OR-SML: A Functional Database Programming Language for Disjunctive Information and Its Applications

Elsa Gunter and **Leonid Libkin**[⋆]

AT&T Bell Laboratories
600 Mountain Ave., Murray Hill, NJ 07974, USA
email: {elsa,libkin}@research.att.com

Abstract. We describe a functional database language OR-SML for handling disjunctive information in database queries, and its implementation on top of Standard ML [12]. The core language has the power of the nested relational algebra, augmented by or-sets which are used to deal with disjunctive information. Sets, or-sets and tuples can be freely combined to create objects, which gives the language a greater flexibility. It is configurable by user-defined base types, and can be used independently or interfaced to other systems built in ML. We give examples of queries which require disjunctive information (such as querying incomplete or independent databases) and show how to use the language to answer these queries.

1 Introduction

Disjunctive information in databases. There are many reasons why disjunctive information may be present in databases. One arises in the areas of design, planning, and scheduling, as was shown in [8]. For example, consider a design template used by an engineer, see figure 1. The template may indicate that the whole part being built consists of two subparts, A and B, but the component A can be built by either module $A1$ or module $A2$. Such a template is structurally a complex object whose component A is the collection containing $A1$ and $A2$; however, its meaning is not $A1$ *and* $A2$ as in the usual database interpretation of sets, but rather $A1$ *or* $A2$. Moreover, B, $A1$ and $A2$ can in turn have a similar structure. In figure 1, vertical lines indicate subparts that must be included, and the slopping lines indicate possible choices. For example, B consists of $B1$ and $B2$. Further down the tree, $B1$ is either w or k and $B2$ is either l or m. Each smallest subpart (a leaf of the tree) may have some parameters like cost and reliability, which affect the properties of a completed design.

A designer employing such a template should be allowed to query the structure of the template, for example, by asking what are the choices for component A, or what is the most reliable choice for component $B2$. We call such queries

[⋆] Partial support was provided by NSF Grant IRI-90-04137 and AT&T Doctoral Fellowship, while this author was a graduate student at the University of Pennsylvania.

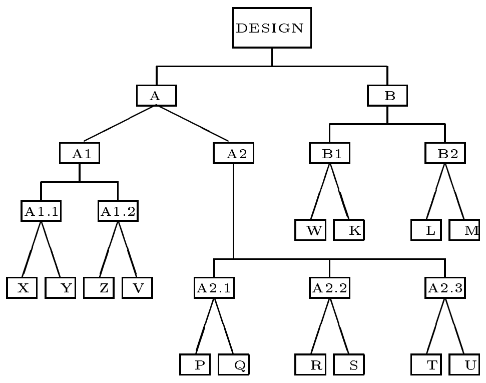

Fig. 1. An incomplete design

structural. They ask about the structure of the object, and they can be easily answered in most of the languages for complex objects.

On the other hand, the designer should also be allowed to query about possible completed designs. Such queries are called *conceptual*, as they ask questions about objects which are not stored in the database, but only represented by those that are. A query that asks to compute the number of completed designs is an example of a conceptual query. For each particular incomplete design this number can be calculated just by looking at the structure and multiplying the numbers of possibilities each disjunctive set contributes.

However, there are problems with this approach. In particular, it is not robust. In existing languages for complex objects a modest change to the structure of the incomplete design will require creating a new query to answer the same question. This query is relatively simple, but it is conceivable that a designer would want to ask if it is possible to complete design using $\$n$ and achieving reliability of at least $r\%$. Writing such a query seems to be a formidable problem that may take hours of programming, and it is even less robust: if the design object is changed slightly, there is no way to reuse the old query to answer the new questions.

Disjunctive information may also show up in the form of *interpretation* of already existing objects with respect to queries being asked. Assume that we have two relations in a university databases. One is the relation of employees, and the other is the relation of teachers of a basic course CS1, that can be taught only by teaching assistants (TAs). Suppose we want to find the set of TAs (or an approximation thereof) assuming that all TAs are employees. In this case we know that *all* teachers of CS1 are TAs, and each TA is *one of* the employees. Interpreting the relation of employees as a disjunctive set helps answer queries about various groups of employees, like TAs, as will be demonstrated later.

