". By adding ".t" to x we cause the compiler to use the F which is defined on teachers. That is, "F(x.t)" uses the Teacher function F, and "F(x.s)" uses the Student function F.

4.5 Abstract Type

More active control of inheritance can be gained with the Abstract Type construct with its transparent (and implicit opaque) clause along with the type condition option.

When a type is defined it automatically inherits all of the operations of its defining type. Sometimes we would like the new type not to have these operations defined (for reasons such as we don't want the users of the type to see its implementation, or we would like to construct our own operations on the type, or rename the inherited ones.) For example, we may implement a queue type as a list with newly defined operations Add and Remove. We would not want the users of the type to be able to Cons elements onto a queue since that is not a queue operation.

```
Queue(Alpha) = Abstract Type

Structure List(Alpha);

Function Add(a:Alpha; s:Queue(Alpha)):Queue(Alpha);
Function Remove(s:Queue(Alpha)):Queue(Alpha);

Export Add, Remove;
end;
```

When the Abstract Type constructor is used the structure of the type is "opaque" and cannot be seen by the user. Only the functions defined in the body and functions renamed or exported can be used. Thus in the type lattice Queue(Alpha) and List(Alpha) defined as above would appear as two mutually separate types, neither being a subtype of the other even though they are share the same structure.

```
          UNIVERSE
            * *
          *   *
        *     *
      *       *
Queue(Alpha)  List(Alpha)
      *       *
        *     *
          *   *
            * *
            EMPTY
```

4.6 Transparent

If the user wishes the structure of an abstract type to be seen he may use the Transparent Structure clause. This causes the new type to inherit the functions of its basic structure. Of course, new operations can be defined as well. One might consider defining an ordered list as a list with some new operations such as Sort. For example,

```
OrderedList(Alpha) = Abstract Type

Transparent Structure List(Alpha);

Function Sort(s:List(Alpha)):OrderedList(Alpha);

end;
```

Here all the operations on lists are available on OrderedLists as well. In addition the new function, Sort, sorts an ordinary list into an ordered one. When the transparent structure is used then the new type becomes a subtype of the old type.

```
          UNIVERSE
             *
             *
             *
         List(Alpha)
             *
             *
             *
     OrderedList(Alpha)
             *
             *
             *
           EMPTY
```

4.7 Type conditions

Of course, the above type definition for OrderedLists assumes that the element type in the list, Alpha, can be ordered, which may not be the case. Thus, we must modify the type definition somewhat to restrict OrderedLists to only those element types which can be ordered. We restrict a type by using a type condition.

```
Type Condition Orderable(Alpha; before:function(Alpha, Alpha):Boolean);

Universal x,y:Alpha;
not before(x,x);
before(x,y) and before(y,z) => before(x,z);
before(x,y) => not before(y,x)
end;
```

A type condition is a predicate on types, the conjunct of the statement predicates after Universal in the example. Type conditions are used in the definition of abstract types. When used, any instantiations of the abstract type must have arguments that pass the type condition to be accepted by the compiler. If the following parametric Abstract Type declaration is present,

```
   OrderableList(Alpha,less: function(Alpha,Alpha) -> Boolean) =
     Abstract Type

   Type Condition Orderable(Alpha,less)

   Transparent Structure List(Alpha);

   Function Sort(s:List(Alpha)):OrderedList(Alpha);

   Export Sort;
   end;
```

then declaring

```
   NumbersList = OrderableList(Number,<)
```

causes the compiler to check several things. First that the (infix) less than function has the correct type, i.e., is a function from Number X Number to Boolean. And second that it meets the three conditions of the type condition, namely that it is areflexive, transitive, and antisymmetric. Type conditions are similar in effect and use to Goguen's theories [2].

4.8 Abstract Type and the Inherits type

The Structure clause in an Abstract Type declaration can be an Inherits type. It is the means by which we can gain some control over multiple inheritance. Inside the Abstract Type body the type defined is a type which has all the functions of all its parent types defined on it. These functions can then be exported or renamed to make a new type with only those functions the user wants being visible. For example, consider a graphics terminal system. One of the types might be a box which is drawn on the screen. The second may be some sort of sequential file. One might define a Window as a type which has both the properties of a SequentialFile(character) and a box. That is, one could read or write from or to it as well as move it about on the screen.

```
Window = Abstract Type
Structure Inherits [b: Box; f: SequentialFile(Character)]

Function1 . . .

Function2 . . .

Exports close.f as CloseWindow, ...
end;
```

The functions defined on Windows must be exported since the Inherits clause is opaque. If both Window and SequentialFile have a function with the same name, we disambiguate the function by using the labels in Inherits clause. Thus close.f means the "close" function on files, while close.w is the Window "close" function.

If the Structure clause is Transparent then all functions of both types are seen by the system as valid functions on the new type. Sometimes we wish to combine several types and add a few new features as well. This can be done by using an Inherits type and a With clause.

```
Window1 = Abstract Type
Transparent Structure Inherits [b: Box; f: SequentialFile(character)]
           With [visible: Boolean];

Exports f.close as CloseWindow, w.close as EraseWindow;
end;
```

In this example, Window1 is both a box and a SequentialFile. All the operations are available, as well as a new function called "visible". The two functions named Close are renamed so as to remove all ambiguity.

In Abstract Types only the Transparent clause creates subtypes. In the case of the first Window type above only those functions specifically exported are available.

```
          UNIVERSE
           * * *
         *   *   *
        *     *     *
      *       *       *
  File      Box     Window
      *       *        *
        *     *        *
          *   *        *
            * *        *
          Window1      *
               *       *
                 *     *
                   EMPTY
```

4.9 Recursive Union

Theoretically there is no problem with a type definition referencing the type being defined. Properly constructed recursive types have well defined semantics and are useful in specifying types which have as substructures elements of the same type as themselves. Lists and sets are examples of non-problematic recursive types. From a practical point of view, recursive definitions which reference themselves through a long chain of mutually recursive types can be hard to type check and reason about. For this reason ADABTPL allows only one kind of recursive type, the recursive union. The recursive union is a disjunctive aggregation with recursion. Its syntax is similar to the normal union type, except the types of the discriminants can involve expressions involving the type being defined or the type Bottom, i.e., the type consisting of constants and the equality relation (used especially for the unconstructable elements of types, such as nil and empty set). In ADABTPL if the discriminant has type Bottom, than the discriminant also names a nullary (constant) function which returns the (unique) element of the union with the bottom discriminant. A recursive union defines a new type that is not a subtype of any other type other than UNIVERSE. The classical example is the list which could be defined structurally by

```
List(Alpha) = Recursive Union [ nil -> Bottom;
                                dtpr -> Cons [Car:Alpha;
                                              Cdr:List(Alpha)]];
```

This uses the named tuple constructor option, for Cons, mentioned in the earlier discussion of tuple types. Typing a discriminant with Bottom is a shorthand for typing it as a componentless tuple with the discriminant as the constructor function. For example, nil could be defined by

```
nil -> nil []
```

using the same way of introducing the constructor function as was used for Cons, i.e., preceding the left bracket. Note that this defines "nil" as a nullary function (constant) and we can write "x = nil" rather than the more cumbersome

```
 Case x of nil-> true ; dtpr-> false end
```

to test if one has reached the "bottom" of a recursive structure.

5 Subtle points about function and array types

ADABTPL subtyping has two subtle points which are not obvious to the casual observer. They involve types created with the Array constructor and function types. Consider the two (false) subtype assertions where it is known that Gamma and Delta are subrange types and Gamma is a subtype of Delta.

```
F = Function(Gamma) -> Beta is a subtype of  G = Function(Delta) -> Beta

Array[Gamma] of Alpha is a subtype of Array[Delta] of Alpha
```

These two subtype expressions are false because types they compare don't meet the semantics of subtyping. Roughly speaking if A is a subtype of B, then anywhere in a program an object of type B is expected, an object of type A could be used without causing an error. In the example above we should be able to use a function of type F wherever we can use a function of type G, if F was a subtype of G. But since the domain type of F (as a set of objects) has fewer objects than G, there may be some objects in the domain type of G not on which F is not defined. Thus for two functions to be subtypes of each other, the normal subtyping order of the domain types is reversed (while it remains the same for the range type.)

```
Function(Delta) -> Alpha is a subtype of Function(Gamma) -> Beta
```

if and only if

```
Gamma is a subtype of Delta  and  Alpha is a subtype of Beta
```

If F is a Subtype of G, Then F must be defined everywhere G is (and possibly more places), but return only a subset (perhaps the same set) of objects G does.

A similar thing happens with the index parameter type of arrays. The index parameter (which has to be a number or enumerated type or a subrange of one of these) does not participate in the normal subtyping order either. In other words,

```
Array[Delta] of Alpha is a subtype of Array[Gamma] of Beta
```

if and only if

```
Gamma is a subtype of Delta  and  Alpha is a subtype of Beta
```

For example,

```
Array[1..100] of Alpha is a subtype of Array[20..50] of Alpha
```

This is because the array access function for the array with indexes from 1..100 is defined everywhere over the array with indexes from 20..50. Thus an access to the smaller array (in terms of the range of the index) can be used anywhere an access to the larger array can be used.

```
                      UNIVERSE
                      ** * *
                    * *  *   *
                  * *    *     *
                *   *    *       *
              *    *     *         *
Function(Gamma) Delta  Beta  Array[20..50] OF Alpha
 -> Beta    *     *      *         *
            *     *      *         *
            *     *      *    Array[1..100] OF Alpha
            *   Gamma   Alpha      *
            *     *      *         *
Function(Delta)   *      *    Array[1..100] OF Boolean
 -> Alpha    *    *      *         *
               *  *      *       *
                 *  *    *     *
                   *  *  *   *
                     *  ** *
                      EMPTY
```

6 The subtyping algorithm

We now give a very rough outline of our subtype algorithm. In this version Subtype is a predicate of x and y that returns either true or false, True if x is a subtype of y and False otherwise. This algorithm is for the simple case where all parameterized types are fullly instantiated. If x and y are allowed to contain type variables, then the algorithm must return a unifier binding the type variables to concrete types. It cannot be a simple predicate.

```
Function Subtype(x,y:types):Boolean;
begin

If (y=top) or (x=bottom)                      -- The Primitive Cases
   then true
   else if (x=top) or (y=bottom)
           then false
else if HasTheSameStructure(x,y)           -- Both have the same STRUCTURE
        then case x.structure
            Function: Subtype(y.inputType,x.inputType) and
                      Subtype(x.outputType,y.outputType);

    NumericSubrange: (x.low >= y.low) and (x.high <= y.high);

 EnumeratedSubrange: sameEnumeration(x,y) and
                      (x.low >= y.low) and (x.high <= y.high);

               array: Subtype(y.indexType, x.indexType) and
                      Subtype(x.elementType, y.elementType);
```

```
              tuple: match(x.names,y.names) and
                     for each t in x.types
                         Subtype(t,coresponding(y.types));

              union: subset(x.labels,y.labels) and
                     for each t in x.types
                         Subtype(t,coresponding(y.types));

           inherits: subset(y.labels,x.labels) and
                     for each t in y.types
                         Subtype(t,coresponding(x.types));

               with: Subtype(x.baseType, y.baseType) and
                     Subtype(x.extensionType, y.extensionType);

              where: Subtype(x.baseType, y.baseType) and
                     (x.whereClause) => (y.whereClause)
                     end;
        --  The RECURSIVE CASES
   else if x.type=where then Subtype(x.baseType,y)
   else if y.type=where then Subtype(x, y.baseType)
   else if x.type=with  then Subtype(x.baseType,y)
   else if y.type=union then for some t in y.types Subtype(x,t)
   else if x.type=inherits then for some t in x.types Subtype(t,y)
   else if x.type=NumericSubrange then Subtype(Number,y)
   else if x.type=EnumeratedSubrange then Subtype(enumeration(x),y)

   else if (x.type=userDefined) and (x.visiblity=transparent)
           -- IF USER DEFINED, USE DEFINITION IF TRANSPARENT
           then Subtype(expand(x), y)
   else if (y.type=userDefined) and (y.visiblity=transparent)
           then Subtype(x,expand(y))
   else false
end
```

7 Summary

We have presented the type construction facilities of the ADABTPL system being developed at the University of Massachusetts. We have concentrated on the effects of the type constructors on the lattice formed by the subtype relation produced by use of the constructors. The contribution of this work is to integrate in a usable manner sophisticated inheritance and encapsulation mechanisms with a robust structural definition facility that is a felicitous evolution of the database schema paradigm.

Acknowledgments

This research was supported by NSF grants DCR-8503613 and IST-8606424 and by the Office of Naval Research University Research Initiative contract number N00014-86-K-0764.

References

[1] Ait-Kaci, H., *A Lattice-Theoretic Approach to Computation Based on on a Calculus of Partially Ordered Type Structures*. Ph. D. Thesis, University of Pennsylvania, 1984.

[2] Burstall, R. M. and Goguen, J. A., Putting Theories together to Make Specifications. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August, 1977, pps. 1045-1058.

[3] Cardelli, L. and Wegner, P., On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, Vol. 17, No. 4, December, 1985, pps. 471-572.

[4] Sheard, T. and Stemple, D., Automatic Verification of Database Trasaction Safety. University of Massachusetts Computer and Information Science Technical Report 86-30. (submitted for publication)

[5] Stemple, D., Sheard, T. and Bunker, B., Abstract Data Types in Databases: Specification, Manipulation and Access, Proceedings of the IEEE Second International Conference on Data Engineering, Los Angeles, California, February 1986, pps. 590-597.

On a Distinction between Congruence Closure and Unification*

Paris Kanellakis
and
Peter Revesz

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

Abstract: In this note we show that single pair congruence closure on dags is in NC^2. This is the problem of computing the congruence closure of an equivalence relation C on the nodes of a directed *acyclic* graph, where $||C|| = 1$. We say that $||C|| = n$ when C is the reflexive, symmetric and transitive closure of n pairs of distinct vertices and n is minimum. Our observation distinguishes congruence closure from unification closure (its directional dual) since single pair unification closure on dags is log-space complete for $PTIME$. In addition, we show that computing the congruence closure on dags is log-space complete for $PTIME$ when $||C|| = 3$.

1 Introduction

Congruence closure and unification are fundamental operations for symbolic computation. They form the basis of interpreters for logic programming languages and many object-oriented programming languages. The two operations exhibit a certain directional duality, namely, congruence closure is defined in a bottom-up and unification in a top-down fashion. In this note we will highlight a distinction between them.

Let $G = (V, A)$ be a directed graph such that each vertex $v \in V$ has 0 or 2 ordered children. Let C be any equivalence relation on V. The *congruence closure* C^* of C is the finest equivalence relation on V that contains C such that for all vertices v and w with corresponding children v_1, w_1 and v_2, w_2 we have:

$$(v_1, w_1), (v_2, w_2) \in C^* \Longrightarrow (v, w) \in C^*$$

Congruence closure is common in decision procedures for formal theories, where it is necessary to determine equivalent expressions. An important use is in solving the following expression equivalence problem, which is called the *uniform word problem for finitely presented algebras:* determine whether an equality $t_1 = t_2$ logically follows from a set of equalities $S = \{t_{11} = t_{12}, t_{21} = t_{22}, \cdots, t_{k1} = t_{k2}\}$,

*This research was supported by NSF grant IRI-8617344. The work of the first author was also supported by an Alfred P. Sloan Foundation Fellowship.

where the t's are terms constructed from uninterpreted constant and function symbols. For this application the directed graph G is *acyclic.*

A special case of the uniform word problem for finitely presented algebras occurs in compiling; it is the well-known *common subexpression elimination problem* where the S above is empty. Downey and Sethi [2] have considered another version of this problem that arises in verifying a restricted class of array assignment programs and is relevant to our exposition. In their application, the S above contains only a single equality.

Kozen has shown that computing the congruence closure of a relation is log-space complete for $PTIME$ [7]. Several authors have suggested algorithms for congruence closure. Downey, Sethi and Tarjan [3] have the fastest known sequential algorithms. Their algorithm for the general case requires $O(nlogn)$ time in the worst case, where n is the number of vertices of G. As defined above G has $O(n)$ arcs. They also give linear time (and therefore optimal sequential) algorithms for some special cases. One such case is the problem that is of interest to us here; namely, single pair congruence closure on dags, where G is acyclic and C contains at most one pair of distinct vertices.

The congruence closure problem could also be defined for directed graphs in which vertices have other than 0 or 2 children. This more general congruence closure reduces to the special definition used here. Moreover, our choice of definition allows a clean comparison between congruence closure and its directional dual, unification closure[1], which is defined as follows.

Let $G = (V, A)$ be a directed graph such that each vertex $v \in V$ has 0 or 2 ordered children. Let C be any equivalence relation on V. The *unification closure* C^+ of C is the finest equivalence relation on V that contains C such that for all vertices v and w with corresponding children v_1, w_1 and v_2, w_2 we have:

$$(v_1, w_1), (v_2, w_2) \in C^+ \Longleftarrow (v, w) \in C^+$$

Unification arises in several important problems such as testing equivalence of finite automata and resolution theorem proving [1], [9], [11].

Computing unification closure is shown to be log-space complete for $PTIME$ in [4]. This is true even when the terms to be unified are linear, that is no variable appears more than once per term [5]. However, unification closure when one of the terms is linear and the two terms share no variables is in NC^2.

The class NC [10] contains the problems that are solvable on a PRAM ([6]) in polylogarithmic parallel time with a polynomial number of processors. NC^2 is the subclass of NC restricted to $O(log^2 n)$ parallel time. A problem is log-space complete for $PTIME$ when every problem in $PTIME$ is log-space reducible to it. By the parallel computation thesis, which was proven for PRAM's [6], any log-space reduction must be in NC. Hence, unless $PTIME \subseteq NC$, which would be an unlikely result of complexity theory, problems log-space complete for $PTIME$ do not have NC algorithms (i.e., parallel algorithms with significant speed-ups).

The goal of this paper is to identify important special cases of congruence closure that are in NC. We show that single pair congruence closure on dags is in NC^2. Since single pair unification closure on dags is log-space complete for $PTIME$ [4], [5], this provides a nontrivial distincion between the two closures. Section 2 contains the formal definitions used. In Section 3 we show that when C is the trivial equivalence relation, that is each distinct vertex is an equivalence class, then congruence closure is in NC^2 (Theorem 1). The tricky issue here is the existence of cycles in directed graph G. The acyclic case was already known to be in NC^2 via common subexpression elimination for

[1]The most general unifiers of two terms[11] can be computed easily from their unification closures.

directed acyclic graphs. In Section 4 we show that when C has at most one pair of distinct vertices and G is acyclic the congruence closure is in NC^2. This is the main result of our note (Theorem 2). Finally, in Section 5 we show (by a simple modification of Kozen's proof in [7]) that congruence closure is log-space complete for $PTIME$ when C has two pairs of distinct vertices (Theorem 3). If we require that G be acyclic, then we need three pairs of distinct vertices in C.

2 Definitions

An ordered directed graph is a directed graph G in which each node's children are ordered. In many applications, e.g. when terms are represented by ordered directed graphs, we get ordered directed *acyclic* graphs. We now define congruence closure and unification closure.

Definition CC *Let $G = (V, A)$ be an ordered directed graph. Let C be any equivalence relation on V. The congruence closure C^* of C is the finest equivalence relation on V that contains C such that for all vertices v and w with children $v_1, v_2, \cdots, v_k$ and $w_1, w_2, \cdots, w_l$, respectively, if $k = l \geq 1$, then:*

$$(v_i, w_i) \in C^* for 1 \leq i \leq k, \Longrightarrow (v, w) \in C^*$$

Definition UC *Let G = (V,A) be an ordered directed graph. Let C be any equivalence relation on V. The unification closure C^+ of C is the finest equivalence relation on V that contains C such that for all vertices v and w with children $v_1, v_2, \cdots v_k$ and $w_1, w_2, \cdots w_l$, respectively, if $k = l \geq 1$, then:*

$$(v_i, w_i) \in C^+ for 1 \leq i \leq k, \Longleftarrow (v, w) \in C^+$$

We refer to the problem of finding the congruence closure for an ordered directed acyclic graph as *dag-CC* and for an ordered graph where each node has exactly 0 or 2 ordered children as CC^2. Similarly we have *dag-UC* and UC^2. Let C be the input equivalence relation. We use $||C|| = n$ when n is the smallest natural number, such that, C is the reflexive, symmetric and transitive closure of n pairs of *distinct* vertices. In general C will have many pairs of equal vertices, one for each node of G.

We now state some propositions from the literature that relate the various cases of CC and UC described. We use the standard notion of log-space reduction, $\leq$.

Proposition 1 *(dag-)CC when $||C|| = n \leq$ (dag-)CC^2 when $||C|| = n$.*

This follows from Downey, Sethi, and Tarjan's reduction method [3], which preserves $||C||$ and works for dags as well as in general. It is also known (see [9]) that,

Proposition 2 *(dag-)UC when $||C|| = n \leq$ (dag-)UC^2 when $||C|| = 1$.*

Note that *(dag-)*UC^2 when $||C|| = 1$ is known to be log-space complete for $PTIME$ [4], [5].

Let u and v be vertices of an ordered directed graph. We write $u \equiv v$, when u and v are the same. Similarly, we write $u \approx v$ and read u and v are *congruent*, when $(u, v) \in C^*$. We will now inductively define a symmetric relation E on pairs of vertices of G. This relation is represented by undirected edges added to G, i.e. we will write uEv when u and v are connected by an undirected edge.

Definition E *Let u and v be vertices of G. Then add uEv*

1. *If (u, v) is a pair in C.*
2. *If u and v have children $u_1, u_2, \cdots, u_k$ and $v_1, v_2, \cdots, v_l$, respectively, with $k = l \geq 1$, and u_iEv_i holds for all i, $1 \leq i \leq k$. In this case, when u and v are distinct adding an undirected edge between u and v is called a propagation step and is denoted by uPv.*
3. *If there is a vertex w in G such that uEw and wEv hold. In this case, when u, v, and w are distinct adding an undirected edge between u and v is called a transitivity step and is denoted by uTw.*

A *proof* of $x \approx y$ is a (possibly empty) sequence of propagation and transitivity steps using the equalities in C that makes xEy true.

Proposition 3 *$x \approx y$ iff it has a proof.*

Proof: See Kozen [7].

3 Congruence Closure when $||C|| = 0$

In this section, we show the following theorem.

Theorem 1 *When $||C|| = 0$ congruence closure is in NC^2.*

Proof: Our proof of this theorem is based on the following two observations. First, when $||C|| = 0$, then any one pair of vertices in C^* can be proven congruent by using propagation steps only. Second, sequences of propagation steps can be done in NC^2. By Proposition 2, it is enough to look at CC^2. The difficulty is that the directed graph used may have cycles.

To show that the first observation is true, suppose that the following congruences are counterexamples, that is all their proofs use some transitivity steps:

$$x_1 \approx y_1, x_2 \approx y_2, \cdots, x_m \approx y_m$$

Let $x_i \approx y_i$ be the congruence with a minimum length shortest proof, and let k be that length (i.e, the number of propagation and transitivity steps). Now look at the transitivity steps in that proof of $x_i \approx y_i$. There are two cases.

1. The last step is a transitivity step.
2. The last step is a propagation step. Then there must be other steps that are transitivity steps $s_1, s_2 \ldots s_l$ with $1 \leq l < k$. Notice that the minimality used above implies that each of the results of these transitivity steps have proofs of length less than k. Hence these have proofs using only propagation steps. Therefore in this case $x_i \approx y_i$ is impossible, since it implies that it has a propagation only proof.

In fact, reasoning as in the second case above, $x_i \approx y_i$ has a proof such that the last step is a transitivity step while the rest of the steps are replaced by propagation steps only. Now we will show the following claim.

Claim: When C is empty, if $v \approx w$ has a proof with only one transitivity step at the end, then it has a proof that uses propagation steps only.

We show that the claim holds by induction on the length of proofs. This claim will imply that the $x_i \approx y_i$ above has a proof of only propagation steps. This contradiction demonstrates that propagation steps suffice.

Base: For proofs of length 1, the claim must hold, because transitivity can not be the first step.

Induction hypothesis: If $v \approx w$ has a proof of length $j \geq 1$, with only one transitivity at the end, then it has a propagation proof.

Suppose $v \approx w$ has a proof of length $j + 1$ with only one transitivity at the end. W.l.g. the sequence must look as follows:

$$P_1, vPu, P_2, uPw, P_3, vTw$$

where P_1, P_2 and P_3 are (possibly empty) sequences of propagation steps.

Since $v \approx u$ and $u \approx w$ were proven by propagation, all of v, u, w must have exactly two children. This is because $\|C\| = 0$. Let their children be v_1, v_2, u_1, u_2, and w_1, w_2, respectively. In addition, if $v_1 \not\equiv u_1$, then v_1Pu_1 must precede vPu, and if $v_2 \not\equiv u_2$, then v_2Pu_2 must precede vPu. Similarly, if $u_1 \not\equiv w_1$, then u_1Pw_1 must precede uPw, and if $u_2 \not\equiv w_2$, then u_2Pw_2 must precede uPw. Therefore, if $v_1 \not\equiv w_1$ we can replace the end of the original sequence to get

$$P_1, vPu, P_2, v_1Tw_1$$

Similarly, if $v_2 \not\equiv w_2$ we can also replace the end of the original sequence to get

$$P_1, vPu, P_2, v_2Tw_2$$

Both of these sequences are proofs. Since both proofs have length less than $j + 1$ with only one transitivity at the end, by the induction hypothesis both $v_1 \approx w_1$ and $v_2 \approx w_2$ have propagation proofs. Therefore $v \approx w$ also has a propagation proof. Hence the claim must hold.

The correctness of our second observation follows from the fact that transitive closure is in NC^2, and from the following reduction from CC^2 with $\|C\| = 0$ and $G = (V, A)$ to transitive closure of the directed graph $G' = (V', A')$, where $V' = \{(v, w) : v, w \in V\} \cup \{t\}$, and A' is as follows:

1. For all $v \in V$ let (v, v) have child t in G'.
2. For all $v, w \in V$ with children v_1, v_2 and w_1, w_2, respectively, let (v, w) have children (v_1, w_1) and (v_2, w_2) in G'.

Then for all $v, w \in V$, $v \approx w$ if and only if the following conditions both hold:

1. Each successor of (v, w) has t as a successor (except for t itself).
2. The successors of (v, w) form an acyclic graph.

The correctness of this reduction can be easily proven by induction on the length of propagation sequences. □

4 Congruence Closure when $||C|| = 1$

The single pair congruence closure problem seems harder than congruence closure with $||C|| = 0$. Given a directed acyclic graph like that in Figure 1, we need transitivity steps, to show for example that $x \approx z_1$. Moreover, to show that $x \approx z_i$, we need i alternations in propagation and transitivity steps. However, since x does not have any children, in this case we could *merge* it with y and then perform propagation and transitivity steps. In this way the problem reduces to the $||C|| = 0$ case, and only propagation steps are needed. The next theorem shows that this can be done in general.

Theorem 2 *When $||C|| = 1$ dag congruence closure is in NC^2.*

Proof: Let $G = (V, A)$ be any directed acyclic graph, x and y two *distinct* vertices in V and C the reflexive, symmetric and transitive closure of (x, y). By Theorem 1, we can "eliminate common subexpressions", and transform $G = (V, A)$ into $G' = (V', A')$, where no two vertices have proofs under the equivalence relation $\{(z, z) : z \in V\}$. Now we will compute the congruence closure of C on G'. If x and y are still distinct vertices, then w.l.g. x and y are incomparable or x is a descendant of y. In either case we can pick an arbitrary ordering of the vertices $N(x)$. We can assume that $N(y) > N(x)$, and the following claim can be proven.

Claim: When G' is acyclic and $N(y) > N(x)$, if $u \approx v$ holds, then $N(u), N(v) \geq N(x)$.

The claim is shown by induction on the length of the proof for $u \approx v$.

Base: Suppose $u \approx v$ has a proof of length 1. Then it must be a propagation proof. Let u_1 and u_2 be the children of u, and v_1 and v_2 be the children of v. Since $u \not\approx v$ when $||C|| = 0$, the proof must depend on $x \approx y$. Then w.l.g $(u_1, v_1) = (x, y)$. Therefore, $N(u), N(v) > N(x)$, and the lemma holds for proofs of length 1.

Induction hypothesis: If $u \approx v$ has a proof of length $k \geq 1$, then $N(u), N(v) \geq N(x)$.

Then suppose $u \approx v$ has a proof of length $k + 1$. There are two cases:

1. The last step was transitivity of the form: $uEw, wEv \Longrightarrow uTv$. Then $u \approx w$ and $w \approx v$ have proofs shorter than $k + 1$, hence by the induction hypothesis, $N(u), N(v), N(w) \geq N(x)$. Hence $N(u), N(v) \geq N(x)$.

2. The last step was propagation. Then $u_1 \approx v_1$, and $u_2 \approx v_2$ have proofs shorter than $k + 1$, hence by the induction hypothesis, $N(u_1), N(u_2), N(v_1), N(v_2) \geq N(x)$. Since the graph is acyclic, $N(u), N(v) \geq N(x)$ also holds.

In both cases the claim holds for $k + 1$, hence by induction it must hold for proofs of any size.

Since nodes that are greater than x cannot use the children of x, by the above claim, we can make the children of x be the same as the children of y. This modification of G' will not change the computation of the congruence closure but will allow us to merge vertex x and y and still get an ordered directed graph. Then the problem reduces to congruence closure with $||C|| = 0$, which by Theorem 1 is in NC^2. This completes the proof of Theorem 2. □

5 Congruence Closure when $||C|| \geq 2$

Theorem 3 *CC when $||C|| = k$ and dag-CC when $||C|| = k + 1$ for arbitrary fixed $k \geq 2$ is logspace complete for PTIME.*

Proof: The proof is by a reduction from the circuit value problem (CVP) which was proven logspace complete for $PTIME$ by Ladner [8]. The circuit value problem is a sequence $g_1, g_2, \ldots, g_n$, where each g_i is either (i) a Boolean variable, which is assigned true or false, or (ii) $NOR(j,k)$, with $j, k < i$. The circuit value problem operation is: for a given circuit and an assignment to the variables find the output of the circuit.

To do the reduction, we introduce two special vertices 1 and 0. Every boolean variable g_i that is assigned true is assigned to 1, and every boolean variable g_i that is assigned false is assigned to 0. In addition, for each g_i that is not a variable we create a vertex with first child g_j and second child g_k. We can encode into the congruence closure problem the function of a NOR gate by adding three congruences in Figure 2. Out of these the congruence $0 \approx z$ can be eliminated by merging the vertices 0 and z (see Figure 3).

Now it is easy to prove by induction that the CVP is true if and only if the node g_n will be congruent to 1. Hence the CVP problem can be reduced to dag congruence closure with $||C|| = 3$ and to congruence closure with $||C|| = 2$. The cases for $k > 2$ are also immediately implied. □

6 Open Problems

An open problem is to decide the status of dag-congruence closure when $||C|| = 2$. This log-space reduces to congruence closure with $||C|| = 1$, which is also open.

References

[1] Clocksin, W.F., Mellish, C.S., *Programming in Prolog*, Springer-Verlag, 1981.

[2] Downey, P.J., Sethi, R., Assignment Commands with Array References, *J. ACM 25*, 4 (1978), pp. 652-666.

[3] Downey, P. J., Sethi, R., and Tarjan, R. E., Variations on the Common Subexpression Problem, *J. ACM 27*, 6 (1980), pp. 758-771.

[4] Dwork, C., Kanellakis, P., Mitchell, J., On the Sequential Nature of Unification, *Journal of Logic Programming 1* (1), pp. 35-50.

[5] Dwork, C., Kanellakis, P., Stockmeyer, L., Parallel Algorithms for Term Matching, IBM Research Report, RJ 5328, (to appear in *SIAM Journal of Computing*).

[6] Fortune, S., Wyllie, J., Parallelism in Random Access Machines, *Proc.* 10th *ACM STOC*, (1978) pp. 114-118.

[7] Kozen, D., Complexity of Finitely Presented Algebras, *Proc.* 9th *ACM STOC*, (1977) pp. 164-177.

[8] Ladner, R., The Circuit Value Problem is Log Space Complete for P, *SIGACT News 7*, 1, (1975) pp. 18-20.

[9] Paterson, M. S. and Wegman, M. N., Linear Unification, *JCSS* 16, (1978) pp. 158-167.

[10] Pippenger, N., On Simultaneous Resource Bounds, in *Proc.* 20th *IEEE FOCS*, (1979) pp. 307-311.

[11] Robinson, J. A., A Machine Oriented Logic Based on the Resolution Principle, *J. ACM 12*,1 (1965) pp. 23-41.

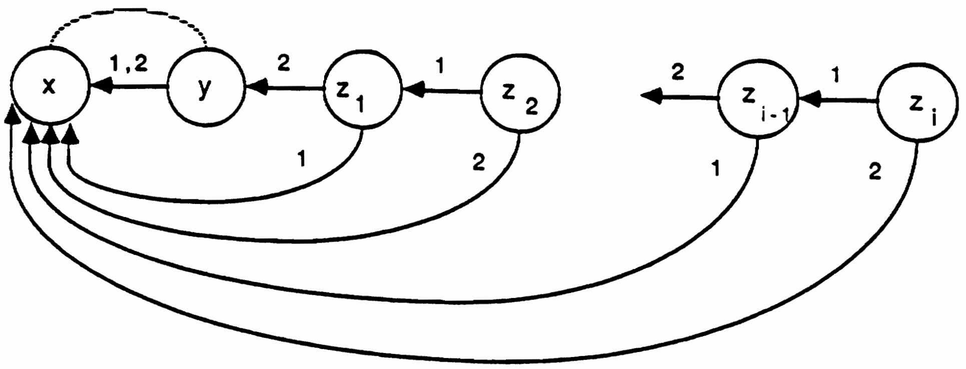
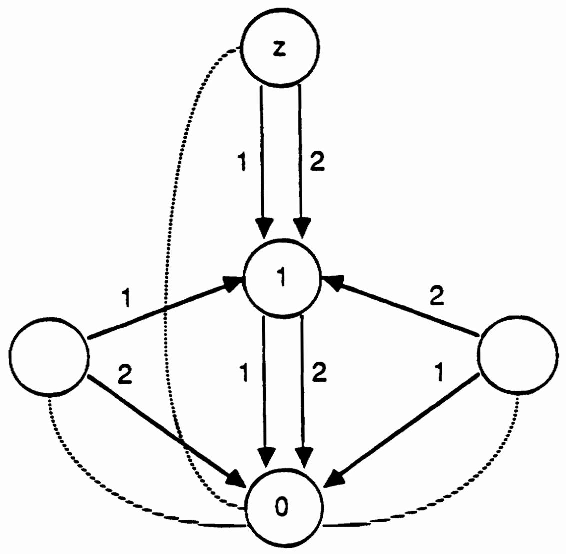


Figure 1

Example of dag-CC instance when ||C|| = 1



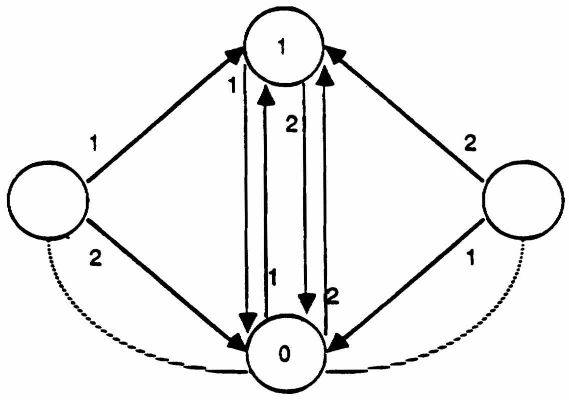


Figure 2

Example of dag-CC instance when ||C|| = 3

Figure 3

Example of CC instance when ||C|| = 2

Class Hierarchies and Their Complexity

Maurizio Lenzerini

Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
via Buonarroti 12, I-00185 Roma, Italy

Abstract

Object oriented systems allow various kinds of relationships to be established among objects. In this paper we are concerned with membership and interdependency relationships. *Membership* is the relationship holding between an object and a class it belongs to. *Interdependency relationships* are used to assert that a certain set relation holds among the extensions of a collection of classes. Our main goal is to present a taxonomy of membership and interdependency relationships, and to sudy the computational complexity of reasoning about them. To this end, we introduce the concept of class hierarchy scheme, which is intended to represent a set of membership and interdependency relationships, and we study the inference problem for class hierarchy schemes, which is the problem of checking if a given relationship logically follows from a set of membership and interdependency relationships. We also study a subclass of class hierarchy schemes, presenting efficient algorithms for the inference problem in such a subclass.

1. Introduction

A fundamental feature of an object oriented database system is to provide modeling primitives for establishing relationships among objects. One of them is classification, which allow objects to be grouped into classes. A *class* represents a set of objects with common properties, called its instances. The set of instances of a class is referred to as its extension. *Membership relationship* is the relationship holding between an object α and any class whose extension includes α.

Various kinds of relationships can be established among classes. An important role is played by the so-called *interdependency relationships* ([Israel 84]), which allow to assert that a certain set relation holds among the extensions of a collection of classes. For example, disjointness is the interdependency relationship holding between two or more classes having no common instances.

In object oriented systems, a class is also defined in terms of behavioral properties, as well as aggregations with other classes; however, these aspects are not dealt with in this paper.

Database languages and models include many types of interdependency relationships, such as the is-a rèlationship (see [Buneman 86] and [Albano 85]), which is used to specify inclusion between the extensions of two classes. Several recent works (see [Atzeni 86], [Atzeni 87], [Lenzerini 87], [Arisawa 86], and [McAllester 86]) have considered more complex interdependencies, with the main goal of devising sound and complete rules for their inference.

In [Atzeni 86] a set of inference rules, with correponding algorithms, is presented for is-a and binary disjointness relationships between classes. The work is extended in [Atzeni 87], where negative statements, assserting that a given binary interdependency relationship does not hold, and class complementation, allowing the definition of a class as the complement of another class, are taken into account.

In [Lenzerini 87], covering relationships among classes, holding when a class is a subset of the union of other classes, are considered, and an algorithm for covering relationship inference is provided. Also, the interaction with disjointness relationships is analysed.

In [Arisawa 86], two interdependency relationships, called intersection extended generalization and union extended generalization, are introduced. The first one allows to define is-a relationships in which intersections of classes are involved. The second one allows union of classes to explicitely referenced in the is-a relationships. For both of these classes, a set of sound and complete inference rules are presented. In the same paper a further type of constraint, expressing that a given disjointness constraint does not hold, is considered, and its interaction with both intersection and union extended generalization is studied.

In [McAllester 86] the usual notion of class is extended to take into account classes which are Boolean combinations of other classes. Interdependency relationships can be expressed in the form (C implies B), where C is a primitive class, and B is a Boolean class expression. Such a relationship specifies that every instance of C is also an instance of B. Membership relationships between objects and primitive classes are also considered. A primitive class C is said to inherit from a class expression E under a set of interdependency relationships S, if (C implies E) is a logical consequences of S. The major goal of the paper is to propose a method for computing inheritance.

In this paper we present a classification of class interdependencies, together with a complexity analysis of reasoning (i.e. performing inferences) about them. To this end, we define the concept of *class hierarchy scheme (CHS)*, which is intended to represent a set of membership and interdependency relationships in an object oriented system. Tipical inferences which are performed on a CHS T include:

a) *Membership inference*: Is the object α an instance of the class C in T ?

b) *Interdependency inference* : Does the interdependency Σ hold in T ?

In Section 2 we present the syntax and the semantics of a general language for class hierarchy scheme specification. Similarly to the work described in [McAllester 86], the language allows to use not only primitive classes, but also expressions denoting classes which are obtained from other classes by means of set operations. It is important to note that such a language allows to specify both that a given relationship (membership or interdependency) holds and that a given relationship *does not* hold in a class hierarchy scheme. As mentioned above, the same approach is taken in [Atzeni 87], for the case of is-a and disjointness relationships. In the same section we show how a class hierarchy scheme can be expressed in first order logic. In particular, it is shown that there is a strong correspondence between class hierarchy schemes and monadic first order theories (i.e. logical theories whose predicates have a single argument).

Using this correspondence, we present in Section 3 a classification of class interdependencies, based on the syntactic form of the logical formulas that can be expressed in monadic theories. Also, we provide a complexity analysis of the inference problem for different types of class hierarchy schemes.

Finally, in Section 4 we consider a subclass of CHSs, namely the Horn CHSs, presenting efficient algorithms for membership and interdependency inference in such a subclass.

2. A Language for Class Hierarchy Specification

In this section we present a general language, called L_{CH}, for specifying class hierarchy schemes. Such a language allows to denote not only primitive classes, but also classes whose extensions are obtained as intersection, union, or complement of the extensions of other classes. Class expressions are then used in the specification of interdependency and membership relationships holding among classes.

The description of the syntax of L_{CH} follows.

```
<class expression> ::= <and class> | <or class> | <class literal>

<and class> ::= <class literal> and <class literal> |
                <class literal> and <and class>

<or class> ::= <class literal> or <class literal> |
               <class literal> or <or class>

<class literal> ::= <class symbol> | non <class symbol> |
                    Everything  |  Nothing

<interdependency assertion> ::= [ not ] <class expression> is
                                        <class expression>

<membership assertion> ::= [ not ] <object symbol> is-instance-of
                                   <class expression>

<assertion>  ::=  <interdependency assertion>  |
                  <membership assertion>
```

In the following, we call positive (negative) assertion any interdependency or membership assertion that does not include (includes) the symbol not. A class literal, or simply a literal, is called negative if it has the form (non <class symbol>), positive otherwise. If C is a positive literal, then (non C) is called its complement. Conversely, C is the complement of the negative literal (non C).

A *class hierarchy scheme* is a finite set of assertions expressed in L_{CH}.

Turning our attention to the semantics of L_{CH}, we define an *interpretation* for a class hierarchy scheme T as a triple <D,O,P>, where D is a finite set of objects, O is a mapping associating to each object symbol of T an element of D, and P is a mapping associating to each class symbol of T a subset of D, with the constraints: P(Everything)=D, and P(Nothing)=∅.

Example 1

Let α be an object symbol, and let A, B, C, D, C and F be class symbols. Then, the following is a class hierarchy scheme expressed in L_{CH}:

T = { F and B is Nothing
A is B or C
not (D and C is non F or E)
α is-instance-of A and non B }

The triple J=<{a,b}, O, P>, where O(α)=a, P(A)={a}, P(B)=∅, P(C)={a,b}, P(D)={b}, P(E)={a}, P(F)={b}, is an interpretation for T.

If I is an interpretation and C is a class expression, then the *extension* of C with respect to I, denoted by EXT(C,I), is determined by the following rules:
- if C is a positive class literal L, then EXT(C,I) = P(L);
- if C is negative class literal (non L), then EXT(C,I) = D - EXT(L,I);
- if C is an And-class (L_1 and ... and L_n), then EXT(C,I) = $\cap_i$ EXT(L_i,I);
- if C is an Or-class (L_1 or ... or L_n), then EXT(C,I) = $\cup_i$ EXT(L_i,I);

An interpretation I *satisfies* the positive interdependency assertion

S is *D*

if and only if the extension of *S* with respect to I is a subset of the extension of *D* with respect to I. Moreover, I satisfies the positive membership assertion

a is-instance-of *D*

if and only if O(a) is an element of the extension of *D* with respect to I. An interpretation I satisfies the negative assertion

not Σ

just in case it does not satisfy the positive assertion Σ.

A *model* for T is an interpretation that satisfies every assertion in T. A class hierarchy specification T is *satisfiable* if there exists at least one model for T, *unsatisfiable* otherwise. *Class hierarchy satisfiability (unsatisfiability)* is the problem of checking if a class hierarchy specifcation is satisfiable (unsatisfiable).

An assertion Σ *logically follows* from T (or, alternatively, T *logically implies* Σ) if every model of T satisfies Σ. In this case we write

$$T \models \Sigma.$$

It can be easily verified that, if Σ is a positive assertion, then T |= Σ if and only if T ∪ {not (Σ)}. Conversely, if Σ has the form not (Σ), then T |= Σ if and only T ∪ {Σ} is unsatisfiable.

If T' is a set of assertions, then T' logically follows from T (written T |= T') if, for each element t' of T', it holds that:

$$T \models t'.$$

Two sets of assertions T and T' are *equivalent* if T |= T' and T' |= T.

Let T be a CHS, and let σ be an assertion; σ is said to be consistent (inconsistent) with T if T ∪ {σ} is satisfiable (unsatisfiable).

A class hierarchy specification T is said to be in *normal form* if the following conditions hold:

1) Every positive interdependency assertion of T is of the form:

S is *D*

where *S* is either a positive literal or an and-class composed by positive literals, and *D* is either a positive literal or an or-class composed by positive literals.

2) Every membership assertion is of the form:

α is-instance-of *D*

where *D* is a class expression in which neither Everything nor Nothing appears.

3) No and-class or or-class in T contains Everything or Nothing; moreover, no assertion contains Nothing in the left hand side or Everyhing in the right hand side.

Every class hierarchy scheme T can be transformed into an equivalent scheme T' which is in normal form. The following algorithm can be used to perform such a transformation.

```
Algorithm NORMAL FORM TRANSFORMATION
Input Class Hierarchy Scheme T
Output Class Hierarchy Scheme in normal form T' equivalent to T
begin
  1. Replace every assertion of the form
                 $A_1$ or ... or $A_n$ is $D$          (with $n > 1$)
     with the following n assertions:
                 $A_i$ is $D$     ($i \in \{1,...,n\}$).
  2. Replace every assertion of the form
                 $S$ is $A_1$ and ... and $A_n$     (with $n > 1$)
     with the following n assertions:
                 $S$ is $A_i$    ($i \in \{1,...,n\}$).
  3. For each assertion of the form
                 $S$ is $D$
     delete any negative literal from $S$ (from $D$), and add its complement to the or-class $D$
     (to the and-class $S$). After all such deletions, replace the empty left hand side (right
     hand side) of any assertion with Everything (Nothing).
  4. Replace every membership assertion of the form:
                 not $\alpha$ is-instance-of D
     with the assertion:
                 $\alpha$ is-instance-of D'
     where D' is determined as follows: if D is a literal, then D' is the corresponding
     complement; if D is an or-class (and-class), then D' is the and-class (or-class)
     constitued by the complements of the literals of D.
  5. For each assertion $\Sigma$, remove Nothing (Everything) from any or-class (and-class)
     appearing in $\Sigma$, and replace any and-class (or-class) that includes Nothing
     (Everything), with Nothing (Everything).
end
```

In the rest of this section we concentrate our attention on the relationship between class hierarchies and first order logic. In particular, we show how a class hierarchy scheme can be expressed in terms of a first order monadic theory, i.e. a first order theory whose predicate symbols are unary. To this end, we describe a mapping MON, which allow to transform any class hierarchy scheme T in normal form into a monadic theory MON(T), such that the set of models of MON(T) is in one-to-one correspondence with the set of models of T.

Let T be a class hierarchy specification T in normal form. MON(T) is defined as follows:

- the constant symbols of MON(T) are in one-to-one correspondence with the object symbols of T; the predicate symbols of MON(T) are in one-to-one correspondence with the class symbols of T; moreover, MON(T) contains two distinguished predicate symbols, namely Everything and Nothing, corresponding to the symbol Everything and Nothing of T; finally, MON(T) includes one variable symbol x;
- the axioms of MON(T) are established by the following rules:
  - MON(T) includes the two axioms: ($\forall x$ Everything(x)) and ($\forall x \neg$Nothing(x));
  - for each positive interdependency assertion

S_1 <u>and</u> ... <u>and</u> S_n <u>is</u> D_1 <u>or</u> ... <u>or</u> D_m

of T, MON(T) includes an axiom of the form

$$\forall x\, (\neg S_1(x) \vee \ldots \vee \neg S_n(x) \vee D_1(x) \vee \ldots \vee D_m(x));$$

- for each negative interdependency assertion

<u>not</u> *S* <u>is</u> *D*

of T, MON(T) includes an axiom of the form

$$\exists x\, (\text{MON\_TRANSF}(S,x) \wedge \neg\text{MON\_TRANSF}(D,x))\,),$$

where MON_TRANSF(E,z) denotes the logical formula obtained from the class expression E by transforming respectively <u>not</u> into $\neg$, <u>and</u> into $\wedge$, <u>or</u> into $\vee$, and every class symbol C of E into the atomic formula C(z).

- for each membership assertion

α <u>is-instance-of</u> *D*

of T, MON(T) includes an axiom of the form MON_TRANSF(D,α).

Example 2

If T is the class hierarchy scheme shown in Example 1, then MON(T) is the monadic theory with constant symbol α, variable symbol x, predicate symbols A, B, C, D, C and F, and the following axioms:

$\forall x$ Everything(x)

$\forall x$ $\neg$Nothing(x)

$\forall x$ ($\neg$F(x) $\vee$ $\neg$B(x) $\vee$ Nothing(x))

$\forall x$ ($\neg$A(x) $\vee$ B(x) $\vee$ C(x))

$\exists x$ (D(x) $\wedge$ C(x) $\wedge$ F(x) $\wedge$ $\neg$E(x))

A(α) $\wedge$ $\neg$B(α)

It is easy to verify that the set of models MON(T) is in one-to-one correspondence with the models of T. In particular, given a model I=<D,O,P> for T, we can construct a model I'=<D',O',P'> for MON(T) as follows:
- D' is the same as D;
- for each constant symbol c of MON(T), O'(c)=O(a), where a is the object symbol of T corresponding to c;
- for each predicate symbol U of MON(T), P'(U)=P(C), where C is the class symbol of T corresponding to U.

An analogous method can be used to construct a model of T from a model for MON(T).

Notice that the axioms of a monadic theory obtained from a class hierarchy scheme by the mapping MON, are "single-argument formulas", i.e. formulas in which all the predicates have the same argument (either a variable or a constant).

It is well known that any monadic theory M can be transformed into a set of clauses (i.e. disjunctions of literals) which is satisfiable if and only if M is satisfiable. It follows that, for any class hierarchy scheme T, one can construct a set of clauses (denoted by CLAUSES(T)), which is satisifiable if and only if T is satisfiable. Notice that CLAUSE(T) may include additional constant symbols (usually called Skolem constant symbols, as opposed to ordinary constant symbols) with respect to MON(T), due to the elimination of the existential quantifiers.

Example 3

We have shown in Example 2 the monadic theory MON(T) corresponding to the class hierarchy specification T of Example 1. From MON(T) one can easily obtain the following set of clauses CLAUSE(T):

$\text{Everything}(x)$
$\neg\text{Nothing}(x)$
$\neg F(x) \vee \neg B(x) \vee \text{Nothing}(x)$
$\neg A(x) \vee B(x) \vee C(x)$
$D(z),\ C(z),\ F(z),\ \neg E(z),\ A(\alpha),\ \neg B(\alpha)$

Notice that "z" is a Skolem constant symbol, whereas α is an ordinary constant symbol.

3. Complexity Analysis of Class Hierarchies

In the first part of this section we present a classification of the basic membership and interdependency relationships expressible in a class hierarchy scheme. We have shown in Section 2 that for any class hierarchy scheme, we can construct a set of clauses that is satisfiable if and only if the original class hierarchy scheme is satisfiable. Starting from this observation, we base our classification on the syntactic form of the possible clauses expressible in monadic first order logic. In Figure 1 we show such a classification in the form of a diagram.

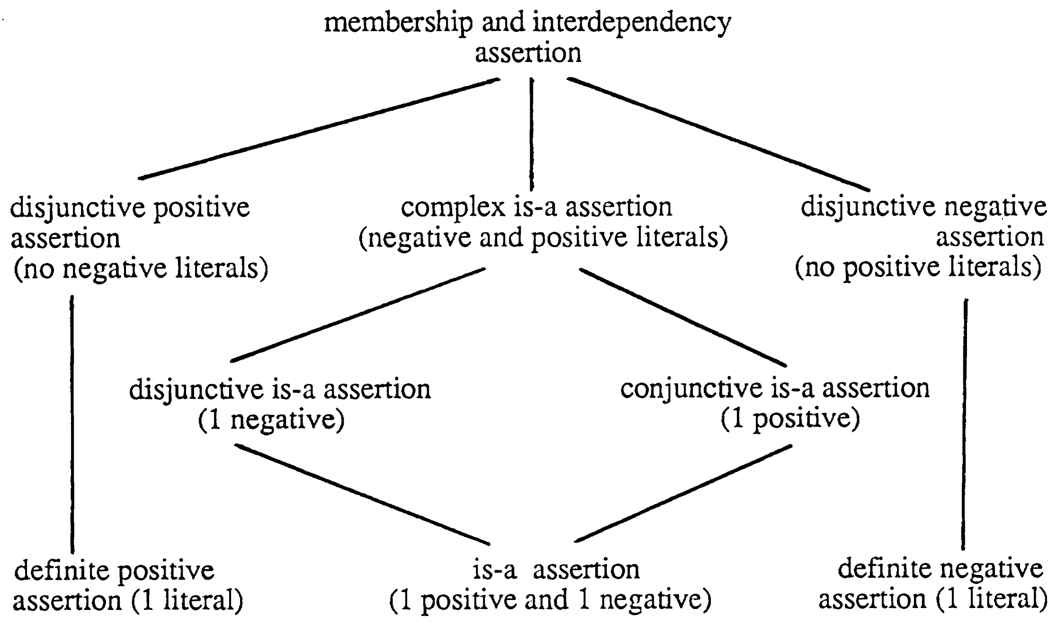

membership and interdependency assertion
disjunctive positive assertion (no negative literals)
complex is-a assertion (negative and positive literals)
disjunctive negative assertion (no positive literals)
disjunctive is-a assertion (1 negative)
conjunctive is-a assertion (1 positive)
definite positive assertion (1 literal)
is-a assertion (1 positive and 1 negative)
definite negative assertion (1 literal)

Figure 1

The formula:

$$\forall x\,(A(x) \vee B(x) \vee C(x))$$

is an example of disjunctive positive assertion; it specifies that every object is an instance of at least one of the classes A,B and C. This corresponds to the interdependency assertion:

Everything <u>is</u> A <u>or</u> B <u>or</u> C

in the language L_{CH}. When applied to a particular object, these assertions specify that an object is an instance of at least one of set of classes. For example, the fact that c is an instance of A or B can be represented by the disjunctive positive assertion:

$$\neg A(c) \vee \neg B(c)$$

which corresponds to the negative membership assertion:

<u>not</u> c <u>is-instance-of</u> A <u>and</u> B

in L_{CH}.

The formula:

$$A(c)$$

is an example of definite positive assertion, which states that c is an instance of A. In L_{CH}, it corresponds to:

c <u>is-instance-of</u> A

Complex is-a assertions allow to state that the extension of an and-class is contained in the extension of an or-class. The formula:

$$\forall x\,(\neg A(x) \vee \neg B(x) \vee C(x) \vee D(x))$$

is an example of this kind of assertions, corresponding to:

A <u>and</u> B <u>is</u> C <u>or</u> D

in L_{CH}. Complex is-a assertions with a single positive literal on the left hand side are called disjunctive is-a assertions, whereas complex is-a assertions with a single positive literal on the right hand side are called conjunctive is-a assertions.

