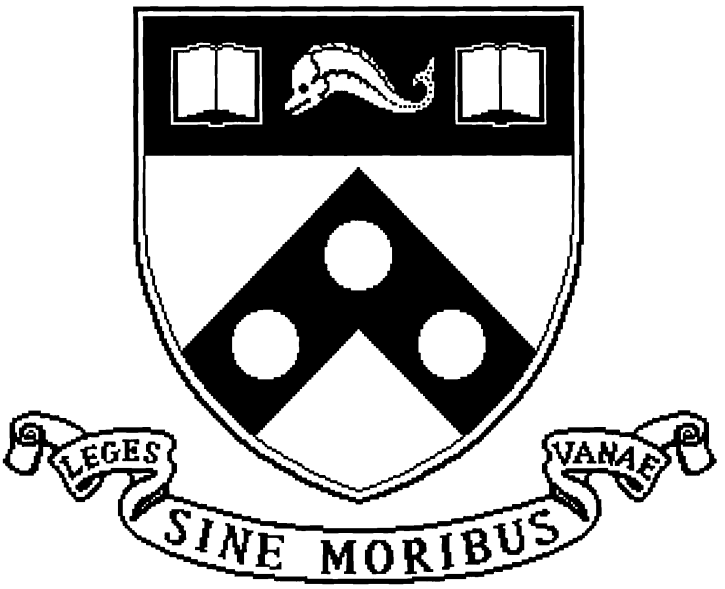
OR-SML: A Functional Database Programming Language for Disjunctive Information

MS-CIS-94-34
LOGIC & COMPUTATION 82

Elsa Gunter
Leonid Libkin



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

July 1994

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

Elsa Gunter
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974
E-mail: elsa@research.att.com

Leonid Libkin[1]
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104
E-mail: libkin@saul.cis.upenn.edu

Abstract

We describe a functional database language OR-SML for handling disjunctive information in database queries, and its implementation on top of Standard ML [21]. The core language has the power of the nested relational algebra, and it is augmented with 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. 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. Since the system runs on top of Standard ML and all database objects are values in the latter, the system benefits from combining a sophisticated query language with the full power of a programming language. OR-SML includes a number of primitives that deal with bags and aggregate functions. It is also configurable by user-defined base types. The language has been implemented as a library of modules in Standard ML. This allows the user to build just the database language as an independent system, or to interface it to other systems built in Standard ML. We give an example of connecting OR-SML with an already existing interactive theorem prover.

Key words: database programming languages, disjunctive information, functional languages, incomplete databases, independent databases, theorem provers.

[1]After September 1, 1994 at the same address as the first author. Partial support was provided by NSF Grant IRI-90-04137 and AT&T Doctoral Fellowship.

1 Introduction

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 [15]. For example, consider a design template used by an engineer. The template may indicate that component A can be built by either module B or module C. Such a template is structurally a complex object whose component A is the collection containing B and C; however, its meaning is not B *and* C as in the usual database interpretation of sets, but rather B *or* C. Moreover, B and C can in turn have a similar structure. 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. On the other hand, the designer should also be allowed to query about possible completed designs by asking if there is a cheap completed design. The same problem arises in a different guise in attempting to plan a proof strategy for a goal interactively in a theorem prover with a given database of related theorems and other information. For more details of this example, see Section 5.2.

Another example arises in the problem of combining a number of databases into one, or querying a number of independent databases. Assume that two databases are combined. One has people's names, Social Security numbers, ages and salaries, the other has names, Social Security numbers, ages and departments they work in. Suppose that for John with SS# 123-45-6789 the recorded age in one database is 24, but in the other is 27. We know that John can not be 24 *and* 27 simultaneously; hence in the combined database we need to store the fact that John is 24 *or* 27. That is, there is some uncertainty in the database that comes from conflicting information and shows up in the form of disjunctive information.

In this paper we describe a functional language, OR-SML, for querying databases with incomplete and disjunctive information. To handle disjunctive information, we allow a new type constructor of *or-sets* (hence the name – OR-SML). Or-sets have been studied in [15, 18, 24]. The original motivation for or-sets came from applications within design, planning, and scheduling areas. Or-sets are in essence disjunctive information, but they are distinguished from the latter by having two distinct interpretations. An or-set can either be treated at a *structural* level or at a *conceptual* level. 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.) For example, a query about possible choices for components of A is a query at the structural level, whereas a query asking if there is a completed design of a given cost is a conceptual level query. The language should support both.