There are various other forms in which disjunctive information appears in database applications. For example, when combining a number of databases, there may be two records from different databases with the same values of the keys (like SS#) and conflicting values of other attributes (like Age). In this case in the combined database we must store the fact that the value of the Age attribute is one of the values from the different databases. Another kind of examples arises in the problem of proof planning in automated theorem provers. We refer the reader to the full paper [7] in which we discuss these problems.

Goals of the paper. We describe a functional language, OR-SML, for querying databases with incomplete and disjunctive information. It is capable of solving a number of problems that typically accompany disjunctive information; in particular, the problems we discussed above. To handle disjunctive information, we allow a new type constructor of *or-sets* (hence the name – OR-SML). Or-sets have been studied in [8, 10, 14]. Or-sets are in essence disjunctive information, but they are distinguished from the latter by having two distinct interpretations. The *structural* level concerns the precise way in which an or-set is constructed. The *conceptual* level sees an or-set as representing an object which is equal to some member of the or-set. For example, the or-set $\langle 1, 2, 3\rangle$ is structurally a collection of numbers; however, conceptually it is either 1, 2, or 3. (Angle brackets $\langle\rangle$ are used for or-sets and $\{\}$ for the usual sets.) The language OR-SML supports both views of or-sets and therefore can answer conceptual queries.

The language design. Our language is based on the functional paradigm. Design of functional database query languages has been studied extensively in the past few years and proved very useful. (See, for example [1, 2, 10, 11, 13, 16].) Functional languages have certain advantages over logical languages for complex objects. They have clear syntax, they can be typechecked, their semantics is generally easy to define and they allow a limited form of polymorphism.

Since entries in databases are allowed to be or-sets possibly containing other sets, the databases are no longer in the first normal form. Therefore, we have to deal with nested relations, or complex objects. The language we describe contains the nested relational algebra as a sublanguage. The standard presentations of the nested relational algebra (cf. [15]) have a cumbersome syntax. Therefore, we have decided to follow the approach of [2], which gives a very clean and simple language that has precisely the expressive power of the nested relational algebra. The language obtained from the nested relational algebra by adding appropriate primitives dealing with or-sets was called *or-$\mathcal{NRA}$* in [10].

One of the problems that should be addressed during the language design is a mechanism for incorporating both structural and conceptual queries into the same language. It was shown in [10] that conceptually equivalent objects can be reduced to the same object by repeated applications of just three *or-$\mathcal{NRA}$* operators which will be described later. The induced normal form is *independent* of the sequence of applications of these operators. Therefore, one can take the conceptual meaning of any object to be its normal form under the rewriting induced by the those operators. Consequently, a conceptual query language can be built by extending a structural language with a single operator `normal` which takes the input object to its normal form. A query at the conceptual level is then simply a query performed on normal forms.

The system OR-SML includes much more than just *or-$\mathcal{NRA}$*. Normalization is present as a primitive. Some arithmetic is added to elevate the language to the expressive power of the *bag* language $\mathcal{BQL}$ of [11]. This allows correct evaluation of aggregate functions. OR-SML is extensible with user-defined base types. It provides a mechanism for converting any user-defined functions on base types into functions that fit into the type system of OR-SML. It also gives a way "out of complex objects" into SML values. This is necessary, for example, if OR-SML is a part of a larger system and the OR-SML query is part of a larger computation that needs to analyze the result of the query to proceed. OR-SML comes equipped with libraries of derived functions that are helpful in writing programs or advanced applications such as querying independent databases.