Disjunctive negative assertions allow to state that a set of classes are mutually disjoint. For example, the formula:

$$\forall x\,(\neg A(x) \vee \neg B(x))$$

which corresponds to:

A <u>and</u> B <u>is</u> Nothing

expresses disjointness between A and B. When applied to a particular object, these assertions specify that an object is not an instance of a set of classes simultaneously. For example, the fact that c is not an instance of both A and B can be represented by the assertion:

$$\neg A(c) \vee \neg B(c)$$

which corresponds to the negative membership assertion:

<u>not</u> c <u>is-instance-of</u> A <u>and</u> B

in L_{CH}. Finally, a definite negative assertion specifies either the fact that a class has no instances ($\forall x\, \neg A(x)$), or the fact that an object is not an instance of a class ($\neg A(c)$).

It is easy to verify that in [Atzeni 87], the membership and interdependency relationships taken into account are is-a assertions, disjunctive negative assertions involving at most two literals, and positive and negative definite assertions. In [Lenzerini 87], disjunctive is-a assertions together with binary disjunctive negative assertions are considered. The intersection (union) extended generalizations introduced in [Arisawa 86] are simply sets of conjunctive (disjunctive) is-a assertions. Finally, the context in which inheritance is studied in [McAllester 86], is the one of a language including complex is-a assertions and definite positive membership assertions.

The above taxonomy provides the basis of our investigation on the computational complexity of the inference problem for class hierarchy schemes. In particular, we now consider in turn different subclasses of class hierarchy schemes, characterized by different types of assertions.

We shall start from the observation that the problem of checking a class hierarchy scheme for satisfiability is in general NP-complete. In fact, the following proposition can be easily verified by considering the relationship between propositional satisfiability and class hierarchy satisfiability (see, for example, [Lenzerini 87]).

Proposition 1 Class hierarchy satisfiability is NP-Complete.

The first subclass of CHSs that we consider, includes conjunctive is-a assertions and binary disjunctive positive assertions (i.e. disjunctive positive assertions with two literals). The following result shows that the membership inference problem in such a subclass is at least as complex as propositional satisfiability. It is easy to see that the same problem is hard also for the "dual" subclass, i.e. the subclass consisting of disjunctive is-a assertions and binary disjunctive negative assertions.

Proposition 2 Let T be a CHS with conjunctive is-a assertions and binary disjunctive positive assertions. Let A be a class symbol of T. Then determining if (T |= α is-instance-of A) is NP-Hard.

Proof Let PROP be a propositional formula in conjunctive normal form with variables $v_1,...,v_p$. Define a class hierarchy scheme Φ(PROP) such that:

- for each variable v_i in PROP, there are two class symbols in Φ(PROP), V_i and V_i'; moreover, Φ(PROP) includes a distinguished class symbol R;
- for each variable v_i in PROP, Φ(PROP) includes an axiom of the form

(Everything is V_i or V_i')

- for each clause

$$\neg v_1 \vee ... \vee \neg v_n \vee v_{n+1} \vee ... \vee v_{n+m}$$

in PROP, Φ(PROP) includes an axiom of the form:

(V_1' and ... and V_n' and V_{n+1} and ... and V_{n+m} is R)

Notice that Φ(PROP) includes only conjunctive ISA assertions and binary disjunctive positive assertions.

We claim that PROP is unsatisfiable if and only if (Φ(PROP) |= α is-instance-of R). Assume that (Φ(PROP) |= α is-instance-of R), and suppose that J is a model for PROP. Define an interpretation I=<{α},C,P> for Φ(PROP) such that:

- if $J(v_i)=0$, then $P(V_i')=\emptyset$ and $P(V_i)=\{\alpha\}$;
- if $J(v_i)=1$, then $P(V_i)=\emptyset$ and $P(V_i')=\{\alpha\}$;
- $P(R)=\emptyset$.

It is easy to see that I is a model for Φ(PROP)$\cup${not α is-instance-of R}, which contradicts the hypothesis that (Φ(PROP) |= α is-instance-of R).

On the other hand, assume that Φ(PROP) $\cup$ {not α is-instance-of R} is satisfiable and let I=<{α},C,P> be one of its models. Define an interpretation J for PROP such that:

- if it is not the case that $\alpha \in P(V_i')$, then $J(v_i)=0$;

- if it is not the case that $\alpha \in P(V_i)$, then $J(v_i)=1$;
- if $\alpha \in P(V_i')$ and $\alpha \in P(V_i)$, then $J(v_i)=1$.

It is easy to see that I is a model for PROP.

Q.E.D.

We now consider the class hierarchy schemes in which only complex is-a assertions and definite positive assertions can be expressed, and show that membership inference is NP-hard also for this type of class hierarchy schemes.

Proposition 3 Let T be a CHS with complex is-a assertions and definite positive membership assertions. Let A be a class symbol of T. Then, determining if (T |= α <u>is-instance-of</u> A) is NP-Hard.

Proof Let PROP be a propositional formula in conjunctive normal form with variables $v_1,\dots,v_p$. Define a class hierarchy scheme Φ(PROP) such that:

- for each variable v_i in PROP, there is a class symbol V_i in Φ(PROP); moreover, Φ(PROP) includes two distinguished class symbols, Y and N;
- Φ(PROP) includes the axiom (α <u>is-instance-of</u> Y);
- for each clause

$$\neg v_1 \vee \dots \vee \neg v_n \vee v_{n+1} \vee \dots \vee v_{n+m} \quad \text{(with } n>0 \text{ and } m>0\text{)}$$

in PROP, Φ(PROP) includes an axiom of the form:

$$(V_1' \ \underline{\text{and}} \ \dots \ \underline{\text{and}} \ V_n' \ \underline{\text{is}} \ V_{n+1} \ \underline{\text{or}} \ \dots \ \underline{\text{or}} \ V_{n+m})$$

- for each clause

$$v_1 \vee \dots \vee v_n$$

in PROP, Φ(PROP) includes an axiom of the form:

$$Y \ \underline{\text{is}} \ V_1 \ \underline{\text{or}} \dots \underline{\text{or}} \ V_n$$

- for each clause

$$\neg v_1 \vee \dots \vee \neg v_n$$

in PROP, Φ(PROP) includes an axiom of the form:

$$(V_1 \ \underline{\text{and}} \ \dots \ \underline{\text{and}} \ V_n \ \underline{\text{is}} \ N)$$

Notice that Φ(PROP) includes only complex is-a assertions and definite positive assertions.

We claim that PROP is unsatisfiable if and only if (Φ(PROP) |= α <u>is-instance-of</u> N). Assume that (Φ(PROP) |= α <u>is-instance-of</u> N), and suppose that J is a model for PROP. Define an interpretation I=<{α},C,P> for Φ(PROP) such that P(Y)={α}, P(N)=∅; moreover, if $J(v_i)=1$, then $P(V_i)=\{a\}$, else $P(V_i)=\emptyset$.

It is easy to see that I is a model for Φ(PROP)∪{<u>not</u> α <u>is-instance-of</u> N}, which contradicts the hypothesis that (Φ(PROP) |= α <u>is-instance-of</u> R).

On the other hand, assume that Φ(PROP) ∪ {<u>not</u> α <u>is-instance-of</u> R} is satisfiable and let I=<{α},C,P> be one of its models. Define an interpretation J for PROP such that if $\alpha \in P(V_i)$, then $J(v_i)=1$, else $J(v_i)=0$. It is easy to see that I is a model for PROP.

Q.E.D.

Finally, we analyse the membership inference problem for class hierarchy schemes with disjunctive positive assertions and disjunctive negative assertions.

Proposition 4 Let T be a CHS with disjunctive positive assertions and disjunctive negative assertions. Let A be a class symbol of T. Then, determining if (T $\models$ α is-instance-of A) is NP-Hard.

Proof Let PROP be a propositional formula in conjunctive normal form with variables $v_1,\ldots,v_p$. Define a class hierarchy scheme Φ(PROP) such that:

- for each variable v_i in PROP, there are two class symbols in Φ(PROP), V_i and V_i'; moreover, Φ(PROP) includes a distinguished class symbol R;
- for each variable v_i in PROP, Φ(PROP) includes an axiom of the form

$$(V_i \ \underline{\text{and}}\ V_i' \ \underline{\text{is}}\ \text{Nothing})$$

- for each clause

$$\neg v_1 \vee \ldots \vee \neg v_n \vee v_{n+1} \vee \ldots \vee v_{n+m}$$

in PROP, Φ(PROP) includes an axiom of the form:

$$(\text{Everything}\ \underline{\text{is}}\ V_1'\ \underline{\text{or}}\ \ldots\ \underline{\text{or}}\ V_n'\ \underline{\text{or}}\ V_{n+1}\ \underline{\text{or}}\ \ldots\ \underline{\text{or}}\ V_{n+m}\ \underline{\text{or}}\ R)$$

Notice that Φ(PROP) includes only disjunctive positive assertions and disjunctive negative assertions.

We claim that PROP is unsatisfiable if and only if (Φ(PROP) $\models$ α is-instance-of R). Assume that (Φ(PROP) $\models$ α is-instance-of R), and suppose that J is a model for PROP. Define an interpretation I=<{α},C,P> for Φ(PROP) such that:

- if $J(v_i)=0$, then $P(V_i)=\emptyset$ and $P(V_i')=\{\alpha\}$;
- if $J(v_i)=1$, then $P(V_i')=\emptyset$ and $P(V_i)=\{\alpha\}$;
- $P(R)=\emptyset$.

It is easy to see that I is a model for Φ(PROP)$\cup${not α is-instance-of R}, which contradicts the hypothesis that (Φ(PROP) $\models$ α is-instance-of R).

On the other hand, assume that Φ(PROP) $\cup$ {not α is-instance-of R} is satisfiable and let I=<{α},C,P> be one of its models. Define an interpretation J for PROP such that:

- if $\alpha \in P(V_i)$, then $J(v_i)=1$;
- if $\alpha \in P(V_i')$, then $J(v_i)=0$.

It is easy to see that I is a model for PROP.

Q.E.D.

Keeping the assumption of classifying hierarchies on the basis on the syntactic form of the expressible membership and interdependency assertions, there are basically three classes that have not been shown to be intractable by the above analysis, namely:

- Class hierarchy schemes including conjunctive is-a assertions, disjunctive negative assertions, and definite positive assertions;
- Class hierarchy schemes including disjunctive is-a assertions, disjunctive positive assertions, and definite negative assertions;

- Class hierarchy schemes including is-a assertions, and binary disjunctive assertions (both positive and negative).

The next section is devoted to the first of these classes.

4. Horn Class Hierarchy Schemes

In this section we describe a method for performing inferences in a subclass of CHSs, namely the Horn CHSs.

A *Horn CHS* (HCHS) is a class hierarchy scheme such that its normal form satisfies the following conditions:

1. Every positive assertion has a class literal on the right hand side;
2. For each negative interdependency assertion

   $$\underline{\text{not}}\ S\ \underline{\text{is}}\ D$$

   if S is an or-class, then it contains at most one positive literal; if D is an and-class, then it contains at most one negative literal;
3. For each memebership assertion

   $$\alpha\ \ \underline{\text{is-instance-of}}\ D$$

   if D is an or-class, then it contains at most one positive literal.

It is easy to see that HCHSs are precisely those class hierarchy schemes whose corresponding sets of clauses are Horn sets.

Our method for performing inferences in HCHSs requires a HCHS to be represented by means of a graph.

Let T be a HCHS. The associated graph G^T is a directed graph <V,R>, where:

- V is the set of nodes, which is partitioned into sets, the P-nodes and the A-nodes. There is one P-node for each class symbol of T, and there is one A-node for each clause of CLAUSE(T) containing two or more negative literals. Moreover, the set of P-nodes includes two distinguished nodes, namely E and N. In the following we denote a P-node of G^T by the name of the associated predicate.
- R is the set of arcs, labeled with the constant and variable symbols of CLAUSE(T). For each clause of the form:

  $$Q(w)$$

  in CLAUSE(T), where Q is different from Everything, there is an arc <E,Q> labeled with w in R. For each clause of the form:

  $$\neg Q(w)$$

  in CLAUSE(T), where Q is different from Nothing, there is an arc <Q,N> labeled with w in R. For each clause of the form:

  $$\neg Q(w) \vee P(w)$$

  in CLAUSE(T), there is an arc <Q,P> labeled with w in R. For each clause of the form:

  $$\neg Q_1(w) \vee \ldots \vee \neg Q_n(w) \vee P(w)$$

  in CLAUSE(T) associated with the A-node A, there are the arcs $<Q_1,A>,\ldots,<Q_n,A>,<A,P>$ labeled with w in R. For each clause of the form:

  $$\neg Q_1(w) \vee \ldots \vee \neg Q_n(w)$$

  in CLAUSE(T) associated with the A-node A, there are the arcs $<Q_1,A>,\ldots,<Q_n,A>,<A,N>$ labeled with w in R.

In the following, we call a *H-graph* any graph associated with a HCHS. If T is a HCHS, and G^T is the associated graph, we say that G^T is also the graph associated with CLAUSE(T).

Example 4

The following is a HCHS:

T = { F and B is Nothing
M is B
not (D and C is non F or L)
C and L is G
α is-instance-of M and non B
β is-instance-of non G
β is-instance-of non M or L }

whose corresponding set of clauses is:

$$\begin{aligned}\text{CLAUSE(T)} = \{ & \text{Everything}(x), \neg\text{Nothing}(x), \neg F(x) \vee \neg B(x) \vee \text{Nothing}(x),\\ & D(z),\ C(z),\ F(z),\ \neg L(z),\ \neg C(x) \vee \neg L(x) \vee G(x),\\ & M(\alpha),\ \neg B(\alpha),\ \neg G(\beta), \neg M(\beta) \vee L(\beta)\ \}\end{aligned}$$

The H-graph G^T associated with T is shown in Figure 2. The A-nodes A1 and A2 are associated with the third and fifth clause above, respectively.

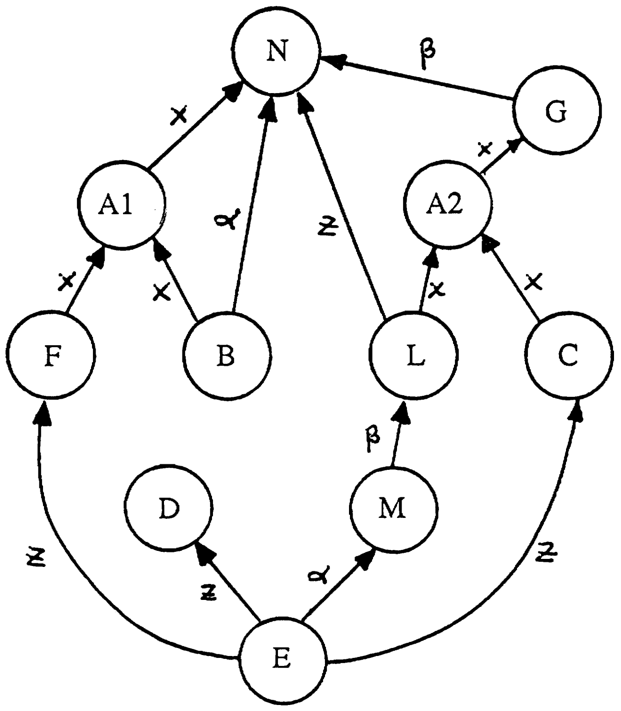


Figure 2

Let G be a H-graph, and let $Q_1,...,Q_n,P$ be P-nodes of G. A subgraph G' of G is a *c-hyperpath* of G from $\{Q_1,...,Q_n\}$ to P if one of the following conditions holds:

1. $P \in \{Q_1,...,Q_n\}$, or
2. there is an A-node A and arcs $<P_1,A>,...,<P_m,A>,<A,P>$ labeled with c or x in G' such that:
   - $P_1,...,P_m$ are all the predecessors of A in G;
   - for each $i \in \{1,...,m\}$, there is a c-hyperpath of G from $\{Q_1,...,Q_n\}$ to P_i in G';
3. there is a P-node P_1 and an arc $<P_1,P>$ labeled with c or x in G', and there is a c-hyperpath of G from $\{Q_1,...,Q_n\}$ to P_1 in G'.

Proposition 5 A HCHS T is unsatisfiable if and only if there is a c-hyperpath from $\{E\}$ to N in G^T, for some constant or variable symbol c of CLAUSE(T).

Proof (sketch) The proof is based on the fact that unit resolution is a sound and complete inference rule for Horn clauses. It is shown that there is c-hyperpath from $\{E\}$ to N in G^T, for some constant or variable symbol c of CLAUSE(T), if and only if there is a unit refutation of CLAUSE(T).

Q.E.D.

We now present an algorithm for checking for the existence of a w-hyperpath in a H-graph. The algorithm makes use of a boolean value mark(P) associated with each node P of the graph.

Algorithm HYPERPATH(G,w, Q)
Input H-graph G, label w, node Q of G, boolean value mark(P) for each node P of G
Output boolean value mark(N), which is true if only if there is a w-hyperpath from $\{Q,Q_1,...,Q_m\}$ to N, where $Q_1,...,Q_m$ are the nodes of G such that the initial value of mark(Q_i) is true

```
begin
    if not mark(Q)
    then if Q is a P-node
            then set mark(Q) to true;
                 if Q=N then return;
                 for each outgoing arc <Q,M> of Q labeled either with w or x
                 do HYPERPATH(G,w, M) enddo
            else if for each predecessor M of Q, mark(M)=true
                 then set mark(Q) to true;
                      HYPERPATH(G,w, P)
                 endif
            endif
    endif
end
```

It is easy to see that there is a c-hyperpath from $\{Q_1,...,Q_m\}$ to N in G if and only if after the execution of:

```
for each node P of G do set mark(P) to false enddo;
for each $i \in \{1,...,m\}$ do HYPERPATH($G^T$,c,$Q_i$) enddo
```

the value of mark(N) is true. Therefore, the algorithm HYPERPATH can be used for checking a HCHS T for unsatisfiability as follows:

```
Algorithm UNSATISFIABLE(T)
Input HCHS T
Output true, if T is satisfiable, false otherwise
begin
  for each constant symbol c (either ordinary or Skolem) of CLAUSE(T)
  do for each node P of $G^T$ do set mark(P) to false enddo;
      HYPERPATH($G^T$,c,E);
      if mark(N)=true then return(true)
  enddo;
  return(false)
end
```

If *n* is the number of constant symbols of CLAUSE(T), which corresponds to the number of objects and negative interdependency assertions of T, and *m* is the size of CLAUSE(T), then the above method can be implemented in *O(nm)* time. Notice that a method similar to the one used in the algorithm HYPERPATH has been presented in [Dowling 84] for the simpler case of propositional satisfiability.

When T is built incrementally, the efficiency of the method can be improved. Suppose we want to construct a HCHS in such a way that new assertions are accepted if and only if the resulting class hierarchy scheme is satisfiable.

Let T be a satisfiable HCHS, and let mark(P)=false for each node P of G^T. . We want to add an assertion A to T, obtaining a new HCHS which logically implies A and is satisfiable.Three cases have to be taken into account, depending on the type of assertion to be added to T.

1. If we want to add an interdependency assertion Σ to T, we perform the following operations: first, we check if $S=T\cup\{\Sigma\}$ is a HCHS; if so, we execute:

   HYPERPATH(G^S,x,E)

   It can be easily verified that S is satisfiable if and only if, after such an execution, mark(N)=false. In this case, S is the resulting HCHS.
2. Analogously, when we add a membership assertion σ of the form:

   α <u>is-instance-of</u> *D*

   to T, we check if $S=T\cup\{\sigma\}$ is a HCHS and, if so, we execute:

   HYPERPATH(G^S,α,E)

   S is satisfiable if and only if, after such an execution, the value of mark(N) is false.
3. Finally, when we add a negative interdependency assertion Σ' of the form:

   <u>not</u> Σ

   to T, we first check if $S=T\cup\{\Sigma'\}$ is a HCHS and, if so, we execute:

   HYPERPATH(G^S,z,E)

   where z is the Skolem constant symbol of CLAUSE(S) which is associated with Σ'. The resulting CHS S is satisfiable if and only if, after such an execution, mark(N)=false.

where z is the Skolem constant symbol of CLAUSE(S) which is associated with Σ'. The resulting CHS S is satisfiable if and only if, after such an execution, mark(N)=false.

These considerations show that the cost of adding an assertion to T is *O(m)*, where *m* is the size of the resulting HCHS S.

Using the above framework, we can efficiently solve the inference problem in HCHSs. In the following, T denotes a HCHS.

Suppose we want to check if a positive interdependency assertion Σ logically follows from T. Let $\{\sigma_1,...,\sigma_n\}$ be the set of assertions obtained by transforming Σ into normal form. Since $(T \models \Sigma)$ if and only if, for each i, $(T \models \sigma_i)$, we can reduce the problem of checking if $(T \models \Sigma)$ to the problem of checking each $(T \cup \{$not $\sigma_i\})$ for unsatisfiability. Notice that each $(T \cup \{$not $\sigma_i\})$ is a HCHS and, therefore, we can proceed as in case 3 above.

With regard to the membership assertions, notice, first of all, that a negative membership assertion can be transfomed into an equivalent positive one. Hence, we deal only with positive assertions in the following. In particular,we distinguish between two cases. If the membership assertion σ has the form:

α is-instance-of L_1 and...and L_p

then $(T \models \sigma)$ if and only if, for each i, $(T \models \alpha$ is-instance-of $L_i)$, i.e. if and only if for each i $(T \cup \{\alpha$ is-instance-of $L_i'\})$ is unsatisfiable, where L_i' is the complement of L_i. Notice that $(T \cup \{\alpha$ is-instance-of $L_i'\})$ is a HCHS (see case 2 above). On the other hand, if σ has the form:

α is-instance-of L_1 or...or L_p

then $(T \models \sigma)$ if and only if $S=T\cup\{\alpha$ is-instance-of L_1' and...and $L_p'\}$ is unsatisfiable. Since S is a HCHS, the problem of checking if $(T \models \sigma)$ can be solved by executing:

HYPERPATH(G^S,α,E)

and checking if, after such an execution, the value of mark(N) is true.

Finally, with regard to negative interdependency assertions, notice that (T |= not *S* is *D*) if and ony if there exists a constant symbol α (either ordinary or Skolem constant symbol) of CLAUSE(T) such that (T |= α is-instance-of *S*) and (T |= not α is-instance-of *D*). Therefore, we can reduced our original problem to the one of checking if two membership assertions are logically implied by T, for each constant symbol of CLAUSE(T).

An interesting application of membership inference allows us to avoid adding membership assertions which are logically implied by the original class hierarchy scheme. In fact, when we add a membership assertion σ to T, we can not only check if $(T\cup\{\sigma\})$ is satisfiable, as specified by case 2 above, but also check if $(T \models \sigma)$; σ will be added to T if and only if it is consistent with T (i.e. $T \cup \{\sigma\}$ is satisfiable) and it is not logically implied by T (i.e. $T \models \sigma$ does not hold).

In order to illustrate how this can be accomplished, let us consider the case in which we want to add a membership assertion σ of the form:

α is-instance-of Q

to T, where Q is a class literal. The following procedure specifies a method for efficiently deal with this case:

```
for each node P of $G^T$ do set mark(P) to false enddo;
HYPERPATH($G^T$,$\alpha$,E);
if mark(Q)=true
then return("$\sigma$ is logically implied by T")
else let S be T$\cup${$\sigma$}
     in HYPERPATH($G^S$,$\alpha$,P)
       if mark(N)=true
       then return("$\sigma$ is inconsistent with T")
       else return("S is the resulting CHS")
       endif
endif
```

It can be shown that with the above procedure σ is added to T if and only if it is consistent with T and it is not logically implied by T. Notice that this method is particularly important for CHSs with a large number of membership assertions, as in database applications.

5. Conclusions

Many recent works in object oriented databases and languages deal with the problem of performing inference on membership and interdependency relationships. In order to provide a common framework for these work, we have presented a taxonomy of such relationships, based on a correspondence between class hierarchy schemes and first order monadic theories. Also, we have studied the computational complexity of the inference problem for class hierarchy schemes. Finally, we have considered a subclass of CHSs, namely the Horn class hierarchy schemes, and we have presented efficient methods for performing inference in such a subclass.

In the future, we aim at extending our method to more expressive class hierarchy schemes. For example, one may wander if there is any method to deal with complex is-a assertions in HCHSs, without falling into the intractability cliff. We believe that one possibility for meeting this requirement is to treat complex is-a assertions differently from the other assertions, namely as integrity constraints. Let T be a CHS in normal form constituted by two disjoint parts: a Horn CHS T_H, and a set T_I of (non-binary) complex is-a assertions. Say that T is *concrete* if, for each object α of T, and for each complex is-a assertion of the form:

$$S \ \underline{\text{is}} \ D_1 \ \underline{\text{or}} \ldots \underline{\text{or}} \ D_p \quad (\text{with } p>1)$$

in T_I, ($T_H \models \alpha$ is-instance-of S) implies that there is at least one D_i such that ($T_H \models \alpha$ is-instance-of D_i). It turns out that in concrete CHSs, membership inference can be efficiently performed. In fact, it can be shown that if T is a concrete CHS, and σ is a membership assertion of the form (α is-instance-of C), where C is a class symbol, then ($T \models \sigma$) if and only if ($T_H \models \sigma$). Obviously, a sound and efficient method is needed for incrementally building a CHS in such a way that the resulting scheme be concrete. We shall deal with this and other aspects in future works.

References

[Albano 85] Albano A., Cardelli L., and Orsini R., "Galileo: A Strongly Typed Interactive Conceptual Language", *ACM Transactions on Database Systems*, Vol.10, No.2, March 1985.

[Arisawa 86] Arisawa H., and Miura T., "On the Properties of Extended Inclusion Dependencies", *Proc. of the TwelvthVLDB Conference*, Kyoto, 1986.

[Atzeni 86] Atzeni P., and Stott Parker D., "Formal Properties of Net-based Knowledge Representation Schemes", *Proc. of the 2nd IEEE International Conference on Data Engineering*, Los Angeles, Ca, February 1986.

[Atzeni 87] Atzeni P., and Stott Parker D., "Set Containment Inference", *Proc. of the International Conference on Database Theory, Lectures Notes in Computer Science*, N.243, Springer-Verlag New York Inc., 1987.

[Buneman 86] Buneman P., and Atkinson M., "Inheritance and Persistency in Database Programming Languages", *Proc. of ACM SIGMOD*, Washington D.C., 1986.

[Dowling 84] Dowling W.P., and Gallier, J.H., "Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae", *Journal of Logic Programming*, Vol.1, No.3, 1984.

[Israel 84] Israel D.J., and Brachman R.J., "Some Remarks on the Semantics of Representation Languages", in Broadie M.L. et. al. (eds.), *On Conceptual Modelling*, Springer-Verlag New York Inc., 1984.

[Lenzerini 87] Lenzerini M., "Covering and Disjointness Constraints in Type Networks", *Proc. of the 3rd IEEE International Conference on Data Engineering*, Los Angeles, Ca, February 1987.

[McAllester 86] McAllester D., and Zabih R., "Boolean Classes", *Proc. of ACM OOPSLA Conference*, 1986.

Static and Dynamic Type-Checking

David C.J. Matthews

Abstract

The purpose of a type checker is to prevent an incorrect operation from being performed. A static type checker does this by stopping the compiler from generating a program with type errors, a dynamic type checker halts the program as it is about to make a type error. It is clearly useless to have a dynamic type checking system for a program which is to be produced, distributed and used by anyone other than the original authors since any type errors that occur would be meaningless to the user of the program.

On the other hand, where a user is guiding a program through some data, a dynamic type-checking system is reasonable. Examples are browsing through a database or structure-editing. Here type-errors have meaning to the user.

The ideal language would be basically statically type-checked but would allow dynamic type-checking when necessary. While this is possible with certain type systems there are others for which it is difficult. The implementation of dynamic type checking in various type systems is considered.

1 Type-Checking

Type checking is an effective way of reducing programming errors. Its function is to identify those values to which an operation can be "sensibly" applied. The definition of "sensible" depends on the type system but usually operations like adding together two functions are not regarded as sensible while adding two numbers is.

1.1 Static and Dynamic Checking

One way of doing the type checking is to tag each value with a few bits which describe its type. Each operation checks the tag bits and gives some sort of failure if the values have the wrong type. *Dynamic* type checking will prevent some of the more obscure errors but has the disadvantage that the failure is only generated when the program is run.

A better method involves placing some restrictions on the programs that can be written so that the compiler can decide *statically* whether a program could possibly generate type failures when it is run. If the program can be shown to be type-correct there is no need for the tags and we know before the program is ever run that type errors will not occur.

1.2 Binding

Related to this is the question of binding. Declarations bind names to values and so have types. When an identifier is looked up the value with its type is returned. If there are several identifiers with the same name there must be rules for deciding which one is meant in a particular context.

One of the restrictions for static type checking to be possible is that the compiler must know the type of all the identifiers in the program. This requires *static binding* to identifiers, that is the identifiers are matched up with their declarations when the program is compiled and does not depend on the execution paths.

It is possible to have *dynamic binding* where an identifier is looked up when the program is run, but static type checking is possible only if all the identifiers with the same name have the same type. General dynamic binding requires dynamic type checking.

2 Static Type-Checking

Testing a program to try and find errors is difficult, and can never guarantee correctness. The ideal programming language would be one which imposed no restrictions but where the compiler could decide whether the program was correct. Unfortunately that is impossible and so we must accept some restrictions and even then we only have a limited form of correctness. However type-correctness is sufficiently useful that paying the penalty in terms of accepting some restrictions is reasonable. Recent developments in the design of type systems, particularly polymorphism[3], have extended the range of static type checking into areas where traditionally dynamic type checking was thought necessary.

2.1 Type Equivalence

At the lowest level a type must describe the structure of its values in terms of type constructors such as records and unions and the primitive types of the language, in order that the primitive operations can be type checked. In one form of type checking, if a type can be given a name it is treated as an abbreviation for the structure and two values are treated as the same type if they describe the same structure. This is *structural type equivalence* used for example in Algol 68[9].

An alternative is to define that two values have the same type only if they have the same type name. If the type names are different the types are incompatible even if the structures of the types are the same. *Name equivalence* does not mean that the structure of the type is not visible, only that it is not used for type equivalence. *Abstract types* are a variation of name equivalence where the association between a type name, the *abstraction* and its structure, the *implementation* is only visible within the abstract type definition[4]. Outside that it has no structure and name equivalence is used. When types can be returned as a result of functions name equivalence or a variation of it is needed to ensure that type checking is decidable[2][6].

3 Dynamic Binding and Type-Checking

Static type checking is useful when a program is being produced which is to be executed later. If the program is to be executed immediately, and particularly if it just consists of a single command, there is really very little difference between static and dynamic type checking. Command line interpreters, or "shells" are an example. The command

```
    edit afile
```

typed to a command interpreter would probably involve a search for the files `edit` and `afile` and checks that `edit` was an executable file and `afile` was a text file. The search and type checking for `afile` might well be done from within the `edit` program. There is no advantage in treating type checking separately from execution.

However, if several commands are put together in a command script it starts to look more like a writing a program. Because the individual commands are dynamically bound and type checked it is not possible to statically type check the completed script even though it resembles a programming language procedure. Apart from command interpreters the Mentor programming environment[7] is another example where structure editing commands in the Mentol language can be put together into procedures.

Apart from the fact that there is no advantage in statically checking a command which is to be executed immediately, there are other reasons why dynamic type checking is used. Programs often create file names, for instance by appending standard suffixes onto a name to make a set of related file names. Since the files are dynamically bound they must be dynamically type checked.

Dynamic binding may not only be by name but by other mechanisms as well. In a structure editor a user may select an item by pointing to it with a mouse. Different parts of the structure will have different types so that changes to the structure are constrained, but the function that returns a selected value must be able to return a value of any type. Dynamic type checking must be used if this value is to be copied somewhere else in the structure.

4 Combining the Two

Static type checking is needed for programs which are to be executed in the future, but dynamic type checking is needed for interactive operations. If we have a system where both of these activities can occur we really need both mechanisms.

The obvious way to do this is to take a static type system and add some additional syntax and a new type *dynamic*.

dynamic x

constructs a value of type *dynamic* by packaging up the value with information describing its type. The inverse operation

coerce d **to** t

checks that **d** is a dynamic value with the type information appropriate to the type **t** and returns the original or raises an exception. The syntax is taken from Amber[1]. Dynamic binding can be done by returning values of *dynamic* type and then coercing them to the appropriate type.

A dynamic value contains a value and a representation of the type. The type representation must contain enough information for the **coerce** operation to do the same kind of checking at run-time as the compiler would do at compile-time. We do not want the dynamic type mechanism to subvert the static type system. This may be more complicated than it appears. The rules for static type equivalence which are applied at compile-time may not be reproducible at run-time. To see how the static type system influences the dynamic typing some static type systems will be examined, both from languages which have dynamic types and those that do not.

4.1 Structural Equivalence

The simplest type systems for this purpose are those such as Amber that have a fixed number of primitive types and use structural equality between types. Each primitive type can be assigned a unique identifier and data structures used to describe the structured types. Because names for types are just synonyms for the structure we can always use a representation of the structure for the type representation. Although the name may be declared locally the structure representation is valid anywhere so dynamic type checking is safe. Type inheritance in Amber does not have any serious effect on this.

4.2 Polymorphism

If the language allows polymorphic operations, as in ML, dynamic type checking has to be arranged more carefully. Consider the following two functions.

fun *get_dynamic d* = **coerce** *d* **to** α;
fun *make_dynamic x* = **dynamic** *x;*

get_dynamic takes a dynamic value and coerces it to the type variable α. If the dynamic type matching rules follow the static type rules these should match for any type since unification of a type variable with any type would succeed. Clearly this would allow the type system to be broken because we could make a dynamic value out of, say, an integer value, pass it into *get_dynamic* and treat the result as a string.

make_dynamic will take any value and make a dynamic value from it. This again could break the type system. A solution to both of these problems is simply to forbid polymorphic types in **coerce** or **dynamic** operations.

4.3 Abstract Types

If the static type system allows the user to create abstract types we have to produce a unique identifier for each abstract type and use those in type representations.

References

[1] Cardelli L. *Amber* AT&T Bell Labs Technical Report 1984.

[2] Demers A. and Donahue J. *Revised Report on Russell* TR79-389, Department of Computer Science, Cornell University, 1979.

[3] Gordon M. et al. *A Metalanguage for Interactive Proof in LCF* Fifth Annual Symposium on Principles of Programming Languages, Tucson 1978.

[4] Liskov B. et al. *CLU Reference Manual* Springer-Verlag, Berlin 1981.

[5] MacQueen D.B. *Modules for Standard ML* AT&T Bell Labs Technical Report 1985.

[6] Matthews D.C.J. *Poly Manual* SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[7] Mélèse B. et al. *The Mentor-V5 Documentation* Technical Report 43, INRIA 1985.

[8] Milner R. *A Proposal for Standard ML* in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming", Austin, Texas 1984.

[9] van Wijngaarden A. et al. *Revised Report on the Algorithmic Language Algol68.* Springer-Verlag, Berlin 1976.

4.5 Types as Values

In languages such as Russell and Poly types can be treated as first class values. A type and operations associated with it are packaged together and treated as a run-time value. To ensure decidability name equivalence has to be used. An expression such as

> **let** *atype* == **if** ... **then** *type1* **else** *type2;*

declares *atype* to be a type which is not the same as either *type1* or *type2,* since it is in general not decidable which is actually being returned.

This causes problems if we try to use the dynamic type scheme suggested above for ML. Suppose *type1* and *type2* are different implementations of trees as in the ML example. They both have *move* functions which return dynamically typed values. If we make a tree of type *type1* we expect the dynamically typed values to be coercible to *type1* but not to *type2* or to *atype*. Similarly values from *atype* should not be compatible with either *type1* or *type2*. Unfortunately if the type representation is put into the dynamic values inside the abstract type declaration the dynamic values returned from *atype* trees will be either *type1* or *type2* depending on the actual type returned by the **if**. There seems to be no way to avoid the dynamic type checking behaving differently to the static type checking.

5 Conclusions

Certain applications require dynamic type checking in an otherwise statically typed language. For some type systems this is relatively easy to arrange, but others require considerable thought if the security of the static type system is not to be undermined.

which would allow the type system to be broken, so clearly this cannot be allowed as it stands. In any case it is difficult to see how it would achieve what is wanted, which is for leaves of *int* trees to be returned as dynamic values coercible to values of type *int*.

However if we treat the parameterised type more like a function so that the parameterised type is always used in its parameterised form we can safely allow dynamically typed values to be created and probably get the required behaviour. In Standard ML this could be done using a parameterised module, called in ML a *functor*[5].

```
functor Tree(Elem: sig type t end) =
  struct
  datatype tree = Leaf of Elem.t | Tree of tree * tree;
  fun move = ... (* As before *)
  end
```

The type *tree* is only available when the functor *Tree* has been applied to a module, in ML a *structure*, containing a type. Other rules in ML ensure that this is a monotype.

```
structure IntTree = Tree(struct type t = int end);
```

This creates a tree whose leaves are integers. In order to get the effect we want the type representation for *int* must have been passed into the functor so that the dynamically typed values returned from leaves are recognisably integers. The tree type *IntTree.tree* itself is a new atomic type so the representation for the type can be created dynamically when the functor is applied.

In CLU, which has parameterised clusters and dynamic types, this is rather more difficult. A cluster parameterised by the same parameter values denotes the same type wherever it appears in a program. Since the type may appear in different segments the CLU linker must examine all the types, construct unique identifiers for each different type, and then pass the identifier to be used for the result of each parameterised type into the type as an additional argument.

Intensional Concepts in a Database Programming Language

David Beech

Hewlett-Packard Laboratories
1501, Page Mill Road
Palo Alto
CA 94304

One way in which database systems need to be given increased semantic power is in the use of intensional concepts and more general inferential ability. A framework for achieving this will be described in terms of extensions to an object data model.

Predicator and Generator types are introduced to define collections of objects, or relations between objects, by formulae which test an object or n-tuple of objects for membership in the Predicator case, or generate them in the Generator case. Operations are defined on these types, such as Assert which provides the hook for dealing with view update by specialized actions where necessary.

Examples are given in a higher-level language syntax to supplement the description of the underlying primitives.

A dynamic value created from the abstract type will contain different type information to a dynamic value created from the representation. This may be difficult if the dynamic value is created inside the abstract type package since there the distinction between the values of the abstract type and the implementation type is blurred.

4.4 Parameterised Types

Parameterised abstract types create another problem. If we have a type which can be parameterised by other types or values we need to ensure that any dynamic values created from values of the result type or the argument types have the correct type representation. If the parameterisation simply involves macro-expansion this is relatively easy but if it is done at run-time the type representations will have to be passed as run-time values. The type identifier for the resultant type may have to be created dynamically when the parameterisation is done.

For example, we may define a type *tree* which is a binary tree parameterised by the type of the leaves. Inside the type definition we write an operation to walk over the tree in response to commands from the user and return either a leaf or a piece of tree as a dynamic type.

```
abstype α tree = Leaf of α | Tree of α tree * α tree
with
  fun move "value" (Leaf l) = dynamic l (* Return the leaf *)
   |  move "value" (Tree t) = dynamic t (* Return the tree *)
   |  move "left" (Tree (l, _)) =
            move (nextcommand()) l (* Move left *)
   |  move "right" (Tree (_, r)) =
            move (nextcommand()) r (* Move right *)
   |  move _ _ = raise bad_command (* Anything else *)
end
```

In this example Standard ML[8] has been used with the addition of the operation to create dynamic types. The dynamic operations have been applied to polytypes

Contents

n arguments (usually called relations in logic when n > 1, although we shall abstain from using the word "relation" to avoid confusion with database parlance).

When we say that a concept has been given an *extensional* definition, we shall mean that its exemplification is solely determined by a succession of explicit assertions that individual objects (or tuples of objects) are or are not examples of the concept. We will show in italics a first approximation to how this might be expressed:

Create concept Person (Object o);
Assert Person(o1), Person(o3), Person(o4);
Retract Person(o1);
Assert Person(o9);

Create concept FatherOf (Person f, Person c);
Assert FatherOf(o3, o4);

An *intensional* definition of a concept employs some formula or algorithm or rule which enables its exemplification to be determined from other information without requiring direct assertions about this concept. For example:

Create concept Father (Person p) as
  Exists Person c such that FatherOf(p, c);

Create concept GrandfatherOf (Person gf, Person gc) as
  Exists Person p such that FatherOf(gf, p) and ParentOf(p, gc);

(Note that the extensional/intensional distinction refers to a particular definition of a concept, not to the concept itself. There may be many ways of defining an interrelated set of concepts with different choices as to what is to be extensional or intensional. In a way, the choice of something as extensionally defined is a confession of arbitrariness or disinterest or ignorance—we may have to be told explicitly who someone's parents are because we were not present at the birth, or lack other sound evidence.)

Of course, an intensional definition may use other extensionally defined concepts. It is also often the case in a world of incomplete information that an intensional definition may be indecisive, and yet may be supplemented by direct extensional information about the same concept. For example:

Assert GrandfatherOf(o1, o9);

```
Create Employee instance Smith;

Add type Pilot to Smith;
```

Types are themselves modelled as objects, and like other objects may be alterable and versionable.

2.2 Actions

Actions are also objects, defined to take arguments of certain types and return a result (possibly many-valued) of a certain type. Actions are *applied* to their arguments—this is not itself an action, but a meta-action of the model. Actions may produce truth-values or results of any other type, and may be defined by explicit update or by formulae:

```
Create action name(Person) ⟶ String;
Assert name(Smith) = 'Z.Y. Smith';
Assert name(Mendoza) = 'Carlos Mendoza';

Create action ManagerName ( Employee e ) ⟶ String
    as Select name(m)
       for each Employee m
       where m = manager(dept(e));
```

Formulae in the model provide recursive computability. Actions may also be defined by algorithms with side-effects, and they may be *foreign actions* written in programming languages provided that their argument and result interfaces are consistent with this model.

2.3 Extensional Collections

We treat extensional collections of objects differently from pure sets, and call them *combinations*. A combination is itself an object, and obeys the usual rules for object identity. Objects must be inserted and removed explicitly, and it is thus possible for two combinations to have the same members without being identical. This corresponds to the semantic situation in a time-varying world where the objects being modelled are distinct, although at a given level of abstraction and at a given time they cannot be distinguished by their components.

```
Create Combination C1, C2;
Insert Hecht, Mendoza into C1;
Remove Hecht from C1;
```

```
    Create type Person;
    Create Person instance Smith;
```

In the latter case, a corresponding predicate IsPerson may be maintained, but there is more to it than this. A type, in our model at least, can be instantiated, whereas setting a predicate True for given arguments does not create a new object—the semantic power of an action to remember such information is primitive, rather than being modelled in terms of other objects which have some other primitive powers of memory. But there is another property of types which is very widespread in programming language and data models. This is their use for type checking of action parameters and results. The type specified for a parameter or result serves as a constraint, yet is clearly a very partial mechanism, governed usually by the desire for simplicity and as much static checking as possible. Type expressions, and more general constraint expressions, are the subject of important research, but have not yet found their way into general practice. Perhaps we shall see type systems evolve to become richer, or perhaps we shall see the existing limited systems survive as a well-judged engineering trade-off between simplicity and power, to be supplemented by more general constraint systems (not limited to argument and result checking) as these become practicable.

Now we are ready to pose the question whether *intensional* concepts of one variable should also be expressible, not only as actions, but also as types? This would certainly be possible, but would conflict with the current tendency for types to be instantiable. It may be argued that system types like Integer are already not explicitly instantiable, and that in systems which support unbounded integer computation, it may be philosophically uncomfortable to some (the author included) to postulate an infinite set of instances already instantiated. However, this suggests a solution, that new instances of such types may be implicitly created as required, and thereafter remain in existence just like explicitly created instances of a type. This leads to consideration of the second conflict with current usage of types—that the instantiation of types becomes highly dynamic and difficult to check. Worse than this, determination of the types of objects would have to be defined very precisely as to when and in what order it was carried out, in case any of the actions involved had side-effects (which are hard to exclude in database systems which are largely designed to achieve side-effects). Then much optimization might have to be excluded in case it led to different results, not merely of the computation, but of the type checking itself.

So for the present, it looks advisable to avoid intensional types. This also helps with the requirement to evolve from the present, without requiring a complete change to a new language. Possible approaches such as the embedding of a data language in a programming language (*à la* SQL), and the sharing of type definitions between languages, are facilitated by adopting a conservative treatment of types.

So we are left with a uniform treatment of the four cases considered, in which extensional or intensional concepts, of one or more variables, may all be modelled by actions. Supplementing this, there is the alternative, in the case of extensional concepts of one variable, of being able to define them as types and to instantiate them explicitly and to have conventional type checking carried out.

```
Create predicator Father(Person p) as
  exists (Select
          each Person c
          where FatherOf(p,c));

Create predicator GrandfatherOf(Person gc)  ⟶ Person gf  as
  Select distinct gf
  for each Person p
  where FatherOf(gf,p) and ParentOf(p,gc);
```

Beneath this slightly higher-level language, the underlying object model makes it possible to iterate over instances of a type, or (tuples of) objects satisfying a predicator. Query evaluation strategies must choose between the alternatives in combining the **each** or **for each** clause with the **where** clause. To evaluate `Select GrandfatherOf(Smith)`, it would be possible, for example, to iterate over the Person type and simply apply the **FatherOf** and `ParentOf` predicators to each p; or to iterate over the `ParentOf` predicator and within that to apply the **FatherOf** predicator. The Iterate primitive in the model passes an Action to be applied to each satisfying tuple, and also an Integer which places an upper bound on the number of iterations if non-negative. The result is a List of the results of the Action from each iteration.

It is also possible to define `GrandfatherOf` as a hybrid predicator:

```
Create predicator GrandfatherOf(Person gc)  ⟶ Person gf
  with combination C1
  as  Select distinct gf
      for each Person p
      where FatherOf(gf,p) and ParentOf(p,gc));
```

The default semantics of applying such a predicator are that the Combination object C1 is searched first for an extensional assertion about the given Persons `gc` and `gf`, and only if none is found will the intensional formula be evaluated. Other treatments of semantics, such as checking for conflicts between the extension and the intension, can be explicitly specified if desired.

Explicit specification of semantics becomes a much bigger issue for update of intensional or hybrid concepts. The default for Assert or Retract is to make some minimal unique change, if such can be found, to some other extensional information so that the effect is achieved via the intensional part of the concept definition. To override these defaults, the Assert and Retract actions can be defined for specific predicators:

References

[Beech 87]

Beech, D. Groundwork for an Object Database Model. In: Shriver, B. and Wegner, P. (eds.) Research Directions in Object-Oriented Languages. MIT Press, 1987.

[Clocksin & Mellish 81]

Clocksin, W.F. and Mellish, C.S. Programming in Prolog. Springer-Verlag (1981).

[Codd 70]

Codd, E.F. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM* 13:6 (1970 June), 377-387.

[Fishman *et al.* 87]

Fishman, D.H., Beech, D., Cate, H.P., Chow, E.C., Connors, T., Davis, J.W., Derrett, N., Hoch, C.G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M.A., Ryan, T.A., and Shan, M.C. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems* 5:1 (January 1987), 48-69.

[Goldberg & Robson 83]

Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley (1983).

[Schwartz 73]

Schwartz, J.T. The SETL Language and Examples of its Use. Courant Institute, New York University. 1973.

Generator is a subtype of Action, and thus a Generator object also has a Formula in its Action part, which must take just an Integer parameter, and may return any Type of result.

The basic generation of the nth element of a sequence is carried out by "applying" the Action part of the composite Action/Generator object to the Integer n. The Generator actions provide additional capabilities for handling a mixture of extensional and intensional information, and for iterating over the elements of the sequence.

Assert and Retract are intended to accommodate updates to the extension of a Generator, as far as possible transparently with respect to the nature of the defining Formula. If the Formula is purely defined in terms of the Assertions part of the Generator, then appropriate changes are made to these assertions. Otherwise, the default semantics are that some minimal permitted extensional change is to be made elsewhere if possible, such that evaluation of the Formula would now show the desired change in the extension, but no other—failing this, an exception is raised.

Iterate takes an Action, which must have just one parameter of the Type returned by the Formula and may return a result of any Type, and applies it to each member of the sequence in turn. An upper bound may be specified on the number of iterations, or may be Null to indicate the absence of a bound. The results of the Actions are collected by appending them to a List which is returned when the Iterate action is complete.

IsGenerated tests whether the given Object is producible by the Generator, and may return Unknown if it is unable prove truth or falsity.

4. Conclusion

A model of intensional concepts in an object-oriented database language has been briefly presented at two levels—an intuitive higher level in order to illustrate its potential as an evolutionary continuation of some existing programming languages and database languages, and a more primitive level defining the essential semantic actions beneath the syntactic sugar.

Something very close to the foundations of this model has been implemented as part of the Iris system [Fishman *et al.* 87], although this presently does not distinguish predicators and generators from other actions, and the view update capabilities are very restricted.

Much future work will be needed to explore how far the default semantics can be carried by the system, how unfruitful searches will be terminated by timing out or other measures, and what new forms of query optimization are called for. The model is intended as a suitable framework for addressing the problems of combining inference engines and database systems, in the hope of cumulative progress rather than an immediate breakthrough.

Another major question which arises, as always, is the extent to which the language used to define actions for specifying intensional concepts and their update semantics needs to approach a full-fledged programming language. The conclusion section of a paper is no place to begin that discussion, but we have indicated earlier that we allow for enough language to give us computability of recursive functions, and also anticipate the use of foreign actions written-in other languages.

expect VISION can be extended in the future.

2. Modeling Complex Applications

2.1. The Application Development Process

Traditionally, information intensive applications have been developed using systems that separate data management facilities from programming environments. Consider Figure 1. Ideally, the database management system is used to codify the declarative semantics of the application, and the operational semantics of the application is captured in one or more programs. In actuality, however, this separation is not so clean. A database management system only manages the declarative semantics of the shared, persistent data; the application programs must define and manage transient data themselves. Although application programs support the bulk of an application's operational definition, the database's data manipulation language provides some operational capabilities as well. In addition, although application programs can be considered part of a shared, persistent information base, they typically are not managed by the data management system.

This seemingly arbitrary division between the database management system and the programming language makes application development awkward at best. Typically, only the simplest applications -- such as traditional record keeping systems -- have made good use of these divided database/programming systems. The reason is that simple applications deal with a relatively small set of simple data types, which can be conveniently isolated by the database system from the programs that manipulate them. As applications begin to model fine-grained, real-world entities and systems, they generate large and complex sets of data types. And because a significant component of a complex type's definition is operational, it becomes increasingly difficult to separate the declarative structure of the data from its operational semantics.

The concept of an *Application Development Platform* is an alternative to this partitioned, dual implementation model. The goal of an application development

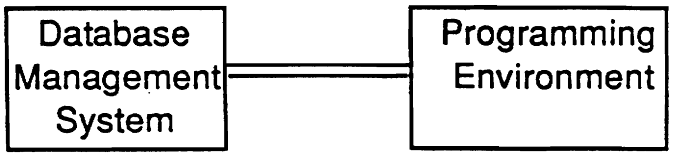


Figure 1. The Traditional Application Development Model

modeled at different levels of abstraction. An application might need to have more detailed information about one aspect of a company than another: either the information is not available, or the users of the application are more interested in some details than others, and are willing to spend more time developing specific parts of the overall model.

Thus investment modeling is a complex application which generates a large number of objects and uses a large quantity of statistical data. Simple, tabular structures like relations do not have the capabilities to model entities at different levels of abstraction. An entity is not just one tuple in a relation; rather, it is several tuples scattered across several relations, which must be related in a specific way.

Time is another important concept that must be captured in order to model investment applications effectively. At least two independent time dimensions exist. A system must be able to capture the history of an object both in the user's world and in the system's model of the world. It must also be possible to discuss various alternative scenarios of what an entity will be like in the future. Time interacts fundamentally with a language's mechanism of update and with a user's view of the database. If an extensible treatment of both is to be provided, an appropriate linguistic home must be found for time.

Database schemas are not static; an application's model of the world must be able to evolve. It is unreasonable to expect a designer to know the complete details of a model when it is created. Consequently, efficient addition, alteration, and deletion of properties applicable to previously existing entities is essential. As types are specialized, it must be possible to selectively refine existing individuals to acquire the behavior of the specializations.

The database required by an investment management decision support system is large. Underlying most applications are several hundred megabytes of statistical data. Iteration over collections of this information is common. Although some iterations are associative, many iterations involve some form of general computation, making them unsuited to traditional "index" based optimizations. Thus the system architecture must be able to handle iteration in non-traditional ways.

The VISION approach is to remove the partition between database management and programming. From the database perspective, we want to eliminate the need for a host language. From the language perspective, we want to add the notion of persistance and sharing. Any capability should be viewable from either perspective. In some cases, such as encapsulating interation, it is important to take the database perspective. And in other cases, such as encapsulating behavior and execution environments, we want to take advantage of efficient programming language techniques.

3. The VISION Language

In general, the operation "*!y <- x specialized*" creates a new collection whose prototype object is *y*, such that *y super* is *x*. Thus, the collection hierarchy is represented by an object tree of prototypes, with *object* as the root prototype.

Prototype objects are very much like other objects in collections. For example, they have function values, and can be manipulated in VISION expressions. The only difference is that the prototype of a collection does not show up in an enumeration of the collection; the prototype is in a sense the "zeroth" object in a collection.

If *y* is a prototype, then the expression "*!z <- y new*" creates a new object that is "just like *y*". That is, it creates a copy of each object in *y*'s superchain. Note that the function values of *y* act as default values for the collection. This new copy can then be given different values for its functions.

For a concrete example, the following VISION code defines part of the scheme in Figure 3, and creates some instances.

```
!company <- object specialized;
!person <- object specialized;
person defineFixedProperty: 'name' .
        defineFixedProperty: 'phoneOf';
!employee <- person specialized .
        defineFixedProperty: 'worksFor';
!gm <- company new;
!joe <- employee new;
joe :worksFor <- gm;
```

In sum, then, a user of the VISION language sees and manipulates only collection instances. There is no need to worry about names of collections, or the difference between *instance-of* and *subcollection-of* edges. Of course, the user has to be aware of the collections, since the semantics of the operations are defined in terms of them. But in general, the language is simplified and made more flexible.

The difference that VISION has from the prototypes of [L] is that the VISION user, when adding a new object *y* similar to *x*, must decide whether it is *x new* or *x specialized*. In the first case, the prototype values are copied; *x* and *y* become equals, sharing the same protocol. In the second case, the prototype's values are shared; changing values of the prototype *x* will change the specialization *y*. The use of prototypes is also similar to the language SELF [US]; its main difference is that the VISION system manages collections internally, for the sake of efficiency.

3.4. Polymorphic Functions

One way that VISION generalizes traditional database languages is in its treatment of polymorphism. In VISION, a function can map a collection to several collections. Such a function is called *polymorphic*. For example, consider the function *phoneOf* from Figure 3. That function is shown as mapping *person* to

In general, the function *x extendTo:y* clones the prototype object *y* and sets the *super* of the new object to be *x*.

3.5. Object Specialization

One feature of non-homogeneous collections is that object trees need not correspond exactly to the collection hierarchy. This property follows from the fact that different objects in the same collection can have different behaviors. Consider the collection *stockholder* above. The object trees for this collection appear in Figure 5. Note that there are three different tree structures: the prototype tree, the tree for person stockholders, and the tree for corporate stockholders.