Now let us describe our approach to the language design. First, 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 [2, 3, 19, 23, 27, 28].) Moreover, there are theoretical foundations for studying such languages [4, 13]. Functional languages have certain advantages over logical languages for complex objects. They have clear syntax (there is no need, for example, to give complicated syntactic rules for range restriction like in COL [1]), 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 in this paper contains the nested relational algebra as a sublanguage.

The standard presentations of the nested relational algebra [8, 25, 26] have a cumbersome syntax. Therefore, we have decided to follow the approach of [3], which gives a very clean and simple language that has precisely the expressive power of the nested relational algebra. The relational language introduced in [3] was based on earlier languages for lists [28, 29] and it was later generalized to other collection types [18, 19]. The language obtained from the nested relational algebra by adding appropriate primitives dealing with or-sets was called *or-$\mathcal{NRA}$* in [18].

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 [18] 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 in the paper. The induced normal form is *independent* of the sequence of applications of these operators. Moreover, given the type of any object, the type of its normal form can be found easily. Therefore, one can take the conceptual meaning of any object to be its normal form under the rewriting induced by these 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-$\mathcal{NRA}$*. First, normalization is present as a primitive. Some limited arithmetic is added to elevate the language to the expressive power of the *bag* language $\mathcal{BQL}$ of [19]. We show how bags and certain aggregate functions can be encoded. OR-SML also allows programming with structural recursion on sets and or-sets. The system 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 [21]. OR-SML benefits from it in a number of ways:

1. OR-SML queries may involve and become involved in arbitrary SML procedures. The usefulness of this is enhanced by the presence of higher-order functions in SML, allowing SML functions to be arguments to queries and queries to be arguments to SML functions. For an example of the value of this interaction see Section 5.2.

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. In Section 5.2, we take advantage of this ability to connect OR-SML to other an existing interactive theorem prover.

3. The stand-alone system version of OR-SML is implemented as a library loaded into the interactive system of SML, and as such is an interactive system itself. 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.

As of now, the system is suitable for querying small and medium size databases (hundreds of records), which are fairly common. To extend its capabilities to handle large databases, certain changes need to be made; in particular, optimizations in the presence of disjunctive information need to be added to OR-SML. As we have just mentioned, due to the modularity of the implementation, such changes can often be made without affecting the way the system looks to the end-user.

The paper is organized as follows. In the next section we describe the basic language whose relational component has precisely the expressive power of the nested relational algebra. We give a few examples of using the main constructs of the language. In Section 3 we explain the normalization process that gives us a way to describe the meaning of an object containing or-sets. We then proceed in Section 4 to describe additional features of the language such as arithmetic functions, programming with structural recursion over sets and or-sets, deconstruction of objects (that is, decomposing a complex object into a number of SML objects), I/O, adding user-defined base types and various libraries of derived functions. Finally, in Section 5 we demonstrate some applications of OR-SML in querying incomplete and independent databases. All examples in this paper are obtained from a working version of OR-SML.

2 The core language

The theoretical language upon which OR-SML is based was developed by Libkin and Wong in [18]. In this section 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 ranges over a collection of base types (which in OR-SML consists of `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. Their specific types as *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 us briefly recall the semantics of these operators. `comp`(f, g) is composition of functions f and g. First and second projections are called `p1` and `p2`. `pair`(f, g) is pair formation: `pair`$(f, g)(x) = (f(x), g(x))$. `id` is the identity function. `bang` always returns the unique element of type *unit*. `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. `emptyset`() is the empty set. This constant also has name `empty`. Similarly, the constant `emptyorset`() is available under the name `orempty`. `sng`(x) returns the singleton set $\{x\}$. `union`(x, y) is union of two sets x and y. `smap`(f) maps f over all elements of a set, that is, `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: `pairwith`$(y, \{x_1, \ldots, x_n\}) = \{(y, x_1), \ldots, (y, x_n)\}$.