We chose Standard ML (SML) as the basis for our implementation in order to combine the simplicity of *or-$\mathcal{NRA}$* queries with features of a functional programming language [12]. OR-SML benefits from it in a number of ways:

1. OR-SML queries may involve and become involved in arbitrary SML procedures. The presence of higher-order functions in SML allows SML functions to be arguments to queries and queries to be arguments to SML functions.
2. OR-SML is implemented as a library of modules in SML. This allows the user to build just the database language as an independent system, or to interface it to other systems built in SML. Using this feature, we were able to connect OR-SML to an existing interactive theorem prover.
3. One interacts with OR-SML by entering declarations and expressions to be evaluated into the top-level read-evaluate-print loop of SML. The results are then bound to SML identifiers for future use.
4. The SML module system makes the implementation of different parts of the language virtually independent and easily modifiable.

In the next section we give a quick overview of OR-SML. In section 3 we show how OR-SML can be used to answer some of the problems we mentioned above. All examples in this paper are obtained from a working version of OR-SML.

2 An overview of OR-SML

The core language. The theoretical language upon which OR-SML is based was developed by Libkin and Wong in [10]. We describe this core language, called *or-$\mathcal{NRA}$*, and show how it is built on top of Standard ML. We have changed the names of all constructs of *or-$\mathcal{NRA}$* to the names that are used in OR-SML.

The *object types* are given by the following grammar:

$$t ::= b \mid \mathit{unit} \mid \mathit{bool} \mid t \times t \mid \{t\} \mid \langle t \rangle$$

Here b is one of base types (which in OR-SML include `int`, `string`, and a user-supplied SML type), *unit* is a special type whose domain has a unique element denoted by $()$, *bool* is the type of booleans, $t \times s$ is the product type, whose objects are pairs of objects of types t and s. The set type $\{t\}$ denotes finite sets of elements of t and the or-set type $\langle t\rangle$ denotes finite or-sets of elements of t.

The specific types of the *or-$\mathcal{NRA}$* operators are given by the rules in the table in Fig. 1. All occurrences of s, t and u in that table are object types. Let

General operators
$\frac{}{\texttt{p1} : s \times t \to s}$ $\quad \frac{}{\texttt{p2} : s \times t \to t}$ $\quad \frac{}{\texttt{bang} : t \to unit}$ $\quad \frac{}{\texttt{eq} : t \times t \to bool}$ $\quad \frac{}{\texttt{id} : t \to t}$
$\frac{g : u \to s \quad f : s \to t}{\texttt{comp}(f,g) : u \to t}$ $\quad \frac{c : bool \quad f : s \to t \quad g : s \to t}{\texttt{cond}(c,f,g) : s \to t}$ $\quad \frac{f : u \to s \quad g : u \to t}{\texttt{pair}(f,g) : u \to s \times t}$
Operators on sets
$\frac{}{\texttt{emptyset} : unit \to \{t\}}$ $\quad \frac{}{\texttt{sng} : t \to \{t\}}$ $\quad \frac{}{\texttt{union} : \{t\} \times \{t\} \to \{t\}}$
$\frac{f : s \to t}{\texttt{smap} f : \{s\} \to \{t\}}$ $\quad \frac{}{\texttt{pairwith} : s \times \{t\} \to \{s \times t\}}$ $\quad \frac{}{\texttt{flat} : \{\{t\}\} \to \{t\}}$
Operators on or-sets
$\frac{}{\texttt{emptyorset} : unit \to \langle t\rangle}$ $\quad \frac{}{\texttt{orsng} : t \to \langle t\rangle}$ $\quad \frac{}{\texttt{orunion} : \langle t\rangle \times \langle t\rangle \to \langle t\rangle}$
$\frac{f : s \to t}{\texttt{orsmap}\ f : \langle s\rangle \to \langle t\rangle}$ $\quad \frac{}{\texttt{orpairwith} : s \times \langle t\rangle \to \langle s \times t\rangle}$ $\quad \frac{}{\texttt{orflat} : \langle\langle t\rangle\rangle \to \langle t\rangle}$
Interaction of sets and or-sets
$\frac{}{\texttt{alpha} : \{\langle t\rangle\} \to \langle\{t\}\rangle}$

Fig. 2. *or-$\mathcal{NRA}$* Type Inference of OR-SML Terms

us briefly recall the semantics of these operators. $\texttt{comp}(f, g)$ is composition of functions f and g. First and second projections are called `p1` and `p2`. $\texttt{pair}(f, g)$ is pair formation: $\texttt{pair}(f, g)(x) = (f(x), g(x))$. `id` is the identity function. `bang` always returns the unique element of type *unit*. $\texttt{cond}(c, f, g)(x)$ evaluates to $f(x)$ if condition c is satisfied and to $g(x)$ otherwise.