Since prototypes are not much different from other objects, functions such as *new* and *specialized* should be applicable to all objects equally. That is, objects should be able to clone or refine themselves on an individual basis. We call such an ability *object specialization*; to our knowledge, no other language has this feature. For example, if *x* is an object, then the expression *x new* creates a copy of *x* and every object in *x*'s superchain. This feature is useful when a collection is non-homogeneous. In the *stockholder* example, we can create new corporate instances by saying *"gm2 new"*; new individual stockholders are created by saying *"joe2 new"*. The expression *"stockholder new"* creates a new *stockholder* object, which is neither a person nor a company.

Another useful consequence of object specialization is that an object can be the *super* of several other objects. This feature is necessary in analyzing future scenarios. Consider for example the object *ford* in class *company*. In order to examine the effect of inflation on the expected future price of the stock, we can create a subclass of *company*, having properties *inflation-rate* and *future-price*. For each inflation rate we wish to examine, we create a new object in this class, which is a refinement of *ford*. That is, we say:

!futureCo <- company specialized;

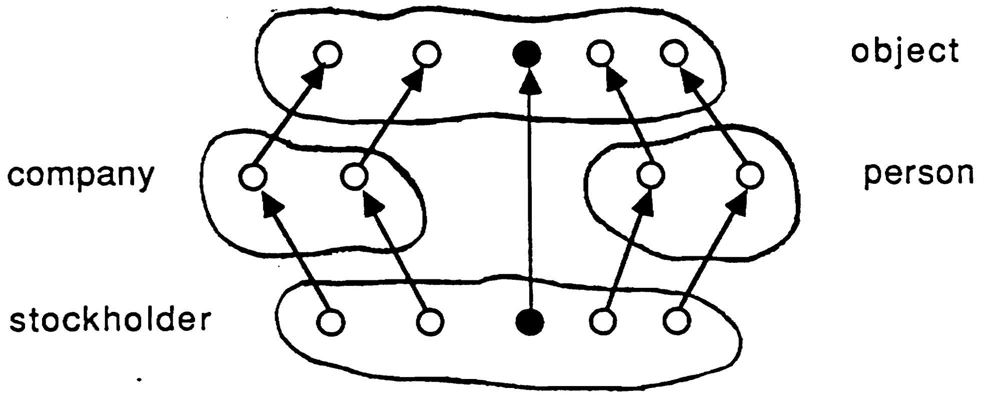


Figure 5. Stockholder Object Trees

ratings must be a function defined in the collection *TA*. The VISION model of this application is shown in Figure 6b. That figure models three entities, one of which is just a student, one is just a teacher, and one is a TA. Again, object specialization allows inheritance to be performed via object superchains, without the need for complex multiple inheritance machinery.

3.6. Functions as Objects

An important feature of VISION is that functions are treated as first-class objects. In VISION, there are two types of message: extensional and intensional. An *extensional message* yields the result of evaluating the function it selects. For example, the message "*gm sales*" yields 96371.63. An *intensional message* yields the function itself. Intensional messages are expressed by placing a colon before the function name. For example, the message "*gm :sales*" yields the function that connects the object *gm* to its sales value.

Since functions are objects, they also belong to collections, respond to messages, and are organized into subtypes. Refer to Figure 7, which shows a portion of the VISION hierarchy for functions. Every function responds to the message *value*, so the method *value* is defined in the collection *function*. The effect of the *value* message is to yield the extension of the message. Thus "*gm :sales value*" is the same as "*gm sales*".

Functions can be *computed* or *enumerated*. Enumerated functions get their values explicitly; that is, they respond to the assignment message "<-". The most common enumerated function is a *property* (or *attribute*). For example, since *sales* is an enumerated function, we can say "*gm :sales <- 96391.2*". Computed functions are either *methods*, which are user-defined, or *primitive*, which are provided by the system.

By providing functions as first-class objects, VISION avoids many of the anomalies and restrictions of other languages. The most obvious example is assignment. In VISION, assignment is not a special operation on objects. Instead,

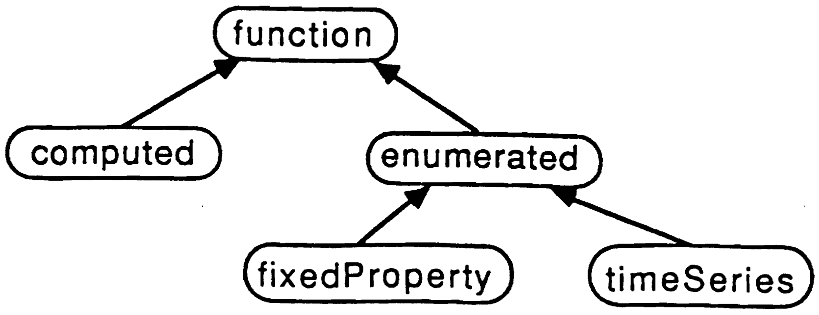


Figure 7. The Function Hierarchy

system creates a new collection containing one function for each local variable. It then extends the object that owns the block to this new collection, and executes the body of the block in the context of this new object.

The most recent context at any time is known as *^current*. The value of *^current* is constantly changing, as blocks are invoked. In particular, each user session has a system-defined environment, which serves as the initial value of *^current*. We have seen that the local variables of a block become functions in the environment. Similarly, all variables created in a user session are just functions in the initial *^current*. That is, the expression *"!gm <- company new"* creates a function called *gm* in *^current*. Consequently, except for certain system-defined names like *^current*, names in VISION do not refer to objects; instead, all references are to functions. Thus the expression *"gm sales"* is technically not legal. It is treated by VISION as a shorthand for the correct expression *"^current gm sales"*.

It is interesting to see how the mechanism for block execution is related to object specialization. For example, consider the following use of the function *extendBy*:

!x <- gm extendBy:[!pe <- price/earnings]

Here, *x* is a specialization of *gm*, which is a member of a new collection having the function *pe*. The effect of executing the function *extendBy* is to create a new environment; however, instead of returning the value of the block, the environment itself is passed back as the value of the expression.

4. The Physical Architecture

Physically, the VISION system is divided into two components: the language interpreter, and the object manager. Although this division superficially resembles the traditional division between the programming language and the database, there is much more cooperation in the VISION system. In particular, both components work together towards encapsulating iteration; both the virtual machine interpreter and object manager operate on collections as their basic unit of computation and structural organization.

4.1. The VISION Virtual Machine

Three performance bottlenecks have traditionally dominated the design of object-oriented systems -- message dispatch, function (or method) activation, and garbage collection. While techniques such as method caching and stack based allocation can reduce the overhead associated with these operations, these optimizations are intrinsically serial in their approach. While serial optimizations reduce the absolute time to perform individual operations, they still interact multiplicatively with common types of collection iteration. Consequently, when traditional object-oriented systems try to scale up from small collections to large ones, performance drops dramatically.

and executes them in parallel. Second, experience has shown that most queries do not request entire objects, but only a small portion of them; consequently, by storing functions the object manager increases clustering and reduces the reading of unneeded information.

The VISION object manager uses an identical representation for transient and persistent data. This strategy allows the virtual machine to not care about the location of an object, and whether it is persistent or transient. The common format eliminates all conversion penalties associated with access to persistent data. As a result, persistent data can be viewed by a process as virtual memory resident. Object faulting is handled as a by-product of hardware load and store instructions and standard virtual memory paging operations. Because faulting is triggered by hardware load and store instructions, access to VISION objects does not require mediation by a separate software buffering layer.

The VISION object manager currently uses a multiversioned optimistic concurrency control method for persistent data. This strategy guarantees that a read-only transaction will always be able to see a consistent view of the database, and will never abort. Optimistic concurrency control appears to be especially appropriate for interactive modeling applications, where a large percentage of transactions are either read-only or affect only a user's private data.

5. The Future

VISION currently is in active commercial use, supporting the interactive use of databases containing several hundred megabytes. In practice, the pipelined parallel architecture of its interpreter has proven effective at reducing the overhead of message dispatch, context switching, and garbage collection associated with operations that iterate over collections. The treatment of functions as first-class objects along with the notion of dynamically bound temporal context has allowed time to be treated in a natural and compact manner.

Current research efforts involve extensions to the function type hierarchy to accommodate type specific concurrency control mechanisms, type-specific query optimization strategies, and object versioning. Additionally, we are investigating mechanisms for extending the global context to areas other than time in order to unify the treatment of encapsulation of objects and access control.

6. References

[C] Codd, "Extending the Database Relational Model to Capture More Meaning". *ACM TODS*, December 1979, pp 397-434.

[CT] Clifford and Tansel, "On an Algebra for Historical Relational Databases: Two Views". *Proc. ACM SIGMOD Conference*, 1985, pp 247-267.

[DKL] Derret, Kent, and Lynbaek, "Some Aspects of Operations in an Object-

Implementing functional databases

Guy Argo
Glagow University

Jon Fairbairn
Glasgow University
(visiting from
Cambridge University)

John Hughes
Glasgow University

John Launchbury
Glasgow University

Philip Trinder
Glasgow University

style. To implement these abstract operations we shall use an ordered binary tree. We use the symbols <, =, > to express the ordering of the name values.

Our binary tree can be represented by the Miranda datatype (with Greek letters instead of asterisks)

```
bintree α ::=   Node (bintree α) α (bintree α) | Nil
```

The intention is that the type α will be the cross product of two other types: one representing names; and the other representing values.

Polymorphic versions of insert and lookup can be written as

```
insert new Nil = Node Nil new Nil

insert (n2,v2) (Node left (n1,v1) right)
                 = Node (insert (n2, v2) left) (n1,v1) right    IF  n2 < n1
                 = Node left (n1,v1) (insert (n2, v2) right)    IF  n1 < n2
                 = Node left (n2,v2) right                      OTHERWISE

lookup key Nil = error

lookup n2 (Node left (n1,v1) right)
                 = lookup key left                 IF  n2 < n1
                 = lookup key right                IF  n1 < n2
                 = v1                              OTHERWISE
```

Notice that insert does not modify the existing database: it returns an entirely new database in which the modification has been performed. This may appear expensive but it involves creating only d new nodes, where d is the depth of the key in the tree. The new nodes point into the original database to share any unchanged nodes (diagram 1). Thus the insert operation has the same order of complexity as an imperative version, (but with a larger constant factor because copying is more expensive than updating in place).

As common nodes in the different versions of the tree are shared, we can cheaply retain a copy of an old database by keeping a pointer to it. Old nodes are reclaimed by the garbage collector when they are no longer referred to. This property has several useful applications. For instance, a large read-only transaction can be given a pointer to the database (which unchanged. This means, for example, that an interactive shell with an undo command (to revert the database to an earlier state) would be trivial to implement by maintaining a stack of database pointers.

The relationship to conventional databases

In our example we have modelled the database with a binary tree. But, as conventional database implementations rely heavily on secondary storage, B-trees are preferred to binary trees. The difference is not an important one. We are not bound to a binary tree implementation; it merely makes the description simpler.

In conventional databases updates are done in place. Whilst this doesn't give the sharing advantages discussed above, it does provide an increase in performance. If our implementation uses a reference counting garbage collector, we can ensure all nodes with a reference count of 1 (i.e. those nodes not shared) are updated in place. This will not alter the semantics of our model, and in such cases will execute almost as efficiently as the imperative version. This combines the advantages of both approaches. The result is like shadow paging [Hecht & Gabbe], a technique used in conventional databases to support abortable transactions. The transaction writes to unused pages and commits by overwriting the root. If the transaction aborts, the root remains unchanged.

2. Multiple Users

To cope with multiple requests a DBMS must be able to handle asynchronous inputs. The issue of combining input from many sources has already been tackled in work on functional operating systems [Henderson, Stoye]. These use variants of Henderson's non-deterministic merge. We assume a similarly appropriate solution is employed in our DBMS and therefore restrict ourselves to regarding the input to the manager as a list of requests.

Requests are functions

What appears in the input stream? In other words, what sort of operations would we want of the shared database? We must still be able to interrogate it with general queries. A query may always be expressed as a function from the database to a domain of answers. This function can be

```
available :: flight -> integer -> db -> boolean
```
(returns True if the flight has sufficient free seats available), and
```
book      :: flight -> integer -> db -> db
```
(will book the seats on the flight - that is, the new database will have the bookings recorded).

We could define a function if_ok_book by

```
if_ok_book flt n dbs
        = ( "Ok", book flt n dbs )          IF available flt n dbs
        = ( "No room", dbs )                OTHERWISE
```

This simple definition will ensure that no two attempts to book the same seats can occur because there is no chance for the database to change between the query and the action. Notice that this is another example of an abortable transaction.

In practice it may turn out that certain ways of combining requests into transactions occur particularly frequently. If so, we could take advantage of this and define combining forms using higher order functions.

In the rest of this paper the terms request and transaction will be used interchangeably. These differ only in the way that one might think about them. In particular they are both functions of the same type, so no confusion should result.

Integrity of the database

Since a transaction is just a function, there could be transactions which do not terminate, take too long (according to some criterion), or corrupt the data in some way. How can we defend the database against rogue transactions? Consider long and non-terminating transactions first. In conventional databases, transactions may be timed out and aborted if they take too long. At first sight, it seems that timeouts cannot easily be fitted into the semantics of functional programming. However, if we are willing to accept non-determinism we can introduce a primitive that decides non-deterministically whether to apply a transaction or not. On a machine level the guiding factor would be the time taken by the function. From the standpoint of the user, the database appears just like a conventional database, in that any transaction submitted may or may not be performed. What we have altered here is not the semantics of the user's depends on the new data. The second transaction cannot be allowed to read the database until the first transaction has finished with it - the data must be locked. The imperative solution is for a transaction to mark all the nodes that it may change, so denying access to other transactions until the updates are performed. In the functional approach, locking takes place automatically, as a result of data dependency. If part of the tree is still being evaluated by one function then no other function can read the value until the tree (or enough of it) has been computed. In a later section we discuss ways of minimising locking. In contrast to the conventional case, "locking" applies to any datum - even individual fields of records.

The drawbacks

Unfortunately, some things do not work so easily. Some of the methods we described above can severely limit concurrency. For example, an abortable transaction effectively locks the entire database throughout its execution. This is because neither the original nor the replacement database is returned until the decision whether to abort has been made. Therefore, no other transaction may use any part of the data until after this decision. Only then will one of the roots be returned. This applies even if the first does not affect the data required for the second.

A similar problem arises if we use balanced trees. Insertion may require rotations anywhere along the path to the inserted item in order to maintain the balance. Usually such rotations are performed deep in the tree, near the point of insertion. But, occasionally, the root of the whole tree is rotated. It is only possible to recognise whether or not this will occur after all the other rotations have been performed. As a result, the insertion function will lock the root throughout its execution, preventing any concurrent operation. An alternative is to leave the tree unbalanced after an insertion, but then rebalance the whole tree periodically. This has problems too: rebalancing a large tree is a time consuming operation, during which no other access to the database is possible.

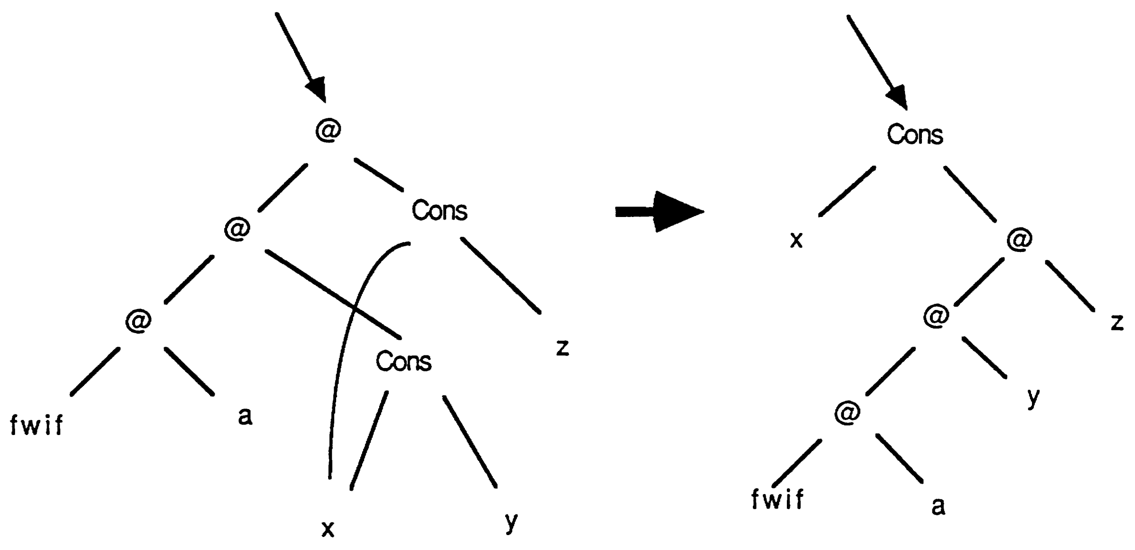
Any of these problems is sufficient to drastically reduce concurrency. In the next few sections we explore methods for solving the problems.

Friedman and Wise if

As we mentioned above, abortable transactions lock the root of the

$$\begin{aligned}
\text{fwif}\ a\ (x{:}y)\ (x{:}z) &= \text{if}\ a\ (x{:}y)\ (x{:}z)\ \sqcup\ ((x{:}y) \sqcap (x{:}z)) \\
&= \text{if}\ a\ (x{:}y)\ (x{:}z)\ \sqcup\ x : (y \sqcap z) \\
&= (\text{if}\ a\ x\ x\ \sqcup\ x) : (\text{if}\ a\ y\ z\ \sqcap\ (y \sqcap z)) \\
&= x : \text{fwif}\ a\ y\ z
\end{aligned}$$

Diagram 2. Fwif returns common parts early



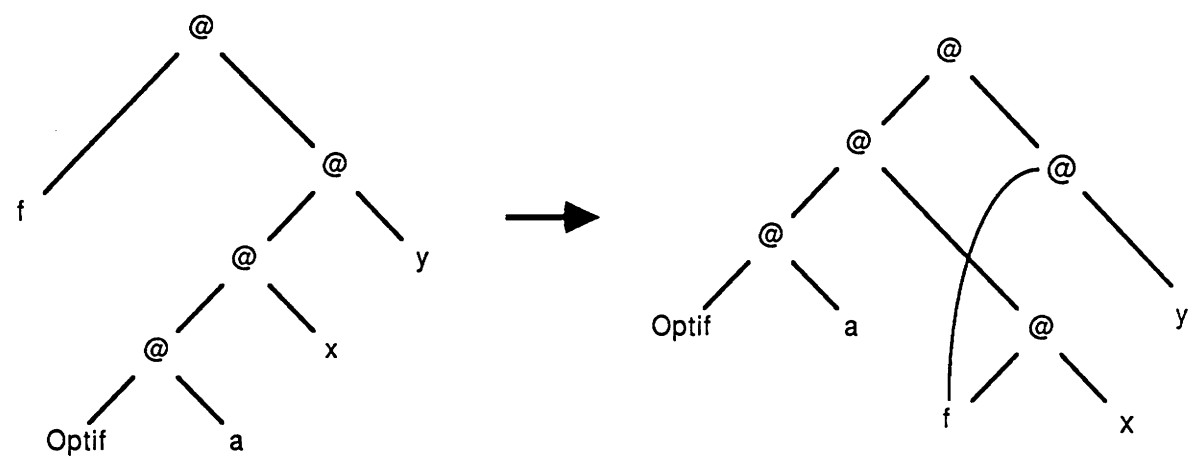
Optimistic if

Often abortable transactions have the form:

```
if predicate db
then transform db
else db
```

It may be that in most cases the predicate will return True allowing the transaction to proceed. In other, rarer, cases the predicate will return False and the transaction will be aborted. Normally the predicate is evaluated, and only when its value is known is one of the branches evaluated. This is a sequential process. In order to increase the level of concurrency, we may take advantage of the supposition that the **then** branch is the most likely to be chosen and start evaluating it immediately. To do this we propose to use an "optimistic" if (**optif**). **Optif** begins the

Diagram 3. Distribute functions over optimistic if



Long Transactions

If one user submits a long transaction, and then submits another which relies on the result of the first then they must expect a delay while the first completes. But, if two users submit interrelating transactions simultaneously, it is not reasonable to expect the shorter transaction to wait on the longer. Instead it would be desirable to impose the ordering that the short transaction is to be dealt with first, and then the longer. How do we decide which is shorter? The solution is to evaluate both transactions concurrently. As soon as one has completed we arrest the other and re-evaluate it on the new database. This gives us the ordering that we require. However, the second transaction may already have been largely evaluated on the old tree, and it may be that the new tree is not very different from the old, which means that computation is repeated. We claim that although some repetition of work is unavoidable, it may be reduced with the use of lazy memo-functions [Hughes]. If the long transaction is memoised, then any intermediate results from unchanged parts of the database are preserved. When the transaction is re-evaluated, these results may be used immediately. In the best case the second transaction may be almost instantaneous.

Balancing

To guarantee good access time, we must keep the database tree balanced. One method would be to schedule a transaction to rebalance the whole tree operational behaviour closely mimics that of a conventional DBMS. We also discovered unexpected bottlenecks that could restrict concurrency severely. We overcame these by introducing unusual concurrent and non-deterministic operators.

We conclude that functional languages are promising for database implementations; and also that new primitives may be necessary if functional languages are to make full use of concurrent machines.

Henderson, P.
"Purely Functional Operating Systems"
Functional Programming and its Applications
edited by Darlington, Henderson, & Turner p 177-189
CUP 1982

Hughes, R. J. M.
"Lazy Memo-Functions"
Proceedings of the Workshop on
Implementation of Functional Languages p 400-421
University of Goteborg & Chalmers University of Technology
Report 17, February 1985

Nikhil, R.
"Functional Databases, Functional Languages"
proceedings of the Persistence and Data Types Workshop,
Appin, August 1985, 209-330

Stoye, W.
"A new scheme for writing Functional Operating Systems"
Technical Report 56, 1984
University of Cambridge Computer Laboratory

RELATIONAL DATABASE CONSTRUCTS

In common with other Pascal-based database programming languages, the relation data type in RAPP is based on the existing record data type. For example, a relation students with attributes *st#* (the key attribute), *stname*, *status*, and *dateofbirth* might be defined by the following declarations:

```
type
    strange         = 0..9999 ;
    string          = packed array [1..30] of char ;
    statustype      = (undergrad, postgrad, research) ;
    datetype        = packed array [1..6] of char ;
    studentrec      = record
                            st#      : strange ;
                            stname   : string ;
                            status   : statustype ;
                            dateofbirth : datetype
                      end ;
    studentrel      = relation [st#] of studentrec ;
var
    students : studentsrel ;
```

A recent development in the RAPP system permits an attribute type to be an abstract data type. The construction of such types is described in the next section.

The operators provided by RAPP for manipulating relations are (in common with the language PLAIN) based on the relational algebra [7]. These operators consist of selection, projection, natural join, Cartesian product and the set operators of union, intersection and difference. A full description of of these operators is given in reference [5].

Relations may be indexed on any attribute by means of an index relation. Index relations are created by the user, but subsequently they are automatically maintained by the system when the base relation is updated.

ABSTRACT DATA TYPES

There are many database application areas where the data structures are of such complexity that the primitive typing facilities offered by commercial database management systems are found to be totally inadequate. In the design of large applications, data abstraction has long been recognised as a means to develop high-level representations of the concepts that relate closely to the application being programmed and to hide the inessential details of such representations at the various stages of program development. Thus many modern programming languages such as Ada and Modula-2 offer very general algorithmic facilities for type definition. Module or 'information-hiding' mechanisms are provided so that arbitrary new types can be defined by both the necessary details for representation, which are hidden from the surrounding program, and the allowable operations to be maintained for objects of that type. Furthermore, since these mechanisms may be applied repeatedly, types may be mapped, step by step, from higher, user-oriented levels to lower levels, ending with the built-in language constructs. At each level, the view of the data may be abstracted from enables a variety of tracing, monitoring and recovery strategies at block level which few other languages support [1]. For example, we could make the execution of block B conditional on the successful opening of the text file by replacing the body of the envelope with the following code:

```
begin
    openfile ;
    if {file successfully opened}
    then begin
            *** ;
            close file
        end
end ;
```

Abstract data types which are to be employed as attribute types in RAPP are most effectively constructed as 'starred' type declarations within envelope modules. As a simple example, let us consider the attribute *dateofbirth* which was declared to be of type *packed array [1..6] of char* in the example above. This is a rather inadequate type and we may wish to define a more structured type for *dateofbirth* and provide operations on objects of that type such as:

1. Compute the number of days between two dates;
2. Given a date d, compute the date n days later;
3. Return the day of the week corresponding to a given date.

An envelope module for the abstract data type *datetype* providing the above operations might take the following form:

```
Envelope Module DateModule ;
type
    *DateType = { the structure of DateType is hidden }
    *DayType = (*Sunday,*Monday,*Tuesday,*Wednesday,*Thursday,
                                *Friday,*Saturday) ;

    Function *NoOfDays ( d1, d2 : DateType ) : Integer ;
        { Computes the number of days between dates d1 and d2 }

    Procedure *NewDate ( d : DateType ; n : Integer ; var result : DateType ) ;
        { Given a date d, computes the date n days later }

    Procedure *DayofWeek ( d : DateType ; var day : DayType ) ;
        { Returns the day of the week on which a date d falls }

begin
    ***
end { DateModule } ;
```

A user of the module *DateModule* may declare variables and attributes of type *DateType* in his program and apply the operations *NoOfDays*, *NewDate*, and *DayOfWeek* to those variables. He does not know, and does not need to know, how *DateType* is implemented.

```
Monitor RelationAccess ;
type
    *AccessMode = ( *Read, *Write, None) ;
instance
    readers, writers : Condition ;
var
    CurrentAccessMode : AccessMode ;
    NoOfReaders : 0..Maxint ;

    Procedure *Acquire ( AccessRequired : AccessMode ) ;
    begin
        case CurrentAccessMode of
            None:   begin
                        CurrentAccessMode := AccessRequired ;
                        if AccessRequired = Read
                        then NoOfReaders := 1
                    end ;
            Read:   if AccessRequired = Write
                    then writers.wait
                    else if writers.Length = 0
                        then NoOfReaders := NoOfReaders + 1
                        else readers.wait ;
            Write:  if accessRequired = Read
                    then readers.wait
                    else writers.wait;
        end {case} ;
    end {Acquire};

    Procedure *Release ;
    var
        NoCurrentReaders : Boolean ;
    begin
        if CurrentAccessMode = Read
        then begin
                NoOfReaders := NoOfReaders - 1 ;
                NoCurrentReaders := (NoOfReaders = 0) ;
                if NoCurrentReaders
                then if writers.Length > 0
                    then writers.Signal
           end
        else
          if readers.Length > 0
          then begin
                while readers.Length > 0 do
                begin
                    readers.Signal ;
                    NoOfReaders := NoOfReaders + 1
                end ;
                CurrentAccessMode := Read
             end
          else if writers.Length > 0
              then begin
                    writers.signal ;
```

2 Experiences in Integrating Results

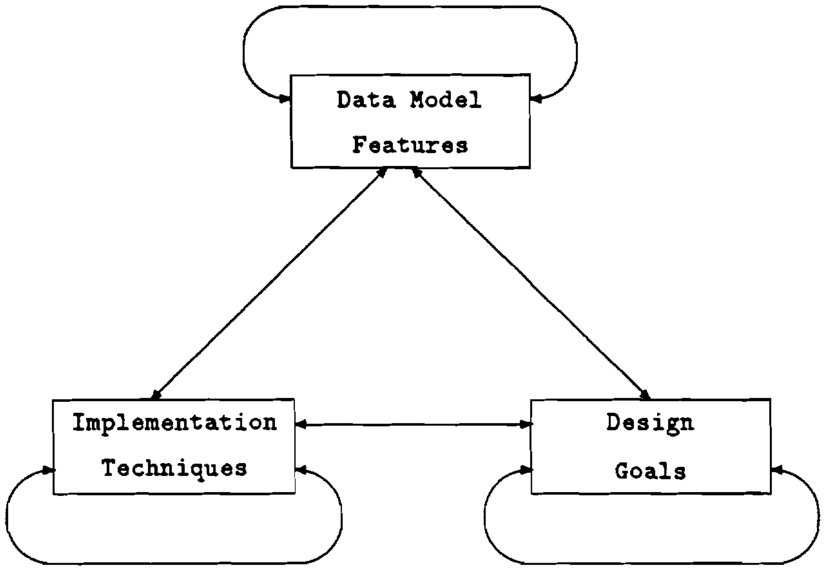
When applying "off-the-shelf" technology from areas such as databases and compilers to semantic data model implementations, our experience has shown that integration problems often arise. Unfortunately, these are often discovered in the middle of an implementation, necessitating some re-design work. This motivates our desire for more systematic tools.

We now discuss examples of integration problems, with some emphasis on our experiences at the University of Toronto [Nixon, 1983] [Chung, 1984] [Nixon, 1987a,b] building a compiler for Taxis, a language for designing large interactive information systems, using some knowledge representation facilities [Mylopoulos, 1980] [Wong, 1981]. Many of our observations apply to semantic data models in general, while some are specific to Taxis.

We see three main components in implementing semantic data models:

1. *data model features* (e.g., abstraction mechanisms, data manipulation operations, programming constructs).
2. *implementation techniques* (e.g., management of processes and of secondary storage).
3. *design goals* (e.g., reliability, safety).

The problem is that one cannot add ingredients in a linear fashion; rather, one must consider interactions between ingredients. Interaction (conflict) can occur between two aspects of one component (e.g., two data model features), or between aspects of two or all three components. We present several examples of the these kinds of interaction.



In the remainder of this paper, we assume that the reader has some familiarity with Entity-Relation-based data models in general, and in particular with the Taxis data model and its implementation.

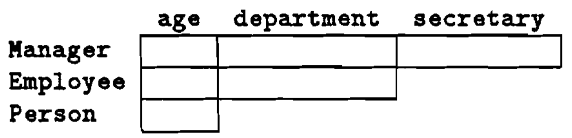
programming constructs.

There are two observations. First, the incorporation of two features which are not completely orthogonal can cause problems. Second, *formal* semantic data models (e.g., [Abiteboul, 1984]) may be more amenable to logic-based techniques for detecting underlying interaction of features during data model design. Related work from the programme verification field includes [Elliott, 1982], which derives conditions for the absence of errors, by way of re-write rules applied to (possibly interacting) programming language constructs. However, the problem of detecting interaction is very difficult.

2.2 Interaction between Features and Implementation Techniques

Our experience has been that when trying to apply existing techniques, the "obvious" method is sometimes inappropriate or inefficient. Let's give a few examples.

When storing large amounts of persistent data, a natural place to look is relational database technology. However, inheritance hierarchies result in collections of attribute values whose appearance is more like a "staircase" than a relation-like grid.



Thus more effort is needed to obtain a compact, efficient representation, such as using vertical or horizontal partitioning of attribute storage [Chan, 1982] and associating information about related attributes or sub-classes.

When implementing "triggers" (conditions which, when satisfied, invoke actions) there are many scheduling techniques available from systems software. However, if a condition such as **`when John.salary > 25000 do`** ... is translated to a monitor-like "wait" construct, the result could be very inefficient due to repeated evaluation of the condition. It would be better to analyse the condition at compilation, and produce code with *selective* checking; see [Chung, 1984] and [Nixon, 1987b] for details. It turns out that there are tradeoffs among the different techniques. This points out the need for a formalism to choose the best alternative.

When modelling the performance of long-term processes one may start with results from operating systems modelling. However, the length of, and variance in, the life-times of persistent entities is much greater in an Entity-Relationship-based system than in an operating system [Rios-Zertuche, forthcoming], making it more difficult to apply existing results. Moreover, in an Entity-Relationship-based system, one cannot use the operating-system assumption that older (persistent) entities are more likely to be deleted than younger ones [Butler, 1987]; again, some existing results are not quite applicable.

In some cases, existing techniques need to be *extended*. For example, the Taxis implementation design of semantic integrity constraints was based on techniques [Sarin, 1977] developed for Entity-Relationship-like models; however, they were extended to handle arbitrary nesting of factual attribute selection (e.g., a constraint could refer to the size of the desk of the manager of a department). In addition, an implementation of *temporal* integrity constraints was designed (using techniques also used for "triggers", mentioned above) [Chung, 1984]. Again, there was a feature-implementation tradeoff: Taxis restricts the form of assertions, which permits efficient checking; in fact, [Chung, 1987] states that enforcement is *linearly* proportional to:

$$(\textit{cardinality of source class}) \times (\textit{length of expression})$$

to be made available as a *data model feature.* It also attains a *design goal* of letting the *programmer handle* a violation of the referential integrity constraint (by modifying relevant attribute values and then deleting an entity), rather than only having the *system detect* the violation.

A seemingly independent constraint is that every Taxis entity has a unique "minimum class" — the unique lowest class in the IsA hierarchy which contains the entity. Assuming an implementation gives each entity a unique internal identifier, encoding the minimum class in each entity's identifier helps achieve the goal of increasing efficiency of run-time operations. For example, common operations such as finding the most specialised constraint applicable to an entity, or invoking the most specialised transaction, can be performed with reduced access to secondary storage.

A third data model constraint, in the current version of Taxis, is that the minimum class of an entity is fixed over time. It would be very desirable to relax this constraint; for example, a person entity could start as a child, and then become an adult, ceasing to be a child, but remaining a person. Assuming that the minimum class would still be unique at any one time, how hard would it be to permit an entity to dynamically change its minimum class? First we have the problem that an entity's internal identifier occurs in many places throughout a database; *each* occurrence contains the minimum class and would have to be changed. However, this otherwise-expensive operation becomes quite feasible if inverse references have been implemented — the system simply finds and modifies the appropriate internal identifiers. So we have quite an intricate interaction among three data model constraints, their associated implementations, and our design goals and decisions.

Static Typing

Taxis is *not* a statically-typed language, and requires some implementation techniques different from other semantic data models such as Galileo [Albano, 1985a]. Perhaps this can best be explained as the cumulative result of a sequence of decisions concerning the data model and design goals.

The Taxis data model has a *structural IsA constraint* which requires subclasses to inherit the attributes of their superclasses, and to have attribute values which are the same or specialisations of the general attribute value.[4] In addition, the data model requires the most specialised constraint to be applied to an entity. For example, it is not possible to over-ride constraints by saying something like: "Update John's age as if he were just a Person, even though he is also a Child." A design goal was to apply constraints uniformly to all attributes, regardless of the attribute value.

```
define entityClass Person with ...
   age: {| 0::120 |}
   friend: Person

define entityClass Child IsA Person with
   age: {| 0::18 |}
   friend: Child
```

Here `Person` has an integer-valued attribute and an entity-valued one, each of which is specialised in the definition of `Child`. Now what are the implications for implementation of (static) type checking? Consider a transaction fragment:

```
locals
  x: Person
```

[4] Additional attributes may also be defined on subclasses.

instances of a metaclass, and then through the entities which are instances of those classes — there is little type information available about the entities which are instances of instances of the metaclass, resulting in poor type checking, conflicting with another of our implementation goals. However, by further constraining the data model, requiring all instances of a metaclass to be arranged in a lattice with a unique highest element — a "most general instance" — reasonable checking can be achieved, by recognising that the most general instance can be used as a first approximation to the type structure of all the relevant classes, thus providing some information about their instances. Similarly, a related data model constraint on multiple inheritance of attributes [Schneider, 1978] [Nixon, 1987a,b] helped achieve a goal of providing more thorough type checking, and also simplified the implementation.

Type checking also interacted with our design goal of providing informative messages regarding possible run-time errors. We could not simply analyse an expression with respect to the declared classes of variables; instead, we also had to consider their sub-classes, or face the prospect of giving very misleading error messages. Consider a general class (say `Person`) which does not have a particular attribute (say `advisor`) defined, but has two specialisations (say `Student` and `Politician`) which do. When checking usage of an attribute selection (say `x.advisor` where `x` is a `Person`), the message `Persons do not have advisors` is less helpful than `Many Persons do not have an advisor, but Students and Politicians do`. We feel the extra checking and reporting is needed even if code is not generated for the "possible error" case. If, however, code is generated, some run-time type checking will be needed. We also note that the (compile-time) type checking mechanism is further complicated by the possibility of having to propagate more than one value for an expression; in our example, the expression could have three *values*: `illegal` (indicating no value at all), `StudentAdvisor` or `PoliticalAdvisor`.

3 Research Directions

Having reviewed some integration problems arising in implementing a semantic data model, we feel that more systematic techniques will help reduce difficulties. We now consider some ongoing and prospective research areas which should be addressed in developing a theory for semantic data model implementation. Many of these areas have been addressed in the database literature. However, we can foresee some interaction problems, particularly between data model features and implementation techniques.

- *Physical Storage Design*
  An important first step towards a performance theory for semantic data models has been made [Weddell, 1987] by applying analytical techniques first developed for databases. For a reasonable semantic data model which permits multiple inheritance, Weddell considers optimality problems such as aligning records to permit static determination of the location of an attribute value. He attains specific results concerning tractability. His work is geared towards main memory databases; of course many more problems can be considered in the context of two-level storage.

- *Query Optimisation*
  Clearly, there is a wealth of database results to draw upon [Jarke, 1984]. However, specialisation hierarchies can complicate the analysis, as they permit an entity to be an instance of more than one class. In addition, powerful facilities for traversing a database and its meta-knowledge make it harder to narrow down the range of values to which an expression can refer, thus decreasing opportunities for optimisation.

which are shared by two classes (and need only be fetched once from the database during constraint enforcement) [Chung, 1987] [Rios-Zertuche, forthcoming].

4 Conclusions

Could we describe an implementation theory for a semantic data model with a reasonable set of features? While the natural starting point seems to be the application of results from databases and other areas, our experience has been that there are many integration issues. We have reviewed several kinds of interaction in this paper. In reviewing some directions for research, we can foresee some interaction problems that will have to be addressed.

Two basic approaches to development of semantic data models have been identified. One is the evolutionary approach, in which advanced features are added *incrementally* to a programming language (e.g., Galileo [Albano, 1985a]) or to a relational database management system (e.g., POSTGRES [Stonebraker, 1986]). The other is the revolutionary approach, which makes no prior commitment to an existing data model or database (See the discussion in [Brodie, 1986]). In this approach, more than one new feature (and possible a new target architecture) may be handled all at once. But since both approaches require a combination of data model features and implementation techniques, one should be concerned about interaction issues, regardless of the approach taken.[7]

We feel that the development of a theory for semantic data model implementation will require a variety of systematic techniques for measuring and comparing performance alternatives, as well as methods for dealing with the interaction of features, in order to integrate data model features, implementation techniques and design goals.

Acknowledgements

We would like to thank Lawrence Chung, David Lauzon, John Kambanis and other members of the Taxis project for continued assistance in preparing this document. We also thank Raj Verma for helpful discussions. The Taxis project has been supported by a three-year Strategic Grant from the Natural Sciences and Engineering Research Council of Canada.

Bibliography

[Abiteboul, 1984] Serge Abiteboul and Richard Hull, IFO: A Formal Semantic Database Model. *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo, Ontario, April 2–4, 1984, pp. 119–132.

[Ait-Kaci, 1984] Hassan Ait-Kaci, Type Subsumption as a Model of Computation. In Larry Kerschberg (editor), *Proceedings of the First International Workshop on Expert Database Systems*, Kiawah Island, SC, October 24–27, 1984, pp. 124–150.

[Albano, 1985a] Antonio Albano, Luca Cardelli and Renzo Orsini, Galileo: A Strongly Typed, Interactive Conceptual Language. *ACM TODS,* Vol. 10, No. 2, Aug. 1985

[Albano, 1985b] Antonio Albano, Conceptual Languages: A Comparison of ADAPLEX, Galileo and Taxis. *Proceedings of the Workshop on Knowledge Base Management Systems,* Crete, June 1985, pp. 343–356.

[7] Where does Taxis fit in? On one hand, it is not simply an extension to a database system, so it is revolutionary. On the other hand, its data model is not as relatively novel as when it was proposed in the last decade.

[Mylopoulos, 1986] John Mylopoulos, Alex Borgida, Sol Greenspan, Carlo Meghini and Brian Nixon, Knowledge Representation in the Software Development Process: A Case Study. In H. Winter (Ed.), *Artificial Intelligence and Man-Machine Systems,* Lecture Notes in Control and Information Sciences, No. 80. Berlin: Springer-Verlag, 1986, pp. 23–44.

[Nixon, 1983] Brian Andrew Nixon, *A Taxis Compiler.* M.Sc. Thesis, Dept. of Computer Science, University of Toronto, April 1983. Also CSRG Technical Note 33, May 1983.

[Nixon, 1987a] Brian Nixon, Lawrence Chung, David Lauzon, Alex Borgida, John Mylopoulos and Martin Stanley, Implementation of a Compiler for a Semantic Data Model: Experiences with Taxis. In Umeshwar Dayal and Irv Traiger (editors), *Proceedings of ACM SIGMOD 1987 Annual Conference.* San Francisco, CA, May 27–29, 1987, pp. 118–131.

[Nixon, 1987b] Brian A. Nixon, K. Lawrence Chung, David Lauzon, Alex Borgida, John Mylopoulos and Martin Stanley, *Design of a Compiler for a Semantic Data Model.* Technical Note CSRI–44, Computer Systems Research Institute, University of Toronto, May 1987.

[O'Brien, 1982] Patrick O'Brien, *Taxied: An Integrated Interactive Design Environment for Taxis,* M.Sc. Thesis, Department of Computer Science, University of Toronto, October 1982. Also CSRG Technical Note 29.

[O'Brien, 1983] Patrick D. O'Brien, An Integrated Interactive Design Environment for Taxis. *Proceedings, SOFTFAIR: A Conference on Software Development Tools, Techniques, and Alternatives,* Arlington, VA, July 25–28, 1983. Silver Spring, MD: IEEE Computer Society Press, 1983, pp. 298–306.

[Rios-Zertuche, forthcoming] Daniel Rios-Zertuche, M.Sc. thesis, Dept. of Computer Science, University of Toronto, forthcoming.

[Sarin, 1977] S. K. Sarin, *Automatic Synthesis of Efficient Procedures for Database Integrity Checking.* M.Sc. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1977.

[Schneider, 1978] Peter F. Schneider, *Organization of Knowledge in a Procedural Semantic Network Formalism.* Technical Report 115, Dept. of Computer Science, University of Toronto, February 1978.

[Smith, 1983] John M. Smith, Stephen A. Fox and Terry A. Landers, *ADAPLEX: Rationale and Reference Manual.* Technical Report CCA–83–08, Computer Corporation of America, Cambridge, MA, May 1983.

[Stonebraker, 1986] Michael Stonebraker and Lawrence A. Rowe, The Design of POSTGRES. In Carlo Zaniolo (Ed.), *Proceedings of ACM SIGMOD '86 International Conference on Management of Data,* Washington, DC, May 28–30, 1986, *SIGMOD Record,* Vol. 15, No. 2, June 1986, pp. 340–355.

[Tsur, 1984] Shalom Tsur and Carlo Zaniolo, An Implementation of GEM — Supporting a Semantic Data Model on a Relational Back-end. In Beatrice Yormark (editor), *SIGMOD '84 Proceedings,* Boston, MA, June 18–21, 1984, *SIGMOD Record,* Vol. 14, No. 2, pp. 286–295.

[Weddell, 1987] Grant E. Weddell, *Physical Design and Query Optimization for a Semantic Data Model (assuming memory residence).* Ph.D. Thesis, Dept. of Computer Science, University of Toronto, 1987.

[Wong, 1981] Harry K. T. Wong, *Design and Verification of Interactive Information Systems Using TAXIS.* Technical Report CSRG–129, Computer Systems Research Group, University of Toronto, April 1981. Also Ph.D. Thesis, Department of Computer Science, 1983.

[Zdonik, 1987] Stanley B. Zdonik, Can Objects Change Type? Can Type Objects Change? *Proceedings of the Workshop on Database Programming Languages,* Roscoff, France, September 1987.

§1 Informal Introduction

§1.1. Families of Subtypes

In this paper we develop an algebraic model of subtype based on the idea that a type is a form of behavior and a subtype is a behaviorally compatible specialization of the behavior. This specialization occurs in two related ways. A subtype typically describes a more restricted set of elements than the supertype, but may also involve more specialized information on these elements. This notion of subtype is suggested by object-oriented inheritance, but is broad enough to capture other notions of subtype. In particular we examine three notions of subtype known as "subset", "isomorphically embedded", and "object-oriented" subtypes. Examples of these are:

1. Subset: **Int(1..10)** is a "subset" subtype of **Int.**
2. Isomorphically embedded: **Int** is an "isomorphically embedded" subtype of **Real.**
3. Object-oriented: **Student** is an "object-oriented" subtype of **Person.**

Our objective is to characterize both the similarities and the differences between these notions of subtype, and to focus on the algebraic characterization of object-oriented subtypes. We begin by examining informal properties of these three kinds of subtypes and consider the motivations that led to inclusion of these notions of subtype in programming languages. In particular in this section we will work only with an informal, intuitive notion of behavioral compatibility, postponing a more formal description to section 2. These motivating remarks will lead to our definitions of "partial" and "complete" subtypes, also in section 2.

§1.1.1 Subset Subtypes

Subset subtypes restrict the domain of the parent type to a subset, without necessarily considering whether operations of the type are closed over the subset. A subtype **Int(1..10)** of **Int** restricts the domain of integers without regard to the closure of operations such as addition, multiplication or successor on the set {1, ..., 10}. If we restrict the domain and range of the successor function in this way, then it becomes a partial function since successor(10) is outside 1..10. This means that the behavior of **Int(1..10)** is only partially compatible with **Int** because the behavior of successor on the argument 10 for the subtype is incompatible with the behavior of successor for this argument in the parent type. However, if we let the permitted range be 2..11 or some isomorphically embedded subtypes. Consider the type **Person** with the following operations:

name : Person → Character String

age : Person → Integer

add-a-year : Person → Person

Let the subtype **Student** have the additional operation gpa (grade-point average).

gpa : Student → Integer

Operations on the names and ages of students have the same closure properties as for persons. The range of name and age is independent of the parent type, while the operation add-a-year modifies a person by changing its age attribute. Thus if the domain of add-a-year is restricted to students, its values will be students.

Note that subtypes which are defined in terms of the restriction of the range of component operations may cause a breakdown for closure properties. For example, if the class of minors is defined as the class of persons under 21 then functions like "add-a-year" are no longer closed.

When operations have a range which is dependent on the supertype, closure may again become a problem. Consider adding the following operation to **Person**:

parent : Person → Person

Specializing the domain and range of the parent operation to students results in closure problems for the partial subtype. The subtype which uses the original range is needed for behavioral compatibility, just as in the case of subset and isomorphically embedded subtypes, since the parent of a student need not be a student. However, object-oriented operations whose range is restricted to traditional types, as well as those which modify the object to which the operation is applied, are generally well behaved in the sense that they are closed over the partial subtype formed by replacing all occurrences of sorts from the supertype with the corresponding ones in the subtype. Thus the problem of distinguishing between partial subtypes and complete subtypes often does not arise since partial subtypes and complete subtypes are usually equivalent.

§1.2 Algebraic Framework

Traditional algebras have just a single sort that denotes the values of the algebra and have a collection of operation symbols that denote operations for transforming tuples of arguments into values. For example the algebra of integers has the sort "Integer" and operation symbols "+" and "*" that denote binary operations on integers.

Programming language types are modelled by many-sorted algebras whose operations may have arguments and values of more then one sort. For example stacks compatibility, whereas that given for subtypes requires (complete) behavioral compatibility. We believe that this distribution of responsibility leads to a more flexible and inclusive modelling of subtypes. We will also define partial subtypes essentially by modifying clause (3) to refer to "partially behaviorally compatible" operations.

Since inheritance is modelled by the presence of overloaded operators, it is quite important to be able to resolve ambiguity introduced by overloaded operators. [Goguen and Meseguer 1986] introduced a syntactic constraint on programming languages, called regularity, to resolve ambiguities caused by overloading and inheritance.

Intuitively, the ordering of sorts yields a derived ordering on both overloaded operations and types. Regularity implies that terms which can be assigned a sort have a unique least sort. We show that regularity, combined with our definition of subtype, allows us to resolve potential ambiguities due to overloading. In particular, we prove the following uniqueness theorem for terms of a generalized order-sorted algebra:

Theorem: Let $(\Sigma, \leq)$ be a regular signature and A a generalized order-sorted algebra for $(\Sigma, \leq)$. If M can be assigned sort s then all interpretations of M in the carrier of s are identical.

This theorem asserts that even if there is more than one way of assigning a sort s to the term M (due to overloading of operators), all interpretations of M in this carrier are unique. The proof of this theorem is similar to that of the initiality theorem of [Goguen and Meseguer 1986], but is given in a more general setting here.

In summary, the contributions of this paper include:

1. A generalization of order-sorted algebras to provide a subtler and more useful notion of ordering on sorts.
2. Algebraic definitions of type and subtype for order-sorted algebras.
3. A classification of subtypes into complete and partial subtypes, based on preserving complete or partial behavioral compatibility.
4. A demonstration that these notions of subtype capture both traditional and object-oriented notions of subtype.
5. A uniqueness theorem for interpreting terms in the presence of overloading.

§2 Algebraic Models of Type

Algebras can be syntactically specified by their signatures, where a signature specifies the sorts and operations of the algebra. The signature forms the basis for the

Since a type in a programming language consists of both objects and operations on the objects, we will use many-sorted algebras to model the notion of type. Because a type may contain many carrier sets, it is not immediately apparent how to define the notion of subtype. It will turn out, for instance, that the notion of subalgebra from mathematics is too restrictive to capture the richness of subtype in programming languages. In the next section we explore generalizations of many-sorted algebras which will enable us to also model subtype.

§2.2 Generalized Order-Sorted Algebras

The notions of subtype and inheritance can be modelled by first defining an ordering relation on the sorts. [Goguen and Meseguer 1986] have introduced the notion of an "order-sorted algebra" to model subtypes and inheritance.

Definition: An **order-sorted signature** is a pair $<\Sigma, \leq>$, where Σ is a many-sorted signature and $\leq$ is a partial ordering on the sorts in Σ. We extend this ordering to ordered tuples of sorts by writing $<s_1,...,s_n> \leq <s_1',...,s_n'>$ if $s_i \leq s_i'$ for $1 \leq i \leq n$. We say that an operation is **overloaded** if it appears in the signature Σ with two different typings. We will say that t is a **subsort** of s, if $t \leq s$.

An order-sorted algebra with signature $<\Sigma, \leq>$ is a many-sorted algebra with an ordering relation on the carriers of its sorts, and is given by the following definition:

Definition: A is an **order-sorted algebra** for order-sorted signature $<\Sigma, \leq>$ if A is a many-sorted algebra for Σ such that:

(1) if $s \leq t$ then $A_s \subseteq A_t$.

(2) If $f : <w, s>$, $f : <w', s'>$ with $<w, s> \leq <w', s'>$ then $A_f : A_w \to A_s$ and $A_f' : A_{w'} \to A_{s'}$ agree on A_w (where $A_f : A_w \to A_s$ and $A_f' : A_{w'} \to A_{s'}$ are the meanings of the overloaded f.).

Conditions (1) and (2) state respectively that the interpretation of a subsort is as subset (although the reverse is not necessarily true), and that corresponding operations on the subsort and supersort must be "behaviorally compatible." A first approximation at a definition of "behavioral compatibility" could be the following: Corresponding operations on a sort and subsort are "behaviorally compatible" if they are defined for exactly the same elements of the subsort, and where defined, they give the same result.

weakening of the conditions results in some fairly major differences from the definition of order-sorted algebras above. The following example illustrates these differences.

Let Σ_C = <{R, C}; sqrt: <<R>, R>, sqrt: <<C>, C > > be a many-sorted signature and let **Comproot** be the many-sorted algebra for Σ_C in which R and C are interpreted as the sets of real and complex numbers, respectively. In this algebra let sqrt: <<R>, R> be interpreted as $\sqrt{}_R$, the usual partial square root function on the reals, whose domain is the non-negative reals. Similarly let $\sqrt{}_C$ be a total square root function on the complex numbers which extends $\sqrt{}_R$. It is then impossible to make **Comproot** into an order-sorted algebra corresponding to the above signature where $R \leq C$, since $\sqrt{}_R(-1)$ is undefined, but $\sqrt{}_C(-1)$ is defined (typically as i). Note that adding new functions to the signature and algebra will not help since (2) of the definition of order-sorted algebra requires the meanings of all versions of overloaded functions to agree if the domains are related.

Our definition of generalized order-sorted algebra does not suffer from this defect. **Comproot** can easily be made into a generalized order-sorted algebra by adding the coercer $c_{R,C}$, where $c_{R,C}$ is the usual coercer from reals to complex numbers (e.g. $c_{R,C}(r) = r + 0i$). Since $\sqrt{}_R$ and $\sqrt{}_C$ agree on the non-negative reals (the domain of $\sqrt{}_R$), they satisfy (3') of the definition of generalized order-sorted algebras.

Not surprisingly, this is an example of a general phenomenon. It is quite common in mathematics to start with an algebra with partial functions and extend to a larger algebra in which the partial functions become total. One commonly views the extension from the natural numbers to the integers as a way of obtaining a set which is closed under subtraction; the extension from integers to rationals as a way of closing under non-zero division, the extension from rationals to reals as closing under limits of Cauchy sequences; and the extension from reals to complex numbers as closing under roots of polynomials. We feel it would be unfortunate to bar these original partial functions from coexisting in order-sorted algebras with the corresponding total functions on the extensions.

On the other hand, the generalized order-sorted algebras may be criticized as failing to preserve behavior since the corresponding function defined in the subsort may be highly undefined relative to the function defined on the supersort when restricted to elements of the subsort. We see this as a positive feature which reflects the flexibility of the system. However, when we formalize the notions of complete subtype below, we will place stronger restrictions on overloaded functions which will ensure that there is at least one version of the overloaded function in the subtype which is completely "behaviorally compatible" with that in the supertype. We will also introduce a weaker notion of subtype without this restriction that we will call "partial subtype."

and an f : <w',s'> in Σ_1.

Note that the syntactic conditions (1) and (2) ensure that operators from **T$_2$** are inherited in **T$_1$**. The requirement that **T$_1$** and **T$_2$** live in the same generalized order-sorted algebra combines with these syntactic restrictions to ensure that the inherited operations are "partially behaviorally compatible" with the operations in the supertype. That is, if f : $<<s_1,...,s_n>,s>$ and f : $<<t_1,...,t_n>, t>$ with $s \leq t$ and $s_i \leq t_i$ for $1 \leq i \leq n$ in Σ, A is an order-sorted algebra for Σ, a_i in A_{si} for $1 \leq i \leq n$, and $A_f(a_1,..., a_n)$ is defined, then $c_{s,t}(A_f(a_1,..., a_n)) = A_f'(c_{s1,t1}(a_1), ..., c_{sn,tn}(a_n))$, where the A_f on the left side of the equation is the interpretation of the f with signature $<<s_1,...,s_n>,s>$ and the A_f' on the right side of the equation is the interpretation of the f with signature $<<t_1,...,t_n>, t>$. That is, the coercers essentially behave as homomorphisms from the A_{si} to A_{ti} with respect to overlapping function definitions. Note that if $A_f'(c_{s1,t1}(a_1), ..., c_{sn,tn}(a_n))$ is defined, we do not insist that $A_f(a_1,..., a_n)$ be defined.

Thus partial subtypes provide "partially behaviorally compatible" inherited functions. A complete subtype will be a partial subtype in which inherited functions are completely "behaviorally compatible."