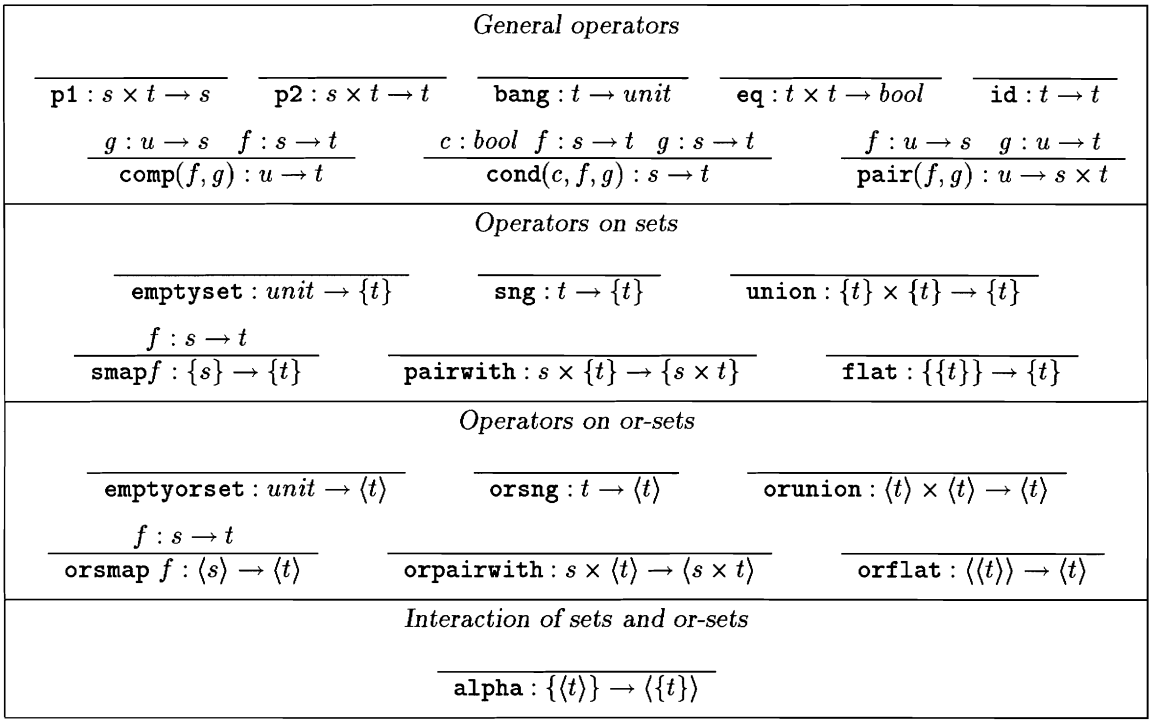


Figure 1: *or-NRA* Type Inference of OR-SML Terms

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 $A_i = \langle a_1^i, \ldots, a_{n_i}^i \rangle$, 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 $\texttt{alpha}(\mathcal{A}) = \langle \{a_{f(i)}^i \mid i = 1, \ldots, n\} \mid f \in \mathcal{F} \rangle$.

In what follows, we shall need some of the SML syntax. In SML, `val` binds an identifier and `-` is the SML 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 which is of type `int`. `fun` is for function declaration. Functions in SML 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 next such application. For example, `- factorial 4;` will cause SML response `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 shall refer to the type of an object or a function in *or-NRA* as its *true type*. True types of complex 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` (interactive creation). The function

`make` is terminated by typing ".". For example[1]:

```
- val a = make();
{ <1,2,3>, <4,5,6>, <7,8> }.
val a = {<1, 2, 3>, <7, 8>, <4, 5, 6>} :: {<int>} : co
- val b = create "(2,'abc')";
val b = (2, 'abc') :: int * string : co
```

Typechecking is done in two steps. Static typechecking is simply SML typechecking; for example, trying to call `union(a,a,a)` will cause 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,b)` even though the true types of `a` and `b` are $\{\langle int \rangle\}$ and $int \times string$. Hence, the remaining type errors are detected dynamically by OR-SML and an appropriate exception is raised. For instance, calling `union(a,b)` will make OR-SML respond by `uncaught exception Badtypeunion`.