The semantics of the set constructs is the following. $\texttt{emptyset}()$ is the empty set. This constant also has name `empty`. $\texttt{sng}(x)$ returns the singleton set $\{x\}$. $\texttt{union}(x, y)$ is $x \cup y$. $\texttt{smap}(f)$ maps f over a set, that is, $\texttt{smap}(f)\{x_1, \ldots, x_n\} = \{f(x_1), \ldots, f(x_n)\}$. `pairwith` pairs the first component of its argument with every item in the second component: $\texttt{pairwith}(y, \{x_1, \ldots, x_n\}) = \{(y, x_1), \ldots, (y, x_n)\}$. Finally, `flat` is flattenning: $\texttt{flat}\{X_1, \ldots, X_n\} = X_1 \cup \ldots \cup X_n$. The semantics of the or-set constructs is similar.

The operator `alpha` provides interaction between sets and or-sets. Given a set $\mathcal{A} = \{A_1, \ldots, A_n\}$, where each A_i is an or-set $A_i = \langle a^i_1, \ldots, a^i_{n_i}\rangle$, let $\mathcal{F}$ denote the set of all functions $f : \{1, \ldots, n\} \to \mathbb{N}$ such that $f(i) \leq n_i$ for all i. Then $\texttt{alpha}(\mathcal{A}) = \langle\{a^i_{f(i)} \mid i = 1, \ldots, n\} \mid f \in \mathcal{F}\rangle$.

We shall need some of the SML syntax. In SML, `val` binds an identifier and `-` is the prompt, so `- val x = 2;` binds `x` to `2` and `val x = 2 : int` is the SML response saying that `x` is now bound to 2 of type `int`. `fun` is for function declaration. Functions can also be created without being named by using the construct `(fn x =>` *body*(x)`)`. If a function is applied to its argument and the result is not bound to any variable, then SML assigns it a special identifier `it` which lives until it is overridden by the next such application. For example, the SML response to `- factorial 4;` is `val it = 24 : int`. `let` ... `in` ... `end` is used for local binding. The `[...]` brackets denote lists; `""` is used for strings.

Let us now describe how OR-SML constructs are represented over SML. Every complex object has type `co`. We refer to the type of an object or a function in *or-$\mathcal{NRA}$* as its *true type*. True types of objects can be inferred using the function `typeof`. They are SML values having type `co_type`. When OR-SML prints a complex objects together with its type, it uses `::` for the true type, as `: co` is used to show that the SML type of the object is `co`. Values can be input by functions `create : string -> co` (or `make : unit -> co` for interactive creation, if the input needs to be broken over several lines). For example:

```
- val a = create "{ <1,2,3>, <4,5,6>, <7,8> }";
val a = {<1, 2, 3>, <7, 8>, <4, 5, 6>} :: {<int>} : co
```

The order in which elements appear in a set (or-set) is irrelevant. The order of elements of `a` was changed as the result of the duplicate elimination algorithm.

Typechecking is done in two steps. Static typechecking is simply SML typechecking; for example, `union(a,a,a)` causes an SML type error. However, since all objects have type `co`, the SML typechecking algorithm can not detect all type errors statically. For example, SML will see nothing wrong with `union(a,(create "5"))` even though the true types of its arguments are $\{\langle int\rangle\}$ and *int*. Hence, this kind of type errors is detected dynamically by OR-SML and an appropriate exception is raised. In our example, OR-SML responds by `uncaught exception Badtypeunion`.

