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.

- Hand-coded 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
From the database point of view, an API module is generated at runtime. The tokens are represented by terms conforming the type definitions. These views are encoded as logic programs 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 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

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(_,X1,.. ,YK) :-
   In(_,X,C),
   <map(ai:Ti)>,
   ...
   <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,_,Z1,..,Zm),
  ( A(C,ai,X1,Z2),
   hasType(S,S(_,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,X1,Z),
     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 compiled to the clause

```
askPort(_S,v,P) :-
   (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.

---

1We 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.
The definition of \texttt{hasType} for the running example is presented in Figure 2.

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

\begin{verbatim}
EmpType(e2341,String("Peter Wolfe"), [Project(p1),Project(p2)], DeptType(d41, String("Marketing"), REF(EmpType,e3331)))
\end{verbatim}

is the term representing a value of EmpType. Values of set types like \texttt{[Project]} are sequences of values of the member type enclosed by brackets. The component for the \texttt{dept} attribute is a value of type \texttt{DeptType}. This shows the representation of pointers as terms \texttt{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,\ldots). All identifiers are constants from the database.

3.1 Mapping of the Example

The definition of \texttt{hasType} for the running example is presented in Figure 2.

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

\begin{verbatim}
EmpType(e2341,String("Peter Wolfe"), [Project(p1),Project(p2)], DeptType(d41, String("Marketing"), REF(EmpType,e3331)))
\end{verbatim}

is the term representing a value of EmpType. Values of set types like \texttt{[Project]} are sequences of values of the member type enclosed by brackets. The component for the \texttt{dept} attribute is a value of type \texttt{DeptType}. This shows the representation of pointers as terms \texttt{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,\ldots). 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 \texttt{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 \(\geq\) T2 holds. The definition of the logic program generator propagates this property to all clauses of the \texttt{hasType} predicate: if \texttt{hasType}(T,\_), then T must be smaller than R. Consequently, the logic program terminates on each goal \texttt{hasType}(T,\_).

A corollary 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 chosen to be conformance with the concept language dialect of BUCHHEIT et al. 1994. A couple of reasoning services are possible, each determining 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 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:

\[
\begin{align*}
ET &= E \cap (\leq 1 \text{ name}.S) \cap (\leq 1 \text{ dept}.DT) \\
DT &= D \cap (\leq 1 \text{ deptName}.S) \cap (\leq 1 \text{ head}.E)
\end{align*}
\]

As prescribed by the logic program, the pointer-valued attribute head of DeptType is not referring 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:\footnote{One has to assume that the underlying database is finite. This is however a standard assumption with databases.}:

\[
\begin{align*}
\text{Figure 2: Logic program for the example}
\end{align*}
\]
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:

\[
E \subseteq \forall \text{name}. T \land \forall \text{proj ect}. T \land \forall \text{dept}. T
\]

\[
D \subseteq \forall \text{dept Name}. T \land \forall \text{head}. T
\]

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 has to fulfill the structural relationship between application program and database in the API modules. Application programs can serve as a mediator between application programs and databases. Both programs 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.

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 program can be kept free of reasoning on foreign key dependencies.

Plateau et al. 1992 present the view system of the O2 as complex type definitions coupled with the database types and with prescriptions for graphical display. The type system contained in the O2 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 represent boolean, integers, rationals, and strings. The values are transferred 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 heterogeneous information sources, the approach is more general than in O2. 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