Definition: Let **T$_1$** and **T$_2$** be types with signatures $< \Sigma_1, \leq_1 >$ and $< \Sigma_2, \leq_2 >$, respectively, in the same generalized order-sorted algebra A whose signature, $< \Sigma, \leq >$, includes $< \Sigma_1, \leq_1 >$ and $< \Sigma_2, \leq_2 >$. Then **T$_1$** is a **complete subtype** of **T$_2$** iff

(1) For every sort t in Σ_2, there is a sort s in Σ_1 such that $s \leq t$.

(2) For all operators f, if f : <w,s> in Σ_2 and $w' \leq w$ for w' in Σ_1, then there is an $s' \leq s$ and an f : <w',s'> in Σ_1 (i.e. f is inherited on w') which satisfies the following property: If $A_f : A_w \rightarrow A_s$ and $A_f' : A_{w'} \rightarrow A_{s'}$ are the meanings of the overloaded f, then for all $a_1 \in A_{s1},..., a_n \in A_{sn}$, if either of $A_f(a_1,...a_n)$ or $A_f'(c_{s1,s1'}(a_1),... c_{sn,sn'}(a_n))$ are defined, then they both are.

Note that (2) can be simplified to read "... if $A_f'(c_{s1,s1'}(a_1),... c_{sn,sn'}(a_n))$ is defined, then so is $A_f(a_1,...a_n)$." The other implication follows from the definition of generalized order-sorted algebra. For clarity, we leave the definition in the above form. It then follows from the definitions of generalized order-sorted algebra and complete subtypes that if either of $A_f(a_1,...a_n)$ or $A_f'(c_{s1,s1'}(a_1),... c_{sn,sn'}(a_n))$ are defined then they both are, and have corresponding values (via the coercers). Thus a complete subtype is a partial subtype in which the inherited functions are completely, rather than partially, behaviorally compatible.

Several examples will be given in the next section to illustrate how this definition subtypes. The interpretation of a sort (the carrier of the sort) is simply a set of objects (with no associated operations). A type is a generalized order-sorted algebra which consists of the carriers of one or more sorts plus operations acting on (and giving results in) the carriers of the sorts. Thus a type has both sets of elements and operations (consisting of interpretations of the symbols in its order-sorted signature). A type **T** is a subtype (either complete or partial) of a type **U**, if **T** and **U** are embedded in the same generalized order-sorted sorted algebra *A* in such a way that every sort of **U** is a supersort of a sort of **T** and overloaded functions are behaviorally compatible. Thus while types are generalized order-sorted algebras, we can only determine if one type is a subtype of another by looking at a larger generalized order-sorted algebras in which both live. Typically this generalized order-sorted algebra will contain all of the sorts and operations defined in the language (or at least in the particular program under consideration).

Unfortunately there is great confusion in programming languages about the difference between a type and a sort. Most programming languages define types to be what we have referred to here as sorts. This leads to discussions about the types of terms, whereas we would instead refer to the "sorts" of terms. We are not happy with this confusion of terminology, but have adopted the terminology used here since it is consistent with that used by others workers modelling types by algebras. A possible solution to this confusion might be to reserve the name abstract data type (or ADT) for what we have termed types, and use type interchangeably with sort. Since our types are not necessarily very abstract, we have resisted this temptation.

§3 Subtypes and Inheritance in Programming Languages

Our formal definitions of types and subtypes in terms of generalized order-sorted algebras were carefully constructed to describe notions of subtype that arise in real programming languages. In particular they are intended to capture the following three notions of subtype:

(1) Subset: **Int(1..10)** is a "subset" subtype of **Int**.
(2) Isomorphic Copy: **Int** is an "isomorphic embedding" subtype of **Real**.
(3) Object-oriented: **Student** is an "object-oriented" subtype of **Person**.

Let us examine each of these situations carefully to see how they fit into our definitions. We begin with "subset" subtypes.

since + then could be interpreted in the subtype in such a way that behavioral compatibility is preserved. If this new sort is added as a subsort of Int, the result will be a complete subtype of **Int**. Other methods of creating such a complete subtype by adding a second "+" to the subtype are suggested by the example given after the definition of complete subtype in section 2.3.

§3.2 "Isomorphic Embedding" Subtypes

Historically, systems of numbers were extended in order to make partial operations more defined. The natural numbers were extended successively to the integers, rationals, reals, and complex numbers, in order to obtain closure under operations such as subtraction, non-zero division, limits, and the taking of roots. In each of these cases the previous set of numbers can be isomorphically embedded in the original. In fact for most purposes we simply assume this set is contained (rather than isomorphically embedded) in the successor. It is not surprising then, that a totally defined operation, such as addition over the integers, will be behaviorally compatible with the corresponding operation over one of its supersorts, such as the reals. In this case the natural partial subtypes will in fact turn out to be complete subtypes. Of course, if the operation is originally defined only in the supersort (e.g., logarithm over the reals), then the corresponding function with domain and range restricted to the subsort (e.g., the integers) is likely to result only in a partial subtype. In this case, to get a complete subsort, we would typically let the range of the function in the subsort be the original range of the function in the supersort. For example, we would let the version of logarithm defined on the integers have as range the set of reals.

The argument for the Integers being a subtype of the reals is similar to that of **Int(1..10)** for **Int**. In this case the coercer $c_{\text{Integer, Real}}$ simply maps each integer to the corresponding real number. To make the example more interesting (although less natural), let us take type **Real** with signature <{R, C} : + : <<R, C>, C> > and the type **Int** with signature <{Int}: + : << Int, Int>, Int> > as above, both interpreted in the natural generalized order-sorted algebra, B , whose signature is the union of those given and where Int ≤ R ≤ C. In B , B_{Int} is the set of integers, B_R is the set of reals, and B_C is the set of complex numbers. Then the natural isomorphic embeddings of the integers into the reals, and of the reals into the complex numbers, are the coercers for this algebra. It is a generalized order-sorted algebra since the coercers preserve the operation +. Since Integer ≤ Real and Integer ≤ Complex, (1) of the definition of complete subtype is satisfied. Since + is total on **Int** and is consistent with the definition on **Real**, (2) is satisfied, so **Int** will be a complete subtype of **Real**. It is worth noting here that **Int**, the behavioral compatibility (or at least partial behavioral compatibility) was preserved by the coercers mapping subsorts to supersorts. The fact that these coercers did not have to be injective was crucial for the object-oriented case. Also in the object-oriented case we saw by an example that the ranges of corresponding functions in subtypes may remain the same as in the supertype or may be relativized to the sorts of the subtype, depending on the kind of operator. In spite of these variations, the definitions proposed in §2 captured all of these notions of subtype.

§4 Overloading, Ambiguity, and the Interpretation of Terms

So far we have discussed the modelling of types and subtypes in generalized order-sorted algebras. We now wish to take this one step further and examine the problem of interpreting first-order terms which represent elements of the types. In traditional programming languages there are few difficulties associated with this. However in the presence of inheritance, especially multiple inheritance, problems arise due to the presence of overloaded operators. In this section we discuss problems which may arise in interpreting terms in generalized order-sorted algebras, and show that under certain syntactic conditions, terms may be interpreted uniquely.

§4.1 Definition and Sort-Checking of Terms

Since we are working in a strongly-typed environment, each term of the language can be assigned a sort. In traditional languages, each term typically can be assigned to a unique sort. In the presence of inheritance this is no longer possible. Instead each term may be assigned many sorts. E.g. the constant "3" can be assigned the sorts integer, real, complex, etc. The following defines both the legal terms and the possible sorts of these terms by defining "sort"-checking rules:

Definition: Let $<\Sigma, \leq>$ be an order-sorted signature with S the collection of sorts in Σ. Let $\Sigma_{w,s}$ be the collection of operations in Σ with signature $<w, s>$. We let λ denote the empty tuple in S^*. Thus $c \in \Sigma_{\lambda,s}$ denotes a constant of sort s. We next define a proof system which will allow us to infer which expressions formed from symbols of Σ form terms which can be assigned sorts.

(A1) $<\Sigma, \leq> \vdash c : s$ if $c \in \Sigma_{\lambda,s}$.

Definition: An order-sorted signature $<\Sigma, \leq>$ is **regular** if whenever $w_0 \leq w_1$ and $f : <w_1, s_1>$, there is a least $<w,s>$ such that $w_0 \leq w$ and $f : <w, s>$.

Thus if f is applied to an element d of type w_0 then this minimal f may be applied. The following theorem is from [Goguen and Meseguer 1986].

Theorem: If $<\Sigma, \leq>$ is a regular order-sorted signature and $t \in T$, then there is a least sort s such that $t \in T_s$.

The proof is by a straightforward induction on the complexity of terms.

§4.3 Interpretation of Terms

We are now ready to define the interpretation of terms in a generalized order-sorted algebra. Since we are working with overloaded operators, we must be concerned with ambiguities of interpretation of terms. We will show that under the condition of regularity, if $t \in T_s$ then all possible ways of determining the meaning of that term which correspond to it being assigned to sort s will result in the same element. More generally, we wish to show that if a term has two comparable "sortings" then the meanings associated with those sortings will be consistent (in the sense that the meaning corresponding to the smaller sort can be coerced to the meaning in the greater sort). We begin by defining the meaning of a term relative to the proof that it has a particular sort. We will then prove that the meaning is dependent only on the target sort and not on the particular proof that the term has that sort.

Definition: Let $t \in T$ and P be a proof that t has sort s. We define the meaning of t in order-sorted algebra A with respect to proof P, $[[t]]_{A,P}$, by induction on the length of P:

(1) Suppose the last step of the proof P is the axiom (A1). Then $t \in \Sigma_{\lambda,s}$ and define $[[t]]_{A,P} = A_t$ where A_t is the interpretation of the t with signature $<\lambda,s>$.

(2) Suppose the last step of the proof P is the rule (R1). Then $<\Sigma, \leq> \vdash t : s'$ for some $s' \leq s$, by a proof P' of length less than that of P. Then $[[t]]_{A,P} = c_{s,s'}([[t]]_{A,P'})$ where $c_{s,s'}$ is the coercer in A from A_s to $A_{s'}$.

(3) Suppose the last step of the proof P is the rule (R2), and thus t is of the form $f(t_1,..., t_n)$. Then $<\Sigma, \leq> \vdash t_i : s_i$ for $1 \leq i \leq n$ via proofs P_i, each of whose lengths is less than that of P, and $f \in \Sigma_{w,s}$. Then $[[t]]_{A,P} = A_f([[t_1]]_{A,P1},..., [[t_n]]_{A,Pn})$ where A_f is the interpretation of the f with signature $<w,s>$.

have demonstrated that this ordering of carriers based on (not necessarily injective) coercion operators can be used to model the notions of complete and partial subtype, especially as used in object-oriented languages. We have also proved that under the assumption of regularity, a term constructed from over-loaded operators using inheritance does in fact have consistently defined meanings, no matter in which legal sort the term is interpreted. A comparison of this paper with earlier work is given below.

[Goguen 1978] introduced order-sorted algebras as a way of handling errors and overloaded operators. In that original paper, the ordering of sorts was represented by (injective) coercion operators, the partial ordering on sorts was a strict lower semilattice, and if an operator f appeared in the signature with typing <w,s> where $w' \leq w$ and $s \leq s'$ then f also appeared with typing <w',s'>. [Goguen and Meseguer 1986] redefine order-sorted algebras as given in §2.2 of this paper, eliminating coercion operators. They introduce the notion of regular signatures, which are used to show that each term has a least type and to show that initial algebras exist. The main focus of their paper is to show how to handle errors using subsorts and supersorts, while supporting the inheritance of operators. A comparison of order-sorted algebras and our generalized order-sorted algebras is given at the end of §2.2. The main differences in the generalized order-sorted algebras lie in requiring only partial behavioral compatibility and allowing non-injective coercion functions rather than simply taking set inclusion as an interpretation of subsort.

[Futatsugi, Goguen, Jouannaud, & Meseguer 1985] discuss the functional programming language OBJ2, which is built on the theoretical foundation of order-sorted algebras. In that paper they discuss three methods ("using", protecting", and "extending") for new modules to import existing modules, but this notion does not seem to be directly comparable to our notion of subtype. The paper [Goguen and Meseguer 1986a] describes the language FOOPS, which uses the notion of subsort described in [Goguen and Meseguer 1986]. In that paper $\leq$ also expresses an ordering on classes (types), with a hint that the definition of $\leq$ on classes is similar to that on sorts (via "reflection").

[Reynolds 1980] examines the use of implicit conversions and generic (overloaded) operators in programming language from a category-theoretic point of view. He argues that implicit conversions should behave as homomorphisms with respect to generic operators. E.g., if $c : \text{Int} \rightarrow \text{Real}$ is to be an implicit coercer from integers to reals, then $c(x +_{\text{Int}} y) = c(x) +_{\text{Real}} c(y)$ for x and y integers. Pre-ordered categories (called category-ordered algebras) are used to model types in the language as well as to denote syntactic categories. Reynolds presents several examples to buttress the case that "subsorts are not subsets", and chooses to model the subsort relation with coercers powerful facilities for supporting polymorphism. In particular the formal specification of the semantics of such languages will often highlight oversights or complexities in a language not foreseen by the language designers.

Acknowledgements

We would like to thank Jose Meseguer and David Gries for a stimulating discussion of our earlier paper [Bruce and Wegner 1986], which contributed to the reshaping of this paper in terms of order-sorted algebras. We also thank Giuseppe Longo for several interesting discussions.

Orderings and Types in Databases*

Atsushi Ohori

Department of Computer and Information Science/D2
University of Pennsylvania
Philadelphia, PA 19104-6389

Abstract

This paper investigates a method to represent database objects as typed expressions in programming languages. A simple typed language supporting non-flat records, higher-order relations, and natural join expressions is defined. A denotational semantics of this language is then presented. Expressions are interpreted into a domain containing Smyth's powerdomain. In order to give semantics to types, a new model of types, a filter model is proposed. Types are then interpreted as filters in a domain. The type inference system of the language is shown to be sound in this model.

1 Introduction

There are a number of attempts to generalize the relational data model beyond first-nomal-form relations [FT83,OY85,RKS84]; there are also other data models that can be seen as generalizations of the relational data model [AB84,BK85]. The motivation of this study is to draw out the connection between these "higher-order" relations and data types in programming languages so that we can develop a strongly typed programming language in which these data structures are directly available as typed expressions.

We regard database objects as *descriptions* of real-world objects. Such descriptions are *ordered* by how well they describe real-world objects. Relations are then regarded as sets of descriptions describing sets of real-world objects. In [BO87], it is shown that *natural join* can be characterized as the least upper bound operation in Smyth's powerdomain of descriptions. Based on this result, we present a simple typed language that supports non-flat records, higher-order relations, and natural join expressions. We then present a denotational semantics of this language.

Expressions of the language are interpreted in a domain containing Smyth's powerdomain. In order to give semantics to types, we propose a filter model of types. We regard types as sets of values having common structures. In a domain of descriptions, such sets have properties that they are upward closed and they are closed under finite greatest lower bounds. We therefore interpret types as filters in a semantic domain and show the semantic soundness of the type system. The filter model is particularly suitable for types of partial objects. This model can also give precise semantics to *multiple inheritance* studied by Cardelli [Card84].

The rest of this paper is organized as follows. In section 2 we introduce non-flat records to represent database objects and define their ordering. We then introduce types of records and define their ordering. In section 3 we extend expressions, types, and their orderings to sets to represent higher-order relations. We then show that natural join expressions can be generalized in typed higher-order relations. In section 4, we give a formal definition of our language. In section 5, we

*This work was supported by grants from AT&T, the Army Research Office, and by the National Science Foundation(IRI 86-10617). The author was also supported in part by Oki Electric Industry Co., JAPAN

and

$$e_2 = (Emp\# \rightarrow 1234, Age \rightarrow 21)$$

then

$$e_1 \sqcup e_2 = (Name \rightarrow 'J.\ Doe', Emp\# \rightarrow 1234, Age \rightarrow 21)$$

However, $(Name \rightarrow 'J.\ Doe', Emp\# \rightarrow 1234) \sqcup (Name \rightarrow 'K.\ Smith')$ does not exist. As we shall see in the next section, natural join operation can be regarded as the lub operation extended to a powerdomain. This lub operation is also known as the *unification* in *unification-based* grammatical formalisms, where data are descriptions of linguistic entities (see [Shie85] for a survey).

Next we define types for these expressions. Since each primitive set of values corresponds to a basic type and each label denotes certain set of values, types for expressions are defined as:

1. For each primitive set of values B_i there is a constant type τ_i.
2. $(l_1 : \sigma_1, \ldots, l_n : \sigma_n)$ is a type if $\sigma_1, \ldots, \sigma_n$ are types and $l_1, \ldots, l_n \in L$, where $l_1, \ldots, l_n$ are all distinct.

These types can be regarded as specifications of structures of database objects. Since database objects are partial descriptions, these types should specify partial structures. A value is regarded as having a type if the value has the partial structure specified by the type. This observation leads us to define the following typing rules syntactically similar to the type system proposed by Cardelli [Card84]:

1. $b : \tau_i$ if $b \in B_i$.
2. $null_{B_i} : \tau_i$.
3. $(l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n) : (l_1 : \sigma_1, \ldots, l_m : \sigma_m)$ if $m \leq n$ and for all $1 \leq i \leq m, e_i : \sigma_i$.

The following is an example of typing:

$$(Name \rightarrow 'J.\ Doe', Emp\# \rightarrow 1234) : (Name : string, Emp\# : int)$$

From the definitions of typing and $\sqsubseteq$ we can show by simple structural induction that:

Theorem 1 *If $e : \sigma$ and $e \sqsubseteq e'$ then $e' : \sigma$.*

Indeed the following typing is also valid:

$$(Name \rightarrow 'J.\ Doe', Emp\# \rightarrow 1234, Age \rightarrow: 21) : (Name : string, Emp\# : int)$$

In our type system, types therefore correspond to upward closed sets of values. Intuitively, this corresponds to the fact that if a database object has certain structure then any better defined objects also have the structure. For example, if a database object has an attribute *Name* with the type *string*, then we expect that all better defined objects also have this structure.

Now if we regard types as sets of values then the above typing rules induce an inclusion ordering on types. We define a syntactic relation $\preceq$ on types to represent this ordering:

1. $\sigma \preceq \sigma$.
2. $(l_1 : \sigma_1, \ldots, l_n : \sigma_n) \preceq (l_1 : \sigma'_1, \ldots, l_m : \sigma'_m)$ if $m \leq n$ and for all $1 \leq i \leq m, \sigma_i \preceq \sigma'_i$.

Since individual expressions correspond to partial descriptions, sets of expressions correspond to sets of partial descriptions and presumably describe sets of real-world objects. We therefore want to treat these sets of descriptions as descriptions of sets of objects and to order them by their goodness of descriptions. If our primary interest in database programming is query processing or information retrieval from given set of data, then an appropriate ordering is:

$$A \sqsubseteq_0 B \text{ iff } \forall b \in B \exists a \in A. a \sqsubseteq b$$

known as Smyth's powerdomain ordering. Intuitively, this is an ordering on sets of descriptions which "over-describe" real-world sets; a set contains enough descriptions to describe all objects in a real-world set but may contain irrelevant descriptions. $A \sqsubseteq_0 B$ means that B is a less ambiguous and better defined description to a real-world set. A query processing can then be regarded as a process which takes a set of descriptions D and return another set of descriptions A such that $D \sqsubseteq_0 A$. Indeed natural join and selection, the two major operations for query processing, have the property that they carry relations higher in this ordering. It should be noted, however, that this ordering is not appropriate for the ordering on databases themselves. If our interests are operations on databases such as database merging then we need other orderings. In [BO87] various properties of orderings on database sets, including this ordering were studied.

For arbitrary sets, however, $\sqsubseteq_0$ is not a partial ordering; it is a pre-ordering and a partial ordering is derived by taking equivalence classes. Define $A \simeq B$ as $A \sqsubseteq_0 B$ and $B \sqsubseteq_0 A$. If $A \simeq B$ then we regard A and B as having same amount of information. We use this equivalence relation as equality between sets of descriptions and regard a set of descriptions as a representative of the corresponding equivalence class. Then $\sqsubseteq_0$ becomes a partial ordering. Thus we now regard equivalence classes of sets of expressions as descriptions of sets of objects and extend expressions to these equivalence classes. We also extend the ordering $\sqsubseteq$ on expressions to these equivalence classes, i.e. if $[A]$ and $[B]$ are equivalence classes of sets of expressions A and B then $[A] \sqsubseteq [B]$ if $A \sqsubseteq_0 B$.

For $\simeq$ we have [Smyt78]:

Theorem 4 *$A \simeq \overline{A}$ and $A \simeq B$ iff $\overline{A} = \overline{B}$, where $\overline{A} = \{e | \exists a \in A. a \sqsubseteq e\}$.*

If we restrict attentions to finite sets, then this theorem says that a set A is equivalent to the co-chain of the set of minimal elements in A, where a co-chain is a set such that no member in the set is greater than any other member in the set. Thus we can use co-chains as canonical representatives of equivalence classes. Intuitive justification for this equivalence is that if an object x is in an answer to a query then we know that any better defined object y such that $x \sqsubseteq y$ also satisfies the query. Thus all better defined objects are redundant and can be eliminated from the answer.

We have seen that sets of expressions can be also regarded as descriptions and the approximation ordering $\sqsubseteq$ on expressions can be extended to sets of expressions. We can then include sets of expressions in our language and allow records to contain these sets as values. Since now sets are regarded as expressions ordered by $\sqsubseteq$, by applying the same argument, we can further extend our language to allow sets of sets of expressions as expressions. Indeed we can carry this extension process to any depth.

In the syntax of the language this extension can be done by simply adding the rule:

4. $\{e_1, \ldots, e_k\}$ is an expression if $e_1, \ldots, e_k$ are expressions.

where we allow the empty set $\{\}$ as an expression, since the empty set can be regarded as a valid response to a query. We call these expressions as *set expressions*. Set expressions are regarded as representatives of corresponding equivalence classes. The extended language not only allows

It is easy to check that this typing rule yields an upward closed set in set expressions under our ordering on sets and the theorem 1 also holds.

This typing rule also induces an inclusion ordering on set types regarded as sets of values (i.e. sets of set expressions). In order to represent this ordering, we first define the following pre-ordering on set types:

$$\sigma \preceq_0 \sigma' \text{ iff } \forall \iota \in \sigma \exists \iota' \in \sigma'. \iota \preceq \iota'$$

As before a partial ordering is obtained by defining equivalence relation $\simeq$ as $\sigma \simeq \sigma'$ iff $\sigma \preceq_0 \sigma'$ and $\sigma' \preceq_0 \sigma$. Then by the definition of typing, $\sigma \simeq \sigma'$ iff for any e, $e : \sigma \Leftrightarrow e : \sigma'$. Therefore this equivalence relation exactly corresponds to the equality between types regarded as sets of values. We therefore regard set types as representatives of equivalence classes.

Parallel to theorem 4, we can show:

Theorem 7 *$\sigma \simeq \underline{\sigma}$ and $\sigma \simeq \sigma'$ iff $\underline{\sigma} = \underline{\sigma'}$, where $\underline{\sigma} = \{\iota | \exists \iota' \in \sigma. \iota \preceq \iota'\}$.*

Therefore set types can be also represented by co-chains.

Note that the definition of $\preceq_0$ is the inverse of the definition of $\sqsubseteq_0$ and the extended ordering $\preceq$ still corresponds to the generality of specifications. If we replace $\sigma \preceq \sigma'$ with $\sigma' \sqsubseteq \sigma$ then we get the same definitions and properties for orderings on expressions and types.

We now extend the ordering relation $\preceq$ on types to set types using the partial ordering $\preceq_0$ on equivalence classes of sets of types. It can then shown that theorem 2 still holds for the extended types. We write $\sigma \wedge \sigma'$ for $\sigma \sqcap \sigma'$ if σ, σ' are set types. From the duality of $\sqsubseteq$ and $\preceq$, we can see that $\sigma \wedge \sigma'$ always exists if σ, σ' are set types.

The following theorem connects $\bowtie$ and $\wedge$:

Theorem 8 *If A, B are set expressions with $A : \sigma_1, B : \sigma_2$ then $A \bowtie B : \sigma_1 \wedge \sigma_2$.*

Proof. Let $a \sqcup b$ be any element in $A \bowtie B$. Since $A : \sigma_1$ and $B : \sigma_2$, there are $\iota_1 \in \sigma_1$ and $\iota_2 \in \sigma_2$ such that $a : \iota_1$ and $b : \iota_2$. Then by theorem 3, $a \sqcup b : \iota_1 \sqcap \iota_2$. But by definition $\iota_1 \sqcap \iota_2 \in \sigma_1 \wedge \sigma_2$. This shows $A \bowtie B : \sigma_1 \wedge \sigma_2$.□

This theorem shows that we have successfully generalized natural join in typed higher-order relations. Figure 3 is an example of a natural join of typed higher-order relations.

4 Definition of the Language

In this section we give formal definition of our language supporting records, higher-order relations, and natural joins.

4.1 Expressions

We use $l, l_1, \ldots$ for elements of L. The syntax of expressions is given by the following abstract syntax grammar:

$$\begin{array}{rcl} e & ::= & b \ (b \in B_i) \mid null_{B_i} \mid \\ & & (l_1 \to e_1, \ldots, l_n \to e_n) \mid (\ldots, l \to e, \ldots).l \mid \\ & & \{e_1, \ldots, e_m\} \mid \{e_1, \ldots, e_n\} \bowtie \{e'_1, \ldots, e'_m\}. \end{array}$$

4.2 Types

We assume that there are constant types $\tau_1, \ldots, \tau_n$ associated with $B_1, \ldots, B_n$. Then the syntax of types for expressions is defined by the following abstract syntax grammar:

$$\begin{aligned}\sigma \; ::= \;& \tau_i | \\ & (l_1 : \sigma_1, \ldots, l_n : \sigma_n) | \\ & \{\sigma_1, \ldots, \sigma_m\} | \\ & \sigma \wedge \sigma' \quad (\text{if } \sigma, \sigma' \text{ are of the form } \{\sigma_1, \ldots, \sigma_n\}).\end{aligned}$$

In order to define axioms of equality of types, we first define the syntactic relation $\preceq$ on the sublanguage of types that do not contain meet types (i.e. types of the form $\sigma \wedge \sigma'$):

$$\begin{aligned}\sigma &\preceq \sigma \\ (l_1 : \sigma_1, \ldots, l_n : \sigma_n) &\preceq (l_1 : \sigma'_1, \ldots, l_m : \sigma'_m) \text{ if } m \leq n \text{ and } \sigma_i \preceq \sigma'_i \text{ for } 1 \leq i \leq m \\ \{\sigma_1, \ldots, \sigma_n\} &\preceq \{\sigma'_1, \ldots, \sigma'_m\} \text{ if } \forall \sigma \in \{\sigma_1, \ldots, \sigma_n\}. \exists \sigma' \in \{\sigma'_1, \ldots, \sigma'_m\}. \sigma \preceq \sigma'\end{aligned}$$

Axiom for set types is then defined as:

$$\{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_n\} = \{\sigma_1, \sigma_3, \ldots, \sigma_n\} \text{ if } \sigma_2 \preceq \sigma_1 \tag{4}$$

This equation makes $\preceq$ a partial ordering. Let $\sqcap$ be the greatest lower bound of this partial ordering. The axiom for meet types is then defined as:

$$\{\sigma_1, \ldots, \sigma_n\} \wedge \{\sigma'_1, \ldots, \sigma'_m\} = \{\sigma_i \sqcap \sigma'_j | 1 \leq i \leq n, 1 \leq i \leq m, \sigma_i \sqcap \sigma'_j \text{ exists}\} \tag{5}$$

4.3 Rules For Type Inference

Not all expressions are meaningful. One goal of a type system is to identify the set of all syntactically meaningful expressions as the set of *well typed* expressions. We write $\vdash e : \sigma$ for e is *well typed* with type σ. Such well typed expressions are systematically inferred by a *type inference system.*

A type inference system consists of axioms for constant types and inference rules for compound types. Axioms for our type system are:

const	$\vdash b : \tau_i$	for all $b \in B_i$
null	$\vdash null_{B_i} : \tau_i$	for all B_i

Inference rules for our type system are:

subtype $$\frac{\vdash e : \sigma \qquad \sigma \preceq \sigma'}{\vdash e : \sigma'}$$

records $$\frac{\vdash e_1 : \sigma_1, \ldots, \vdash e_n : \sigma_n}{\vdash (l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n) : (l_1 : \sigma_1, \ldots, l_n : \sigma_n)}$$

dot $$\frac{\vdash e : (\ldots, l : \sigma, \ldots)}{\vdash e.l : \sigma}$$

where $+$ is the *separated* sum domain constructor, $\mathcal{B}_i = B_i \cup \{\perp_{\mathcal{B}_i}\}$ with ordering $\perp_{\mathcal{B}_i} \sqsubseteq x$ for all $x \in B_i$, and w is used to interpret the *wrong* value. For $\mathcal{P}(\mathcal{D})$ we include $\emptyset$, the empty set.

A solution of the equation (6) can be found in a particular class of complete partial orders (c.p.o.) called a *bounded complete ω-algebraic* c.p.o., or simply *domain*.

A c.p.o. is a partial order $(D, \sqsubseteq)$ satisfying:

1. D has the minimal element $\perp_D$.
2. each directed subset $X \subseteq D$ has a least upper bound $\sqcup X$ where a subset X is directed iff $\forall x, y \in X \exists z \in X.x \sqsubseteq z, y \sqsubseteq z$.

An *isolated* (*finite*) element of a c.p.o. $(D, \sqsubseteq)$ is an element $e \in D$ such that for any directed subset $X \subseteq D$ if $e \sqsubseteq \sqcup X$ then there is $x \in X$ such that $e \sqsubseteq x$. We write D^o for the set of isolated elements of D. A c.p.o. is said to be *ω-algebraic* iff D^o is countable and for all $x \in D$ we have $x = \sqcup\{e | e \in D^o, e \sqsubseteq x\}$. A c.p.o. is said to be *bounded complete* (*consistently complete*) if any bounded subset of D has a least upper bound, where a subset X is bounded if it has an upper bound in D.

Construction of a recursive domain without containing powerdomain can be found in many places such as [MPS86,Bare84,Schm86]. In [Smyt78] Smyth showed that domains are closed under the powerdomain construction based on the pre-ordering $\sqsubseteq_0$ and that a domain equation like (6) can be solved. In what follows we use $\mathcal{D}$ for a domain satisfying (6). We also use injections of component domains $\mathcal{B}_1, \ldots, \mathcal{P}(\mathcal{D})$ into $\mathcal{D}$ implicitly and treat them as if they were actual inclusions.

We use the following notations to represent elements in $\mathcal{D}$.

1. $(l_1 \mapsto d_1, \ldots, l_n \mapsto d_n)$ for the function $f \in (L \to \mathcal{D})$ defined as $f(l) =$ if $l = l_i, 1 \leq i \leq n$ then d_i else $\perp_{\mathcal{D}}$, where we assume that $d_i \neq \perp_{\mathcal{D}}$.
2. $[d_1, \ldots, d_n]$ for the element $d \in \mathcal{P}(\mathcal{D})$ such that $\{d_1, \ldots, d_n\} \in d$, i.e. the equivalence class containing $\{d_1, \ldots, d_n\}$.

It should be noted that the domain $\mathcal{D}$ is equipped with the ordering $\sqsubseteq$. This ordering was originally introduced to model computation. However, if we regard values in $\mathcal{D}$ as descriptions then this ordering corresponds to the approximation ordering on descriptions we discussed in section 2. We therefore believe that the domain $\mathcal{D}$ is an appropriate model of our language.

5.2 Semantics of Expressions

Let *Expr* be the set of expressions. We define a semantics of expressions by the semantic function:

$$\mathcal{E} : Expr \to \mathcal{D}$$

as follows:

$$\begin{array}{rcl}
\mathcal{E}[\![b]\!] & = & b \text{ for all } b \in B_i \\
\mathcal{E}[\![null_{B_i}]\!] & = & \perp_{\mathcal{B}_i} \\
\mathcal{E}[\![(l_1 \to e_1, \ldots, l_n \to e_n)]\!] & = & (l_1 \mapsto \mathcal{E}[\![e_1]\!], \ldots, l_n \mapsto \mathcal{E}[\![e_n]\!]) \\
\mathcal{E}[\![e.l]\!] & = & \text{if } \mathcal{E}[\![e]\!] = (\ldots, l \mapsto d, \ldots) \text{ then } d \text{ else } w \\
\mathcal{E}[\![\{e_1, \ldots, e_m\}]\!] & = & [\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_m]\!]] \\
\mathcal{E}[\![e \bowtie e']\!] & = & \text{if } \mathcal{E}[\![e]\!] \sqcup \mathcal{E}[\![e']\!] \text{ exists then } \mathcal{E}[\![e]\!] \sqcup \mathcal{E}[\![e']\!] \text{ else } w
\end{array}$$

From this definition, we can easily show, by induction on the structures of expressions, the soundness of the ordering relation on expressions:

If filter has a minimal element d them it is a *principal* filter and written as $d\uparrow$. Let $\mathcal{F}(\mathcal{D})$ denote the set of all filters in $\mathcal{D}$ that do not contain w. $\mathcal{F}(\mathcal{D})$ is ordered by set inclusion. Lub and glb are defined as:

1. $F \sqcup F' = \{d | \exists f \in F \exists f' \in F'. f \sqcap f' \sqsubseteq d\}$.
2. $F \sqcap F' = F \cap F'$.

Note that $F \sqcap F'$ dose not necessarily exist.

In order to interpret types in $\mathcal{F}(\mathcal{D})$, we define filter constructors corresponding to type constructors.

1. *Records.*
   Let $F_1, \ldots, F_n$ be filters in $\mathcal{D}$. Define $(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) = \{(l_1 \mapsto f_1, \ldots, l_m \mapsto f_m) | n \leq m, f_i \in F_i, 1 \leq i \leq n\}$.

   Prop. 11 *$(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n)$ is a filter in $(L \rightarrow \mathcal{D})$.*

   Proof. It is clear that $(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n)$ is upward closed. To see that this set is closed under pairwise glb, we note that glb in $(L \rightarrow \mathcal{D})$ is pointwise.□
   From the definition, we have:

   Prop. 12
   $$(l_1 \Rightarrow F_1, \ldots, l_n \Rightarrow F_n) \sqsubseteq (l_1 \Rightarrow F'_1, \ldots, l_m \Rightarrow F'_m) \text{ if } m \leq n, F_i \sqsubseteq F'_i, 1 \leq i \leq m$$

2. *Sets.*
   Let $F_1, \ldots, F_m$ be filters in $\mathcal{D}$. Define $[F_1, \ldots, F_m] = \{[f_1, \ldots, f_k] | \forall f \in \{f_1, \ldots, f_k\} \exists F \in \{F_1, \ldots, F_m\}. f \in F\}$

   Prop. 13 *$[F_1, \ldots, F_m]$ is a filter in $\mathcal{P}(\mathcal{D})$.*

   Proof. It is clear that $[F_1, \ldots, F_m]$ is upward closed. Let $[f_1, \ldots, f_k], [f'_1, \ldots, f'_l] \in [F_1, \ldots, F_m]$. Since $[f_1, \ldots, f_k] \sqcap [f'_1, \ldots, f'_l] = [f_1, \ldots, f_k, f'_1, \ldots, f'_l]$ and $[f_1, \ldots, f_k, f'_1, \ldots, f'_l] \in [F_1, \ldots, F_m]$, $[F_1, \ldots, F_m]$ is closed under pairwise glb.□
   From the definition, we have:

   Prop. 14 *(a) $[F_1, \ldots, F_n] \sqsubseteq [F'_1, \ldots, F'_m]$ if $\forall F \in \{F_1, \ldots, F_n\} \exists F' \in \{F'_1, \ldots, F'_m\}. F \sqsubseteq F'$.*
   (b) $[F_1, F_2, F_3, \ldots, F_n] = [F_1, F_3, \ldots, F_n]$ if $F_2 \sqsubseteq F_1$.
   (c) $[F_1, \ldots, F_n] \sqcap [F'_1, \ldots, F'_m] = [F_i \sqcap F'_j | 1 \leq i \leq n, 1 \leq j \leq m, F_i \sqcap F_j$ exists$]$ where we define $[] = \{\emptyset\}$.

We now give semantics to types by the semantic function $\mathcal{T} : Texp \rightarrow \mathcal{F}(\mathcal{D})$ where $Texp$ is the set of types defined in the previous section:

$$\begin{aligned}
\mathcal{T}[\![\tau_i]\!] &= \mathcal{B}_i \\
\mathcal{T}[\![(l_1 : \sigma_1, \ldots, l_n : \sigma_n)]\!] &= (l_1 \Rightarrow \mathcal{T}[\![\sigma_1]\!], \ldots, l_n \Rightarrow \mathcal{T}[\![\sigma_n]\!]) \\
\mathcal{T}[\![\{\sigma_1, \ldots, \sigma_m\}]\!] &= [\mathcal{T}[\![\sigma_1]\!], \ldots, \mathcal{T}[\![\sigma_m]\!]] \\
\mathcal{T}[\![\sigma \wedge \sigma']\!] &= \mathcal{T}[\![\sigma]\!] \sqcap \mathcal{T}[\![\sigma']\!]
\end{aligned}$$

Acknowledgements

This work is a continuation of [BO86,BO87]. I would like to thank Peter Buneman for discussions and suggestions.

References

[AB84] S. Abiteboul and N. Bidoit. Non First Normal Form Relations to Represent Hierarchically Organized Data. In *Proc. 3rd ACM PODS*, Waterloo, Ontario, Canada, 1984.

[Bare84] H.P. Barendregt. *The Lambda Caluculus*. Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1984. revised edition.

[BK85] F. Bancilhon and S. Khoshafian. *Calculus for Complex Objects*. Technical Report, MCC, Austin, Texas, October 1985.

[BO86] P. Buneman and A. Ohori. A Domain Theoretic Approach to Higher-order Relations. In *International Confenerce on Database Theory, Lecture Notes in Coputer Science 243*, Springer-Verlag, 1986.

[BO87] P. Buneman and A. Ohori. Using Powerdomains to Generalize Relational Databases. 1987. Submitted to Theoretical Computer Science.

[Card84] L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer-Verlag, 1984.

[FT83] P.C. Fischer and S.J. Thomas. Operators for Non-First-Normal-Form Relations. In *Proc. IEEE COMPSAC*, 1983.

[MPS86] D.B. MacQueen, G.D. Plotkin, and Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[Ohor86] A. Ohori. *Denotational Semantics of Relational Databases*. Master's thesis, Department of Computer and Information Science, University of Pennsylvania, 1986.

[OY85] Z. Özsoyoğlu and L. Yuan. A Normal Form for Nested Relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 251–260, Portland, March 1985.

[RKS84] A.M. Roth, H.F. Korth, and A. Silberschatz. *Extended Algebra and Calculus for ¬1NF Relational Databases*. Technical Report TR-84-36, Department of Computer Sciences, The University of Texas at Austin, 1984. revised 1985.

[Schm86] D.A. Schmidt. *Denotational Semantics, A methodology for Language Development*. Allyn and Bacon, 1986.

[Shie85] S.M. Shieber. An Introduction to Unification-Based Approaches to Grammar. In *Proc. 23rd Annual Meeting of the Association for Computational Linguistics*, 1985.

[Smyt78] M.B. Smyth. Power Domains. *Journal of Computer and System Sciences*, 16(1):23–36, 1978.

[Zani84] C. Zaniolo. Database Relation with Null Values. *Journal of Computer and System Sciences*, 28(1):142–166, 1984.

1 Introduction

Recently, a number of papers have been published concerning formal studies of set-theoretic properties of type hierarchies, in the context of databases and knowledge bases. These include work of Atzeni and Parker [7,8,9] on containment, disjointness, and intersection constraints, of Lenzerini [10] on covering and disjointness constraints, of Arisawa and Miura [4] on variations of containment constraints. Other related work was published by Schubert et al. [13,14,15], Attardi and Simi [6], Spyratos and Lecluse [16].

We show how some of these results can be incorporated into a system capable of handling queries of constraints between types. We consider a system that allows the representation of types (i.e., sets of elements of a given universe), and binary containment (*isa*) constraints.[1] An important point is that negation is allowed, as in [8], in two ways:

1. It is possible to represent *complements of types*; for example, we can express the fact that the set of the *students* is a subset of the complement of the set of *professors*. (As an aside, this is equivalent to saying that the sets *students* and *professors* are disjoint).
2. It is possible to express negative statements; for example, we can state that it is not the case that the *instructors* are a subset of the *professors*.

Essentially, we have *positive constraints*, which express containment between types or their negations, and *negative constraints*, which negate containment, and therefore specify nonempty intersection: if the set *instructors* is not a subset of the set *professors*, then the set *instructors* intersects the complement of the set *professors*.

[1]Obviously, any knowledge representation system would provide more general kinds of relationships among types, than just containment and its negation. Here, we consider only the implementation of the subsystem dealing with type containment inference.

1. $I(X) \subseteq I(U)$
2. $I(\mathbf{non}(\mathbf{non}(X))) = I(X)$
3. $I(\mathbf{non}(X)) = I(U) - I(X)$

In other words, with an interpretation, the type term $\mathbf{non}(X)$ denotes the complement under U of the set denoted by X.

From condition 2 above, it follows that for every type term X,

$$I(X) = I(\mathbf{non}(\mathbf{non}(X))) = I(\mathbf{non}(\mathbf{non}(\mathbf{non}(\mathbf{non}(X))))) = \cdots$$

Therefore, it is possible to introduce the notion of *type descriptor* of the type scheme T/U, as an equivalence class of type terms,

$$\{X, \mathbf{non}(\mathbf{non}(X)), \mathbf{non}(\mathbf{non}(\mathbf{non}(\mathbf{non}(X)))), \ldots\},$$

where X is a type term of the form S or $\mathbf{non}(S)$, and S is a type symbol in U; a type descriptor is designated by any element of the class, but usually by X. Therefore, the type scheme $T/U = \{U, T_1, \ldots, T_n\}$ has the type descriptors U, $\mathbf{non}(U)$, T_1, $\mathbf{non}(T_1)$, $\ldots$, T_n, $\mathbf{non}(T_n)$.

The interpretation I is *trivial* if $I(U) = \emptyset$, and so $I(X) = \emptyset$, for each type term X.

A *positive (binary) constraint* p has the form $p : X$ *isa* Y, where X and Y are type descriptors. The constraint p is *satisfied* by the interpretation I if $I(X) \subseteq I(Y)$. A *negative constraint* has the form $\mathbf{not}(p)$, where p is a positive constraint. It is satisfied if p is not.

Note that the positive constraint p is satisfied by I if and only if $I(\mathbf{non}(X)) \cup I(Y) = I(U)$, or, equivalently, if and only if $I(\mathbf{non}(X)) \cap I(Y) = \emptyset$. Therefore, the negative constraint $\mathbf{not}(p)$ is satisfied if and only if $I(X) \cap I(\mathbf{non}(Y)) \neq \emptyset$. In other words, positive constraints make assertions about *inclusions* between types, while negative constraints make assertions about *intersections* between types. Therefore, in order to improve expressiveness, we will write X *int* $\mathbf{non}(Y)$, instead of $\mathbf{not}(X$ *isa* $Y)$.

inference problem is to tell whether C implies c. Algorithms for the solution of the inference problem (called *inference algorithms*) have correctness proofs that are usually based on sound and complete sets of inference rules. An *inference rule* $C \vdash c$ is a rule asserting that the constraint c holds whenever the set of constraints C holds. For example, the rule

$$X \textit{ isa } Y \text{ , } Y \textit{ isa } Z \vdash X \textit{ isa } Z$$

asserts that the inclusion predicate *isa* is transitive.

Relative to a specific set of inference rules, we write $C \vdash c$ if c can be derived from C using applications of the rules.

The basic requirement for each inference rule is to be *sound*, i.e., that it derive from C only constraints c such that $C \models c$. Moreover, it is important to have sets of inference rules that are *complete*, i.e., that allow the derivation of *all* the constraints c such that $C \models c$. Thus, a set of rules is sound and complete when $\vdash$ is equivalent to $\models$.

Recently, we have shown that the following set of inference rules is sound and complete for containment constraints [8,9]. (X, Y, Z represent arbitrary type descriptors).

INT0. $X \textit{ int } Y \vdash X \textit{ int } X$

INT1. $X \textit{ int } Y \vdash Y \textit{ int } X$

INT2. $X \textit{ int } Y \text{ , } Y \textit{ isa } Z \vdash X \textit{ int } Z$

INC0. $X \textit{ int } \mathrm{non}(X) \vdash Y \textit{ isa } Z$

INC1. $X \textit{ int } \mathrm{non}(X) \vdash Y \textit{ int } Z$

ISA0. $\vdash X \textit{ isa } U$

ISA1. $\vdash X \textit{ isa } X$

ISA2. $X \textit{ isa } Y \text{ , } Y \textit{ isa } Z \vdash X \textit{ isa } Z$

ISA3. $X \textit{ isa } Y \vdash \mathrm{non}(Y) \textit{ isa } \mathrm{non}(X)$

TRIV0. $X \textit{ isa } \mathrm{non}(X) \vdash X \textit{ isa } Y$

number of properties follow from the construction of the graph. An immediate fact is that the black graph is reflexive and the blue graph is symmetric. As a consequence, we can consider *undirected* blue edges, corresponding to each pair of directed edges: the edges (X, Y) and (Y, X) will be replaced by the edge $\{X, Y\}$.

Theorem 1 *Let S be a satisfiable, nontrivial containment scheme. If the type X is not trivial, then C implies X isa Y if and only if there is a black path from the node (corresponding to the type) X to the node Y.*

Proof. Follows directly from Fact 1 and the definition of the graph: the presence of the "dual" edge for each *isa* in C is the counterpart to the double possibility $((Z_{i-1} \mathit{isa} Z_i) \in C$ or $(Z_i\ \mathit{isa}\ Z_{i-1}) \in C$) provided by Fact 1. $\square$

Theorem 2 *Let S be a satisfiable, nontrivial containment scheme. The type X is trivial if and only if there is a black path from the node X to the node* $\mathbf{non}(X)$

Sketch of the proof. It can be easily shown that X is trivial if and only if X *isa* $\mathbf{non}(X)$ can be derived by means of the inference rules. Therefore it suffices to show that X *isa* $\mathbf{non}(X)$ can be derived if and only if there is a black path from X to $\mathbf{non}(X)$. The *if* part is easy, so we concentrate on the *only if* part. If X *isa* $\mathbf{non}(X)$ can be derived without making use of rule TRIV0, then we can reason as in the proof of the previous theorem. Otherwise, proceeding by induction, we can show that the derivation always involve a constraint Y *isa* $\mathbf{non}(Y)$ (with a shorter derivation; so the path from Y to $\mathbf{non}(Y)$ is in the graph) such that X *isa* Y can be derived without making use of TRIV0. Therefore, by the previous theorem, the graph contains a path from X to Y and (due to the duality in the construction of the graph) a path from $\mathbf{non}(Y)$ to $\mathbf{non}(X)$, and so the graph contains a path from X to $\mathbf{non}(X)$. $\square$

bit-parallel machines. This approach was followed by Aït-Kaci et al. [2,3] for positive binary containment.

In database theory, these models are called *Armstrong* models [5]. Most of the classes of constraints considered in database theory admit an Armstrong model for any set of constraints. The situation is different here, since negation of constraints is allowed, as opposed to what happens in database theory.

Fact 3 *The existence of Armstrong models is not guaranteed for containment schemes.*

Proof. Let S be a containment scheme and c a constraint such that neither c nor its negation $\mathbf{not}(c)$ are implied by S. Then an Armstrong model should violate both, and this is clearly impossible. $\square$

However, it is still possible to follow the idea, by using two models (or even just interpretations), one for the positive constraints and one for the negative ones. To be more precise, let us consider a satisfiable, nontrivial containment scheme $S = (T/U, C)$, such that $C = P \cup N$, where P is a set of positive constraints and N is a set of negative constraints; also, let C^+ be the set of containment constraints implied by C (the closure of C), and P', N' be the positive and negative constraints, respectively, in C^+. Then, our goal is to have two interpretations, $I_{P'}$ and $I_{N'}$, such that the positive constraints satisfied by $I_{P'}$ are exactly those in P' and the negative constraints satisfied by $I_{N'}$ are exactly those in N'.

Note that $I_{P'}$ and $I_{N'}$ need not be models: $I_{P'}$ ($I_{N'}$) could satisfy all the *isa* (*int*) and violate some of the intersection (*isa*) constraints. In fact, it follows from the inference rules in Section 3 that the intersection constraints do not influence the positive constraints: for any consistent scheme with constraints $C = P \cup N$, the positive constraints in C^+ are exactly those implied by P. Therefore, an interpretation satisfying exactly the positive constraints in P would be a perfectly suitable $I_{P'}$.

In order to explain gradually how $I_{P'}$ can be built, let us first consider a simpler

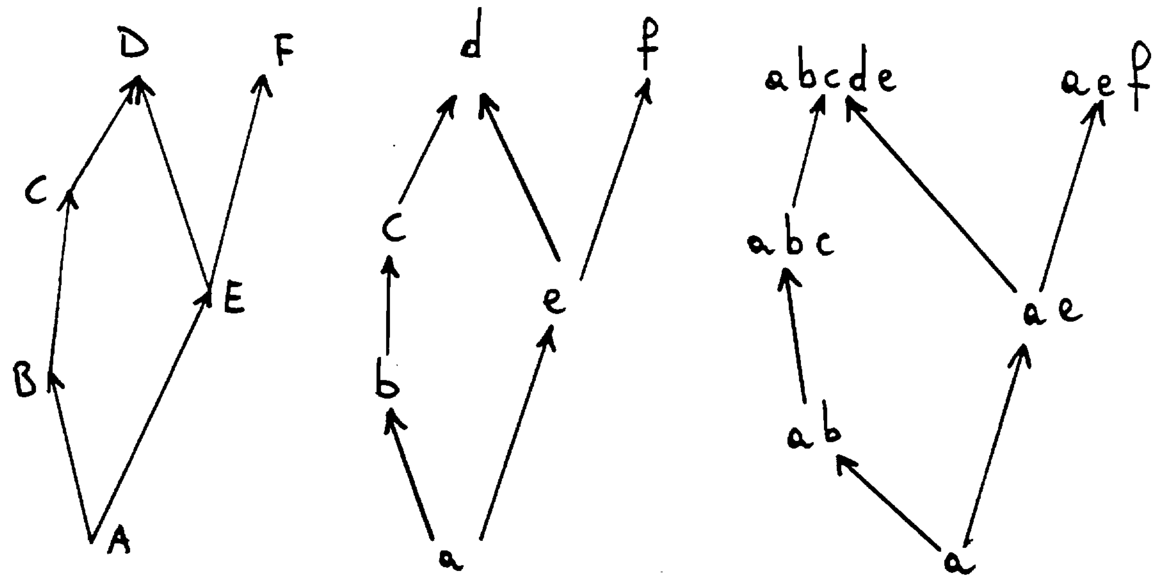


Figure 1:

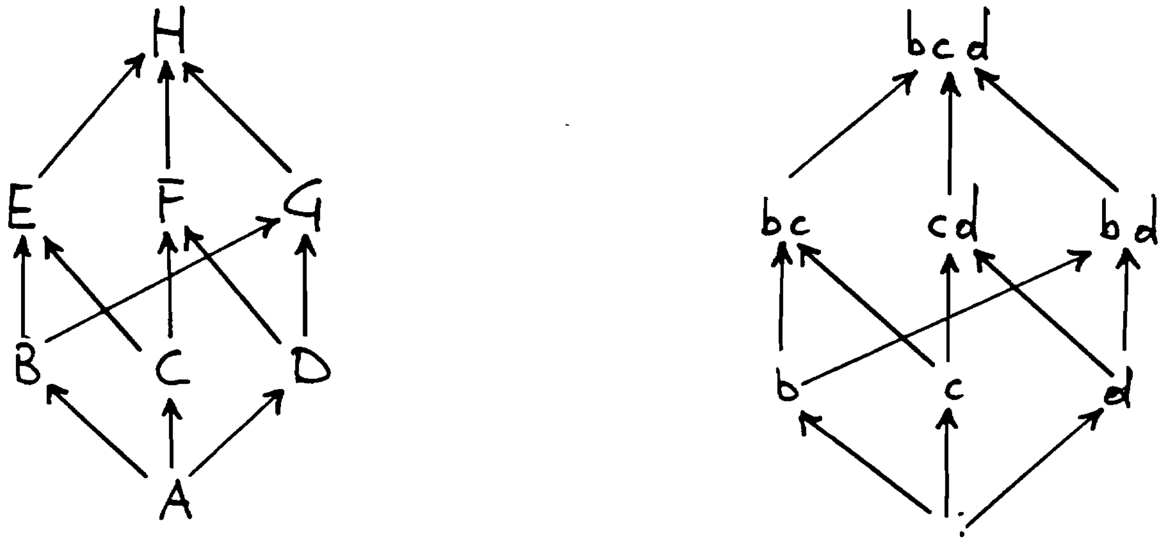


Figure 2:

$$\begin{tikzcd}
 & & U & & \\
\mathit{non}\,(Y) \arrow[urr] & & & & \mathit{non}\,(X) \arrow[ull] \\
 & W \arrow[uur] & & \mathit{non}\,(W) \arrow[uul] & \\
X \arrow[uu] & & & & Y \arrow[uu] \\
 & & \mathit{non}\,(U) \arrow[ull] \arrow[uul] \arrow[uur] \arrow[urr] & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & u & & \\
y' \arrow[urr] & & & & x' \arrow[ull] \\
 & w \arrow[uur] & & w' \arrow[uul] & \\
x \arrow[uu] & & & & y \arrow[uu] \\
 & & \cdot \arrow[ull] \arrow[uul] \arrow[uur] \arrow[urr] & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & & u\ x x'\ y y'\ w w' & & \\
x y' \arrow[urr] & & & & x' y \arrow[ull] \\
 & w \arrow[uur] & & w' \arrow[uul] & \\
x \arrow[uu] & & & & y \arrow[uu] \\
 & & \cdot \arrow[ull] \arrow[uul] \arrow[uur] \arrow[urr] & &
\end{tikzcd}$$

Figure 3:

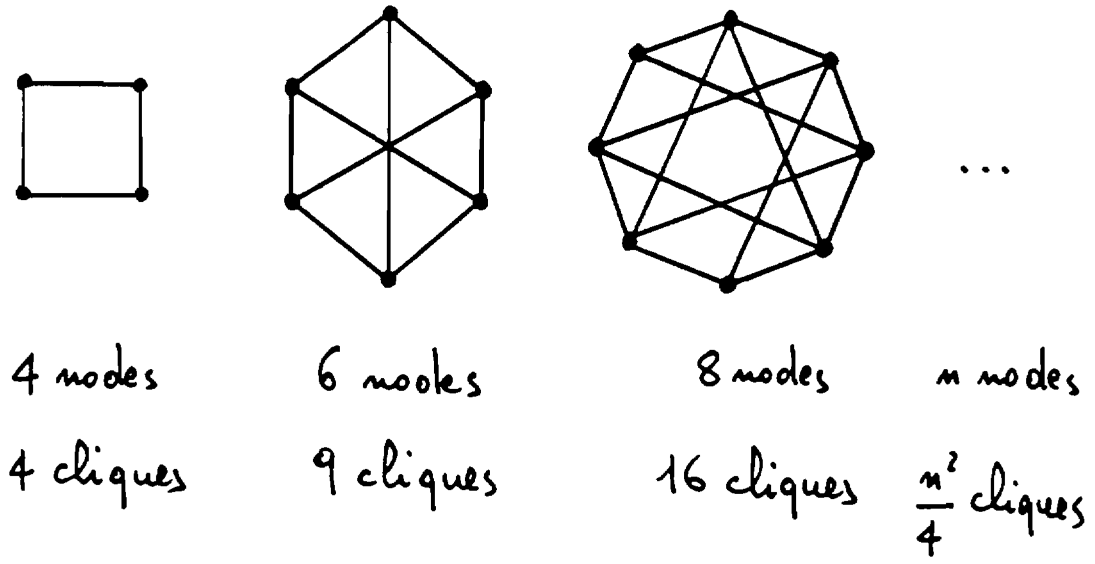

4 nodes
4 cliques
6 nodes
9 cliques
8 nodes
16 cliques
n nodes
$\frac{n^2}{4}$ cliques
...

