Generating queries from complex type definitions*

Manfred A. Jeusfeld
Informatik V, RWTH Aachen, D-52056 Aachen
jeusfeld@informatik.rwth-aachen.de

Abstract

Many information systems are implemented as application programs connected to a database system. A characteristic problem of such systems is the famous impedance mismatch, i.e., the conceptual distance between the programming and the database languages. The traditional solution is to implement an interface that transforms one representation into the other. Commercial database systems offer preprocessors that allow to embed the database language (e.g., SQL) into the programming language (e.g., C). Such an approach frees the application programmer from the task to specify details of the communication. However, the impedance mismatch is not solved but aggravated. The set-oriented database language is intermixed with the element-oriented programming language, a notorious cause for programming errors. Moreover, there is no support in mapping the restricted data representation of databases into the more complex type system of programming language. This paper proposes an intermediate language, the API modules, for specifying the relationship between the representations in the database and in the application program. The query for retrieving the information and the data types for storing it can be generated from the API module. The modules are simple enough to allow reasoning on queries generated from them.

1 Introduction

The purpose of a database system is to maintain a large amount of information for a variety of application programs. The application-specific clustering is either described as a database view definition or performed by filters inside the application program. Both approaches have their disadvantages:

- View definition languages are restricted to the type system of the database system. In the case of relational databases, only flat relations can be expressed. In the case of object-oriented databases, the type system depends on the specific data model of the database system.
- Handcoded clustering by filter procedures within the application program is error-prone and gives away the chance of reasoning on the relationship between the information in the database and in the application program.

Section 2 introduces *API modules* as the interface between the database and the application program. Base types are imported from the database. Application specific types are defined by using tuple, set, and pointer constructors. The latter allows to represent recursive concepts of the database schema.

Section 3 presents the mapping of the API modules to a logic program delivering complex terms. These terms are read by a parser that itself is generated from the API modules.

Section 4 relates the types in an API module to statements of a concept language. Thereby, types of two different API modules can be checked for subsumption and consistency.

2 Interface Modules

Interfaces between imperative-style programming languages should both reflect the major type constructors and the facilities of the database query language. The most common type constructors are *tuple* and *set*. Some languages also support *lists*. Pseudo-recursive type definitions are possible when allowing pointer types, e.g. in C and Modula-2. Common base types are *Integer* and *String*. The denotational semantics of a type expression is a potentially infinite set of values, for example *[Integer,String]* denotes the cartesian product of the semantics of the component types.

2.1 Example

Assume a database provides information about a company. An application programmer has the task to process the information about the projects of employees who work for a given department. The API module could look as in Figure 1.

The FROM clause imports concepts from the database schema. They are used like (finite) base types. Their extension is represented in the current database state. The TYPE clauses declare complex

*This work was partly supported by the Commission of the European Union under ESPRIT BRA 6810 (Compulog 2).

```
API-MODULE Emps;
FROM CompanyDb IMPORT Employee, Project,
     Department, String;
TYPE
     EmpType/Employee = [name: String;
                         project: {Project};
                         dept: DeptType];

     DeptType/Department = [deptName: String;
                            head: *EmpType];
PORT
     e: {EmpType| dept.deptName=$N};
END.
```

Figure 1: API module for the company example

data structures on top of the imported concepts. EmpType is a record type which represents the name of an employee, his projects, and the department. The latter is given by the name and the reference (pointer) to that employee who is the head of the department. The purpose of the pointer is to encode recursive type definitions. The PORT declaration defines which information of the database should be teransfered to the application program. Here, all employees who have a department named by $N are of interest. The token $N denotes a placeholder for a string whose value is inserted by the application program at run time.

3 Query Generation

From the database point of view, an API module is a collection of simple view definitions whose extensions are represented by terms conforming the type definitions. These views are encoded as a logic program defining a predicate hasType(T,V). It formally defines the set of values V having type T, i.e., the semantics of the type T. The database system is modelled by two predicates for accessing information:

- In(X,C) denotes that the database object X is an instance of the concept C, e.g., In(e2341,Employee), In("Peter Wolfe",String).
- A(C,a,X,Y) states that the object X is related to the object Y by an attribute a which is defined in class C, e.g., A(Employee,name,e2341,"Peter Wolfe").