The language can express many functions commonly found in query languages. Among them are boolean connectives, membership and subset tests, difference, selection, cartesian product and their counterparts for or-sets, see [2, 10]. These functions are included in OR-SML in the form of a structure called `Set`. Some examples are given below.

```
- alpha (create "{<1,2>,<2,3>}");
val it = <{2}, {1, 2}, {1, 3}, {2, 3}> :: <{int}> : co
- val x1 = create "{1,2}";
val x1 = {1, 2} :: {int} : co
- smap (pair(id,id)) x1;
val it = {(1, 1), (2, 2)} :: {int * int} : co
- Set.cartprod(x1,x1);
val it = {(1, 1), (1, 2), (2, 1), (2, 2)} :: {int * int} : co
```

OR-SML allows a limited access to user-defined base types. Values of these types have type `base` in OR-SML. The user is required to supply a structure containing basic information about the base type when a particular version of OR-SML is built. Objects of base type are printed in parentheses and preceded by the symbol `@`. They also must be input accordingly, so that the parser would recognize them.

There are a number of functions that make complex objects out of SML objects. For example, `mkintco: int -> co` and `mksetco : co list -> co` make an integer complex object, or a set whose elements come from a list of complex objects. These functions can be used as an alternative to `create` and `make`. OR-SML has a variety of printing styles which can be changed at will.

Normalization. Assume that an object x of type t contains some or-sets. What is x conceptually? Since we want to list all possibilities explicitly, it must be an object $x' : \langle t' \rangle$ where t' does *not* contain any or-set brackets. Intuitively, for any given object x we can find the corresponding x' but the question is whether we can do it in a coherent manner.

Such a way was found in [10]. Define the following rewrite system on types:

$$t \times \langle s \rangle \to \langle s \times t \rangle \quad \langle s \rangle \times t \to \langle s \times t \rangle \quad \langle \langle s \rangle \rangle \to \langle s \rangle \quad \{\langle s \rangle\} \to \langle \{s\} \rangle$$

Intuitively, we are trying to push the or-set brackets outside and then cancel them. With each rewrite rule we associate a basic OR-SML function as follows:

$$\begin{array}{rl} \texttt{orpairwith} : t \times \langle s \rangle \to \langle s \times t \rangle & \texttt{orpairwith1} : \langle s \rangle \times t \to \langle s \times t \rangle \\ \texttt{orflat} : \langle \langle s \rangle \rangle \to \langle s \rangle & \texttt{alpha} : \{\langle s \rangle\} \to \langle \{s\} \rangle \end{array}$$

where `orpairwith1` is "pair-with" with changed arguments. This function is definable in OR-SML.

If $s_1 \to \ldots \to s_n$, $n \geq 1$ by rewrites in the above rewrite system, then we write $s_1 \twoheadrightarrow s_n$. We associate with each sequence $s_1 \to \ldots \to s_n$ a *rewrite strategy* $r = [r_1, \ldots, r_{n-1}] : s_1 \twoheadrightarrow s_n$, where each r_i is the basic OR-SML function associated with $s_i \to s_{i+1}$. It is possible to "apply" a rewrite strategy $r : s_1 \twoheadrightarrow s_n$ to any object $x : s_1$, getting an object of type s_n which is denoted by $\mathsf{app}(r)(x)$. It can be obtained by using functions from the core language, see [10]. Moreover, the following result was proved in [10]:

Theorem (Coherence) *The rewrite system above is Church-Rosser and terminating. In particular, every type t has a unique normal form denoted $nf(t)$. Moreover, for any two rewrite strategies $r_1, r_2 : t \twoheadrightarrow nf(t)$ and any $x : t$, $\mathsf{app}(r_1)(x) = \mathsf{app}(r_2)(x)$.* □

This theorem tell us that a new primitive `normal : co -> co` can be added to OR-SML to give it adequate power to work with conceptual representations of objects. The true type of `normal` is $t \to nf(t)$ and its semantics is $\mathsf{app}(r)$ where $r : t \twoheadrightarrow nf(t)$. Normalization of types is represented by the function `normalize` of type `co_type -> co_type`. For example:

```
- val x = create "{(1,<2,3>),(4,<5,6>)}";
val x = {(1, <2, 3>), (4, <5, 6>)} :: {int * <int>} : co
- normalize (typeof x);
val it = <{int * int}> : co_type
- val y = normal x;
val y = <{(1,2),(4,5)}, {(1,3),(4,5)}, {(1,2),(4,6)}, {(1,3),(4,6)}> : co
```