Figure 5:

Sharing, Persistence, and Object Orientation: a database perspective

Setrag Khoshafian
Patrick Valduriez

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759-6509

Abstract

This paper defines the notions of sharing and persistence, in an object-oriented framework from a database perspective. It illustrates the concurrent and referential aspects of sharing, and demonstrates the variations in the degree of persistence. Guided by the lessons learned from the design and ongoing implementation of a structurally and operationally object-oriented database management system (DBMS), the paper also shows the representation and propagation of persistence and sharing in the different modules of a DBMS.

1. Introduction

In this paper we clarify different aspects of persistence and sharing in an object-oriented framework from a database perspective. These terms (including object orientation) mean different things to different people. The AI, database, object-oriented and programming languages communities have been using these concepts in a conflicting and sometimes contradictory manner.

Sharing

In a database framework "sharing" relates to synchronizing concurrent accesses to objects to ensure the consistency of information stored in the database. The database is accessed and updated through *transactions*, where a transaction is a program which is either executed entirely or not executed at all (i.e. transactions are *atomic)*. Serializability of transactions is required [Eswaran et al. 1976, Papadimitriou 1979], and is typically achieved through locking. Shared locks on an object allow multiple "readers" to access it, whereas an exclusive lock grants access to only one user ("writer"). Objects which are accessed concurrently by multiple users (transactions) will henceforth be called *concurrently shared objects.*

(ii) *System Failures:* usually caused by software errors in the operating system or the DBMS or by hardware failure other than the disk media.

(iii) *Media Failures:* usually caused by hard disk crashes.

There exists a fundamental relationship between sharing and persistence in databases. Transaction updates of the database must persist. But since the persistent database is concurrently accessed (i.e., it is shared), we must serialize the execution of the transactions. Recovery techniques typically require the use of logs [Gray 1978]. These logs record before and after images of updated objects. If a transaction must be aborted due to conflicts, its effects are *undone* using the log. The log is also used for system and media recovery. Another technique to achieve high resilience is through data replication as in the Tandem transaction processing systems [Borr 1981]. Some attempts have been made to extend programming languages such as PS-Algol, to provide support of concurrent transactions and deal with transaction failures [Krablin 1985].

Most commercially available DBMS's attempt to deal with all three types of failures. In fact, the recovery manager and the exception handler represent a substantial part of the DBMS code. For this reason the programming language perspective on "persistence" in terms of the data being maintained on secondary storage after a program terminates, appears as rather naive from a database perspective.

Object-Orientation

Even the notion of object orientation has different connotations and meaning for the different communities.

From a database perspective, object orientation is a rather novel concept being incorporated in recent database data models. Dittrich [1986] has identified three levels of object orientation for DBMS's:

(a) *Structurally object-oriented:* implies the capability of representing arbitrarily structured complex objects.

(b) *Operationally object-oriented:* implies the ability to operate on complex objects in their entirety, through generic complex object operators.

(c) *Behaviorally object-oriented:* implies typing in the object-oriented programming sense (classes), with the specification of the types and operations (messages)

An interesting arena where these differences show up quite clearly is the special purpose machine architectures, microcode implementations, as well as software algorithms and technologies developed for the different paradigms. For AI, the special purpose architectures as well as the software algorithms and technologies tend to be *language-oriented.* Thus, LISP machines attempt to support symbolic list processing through tagged architectures [Moon 1985]. Other features provided by these LISP machines include runtime type checking, large virtual address spaces and efficient garbage collection [Creeger 1983, Hayashi et al. 1983]. For object-oriented languages, the microcoded implementation of Smalltalk-80 on the Dorado provides interpretation of the language with good performance [Deutsch 1983]. Special purpose architectures for Smalltalk-80 such as Swamp [Lewis et al. 1986], which, among other features, supports Smalltalk contexts directly in hardware, have demonstrated even better performance. These architectures and technologies demonstrate that the main problems being dealt with are primarily processing (CPU) but sometimes primary storage (RAM) bottlenecks.

In contrast the main bottleneck of DBMS's is the I/O [Boral and DeWitt 1983] (i.e., the secondary storage accesses). It should be noted that DBMS applications usually deal with much larger disk-resident persistent databases. An (extreme) case in point is the United Airlines Apollo reservation system based on IBM's Transaction Processing Facility, which uses 135 IBM 3380 disk drives and services 55,000 terminals [Krause 1985]! Hence, many of the proposed Database Machine architectures attempt to alleviate the I/O bottleneck through increasing the I/O bandwidth. Similar to the commercially available Teradata machine architecture [Neches 1985], the de-facto architecture of these machines is a collection of processing units which "share nothing" [Stonebraker 1986] and each of which has its own disk (or I/O subsystem). The persistent data itself is horizontally partitioned (*declustered*) [Livny et al. 1987] across the disks of the processing units. A fast interconnection network provides the inter-processing unit communication. Examples of shared nothing database machine architectures currently investigated by researchers are GAMMA [DeWitt et al. 1986], GRACE [Fushimi et al. 1986] and MBDS [Demurjian et al. 1986]. It is important to emphasize that unlike many AI and object-oriented architectures, the processing elements of these database machines are not special pur-

[Khoshafian and Copeland 1986]. FAD is structurally and operationally object-oriented. In FAD, objects are defined as follows:

Assume we are given a set of attribute names A, a set of identifiers I, and a collection of base atomic types.

An *object o* is a triple (*identifier, type, value*) where: the identifier is in I, the type is in {*atom, set, tuple*} the value is one of the following:

> if the object is of type *atom* then the value is an element of a user defined domain of atoms.
>
> if the object is of type *set* then the value is a set of distinct identifiers from I.
>
> if the object is of type *tuple* then the value is of the form [a1:i1, a2:i2,..., an:in] where the ai's are distinct attribute names, and the ij's are identifiers. ij is the value taken by the object on attribute aj. It is denoted o.aj

An *Object System* is a set of objects. An object system is *consistent* iff (i) no two distinct objects have the same identifiers (unique identifier assumption) and (ii) for each identifier present in the system there is an object with this identifier (no dangling identifier assumption).

The notion of *persistence* is also built in FAD through a **database** root (where "**database**" is a reserved key word of FAD). Every object "reachable" from **database** is persistent. More specifically, objects reachable from **database** are defined recursively as follows:

(i) **database** is reachable from itself.

(ii) if O is a set object reachable from **database**, then so is every o in O.

(iii) if O is a tuple object reachable from **database**, then so is O.a for all a.

Objects which are not reachable from **database** are called *transient*. In order to avoid confusion with other sorts of persistent objects, the conceptual FAD objects which are reachable from the **database** root will be called *recoverable objects*.

Note that since FAD supports object identity, referential sharing of objects is possible. In fact, objects can be shared in either the persistent conceptual object space or the transient object space.

This graphical representation clearly illustrates referential sharing of objects. Note that both persistent and transient objects can have referential object sharing. Needless to say, object identity is the feature which enables this type of object sharing.

Mapping to the Internal Model

In our implementation, transactions expressed in FAD are compiled into the language of the internal layer. Programs in this language manipulate *stored physical* objects which correspond to and are determined by the conceptual objects of FAD. The recoverable objects of the conceptual layer are actually stored as recoverable files, tuples, sets etc. in the internal layer. An important feature of the internal model is the fact that it is "value based". This means objects are identified through "key" attributes, such as the EmployeeNumber in an Employees relation, or DepartmentName in a Departments relation. Furthermore, the internal layer supports the direct representation and storage of complex objects (similar to some implementations of non–first normal form relational models [Deppisch et al. 1986]). Similar to the conceptual model, the objects in the internal model are constructed through sets, tuples and atomic objects. *Surrogates* which are system generated unique identifiers independent of physical addressability or content [Khoshafian and Copeland 1986] are introduced to support the conceptual model's identity. Thus, one possible representation of the conceptual object in Figure 1 is given in Figure 2, where S1, S2, T1, T2, T3, T4 are surrogates.

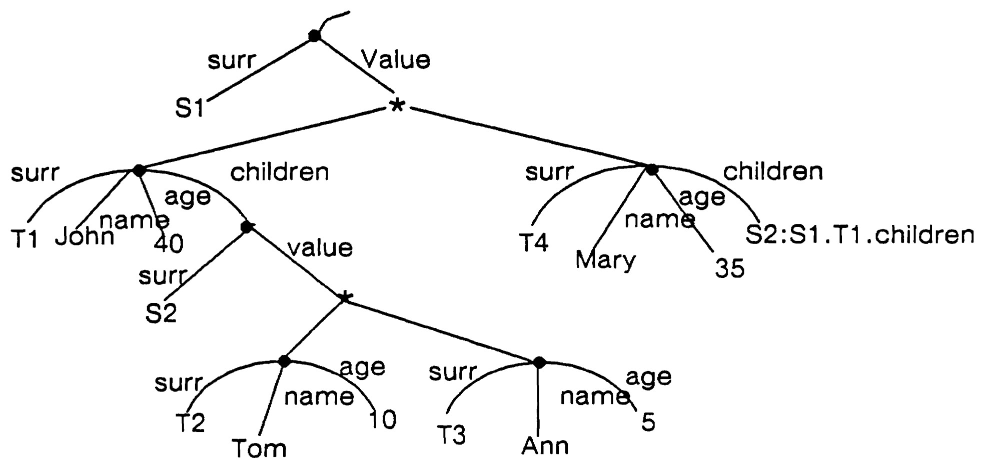


Figure 2: Internal Object Representation

and which persist for the duration of the transaction. The concurrency control/recovery system uses a *shadowing* [Lorie 1977] mechanism for the persistent database. This means, database objects updated by the transaction will be shadowed and maintained (persist) in the workspace of the transaction since the transaction needs to see the effects of its updates in subsequent accesses. The scheme is similar to the one used in Gemstone [Maier et al. 1986]. There are also system structures (i.e., structures generated by the DBMS on behalf of the transaction) which only persist for the duration of the transaction. One example is the data structure which maintains the correspondence between the shadow and persistent pages. Another example is the list of transaction identifiers which could potentially conflict with the currently executing transaction.

As far as persistence is concerned, the database is resilient to transaction, system, and media failures. Transaction failures arise due to conflicting accesses to concurrently shared objects by different transactions. A certification [Kung and Robinson 1981] based concurrency control synchronizer aborts one of the conflicting transactions. The user interacting with the system is informed of the abort and might choose to retry (re-execute) the transaction.

Sessions

A user interacts with transaction management systems in *sessions*. Simply stated, a session refers to the duration that a user is logged into the database management system. During the session a certain environment (expressed in *session variables*) is established and the user submits one or more transactions to the DBMS in this environment

Some examples of session variables are the terminal/window parameters of the user interface, statistics on number of transactions executed and duration of each, as well as trace options set and reset at different points of time during the sessions. These session variables are shared by all the transactions which get executed during the session and persist for the duration of the session.

As for running transactions within a session, three types of interaction schemes are feasible. Below we present these schemes in increasing order of "optimism" in concurrent accesses.

(1) *Checkout–Checkin*: in this scheme, after starting a session, the user submits simple transactions which retrieve "large" objects from the database, *explicitly checking out*

Environment (and hence persist as long as the system is up). We already described examples of the former two. For Execution Environment data structures, perhaps the most important is the Buffer table which contains the page identifiers of all the physical persistent and concurrently shared data base pages buffered by the buffer manager. The disk image describing the free and used pages of the disk and access tables used for determining conflicting accesses are other examples of Execution Environment persistent data structures. These Execution Environment structures are shared by all the sessions which get created during an *activation* of the Execution Environment. These activations correspond to system re-starts.

4. Representation and Storage of Persistent/Shared Objects

We mentioned in the previous section the transaction workspace and the session variables in the run-time execution environment. Parts of these workspaces will be allocated to storing FAD objects in the conceptual model's format. These conceptual FAD objects will be either transient or persistent (i.e., accessible from the **database** root). Another portion of the workspaces will be storing paginated objects in the internal model's format. These internal storage objects in the workspace of a transaction will consist of those objects which reside in pages updated by the transaction.

The DBMS also buffers concurrently shared data pages of the indexed files which store the internal database objects. However, these pages are shared across all currently executing transactions and do not belong in the private workspace of any particular transaction (the private transaction workspace disappears once the transaction terminates).

Objects are referenced through an *identifier* which specifies the table (Persistent or Transient) and a table entry corresponding to an object:

[TableSelector, Index]

Hence, TableSelector is either TRANS or PERS. **Index** is an index to the corresponding object table.

Associated with each of the object tables we have an Object Value Table. This table stores the actual values of objects. The values of string atomic objects are stored as self–describing: **[STRING, ValueByteString]**

The value of a set or tuple is a list of the form:

([AttrName, Identifier])

if the object is a set, the AttrName field is NIL.

Small fixed size atomic objects (like integers) will be stored directly in the Object Tables (vs object Value Tables).

Figure 4 gives a more detailed description of the conceptual transaction object space. We should also emphasize that it is permissible to have persistent objects referenced from the Transient Object Table or the Transient Object Value Table. However, *all* references from the Persistent Object Table or the Persistent Object Value Table must be to entries in these Persistent Tables only.

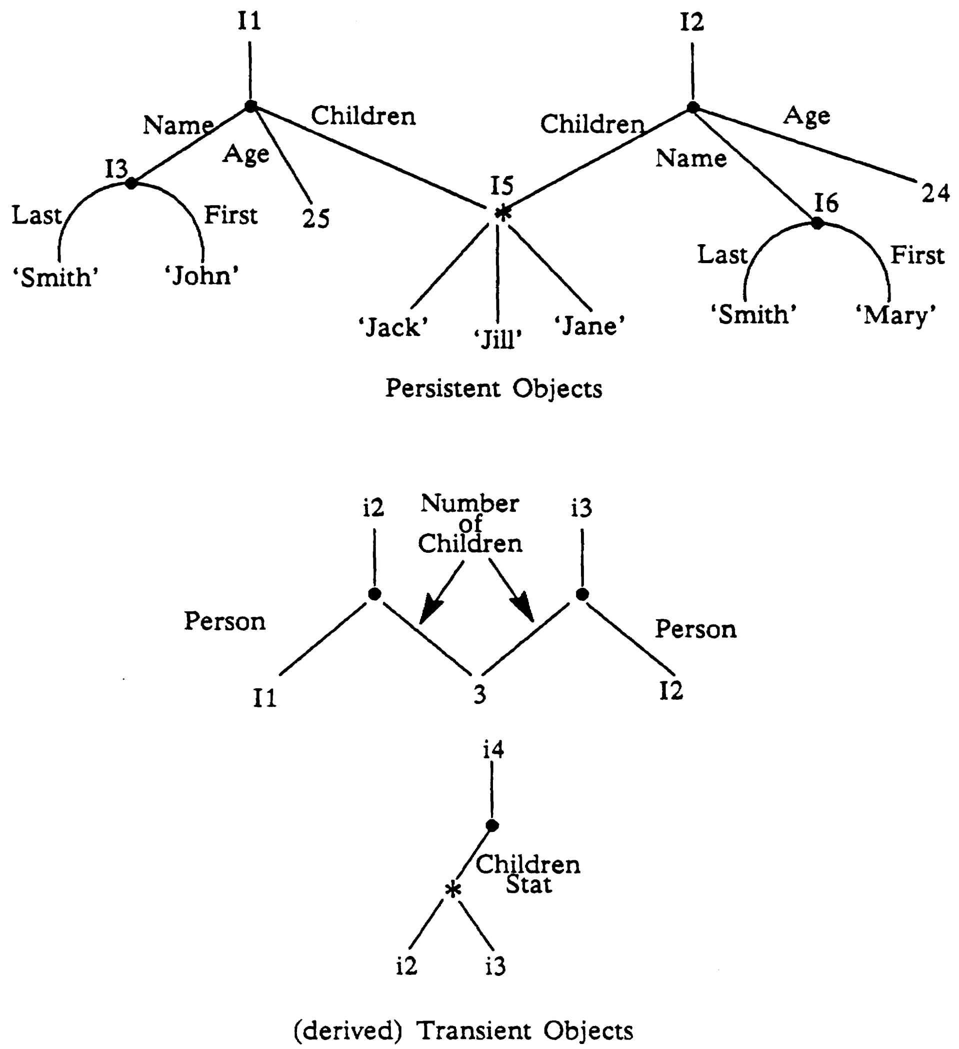


Figure 5: Graphical Representation of the Objects

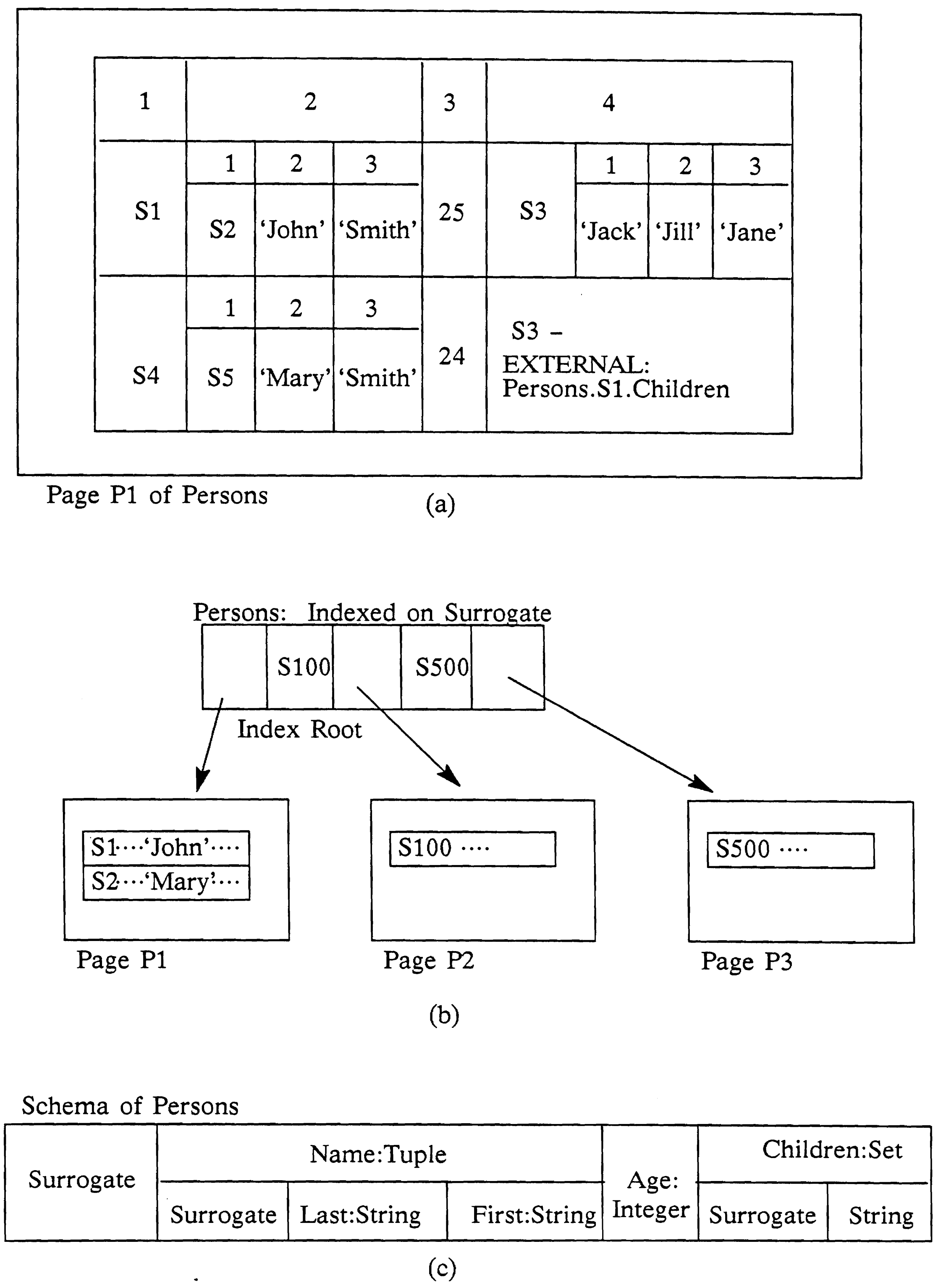


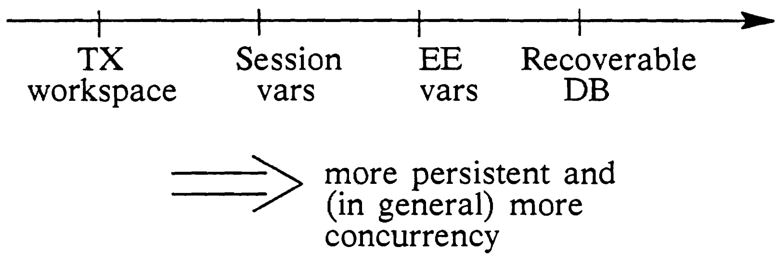
Figure 8: Object Representation in the Internal Layer

copies of Persons with an alternative organization and clustering are also not shown in Figure 8.

5. Summary

In this paper we have attempted to present aspects of sharing and persistence, in an object-oriented framework. The perspective given to these concepts was database-oriented and influenced from the design and implementation of one particular DBMS.

We saw an increasing degree of persistence going from a transaction workspace, to session variables, to the execution environment, and finally to the recoverable persistent database.



As far as concurrently shared objects are concerned, we can similarly characterize objects and data structures shared within a transaction (such as shadowed pages), within the session of a user (such as the session variables), within an execution environment, and finally the most important space, namely the recoverable database. Therefore, similar to persistence, there is an increase in the degree of concurrency in going from transaction workspace to the recoverable database.

We should note that, although in general the recoverable database and the database shared concurrently across multiple transactions are one and the same, there are numerous DBMS's which restrict user accesses to certain subsets of the recoverable database, depending upon access grants [Fernandez et al. 1981]. In other words, it is conceivable to have portions of the recoverable database accessed by, say, only one "special" type of transaction. The bottom line is that recoverable does not necessarily mean concurrently shared. Of course, session and execution environment variables show that concurrent sharing does not automatically imply recoverablity either. However, the concurrently shared variables of these environments *do* persist as long as the environment is operational.

[Atkinson et al. 1983] Atkinson M., Bailey P., Cockshott W., Chisholm K., Morrison R., *"An Approach to Persistent Programming"*, Computer Journal, Vol. 26, No. 4, 1983.

[Bancilhon et al. 1987] Bancilhon F., Briggs T., Khoshafian S., Valduriez P., *"FAD, a Powerful and Simple Database Language"*, MCC report submitted to publication, 1987.

[Bancilhon and Khoshafian 1986] Bancilhon F., Khoshafian S., *"A Calculus for Complex Ojbects"*, Int. Symposium on PODS, March 1986.

[Batory 1985] Batory D.S., *"Modeling the Storage Architectures of Commercial Database Systems"*, ACM Transactions on Database Systems, Volume 10, Number 4, 1985.

[Boral and DeWitt 1983] Boral H., DeWitt D.J., *"Database Machines: an Idea Whose Time has Passed? a Critique of the Future of Database Machines"*, Int. Workshop on DBM, Munich, September 1983.

[Borr 1981] Borr A.J., *"Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing"*, Int. Conf. on VLDB, Cannes, September 1981.

[Cardelli 1984] Cardelli L., *"Amber"*, AT&T Bell Labs Technical Memorandum 11271-840924-10TM, 1984.

[Cardelli and Wagner 1985] Cardelli L. and Wagner P., *"On Understanding Types, Data Abstraction, and Polymorphism"*, ACM Computing Surveys, Vol. 17, No. 4, December 1985.

[Chen 1976] Chen P.P., *"The Entity-Relationship Model -- Toward a Unified View of Data"*, ACM Transactions on Database Systems, Vol. 1, No. 1, 1976.

[Creeger 1983] Creeger M., *"Lisp Machines Come Out of the Lab"*, Computer Design, November 1983.

[Demurjian et al. 1986] Demurjian S., Hsiao D., Menon J., *"A Multi-Backend Database System for Performance Gains, Capacity Growth and Hardware Upgrade"*, Int. Conf. on Data Engineering, Los Angeles, February 1986.

[Deppisch et al. 1986] Deppsich U., Paul, H-b., Schek H-J., *"A Storage System for Complex Object"*, in Proceedings of 1986 Intl. Workshop on Object-Oriented Database Systems, Pacifica Grove, California, September 1986.

[DeWitt et al. 1986] DeWitt D.J. et al., *"GAMMA - a High Performance Dataflow Database Machine"*, Int. Conf. on VLDB, Kyoto, August 1986.

[Deutsch 1983] Deutsch L.P., *"The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture"*, in Smalltalk-80: Bits of History, Words of Advice, ed. Glenn Krasner, Addison Wesley, 1983.

[Dittrich 1986] Dittrich K.R., *"Object-Oriented Database Systems: The Notion and the Issues"*, Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, Ca., September 1986.

[Eswaran et al. 1976] Eswaran K.P., Gray J.N., Lorie R.A., Traiger I.L., *"The Notions of Consistence and Predicate Locks in a Database System"*, Comm. ACM 19(11), 1976.

[Maier et al. 1986] Maier D., Stein J., Ottis A., Purdy A., *"Development of an Object-Oriented DBMS"*, OOPSLA-86, Portland, Oregon, September 1986.

[Mattson et al. 1970] Mattson R., et al., *"Evaluation Techniques for Storage Hierarchies"*, IBM Systems Journal, Vol. 3, No. 2, June 1970.

[McLeod et al. 1983] McLeod D., Narayanaswamy K., Rao Bapa K., *"An Approach to Information Management for CAD/VLSI Applications"*, ACM-SIGMOD Int. Conf., San Jose, May 1983.

[Moon 1985] Moon D.A., *"Architecture of the Symbolics 3600"*, Proceedings of the 12th Annual Int. Symposium on Computer Architecture, 1985.

[Neches 1985] Neches P., *"The Anatomy of a Database Computer System"* COMPCON 85, San Francisco, CA, February 1985.

[Papadimitriou 1979] Papadimitriou C.H., *"Serializability of Concurrent Database Updates"*, Journal of the ACM 26(4), 1979.

[Rentsch 1982] Rentsch T., *"Object-Oriented Programming"*, SIGPLAN Notices, September 1982.

[Selinger et al. 1979] Selinger et al., *"Access Path Selection in a Relational Database Management System"*, ACM-SIGMOD, Boston, 1979.

[Shipman 1981] Shipman D., *"The Functional Data Model and Data Language DAPLEX"*, ACM Transactions on Database Systems, Vol. 6, No. 1, 1981.

[Stefik and Bobrow 1986] Stefik M., and Bobrow D.G., *·Object-Oriented Programming: Themes and Variations"*, The AI Magazine, 6(4), 1986.

[Stonebraker 1986] Stonebraker M., *"The Case for Shared Nothing"*, Database Engineering, Vol. 9, No. 1, March 1986.

[Stroustrup 1986] Stroustrup B., *"The C++ Programming Language"*, Addison Wesley Publishing Co., Reading, MA, 1986.

[Ungar 1984] Ungar D., *"Generation Salvaging: A Non-disruptive High Performance Storage Reclamation Algorithm"*, ACM Software Eng. Notes/SIGPLAN Notices Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984.

[Valduriez et al. 1986] Valduriez P., Khoshafian S., and Copeland G., *"Implementation Techniques of Complex Objects"*, Proceedings of Very Large Databases: 12th International Conference, Kyoto, Japan, 1986.

Introduction

Names are used for many categories of objects within programming languages - for example, to name constants, variables, points in the program, exceptions etc. When they name fields of records, then it is often the case that some input and output operations could use those names. For example, in a form filling system, or in a browser [Dearle & Brown, 87]. Diagnostic tools and program construction aids need to manipulate, input and output these names.

In operating system command languages, editors and other user interfaces, they are used to identify objects from different sets of categories, especially file directories and files. At present these names may obey different rules from those in the programming language. As we attempt to develop a single coherent system in which long and short term data (code, objects, etc.) are treated consistently [Atkinson *et al.* 81, Atkinson *et al.* 83, Atkinson & Morrison 85b] it has been necessary to consider carefully the treatment of names.

Unfortunately, during the development described in those cited papers there were two flaws in our treatment of names:

i) the interpretation of field names in the type checking rules implied a single universe of names for fields - which is known to be unmanageable in large evolving systems; and
ii) program identifiers were used to name some things (e.g. procedures and structure classes) while strings were used to name other things (notably databases and entries in databases).

The former problem appears in many systems as we note in various surveys [Atkinson & Buneman 88, Buneman & Atkinson 86, Atkinson *et al.* 87]. The latter problem manifests itself in most languages as the use of strings for file names. It has the inconvenience of introducing a quite different, dynamic binding rule for the interpretation of these names. Normally, the operating system is responsible for providing this rule. The inconsistency introduced makes programming more difficult and requires program alteration when programs are moved between operating systems

In PS-algol and its descendents we have wished to encompass more of the semantics that affect the execution of programs to give the programmer a consistent world for the total computation. We have therefore sought to remove these anomolous string-names and their inconsistent interpretation. A similar motivation has influenced other work [Buhr & Zarnke 87, Richardson *et al.* 87]. We envisage that by continuing this development most of the functions of an operating system can be given a consistent semantics which is also consistent with the command languages and the programming languages provided. The task of learning to use the composition of these, and of implementing them is then much simplified.

The operations on names are:

type test,
input and output;
type consistent assignment; and
lexical ordering

Names may be used to index a **struct** or **env** object, and to construct new quadruples to insert into an **env** object.

These operations are further defined below. There are also two transfer functions on names:

let *nameToString* = **proc**[*t* : **type**] (*n* : **name** [*t*] $\rightarrow$ *string*)

and

let *stringToName* = **proc**[*t* : **type**] (*s* : *string* $\rightarrow$ **name** [*t*])

The type test has the form

<exp> **is** <ptype>

and type rule

$\forall$ t: t **is** ptype $\Rightarrow$ bool

where ptype is:

a) any one of the predefined types (e.g. *int, real, bool*);
b) any user defined type name, (i.e. an in scope occurrance of <type_name> from **type**<type_name> **is** ...);
c) any type expression (i.e. such as may appear after **is** in **type** ... **is** ...);
d) any type *constructor* (e.g. **abstype**, which might have been used in **type** *stack* **is** **abstype** ...).

Figure 1 illustrates the use of the type test, is exactly equivalent to

nameToString (<exp$_1$>) < *nameToString*(<exp$_2$>)

We use this ordering when defining iterators.

The Universal Extensible Union Type

In PS-algol we had an extensible union type, **pntr**, and we grew to appreciate its utility; indeed much of the database programming, including the interface to persistent data and data model implementation depended on it [Atkinson *et al.* 87, Cooper *et al.* 87].

We refer to it as a *union* type because it may refer to an instance of *any* structure class. We refer to it as *extensible* as new classes declared after the use of **pntr** are eligible as referends, thus the set of possible referends is increased when each structure class is declared. It was not *universal* as there were types, e.g. **int**, which were excluded from its set.

It was valuable because it allowed a type check to be delayed, because it allowed us to limit the traversal of the type match algorithm, and because it allowed generic code to be written applicable to future types, possibly with the execution taking into account the actual type. It was, however, overused, as no more specific alternative was available when referend types were predetermined. It was also unfortunate as its pronunciation 'pointer' evoked connotations of other languages where such things provide a loop-hole in the type system and even pointer arithmetic. Of course, these horrors do not exist in PS-algol.

In Napier we therefore allow proper constraint of referend type where appropriate in data structures, and we use polymorphism to implement most generic code. But we have retained the valuable properties of **pntr** in a type **any**, but removed an irksome restriction by making it universal.

There are few operations on values of type **any** (only equality, inequality and assignment) thus it is safe. To gain access to other operations on the values it is necessary to project out of the union, just as one projects out of a statically defined union. A delayed type check is needed in both cases. We now make this projection explicit. (The implicit projection from **pntr** was one of the causes of a single name space of field names.) Thus our **any** is similar to Cardelli's **dynamic** [Cardelli & MacQueen, 85, Cardelli 85].

Note **name** [**any**] is the type which includes all possible names.

Polymorphic Input and Output

The output statment **print** in PS-algol [PPRR-12] is already polymorphic, and handles multiple fonts, multiple destinations and its default actions may be replaced by other code. In Napier we retain the essence of this **print** clause but we are revising details [Philbrow *et al.*

Polymorphic Iterations

When a polymorphic procedure is defined this indicates that different applications of the procedure may have parameters of different type, but that for each application the procedure body will be executed with a consistent and constant substitution of the type variables. The polymorphic iterator is defined correspondingly. Each traversal of the iteration may be with a different type substitution, but within each execution of the controlled statement the type substitution is constant and consistent.

There are iterators to perform defined sequences of operations in the language e.g.

for *i* = 1 **to** 10 **do** ...

with the usual semantics and options. Note that *i* is a constant declared here with the scope of this **for** statement.

There is a similar iteration construct, introduced by **for each**, which iterates over compound objects. Each of the compound objects may be considered a map, e.g. a vector of type $*\tau$ is a stored map from *int* to τ. Identifiers may be provided in the iteration statement to range over the sequence of values in the map, and for every type of map the iteration sequence is defined. e.g.

for each *k* $\rightarrow$ *u* **in** *vs* **do** ...

where *vs* is a vector of strings would apply the controlled clause first with *k* set to the lower bound of *vs* and *u* set to the first string, and repeat for increasing index up to the upper bound. Either control variable may be omitted, e.g.

for each *k* **in** *vs* **do** ...

and

for each $\rightarrow$ *u* **in** *vs* **do** ...

Similar arrangements are available for iterating over indexes, with multiple keys having corresponding multiple control variables.

To illustrate the iterator construct more fully suppose that environments have been chosen to represent some entity, and that now a new property is to be recorded for every instance. The programmer/data designer has decided that such transitions are likely, and considered it worth incurring the additional costs of using **envs** rather than static records. The iteration in Figure 4 would then achieve this.

```
...
print  "'n is the field updateable?"
let    constantField = replyAffirmative ( )
print  "'n What is the name of the new integer field?"
let    newName = read [name[int]]
for each →      anEnv in theIndexToEnvs do          ! don't care about the key
                begin                               ! once for each env
                              ! show the user the environment
                envShow (anEnv)
                print "'nWhat is the initial value for ", newName, " ?"
                let initialValue = read [int]
                if constantField then
                              insert newName  = initialValue in anEnv
                else
                              insert newName := initialValue in anEnv
                end                ! of iteration through index
```

Figure 4: An example program fragment to add to a new integer field to all the environments in an index

That example has assumed the existence of a procedure, *envShow*, capable of printing any environment. A simple implementation, utilising polymorphic iteration, is shown in Figure 5.

Figure 6 shows a procedure to copy one element of an environment, then Figure 7 shows how that and polymorphic iteration can be used to construct a back-up copy of any environment.

Figure 8 shows how two environments may be combined using the same facilities, and figure 9 shows how a user controlled directory (environment) editor might be built.

```
let     mergeEnvs = proc (env1, env2 : env)
        begin     !adds to env1 all the bindings in env2
        let duplicates = emptyEnv ( )
        for each [t : type] n : name [t] in env2 do
                if n in env1 then
                    copyOneEntry [t] (env2, duplicates, n)
                else
                    copyOneEntry [t] (env2, env1, n)
        if size duplicates ≠ o do raise nameClashes (duplicates)
        end
```

Figure 8: A procedure to add the contents of one environment to another

```
let     userControlledCopy = proc (e : env  → env)
        begin
        let res = emptyEnv ( )
        for each [t : type] n : name [t] in e do
                begin
                print "'n include", n, "?"
                if replyAffirmative ( )  do
                   copyOneEntry [t] (e, res, n)
                end
        res
end
```

Figure 9: Procedure that allows the user to control the parts of an environment copied

Finally a program to emulate the *ls* shell command (a simple version) as in UNIX™ is shown as figure 10. Note that *nameToString* is used explicitly because otherwise the name would be printed like a name literal expression, e.g.

name[*int*]*fred*

since a language must be able to read its own handwriting.

```
let     listEnv = proc (e : env)
            for each [t :type] n :name [t] in e do
                print nameToString (n)
```

Figure 10: Procedure to list the contents of a name space c.f. ls in UNIX™

```
let allIn = proc [ type t] (rel: *env; names: *names [t])
        for each  → n in names  do
            if not  (n in rel(1)) do
                raise wrongColumn
```

Figure 12: check all the columns names are in the first environment

```
let match = proc [type t] (t1, t2: env; cols: *name [t]  → bool)
        begin
        let equal:= true
        for each  → n in cols do
            equal: = equal and t1(n) = t2(n)
        equal
        end
```

Figure 13: test two tuples for equality

```
Let  merge = proc (t1, t2: env   → env)
        begin
        let newTuple = emptyEnv ( )
        mergeEnvs (newTuple, t1)                    ! all columns from rel1 - see fig 8
        for each [t: type] n: name [t] in t2 do
            if not (n in t1) do
                    copyOneEntry [t] (t2, newTuple, n)            ! see fig 6
        newTuple
        end
```

Figure 14: generate the new tuple from the two that matched

Subproblem (i) is solved using this type system. We consider below whether the solution is adequate. The check (subproblem (ii)) has been programmed - figure 12 - verifying that all the columns appear in each relation. The dynamic specification of this condition is acceptable since the check is inherently dynamic; the relevant properties of the parameters may not be determined until the code which calls *equijoin* is executed. The result type (iii) is statically specified and consequently the third subproblem is avoided.

For the moment we remain unable to define an adequate type system for generic applications, and we overcome the problem by synthesising a specific procedure for each type parameterisation of join when it is needed, and then using the callable compiler to build the operation before applying it. Persistence and the universal extensible union type allow us to memoise this operator construction [Cooper *et al.*87]. It is not clear whether a type system which does better than this is achievable.

Conclusions

The sequence of examples show that scanning directories is now possible, and that other data dependent generic algorithms can be written. The constructs introduced to achieve this - polymorphic name types, type constrained name values, environments and polymorphic iterators - are individually simple to understand and use, they combine well, and they do not result in a loss of type control or incomprehensible computations.

Use of these constructs to build replacement operating system structures will eliminate strings as names. We need to start the bootstrap as a program binds to its environment, and do this by introducing one standard variable *PS* (Persistent Space).

These structures need to be updated to reflect changes in the environment, e.g. addition of new network addresses, new discs etc. It does not appear possible to include that within the language. However we extend the scope of the language there will always be external agents affecting the computation, and consequently a closed universe is impossible, i.e. *deus ex machina* will occur. If we wish to use the same naming system for everything, then we need to expand the type system to contain everything we wish to name. Examples might be machines, devices etc. if they may be explicitly manipulated or selected by the user/programmer. But this makes it difficult to adhere to the principle of data type completeness.

The section on the implementation of a join procedure is included to show that type systems are *still* not adequate for all we would wish to do. We pose the question: "Can we do better than synthesis of code followed by calling the compiler?" for these remaining generic tasks. The advantage of that approach is that more than type checking may be 'statically' determined, i.e. factored out of the operator's iterations.

Acknowledgements

This work was done at the Universities of Glasgow and St. Andrews with support from the British SERC, from the Alvey programme, and from the SETC and URC sections of STC Ltd. The authors also acknowledge the peace and stimulation of walking on Blackford Hill during their many discussions.

Buneman 85 Buneman, O.P. Data types for Database programming in proceedings of the 1st International Workshop on Persistent Object Systems: Date Types and Persistence, Appin, Scotland (Aug. 1985) PPRR-16-85 285 - 98.

Buneman & Atkinson 86 Buneman, O.P. and Atkinson, M.P. - "Inheritance and Persistence in Database Programming Languages", in Proceedings of ACM SIGMOD CONF. '86, Washington, USA, (May 1986).

Cardelli 84 Cardelli, L, "Amber", *Technical Report*, AT&T, Bell Laboratories, Murray Hill, N.J. USA, 1984.

Cooper & Atkinson 87 Cooper, R.L. and Atkinson, M.P. - "Requirements Modelling in a Persistent Object Store", in Proceedings of the 2nd International Workshop on Persistent Object Systems: their Design, Implementation and Use, Appin, Scotland (Aug. 1987) PPRR-44-87*.

Cooper *et al* 86 Cooper, R.L., Atkinson, M.P. and Blott, S.M. - "Using a Persistent Environment to Maintain a Bibliographic Database", PPRR-24-86*.

Cooper *et al* 87 Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane D. - "Constructing Database Systems in a Persistent Environment", in Proceedings of the 13th International Conference on Very Large Data Bases, Brighton, England (Sept 1987), 117-125.

Dearle & Brown 87 Dearle, A. and Brown, A.L. - "Safe Browsing in a Strongly Typed Persistent Environment", PPRR-33-87*.

Philbrow *et al* 87 Philbrow, P., Armour, I., Atkinson, M.P. and, Livingstone J. - "A Device-independent Output Statement", to be submitted to *ACM SIGPLAN Notices*.

PPRR12 "The PS-algol Reference Manual: Fourth Edition", PPRR- 12 - 87*.

Richardson *et al 87* Richardson, J.E., Carey, M.J., DeWitt, D.J. and Schuh, D.T. - "Persistence in Exodus", in Proceedings of the 2nd International Workshop on Persistent Object Systems : their Design, Implementation and Use, Appin Scotland (Aug. 1987) PPRR-44-87*.

*Persistent Programming Research Reports (PPRRs) are available from the Computing Science Departments at the Universities of Glasgow and St. Andrews, Scotland.

Our approach strongly differs from standard object-oriented ones [Goldberg & Robson 83], [Bobrow & al 86], [Cox 86] , [Stroustrup 86] in that we do not only deal with typed data but also with highly structured ones. We use the set and tuple constructors to define arbitrarily complex objects. These objects are grouped into types which define a minimal common structure (like Cardelli's approach [Cardelli 84]) and common behaviour. We want our type system to be as safe as possible without loosing the advantages of late binding. We think it is necessary in the scope of data base applications to have a strongly typed system.
This paper gives a formal set-theoretic semantics for types in our system. It is the basis on which we shall implement the O_2 object oriented data base system [Barbedette & al 87]. Our model differs from that of [Bruce & wegner 86] in that methods are not objects and that the type system is more permissive. On the other hand, the counterpart is that our type system is not totally safe (well-typed expressions can give run-time errors). However, our set theoretic semantics for type structures (resp methods) corresponds to the classical database interpretation as sets of objects (resp sets of functions). The subtyping relationship is interpreted as inclusion of interpretations. Furthermore, our model allows the definition of cyclic types, such as:

Person=<name:String,children:{Person}>

This paper is organized as follows.
Section 2 gives an informal overview of our approach and exposes it through examples. Section 3 gives a definition of objects. Section 4 gives the semantics of types and inheritance relationship. Finally, the notion of database is introduced in Section 5. Section 6 contains some concluding remarks and open problems.

2. Informal Overview

The O_2 system which is currently implemented in the *Altaïr* Group, puts together object-oriented features like inheritance and late binding (i.e , the actual code of a function is determined at run-time) with data base requirements such as complex objects manipulation, efficient retrieval and updates and persistence of data. Some of these goals do not mix well. Associative search using late binding, for example, is more expensive than a procedure call. A way to solve these requirements is to design systems which perform as much static type checking as possible and accepts late binding when needed. In the same way, opposed to languages such as SmallTalk 80 [Goldberg & Robson 83], a database programming language must allow the user to specify access paths to his/her data. This implies the possibility to describe the structure of the data and specify keys. Our system will thus deal with objects (our data) and types (their structural and behavioural description). A well-known problem is the correlation between object-oriented inheritance and the notion of subtyping [Schaffert & al 86], [Sandberg 86], [Bruce 87]. One way to describe subtyping is to say that objects of a subtype can be used in the same way as objects of the super types but are distinct. That is, they accept behavioural properties described in their supertypes without being instances of these supertypes. This is the approach of SmallTalk 80. An other approach associates inclusion of extensions to sub-typing. That is, objects of a subtype also belongs to its supertypes. This approach has the advantage of being natural : an employee is also a person and a mammal. In order to have a well founded system, we need to associate to our object and type definitions a semantics which characterizes the subtyping relationship in terms of inclusion of interpretations (of types). This is the goal of the formal datamodel described in the sequel of the paper.
Let us introduce some of the notions of this model, using examples. Objects are representing our (computer) world. They are made up of an object identifier (a name for the object) and a value. Values can be atomic values (string, integers, reals,...) tuples values and set values.

(ob_1, "this is a string")
(ob_2, 3.14159265359)
(ob_3, 1243)
(ob_4, <name: "Smith", age: 32>)
(ob_5, <name: "Doe", age: 29, salary: 9700)
(ob_6, {12, 13, "john"})

Futhermore, we want to be able to model cyclic objects, that is, objects which are components of themselves. Such objects may seem a little bit strange but are, actually, used very often in practice.

We suppose given:

- A finite set of *domains* $D_1, ..., D_n$, $n \geq 1$ (for example, the set **Z** of all integers is one such domain). We note **D** the union of all domains $D_1, ..., D_n$. We suppose that the domains are pairwise disjoint.
- A countably infinite set **A** of symbols called ***attributes***. Intuitively, the elements of A are names for structure fields as we shall see later.
- A countably infinite set ID of symbols called ***identifiers***. The elements of ID will be used as identifiers for objects.

Let us now define the notion of ***value***.

Definition 1 :

(i) The special symbol ***nil*** is a value, called a ***basic value***.

(ii) Every element v of D is a value, called a ***basic value***.

(iii) Every finite subset of ID is a value, called a ***set-value***. Set-values are denoted in the usual way using brackets.

(iv) Every finite partial function from A into ID is a value, called a ***tuple-value***. We denote by $<a_1 : i_1 , ..., a_p : i_p>$ the partial function t defined on $\{a_1, ..., a_p\}$ such that $t(a_k)=i_k$ for all k.

We denote by *V* the set of all values. □

We can now define the notion of object.

Definition 2 :

(i) An ***object*** is a pair o = (i, v), where i is an element of ID (an identifier) and v is a value.

(ii) O is the set of all objects, that is $O = ID \times V$.

(iii) We define, in an obvious way, the notion of ***basic*** objects, ***set-structured*** objects and ***tuple-structured*** objects.

(iv) If o=(i,v) is an object then ***ident***(o) denotes the identifier i and ***value***(o) denotes the value v.

(v) ***ref*** is a function from O in 2^{ID} defined as follows :
ref(o) = ∅ for all basic objects.
ref(o) = value(o) for all set-structured object x.
ref(o) = $\{i_1, ..., i_n\}$ for all tuple-structured object o such that value(o) = $<a_1 : i_1 , ..., a_n : i_n>$. □

Intuitively, *ref*(o) is the set of all identifiers that are referenced in the value of o.
This "tuple-and-set" construction of objects (generally called "complex objects") is similar to that of [Bancilhon and Khoshafian 87], [Bancilhon & al 87], [Abiteboul & Beeri 87] and specially to that of [Kuper and Vardi 84] where identifiers (called addresses) are also introduced. We can use a graphical representation for objects as follows :

Definition 3 :
If Θ is a set of objects, then the graph ***graph(Θ)*** is defined as follows:

Definition 5 :

(i) ***0-equality*** : two objects o and o' are 0-equal (or ***identical***) iff o=o' (in the sense of mathematical pair equality),

(ii) ***1-equality*** : two objects o and o' are 1-equal (or simply ***equal***) iff value(o) = value(o'),

(iii) ***ω-equality*** : two objects o and o' are ω-equal (or ***value-equal***) iff span-tree(o) = span-tree(o') where span-tree(o) is the tree obtained from o by recursively replacing an identifier i (in a value) by the value of the object identified by i. □

Equality implies value-equality, but the converse is not true since many distinct objects may have the same span tree.
Let us put these definitions to work with a few examples :
$o_1=(i_1, <a:i_3, b:i_4>)$
$o_2=(i_2, <a:i_3, b:i_4>)$
$o_3=(i_3,$ "Fred")
$o_4=(i_4,$ "Mary")
$o_5=(i_5,$ "Fred")
$o_6=(i_6,$ "Mary")
$o_7=(i_7, <a:i_5, b:i_6>)$
We have o_1 equal o_2 because value(o_1)=value(o_2) but not o_1 equal o_7 because the values differ.
However, if we replace the identifiers by the value they identify in o_1 and o_7, we obtain :
for o_1 : <a:"Fred", b:"Mary">
for o_7 : <a:"Fred", b:"Mary">
and so, o_1 and o_7 are ***value-equal.***
We must notice that the span-tree build from an object may be infinite (in the case of cyclic objects). So, this construction cannot be used (directly) as a decision procedure for testing value-equality. For space reasons, we do not detail in this paper the algorithm which will be used in the implementation of our system.

4. Types

A type is an abstraction that allows the user to encapsulate in the same structure data and operations. In our model, the static component of a type is called a type structure. A type structure is a way of classifying objects with respect to their structure. The operations will be called methods.

As we have basic objects, set-structured objects and tuple-structured objects, we define ***basic types, set-structured types*** and ***tuple-structured types.*** More formally, a type name is defined as follows:

Definition 6 :

Bname is a set of names for basic types containing :

(i) the special symbols *Any* and *Nil*,

(ii) a symbol d_i for each domain D_i. We shall note D_i=dom(d_i),

(iii) a symbol 'x for every value x of D.

Cname is a set of names for constructed types which is countably infinite and disjoint with *Bname*.

Tname is the union of *Bname* and *Cname* and it is the set of all names for types. □

In order to define types, we assume that there is a set M whose elements are called methods and which shall play the role of operations on our data structures. For the moment, we can think of the elements of M as uninterpreted symbols. We shall define them in section 3.2.

Definition 7 :

A set Δ of constructed type structures is *consistent* (or is a *schema*) iff

(i) Δ is a finite set,

(ii) *name* is injective on Δ (only one type structure for a given name),

(iii) $\forall$ st $\in \Delta$, *refer*(st) $\cap$ Cnames $\subseteq$ *names*(Δ) (i.e. there is no dangling identifiers). □

Nota Bene :
In a schema, we can identify a ***type name*** of name(Δ) with the corresponding type structure in Δ, and we shall use this convention in the sequel of the paper.
We illustrate the notion of a schema with two examples :
Let Δ be the set consisting of the following type structures :

age = integer,
person = <name : string, age : age>
persons = {person}

Δ is a schema. If we take off the type structure "age" from Δ, it is no longer a schema.
On the other hand, the following set of type structures is also a schema :

person = human
human = person

This set of type structures may be not useful but it is well defined and has an interpretation as we shall see in the next section.

4.1.2. Interpretation

This section deals with the definition of the semantics of the type structure system presented above. It will be given by a particular function which associates subsets of a consistent set of objects to type structure names.

Definition 10 :

Let Δ be a schema and Θ be a consistent subset of the universe of objects O. An ***interpretation*** I of Δ in Θ is a function from Tnames in $2^{\text{ident}(\Theta)}$, satisfying the following properties:

Basic Type Names

1) $I(\text{Nil}) \subseteq \{i \in \text{ident}(\Theta)$ [1] $/ (i, \text{Nil}) \in \Theta\}$

2) $I(d_i) \subseteq \{ id \in \text{ident}(\Theta) / \Theta(id) \in D_i \} \cup I(\text{Nil})$

3) $I('x) \subseteq \{ id \in \text{ident}(\Theta) / \Theta(id) = x \} \cup I(\text{Nil})$

Constructed Type Names

4) if $s = <a_1 : s_1 , \dots , a_n : s_n>$ is in Δ then
$I(s) \subseteq \{ id \in \text{ident}(\Theta) /$ $\Theta(id)$ is a tuple structured value defined (*at least*) on $a_1,\dots,a_n$ and $\Theta(id)(a_k) \in I(s_k)$ for all k$\} \cup I(\text{Nil})$

5) if $s = \{ s' \}$ is in Δ then
$I(s) \subseteq \{ id \in \text{ident}(\Theta) / \Theta(id) \subseteq I(s') \} \cup I(\text{Nil})$

6) if s = t is in Δ then
$I(s) \subseteq I(t)$

Undefined Type Names

7) if s is neither a name of basic type nor a name of the schema Δ, then
$I(s) \subseteq I(\text{Nil})$ □

An interpretation I is *smaller* than an interpretation I' iff
$\forall$ s $\in$ Tname , $I(s) \subseteq I'(s)$ □

[1] Recall that ident(Θ) denotes the set of the identifiers of all objects of Θ and that Θ(id) denotes the (only) value v such that (id, v) is in Θ.

Let s and s' be two type structures of a schema Δ. We say that s is a substructure of s' (denoted by $s \leq_{st} s'$) iff $M(s) \subseteq M(s')$ for all consistent set Θ. □

For example, if Δ consists of the following type structures :

$$\begin{aligned}
s_1 &= <a{:}\text{Integer}>, \\
s_2 &= <a{:}\text{Integer}, b{:}\text{Integer}>, \\
s_3 &= <c{:}s_1>, \\
s_4 &= <c{:}s_2>, \\
s_5 &= \{s_1\}, \\
s_6 &= \{s_2\} \\
s_7 &= <a{:}'1>
\end{aligned}$$

then the following relationships holds among these structures :

$s_2 \leq_{st} s_1$ $\quad$ $s_4 \leq_{st} s_3$

$s_7 \leq_{st} s_1$ $\quad$ $s_6 \leq_{st} s_5$

The first relationship ($s_2 \leq_{st} s_1$) comes from the interpretation of tuple type structures. Let us establish the second one ($s_4 \leq_{st} s_3$). Let id be (the identifier) of an object belonging to $I(s_4)$. We know from the definition that $\Theta(id)(c)$ belongs to $I(s_2)$ and so to $I(s_1)$ because we have $s_2 \leq_{st} s_1$. We conclude that id belongs to $I(s_3)$ and so $I(s_4) \subseteq I(s_3)$. The inequality $s_6 \leq_{st} s_5$ can be established in the same manner and the relation $s_7 \leq_{st} s_1$ is obviously true.

Definition 12 gives a semantic definition for the subtyping relationship $\leq_{st}$. The following theorem gives a syntactic characterization of it.

Theorem 1 :

Let s and s' be two type structures of a schema Δ. s is a substructure of s' ($s \leq_{st} s'$) iff

(i) either:
s and s' are tuple structures names, s = t, s' = t' such that t is more defined than t' and for every attribute "a" such that t' is defined, we have $t(a) \leq_{st} t'(a)$.

(ii) or:
s and s' are set structures names, $s = \{s_1\}$, $s' = \{s'_1\}$ and $s_1 \leq_{st} s'_1$.