The logic program can automatically be generated from the type definitions by a simple top down traversing algorithm on the syntax tree of a type definition[1]:

For each concept C imported in the API module we include a clause which delivers all values of type C.

```
hasType(C,C(_X)) :-
  In(_X,C).
```

A tuple type has the general form T/C = [a1:T1,...,ak:Tk]. The decoration C is called the "class" of T. It is mapped to the clause pattern

```
hasType(T,T(_X,_Y1,...,_YK) :-
  In(_X,C),
  <map(a1:T1)>,
  ...
  <map(ak:Tk)>.
```

The parts <map(ai:Ti)> have to be mapped as follows:

- If Ti is a set type {S} where S is a type name for a tuple-valued type with arity m then <map(ai:Ti)> is replaced by

```
SET_OF(S(_Z,_Z1,...,_Zm),
   ( A(C,ai,_X,_Z),
     hasType(S,S(_Z,_Z1,...,_Zm))),
   _Yi)
```

- If Ti is a set type {*S} where S is a type name of a tuple type with class D then <map(ai:Ti)> is replaced by

```
SET_OF(REF(S,_Z),
      ( A(C,ai,_X,_Z),
        In(_Z,D)),
      _Yi)
```

- If Ti is a tuple type with arity m then the macro is replaced by

```
_Yi = Ti(_Y,_Z1,...,_Zm),
A(C,ai,_X,_Y),
hasType(Ti,_Yi)
```

- Finally, pointer types *Ti where Ti is a record type with class D are mapped to the condition

```
(_Yi = REF(Ti,_Y),
A(C,ai,_X,_Y),
In(_Y,D);
_Y = null_value)
```

The operator ';' stands for a logical disjunction. There will be no backtracking on this disjunction. Thus, _Y will either be bound to a term REF(.,.) or to the special value null_value.

The PORT clauses specify those subsets of types which are of interest to the application program. A port definition

```
PORT v: {T| a1.a2...an=$P}
```

is compile to the clause

```
askPort(_S,v,_P) :-
  SET_OF(_X,
     (hasType(T,_X),
      path(_X,[a1,a2,...,an],_P),
     _S).
```

The predefined predicate path evaluates the path expression a1.a2...an starting from _X. Note that the parameter $P becomes an argument of the askPort predicate. It is instantiated by the application program when calling the goal askPort. The result is returned in the first argument.

The restriction in the port definition can easily be extended to contain several conditions. Moreover, one can allow a constant or a second path expression instead of the parameter on the right-hand side of the equality.

[1] We adopt the syntax of Prolog to denote the clauses. Variables start with an underscore. The meta predicate SET_OF(x,c,s) evaluates s as the set of all elements x satisfying the condition c

```
hasType(String,String(_S)) :-
  In(_S,String).

hasType(Project,Project(_P)) :-
  In(_P,Project).

hasType(DeptType,DeptType(_D,_DN,_M)) :-
  In(_D,Department),
  _DN = String(_Z1),
  A(Department,_D,deptName,_Z1),
  hasType(String,_DN),
  _M = REF(EmpType,_Z2),
  A(Department,_D,head,_Z2),
  In(_Z2,Employee).


hasType(EmpType,EmpType(_E,_N,_PS,_DT)) :-
   In(_E,Employee),
   _N=String(_Z1),
   A(Employee,_E,name,_Z1),
   hasType(String,_N),
   SET_OF( Project(_Z2),
     (A(Employee,_E,project,_Z2),
      hasType(Project,Project(_Z2))),
      _PS),
   _DT = DeptType(_D,_DN,_M),
   A(Employee,_E,dept,_D),
   hasType(DeptType,_DT).

askPort(_S,e,_N) :-
   SET_OF(_X,
       (hasType(EmpType,_X),
       path(_X,[dept,deptName],_N),
       _S).
```

Figure 2: Logic program for the example

3.1 Mapping of the Example

The definition of `hasType` for the running example is presented in Figure 2.