Additional features. OR-SML has integers as a base type with a number of supported operations. Among them are two summation constructs. `sum` takes a function f of true type $s \to \mathit{int}$ and a set $\{x_1, \ldots, x_n\}$ of true type $\{s\}$ and returns $f(x_1) + \ldots + f(x_n)$. `orsum` acts similarly on or-sets. These operations endow a set language with the power of languages for nested *bags* as in [5, 11]. Equivalently, they allow us to define and correctly evaluate a number of aggregate functions. For example, `sum p2` is "add up all elements in the second column".

The system provides a way of making functions on user-defined base types into functions that fit into its type system. For example, if the user-defined base type is `real`, the function `apply_unary` will take a function `fn x => x + 1.0` of SML type `real -> real` and return the function `addone_co` of type `co -> co` whose semantics is $\lambda x.x + 1.0$. OR-SML also provides a way of converting binary functions and functions with arbitrary number of arguments into functions on complex objects. Predicates on base types can also be converted by means of `apply_test` to be used later with `cond` or `select`.

Structural recursion [1] is a very powerful programming tool for query languages. Even though it is not guaranteed to be well-defined, it is often helpful in writing programs or changing types of big databases (rather than reinputting them). It is available in OR-SML as two constructs `SR.sr` and `SR.orsr` that take an object e of type t and a function f of type $s \times t \to t$ and return a function `SR.sr`(e, f) of type $\{s\} \to t$ (or `SR.orsr`(e, f) of type $\langle s \rangle \to t$.) Their semantics is as follows: `SR.sr`$(e, f)\{x_1, \ldots, x_n\} = f(x_1, f(x_2, f(x_3, \ldots f(x_n, e) \ldots)))$ and similarly for `SR.orsr`. For example, to find the product of elements of a set, one may use structural recursion by first producing a function `set_mult: co -> co` as `val set_mult = SR.sr((create "1"),mult)` and then applying it to a set, say {`1,2,3,4,5`}, obtaining 120.

To support a form of persistence for databases, OR-SML provides means for writing lists of complex objects to files and reading them back in later. There are two modules for file I/O: one for binary files and one for ASCII files.

To enable the user to write programs to deal with the results of queries, OR-SML provides a way out of complex objects to the usual SML types. See the system manual [7] for details.

3 Applications of OR-SML

Querying incomplete design databases. Recall the example of an incomplete design from figure 1. Assuming that each smallest subpart has two parameters – its cost (of type *int*) and its reliability (of type *real*) – we can use or-sets to represent the incomplete design as an object in OR-SML as follows:

```
val design =
  (<{<('z', (13, @(0.95))), ('v', (14, @(0.955)))>,
     <('y', (20, @(0.98))), ('x', (21, @(0.999)))>},
    {<('p', (12, @(0.95))), ('q', (13, @(0.96)))>,
     <('s', (17, @(0.96))), ('r', (18, @(0.97)))>,
     <('t', (19, @(0.98))), ('l', (20, @(0.99)))>}>,
(<('k', (11, @(0.93))), ('w', (17, @(0.96)))>,
 <('l', (12, @(0.94))), ('m', (14, @(0.95)))>)) : co
```

Assume that we want to answer the following conceptual queries. How many completed designs are there? Is it possible to complete the design using \$62? What is the most reliable design that costs under \$$n$?

To answer these queries, we first infer the type of the normalized database.

```
val ndt = <({(string * (int * real))} *
    ((string * (int * real)) * (string * (int * real))))> : co_type
```

Guided by this type, we can write the cost and reliability functions for the completed designs. The function `cost` adds up all occurrences of integers in `ndt`. The function `reliability` can also be written straightforwardly in OR-SML for any type of connection of subparts. Assume parallel connection of $B1$ and $B2$ and series connection of A and B. Now to answer the first query, we write

```
- val nd = normal design; (* output omitted *)
- val num_choices = orsum (fn z => mkintco(1)) nd;
val num_choices = 48 : co
- orsmap (fn x => mkprodco ((cost x), (reliability x))) nd; (* output omitted *)
```

Thus, we have 48 completed designs. Notice that the query for `num_choices` is independent of the internal structure of the incomplete design. The output of the last query shows that the price range is from \$56 to \$82. Hence, the design can be completed with \$62.