(iii) or:
s = 'x, s' is a basic type structure and x is in dom(s'). □

Proof :
The validity of this characterization can be easily established by induction. The completeness can be established with a case study, inspecting successively tuple structured types, set structured types and basic types. □

This theorem can lead to a syntactical check of testing type structure inequality.

4.2. Methods

In Section 3.1, we have presented the syntax and semantics of type structures. In this subsection, we define, in the same way, the syntax and semantics of operations, which we call methods in this context. These operations will consist of (first order) functions.

4.2.1. Definition

We assume that we have a countable set Mnames of symbols that will be used as names for methods.

Definition 13 :

$M(\text{boolean}) = \{i_0, i_{14}, i_{15}, i_{16}\}$

The model of the signature σ_1 is the set of all **partial** functions from

$$\{i_0, i_5, i_6\} \times \{i_0, i_1, i_2, i_3, i_4\} \text{ in } \{i_0, i_{14}, i_{15}, i_{16}\}.$$

Intuitively, the model of the signature σ_1 is the set of functions assigning a boolean object to some pairs (i,j) where i is (the identifier of) a set of persons object and j is (the identifier of) a person object.
We shall use this interpretation of signatures in the following subsection which introduces an ordering among signatures.

4.2.3. Partial order among signatures.

Definition 16 :
Let Δ be a schema and f and g two signatures over Δ. We say that f is smaller than g (or that *f refines g*) iff $M(f) \subseteq M(g)$ for all consistent set Δ. This ordering will be denoted by $\leq_m$. □

Looking at the schema of the previous example , we can see that the following inequalities hold:
$\sigma_2 \leq_m \sigma_1$ and $\sigma_4 \leq_m \sigma_3$
Indeed, let Θ be any consistent set of objects and f be a partial function in $M(\sigma_2)$. f is a (partial) function from M(employees) $\times$ M(employee) in M(Boolean). We have seen in subsection 3.1.3 that employees $\leq_{st}$ persons and employee $\leq_{st}$ person, and hence, M(employees) $\subseteq$ M(persons) and M(employee) $\subseteq$ M(person). So f is also a partial function from M(persons) $\times$ M(person) in M(boolean), so f is in $M(\sigma_1)$.
A similar proof can be constructed for the inequality $\sigma_4 \leq_m \sigma_3$.

Intuitively, $\sigma \leq_m \sigma'$ means that we can use a method of signature σ' "in place of" a method of signature σ. In the example above, we can apply a method of signature
σ_1 = persons $\times$ person $\rightarrow$ boolean
to a set of employees and an employee because, employees are persons. This partial order models inheritance of methods, just as the ordering $\leq_{st}$ models inheritance of data structures. In the following section, we put data structures and methods together to define type systems and we use the ordering $\leq_{st}$ and $\leq_m$ to define inheritance of types. The following theorem gives an easy syntactical equivalence to the definition of the partial order $\leq_m$ among signatures.

Theorem 2
Let f and g be two signatures over a schema Δ. Then, $f \leq_m g$ iff :
$f = s_1 \times \ldots \times s_n \rightarrow s$
and $g = s'_1 \times \ldots \times s'_n \rightarrow s'$
and $s_k \leq_{st} s'_k$ for k=1,2,...,n
and $s \leq_{st} s'$. □

Proof :
In order to clarify the proof, we assume, without loss of generality, that the methods signatures are of the form:

$$\sigma = s_1 \rightarrow s, \text{ and } \sigma' = s'_1 \rightarrow s'.$$

Suppose that $\sigma \leq_m \sigma'$. Every partial function from $M(s_1)$ to M(s) is then a partial function from $M(s'_1)$ to M(s'). So, we necessarily have $M(s_1) \subseteq M(s'_1)$ and $M(s) \subseteq M(s')$.
Conversely, if these two inclusions hold, then every partial function from $M(s_1)$ to M(s) is clearly also a partial function from $M(s'_1)$ to M(s'). □

Definition 19 :

An *object* o is a triple (i, v, m) where i and v are as in Definition 2 and m is a set of methods. The first component of the signature of every method of m is a type structure whose interpretation contains o. □

This notion is classical in object-oriented approaches. An object is characterized by the methods which can be applied to it. We do not need to be aware of its structure to use it. The set of methods of an object can be empty (in this case, it will be manipulated through the methods of the type it possesses). This is a very useful tool to handle exceptions. For example, let us assume that we define an "employee" type which contains a generic method to compute the salary of an employee. Suppose that one of these employees is the CEO and that his salary has to be computed in different way than for regular employees. One could create a specific subtype of employee in order to override the "increase salary" method of type employee. This would be heavy and it is more natural to define a specific method for the CEO object.

5. Databases

In this section, we introduce the notion of database. Informally, a database is a type system together with a consistent set of objects representing the instances of the types at a given moment.

Definition 20 :

A database is a tuple $(\Pi, \Theta, <_{db}, \text{ext}, \text{impl})$ where

(i) Π is a type system, and Δ is the associated schema,

(ii) Θ is a consistent set of objects,

(iii) $<_{db}$ is a strict partial order among Π,

(iv) ext is an interpretation of Δ in Θ.

(v) impl is a function assigning a function to every method m of a type t.

Moreover, we impose that the following properties hold:

(1) $t <_{db} t'$ implies $t \leq t'$.

(2) If $t <_{db} t'$ and $t <_{db} t''$ then t' and t'' are comparable.

(3) $\Theta = \bigcup_{t \in \Pi} \text{ext}(t)$.

(4) $\text{ext}(t) \cap \text{ext}(t') = \emptyset$ if t and t' are not comparable.

(5) If t is a type of Π and m a method of t having signature $t \times \ldots \times s_n \rightarrow s$, then impl(m) is a function defined at least from $\text{ext}(t) \times \ldots \times \text{ext}(s_n)$ in ext(s). □

This definition deserves some comments. The extension of a type is an interpretation but may not be a model. Indeed, all the possible objects of the type may not be present in the data base. The ordering of definition 18 ($\leq$) models the notion of subtyping. That is two types t and t' are comparable using $\leq$ if one *can be* a subtype of the other. The ordering $<_{db}$ is the actual inheritance types hierarchy, as *defined* by the user. This ordering must satisfy property (1), that is, the user can declare that t is a subtype of t' ($t <_{db} t'$) only if it is allowed by the model ($t \leq t'$). For example, the type system may contain the types:

Age = (Integer, {+,-}) and

Weight = (Integer, {+,-})

with corresponding signatures for the methods + and -. We have the inequalities (Age $\leq$ Weight) and (Weight $\leq$ Age) but the user does not intend to consider an age as a weight nor a weight as an age, and Age and Weight will be incomparable for $<_{db}$.

Property (2) says that we do not allow multiple inheritance. This a constraint we introduced for the O_2 system because it is still an open problem to decide whether multiple inheritance is a useful modelization tool. In any case, our semantics would still be valid in the context of multiple notion.

(iv) Higher Order Methods: in this model, we made the simplifying assumption that the methods are not objects of the model. So methods can be model as first order functions. It should be interesting to extend the model to treat methods as objects and to allow higher order methods.

Acknowledgements

Most of the ideas presented here were generated with F. Bancilhon. This paper also benefits from the careful reading of S. Abiteboul and our colleagues from *Altaïr*, in particular D. Excoffier. Thanks also go to P. Buneman, A. Borgida and D. DeWitt for the fruitful discussions we had on this model.

7. References

[Abiteboul & Beeri 87]
"On the power of languages for the manipulation of complex objects", S. Abiteboul, C. Beeri, *in Int. Workshop on theory and applications of nested relations and complex objects, Darmstadt, 87*

[Albano & al 85]
"GALILEO: A strongly typed, interactive conceptual language", A. Albano, L. Cardelli and R. Orsini, *ACM TODS, Vol 10 No. 2, March 85.*

[Bancilhon and Khoshafian 86]
"A Calculus for Complex Objects", F. Bancilhon, S. Khoshafian, *ACM Conference on Principles of Database Systems, 86*

[Bancilhon & al 87]
"FAD, a Powerful and Simple Database Language", F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, *13th Conference on Very Large Data Bases, Brighton, England, 87.*

[Barbedette & al 87]
"The O_2 Programming Environment", G. Barbedette, C. Lécluse, P. Richard, F. Velez. *ltaïr internal report, Sept, 87.*

[Bobrow & al 86]
"CommonLoops:Merging Lisp and Object-Oriented Programming", D. G. Bobrow et al. *OOPSLA 86, Portland, Oregon, Sept 86.*

[Bruce & Wegner 86]
"An Algebraic Model of Subtypes in Object-Oriented Languages", K., B. Bruce, P. Wegner. *SIGPLAN notices V21 #40, October 86.*

[Bruce 87]
"An Algebraic Model of Subtype and Inheritance", K. Bruce, *Proc. of the workshop on Database Programming Languages, Roscoff, Sept 87.*

[Cardelli 84]
"A Semantics of Multiple Inheritance", L. Cardelli, *in Semantics of Data Types, Lecture notes in Computer Science, Vol 173 pp. 51-67, Springer Verlag, 84*

[Cox 86]
"Object-Oriented Programming, An Evolutionary Approach", B. J. Cox, *Addison Wesley 10393.*

Can Objects Change Type? Can Type Objects Change? (Extended Abstract)

Stanley B. Zdonik
Brown University
Department of Computer Science

Abstract

Types provide a powerful system structuring capability that has been shown to be useful in large-scale software development. They also introduce a set of definitions that can be difficult to change as the system evolves. This is particularly true for persistent object systems in which an object has a very long lifetime. The need to support evolution at the level of types seems to be required for many new application classes, for example those that address the process of design (e.g., software engineering environments).

This paper looks at two aspects of change with respect to types. It first considers the problem of how to allow a particular object to change its type over its lifetime, and then it considers the problem of allowing the type hierarchy itself to shift. For the first problem, we look at rules for how changes to an object's type can occur and how we might solve some of the potential problems introduced by aliasing and type checking. For the second problem we outline what we would like to achieve and sketch two possible solutions.

1 Introduction

One of the major challenges in the engineering of large-scale software systems is to provide mechanisms that allow for evolution and change. This change can take many forms. Some aspects of this problem have been addressed by modern programming languages and environments. For example, data abstraction makes it easier to change the implementation of a module without having to make changes to the modules that use it.

Database systems represent an environment in which these problems are particularly severe. After all, databases are concerned with data that will survive for a very long time. This data may have been created with one set of assumptions, and as the system develops, those assumptions will shift. How can the database system provide support for this inevitable process?

Some proposals [Li86] have advocated the use of prototypes. In these schemes, there is no notion of type. Therefore, it appears that these systems are more able to change. We feel that the flexibility offered by the prototype approach has one very serious drawback for use in database systems. That is, that database systems need to rely on the uniformity that is imposed by a typed universe in order to achieve high performance. For example, the fact that all employees have an employee number (defined by the type Employee) that allows us to compute an index on this attribute.

We are, therefore, interested in trying to balance these two views. We wish to retain a notion of typing while,at the same time, allowing for more flexibility in system (except in fairy tales). In order to capture this kind of knowledge, we need a simple mechanism for expressing these constraints at the type level.

A simple observation can help to explain this phenomenon. Some types cannot be lost. If an object is created as an instance of type Student, that instance might change to type Professor, but both Student and Professor share a common supertype, type Person. Any instance that has type Person as a supertype, can never loose that type. We say that type Person is an *essential type* . That is it defines the essence of its subtypes and must always be present in the list of types of an individual that was created as an instance of it.

Designating a type as an essential type is an activity that would be done by the type designer. It adds some extra semantics about the potential behavior of instances of that type. It builds a simple constraint into the information provided by the type definitions. It is similar to constraints related to object uniqueness (i.e., keys) and referential integrity.

It is possible to have more than one essential type. If our type system allows for multiple inheritance, we might have several essential types contributed by different paths in the lattice. For example, type Car might be a subtype of both Movable_Object and Sellable_Object. Moreover, both of these types might be essential. That is, it might be possible to make a car into a truck by modifying the body, but it must always remain movable and sellable.

2.1.2 Exclusionary types

In a similar way, we might also designate a type T as *exclusionary* if an object can only acquire T at the time of its creation. T is called exclusionary because in moving an object x from some type R to some other type S, it is illegal to move through a type that would have T as a supertype. We are therefore excluded from it as a new type.

Notice that essential types are not exclusionary. It is possible to add a type that is essential to an object. Of course, once it has been added, it cannot be lost. Moreover, an exclusionary type is not essential because it can be removed. Of course, the definition of an exclusionary type requires that once it has been removed, it can never be regained.

Often an object must change types in some predefined sequence. For example, a person starts out as a child, becomes a student, graduates and becomes a professor, and then retires. It is possible to use exclusionary types to simulate this requirement. Suppose that the hierarchy in the following figure is used to model this situation.

This is similar to a common problem that comes up in database programming languages. The problem concerns the ability to explicitly delete objects. In database systems it is common to have an explicit command that deletes an object (e.g., removes a tuple from a relation). We will call this view the explicit deletion view. In some languages, there is no facility for explicitly deleting objects. Instead, the system reclaims storage for an object (i.e., garbage collection) when there is no longer any reference to that object. We will call this the garbage collection view.

These two views are hard to reconcile in a database programming language. In databases, and in some languages (e.g., Galileo [Al] through its class mechanism) there is always some way to refer to an object. The object can be named through its container (e.g., its class object). In the case of relations, one can always get at a tuple through the relation that contains it. In models like this, since the relation provides a reference that cannot be broken by other means (i.e., reassignment) there is a need for an explicit delete.

In languages for which there is a uniform referencing mechanism, all references can be broken. When there are no more references left, the object is effectively deleted. The garbage collector performs a space optimization by actually reclaiming the inaccessible storage. It has been argued that this kind of approach simplifies programming because programmers do not have to keep track of when an object is referenced by other objects. It is impossible with this approach to get dangling references.

In the explicit delete case, one can place a "tombstone object" in place of the deleted object. This eliminates the problem of dangling pointers because a pointer can never be dereferenced to another real object. It will always produce the original object or that object's tombstone. The remaining problem with the tombstone solution is that all programs that do pointer dereferencing would have to able to handle the case in which the expected object has been deleted out from under a given reference. This complicates application code since expressions as simple as x.p (which should return the object referred to by the p field of x) has to be prepared to handle an exception generated by the object's not being there (i.e., a tombstone is there instead).

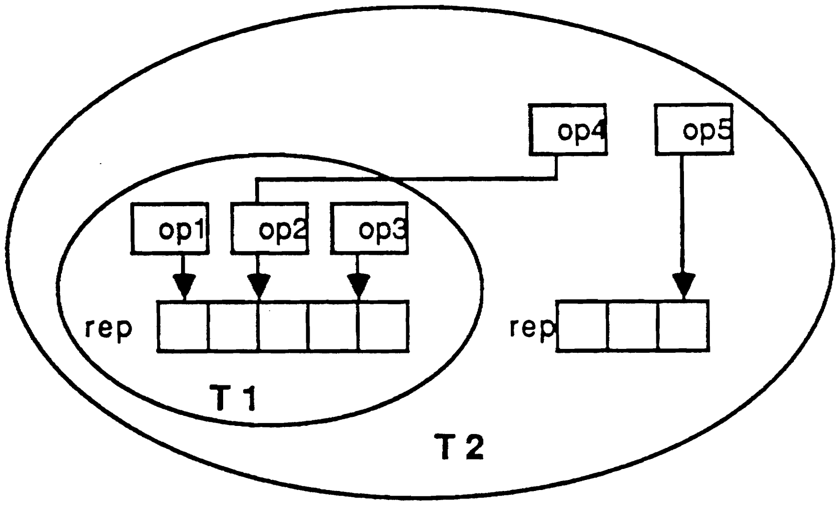
As we saw in a previous section, a similar problem can occur when we delete a type T from an object x. There might still be other objects that are referring to x with the expectation of its having type T.

2.4 The reference-bundle approach

In an object-oriented language, objects can have many types. For a given object, there is a piece of state that represents each of its types. This corresponds to the instance variables defined by each type. We will call each of these fragments of state a *type piece* .

We can think of references as being somewhat more complicated than a single pointer. Since objects are polymorphic, we can think of a reference to an object as consisting of a bundle of pointers, one to each type piece. We will call this kind of reference a *reference bundle* . A reference to a Toyota, then, would consist of a set of references, one to the Vehicle piece, one to the Car piece, and one to the Toyota piece.

Removing a type would correspond to deleting a component of the reference bundle. If we perform the following operation on c, a variable of type Car that holds an instance of the type Toyota:



T1 is the the part of the object defined by the first version of type T. It has a representation which is used to store its state. It also has three operations that are used to access this state. T2 is the part of the object that is defined by the second version of T. It adds some additional state to support its new operations. One of the new operations uses the operation op2 from T1. Op4 might do nothing more than invoke operation op2. In this way, we have op4 = op2 thereby indicating that there is no change in this operation. Op1 and op3 are not available in the T2 version of T.

Whenever a type change occurs, all old instances are, at least conceptually converted to this form. Note, that the actual conversion of storage may be deferred until the object is actually referenced.

When a program uses an object of a particular type it is able to use the appropriate interface in a consistent way. If a program, is expecting an object of type T2, it will use the T2 interface, even if the object was created as an instance of T1. In this approach an object is not an instance of a single type version as in [SZ].

4 Summary

We have suggested a couple of ways in which our notion of type might be relaxed to increase the flexibility of our type systems. We have also looked at a couple of problems that these more flexible models introduce and have sketched some preliminary solutions to these problems. It is clear that there is a need for this kind of capability in many application areas. The challenge is to stretch our notions of type as far as we can in these directions without decreasing our understanding or eliminating the advantages of current type systems.

We need to gain more experience with these models and investigate the feasibility of implementing them in the context of a real object-oriented database programming language. The theoretical ramifications of these proposals deserve further study.

5 References

[ABBHS] M. Ahlsen, A. Bjonerstedt, S. Britts, C. Hulten, L. Soderlund, "Making Type Changes Transparent", University of Stockholm, SYSLAB Report No. 22, February, 1984.

which is unique over all time (at least as far as we can tell). Once an object is created, it persists until it can no longer be accessed from any world.

The details of the possible kinds of states of objects are probably not that important, but I suggest one possibility for concreteness. An object can consist of either an array of bytes, or an array of *slots*. Every slot contains the id of an object. Thus, the object correspond roughly to objects in a language such as Smalltalk, except that I am ignoring classes and subclass relationships. Even this simple object world presents enough interesting features, however.

Though an object is thought of as changing state over time, I consider an object to be a set of *versions*, where each version represents the state of the object at some point in time (in some world). In the normal course of affairs we get a linear sequence of versions, with the most recent one being of primary interest.

A *world* is, on the one hand, a set of objects, and, on the other hand, a mapping from ids to versions. If we are operating "within" a given world, and we modify some object x, then we are really just changing the binding between x's id and its version *for this world only*. Thus, typical computation causes a world to evolve, presumably towards some state that is more useful than its current state.

If every object had a unique world, then worlds would simply partition the objects. However, I believe that it is quite useful to allow objects to reside in multiple worlds, simultaneously. In this state of affairs, when we change an object we need to know which worlds should be affected, of the many worlds that might contain the object. Suppose that at any point in time there is established a list of worlds, and when a given object is modifed, we modify it in the first world on the list in which the object appears. This is similar to a search path in a file system. An interesting question is how should this list be maintained during computation, and whether there are reasonable schemes other than a simple list for organizing the worlds.

New worlds are created by copying old ones. When such a copy is made, the copy and the original can and will evolve separately. This might be implemented by copying the object table of the original world, and copying the objects only when they are actually modified (a kind of per-object copy-on-write facility).

Subworlds and Versions of Worlds

The model as described so far supports diverging worlds only. This could be very useful for "what-if" computations, but is of limited utility unless we can install results from a separate computation back into the main stream. To allow merging, I introduce the notion clearly need some sort of concurrency control. Further, a transaction might modify a number of worlds, and desire to commit only if all of the versions can be installed. Thus we need an atomicity mechanism in support of transaction commit. At this point it would appear that we have something similar to optimistic concurrency control.

Cooperation and Sharing

Even though subworld versioning allows tentative computations to be kept separate and then merged in later, we still have not dealt effectively with cooperation. Frankly, what I have to offer is still along the lines of mechanism rather than a real model, but perhaps it will be of some help. I propose two extensions to the mechanism as it stands.

First, when a subworld is installed, we have the option of not installing some of *its* subworlds. In particular, if someone else has modified a piece of the subworld, we may accept their changes. This would seem reasonable (in many cases) provided none of their changes overlap ours. However, it would definitely help to add some notion of semantics and integrity contraints. This leads to the second extension. We can support recording the operations performed on or within a (sub)world. This information could be used to determine if subworld installation is all right.

It seems that a subworld (or perhaps a world in general) might best be used to contain the pieces of an abstract object, and that the subworld gives a good way to localize and control the state of that object, even when it is spread over a number of the simple storage objects first discussed. Note that subworld installation now makes a little more sense – it represents the atomic application of some operations to a single abstract object. However, since we added the logs, we can do more then before: we can actually attempt to merge the operations, either by merging changed data (when that is possible), or by executing appropriate operations to form the logical merge state.

At the workshop it was suggested that it is not so much the notion of serializability that is wrong, but our concept of exactly what forms a transaction. I am inclined to agree, but it is difficult to make this notion precise; the difficulty is determining what collections of actions are (should be) meaningful transactions and defining consistency in a suitable way. One suggestions was that if there are multiple participants in a transaction they must (a) each have read all the others' writes, and (b) all request commit of the transaction. This captures agreement or acceptance of all changes by all parties operationally. Perhaps we can find a way to capture it more axiomatically, which would make we rather more comfortable (how does a party to a transaction decide if another party's update is all right?). Even the operational statement may not be quite right. Perhaps part (a) should

A Practical Language To Provide Persistence and a Rich Typing System

Deborah A. Baker, David A. Fisher and Jonathan C. Shultis
Incremental Systems Corporation
319 S. Craig Street
Pittsburgh, PA 15213
U.S.A.

I knew an old woman
Who swallowed a fly.
I don't know why
She swallowed the fly.
Perhaps she'll die!

— Traditional

Abstract

There is a pressing need for practical languages that support production of reliable, efficient and reusable software over a wide range of applications and act as cooperative elements of an integrated software development environment. Such a programming language must have a rich type system to formalize the software development concepts and mechanisms for managing "objects" of those types. Current databases do not provide an adequate type system. Current programming languages do not adequately address persistence of objects.

We are designing a language, tentatively called **prism**, which merges database and programming language concepts by expanding the range of types from a small fixed set of types typical of modern database systems to encompass all types definable within a programming language, and simultaneously expanding the extent rules of typical programming languages to encompass universal extent.

Manifesto

The woeful inadequacy of current software engineering practice is widely recognized, and there is no need to belabor the problems here. However, what is not so widely recognized, is that we cannot make significant progress without radically altering the way we design and build software systems.

Programming languages, operating systems, and databases, as long as they remain separate entities, create insurmountable obstacles to effective software design and maintenance. In particular, their separateness prohibits the specification and exploitation of global information about entire applications. Consequently, it is impossible to ensure the global integrity of applications, to maintain integrity over time, or to obtain efficient implementations.

information. In particular, the abstract properties of the global, persistent state of an application have to be made explicit and formal. Only by doing so can the global integrity of the application be maintained automatically. Only by doing so can requirements, specifications, designs, code, tests, versions, and so forth be composed, checked, derived, and otherwise manipulated automatically. Only by doing so can this information be used to generate efficient code specifically for each application, instead of interpreting general-purpose command languages. (Note that command language expression (query) optimization is only a halfway measure; the optimized expression is still interpreted.)

We are designing and implementing an experimental language, called **prism**, which seeks to encompass the full spectrum of concerns in a software development environment. The success of such an enterprise requires keeping tight control over the number and complexity of the language features, for fear of engendering an unwieldy monster. Our basic thesis is that there are relatively few fundamental concepts underlying all aspects of software development, and that most of the complexity and lack of integration of software development environments today results from the proliferation of incompatible special cases of these general concepts. Efficient implementation of the particular combinations of these concepts appearing in an application depends upon the availability of information about the properties of those combinations, i.e. type information.

Our goal here is to sketch in general terms the features of **prism**, with particular attention to the issues of "database programming languages". Specifically, we address the issues of persistent data, evolving applications, types, and error handling. We close with a few remarks about the ingredients in our design, but without discussing details of syntax and semantics, which would be premature.

Persistence

If we look for an explanation of why the semantics of applications has become spread out over so many different system components, it seems that the sharp separation of the internal state of a program from its external environment is at fault.

During the execution of a program, it is the responsibility of the compiler to ensure that the programmer's intentions are carried out correctly and consistently. The programming language is the means by which the programmer expresses those intentions. By relegating the results of programs to files outside the scope of the language, the designers of our early languages implied that once a program is done processing some data, the programmer has no intentions to express, i.e., is no longer interested in those data.

differently.

The mistake in this thinking is that it confuses the abstraction with its implementation. There is nothing at all wrong with choosing one implementation (e.g. fixed offsets) for program data, and another (e.g. directories with modifiable location bindings) for persistent data. The compiler should be free to choose any representation that correctly represents the intentions of the programmer. In practice, we would expect to use many different representations for any given abstraction.

The only thing required to bring persistent data into the realm of programming is the notion of *universal extent*. Conceptually, there is a single routine, the universe, in which all things exist and take place. The entities in the universe can be counted (over time), and hence can be uniquely identified, by binding each one to a universal name. The universal name of an entity contains no information about that entity; that is, universal names are an unbounded, unordered, discrete type. Note that in order to support things like removable media and network growth, the name space has to be shared among all systems, everywhere, over all time.

The abstraction of universal names is a trivial generalization of the notion of access types (pointers). The representation of a universal name may vary considerably, depending on such things as storage device characteristics and extent. Moreover, the language implementation is free to convert between representations as it sees fit. How and when such conversions are carried out will depend upon details of the application, and the intelligence of the "compiler".

Of course, it must be possible to attach a name to a value of any type in the language. That is, data of any type can be persistent. And, the scope rules of the language will dictate some obvious constraints on the relationships between persistent objects; for instance, no object can outlive its type, so types must also be capable of being persistent data. And so forth. Some examples, illustrating how universal names are used to solve persistent data problems, are given in a companion paper [BFS87].

Evolution

The distinguishing feature of large, long-lived applications is that they evolve over time. Evolution occurs through the continuing interaction of independently activated, concurrent, and distributed processes. Some applications eventually become extinct (i.e. terminate), while others are expected to continue until the universe (the *real* main program) ends.

From these remarks we see that, like routines, applications can be initiated programmer be prevented from specifying information that is deemed useful, whether intended for the compiler, an analysis tool, a human reader, or for any other purpose. Information useful to the compiler, for instance, might include bounds on the length of a sequence, which could help the compiler to choose among alternative representations of sequences or to transform the program into a more efficient form.

Type theory has focussed recently on the Curry-Howard isomorphism between types and propositions, viewing type systems as logico-deductive mechanisms. The idea is that the type of a program (expression) asserts something about the outcome (conclusion, result) of the program. The problem is that this does not apply neatly to applications which, as we pointed out in the preceding section, do not generally conclude. Moreover, when they do conclude, we have no interest in the outcome. On the contrary, we are only interested in the intermediate stages of applications.

Another aspect of applications which does not fit neatly into most of the recent work on types is that they involve concurrency. To our knowledge, the only serious attempt to treat concurrency in a logical framework is linear logic [Gir86]. Even there, however, there is a serious problem associated with attaching a meaning to nonterminating deductions.

The time dependence of applications immediately suggests some kind of modal logic, if we want to adopt the idea of applications as proofs. It seems more natural to us, however, to consider applications not as proofs but as theories. The computation of an intermediate result is therefore treated as a deduction within that theory. Sound applications evolve internally by adding and deleting nonlogical axioms that are independent of both the theory and one another.

The principle formal concept underlying **prism** is therefore not *proposition*, but *theory*. Theories, in turn, are simply bodies of information which are required to be internally consistent. The criteria for consistency of

compiler, the more it should be able to infer. Moreover, the *kind* of information from which it infers the implementation is allowed to vary, so that any program specification paradigm can be accomodated. For this reason, there is no *required* syntactic form for anything in the language, though there is a set of predefined forms which the programmer is always free to use. The basic concepts of the language can be extended by defining new semantic abstractions, including the relations between abstractions needed by the compiler. Example of such relations are the consistency relation between Ada package specifications and bodies, the realizability relation which enables computational terms to be derived from proofs, and the resolution algorithm which enables sets of ground terms satisfying a proposition to be inferred from a set of nonlogical axioms within an appropriate framework such as Horn logic. From these remarks it should be clear that the intelligence of the compiler is not fixed, but can continually accumulate programming knowledge to the benefit of all users.

language of propositions, where as usual we interpret such things as "integer" to be propositional constants, "record", "function", etc. as propositional connectives, and various kinds of constraints as special predicates. The theorems of this theory consist of the closure of the declarations under the composition rules of Ada, where the type checking rules serve as inference rules.[5]

A package body, on the other hand, declares a different kind of information; formally, it defines a representation morphism yielding a model of the theory given by the specification. As such, it is an example of a specific deductive mechanism which is consistent with the specified theory.

Packages illustrate the two basic kinds of information in **prism** alluded to earlier. The package specification is an example of declarative (syntactic, specification) information, and the body is an example of deductive (semantic, composition) information. The example at hand illustrates mechanisms for declaring and deducing (computing) with a certain class of theories. As indicated earlier, however, **prism** allows arbitrary new types of information of both classes to be defined.

To achieve the required level of generality, the fundamental notions of declaration and composition in **prism** have a categorial flavor. Specifications are analogous to objects and deductions are categorial constructions. To some extent, therefore, we share inspiration with CAML [Cur83]. However, CAML restricts itself to a very special category, of sets and functions, in order to fix an interpretation of composition. By making composition and its specification abstract, however, **prism** programmers are free to attach any interpretation to composition that yields a model. This is perhaps the most important purpose for which multiple representations can be attached to an abstraction! That is, the crucial feature of **prism** which makes it different from all other data abstraction languages is its lack of rules about how the programmer is to give meaning to specifications, and the nature of that meaning. It follows directly from this that things can share a set of formal properties but differ considerably in the details of their meaning.[6]

[5]Note that package specifications can use a number of mechanisms for synthesizing theories, including extension and inheritance (with clauses). The particular set of theory synthesis mechanisms in Ada is, however, rather ad-hoc. A clearer and more complete set of mechanisms is apparent in the Larch shared language [GHW85].

[6]In retrospect, it is somewhat surprising, in light of the demonstrated capacity of category theory to unify so much of mathematics through abstraction, that this kind of separation of syntax and semantics has not been incorporated in programming languages before. Put another way, category theory has infinitely more polymorphism than any programming language because of its decoupling of models and theories, and we propose to follow its example.

transfer of control from probes back to the system, when this is meaningful (recovery).

Here are some illustrations of how the language issues of visibility, binding, and resource allocation arise in the context of instrumentation. The environment in which a probe executes determines what user-defined types and data it can access (if any), or whether certain run-time system information is visible. Binding time determines such things as whether breakpoints can be installed interactively, or have to be "compiled in". In performance instrumentation, resources must be apportioned among the observed system, data collection, data reduction and analysis, and presentation and user interaction (if any) so as to minimize intrusiveness.

Currently, the **prism** core includes mechanisms for raising and handling exceptions which are similar to those in Ada. On a more fundamental level, these mechanisms depend on synchronous and asynchronous control transfer. However, we are acutely aware that this is only a start.

The most difficult problem here is in the area of abstraction. Ideally, one would like to say "measure the X of system Y", and have any necessary probes, data reduction facilities, etc., generated, installed, and run automatically. Or, better yet, "determine how well system Y's behavior matches hypothesis Q", thereby tying testing back to design specifications. The realization of these ideals requires mechanisms for defining and manipulating abstract properties of systems. Unfortunately, we don't yet understand the logical/type theoretic aspects of error detection and handling well enough to know how to support the necessary abstractions.

Design

Naturally, good language design practice is required in the design of any language [Wei71, Hoa73, Iron76]. What constitutes good design depends in part on how, by whom, and for what purposes the language will be used. Some guidelines that we have adopted in the design of **prism** are the following.

Because the applications are varied and many, it is necessary to provide a small number of highly composable mechanisms, instead of a large number of mechanisms specialized to an arbitrarily chosen set of anticipated applications. To retain simplicity in the language each primitive mechanism must isolate some unique functionality in a form that is easily composed with the other primitives. Every effort should be applied to avoid language features that will lead to psychological ambiguities in programs. The design should emphasize readability over ease of writing programs. It should emphasize the semantic integrity and completeness of the language. It should provide redundancy without duplication. It should avoid default mechanisms that like Ada, on the other hand, provides efficient mechanisms which can be combined to obtain an efficient implementation, but at the loss of the general solution.

The problem is that the common Lisp programmer can't convey enough information about the application to the compiler for it to obtain an efficient implementation, while an Ada programmer cannot avoid conveying so much information about the details of his particular solution that the compiler is unable to abstract the general solution. In a full spectrum language, the programmer should be able to communicate to the compiler, as part of the program, any information it needs to derive an efficient implementation of a specialized solution.

Typed functional languages enable more efficient implementations by including type information in programs. Types constrain the application, promote checking and representation decisions to an earlier point in the computation, and enable a wide class of optimization transformations.

The more intricate a type system is, the more information can be expressed. For instance, dependent types can be used to inform the compiler to represent a list as an array if the length of the list is known to depend on a numeric parameter. In the extreme, virtually any logical property that has constructive significance can be embodied in type information (at which point we say we are doing "logic programming").

The information a compiler needs isn't restricted to functionality, however. To cite a few examples, the criteria to be used in optimization, expected statistical characteristics of input data, and complexity measures of components can all be used to guide the compiler's selection of algorithms and data structures.

As language implementors, we know how to make compiler components that are driven by user-supplied information and are hence open-ended. What is less clear is what high-level syntactic mechanisms should be supplied to enable the application designer to express information and convey it to the portions of the compiler that need it. This is the most difficult syntax design challenge we face. Although we have worked out some prototype models of the language, we are not yet sufficiently satisfied with any of them to expose their details. We do expect, however, to have a preliminary design available for review in about one year.

Object-oriented languages, operating systems, and databases are currently experiencing the greatest experimental activity in the areas of inheritance mechanisms and persistent data issues, and so we look to them to supply perspectives and mechanisms in these areas. In particular, these languages contribute a third baseline of features, in addition to those found in Ada and

VII, January 1985.

[Cur83] Curien, P.L., "Combinateurs Catégoriques, Algorithmes Séquentiels et Programmation Applicative", Thèse de Doctorat d'Etat, Université Paris VII, December 1983.

[Fai83] Fairbairn, J., "Ponder and Its Type System", *Polymorphism, Vol. 1, No. 2*, The ML/LCF/Hope Newsletter, April 1983.

[FS79] Fisher, D.A. and Standish, T.A., "Initial Thoughts on the Pebbleman Process", Institute for Defense Analyses (IDA) Paper P-1392, June 1979.

[FW86] Fisher, D.A. and Weatherly, R.M., "Issues in the Design of a Distributed Operating System for Ada", *IEEE Computer*, Vol. 19, No. 5, May 1986, pp. 38-47.

[Gan86] Ganziner, H. and Jones, N.D., editors, "Programs as Data Objects", *(Workshop Proceedings)*, LNCS 217, Springer-Verlag, April 1986.

[GHW85] Guttag, J.V., Horning, J.J. and Wing, J.M., "Larch in Five Easy Pieces", Report #5, Digital System Research Center Reports, July 1985.

[Gir86] Girard, J.Y., "Linear Logic", Université Paris, October 1986.

[GMW79] Gordon, M.J., Milner, A.J. and Wadsworth, C.P., "Edinburgh LCF", *Lecture Notes in Computer Science, No. 78*, Springer-Verlag, Berlin, 1979.

[Gol86] Goldberg, A.T., "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques", *IEEE Transactions on Software Engienering, SE-12* (7), July 1986, pp. 752-768.

[Gol85] Goldsack, S.J., editor, *Ada for Specification: Possibilities and Limitations*, Cambridge University Press, 1985.

[GR83] Goldberg, A. and D. Robson, *Smalltalk-80 : The Language and its Implementation*, Addison Wesley, 1983.

[Hai86] Hailpern, B., "Multiparadigm Languages and Environments", *IEEE Software*, 3(1), January 1986.

[Hoa73] Hoare, C.A.R., "Hints on Programming Language Design", *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, October 1973.

[How80] Howard, W.A., "The Formulæ-As-Types Notion of Construction", Unpublished manuscript 1969. Reprinted in H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Seldin, J.P. and Hindley, J.R. editors, Academic Press, 1980.

[Hue87] Huet, G., "A Uniform Approach to Type Theory", INRIA, 1987.

[Iron76] "Ironman", Department of Defense Requirements for High Order Computer Programming Languages, HOLWG Report, June 1976.

[KC86] Khoshafian, S.N., and Copeland, G.P., "Object Identity", *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, October 1986, (also *SIGPLAN Notices*, November 1986), pp. 406-416.

[Kin85] King, R.M., "Knowledge-Based Transformational Synthesis of Efficient Structures for Concurrent Computation", Ph.D. Thesis, Rutgers University, Kestrel Institute Report, KES.U.85.5, April 1985.

[Lee87] Lee, P., "The Automatic Generation of Realistic Compilers from High-level Semantic Descriptions", Ph.D. Thesis, University of Michigan, 1987.

[Lei83] Leivant, D., "Reasoning About Functional Programs and Complexity Classes Associated with Type Disciplines", *Twenty-fourth Annual Symposium on Foundations of Computer Science*, Tucson, Arizona, 1983, pp. 460-496.

[LS83] Lampson, B.W. and Schmidt, E.E., "Organizing Software in a Distributed Environment",

Database Updates in Logic Programming

Shamim Naqvi
Ravi Krishnamurthy

MCC
P. O. Box 200195
Austin, TX 78720

1 Preamble

The idea that the meaning of a logic program is the minimal model associated with that program has attracted wide acceptance since Kowalski [8] introduced the essential ideas of logic programming thirteen years ago. Since then, notions such as layering and stratification [1,11,5], and the introduction of set terms [4,9] have implicitly introduced a certain amount of ordering, i.e., procedurality, without sacrificing the model-theoretic semantics of logic programs.

We feel that the time has come to *explicitly* add some procedural notions to logic programs. Our feeling derives in part from the implicit ordering imposed by stratified programs containing negation and set terms, and from some of our recent work on adding updates to a logic database language. Update transactions are inherently procedural; one could attempt to hide this procedurality by writing the rules in some complicated manner so that the required order within operations is maintained. There is no guarantee that such re-writing is always possible. Even in those cases when it is, the disadvantage of such an approach is that the resulting programs would be opaque to the human reader as well as to any optimization strategy. An obvious case in point is the well-known and simple *if-then-else* construct which when written in logic programs requires a number of distinct rules, making programs hard to read and optimize.

These and other such considerations have lead us to believe that one should, to use a colloquial expression, *bite the bullet* and make the procedurality, along with its concommitant notion of state, explicit in the language. What we would then have is a logic programming language with procedural constructs and what one should strive for is a *declarative semantics* for programs in such a language.

It is not possible here for us, given the brevity of an extended abstract, to consider this problem in all its generality. Hence, in this paper we decided to concentrate on one subproblem of the general problem outlined above and present a solution for it. Our intention is to impart the essence of our approach through this solution.

The syntax of a query is: $\leftarrow B_1, \ldots, B_n.$ where $(\forall 1 \leq i \leq n) B_i \in \Phi_0$.

A query $Q1$ is ***equivalent*** to a query $Q2$, $Q1 \equiv Q2$, if the following holds for some $\alpha, \beta \in \Phi_0$:

$Q1 :\leftarrow \ldots, \alpha, \beta, \ldots,$ *and* $Q2 :\leftarrow \ldots, \beta, \alpha, \ldots$

i.e., $Q1$ and $Q2$ differ exactly in two comma-separated adjacent positions; or there exists a query $Q3$ such that $Q1 \equiv Q3$ and $Q3 \equiv Q2$. Intuitively, Q1 and Q2 result in the same final state of the database, from the same starting state. We shall formally define this semantic notion of equivalence later.

A *well-ordering* of a given query is the left-to-right predicate occurrences of an equivalent query. A variable in a predicate is said to be *covered* if it occurs in some predicate occurrence preceding it in a well-ordering. A *well-formed* query is a query in which all variables (if any) in update predicates are covered. For example, the query [$\leftarrow$ +p(X), q(X)], is well-formed since *q(X),+p(X)* is an equivalent query in which the variable "X" is covered.

For convenience of writing short programs, we allow two additional compound update predicates: For $P \in \Phi_{-2}$, and $\alpha, \beta \in \Phi_0$,

$[P?\alpha] \equiv (*(P?\alpha), \neg P?)$ (this stands for "while P do α").

$\{P, \alpha, \beta\} \equiv ((P?\alpha), (\neg P?\beta))$ (this stands for "if P then do α else do β").

3 Examples of Programs

Before giving formal semantics of UDatalog programs, we present examples of programs in this section.

Insert a tuple (john,db,20K) in the relation eds.

$\leftarrow +eds(john, db, 20K).$

Delete a tuple (peter,db,30K) from the relation eds.

$\leftarrow$ *−eds(peter,db,30K).*

Give every database employee a 10% salary increase.

$\leftarrow$ *eds(X,D,S) , S1=S*1.1, (−eds(X,D,S) ; +eds(X,D,S1)).*

Notice that the query predicates provide bindings for the variables in the update predicates.

Continue increasing salaries by 10 percent while Francois' salary exceeds 300K.

$\leftarrow$ *[(eds(francois,db,S),S≤300K)? eds(X,db,S) , S1=S*1.1 ,*
(−eds(X,D,S) ; +eds(X,D,S1))].

Fire all employees who make more than their managers:

given by a binary relation, ρ, on states, including the pair (s,t) in ρ iff the predicate in question is true in some state "t", and there exists a state "s" such that the predicate maps "s" to "t". The meaning of a formula Q is to be the subset of W consisting precisely of those states which satisfy it".

We shall now provide the details of the mapping ρ. $\rho : \Phi_0 \rightarrow 2^{W \times W}$ assigns to each formula, say α, some binary relation on states with the intended meaning $(s,t) \in \rho(\alpha)$ iff execution of α in state s can lead to the state t.

Let P be a ground predicate.

$\rho(P) = \{(s,s) \mid P \in s\}$. $\qquad \rho(+P) = \{(s,t) \mid t = s \cup \{P\}\}$
$\rho(\neg P) = \{(s,s) \mid P \notin s\}$. $\qquad \rho(-P) = \{(s,t) \mid t = s - \{P\}\}$

Thus "+" and "–" respectively add and delete a ground fact from the given database leaving all other facts unchanged, and query predicates may be considered as updates which do not change the state of the program. Thus, the states s and t above are minimally different from each other. The following proposition shows that the "+" and "–" are deterministic, i.e., there is a unique final state for each basic update predicate.

Proposition 1: For $\alpha \in \Phi_{-1}$, i.e., α is a basic update predicate, if $(s,t) \in \rho(\alpha)$ and $(s,t') \in \rho(\alpha)$ then $t = t'$.

Proof: Omitted. ■

We now turn our attention to compound predicates. Let α and $\beta \in \Phi_0$ be arbitrary predicates and $P \in \Phi_{-2}$ be a query predicate, or a conjunction of query predicates.

$\rho((\alpha)) = \rho(\alpha)$
$\rho(*\alpha) = \{(s,t) \mid \exists k \exists s_0, \ldots, s_k(s_0 = s \text{ and } s_k = t)$ and $((\forall 1 \leq i \leq k)(s_{i-1}, s_i) \in \rho(\alpha))$
and $\forall s' \neq t(s_k, s') \notin \rho(\alpha)\}$
$\rho((P?\alpha)) = \{(s,t) \mid (s,s) \in \rho(P), (s,t) \in \rho(\alpha)\} \quad \cup \quad \{(s,s) \mid (s,s) \notin \rho(P)\}$
$\rho((\alpha;\beta)) = \{(s,t) \mid \exists u(s,u) \in \rho(\alpha), (u,t) \in \rho(\beta)\}$.
$\rho(\alpha,\beta) = \rho(\alpha;\beta) = \rho(\beta;\alpha)$
$\rho(P \leftarrow \alpha) = \rho(P), \rho(\alpha) \text{ or } \rho(\neg\alpha)$

Proposition 2: Let α be a compound update predicate. Then if $(s,t) \in \rho(\alpha)$ and $(s,t') \in \rho(\alpha)$ then $t = t'$.

Proof: Omitted. ■

Note that the meaning of the $(\alpha;\beta)$ construct in this report places the responsibility of the order of execution upon the user. The semantics of the language make no use and no claim that α executed

$\delta'(P) = def(P)$
$\delta(\alpha) = \{p(t_1, \ldots, t_n) \mid p \text{ is a predicate symbol of arity n}, \quad p \in \delta'(\alpha), \text{ and } (\forall 1 \leq i \leq n) t_i \in U\}$

We use this notion of reference sets to state the syntactic condition to guarantee satisfaction of CRP of the update predicate α, β.

Theorem [CRP]: α, β satisfies CRP if $\delta(\alpha) \cap \delta(\beta) = \phi$.

Proof: Omitted. ∎

We now define truth-values of formulae. Recall that W is a subset of the powerset of the Herbrand Base of a program. The elements of W are called *states*. It is convenient to define a mapping $\tau : \Phi_0 \rightarrow 2^W$ as follows. Let $A \in \Phi_0$. Then $\tau(A) = \{t \mid \exists s(s,t) \in \rho(A)\}$.

Given a structure $M = (W, \rho)$, if $A \in \Phi_0$ is a ground atomic formula then A *is said to be true in state s* (or that *s satisfies A*), written as $M, s \models A$, if $s \in \tau(A)$. We write $M \models A$ and say that A is *true*, or more precisely *M-true*, if $M, s \models A$ for all $s \in W$.

Let $M = (W, \rho)$ be a structure, and *L=(R,S,Q)* be a UDatalog program. Note that S is an element of W. Let $R(S)$ denote the fixpoint of applying the rules R to the set of facts S. The notion of a model of a program *L=(R,S,Q)* is defined as follows. $M = (W, \rho)$ is a model of L iff for every $s \in W$,

1. $(M, s) \models Q$, i.e., Q is *M-true*.
2. every rule in R is satisfied by $R(s)$.

Informally, M is a model of a program L iff every state of M satisfies the query Q of the program, and the fixpoint of every state satisfies every rule of the program.

We now define the notion of minimal models of an extended program. $M = (W, \rho)$ is a minimal model of L iff

it is a model of L,
$(\forall s \in W) R(s)$ is a minimal model of the rules in R, and
there is no $M' = (W', \rho')$ different from M such that

(a) M' is a model of L and
(b) For every base predicate p, $\rho'(p) \subseteq \rho(p)$,

In other words, M is a minimal model of L iff it is a model, and every state in the model is as "small" as possible, and the cardinality of ρ is as "small" as possible. It is possible to propose a constructed model as it was done in the definition of logic programs, and subsequently show that the constructed model and the above declarative model are the same model. However, for the sake of brevity, we avoid this exercise in this extended abstract.

$s(Y) \leftarrow s(X), g(X,Y), \neg us(Y).$
$us(Z) \leftarrow g(W,Z), \neg s(W).$

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 UDatalog program that constructs the unique minimal model of the problem above:

$s(Y) \leftarrow s(X), g(X,Y), \neg us'(Y).$
$us(Z) \leftarrow g(W,Z), \neg s'(W).$
$\leftarrow *(s(Y), us(Z), s'(Y1), us'(Z1); -s'(Y1), +s'(Y), -us'(Z1), +us'(Z)).$

Note that s' and us' are base predicates in the above program with an empty set of tuples initially. We claim that this program generates the unique minimal model of the graph problem above.

The above two cases exemplify the generality of the approach to adding procedurlity to logic programs while still retaining declarative semantics.

6 Future Research Directions

As was alluded to in the introduction of this paper, we consider the major contribution of this work to lie in the formal basis that has been established to look at adding procedurality to logic programs. We have not discussed the ability to state procedural constructs in the rules of a program. This requires a modification to our model semantics and the bottom-up model construction. Further, the uniqueness and minimality of models needs to be re-established.

Implementation considerations lead to limitations on allowable predicates. For example, consider a rule whose body contains the conjunct *+u1; q; +u2* where "u1" and "u2" are updates and "q" a query. Now, after doing "u1" if "q" fails then we have to backtrack and "un-do" the effects of "u1". In order to inhibit such behaviors we may require that "u2" be written as an "always true" predicate, say as "true?u2". Similarly, one may limit the predicates occurring within a "*" for efficiency reasons. These and other such topics are the subject of a fuller presentation.

We thank Shalom Tsur and Oded Shmueli for helpful suggestions and Carlo Zaniolo, Haran Boral and Patrick Valduriez for a careful review of the manuscript. Carlo has recently helped us with many implementation ideas and in particular the second example in section 5 was suggested by him. Some update examples were taken from a private manuscript by S. Tsur and D. Maier.

Quelqu'un pourrait dire de moi
que j'ai seulement fait ici
un àmas de fleur étrangères,
m'y ayant fourni du mien que le filet à les lier.
—MONTAIGNE, **Essais** (1533-1592), III.xii

queries; these are passed to an underlying off-the-shelf relational database system for query optimization.

This paper describes a fully integrated compile-time approach that ensures both safety and optimization to guarantee the amalgamation of the database functionality with the programming language functionality of LDL. Therefore, the LDL optimizer subsumes the basic control strategies used in relational systems as well as those used in [MUV 86]. In particular for LDL programs that are equivalent to the usual join-project-select queries of relational systems, the LDL optimizer behaves as the optimizer of a relational system[Sel 79].

The technical challenges posed by the LDL optimizer follow from its expressive power extending far beyond that of relational query languages. Indeed, in addition non recursive queries and flat relational data, Horn Clauses include recursive definitions and complex objects, such as hierarchies, lists and heterogeneous structures. Beyond that, LDL supports additional constructs including stratified negation [BN 87], set operators and predicates [TZ 86, BN 87], and updates [NK 87]. Therefore, new operators are needed to handle complex data, and constructs such as recursion, negation, sets, etc.. Moreover, the complexities of data and operations emphasize the need for new database statistics and new estimations of cost. Finally, the presence of evaluable functions and of recursive predicates with function symbols give the user the ability to state queries that are *unsafe* (i.e., do not terminate). As unsafe executions are a limiting case of poor executions, the optimizer must guarantee that the resulting execution is safe.

In this we limit the discussion to the problem of optimizing the pure fixpoint semantics of Horn clause queries [Llo 84]. After setting up the definitions in Section 2, the optimization is characterized as a minimization problem based on a cost function over an execution space in Section 3. The execution model is discussed in Section 4, using which the execution space is defined in Section 5. We outline our cost function assumptions in Section 6. The search strategy is detailed in Section 7 by extending the traditional approach to the nonrecursive case first; and then extended to include recursion. The problem of safety is addressed in section 8, where we extend the optimization algorithm to ensure safety.

2. Definitions

The knowledge base consists of a *rule base* and a *database* (also known as fact base). An example of rule base is given in Figure 2-1 . Throughout this paper, we follow the notational convention that Pi's, Bi's, and f's are *predicates, base predicates* (i.e., predicate on a base-relation), and *function symbols*, respectively. The Bi's are relations from the database and the Pi's are the derived predicates whose tuples (i.e., in the rela-

3. The Optimization Problem

We define the optimization problem as the minimization of the cost over a given execution space (i.e., the set of all allowed executions for a given query). This is formally stated below.

Logic Query Optimization Problem:
Given a query Q, an execution space E and a cost function defined over E, find an execution pg in E that is of minimum cost; i.e.

$$\underset{pg \in E}{\mathrm{MIN}} \left[\text{cost of } pg(Q) \right]$$

Any solution to the above optimization problem can then be described along four main coordinates, as follows:
i) the model of an execution, pg;
ii) the definition of the execution space, E, consisting of all allowable executions;
iii) the cost functions which associate a cost estimate with each point of the execution space; and
iv) the search strategy to determine the minimum cost execution in the given space.

The model of an execution represents the relevant aspects of the processing so that the execution space can be defined based on the properties of the execution. The designer must select the set of allowable executions over which the least cost execution is chosen. Obviously, the main trade-off here is that a very small execution space will eliminate many efficient executions, whereas a very large execution space will render the problem of optimization intractable, for a given search algorithm. In the next sections we describe the design of the execution model, the definition of the execution space, and the search algorithm. The cost formulae are in most cases system dependent. Thus we will consider the cost formulae as a black box, where the actual formulae are not discussed except for those assumptions that impact the global architecture of the system.

4. Execution Model

LDL's target language is a relational algebra extended with additional constructs to handle complex terms and fixpoint computations. An execution over this target language can be is modelled as a rooted directed graph, called '*processing graph*', as shown in Figure 4-1b for the example of of Figure 2-1. Intuitively, leaf nodes (i.e., the nodes with non-zero in-degree) of this graph correspond to operators and the results of their predecessors are the input operands. The representation in this form is similar to the predicate connection graph [KT 81], or rule graph [Ull 85], except that we give specific semantics to the internal nodes, and use a notion of contraction for recursion as described below.

Associated with each node is a relation that is computed from the relations of its predecessors, by doing the operation (e.g., join, union) specified in the label. We use a square node to denote materialization of relations and a triangle node to denote the pipelining of the tuples. A pipelined execution, as the name implies, computes only those tuples for the subtree that are relevant to the operation for which this node is an operand. In the case of join, this computation is evaluated in a lazy fashion as follows: a tuple for a subtree is generated using the binding from the result of the subquery to the *left* of that subtree. This binding is referred to as *binding implied by the pipeline.* Note that we impose a *left to right* order of execution. Subtrees that are rooted under a materialized node are computed bottom-up, without any sideways information passing; i.e., the result of the subtree is computed completely before the ancestor operation is started.

Each interior node in the graph is also labeled by the method used (e.g., join method, recursion method etc.). The set of labels for these nodes are restricted *only* by the availability of the techniques in the system. Further, we also allow the result of computing a subtree to be filtered/projected through a selection/restriction/projection predicate. We extend the labeling scheme to encode all such variations due to filtering and projecting. The label for a CC node is to specify the choices for the fixpoint operation, which are the choices for SIPs and recursive method to be used.

The execution corresponding to a processing tree proceeds bottom-up left to right as follows: The leftmost subtree whose children are all leaves is computed and the resulting relation replaces the subtree in the processing tree. The computation of this subtree is dependent on the type of the root node of the subtree -- pipelined or materialized -- as described above. If the subtree is rooted at a contracted clique node, then the fixed point result of the recursive clique is computed, either in a pipelined fashion or in a materialized fashion; the latter requires the use of techniques such as Magic Sets or Counting [BMSU 85, SZ 86].

5. Execution Space

Note that many processing trees can be generated for any given query and a given set of rules. These processing trees are logically equivalent to each other, since they return the same result; however very different costs may be associated with each tree, since each embodies critical decisions regarding the methods to be used for the operations, their ordering, and the intermediate relations to be materialized. The set of logically equivalent processing trees thus defines the *execution space* over which the optimization transformational rule defined above}. For example, {MP, PR}, {MP, PR, PS, PP} are execution spaces.