The values of the imported concepts are represented as unary terms, e.g. `String("Peter Wolfe")`. Values of complex terms have more components according to the type definition. For example,

```
EmpType(e2341,String("Peter Wolfe"),
        [Project(p1),Project(p2)],
        DeptType(d41,String("Marketing"),
         REF(EmpType,e3331)))
```

is the term representing a value of `EmpType`. Values of set types like {`Project`} are sequences of values of the member type enclosed by brackets. The component for the `dept` attribute is avalue of type `DeptType`. This shows the representation of pointers as terms `REF(T,X)` where `X` is the identifier of the value (of type `T` pointed to. The identifier is always the first component of a term `T(X,...)`. All identifiers are constants from the database.

4 Properties of Interfaces

Termination of the logic program is guaranteed, and the types defined in API modules can be compared with the database schema and with each other.

4.1 Termination

On first sight, the generated logic program is recursive in the `hasType` clause and it contains complex terms as arguments. Thus, one has to ensure termination when evaluation it by the SLD strategy for logic programs.

Fortunately, if one makes sure that the types in the API module are defined non-recursively, then there is a partial order on the type names. If a type definition for `T1` uses a type `T2` on the right-hand side, then $T1 > T2$ holds. The definition of the logic program generator propagates this property to all clauses of the `hasType` predicate: if `hasType(T,.)` occurs in the condition of a clause `hasType(R,.)` then `T` must be smaller than `R`. Consequently, the logic program terminates on each goal `hasType(T,X)`[2].

A corrolar of this proposition is the finiteness of the sets interpreting the types in the API module.

4.2 Reasoning Services

The constructs in the API module were deliberately choses to be conformant with the concept language dialect of Buchheit *et al.* 1994. A couple of reasoning services are possible, each determing a different set of axioms to be reasoned about. We illustrate only one service, type checking against the database.

The type definitions in an API module make assumptions about the structure of the imported database concepts. In the example of Figure 1, the concepts `Employee` must at least have three attribute categories `name`, `project`, and `dept`. For the `Department` concept, two attributes categories `deptName` and `head` are required. Moreover, attribute cardinalities for the answer objects are stated:

- a set-valued attribute like `project` does not induce any cardinality constraint;
- a pointer-valued attribute like `head` restricts the the number of attribute fillers to be less or equal 1;
- the remaining attributes like `dept` must have exactly one filler.

Please note that these properties apply to the defined concepts like `EmpType` (ET) and not to the imported concepts like `Employee` (E). The concept language expression is:

$$ET = E \sqcap (= 1\ name.S) \sqcap (= 1\ dept.DT)$$
$$DT = D \sqcap (= 1\ deptName.S) \sqcap (\leq 1\ head.E)$$

As prescribed by the logic program, the pointer-valued attribute `head` of `DeptType` is not refering to `EmpType` directly but to its associated class `Employee`. Thereby, circular concept definitions are prevented.

These equalities for the type definitions are true provided the database schema has a schema consistent to it. At least it has to fulfill the following "well-typedness" axioms[3]:

[2] One has to assume that the underlying database is finite. This is however a standard assumption with databases.

[3] The symbol $\top$ stands for the most general concept.

$$E \sqsubseteq \forall name.\top \sqcap \forall project.\top \sqcap \forall dept.\top$$
$$D \sqsubseteq \forall deptName.\top \sqcap \forall head.\top$$

One can check this by adding it to the database schema and checking its consistency. The service would just make sure that all referenced attributes are defined in the database schema.

With a stricter regime, one can demand that the database schema must have the same or sharper cardinality constraints and that the well-typedness is refined to the concepts appearing as attribute types in the API module:

$$E \sqsubseteq ET \sqcap \forall name.S \sqcap \forall project.P \sqcap \forall dept.DT$$
$$D \sqsubseteq DT \sqcap \forall deptName.S \sqcap \forall head.ET$$

Here, the database schema has to fulfill the structure of the types in the API module. Consequently, all instances of the database concepts will apply to the type definitions. The type definitions would only project on the attributes of interest. Even if one regards this as a too narrow coupling, the test on consistency of the above axioms with the database schema returns useful information to the designer of an API module.

5 Programming Language Embedding

From the API modules, programming language data types can be derived. Currently, a prototype for the C++ language is implemented. The tuple types are mapped to C++ structures, the sets to linked lists, and the pointers to C++ pointers. While the concept language view makes no difference between pointer-valued attributes (like `*EmpType` and their associated class `Employee`, the representation within the application program is very different:

- A value `Employee(X)` is stored in a variable with C++ type `char*` because `X` is just an string representing a database constant.
- A value `REF(EmpType,X)` is represented as a main memory address pointing to the location where the value `EmpType(X,...)` is stored. This allows the application program to follow attribute chains by fast main memory adressing.