To find the design that has the best reliability for a given cost, we first write a query `is_best` that selects the design with the best reliability from a given collection (this can be done using just the structural component of the language.) Then `bestunder` selects the most reliable design with a given cost limit.

```
- fun bestunder n = let val des_under_n = (Set.orselect
     (fn y => eq(mkintco(0), monus(cost(y),mkintco(n)))) nd)
in Set.orselect (fn y => is_best(y,des_under_n)) des_under_n end;
val bestunder = fn : int -> co
```

Applying this function yields some intersting results. The most reliable design (obtained by writing `bestunder 82`) costs only \$66. The most reliable design that costs less than \$62 has an actual cost of \$60. So, as it often happens, one does not have to buy the most expensive thing to get the best quality.

Summing up, we see that normalization is a very powerful tool for answering conceptual queries. Many queries that would be practically impossible to answer in just the structural language, now can be programmed in a matter of minutes in OR-SML.

Querying independent databases. Let us see how OR-SML can be used to solve a simple problem of querying independent databases. Consider the problem described in the introduction. The university database has two relations, Employees and CS1 (for teaching the course CS1) and we would like to compute the set of TAs. We know that only TAs can teach CS1 and every TA is a university employee. In this paper, we also assume that the Name field is a key.

Employees :

Name	Salary
John	15K
Mary	12K
Sally	17K

CS1 :

Name	Room
John	076
Jim	320
Sally	120

Note that the databases are inconsistent: Jim teaches CS1 and hence he is a TA and an employee, but there is no record for Jim in Employees. If we believe the Employees relation, then, to get rid of this anomaly, we must remove Jim from CS1. After that, we find an approximation of the set of TAs; that is, we find people who certainly are TAs and those who could be.

We always assume that all records have the same fields. It can be achieved by putting $-$ (null) into the missing fields or, in OR-SML representation, by using empty sets to represent nulls. This also allows us to take joins and meets of records. For example, $\boxed{\text{John}}\boxed{\text{15K}}\boxed{-} \vee \boxed{\text{John}}\boxed{-}\boxed{076} = \boxed{\text{John}}\boxed{\text{15K}}\boxed{076}$ and $\boxed{\text{John}}\boxed{\text{15K}}\boxed{-} \wedge \boxed{\text{John}}\boxed{-}\boxed{076} = \boxed{\text{John}}\boxed{-}\boxed{-}$. Notice that the join of two records is not necessarily defined.

In our solution we rely on the theory of partial information conveyed by means of partial orders which was worked out in [3, 4, 9, 10]. In particular, we use the fact that orders can be defined at arbitrary types, and consequently we have an OR-SML library of orderings and functions `meet`, `join` : $s \times s \to \langle s \rangle$ (the empty or-set indicates a non-existent join or meet; otherwise a singleton or-set is produced). Using these functions, it is easy to write a function (called `compatible`) to test whether two records have a join.

We treat Employees as a relation of possible upper bounds for TAs, so we make it an or-set. All entries in CS1 are TAs, so CS1 is a set. We represent the data as below, and remove the anomaly (Jim) using the test for a join `compat` as a parameter:

```
val emp = <('Mary', ({@(12.0)}, {})), ('John', ({@(15.0)}, {})),
   ('Sally', ({@(17.0)}, {}))> : co
val cs1 = {('John', ({},{76})), ('Sally', ({},{120})), ('Jim', ({},{320}))} : co
- fun remove_anomaly compat (R,S) = let fun compat_to_X (X,x) =
        Set.ormember(mkboolco(true),(orsmap (fn z => compat(z,x)) X));
   in Set.select (fn z => compat_to_X (R,z)) S end;
- val new_cs1 = remove_anomaly compatible (emp,cs1);
val new_cs1 = {('John', ({}, {76})), ('Sally', ({}, {120}))} : co
```