As mentioned before, the choice of proper execution space is a critical design decision. By limiting ourselves to the above transformations, we have excluded many other types of optimizations like peep-hole optimizations, semantic optimizations, etc. This is a reflection of the restrictions posed in the context of relational systems from which we have generalized and is not meant to imply that they are considered less important. As in the case of relational systems, these supplementable optimizations can also be used. Even in the realm of above transformations, we were unable to find an efficient strategy for the entire space. Consequently, we limit our discussion in this paper to the space defined by {MP, PS, PP, PR, PA, EL} (i.e., Flattening and Unflattening are not allowed). As discussed in Section 8, programs can be constructed for which no safe (and therefore, no efficient) executions exists without flattening. Our experience with rule based systems, however, has been that these are artificial situations which the user can be expected to avoid without any additional inconvenience.

6. Cost Model:

The cost model assigns a cost to each processing tree, thereby ordering the executions. Typically, the cost spectrum of the executions in an execution space spans many orders of magnitude, even in the relational domain. We expect this to be magnified in the Horn clause domain. Thus "it is more important to avoid the worst executions than to obtain the best execution", a maxim widely assumed by the query optimizer designers. The experience with relational system has shown that the main purpose of a cost model is to differentiate between good and bad executions. In fact, it is known, from the relational experience, that even an inexact cost model can achieve this goal reasonably well.

The cost includes CPU, disk I/O, communication, etc., which are combined into a single cost that is dependent on the particular system. We assume that a list of methods is available for each operation (join, union and recursion), and for each method, we also assume the ability to compute the associated cost and the resulting cardinality. For the sake of this discussion, the cost can be viewed as some monotonically increasing function on the size of the operands. As the cost of an unsafe execution is to be modeled by an infinite cost, the cost function should guarantee an infinite cost if the size approaches infinity. This is used to encode the unsafe property of the execution.

Intuitively, the cost of an execution is the sum of the cost of individual operations. This amounts to summing up the cost for each node in the processing tree.

The results showed that the quadratic algorithm chooses the optimal permutation in most cases and in more than 90% of the cases, it produces no worse than twice/thrice the optimal. These results have been shown to have a statistical confidence of 95% with a 3% error.

Another approach to searching the large search space is to use a stochastic algorithm. Intuitively, the minimum cost permutation can be found by picking, randomly, a "large" number of permutations from the search space and choosing the minimum cost permutation. Obviously, the number of permutations that need to be chosen approaches the size of the search space for a reasonable assurance of obtaining the minimum. This number is claimed to be much smaller by using a technique called Simulated Annealing [IW 87]. We use this technique to the optimization of conjunctive queries as follows. For any given permutation, define a neighbor to be any permutation that differs in exactly two places (i.e., two positions in one permutation is interchanged to get the other). It is easy to prove that the closure of the neighbor (equivalence) relation is indeed the set of all permutations (i.e., the execution space for conjunctive queries). The simulated annealing can then be viewed as a "random" walk of the execution space using this neighbor relation. If we ignore the annealing parameters, then the neighbor relation completely characterizes the simulated annealing process. We shall use this notion to characterize the strategy using simulated annealing.

In short, we have summarized three generic strategies: exhaustive, quadratic and stochastic. The main trade-offs amongst these strategies is between efficiency (i.e., time complexity) and flexibility. Note that the quadratic strategy is the most efficient, whereas it is least flexible in terms of the possible modifications to cost functions, query structure, etc. Our goal is to present a design for the search strategy that is capable of using multiple strategies interchangeably. The main reason for requiring the system to be flexible is that the system is initially intended as an experimental vehicle since there is no prior experience in the design of an optimizer for a logic language and the field of logic languages is in its infancy; thus new ideas will be forthcoming that the design should be capable of incorporating into the system.

7.2. Nonrecursive Queries

Initially, we extend the exhaustive strategy that was used in the case of conjunctive queries to the nonrecursive case, which is then extended to the other two strategies. Extrapolating from the conjunctive case, selects/projects are always pushed down any number of levels for non-recursive rules by simply migrating to the lower level rules the

```
NR-OPT: Compute a processing tree for a nonrecursive logic query.
        Input is a processing tree rooted at a node N.
        Output is an optimized processing graph.

1) Node N is an AND node, say As:
    I) For each permutation of the sequence of subtrees,
       Using the binding implied by the permutation do:
         a) For each OR-subtree Os of As do: Compute NR-OPT(Os).
         b) Compute the cost for this permutation using the cost model.
         c) Maintain the minimum cost permutation.
    II) Return cost, cardinality, and the graph for the minimum cost processing graph.

2) Node N is an OR node say Os:
    1) IF this subtree, Os, has NOT already been optimized for this binding
       THEN do:
         a) For each AND-subtree As of Os : Compute NR-OPT(As).
         b) Compute the cost of the union of the children.
         c) record the cost, cardinality, graph, etc., for Os, indexed by the binding
    2) ELSE read cost, cardinality, graph, etc., for Os, based on the binding.
```

Figure 7-1: NR-OPT algorithm for non-recursive query.

79], we reduce the *n!* permutations to 2^n choices. Thus the worst case complexity becomes $O(N * 2^k * 2^n)$. Normally, the number of arguments per predicate (*k*) is usually less than five and number of predicates per conjunct (*n*) is usually less than 10. For these values of *k* and *n*, we conclude the feasibility of this approach based on the experience from commercial database systems.

The algorithm of Figure 7-1 becomes impractical for large values of *k* and/or *n*. The main practical concern is *n* since the number of arguments in recursive predicates is either small, or reducible to a small number by the use of complex terms. We discuss below how the algorithm in Figure 7-1 can be easily modified to take advantage of the quadratic strategy [KBZ 86] or of simulated annealing.

Note that the step 1) of the algorithm NR-OPT is responsible for the exponential behavior w.r.t. *n*. This step is a generalization of the optimization search for conjunctive query. Consequently, replacing the exhaustive strategy with the stochastic strategy is straightforward, whereas the incorporation of quadratic strategy is little more involved requiring the generalization of the ASI property. As this involves more detail discussion of the ASI prop-

and for each rule that has P.a in the head, we generate an adorned version for the rule as described below and add it to *Pgm'*. We then mark P.a. Note that the adorned version of a rule may generate additional predicates that are adorned. The process terminates when no unmarked adorned predicates are left.

The adornment for a recursive predicate in the body is assigned as follows: an argument is bound if the variable(s) in the argument occurs either in a bound argument of the head literal or in a goal that precedes it in the chosen permutation. All other arguments of this literal are adorned as free. Each literal P that is associated with a binding a is renamed as 'P.a'. We present below, the adorned programs for the query forms sg.bf and sg.bb, in which the chosen SIP for all replicated rules is self evident.

Original Rule: sg (X,Y) <- up(X,X1), sg(Y1,X1), dn(Y1,Y)

Adorned clique for the query sg.bf: ('bf' is the binding)

sg.bf (X,Y) <- up(X,X1), sg.fb(Y1,X1), dn(Y1,Y)
sg.fb (X,Y) <- dn(Y1,Y), sg.bf(Y1,X1), up(X,X1)

Adorned clique for the query sg.bb:

sg.bb (X,Y) <- up(X,X1), sg.fb(Y1,X1), dn(Y1,Y)
sg.fb (X,Y) <- dn(Y1,Y), sg.bf(Y1,X1), up(X,X1)
sg.bf (X,Y) <- up(X,X1), sg.fb(Y1,X1), dn(Y1,Y)

Note that for a given subquery and a permutation for each rule in the clique, the resulting adorned program is unique. Further, for a given adorned program, the transformed program by Magic Sets or Counting is also unique. As a result, the execution (and the associated cost) is uniquely determined, for a given cost and size estimates for all the literals (in the rules of the clique) that are not in the clique. From this we can conclude that the space of executions that are to be enumerated is defined by the different permutations of the rules in the clique. In other words, if there are *nc* rules in the clique, then each possible cross product of *nc* permutations defines a *c-permutation.* For each c-permutation, and a subquery there is an adorned program. Note that all of them are not distinct, but collectively they exhaust the possible adorned programs.

We extend the algorithm presented in the previous section to include the capability to optimize a recursive query. When a subtree rooted at a CC node is to be optimized, the choice is in adorning the node with the proper label. We have to enumerate all the c-permutations for the clique. For each such assignment of c-permutations, the rules are tions for the rules in the clique is impractical even for small number of rules in the clique. It is conjectured by many researchers that the mutual recursions are not common and complicated ones are used even less. So if this conjecture is true then exhaustive search may not be impractical.

Nevertheless, we are interested in being able to optimize larger class of queries. For this we present the use of the stochastic strategy. Note that if the enumeration of the search space consisting of all possible c-permutations of a clique (in case 3 of the algorithm) is improved, then the algorithm can be used for a larger class of queries. Further note that we observed that by specifying the neighbor relation for a given execution, such that the closure of this relation defines the space to be searched, we can characterize the simulated annealing process. We present such a neighbor relation here. Let us define a neighbor of a c-permutation, CP1, to be another cross product of *nc* permutations, CP2, such that all but one of these *nc* permutations in CP2 are identical to the ones in CP1 and the one that differs, is obtainable by interchanging exactly two literals in the permutation. Obviously, the closure of this (equivalence) relation is the space that we set out to search by simulated annealing. Consequently, we have characterized the simulated annealing process and the iterative loop choosing the c-permutations in the algorithm OPT can be replaced by the simulated annealing process.

An interesting open question is the incorporation of a polynomial time algorithm by superimposing some linearity property on the cost function for a recursive clique, as it was done for the conjunctive case in [KBZ 86].

8. Safety Problem:

Safety is a serious concern in implementing Horn clause queries. Any evaluable predicates (e.g., comparison predicates like x>y, x=y+y*z), and recursive predicates with function symbols are examples of potentially unsafe predicates. While an evaluable predicate will be executed by calls to built-in routines, they can be formally viewed as infinite relations defining, e.g., all the pairs of integers satisfying the relationship x>y, or all the triplets satisfying the relationship x=y+y*z [TZ 86]. Consequently, these predicates may result in unsafe executions in two ways: 1) the result of the query is infinite; 2) the execution requires the computation of a rule resulting in an infinite intermediate result. The former is termed the lack of *finite answer* and the latter the lack of *effective computability or EC.* Note that the answer may be finite even if a rule is not effectively computable. In this section we outline our approach with the emphasis on the interaction with the optimizer. For a more complete treatise on this topic see [KRS 87].

the optimizer is not less than this extreme value, a proper message must inform the user that the query is unsafe.

8.3 Comparison with Previous Work

The approaches to safety proposed in [Col 82, Nai 85, AN 86] is also based on reordering the goals in a given rule; but that is done at run-time by delaying goals when the number of instantiated arguments is insufficient to guarantee safety. This approach suffers from run-time overhead, and cannot guarantee termination at compile time or otherwise pinpoint the source of safety problems to the user -- a very desirable feature, since unsafe programs are typically incorrect ones. Our compile-time approach overcomes these problems and is more amenable to optimization.

The reader should, however, be aware of some of the limitations implicit in all approaches based on reordering of goals in rules. For instance a query

```
        p(x, y, z),  y= 2*x ?
```

on the rule

```
        p(x, y, z) <-- x=3, z=x*y
```

is obviously finite since the only answer is <x=3, y=6, z=18>. However, this answer cannot be computed under any permutation of goals in the rule. Thus both the approach given in [Col 82, Nai 85, AN 86] and the above optimization cum safety algorithm will fail to produce a safe execution for this query. Two other approaches, however, will succeed. One, described in [Za 86], determines whether there is a finite domain underlying the variables in the rules using an algorithm based on a functional dependency model. Safe queries are then processed in a bottom up fashion with the help of "magic sets", which make the process safe. The second solution consists in flattening, whereby the three equalities are combined in a conjunct and properly processed in the obvious order referred to earlier.

This example clarifies the drawbacks that follow from our expedient decision of not pursuing fllattening in the first version of the optimizer. Some flattening is being considered for later versions of the optimizer. Observe that, unlike previous approaches to control where such strategic decisions were wired-in into the system, an extension of the LDL optimizer to support flattening only requires adding another equivalence-preserving transformation.

[BMSU85] Bancilhon, F., D, Maier, Y. Sagiv and Ullman, Magic Sets and other Strange Ways to Implements Logic Programs, *Proc. 5-th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems,* pp. 1-16, 1986.

[BR 86] Bancilhon, F., and R. Ramakrishan, An Amateur's Introduction to Recursive Query Processing Strategies, *Proc. 1986 ACM-SIGMOD Intl. Conf. on Mgt. of Data,* pp. 16-52, 1986.

[BN 87] Beeri, C., S. Naqvi, R. Ramakrishnan, O. Shmueli, S. Tsur, Sets and Negation in a Logic Database Language, *Proc. 6-th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems,* 1987.

[Col 82] Colmemauer, A. et al., Prolog II: Reference Manual and Theoretical Model, Groupe d'Intelligence artificielle, Faculte de Sciences de Lumin, 1982.

[GM 82] Grant, J. and Minker J., On Optimizing the Evaluation of a Set of Expressions, *Int. Journal of Computer and Information Science, 11, 3 (1982), 179-189.*

[IW 87] Ioannidis, Y. E, Wong, E, Query Optimization by Simulated Annealing, *Proc. 1987 ACM-SIGMOD Intl. Conf. on Mgt. oof Data,* San Francisco, 1987.

[Kw 79] Kowalski, R.A., "Algorithm = Logic + Control", *CACM, 22, 7,* pp. 424-436, (1979).

[KBZ 86] Krishnamurthy, R., Boral, H., Zaniolo, C. Optimization of Nonrecursive Queries, *Proc. of 12th VLDB,* Kyoto, Japan, 1986.

[KRS 87] Krishnamurthy, R., R. Ramakrishnan. O. Shmueli, "A Framework for Testing Safety and Effective Computability", MCC Report 1987 and also submitted for external publication.

[KT 81] Kellog, C., and Travis, L. Reasoning with data in a deductively augmented database system, in *Advances in Database Theory: Vol 1,* H.Gallaire, J. Minker, and J. Nicholas eds., Plenum Press, New York, 1981, pp 261-298.

[Llo 84] Lloyd, J. W., *Foundations of Logic Programming,* Springer Verlag, 1984.

[M 84] Maier, D., *The Theory of Relational Databases,* (pp. 553-542), Comp. Science Press, 1984.

[MUV 86] K. Morris, J. D. Ullman and A. Van Gelder, Design Overview of the Nail! System, *Proc. Third Int. Symposium on Logic Programming,* pp. 127-139, 1986.

[Nai 85] Naish, L., Negation and Control in Prolog, Ph. D. Thesis, Dept. of CS, Univ. of Melbourne, Austr., 1985.

[NK 87] Naqvi Shamim and R. Krishnamurthy, Semantics of Updates in Logic Programming, Workshop on Database and Programming Languages, Roscoff, France 1987.

[Per 82] Pereira Luis Moniz, Logic Control with Logic, UNL Report 2/82 (1982).

[RBK 87] Ramakrishnan, R, C. Beeri, R. Krishnamurthy, Optimizing Existential Queries, MCC Technical Report, 1987, (also submitted for external publication).

COL: A LOGIC-BASED LANGUAGE FOR COMPLEX OBJECTS[1]

COL: un langage pour objets complexes
basé sur la logique

Serge Abiteboul Stéphane Grumbach

I.N.R.I.A.
78153 Le Chesnay, FRANCE

Abstract: A logic-based language for manipulating complex objects constructed using set and tuple constructors is introduced. Under some stratification restrictions, the semantics of programs is given by a canonical minimal and causal model that can be computed using a finite sequence of fixpoints. Applications of the language to procedural data, semantic database models, heterogeneous databases integration, and Datalog queries evaluation are presented.

1. This research was supported in part by the Projet de Recherches Coordonnées BD3.

(ii) In COL, data can be viewed both in a functional and in a relational manner. As a consequence, the language can be used in a heterogeneous databases context (e.g., relational view on a functional data base; integration of a relational database with a functional one).

(iii) COL can also be used as a kernel language for semantic database models like SDM [HM], IFO [AH] or Daplex [Sh].

(iv) Some evaluation techniques for datalog queries like Magic Sets or others [B+,GM] make extensive use of particular functions. These functions can be formalized using our model.

As mentioned above, two other approaches have been independently followed to obtain a rule-based language for complex objects [Be+,K]. In [Be+], they do not insist on a strict typing of objects. In [K], only one level of nesting is tolerated. However, both approaches could easily be adapted to the data structures considered in this paper. Furthermore, in [AB], it is argued that all these approaches yield essentially the same power (i.e., the power of the safe calculus of [AB]). The points (i-iv) above clearly indicate advantages of our approach.

The paper is organized as follows. In the first section, types and typed objects are described, and examples of COL rules given. The second section is devoted to the formal definition of the language. The stratification is introduced in Section 3. In the fourth section, it is shown that each stratified program has a canonical, causal and minimal model which can be computed using a sequence of fixpoints. Advantages of the language are briefly considered in a last section. The proof of key results of Section 4 can be found in an appendix.

I. PRELIMINARIES

In this section, types and typed objects are described, and examples of COL rules given.

The existence of some *atomic types* is assumed. A set of *values* is associated with each type A. This set is called the *domain* of A, and denoted dom(A). More complex types are obtained in the following way.

Definition: if $T_1,...,T_n$ are types $(n \geq 1)$, then

Intuitively, the functions $\cap$, $\cup$ and Difference define sets by stating explicitly what are the elements of each set. Thus the term $\cap$(X,Y) for instance is interpreted as the set of all the elements x such that $x \in X$, and $x \in Y$. Using these functions, the predicates $\subseteq$, $\subset$, Disjoint, Union, and Disjoint-union are now defined:

$$\subseteq(X,Y) \leftarrow \cup(X,Y) = Y,$$
$$\subset(X,Y) \leftarrow \subseteq(X,Y),\ x \in \text{Difference}(Y, X),$$
$$\text{Disjoint}(X,Y) \leftarrow \cap(X,Y) = \phi,$$
$$\text{Union}(X,Y,\cup(X,Y)) \leftarrow,$$
$$\text{Disjoint-union}(X,Y,\cup(X,Y)) \leftarrow \text{Disjoint}(X,Y).$$

The language allows the manipulation of complex objects, and also of "nested relations" [ABi,FT,JS,...] which are special cases of complex objects.

Example 2 (nested relations):

Let N denote the set of integers. Consider the predicate R(N,N,N) and the three predicates S(N,{[N,N]}), S'(N,{[N,N]}), S"(N,{[N,N]}). (The first field of S, S' and S" contains an integer, and the second a binary relation.) Let Z be a variable of type {[N,N]}; and F and TC be functions of the appropriate types.

Unnest:

$$R(x,y,y') \leftarrow S(x,Z),\ [y,y'] \in Z.$$

Nest:

$$[y,y'] \in F(x) \leftarrow R(x,y,y'),$$
$$S'(x,F(x)) \leftarrow R(x,y,y').$$

Transitive closure of the second field of S':

$$[x,z] \in TC(Z) \leftarrow [x,z] \in Z,$$
$$[x,z] \in TC(Z) \leftarrow [x,y] \in Z,\ [y,z] \in TC(Z),$$
$$S''(x,TC(Z)) \leftarrow S'(x,Z).$$

Example 3 (heterogeneous sets):

Let STRING be a type. Consider the following typed symbols:

- P({{N,STRING}}) (i.e., P is a unary predicate, and its unique field contains a set of sets of integers and strings).
- F is a function of type {N,STRING} $\rightarrow$ {N};

In the remainder of the paper, the word "function" will only refer to data functions, and not to tuple or set functions. It is assumed that all the functions that are considered in the following are set-valued, i.e., an image by a data function is always a set. In the last section, this limitation is discussed, and an extension of the language to remove it considered.

Note that $\in_{T,S}$ is a symbol of the language. Clearly, $\in_{T,S}$ is interpreted by the classical membership of set theory. Indeed, when the types are understood, $\in_{T,S}$ is simply denoted by $\in$. A constant of a certain type T is interpreted as an element of dom(T).

The terms of the language are now defined:

Definition: A constant or a variable is a *term*. If $t_1,...,t_n$ are terms and F is an n-ary data, tuple or set function symbol, $F(t_1,...,t_n)$ is a *term*. (The obvious restrictions on types are of course imposed.)

A *closed term* is a term with neither variables, nor data functions.

Example II.1: The term $[1,\{2,3\},\{7\}]$ is a closed term. On the other, $[1,\{2,3\},F(2)]$ is not closed. These two terms are different, but they may have the same interpretation (if $F(2) = \{7\}$).

Literals are defined by:

Definition: Let R be an n-ary predicate, and $t_1,...,t_n$ terms, for $n \geq 0$. Then (with the obvious typing restrictions) $R(t_1,...,t_n)$, $t_1 = t_n$, and $t_1 \in t_n$ are positive literals.

If ψ is a positive literal, $\neg\psi$ is a negative literal.

Arbitrary well-formed formulas are defined from literals in the usual way. We have defined here the language of a first order logic. One can define a model theory and a proof theory for this language. This is not in the scope of the present paper. We next introduce a clausal logic based on this first order logic. A key component of that clausal logic is the notion of "atom". An *atom* is a literal of the form $R(t_1,...t_n)$ or $t_1 \in F(t_2,...,t_n)$. If $t_1,...,t_n$ are closed terms, the atom is said to be *closed*.

Now we have:

In order to define the notion of satisfaction of a rule, and thus of a program, the concept of valuation is introduced. Valuations play here the role of substitution in classical logic programming. Note that the valuations are written on the left of the terms or atoms, for conveniencés sake.

Definition: Let θ be a ground substitution of the variables, and I an interpretation. The corresponding *valuation* θ_I is a function from the set of terms to the set of closed terms defined by[2]:

(i) θ_I is the identity for constants, and $\theta_I x = \theta x$ for each variable,

(ii) $\theta_I[t_1,...,t_n] = [\theta_I t_1,...,\theta_I t_n]$, $\theta_I\{t_1,...,t_n\} = \{\theta_I t_1,...,\theta_I t_n\}$, and

(iii) $\theta_I F(t_1,...,t_n) = \{ a \mid [a \in F(\theta_I t_1,...,\theta_I t_n)] \in I \}$.

The function θ_I is extended to literals by:

(iv) $\theta_I P(t_1,...,t_n) = P(\theta_I t_1,...,\theta_I t_n)$,

(v) $\theta_I(t_1=t_2) = (\theta_I t_1 = \theta_I t_2)$, $\theta_I(t_1 \in t_2) = (\theta_I t_1 \in \theta_I t_2)$, and

(vi) $\theta_I(\neg A) = \neg\, \theta_I A$.

A valuation in this context depends on the interpretation that is considered. This comes from the need to assign values to terms built using function symbols. As we shall see, this is a major reason for the non monotonicity of the operators that will be associated to COL programs.

Using valuations, we now define the notion of *satisfaction* of rules and programs:

Definition: The notion of satisfaction (denoted by $\models$) and its negation (denoted by $\not\models$) are defined by:

- For each closed positive literal, I $\models P(b_1,...,b_n)$ iff $P(b_1,...,b_n) \in I$; I $\models b_1 = b_2$ iff $b_1 = b_2$ is a tautology; and I $\models b_1 \in b_2$ iff $b_1 \in b_2$ is a tautology.
- For each closed negative literal $\neg$ B, I $\models \neg$ B iff I $\not\models$ B.
- Let $r = A \leftarrow L_1,...,L_m$. Then I $\models$ r iff for each valuation θ_I such that for each i, I $\models \theta_I L_i$, then I $\models \theta_I A$.

[2] The reader has to be aware of a subtlety in (iii). The symbol $\in$ in $[a \in F(\theta_I t_1,...,\theta_I t_n)]$ is a symbol of the language COL, whereas the other occurrence of $\in$ denotes the usual membership of set theory.

$$S(x,F(x)) \leftarrow R(X,y)$$

The symbol F is the defined symbol of the first rule; and S that of the second. The symbols R and F are determinants of the two rules.

To define the notion of stratification, we use the auxiliary concepts of "total" and "partial" determinants of a rule. We say that an occurrence of a determinant predicate P is *partial* in a rule if that occurrence arises in a positive literal. Similarly, the occurrence of a determinant function F in a positive literal $t_1 \in F(t_2,...,t_n)$ is said to be *partial.* A determinant is *partial* (in a rule) if all its occurrences are partial; a determinant is *total* otherwise.

For instance, consider the rule:

$$x \in F(G(y)) \leftarrow y \in H(x),\ R(x,y),\ \neg\ S(y,z),\ y \in H'(H'(x))$$

In that rule, F is the defined symbol. The symbols R and H are partial determinants, and the symbols S and G total determinants. The symbol H' has one total and one partial occurrence, and thus is a total determinant.

The distinction between total and partial determininant is quite natural. To derive a new atom using the previous rule it suffices to know some partial information on R and H (i.e., R(x,y) and $y \in H(x)$). On the other hand, S has to be completely known to be able to assert S(y,z). Similarly, H'(x) must be completely known.

Intuitively, if Y is defined by the rule, and X is a total determinant, then X must be "completely defined" before Y. This is denoted by $X < Y$. If X is only a partial determinant, then X must be defined no later than Y. This is denoted by $X \leq Y$. For each program P, a marked graph G_P is constructed as follows:

- the nodes of G_P are the predicate and function symbols of P,
- there is an edge from X to Y if $X \leq Y$, and
- there is a marked edge from X to Y if $X < Y$.

We are now ready to define the condition for stratification:

Definition: A program P is *stratified* iff the associated graph G_P has no cycle with a marked edge.

Remark: We have defined stratification of programs using both negation and data functions. As

IV. FIXPOINT SEMANTICS OF STRATIFIED PROGRAMS

In this section, the semantics of stratified programs is defined using canonical, minimal and causal models.

The following three well-known concepts are used:

- an operator T is *monotonic* if $I \subseteq J$ implies that $T(I) \subseteq T(J)$;
- I is a *fixpoint* of T, if $T(I) = I$; and
- I is a *pre-fixpoint* of T, if $T(I) \subseteq I$.

With each program P, we associate an *operator T_P* defined as follows:

Definition: Let P be a program, and I an interpretation of P. Then a closed term A is the *result of applying the rule* $A' \leftarrow L_1,...,L_m$ *with* a valuation θ_I if

- $I \models \theta_I L_i$ for each $i \in [1..m]$, and
- either $A' = P(t_1,...,t_n)$ and $A = P(\theta_I t_1,...,\theta_I t_n)$,
  or $A' = [t_1 \in F(t_2,...,t_n)]$, and $A = [\theta_I t_1 \in F(\theta_I t_2,...,\theta_I t_n)]$.

The operator T_P is defined by:

$$T_P(I) = \{ A \mid A \text{ is the result of applying a rule in P with some } \theta_I \}.$$

For a program P, T_P is not monotonic in general. For instance, consider the program P consisting of the single rule $Q(F) \leftarrow$. Then

$$T_P(\{1 \in F\}) = \{Q(\{1\})\} \not\subseteq \{Q(\{1,2\})\} = T_P(\{1 \in F, 2 \in F\}).$$

The following proposition links the notion of model of P to that of pre-fixpoint of T_P.

Proposition IV.1: Let P be a program, and M an interpretation of P. Then the next two statements are equivalent:

- M is a (minimal) model of P,
- M is a (minimal) pre-fixpoint of T_P.

Proof: It is clearly sufficient to prove that M is a model of P iff M is a pre-fixpoint of T_P.

- $T_P\uparrow\omega(I)$ is a minimal pre-fixpoint of T_P containing I.
- $T_P\uparrow\omega(\phi)$ is a minimal fixpoint of T_P.

This result shows that $T_P\uparrow\omega(\phi)$ can be viewed as a canonical model of the monostratum program P since by Proposition VI.2, it is a minimal causal model of P.

To prove that result, we will use three properties of monostratum programs. But, first, we introduce some notation which allows us to consider particular subsets of a given interpretation.

Notation: Let I be an interpretation, and X a set of predicate and data function symbols. We denote by $I|_X$ the following subset of I:

$$I|_X = \{P(a_1,\ldots,a_n) \in I \mid P \in X\} \cup \{[a_1 \in F(a_2,\ldots,a_n)] \in I \mid F \in X\}.$$

To prove Theorem IV.1, we shall show that monostratum programs are "growing", "X-finitary" and "stable on X" for some X.

Definition: Let P be a program and X a set of symbols. Then:

(1) T_P is *growing* [ABW] if for each interpretation I, J and M such that $I \subseteq J \subseteq M \subseteq T_P\uparrow\omega(I)$, then $T_P(J) \subseteq T_P(M)$.

(2) T_P is *X-finitary* if for each sequence (I_n) of interpretations such that for each n $(0 \leq n)$, $I_n \subseteq I_{n+1}$, and $I_n|_X = I_0|_X$, then $T_P(\bigcup_{n=0}^{\infty} I_n) \subseteq \bigcup_{n=0}^{\infty} T_P(I_n)$.

(3) T_P is *stable on X* if for each I, $(T_P(I))|_X \subseteq I|_X$.

The proof of Theorem IV.1, can be found in the appendix. Indeed, it is shown there that for some X, monostratum program are X-finitary and stable on X (Lemma A.2), that they are growing (Lemma A.3); and for each operator T with these three properties, and for each interpretation I,

(a) $T(T\uparrow\omega(I)) \subseteq T\uparrow\omega(I)$, and

(b) $T\uparrow\omega(I) \subseteq T(T\uparrow\omega(I)) \cup I$ (Proposition A.1).

Theorem IV.1 is then a consequence of these results (see Appendix).

The proof of Theorem IV.2 can also be found in the appendix.

This is the main result for COL programs. It is interesting to note that negation can be simulated using data functions. Let P be a predicate. The following program gives an equivalent form of $\neg$ P.

$$t \in F(t) \leftarrow P(t),$$
$$A(t, F(t)) \leftarrow ,$$
$$Q(t) \leftarrow A(t,\{\}).$$

It is easy to see that Q(t) is equivalent to $\neg$ P(t). Consider the stratification condition imposed by the previous program. From the first rule, $P \leq F$; from the second, $F < A$, and from the third, $A \leq Q$. As a consequence, $P < Q$ which leads to the classical notion of stratification for negation.

V. DISCUSSION

In this section, we briefly consider some applications and extensions of the language. More precisely, we illustrate the following points:

(i) procedural data;
(ii) heterogeneous databases (functional and relational);
(iii) semantic database models; and
(iv) evaluation techniques for datalog queries.

During the presentation, we encounter various extensions of the language which are left for future research.

V.1 Procedural Data

One of the reasons for considering a functional database model versus a relational one is to remove the dichotomy between data and queries. The removal of that dichotomy is also the motivation for introducing procedural fields in Postgres [S]. However, if the procedural fields solution is interesting as being an extension of the popular relational model, it certainly lacks the elegance of the functional solution. We believe that COL presents the advantages of both approaches by first being a relational extension, and also by making explicit use of functions to handle procedural-like data. The purpose of this section is to briefly investigate this issue.

Procedural data is introduced in [S] in order to blur the dichotomy between data and where the HOB_BOSS function is defined by:

$$x \in \text{HOB\_BOSS}(y) \leftarrow R(y,z,X),\ x \in \text{HOB}(z).$$

The above program is also not stratified. Indeed, it is not even locally stratified according to [P]. The complex structure of facts should also be taken into account. For instance, two objects, say A and B, may be both intensionally defined with a subobject of each one of them depending on a subobject of the other.

V.2 Heterogeneous databases

We show how to integrate a relational database, and a functional one into a COL database. It is also possible to use a similar approach to define heterogeneous views when relations and functions are considered, and to restructure a relational database into a functional one, or conversely.

The main problem encountered in this context is that functional database models like FQL [BF] or Daplex [Sh] allow monovalued functions. A not too clean solution is to represent them using multivalued ones and enforce a oneness constraint. A more interesting solution is to extend the language with monovalued data functions. Rules like

$$x = F_1(y) \leftarrow R(x,y), \text{ and}$$

$$x = F(y) \leftarrow R(x,y),\ y = H(x)$$

have to be considered. The first rule yields inconsistency if in the extension of R, the first attribute does not functionally determine the second one. This can not be the case in the second rule. In both rules, the derived function may be only partially defined.

We now present an example with multivalued functions only. Consider the following two databases:

(a) *A RELATIONAL DATABASE:*

SHOW(film,theater,time)
PLAYS(actor,film)
LOCATION(theater,address)

(b) *A FUNCTIONAL DATABASE*

CASTING: film $\twoheadrightarrow$ actor
LOCATED: theater $\twoheadrightarrow$ address
EXHIB: film $\twoheadrightarrow$ theater, time

schema is shown in Figure V.1. We present a corresponding COL database, and then discuss the extensions of the language that need to be considered, and the limitations of the COL representation:

ABSTRACT TYPES are represented by basic domains:

hull

car

person

motor

manufacturer

CONSTRUCTED TYPES are represented by base objects:

MOTORBOAT(hull,motor)

CAR-ID(string,integer)

Figure V.1: an IFO schema

functional equation:

$$F_{ANC} = F_{PAR} + F_{PAR} \circ F_{ANC}$$

where "+" stands for union and " $\circ$ " for the composition of multivalued functions.

In another proposal for evaluating datalog queries [B+], namely the magic sets approach, particular terms called "grouping terms" are used. It is easy to see that these terms correspond to particular derived data functions.

VI. CONCLUSION

The paper presents a language to manipulate complex objects based on recursive rules. The novelty is the use of data functions. The semantics of COL programs is defined as a canonical causal and minimal model using a sequence of fixpoint operators. In that sense, the semantics is constructive in nature.

We illustrated the use of the language in various database contexts: heterogeneous databases, semantic modelling, procedural data, and evaluation of datalog queries. This suggested extensions of the language: single-valued functions, explicit union of types constructor, structural stratification. Besides these issues which were just sketched in the present paper, other important questions are raised:

- the role of inheritance in the language, and
- updates for COL databases.

Last but not least remains the issue of an efficient implementation. There has been a lot of work on nested relations and complex objects. Few of them have so far been followed by an efficient implementation (e.g., the Verso system at Inria [V], and the Aim project at IBM Heidelberg [D]). We believe that the fixpoint semantics of COL programs makes such an implementation feasible. Indeed, the operators which are described in Section 4 can all be expressed in the algebra of complex objects of [AB].

ACKNOWLEDGMENT:

[D] Dadam, P., History and Status of the Advanced Information Management Prototype, Proc. International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt (1987)

[FT] Fischer, P., and Thomas, S., Operators for non-first-normal-form relations, Proc. 7th COMPSAC Chicago,(1983).

[GM] Gardarin G., C. de Maindreville, Evaluation of Database Recursive Logic Programs as Recurrent Function Series, proc. ACM SIGMOD conf. on Management of Data (1986)

[G] Van Gelder, A., Negation as Failure Using Tight Derivations for General Logic Programs, Proc. of Workshop on Foundations of Deductive Database and Logic Programming (1986)

[HK] Hull, R., R. King, Semantic database modeling: Survey, applications, and research issues. U.S.C. Computer Science Technical Report (1986) to appear in ACM computing surveys

[HY] Hull, R., C.K. Yap, The format model: A theory of database organization. Journal of the ACM 31(3) (1984)

[J] Jacobs, B., on Database Logic, *Journal of the ACM* (1982).

[JS] Jaeschke, B., H.J. Schek, Remarks on the algebra of non first normal form relations, Proc. ACM SIGACT/SIGMOD Symposium on Principle of Database Systems, Los Angeles (1982)

[Ko] Kobayashi, I. "An overview of database management technology," TR CS-4-1, Sanno College, KAnagawa 259-11, Japan, (1980).

[K] Kuper, G.M., Logic Programming with Sets, Proc. ACM SIGACT/SIGMOD Symposium on Principle of Database Systems (1987)

[N] Naqvi, S.A., A Logic for Negation in Database Systems, Proc. Workshop on Foundations of Deductive Databases and Logic Programming ed. J. Minker (1986)

[P] Przymusinski, T. C. On the Semantics of Stratified Deductive Databases and Logic Programs, to appear in *Journal of Logic Programming*

[SS] Schek H., and M. Scholl, the Relational Model with relation-valued attributes, in *Information Systems* (1986)

[S] Stonebraker M., Object Management in Postgres using Procedures, in the Postgres Papers, UCB report (1986)

[Sh] Shipman, D., The Functional Data Model and the Data Language Daplex, *ACM*

APPENDIX

In this appendix, Theorems IV.1 and IV.2 are proven.

To prove Theorem IV.1, we first show that each monostratum program is growing, X-finitary and stable on X, for some X. To do that, we use the following technical lemma:

Lemma A.1: Let J and K be two interpretations such that $J|_X = K|_X$ for a given set X of symbols, and θ_J and θ_K two valuations with $\theta_J x = \theta_K x$ for each variable x. If t is a term such that each function symbol occurring in t belongs to X, then $\theta_J t = \theta_K t$.

Proof: The result is obvious if t contains no function symbols. Now consider $t = F(t_1,\ldots,t_n)$ where F is in X and $t_1,\ldots,t_n$ contain no function symbol. Then

$$\begin{aligned}
\theta_J F(t_1,\ldots,t_n) &= \{x \mid [x \in F(\theta_J t_1,\ldots, \theta_J t_n)] \in J\}, \text{ by definition,}\\
&= \{x \mid [x \in F(\theta_J t_1,\ldots, \theta_J t_n)] \in J|_X\}, \text{ since F is in X,}\\
&= \{x \mid [x \in F(\theta_J t_1,\ldots, \theta_J t_n)] \in K|_X\}, \text{ since } J|_X = K|_X,\\
&= \{x \mid [x \in F(\theta_K t_1,\ldots, \theta_K t_n)] \in K|_X\}, \text{ since } t_1,\ldots,t_n \text{ contain no function symbol,}\\
&= \{x \mid [x \in F(\theta_K t_1,\ldots, \theta_K t_n)] \in K\}, \text{ since F is in X,}\\
&= \theta_K F(t_1,\ldots,t_n).
\end{aligned}$$

By induction of the imbrication of function symbols, $\theta_J t = \theta_K t$ for each term t containing only function symbols in X. □

We now consider X-finitarity and stability.

Lemma A.2: Let P be a monostratum program, and X the set of symbols in P which are not defined in P. Then T_P is X-finitary and stable on X.

Proof: Consider first stability on X. For each interpretation I of P, $T_P(I)$ contains only atoms that are built from a defined symbol. Thus $(T_P(I))|_X = \phi \subseteq I|_X$, so T_P is stable on X.

We next prove that T_P is X-finitary. Let (I_n) be a growing sequence of interpretations such that $I_n|_X = I_0|_X$ for all n. Let $J = \bigcup_{n=0}^{\infty} I_n$, and let $A \in T_P(J)$. To conclude the proof, it suffices to

Proof: Let P be a monostratum program. Let I, J, M be interpretations such that $I \subseteq J \subseteq M \subseteq T\uparrow\omega(I)$. We prove that if $A \in T_p(J)$, then $A \in T_p(M)$.

Suppose that $A \in T_p(J)$. Then A is the result of applying the rule $r : A' \leftarrow L_1,...,L_m$ in P with a valuation θ_J. Let θ_M be a valuation such that $\theta_M x = \theta_J x$ for all variables x.

Let X be the set of symbols that are not defined in P. Clearly, $I|_X \subseteq J|_X \subseteq M|_X \subseteq (T_P\uparrow\omega(I))|_X = I|_X$. Let $t_1 \in F(t_2,...,t_n)$ or $P(t_1,...,t_n)$ be an atom in rule r, and let $i \in [1..n]$. Each function symbol G appearing in t_i is a total determinant, and thus is not defined since P is monostratum. Since $J|_X = M_X$, $\theta_M t_i = \theta_J t_i$ by Lemma A.1. Thus

(+) $\theta_M t_i = \theta_J t_i$, for each atom $t_1 \in F(t_2,...,t_n)$ or $P(t_1,...,t_n)$ in rule r, and each $i \in [1..n]$.

We prove that $M \models \theta_M L_i$ for each i. Like in the previous lemma, there are four cases. We consider here the last case only. The others are left to the reader.

(4) Let $L_i = \neg [t_1 \in F(t_2,...,t_n)]$. Since A is the result of applying the rule with θ_J, $J \models \theta_J L_i$. Thus $[\theta_J t_1 \in F(\theta_J t_2,...,\theta_J t_n)] \notin J$. Thus, by (+), $[\theta_M t_1 \in F(\theta_M t_2,..., \theta_M t_n)] = [\theta_J t_1 \in F(\theta_J t_2,...,\theta_J t_n)] \notin J$. Let $B = [\theta_M t_1 \in F(\theta_M t_2,..., \theta_M t_n)]$. Since $B \notin J$, $B \notin J|_X = M|_X$. Since the literal is negative, F is a total determinant of P. Thus F is not a defined symbol of P (P is monostratum), i.e., $F \in X$. Hence $B \notin M$. Therefore, $[\theta_M t_1 \in F(\theta_M t_2,..., \theta_M t_n)] \notin M$, i.e., $M \models L_i$.

In each case, $M \models \theta_M L_i$. Let A" be the result of applying rule r with θ_M. By (+), A" = A. Thus $A \in T_P(M)$. □

The following proposition will be essential in the proof of Theorem IV.1.

Proposition A.1: Let T be an X-finitary, stable on X, and growing operator. Then for all I,

(a) $T(T\uparrow\omega(I)) \subseteq T\uparrow\omega(I)$, and

(b) $T\uparrow\omega(I) \subseteq T(T\uparrow\omega(I)) \cup I$.

Proof: First consider (a). Since T is stable on X,

$$(T\uparrow(n+1)(I))|_X = (T\uparrow n(I))|_X = (T\uparrow 0(I))|_X.$$

Thus the sequence $(T\uparrow n(I))$ is growing and $(T\uparrow n(I))|_X = (T\uparrow 0(I))|_X$. By the X-finitarity of T, of a sequence of operators, and the locality property [ABW].

Definition: Let $T_1,\ldots,T_m$ be a sequence of operators. The **iterative powers** of that sequence w.r.t. an interpretation I are defined by:

- $K_0 = I$, and
- $K_i = T_i\uparrow\omega(K_{i-1})$ for each $i \in [1..m]$.

The sequence of operators $T_1,\ldots,T_m$ is **local**, if for each I and J such that $I \subseteq J \subseteq K_m$, $T_i(J) = T_i(J \cap K_i)$.

Let $P = P_1 \uplus \ldots \uplus P_m$ be a stratified program. With the first stratum, we associate an operator T_1; with the second one, an operator T_2; and so on. Then we have:

Lemma A.4: Let $T_1,\ldots,T_m$ be the sequence of operators corresponding to a stratified program $P = P_1 \uplus \ldots \uplus P_m$. This sequence is local.

Proof: First suppose that $T_i(J) \not\subseteq T_i(J \cap K_i)$ for some i. Let A be in $T_i(J) - T_i(J \cap K_i)$. Then A is the result of applying some rule r in P_i. Since $J \cap K_i \subseteq J$, and $A \notin T_i(J \cap K_i)$, the application of the rule uses a fact B not in $J \cap K_i$. Suppose that $B = [b_1 \in F(b_2,\ldots,b_n)]$. (The case $B = P(b_1,\ldots,b_n)$ is similar). Since B is used in the application of r,

(i) F is a determinant of r in P_i.

Since B is in $K_m - K_i$, B is the result of the application of a rule r' in P_j for some $j > i$. Thus

(ii) F is the defined symbol of a rule r' in P_j for $j > i$.

Clearly, (i) and (ii) together contradict the stratification condition on $\mathbf{P}_1 \uplus \ldots \uplus \mathbf{P}_{i-1}$. Hence, $T_i(J) \subseteq T_i(J \cap K_i)$. The reverse inclusion is proved in a similar way. □

Theorem IV.2 will be a straightforward consequence of the following proposition:

Proposition A.2: Let $T_1,\ldots,T_m$ be a local sequence of operators such that for each $i \in [1..m]$, T_i is growing, X_i-finitary and stable on X_i, for some X_i. For each instance I, let (K_i) be the iterative powers of $T_1,\ldots,T_n$ w.r.t. I. Then[3]

[3] By definition, $(\bigcup_{i=1}^{m} T_i)J = \bigcup_{i=1}^{m} (T_iJ)$.

Then K_m is a minimal fixpoint of $\bigcup_{i=0}^{m} T_i$. Thus K_m is a minimal causal model of P.

Proof: By Proposition IV.2, it suffices to show that K_m is a minimal fixpoint of $\bigcup_{i=0}^{m} T_i$. By Lemma A.4, the sequence of operators is local. Thus, by Proposition A.2,

(1) $(\bigcup_{i=1}^{m} T_i)K_m \subseteq K_m$,

(2) $K_m \subseteq (\bigcup_{i=1}^{m} T_i)K_m$.

Therefore, K_m is a fixpoint of $\bigcup_{i=0}^{m} T_i$. It remains to show the minimality.

Let J be a pre-fixpoint of $\bigcup_{i=1}^{m} T_i$. We prove by induction on k that

$$(*) \text{ if } J \subseteq K_k \text{, then } K_k \subseteq J.$$

For $k = 0$, $K_0 = \phi \subseteq J$. Suppose (*) is true for a certain k (first induction hypothesis). We prove by induction that :

$$(**)\ T_{k+1} \uparrow j(K_k) \subseteq J,$$

For $j = 0$, it is by hypothesis. Suppose it is true for a certain j (second induction hypothesis). By (**), $K_k \subseteq T_{k+1}\uparrow j(K_k) \subseteq J \cap K_{k+1} \subseteq T_{k+1}\uparrow\omega(K_k)$. Since T_{k+1} is growing,

$$(+)\ T_{k+1}\ (T_{k+1}\uparrow j(K_k)) \subseteq T_{k+1}\ (J \cap K_{k+1}).$$

Hence, $T_{k+1} \uparrow(j+1)(K_k) = T_{k+1}\ (T_{k+1}\uparrow j(K_k)) \cup T_{k+1}\uparrow j(K_k)$, by definition,

$\subseteq T_{k+1}(T_{k+1}\uparrow j(K_k)) \cup J$, by second induction hypothesis,

$\subseteq T_{k+1}(J \cap K_{k+1}) \cup J$, by (+),

$= T_{k+1}(J) \cup J$, by locality,

$\subseteq J$, since J is a pre-fixpoint of T_{k+1}.

Thus (**) holds for all j. By induction, (*) holds for all k. In particular, for $k = m$, if J is a pre-fixpoint of $\bigcup_{i=1}^{m} T_i$ such that $J \subseteq K_m$, then $K_m \subseteq J$ which concludes the proof. □

to be placed over it.

2. What Makes a Database Computation Model Powerful?

I claim that the power of relational algebra as an abstraction of disk storage comes from its encapsulation of iteration. Seven or eight common forms of iteration over sets of records are identified, and queries are expressed in terms of them. Since there are a small number of forms, their interactions can be studied in detail, giving rise to transformations that can be used for optimizing queries. Effort can be directed at efficient implementation of this handful of iteration forms. Since the iteration is expressed at a high level, multiple orders for accessing records are allowable, and the physical ordering of records and foreknowledge of access patterns can be used to great advantage. Further, use of auxiliary access structures can be embedded in the evaluation methods for the algebraic operators, making applications independent of the presence or absence of such structures, and simplifying that code. A query processor can delay choosing a particular evaluation plan for an algebraic expression until the nature of the arguments is known, allowing even more efficiencies in execution. None of these advantages is available when database manipulations are expressed with explicit looping structures. The resulting code gives a particular implementation of the query, from which it is nearly impossible to infer the intent. Thus, the range of transformations and evaluation choices is severely limited. Moreover, the record-at-a-time nature of explicit iterations places high demands on the communication bandwidth between the application program and the database system.

I expect the next generation of database systems to reside on a network of workstations, with a central or distributed storage manager, shared by application programs over the network. Here, the importance of being able to express iterations and other data-intensive operations succinctly is even greater. Whatever the database programming model, it must allow complex data-intensive operations to be picked out of programs for execution by the storage manager, rather than forcing a record- or object-at-a-time interface. As mentioned in the introduction, the definition of a complex operation should be storable as a database object, so its

4. Embedded DML

I doubt many of us believe that an embedded data manipulation languages is best strategy for database programming. The problems with this approach are manifest, the most serious being *impedance mismatch* at the interface of the application language and the DML. The programming paradigms of the two languages are frequently at odds, as are the data structures supported. Much information is reflected back at the junction of the two. There is no type system spanning the application code and the DML, so little checking can be done on type agreement across the junction. The persistent programming approach does have the advantage of a single type system. However, the type systems of most languages were not conceived with persistent data in mind, particularly the difficulties in modifying type definitions when instances of those types persist.

It is interesting to observe how 4GLs and application generators deal with this typing problem. They generate the application code working off the type definitions of the database (the scheme or an extension of it), trying to ensure agreement between the types of database objects and their uses in the application code. The code is generated to be type correct, but still typing across the boundary can't be checked.

5. Extending the Application Language

Another approach to capturing the high-level operators on data in the application language is to extend an imperative language with associative access constructs [M+]. The problem with this approach is that the resulting language is quite complex, and probably lacks orthogonality and transparency. The language ends up with multiple ways to do the same thing (but with only one amenable to optimization) and there are limitations on embedding imperative statements in the declarative extension. Supporting such an extension also means having to modify the parser for the original language.

What I need to know for maintaining an index on rectangles is that there is some message pair, say "origin" and "setOrigin:" that have a certain specification on their interaction: two invocations of "origin" return the same result, as long as there is no intervening "setOrigin:" invocation. In essence, I want to say there is a "origin" field in the Rectangle. I more or less want to dispense with the encapsulation and know about the structure of Rectangle objects. Must encapsulation just go out the window?

Some data models get around this problem by saying that certain structural aspects of the object are visible externally, as *components* [Sc+] or *properties* [Ont]. We all know that a jet has engines, so let's just admit it in the protocol. Indexing, if allowed only on these visible subobjects, is supportable without violating encapsulation. An interesting kink arises in Trellis/OWL, however. The implementation of a component may be specified as "field," meaning represent the component as a field in the object's private state, and do get and set in the obvious way. However, the get and set operations can also be implemented with arbitrary code, in which case there are no guarantees they will exhibit behavior necessary for index maintenance.

7. Abstract Objects

I propose here the notion of an *abstract object* as a basic building block for database programming objects. My approach is colored by experience with object-oriented databases. In particular, I assume a data model with complex objects having identity, where objects can be shared subparts of other objects. An abstract object acts much as a term or pattern in a logic language, and it can be used both for decomposing and building concrete objects, much as a logical term acts under unification. However, abstract objects are objects, so they can be created, stored, manipulated and viewed just as concrete database objects. They can be composed to create compound queries and manipulation commands. These abstract objects are very structural in nature, but they do possess a formal semantic theory built on a logic that incorporates identity and type hierarchies [Ma, Zh].

tual approximation of the actual command object.) Abstract objects can also be used for updates. If I wanted to update the original Rectangle in the RectSelect, instead of making a new one, I would write

```
Rectangle:R(origin --> Point:P1,
            corner --> Point:P3) <==
      RectSelect:RS(rect --> Rectangle:R(
                           origin --> Point:P1(x --> 0),
                           corner --> Point:P2(y --> Int:N))
                     cursor --> Point:P3(x --> Int:M, y --> Int:N)).
```

If I wanted to merely modify the corner point of :R, rather than replace it, I would use

```
Point:P2(x --> :M) <==
      RectSelect:RS(rect --> Rectangle:R(
                           origin --> Point:P1(x --> 0),
                           corner --> Point:P2(y --> Int:N))
                     cursor --> Point:P3(x --> Int:M, y --> Int:N)).
```

Or, I could create a new point for the corner for :R with the same coordinates as :P3.

```
Rectangle:R(corner --> Point:P4(x --> :M, y--> :N)) <==
      RectSelect:RS(rect --> Rectangle:R(
                           origin --> Point:P1(x --> 0),
                           corner --> Point:P2(y --> Int:N))
                     cursor --> Point:P3(x --> Int:M, y --> Int:N)).
```

I can also introduce computation into commands

```
Point:P2(x --> :L - :M) <==
      RectSelect:RS(rect --> Rectangle:R(
                           origin --> Point:P1(x --> 0),
                           corner --> Point:P2(x --> Int:L, y --> Int:N))
                     cursor --> Point:P3(x --> Int:M, y --> Int:N)).
```

The important facet of such a command is that its processing can be separated into structural matching and making phases, with an intervening computational "mapping" phase. Such simple commands can be grouped and named to create compound commands.

8. Ramifications and Extensions

Some advantages that accrue from using abstract objects as the building blocks of database commands:

multiple ways.

2. For a command object, what are strategies for evaluating portions of it on different processors? For example, the structural access could be done on a central storage server, and the computational part on a local workstation.

3. I don't think abstract objects are quite equivalent to logical variables. (I don't see how to unify two abstract objects.) I think objects with logical variables would be useful for expressing and constraining partially defined objects and for representing alternative configurations or versions of an object. Perhaps the ability to store a name from a binding environment in place of a value would give equivalent power [AM].

9. References

[A-K]
H. Ait-Kaci
A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures
Ph.D. Thesis, University of Pennsylvania, 1984

[At+]
M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, R. Morrison
An approach to persistent programming
The Computer Journal 26:4, 1983

[AM]
M. P. Atkinson, R. Morrison
Types, bindings and parameters in a persistent environment
Proceedings of Data Types and Persistence Workshop
Appin, August 1985

[AMP]
M. P. Atkinson, R. Morrison, G. D. Pratten
Persistent information architectures
Univ. of Glasgow/Univ. of St. Andrews
Persistent Programming Research Report 36, June 1987

[KK]
T. Kaehler, G. Krasner
LOOM—large object-oriented memory for Smalltalk-80 systems
In Smalltalk-80: Bits of History, Words of Advice
Addison-Wesley, 1983

[Ma]
D. Maier
A Logic for Objects
Oregon Graduate Center TR CS/E-86-012, November 1986
Presented at the Workshop on Deductive Databases and Logic Programming
Washington, DC, August 1986

Data and Knowledge Model: A Proposal

Maurice A.W. Houtsma   Peter M.G. Apers

University of Twente
P.O. Box 217
7500 AE Enschede
the Netherlands

October 23, 1987

Abstract

The need to enhance databases with facilities to store and manipulate knowledge is growing. Currently, most of the knowledge is hardwired in the applications running on top of a database. Changing the knowledge may require rewriting applications, possibly several because of redundancy. Obviously, there is a need to have one repository of knowledge, preferably represented in a declarative form. Several papers on interfacing Prolog with a relational database have been presented. In this architecture, knowledge is represented in Prolog and data in the database. Drawbacks are: inefficient query processing and an artificial separation of modeled data and knowledge. Currently, at the University of Twente there is a research effort on integrating knowledge and data. This proposal reports on ongoing research on this topic.

Several knowledge representations exist, most of them lacking modularity. Therefore, for our Data and Knowledge Model (DK model) the Entity-Relationship model is chosen as a basis, although we expect our ideas to hold as well for other semantic data models. Below we will discuss the main features of the DK model.

The DK model consists of entities, relationships, and ISA-links. The latter are used to represent generalization. An *entity* consists of three parts: attributes, rules, and constraints. An *attribute* may be an ordinary attribute as in the ER model or it may be a virtual attribute defined in the rule part. Queries addressing attributes will not notice the difference. Attributes may be complex in the sense that they are structured or they may represent sets.