Communication between an applications program and the database is routed through the ports. The term representations of port `p` returns in argument `s` of the query `askPort(s,p,x1,...,xn)` are read by the application program. The arguments `x1,...,xn` contain the constants for the selection conditions[4]. The "read" procedure, basically a simple term parser, stores the values in the C++ data structures. Both the parser and the data structures are generated from the API module by a compiler. Since the `askPort` predicate can only return syntactically correct terms, an exception handling for malformed answers is superfluous.

[4] Like for types the properties of port definitions can be investigated within the framework of concept languages. If the parameters `x1,...xn` are known, then the selection conditions are path agreements. Moreover one may allow path expressions of the form $a_1.a_2...a_r = b_1.b_2...b_s$ without compromising on the theoretical complexity of the reasoning.

6 Related Work

Lee and Wiederhold 1994 present a mapping from relational database schemas to complex objects. It is more general in the sense that arbitrary arities of the relations are allowed. In this paper, we assume a totally normalized schema of the database consisting of unary relations for class membership and binary relations for attributes. The advantage of our approach is that the algorithm for the generation of the logic programm can be kept free of reasoning on foreign key dependencies.

Plateau *et al.* 1992 present the view system of O_2 as complex type definitions coupled with the database types and with prescriptions for graphical display. The type system contained in the O_2 data model. Reasoning on type correctness is done by the compiler.

The Interface Definition Language IDL by Nestor *et al.* 1992 has four type constructors for records, lists, sets, and classes (unions of different record types). The base types represents boolean, integers, rationals, and strings. The values are transfered between two programs by using a term representation similar to ours. The difference is the missing formal relationship between type definitions and (database) concepts.

A recent proposal by Papakonstantinou *et al.* 1994 encodes all type information with the term representation of a value. An application program has to provide generic data structures capable of storing arbitrary values (though restricted to a fixed set of base types). The advantage is the flexibility of the approach. A disadvantage is missing compile time type checking.

Persistent object systems, esp. Tycoon by Matthes 1993, "lift" the type systems of information sources and application programs into a single type system. Because of the heterogenous information sources, the approach is more general than in O_2. Reasoning is again restricted to type checking.

7 Conclusion

We defined API modules as mediators between application programs and databases. Both programming language data types and database queries are generated from the module description. The language is simple enough to guarantee termination of the query and efficient reasoning on the type definitions. Pointer types are introduced to simulate recursive datatypes and find a natural counterpart in the database query.

In future, we plan to eliminate the distinction between application program and database in the API modules. Application programs can serve as a "database" provided they offer the ability the interpret queries on their information. Then, information flow design between a collection of programs can be supported by reasoning on the relationship between the type definitions.

Acknowledgement. Many thanks to Claudia Welter and Martin Staudt for attacking weak points in earlier versions of this paper.

References

[Buchheit *et al.*, 1994] M. Buchheit, M.A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between queries to object-oriented databases. *Information Systems*, 19(1):33–54, 1994.

[Lee and Wiederhold, 1994] B.S. Lee and G. Wiederhold. Outer joins and filters for instantiating objects from relational databases trough views. *IEEE Trans. Knowledge and Data Engineering*, 6(1):108–119, 1994.

[Matthes, 1993] F. Matthes. Persistente Objektsysteme. Springer-Verlag, 1993.

[Papakonstantinou *et al.*, 1994] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. Submitted paper, 1994.

[Plateau *et al.*, 1992] D. Plateau, P. Borras, D. Leveque, J. Mamou, and D. Tallot. Building user interfaces with Looks. In F. Bancilhon, C. Delobel, P. Kannelakis (eds.):*Building an Object-Oriented Database System - The Story of O2*, Morgan-Kaufmann, 256–277, 1992.

[Nestor *et al.*, 1992] J. R. Nestor, J. M. Newcomer, P. Giannini, and D. L. Stone. IDL - The language and its implementation. Prentice Hall, 1990.