Now consider the solution proposed in [3] (see also [6]). Given an element $x \in$ CS1, let $y_1, \ldots, y_n$ be those elements in Employees that can be joined with x. Then $x' = \bigwedge_i (x \vee y_i)$ is called a *promotion* of x. (Intuitively, it adds all information about x from Employees.) The solution to the TA query is to take all promotions of elements in CS1 as "sure TAs" and elements of Employees not consistent with those promotions as "possible TAs". (We use the function big_meet that calculates the meet of a family.)

```
- fun promote compat (R,S) =
   let fun compat_to_x (X,x) = Set.orselect (fn z => compat(z,x)) X
   in alpha (smap (fn z => big_meet (orflat(orsmap (fn v => join(z,v))
  (compat_to_x (R,z))))) S) end;
- val promoted_cs1 = promote compatible (emp,new_cs1);
val promoted_cs1 = <{('John', ({@(15.0)}, {76})),
   ('Sally', ({@(17.0)}, {120}))}> : co
```

Thus, John from office 76 and Sally from office 120 are certainly TAs (and we know their salaries) and Mary could be a TA.

If the name field is not a key, this solution will not work: if there are several Johns in Employees, all will be joined with John from CS1, and when the meet is taken, the salary field is lost. But this is not what the information in the database tells us. We know that one John from Employees teaches CS1, but we do not know which John. Since either could be, the solution is to use an *or-set* to represent this situation. See [7] for details.

Acknowledgements: We would like to thank Peter Buneman and Limsoon Wong for many helpful discussions.

References

1. V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. of DBPL-91*, pages 9–19.
2. V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proc. ICDT-92*, pages 140–154. Springer, October 92.
3. P. Buneman, S. Davidson, A. Watters, A semantics for complex objects and approximate answers, *JCSS* 43:170–218, 1991.
4. P. Buneman, A. Jung, A. Ohori, Using powerdomains to generalize relational databases, *Theoret. Comp. Sci.* 91:23–55, 1991.
5. S. Grumbach, T. Milo, Towards tractable algebras for bags, *Proceedings of the 12th PODS*, Washington DC, 1993, pages 49–58.
6. C. Gunter, The mixed powerdomain, *Theoret. Comp. Sci.* 103:311–334, 1992.
7. E. Gunter and L. Libkin. A functional database programming language with support for disjunctive information, AT&T Technical Memo, 1993.
8. T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *Proc. of SIGMOD, Denver CO, May 1991.*
9. L. Libkin, A relational algebra for complex objects based on partial information, In *LNCS 495: Proc. of MFDBS-91*, Springer-Verlag, 1991, pages 36–41.
10. L. Libkin and L. Wong, Semantic representations and query languages for or-sets, *Proceedings of the 12th PODS*, Washington DC, 1993, pages 37–48.
11. L. Libkin and L. Wong, Some properties of query languages for bags, In *Proc. of DBPL-93*, Springer Verlag, 1994, pages 97–114.

12. R. Milner, M. Tofte, R. Harper, *"The Definition of Standard ML"*, The MIT Press, Cambridge, Mass, 1990.
13. A. Ohori, V. Breazu-Tannen and P. Buneman, Database programming in Machiavelli: a polymorphic language with static type inference, In *SIGMOD 89*.
14. B. Rounds, Situation-theoretic aspects of databases, In *Proc. Conf. on Situation Theory and Applications*, CSLI vol. 26, 1991, pages 229–256.
15. H.-J. Schek and M. Scholl, The relational model with relation-valued attributes, *Inform. Systems* 11 (1986), 137–147.
16. P.W. Trinder and P.L. Wadler, List comprehensions and the relational calculus, In *Proceedings of the Glasgow Workshop on Functional Programming*, pages 187–202.

This article was processed using the LaTeX macro package with LLNCS style