- A model should capture the semantics of the world modeled at the right abstraction level.
- It should support modularity.
- It should provide a good way of communication between the database designer and the user.
- It should allow for dynamic components, like virtual attributes.
- Knowledge should be specified in a form that is clear to the designer and the user of the system, eases explanation, and is easily modifiable.
- The processing of queries should be handled efficiently.

In the development of our DK model we start from a database point of view. This has the advantage of a strong theoretical background, availability of well-founded semantic data models and the availability of large, reliable, multi-user systems. Our model is able to treat data and knowledge in a uniform and powerful way.

We use the Entity-Relationship model [**5**] as a basis for our ideas about integrating data and knowledge in one model. The main reason is that it is a well developed semantic data model that fulfills the first three requirements mentioned above. However, we would like to stress that we expect our ideas to hold for other semantic data models like e.g. [**12**] as well.

Besides normal attributes we allow virtual attributes in our model. These attributes are not associated with a value, but with a rule that describes how to compute a value for these attributes. However, queries addressing the attributes do not notice the difference between normal attributes and virtual attributes.

The rules that give definitions for virtual attributes are specified in the form of Datalog-like clauses. This declarative way of specifying knowledge rules eases the support of explanation facilities. Moreover, the optimization of the handling of knowledge is not visible to the user but remains a separate part of the system. By storing the knowledge rules with the associated entity the modularity of the system is ensured. Finally, the model we have in mind can be mapped onto the Relational Algebra extended with recursion [**1**, **2**, **3**]. This will guarantee an efficient processing of queries.

1.3 Entities

As described above entities consist of three parts; attributes, rules and constraints. Each will be described now.

1.3.1 Attributes

As said before, data by its nature is structured. Data are the basic facts we know about real-world entities. Or at least the basic facts we think important enough to model. For instance, the data of employees could be: employeenumber, name, address, job, department, salary. These basic facts are represented by attributes in a data model.

Relational database systems are concentrated around structured data: relations. With the introduction of the relational data model strategies were developed to structure these data. The process of normalization structures data in a way to avoid redundancy, and thereby ambiguity problems. Once these structures are defined, they are intended to be stable over a long period of time. The reason is that the Universe of Discourse, which is reflected in the data structures, does not change very often. But the contents of the database can change very frequently.

By the influence of new applications on database systems other demands are made towards a data model. From CAD/CAM there arises a need to have complex objects and sets as datatype, in modeling reality. Therefore attempts are made to extend existing models with these requirements [**11**], or define new models that support them [**9**, **10**].

Besides requirements that arise through new applications, some existing extensions have to be dropped. Especially, because of the influence of logic on databases, handling null values is not clearly understood. Besides that, at the moment there is no good semantics for null values. Of course, in e.g. SQL there is a way of dealing with null-valued attributes, but this seems to be rather ad hoc and there is certainly no clear semantics behind it.

Another restriction being made in present data models is that every attribute of a tuple should have a fixed value. We allow so-called virtual attributes in our model. This means that instead of a value being associated to an attribute, we also allow a rule to be associated with an attribute. This rule has to describe how to compute a value for the attribute. Such rules should be given amongst the other rules associated with an entity, in the rule part. This part will be described in the next section.

So, we have seen that the modeling of structured data is well developed and can be captured by the notion of attributes. Recently, sets and complex stated that rules are Horn clauses, we will now make this more explicit. A rule consists of a head and a body. The predicate name in the head of a rule is the name of the virtual attribute the rule describes. The first argument of this predicate is a variable denoting the key value of the entity under consideration. When the entity has a multiple key the first arguments of the predicate will correspond with the key. The last argument of the predicate in the head denotes the value assigned to the virtual attribute.

The body of a rule describes how to compute the value for a virtual attribute. It consists of predicates, comparison operations and simple arithmetic. We will now describe the mapping of the body of a rule onto other components of our model.

Predicates in the body of a rule can map onto different types of components. They can map onto attributes, virtual attributes and relationships. The mapping between a predicate and an attribute is equal to the mapping described above. The predicate name denotes the attribute involved, the first argument(s) the key of the entity and the last argument the value of the attribute. This is exactly the same for virtual attributes, because they behave just like normal attributes.

A predicate maps onto a relationship if their names correspond. For simplicity there is an order on the entities involved in a relationship. This means the first argument(s) denotes the key of the first entity involved in the relationship. The last argument(s) consequently denotes the second entity involved the relationship.

Now that we have defined the matching between predicates and the other components of the model we will describe how to form meaningful rules with them. In other words, we will describe the semantics of rules.

The main question is what predicates are allowed in the body of a rule. First of all an object can address its own attributes and virtual attributes in the body of a rule, as described before. But it can also address attributes and rules from elsewhere in the ISA-hierarchy. It will be allowed to address components of objects higher in the hierarchy (more general objects), but also components lower in the hierarchy (more specialized objects). In this last case it is not guaranteed that a value will be found, an object can be specialized but this is not mandatory. Modeling will be very important: the careful placement of rules. Attributes and rules will be placed as high in the hierarchy as possible, at a most general place.

Besides with attributes and rules from elsewhere in the ISA-hierarchy, predicates can also match with attributes and virtual attributes from anywhere in the system. However, it is mandatory that the objects that own

1.4 Relationships

Besides entities there appear of course also relationships in our model. The sole reason for their existence is to connect entities, and therefore they can be of a very simple nature. As soon as there is a need to let them have (virtual) attributes, they should be made into entities.

We only model binary relationships, as is done by most people. To simplify query processing we suppose an order imposed on the relationship. This makes role names spurious. We do take into account the number of times a particular entity can appear in a certain relationship. This models the type of relationship (one-to-one, one-to-many, many-to-many) and can help answering some queries. As already discussed when describing the rule part of entities, entities are represented by their key when appearing in a relationship.

As opposed to semantic networks we do not take into account different type of links. Relationships and ISA-links (discussed in the next section), are the only type of links we consider. Of course, having different type of links can sometimes supply extra information, but it can also lead to confusion. When traversing long paths of various type of links the semantics can become very unclear. Also, the inference process becomes more difficult, having to take into account which links can be traversed when, and what is their exact meaning.

1.4.1 Constraints

Although relationships are not allowed to have associated (virtual) attributes, they are allowed to have associated constraints. The reason is that many constraints are inherently part of relationships and are not in the right place when put inside entities. Constraints associated with relationships are syntactically exactly the same as those associated with entities.

Again, let us stress the fact that constraints are not enforced by the system as they are specified here. They are only used to process queries, and can in fact be looked upon as a kind of pre-deduced information about entities involved in the relationships.

1.5 ISA-links

In our model, ISA-links are used to model generalization/specialization. We believe, as many others, that it is a desirable concept. From a specification

```
      field:STRING;
   RULES
      knows_of(ENR, {TOPIC}) ← visits(ENR, CNAME) &
              appears_in(CNAME, TOPIC) & field(TOPIC, F) ∧ field(ENR, F).
END Researcher
```

The rule states that researchers know about all topics that have been presented at a conference they visited, where the topic is inside their own field. The curly brackets mean that all values of the variable TOPIC that are found, are gathered in one set. When these brackets are not used, a set of (enr, topic_name)-tuples will be presented to the user. Now, the rule results in a set of (enr, {topic_name})-tuples.

```
ENTITY Associate ISA Researcher
   ATTRIBUTES
   date_of_hire:DATE;
   duration_of_contract:{2, 4};
   RULES
      knows_of(ENR, TOPIC) ← manages(PROF_ENR, ENR) ∧
              knows_of(PROF_ENR, <TOPIC>).
END Associate
```

The rule states that an associate knows about all topics that the professor who manages him knows about. Here the brackets around TOPIC denote that it is a variable that denotes a set of values. Therefore, the result of this rule will also be a set.

```
ENTITY Professor ISA Researcher
   ATTRIBUTES
      status:STRING;
   RULES
      knows_of(ENR, {TOPIC}) ← prog_comm(ENR, CNAME) ∧
              appears_in(CNAME, TOPIC).
   CONSTRAINTS
      salary > 80,000
END Professor
```

```
    BETWEEN Professor, min:3, max:20
    AND Associate, min:1, max:1;
    CONSTRAINTS
    manages(PNR, ANR) ∧ salary(ANR, X) ∧ salary(PNR, Y) ∧ X < Y.
END
```

Here we see an example of a constraint associated with a relationship. It states that the salary of a professor should exceed the salary of the associates he manages.

As can be seen from the example above the use of ISA-links eases the modeling. All (virtual) attributes from generalizations are inherited by specializations. Therefore it is e.g. not necessary to give a key to professor, this key is already inherited from researcher. It can also be seen very clearly that attributes and virtual attributes can be used in the body of rules inside specializations. This can e.g. be seen in the rule part of associate, where a virtual attribute of professor is used to compute an answer to the question what topics an associate knows about. This example is visualized by means of the Entity-Relationship diagram in fig. 1.

So, this example has shown some of the power of our Data and Knowledge Model. The use of entities, relationships and ISA-links helps to model things at the right abstraction level and it supports modularity. Inheritance of attributes allows to concentrate on data relevant to an object. The use of dynamic components in the form of virtual attributes has been shown. Rules that describe how to compute values for virtual attributes are inherited by specializations, which are allowed to add their own definition to the rule. This increases modeling power considerably. In section 3, the processing of queries in our model is discussed along the lines of this same example.

2 Recursive Views

In our model we also allow for views, in particular even recursive views. Views represent another way of looking at the modeled entities. Therefore they have no graphical representation in our model. In fact, they can best be looked upon as rules describing how to look upon the data.

Views can be expressed as normal queries like: "All associates that earn between 20K and 30K". They can also be expressed as rules. Take as an example the entity *person* and the relationship *parent_of* between two entities. A recursive *ancestor* view can now be defined by the rule:

$$\text{anc\_view}(X, Y) \leftarrow \text{parent}(X, Y) \vee \\ (\text{parent}(X, Z) \wedge \text{anc\_view}(Z, Y)).$$

The variables denote the keys used in the relationship parent. They are used to build an ancestor view that is presented to the user.

3 Queries and their Processing in the Data and Knowledge Model

3.1 Introduction

Now that we have introduced our model in some detail and have shown an example of its modeling power, we will concentrate on the processing of queries. We will talk about the kind of queries that we foresee, and how we can make optimal use of the facilities our model provides to solve queries. An outline of a query processing algorithm will be sketched.

We will not describe a detailed query language at this moment. A query language should be the final step in providing a complete system, but to develop a full-fledged query language before the model has completely developed itself seems premature. Probably our query language will bear some resemblance with e.g. [**6**].

3.2 Query Processing

In our system there are two main types of queries that can be posed.

- One can ask the actual value of (virtual) attributes of entities. This means that rules and constraints are used to compute values, and the database system is searched for values.
- One can ask how the answer to a query is obtained. This means that relevant rules and constraints are shown to the user.

These two types of queries will now be handled respectively.

3.2.1 Value-oriented queries

When the user asks for the value of one or more (virtual) attributes, it is the systems task to answer this query. It will therefore combine user supplied

```
        Then execute associated rule;
              $\forall X$ where ISA(E, X) Do search(X, Q);
              $\forall Y$ where ISA(Y, E) Do search(Y, Q);
        Else $\forall X$ where ISA(E, X) Do search(X, Q);
        Fi
   Fi
END
```

As can be seen from the algorithm sketched above, the inference process stops as soon as a real attribute is encountered. As long as nothing is encountered, the generalization hierarchy is traveled upwards in search for the attribute. As soon as a virtual attribute is encountered the generalization hierarchy is traveled upwards as well as downwards. After all, for the instantiations that can be specialized a value may be found in a more specialized entity as well. Notice that the execution of rules can also lead to a separate inference process, that uses the same search procedure.

Now let us take some example queries and show how they are solved. As a first query we take the following: "What is the salary of professor Persa?". The inference process starts in the entity type Professor, and because the answer cannot be found there it is moved to the entity type Researcher. Here it is noticed that *salary* is an attribute of Researcher, and the database can be searched for the value of the attribute *salary* that is inherited by professor Persa. The inference process is stopped now.

Another example is the query "What topics does researcher Smith know about". It is noticed that *knows_of* is a virtual attribute of Researcher and therefore the corresponding rule has to be executed. This means looking at the conferences Smith has visited and selecting all the topics presented there that are associated with his field. These are then gathered into one set and give part of the solution. If Smith happens to be a professor as well, the rule that provides answers to *knows_of* for professors will also be executed. The answers are then combined and presented to the user.

Our last example will show even more clearly the power of the model and the use of inheritance of rules. Let us consider the question "What topics does associate Wesson know about". The inference process starts by executing the rule for *knows_of* associated with Associate. This rule leads to two different inference processes: the *knows_of* rule for Professor is executed, and the *knows_of* rule for Researcher is executed *for the specific professor who manages Wesson.* These answers are combined and form part the total able to explain the reason for deductions to the user. However, there still is some more research to be done on this subject. We hope we can profit from Expert System developments here.

4 Mapping from the Data and Knowledge Model onto the Database System

Although our Data and Knowledge Model provides considerable modeling power and query processing facilities it is of a conceptually simple nature. Therefore the mapping of our Data and Knowledge Model to an underlying relational database system is rather straightforward. The entities, with their attributes, and relationships can be mapped onto relations. Keys should be propagated downto specializations. Queries can then be translated into normal relational algebra operations like joins [**14**]. Rules can be mapped onto Relational Algebra plus recursion. Our work on this subject [**1, 2, 3**] can be very helpful in this respect.

Although an architecture for a system to support our Data and Knowledge model still has to be investigated, we have developed some ideas. Especially the use of parallel systems like in [**8**] seems to be very promising.

5 Conclusion

In this paper we have presented a Data and Knowledge Model that integrates the representation of data and knowledge. A declarative way of specifying knowledge, in the form of Horn clauses, is chosen. The main concepts of the DK model are modularity of modeling, generalization/specialization hierarchies, dynamic components in the form of virtual attributes and inheritance of attributes and knowledge rules. Queries that are value-oriented and queries that ask for deductive steps performed are supported. Because of the straightforward way of mapping the DK model onto Relational Algebra plus recursion, efficient query processing is possible.

[**1**] P.M.G. Apers, M.A.W. Houtsma & F. Brandse, "Extending a Relational Interface with Recursion," Technical Report, Twente University of Technology, Enschede, The Netherlands, 1986.

The Semantics of Update in a Functional Database Programming Language

R.S.Nikhil

MIT Laboratory for Computer Science

545 Technology Square,
Cambridge MA 02139, USA

Arpanet: nikhil@xx.lcs.mit.edu

Databases that can store complex, nested objects may suffer performance penalties for their generality. Parallelism may be a solution. However, we need database languages that can express parallelism, and implementations that can exploit it. Functional languages and and their dataflow implementations are one approach, at least for queries. However, it has not been easy to express database updates in functional languages. In this paper we present a model for databases and updates in a functional language, with an intended dataflow implementation. The update language is declarative, parallel, and determinate, and can be extended to model historical data.

1 Introduction

The dichotomy between databases and programming languages is one of expedience. Ideally, it should be possible for arbitrary objects created and manipulated by programs to be persistent. But today, we know how to implement persistence efficiently only by restricting the structure of persistent objects and the operations that can be done on them.

For example, in current relational database systems, persistent objects must be flat, rectangular tables containing scalar values, and they must be manipulated only by a given set of relational operations. It is generally not easy to change the structure of the tables or to write arbitrary programs to manipulate them. Because of these restrictions, the database implementor can pre-plan disk layouts for the tables, can create indexes that use knowledge of these layouts, and compile queries so that they exploit this information thoroughly.

[0]This research was done at the MIT Laboratory for Computer Science. Funding for this project was provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

- When the user enters an *expression*, it is evaluated in the current environment, and an answer is printed. Viewing the environment as a database, this is a database *query*.

We model a database on exactly this idea. A database is an environment of bindings; update transactions specify new environments in terms of old; and queries are simply expressions evaluated in the latest environment. Thus, the operation of a single-user database system can be specified as a function from a list of transactions to a list of responses:[2]

```
Def dbsystem db (cons xact xacts) =
        {  resp, new_db = eval db xact
         In
           cons resp (dbsystem new_db xacts) } ;

dbsystem empty_db xacts
```

The phrase `(cons xact xacts)` is a pattern that matches the input list, binding the name `xact` to the first transaction and `xacts` to the rest of the list of transactions. `Xact` is evaluated in the database environment to produce a response and a new database environment (of course, for query transactions, the new database will be the same as the old). Finally, we construct and return the list of responses, beginning with this response and followed by the remaining list of responses obtained by running the remaining transactions against the new database.

A database shared among multiple users needs a little more packaging: we need a *manager* that non-deterministically receives transactions from individual users and merges them into a single list of transactions as input to `dbsystem`. The responses from `dbsystem` must then be despatched to the appropriate users. The details are outside the scope of this paper; the interested reader is referred to [9] or [3] for suggested solutions.

Each binding in the database associates a name to a *database type* or to a *value* of arbitrary type. The type structure is:

- Primitive types: `V` (void), `N` (numbers), `B` (booleans), `S` (strings), `SYM` (symbols) ...

[2]We use Id notation here. As in most modern functinal languages, application of a function `f` to argument `a` is written by juxtaposition: `f a`. Blocks (analogous to `let` or `where` expressions) are written

```
{    statement ;
    ...
    statement
In
    expression }
```

The left-hand sides of statements can be *patterns* that match the structure of the values returned by the right-hand-sides.

```
Course       : TYPE
CName        : Course <=> S
CUnits       : Course => N
CPrereq      : Course *<=>* Course
```

Course is another database type. CPrereq maps a Course into a set of Courses that are its prerequisites. "^ CPrereq" maps a Course into a set of Courses for which it is a prerequisite.

The (type of the) rest of the database:

```
Enrollment   : TYPE
EGrade       : Enrollment => S

S-Enroll     : Student <=>* Enrollment
C-Enroll     : Course  <=>* Enrollment
```

S-Enroll maps a Student into the set of his Enrollments, while "^ S-Enroll" maps an Enrollment into the corresponding Student.

The database type Enrollment (with associated functions) was introduced to model the event of a student enrolling in a course, which allows associating various data with that event, such as grade, date of enrollment, name of supervisor who approved it, *etc.* An alternative strategy would be to define the following functions directly on Students and Courses:

```
Takes-Courses: Student *<=>* Course
Grade        : (Student,Course) -> S
```

In conventional database terminology, our database types correspond to distinct record types. "=>" corresponds to an ordinary record field, whereas "<=>" corresponds to a record field that is also a key. The other indexed types correspond to one-to-many and many-to-many relationships, usually obtained by set owner/member links in CODASYL databases, and by joins in relational databases.

2.1 Queries

Queries are arbitrary applicative expressions evaluated in the database environment. A very powerful notation for expressions on collections is the "set comprehension" notation invented by Turner [14,13]. This notation can be regarded as a significant generalization of relational calculus languages like SQL.

For example, here is a query to find the names of all special-status students taking 15-unit courses:

```
{  e_15_special e  =      (CUnits  (^ C-Enroll e) == 15)
                      and (SStatus (^ S-Enroll e) == "special")
 In
   map (compose SName (^ S-Enroll))
       (filter e_15_special
               (all Enrollment)) }
```

`e_15_special` is a predicate that decides if the course related to enrollment `e` has 15 units and the student related to `e` has special status. Using it, we filter all enrollments, and map the composition of `SName` and `^ S-Enroll` over the remaining enrollments to produce the desired set. This operator-based view of functional query languages and methods to implement them are explored at length in [12].

Because of our parallel model of computation, the enumeration of enrollments, the filtering and the final mapping are all overlapped in a pipelined manner (see [11]).

The function `STotalUnits` whose type was shown in the database environment is an ordinary function. Here is a possible definition for it:

```
Def STotalUnits s =
         fold (+) 0
                 { CUnits (^ C-Enroll e) | e <- S-Enroll s }
```

i.e., when applied to a `Student`, it computes that student's total units using other database functions. This is sometimes called a "derived function" in the database literature.

Here is a recursive query that checks if the course `"6.001"` is directly or indirectly a prerequisite for the course `"6.004"`:

```
{ q c1 c2 = if (c1 == c2) then true
            else fold (or) false (map (q c1) (CPrereqs c2)) ;

 In
  q (^ CName "6.001") (^ CName "6.004") }
```

Note that one mixes indexed and ordinary functions freely. Definitions for ordinary functions may use recursion, conditionals, *etc.* In short, the query language is a complete, high-level programming language.

3 Operations on Indexed Functions

Indexed functions differ from ordinary functions in that they are defined incrementally with many statements, rather than in a single statement. An indexed function is first created using the "`empty`" construct, at which point it has an empty domain (undefined everywhere). It has zero information content, and is said to be "open". As the transaction

```
f [e1] = undef
```

specifies that (f v) is always undefined. Any other attempt to define f at v is an error.

The treatment of <=> is similar. For an indexed function f: t1 <=> t2 and expressions e1:t1 and e2:t2 that evaluate to v and w respectively, the statement:

```
f [e1] = e2
```

defines (f v) to be w and (^ f w) to be v. It will succeed only if f was previously undefined at v *and* if (^ f) was previously undefined at w.

For an indexed function f: t1 <=> t2 and an expression e1:t1 that evaluate to v, the statement:

```
f [e1] = undef
```

specifies that (f v) is always undefined. Any other attempt to define f at v is an error.

3.2 Multiple-Valued Index Functions: =>*, <=>* and *<=>*

Multiple-valued indexed functions initially map all arguments to $\perp_{set}$, the undefined set. As incremental definitions at some argument v are executed, the mapping improves to (insert w1 $\perp_{set}$), (insert w1 (insert w2 $\perp_{set}$)), and so on. If, at the end of the transaction, (f v) is

(insert w1 (... (insert wn $\perp_{set}$)))

then it becomes closed with those values, *i.e.,* (f v) is

```
(insert w1 (... (insert wn EmptySet)))
```

for subsequent transactions.

For an indexed function f: t1 =>* t2 and expressions e1:t1 and e2:t2 that evaluate to v and w respectively, the statement:

```
f [e1] += e2
```

extends the definition of f so that (f v) includes w.

Again, as we shall see later, in update transactions a new f: t1 =>* t2 automatically inherits mappings from an old version unless specified otherwise. To inhibit this, for expressions e1:t1 and e2:t2 that evaluate to v and w respectively, the statement:

```
f [e1] -= e2
```

database environment itself, which, for uniformity, can be regarded as an indexed function of type `SYM => object`. The special symbol "`db`" in the database environment evaluates to the database environment object itself.

An update transaction is a program that specifies the new graph in terms of the old. At the beginning of the transaction, every node in the graph has a new "shadow" version. Nodes corresponding to indexed functions are open and empty, *i.e.*, with no outgoing edges in the graph. If `e` is an expression that refers to an object in the old graph, then "`new e`" refers to its new version (thus "`new db`" refers to the new database environment itself). The update transaction contains incremental definitions for the new versions of objects. At the end of the transaction, *i.e.*, when the program has terminated, the new version of each object inherits any old contents that were not incrementally redefined, after which it becomes closed.

The new extension of a type, *e.g.*, `(new (all Student))` is $\perp_{set}$ until the end of the transaction, when it becomes closed, containing all objects of that type that are present in the new version of the database.

4.1 Examples

An update to increase the number of units for the course 6.006 by 3 units:

```
(new CUnits) [ (^ CName "6.006") ] = (CUnits c) + 3
```

The update consists of a single statement that specifies an incremental definition of the new version of the indexed function bound to `CUnits`. The new version differs from the old in that the course referred to by "`^ CName "6.006"`" is now mapped to a number 3 units greater than before.

An update to change the name of student John Xiao to John Zhao:

```
(new SName) [^ SName "John Xiao"] = "John Zhao"
```

An update to increase the units of all courses by 3:

```
{ f c = { (new CUnits) [c] = (CUnits c) + 3 } ;
  mapdo f (all Course) }
```

The first statement defines a temporary function `f` that increases the units of a course by 3. The second statement applies this to all courses (`mapdo` is like `map` in that it applies `f` to each course, but is different in that there are no results to be returned). In our parallel model of computation, all the applications of `f` can be performed in parallel.

An update to remove a grade erroneously recorded for John Zhao in the course 6.001:

```
{ record_grades Cn SnGs =
      {  f (Sn,G) = { e = theEnrollmentFor Sn Cn;
                      (new EGrade) [e] = G } ;
         mapdo f SnGs } ;

   (new db) ['record_grades] = record_grades } ;
```

The first statement defines the function value itself, and the second statement records it in the new database.

Another update introducing a function that can be used in subsequent transactions: given a student name and a course name, it adds that enrollment:

```
{  add sn cn =  { s = ^ SName sn ;
                  c = ^ CName cn ;
                  e = make Enrollment () ;
                  (new S-Enroll) [s] += e;
                  (new C-Enroll) [c] += e } ;

    (new db) ['add] = add }
```

Again, the first statement defines the function value itself, and the second statement records it in the new database.

An update introducing a function that, given a student name and a course name, deletes that enrollment:

```
{  drop sn cn =  { s = ^ SName sn ;
                   c = ^ CName cn ;
                   e = theEnrollmentFor sn cn ;
                   (new S-Enroll) [s] -= e ;
                   (new C-Enroll) [c] -= e ;
                   (new EGrade) [e] = undef } ;

    (new db) ['drop] = drop }
```

Note that the way to remove an object from the database is to ensure that there is no function defined on it. The object then disappears from the database.

5 Discussion

5.1 Parallelism

The major issues in designing a database programming language with parallelism are:

- by name of creator, assuming that `dbsystem` records the name of the creator of each database environment.
- by arbitrary property, *i.e.,* the most recent database environment in which a given boolean expression evaluates true.
- ...

Once we can specify particular environments, the phrase:

```
with environment-expression
     expression
```

can be used to evaluate an expression within that environment. Thus, we can write queries and updates that depend on any or all previous states of the database.

5.3 Concurrency Between Transactions

The parallelism that we have focused on so far is all within a single transaction. Referring to the database system model of Section 2, the parallelism is within the phrase: `(eval db xact)`. Within `dbsystem`, the result database from one transaction is used as the environment in which to evaluate the next transaction.

This is not to imply that there cannot be any parallelism *between* transactions. First, since a closed database environment is never subsequently modified, a read-only transaction (query) can continue using an old database as long as necessary, without holding up subsequent update transactions. Second, even update transactions can be overlapped: the lenient semantics of our language allows `(eval db xact)` to return a value (the response and the new database) immediately, before the transaction has completed (this behavior is also exhibited by languages with lazy evaluation). This permits `dbsystem` to begin evaluating the next transaction immediately.

A problem arises due to aborted transactions, which can cascade through all subsequent transactions that have already begun executing. To avoid this, one will have to employ the usual solutions: either prevent multiple transactions from overlapping (pessimistic), or allow them to overlap, keeping track of which parts of the database they actually see, so that an abort does not cascade through non-interfering transactions (optimisitic).

5.4 Comparison With Other Approaches

The top-level definition of the database system (`dbsystem`) that we presented in Section 2 is almost identical to other "functional" views ([9], [2]). The differences arise in the meaning of the phrase `(eval db xact)`— what is a database, what is a transaction, and what is the `eval` function?

6 Future Directions

The work described here is a preliminary attempt to design a declarative update language within the framework of a functional database system. There are many details to be completed, many issues still to be investigated. As a vehicle for this research, we are constructing a prototype of the system. This is initially implemented in Lisp to take advantage of Lisp's rich programming environment; later we expect to incorporate it into Id and to run it on our dataflow multiprocessor (emulated for now, a real one later). Until we have more experience with writing applications in our prototype, we cannot make a convincing judgment as to whether it is easy or difficult to express updates in this model.

Despite the title of this paper, what we have presented is by no means a formal semantics, and until that event, we cannot possibly be precise in our claims about parallelism, determinacy, *etc.* Once the language has reached a reasonably stable point, we expect to extend the formal semantics of Id, expressed as rewrite rules [11] to cover this database model.

There is a disturbing lack of type-orthogonality in the indexed types— currently, the domain and range of an index type can only be database or primitive types. We are taking this position currently for pragmatic reasons— it is not clear what it means to index on tuples, sets, nested structures, *etc.*

In our model, currently an object is deleted automatically from the database when it no longer participates in any mappings (no query can be asked of it). The reason for this choice, rather than a command to delete an object directly, was that it is not clear what happens to the mappings in which the object participates. However, removing it from all mappings can be quite tedious to specify. This issue requires more investigation. A more difficult question: when can a *type* be deleted from the database, *i.e.,* what happens to existing objects of that type, mappings on those objects, *etc.*?

The transaction language, like Id with I-structures, is not a purely functional language any more, though it does retain the parallelism and determinacy (and, we claim, declarative nature) of functional languages. The loss of referential transparency is not without cost: it can inhibit certain optimizations that are possible in functional languages. In Id, we have developed a programming methodology whereby we use I-structures only to define new, efficient functional array abstractions, after which the bulk of the program is written functionally [4]. Can such a methodology be extended to deal with our database extensions?

In a related project, we are looking at architectural and low-level programming issues in implementing arbitrary object persistence in the Tagged-Token Dataflow architecture, assuming explicit commands to store and retrieve objects. The gap between that implementation and the database model presented here is yet to be bridged.

[14] D. A. Turner. The semantic elegance of applicative languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 85–92, ACM, October 1981.

1 Introduction

The use of modularization and information hiding is widely accepted as being central to managing the complexity of large software systems. The past decade has witnessed the development of several programming languages which provide features for modularization and separate compilation, e.g., Ada [USDD83] and Modula2 [Wirt82]. However, the capabilities of these languages are not sufficient to support the evolution of large and complex systems where many versions exist. Research has lead to the development of a variety of tools which support version control, software reuse, and system evolution, e.g., [Tich85], [Notk85], [Estu85], [Prie86], [Bern87], [Nara87], [Lebl87] and [Wink87].

This report introduces a simple yet formal model of module interconnection and version selection which incorporates and extends many current ideas in the area. The model provides a conceptual basis for the construction of modules from submodules, and for the selection of versions. Our current focus is to present an abstract model which is suitable for theoretical investigation. Ultimately the model is intended to provide a basis for program development tools, although the concepts may then be formulated in a different manner. Also, in this report we do not address the important issue of "manufacturing" or "deriving" software objects, in the sense of MAKE [Feld79] and the models presented in [Bori86] and [Pola86]. We believe that our model can be integrated with one such as [Bori86] to provide a comprehensive framework for configuration management.

The primary innovation of this report is a new formalism for specifying the structure of a system called a *module interconnection grammar* (MIG). A MIG is essentially a context-free grammar where the symbols are the names of module families, and the productions represent ways of constructing modules from submodules. A MIG tree, corresponding to a derivation tree, represents one possible way of constructing a system. A system instance tion 4 we show how MIG trees can be reduced to a minimal form which concisely describes the flow of resources between module instances. In section 5 we introduce the notion of equivalence between MIGs and equivalence-preserving transformations on MIGs. Finally, in section 6 we briefly discuss attributes and constraints.

2 Overview of the Model

It is generally accepted that large software systems should be decomposed into modules which share resources, such as procedures, functions, types, and variables, among themselves. While a variety of different module interconnection schemes have been proposed, no consensus among them has emerged. We adopt a scheme, based primarily on the module interconnection language NuMIL[Nara85], which is particularly suitable for programming in the large. There are two kinds of modules in this scheme; *atomic* modules and *compound* modules. Atomic modules are indivisible units in which resources originate and are used. Compound modules, which are composed of submodules, provide structure to the system. Every module must explicitly state which resources it imports and exports. A compound module provides a scope or name space in which the imports and exports of its submodules can be matched.

It is instructive to compare this approach with the one taken in programming languages such as Ada. Consider the following skeleton of an Ada program unit.

where the symbols are the names of module families and the productions represent ways of constructing modules from submodules. The above example describes a MIG with two productions. In general, there may be many ways of constructing a module, for example,

$$\begin{array}{l} MAIN \rightarrow A\ \ P \\ MAIN \rightarrow A\ \ Q \\ P \rightarrow P\_DRIVER\ \ B \\ Q \rightarrow Q\_DRIVER\ \ C \end{array}$$

provides two ways of constructing $MAIN$.

A MIG is always interpreted with respect to a signature which describes the modules which appear in the system. In particular, a signature names the imports and exports of each module. For convenience, we often write the imports and exports of a module directly above and below its occurrence in a production. For example,

$$\begin{array}{cccc} doit & & foo & doit \\ MAIN & \rightarrow & A & P \\ helper & & helper & foo \end{array}$$

$$\begin{array}{cccc} doit & & doit & hoo \\ P & \rightarrow & P\_DRIVER & B \\ foo & & hoo & foo \end{array}$$

states that $MAIN$ exports a resource *doit*, which originates in P_DRIVER, and imports a resource *helper*, which is used in A. There are various consistency conditions on productions which ensure that resources are introduced appropriately. For example, the production $MAIN \rightarrow A\ \ P$ is consistent because the export of $MAIN$ and the imports of A and P are uniquely provided. It is possible to specify that resources be renamed within a scope, for example, $MAIN \rightarrow A[foo/goo]\ P$ specifies that the resource *foo* in A is to be called *goo* within the scope.

A MIG tree, loosely analogous to a derivation tree, represents one possible way of constructing a system. For example, the following MIG tree represents one possible way of constructing $MAIN$.

node *root* is included to represent the imports and exports of the system as a whole, as described by conditions 5 and 4 respectively. The following is easily verified.

Proposition:
For a MIG tree T, the graph $red(T)$ defined above is a RFG.

Note that $red(T)$ cannot be constructed by matching imports and exports of the leaves of T directly, since the same resource name may be used several times in different contexts. Moreover, if resource name changes are incorporated, then renaming might occur at each edge of a witness path $z_1, \cdots, z_n$.

To simplify compilation in the programming language Ada, cycles are not permitted in the import/export relationships between modules. This motivates us to study those MIG trees T for which $red(T)$ is acyclic. We now give a sufficient condition for acyclicity.

Definition:
A production $A \rightarrow B_1 \cdots B_n$ is *acyclic* if the import/export relationships between $B_1 \cdots B_n$ are acyclic.

Proposition:
If all productions used in constructing T are acyclic then $red(T)$ is acyclic.
Proof Essence:
If $red(T)$ is cyclic then the production used at the least common ancestor in T of all modules participating in the cycle must be cyclic.

The converse of this proposition is not true: it is possible to construct a T using a cyclic production for which $red(T)$ is acyclic, as the following example shows.

If the members of a module family have little in common, then such constraints cannot be imposed. In this case, integrity checking must be performed after particular module instances have been selected.

4 Reducing MIG trees

Suppose that T is a MIG tree, and I is an instance of it. Intuitively, T describes the manner in which resources are interchanged among module instances given by I. In this section we introduce an abstraction called "resource flow graphs" for representing this linkage information directly, and describe how a MIG tree T can be "reduced" to a resource flow graph $red(T)$. Intuitively, T and $red(T)$ specify the same flow of resources -- I can also be interpreted as an instance of $red(T)$ -- and differ only in the structural information they provide. This formalizes the notion that the compiled version of a large software system may contain less structural information than the representation maintained by the programming environment.

Definition:
Let $S = <M, R, i, e>$ be a signature.
A *resource flow graph* (RFG) for S is a directed graph $H = <W, F, \mu, \rho>$ where

$<W, F>$ is a directed graph (duplicate edges are permitted);

μ is a mapping from W to M, i.e., a node labeling,

ρ is a mapping from F to R, i.e., an edge labeling,

which satisfies the following conditions:

for all edges f from x to y, $\rho(f) \in e(\mu(x))$ and $\rho(f) \in i(\mu(y))$; and

for all edges f and g to y, $\rho(f) \neq \rho(g)$.

Definition:
Let $G=<S\ ,P\ ,C>$ be a MIG over the signature $S=<M\ ,R\ ,i\ ,e>$.
A *MIG tree* of G is a labeled tree $T=<V\ ,E\ ,\lambda>$ where

$<V\ ,E>$ is a tree (i.e., a directed, rooted, strongly acyclic graph with vertices V and edges E contained in V x V) and

λ is a function from V to M, i.e., a node labeling,

which satisfies

for root w, $\lambda(w)=C$;

if v is an internal node with children $v_1\ ,\ ...\ ,\ v_n$, then $\lambda(v)\rightarrow\lambda(v_1)\ \cdots\ \lambda(v_n)\in P$; and

if v is a leaf node, then $\lambda(v)\in atom(M)$.

Each node of a MIG tree corresponds to a module and a node's children correspond to submodules of that module. The same module name may occur more than once in a system -- λ need not be 1-1 -- and there may be distinct (non-isomorphic) subtrees below each occurrence.

Our next major step is to define "instances" of a MIG tree, i.e., actual pieces of code structured according to the tree. To do this, we need the notion of a library of module instances.

Definition:
A *library* is a 4-tuple $L=<N\ ,R\ ,i\ ,e>$ where

N is a set of abstract names called *module instance names*;

R is a set of resource names; and

i and e are functions from N to the powerset of R,

which satisfies the following condition.

Module consistency: For all $n\in N$, $i(n)\cap e(n)=\{\}$.

Definition:
A *module interconnection grammar* (MIG) is a triple $G = <S, P, C>$ where

$S = <M, R, i, e>$ is a signature;

P is a set of *productions* (or rules) of the form $p = A \rightarrow B_1 \cdots B_n$ where $n \geq 1$ and $A \in comp(M)$ and $B_1, \cdots, B_n \in M$; and

$C \in M$ is called the *root module,*

which satisfies the following consistency conditions:

Non-recursiveness:
There is no sequence of rules $p_1, \cdots, p_n (n \geq 1)$ where for each j, $1 \leq j \leq n$, the head of p_{j+1} occurs in the tail of p_j, and the head of p_1 occurs in the tail of p_n.

Resource completeness:
For each rule $A \rightarrow B_1 \cdots B_n \in P$, $\bigcup_{k=1}^{n} i(B_k) \cup e(A) \subseteq \bigcup_{k=1}^{n} e(B_k) \cup i(A)$.

Resource uniqueness:
For each rule $A \rightarrow B_1 \cdots B_n \in P$, the sets $i(A)$, $e(B_1)$, $\cdots$, $e(B_n)$ are pairwise disjoint.

The first condition rules out the possibility of a module appearing within itself. The second condition guarantees that each resource required in a scope is provided. (This resembles conditions on resources specified in [Tich85].) The third condition guarantees that every resource is uniquely provided, i.e., that name conflicts do not occur. Note that atomic module names are analogous to terminal symbols and compound module names are analogous to nonterminal symbols.

A number of generalizations of this definition are possible. For example, the distinction between atomic and compound module names could be dropped. Two other generalizations are presented in the remarks below.

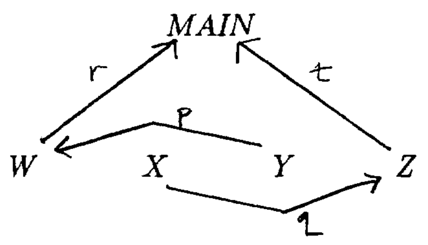
Remark 1

Modules which are developed independently may not be consistent in their naming of shared resources and some mechanism for renaming resources within a scope must be provided. We define a *name change function* on a set of resource names R to be a mapping

$$\underset{}{\overset{r,t}{MAIN}} \rightarrow \underset{p}{\overset{q,r}{A}} \;\; \underset{q}{\overset{p,t}{B}}$$

$$\underset{p}{\overset{q,r}{A}} \rightarrow \underset{p}{\overset{r}{W}} \;\; \overset{q}{X}$$

$$\underset{q}{\overset{p,t}{B}} \rightarrow \overset{p}{Y} \;\; \underset{q}{\overset{t}{Z}}$$



This system exhibits very poor structure; it is hard to imagine a circumstance where the atomic modules should be grouped in this way. In fact, a criterion of good design might be that if *red*(*T*) is acyclic then all productions used in constructing *T* should be acyclic.

5 Equivalence of MIG Trees and MIGs

A fundamental research issue concerns the development of a general theory of system evolution. In this section we indicate one direction that can be pursued in this area. In particular, we introduce a notion of equivalence between MIG trees, based on the RFGs associated with them, and then extend this notion to equivalence between MIGs. This allows us to define several local structural transformations on MIGs which preserve equivalence. We expect that, in the context where resource renaming is permitted, a natural extension of these transformations can be defined which is complete in the sense that it allows a MIG to be transformed into any equivalent MIG.

We begin with the definition of equivalence between MIG trees.

Definition:
Let G be a MIG and let $\hat{G}$ be a MIG which is identical to G except that the order of nonterminals within some production p has been permuted. Then $\hat{G}$ is the result of *reordering* G at p.

It is intuitively clear that reordering preserves equivalence.

We now introduce a transformation which allows a collection of modules to be encapsulated together into a single module.

Definition:
Let $S = <M, R, i, e>$ be a signature, $G = <S, P, C>$ be a MIG and $p \in P$ have the form $A \rightarrow B_1 \cdots B_j B_{j+1} \cdots B_n$. Let $\hat{G} = <\hat{S}, \hat{P}, \hat{C}>$ be the MIG which is identical to G except that

1) a new module name X has been added to $\hat{S}$;

2) the production p in $\hat{P}$ has been replaced by $A \rightarrow B_1 \cdots B_j X$;

3) the production $X \rightarrow B_{j+1} \cdots B_n$ has been added to $\hat{P}$; and

4) $\hat{i}(X) = \bigcup_{k=j+1}^{n} i(B_k) - \bigcup_{k=j+1}^{n} e(B_k)$; and $\hat{e}(M) = \bigcup_{k=j+1}^{n} e(B_k)$.

Then $\hat{G}$ is the result of *nesting* $B_{j+1}, \cdots, B_n$ at p.

Proposition:
Nesting preserves equivalence.
Proof sketch:
Show $G \leq \hat{G}$ and $\hat{G} \leq G$. In both cases, show by construction that for any MIG tree generated by the dominated grammar, there is an equivalent MIG tree generated by the dominating grammar.

Note that an arbitrary collection of modules can be nested by first applying reordering.

We now define "unnesting", the inverse of nesting. A general definition of unnesting is possible only if renaming is permitted, since the exposure of hidden names may result in name conflicts. The following restricted definition, in which unnesting is permitted only if there are no name conflicts, indicates the general approach.

modules, adding new imported and exported resources to a module, and changing the source of a resource from one module to another.

6 Attributes, Equations, and Constraints

In this section we briefly discuss adding attributes, equations, and constraints to our scheme. In this more general context, instances of a module family are distinguished by attributes which describe their characteristics. Attributes can be associated with modules and/or particular resources in modules. Attribute values may be given by the programmer, derived from the code, or computed using attribute equations. Equations can be associated with productions and, in some cases, with the resource attributes of atomic modules. If I is a system instance, the atomic attribute values and the equations together imply attribute values for the compound modules of I, and ultimately the root of I. It is possible to impose constraints on attribute values that limit which instances are appropriate in a particular circumstance. The process of constructing a system instance entails selecting module instances which satisfy these constraints.

We now present three examples which illustrate our general approach, and indicate the kinds of research problems we hope to address. For this discussion we focus on a simple MIG containing the one production

$$\overset{\substack{join\\ format}}{QUERY\_PROC} \rightarrow \overset{sort}{SORTER}\ \ \underset{\substack{sort\\ format}}{\overset{join}{JOINER}}\ \ \overset{format}{FORMATTER}$$

which we abbreviate as $QP \rightarrow S\ J\ F$. This corresponds to a simple relational database query processor. We suppose further that we have a library containing module instances S_1 and S_2 which implement S; J_1, J_2 and J_3 which implement J; and F_1 and F_2 which implement F.

In the present situation, it is also possible to use a top-down computation to infer constraints on submodules from constraints on the root module. To illustrate, suppose that we are interested in finding all system instances I which run on a VAX. This is expressed using the constraint $QP.tm \supseteq \{VAX\}$. From the equation we see that a system instance will satisfy this if and only if the following three constraints are satisfied:

$$S.tm \supseteq \{VAX\}$$
$$J.tm \supseteq \{VAX\}$$
$$F.tm \supseteq \{VAX\}$$

Using this characterization, we easily see that the set of system instances which run on the VAX contains precisely $<S_1,J_1,F_1>$, $<S_1,J_1,F_2>$, $<S_2,J_1,F_1>$, and $<S_2,J_1,F_2>$. We note that this top-down approach to finding system instances is closely related to that of Winkler [Wink87], although in the present context it is more restricted. Also, it suggests that in a practical implementation of a system library, efficient access to module instances via attribute values should be provided.

In the case of the attribute *tm*, satisfaction of the constraint $QP.tm \supseteq \{VAX\}$ is accomplished by the submodules in an essentially independent manner. In our next example, we present an equation which forces the attribute values to interact. Specifically, suppose that an attribute *mmu* for main_memory_usage is defined for the three atomic modules, and suppose that the equation

$$QP.mmu = J.mmu + max(S.mmu, F.mmu)$$

is associated with the production. The constraint $QP.mmu \leq 100K$ now restricts attention to system instances I such that $I(J.mmu) + max(I(S.mmu), I(F.mmu)) \leq 100K$. One way to find such instances is to use a backtracking algorithm. A fundamental direction for our research is to explore other approaches to finding these instances.

a back-tracking approach.

The above discussion provides a bottom-up mechanism for checking whether constraints are satisfied. In some cases, a top-down approach can be used to infer constraints at the leaves which are implied by constraints at the root. Algorithms based on dynamic programming can also be used. In general, the problem of efficiently inferring constraints and selecting system instances which satisfy them is an open research problem.

References

[Bern87] Bernard, Y., M. Lacroix, P. Lavency, M. Vanhoedenaghe, Configuration management in an open environment. *Porc. 1st European Software Engineering Conf.*, De Strasbourg, France (September 1987), 37-45.

[Bori86] Borison, E. A model of software manufacture. In *Proc. of the Intl. Workshop on Advanced Programming Environments*, IFIP WG 2.4, Trondheim, Norway (June 1986), 197-220.

[Estu85] Estublier, J. A configuration manager: the Adele database of programs. *Workshop on Soft. Eng. Env. for Prog. in the Large*, Cape Cod, June (1985), 140-147.

[Feld79] Feldman, S.I. MAKE - A program for maintaining computer programs. *Software - Practice and Experience* 9 (1979), 255-265.

[Katz87] Katz, R., Managing change in a computer-aided design database. *Proc. Intl. Conf. on Very Large Data Bases*, Brighton, England (September 1987), 455-462.

[Lebl87] Leblang, D.B. and Chase, R.P. Jr., Parallel software configuration management in a network computing environment. *IEEE Software* (1987) to appear.

A DML for Complex Objects *

M. Lacroix and M. Vanhoedenaghe

Philips Research Laboratory, Brussels
Av. Van Becelaere, 2, box 8
B-1170 Brussels, Belgium

ABSTRACT

A data manipulation language for handling complex objects that are represented as structured values is discussed. The language is strongly typed and contains primitives for manipulating subtypes and union types in a more traditional framework than object oriented languages. Its functional style facilitates its integration in general purpose programming languages.

1. Introduction

Engineering applications require database systems offering other facilities than those available in current commercial systems. A key requirement is the support of complex objects, i.e. data structuring facilities richer than those offered by flat data models such as the relational and entity-relationship models. The ADDL data model [1], whose DML is presented in this paper relies on classical constructors such as set, list, n-tuple, mapping, union, and recursive combination thereof. Although it is not a flat data model, it is nevertheless similar to the relational model in that it represents everything as values in the database.

Another major requirement of engineering applications is the availability of the data manipulation operations at the application programming interface. Tools in software engineering and CAD applications are typically written in general purpose programming languages, and can implement quite sophisticated algorithms accessing and manipulating complex objects. The application programming interface to database systems is generally difficult due to mismatches between the operations and objects of the DBMS and those of the host programming language [2]. The data manipulation operations described in the present paper are designed so as to facilitate their embedding in general purpose languages. The use of a data model where the objects are values facilitates the use of an applicative style for the manipulation operations. These operations can then be made available as

* This work is funded in part by the "Services de Programmation de la Politique Scientifique" under Contract KBAR/SOFT/4.

'user_name' and a value of type 'bytes'; these two types are not further refined here, and are basic types.

As in the relational model, everything is represented by values, i.e. there is no notion of entity existing independently of the values of its attributes.

The naming for the objects is supported by the mappings. As a first approximation, a mapping is similar to a relation in the relational model (with the domain of the mapping corresponding to the primary key attributes, and the range corresponding the non-primary key attributes). The essential difference being that the types of the domain and range of a mapping are not limited to scalar types. In practice, it appears that the form of the domain of mappings can reasonably be restricted to basic types or n-tuples defined on basic types. A similar restriction can be found in the data model discussed in [3]; it is adopted in the current prototype implementation of ADDL.

The use of mappings in the range of mappings as in the above schema, where a 'directory' maps 'name's to values which can again be of type 'directory' allows for a recursive naming structure. The mapping also nicely describes in which context a name of a particular type uniquely identifies a value. In the above example a 'name' only uniquely identifies a value in a 'directory'. This is to be contrasted with the fact that a 'user_name' always uniquely identifies a 'directory', since there can only be one occurrence of this mapping in the database.

Values of a union type only belong to one of the alternative types constituting the union. In the above schema, a value in the range of a 'directory' mapping is either of type 'directory' or 'file', and never of both types.

3. The Data Manipulation Language

Engineering applications generally involve the creation and manipulation of a lot of intermediate values. The data structuring and manipulation facilities that are used for the database values are also available for the intermediate values in the program space. The only difference between the database values and the intermediate values is that the database values are component values of one special value representing the whole database.

The DML operators of ADDL are strongly typed. The type of an operator must match with the type of its operands. If two types have the same name or if they have the same textual description, then they match. As a consequence, two types having the same structure but using different names for component types do not match. This rule is further refined for union types (Section 3.3), subtypes (Section 3.4) and generic types (Section 3.1).

Example 1

Suppose that the example database contains the following facts. The user "John" has a home_directory. This directory contains a sub-directory named "sources" and this sub-directory contains the file named "test1.c". For accessing the owner of this file one has to use the following selection expression :

```
select('owner',
       map(map(map(root(),
                   "John"),
               "sources"),
           "test1.c"))
```

where "select"
is a selection operator that returns the indicated field of an aggregate;

"map"
is a selection operator that given a mapping value and a value of its domain type returns the associated value of the range type.

3.2. The update operator

The operators defined in previous sections allow to retrieve component values and to construct new complex values. However, for modifying component values of complex values the language has to include an update operator.

The ADDL data model only makes use of values for representing the objects. It does not rely on notions such as pointers and locations which are traditionally used in general purpose programming languages for building complex structures. The style of update of those languages, which consists in changing pointers and the contents of locations thus cannot be adopted here.

The approach that is usually adopted in applicative languages for "changing" a value is to construct a new one, generally by using selected parts of the old one. Indeed, we might consider using a similar approach here. The construction operators defined so far allow to construct new complex values using existing or newly created components. Because we consider the database as a complex value an update of the database can be done by replacing the whole database value by a new one (constructed from components of the old database value and newly constructed values). This means that for updating a small part of the database, i.e. a small component value of the huge database value, one first has to select that component value and then build the whole database value again, using a new value and all the remaining parts of the database value. This reconstruction of a new database value is very impractical and it makes concurrent updates virtually impossible.

The update operations that are offered in relational systems or in the is simply a mechanism for giving a name to a (non-evaluated) selection expression. The expression is to be re-evaluated each time the name is referred. Definitions can be nested.

Example 3

Suppose that we have the same database as in Example 1. We then can define MY-PATH as the selection expression that returns the sub-directory "sources" of the home_directory of the user "John" issuing the following definition :

```
let(MY-PATH, map(map(root(),
                     "John"),
                 "sources"))
```

Using this definition the update operation of Example 2 can be rewritten :

```
update(select('owner',
              map(MY-PATH,
                  "test1.c")),
       "Beth")
```

3.3. Union types

Besides its importance for defining variant structures for the values, type union is essential in the definition of values with a recursive structure.

A union type matches itself and any of its alternative types. (In the latter case, the "discriminate" operator will be used to regain static type checking.)

Given an expression whose type is a union of alternative types, the discriminate operator is used for applying on the expression operators that are different for each of the alternative types. For example, if "expr" is of type '(t1 OR t2)', op1 is an operator defined on the type 't1', and op2 is defined on type 't2', one will write

```
discriminate( "expr",
              t1 : op1 ,
              t2 : op2 )
```

3.4. Subtypes

The subtype constructor offers the facilities necessary for representing hierarchical type structures. The general from of a subtype definition is

```
subtype : supertype
```

represented by compound terms. These compound terms are only to be "accessed" by the predicates that implement the DML operations.

(2) There is no type checking in Prolog, so the type checking of the ADDL operations, which is in essence static, has to be done dynamically by relying on a type information contained in the internal representation of the compound values. A special make predicate is available for building an ADDL basic value from a Prolog atom or number.

(3) The operations that are passed as arguments to predicates such as discriminate are predicate names.

In C, the DML operations represented as library functions.

(1) The C scalar values correspond to atomic ADDL basic values. The compound ADDL values are represented as pointers. The data referred by these pointers are only to be accessed by the functions corresponding to the DML operations. These pointers can be passed as parameters to user-defined functions, and can also be assigned in variables. The structured values of C such as arrays and structures could as well be used as "big" basic ADDL values; they are atomic as far as ADDL is concerned, and compound in C.

(2) The type system of C is too weak for supporting the ADDL type system. For instance, there is no way to define the type of a function such as head (Section 3.1) since this type is generic. Dynamic type checking of the operations similar to what is done at the Prolog interface is the simplest solution to implement. Apart from performance problems, it presents the drawback of not allowing the type checking of the user-defined functions to which ADDL values are passed. The feasibility of a static type checker for C programs embedding ADDL operations has not been explored. Similarly to what is done for Prolog, make functions are used for turning C values into the appropriate ADDL basic types.

(3) Pointer to functions can be passed as arguments to the functions implementing the higher order ADDL operators.

5. Concluding remarks

In ADDL hierarchical type structures can be defined using subtypes. This structuring capability together with the automatic type coercion offers a facility similar to the one offered in object oriented systems where the methods defined on a class are inherited by all its subclasses. However, the modeling of the information specific to that subclass is easier in object oriented systems (addition of instance variables) than it is in ADDL.

Our approach to persistency is very similar to PS-Algol [5]. Our persistent values are the components of the database value; in PS-Algol the

6. P. Buneman and M. P. Atkinson, Inheritance and Persistence in Database Programming Languages, Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington,D.C., June 1986, 4-15.

7. W. Lamersdorf, G. Mueller and J. W. Schmidt, Language Support for Office Modelling, Proceedings of the 10th International Conference on Very Large Data Bases, Singapore, , August 1984, 280-288.

APPENDIX A: ADDL DDL grammar

In the grammar below, the symbols <,>,{,},+,*,|, and ::= are metasymbols. Symbols enclosed in double quotes denote themselves, symbols enclosed with < and > are non_terminal symbols, {...}+ denotes one or more occurrences of a syntactic element, {...}* denotes zero or more occurrences of a syntactic element, and | separates several alternatives.

```
<ADDL_schema>        ::= { <type_definition> }+

<type_definition>    ::= type_name ":" <type_expr>

<type_expr>          ::=  basic_type

                        | type_name

                        | "SET" "OF" <type_expr>

                        | "LIST" "OF" <type_expr>

                        | "<" selector ":" <type_expr>
                             { "," selector ":" <type_expr> }* ">"

                        | "(" <type_expr> "-->" <type_expr> ")"

                        | "(" <type_expr> { "OR" <type_expr> }+ ")"
```