The language we presented can express many functions commonly found in query languages. Among them are boolean *and*, *or* and negation, membership test, subset test, difference, selection, cartesian product and their counterparts for or-sets, see [3, 18]. These functions are included in OR-SML in the form of a structure called `Set`. Some examples of programming using the core language and functions from `Set` 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
- val x2 = create "{3,4}";
val x2 = {3, 4} :: {int} : co
- union(x1,x2);
val it = {1, 2, 3, 4} :: {int} : co
- Set.cartprod(x1,x2);
val it = {(1, 3), (1, 4), (2, 3), (2, 4)} :: {int * int} : co
```

OR-SML allows a limited access to user-defined base types. Values of these types have the user-defined SML 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, such as the name to be used as the true type of these base objects. One of the functions that is included in this user-supplied structure is parsing; its type is `string -> base`. If user-defined base types are used, then input of objects requires special care. Objects of base type are printed in parentheses and preceded by the symbol `@`. They also must be input accordingly if `make` or `create` is used. For example, in a version of OR-SML with real numbers, one would write:

```
- val a = create "@(2.5)";
val a = @(2.5) :: real : co
```

[1] Observe that the order in which elements appear in a set is changed in one of the examples. This emphasizes the fact that the order in which elements appear in a set (or-set) is irrelevant. The order changed in this particular case as the result of the duplicate elimination algorithm.

In the case of real numbers, the symbol "." plays a crucial role and can not be used to indicate the end of the input to `make`. There is a way to change the symbol whose meaning is "end of object". To do so, assign the new "end of object" symbol to `End_symb`. For example, after saying `End_symb := "!"` all inputs of complex objects must end with `!`. We shall use this symbol when working with reals.

There are also a number of functions that make complex objects out of SML objects. These are necessary, for example, if a user-defined base type is supplied without a parser. In this case objects can be created using constructor functions. For example:

```
- val a = [[2.5,3.7],[4.5,5.3]];
val a = [[2.5,3.7],[4.5,5.3]] : real list list
- val co_a = mksetco(map (fn z => mkorsco(map mkbaseco z)) a);
val co_a = {<@(2.5), @(3.7)>, <@(4.5), @(5.3)>} :: {<real>} : co
```

There are various styles for printing objects and object types. Some of them are better suited for printing normalized objects (see section 3), while others do not distinguish between sets and or-sets. All styles for objects and types can be freely combined, giving OR-SML a total of nine different printing styles. A new printer can be installed by using functions `printer` and `printer_type` of type `int -> unit`. These functions can be invoked at any time. Further details can be found in the system manual [12].

3 Normalization

As we discussed before, while an object $\langle 1, 2, 3\rangle$ is structurally just a set, conceptually it is a single integer which is either 1 or 2 or 3. Assume we are given an object $x : t$ where type t contains some or-set brackets. What is this object 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 there exists a coherent way of obtaining all objects which the given object can conceptually represent. Such a way was found in [18].

First, we define the following rewrite system on types:

$$t \times \langle s\rangle \to \langle s \times t\rangle \qquad \langle s\rangle \times t \to \langle s \times t\rangle$$

$$\langle\langle s\rangle\rangle \to \langle s\rangle \qquad \{\langle s\rangle\} \to \langle\{s\}\rangle$$

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:

$$\texttt{orpairwith} : t \times \langle s\rangle \to \langle s \times t\rangle \qquad \texttt{orpairwith1} : \langle s\rangle \times t \to \langle s \times t\rangle$$

$$\texttt{orflat} : \langle\langle s\rangle\rangle \to \langle s\rangle \qquad \texttt{alpha} : \{\langle s\rangle\} \to \langle\{s\}\rangle$$

where `orpairwith1` is "pair-with" with changed arguments. It can be implemented in OR-SML: $\texttt{orpairwith1}(x) = \texttt{orsmap}(\texttt{pair}(\texttt{p2},\texttt{p1}))(\texttt{orpairwith}(\texttt{pair}(\texttt{p2},\texttt{p1})(x)))$.

If $s_1 \to \ldots \to s_n$, $n \geq 1$ by rewrites in the above rewrite system, we write $s_1 \longrightarrow\!\!\!\!\!\rightarrow s_n$. We associate with each sequence $s_1 \to \ldots \to s_n$ a *rewrite strategy* $r = [r_1, \ldots, r_{n-1}] : s_1 \longrightarrow\!\!\!\!\!\rightarrow s_n$, 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

$r : s_1 \longrightarrow\!\!\!\!\!\rightarrow s_n$ to any object $x : s_1$, getting an object of type s_n which is denoted by $\mathsf{app}(r)(x)$. Such an object can be obtained by using functions from the core language, see [18]. Moreover, the following result was proved in [18]:

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 $r_1, r_2 : t \longrightarrow\!\!\!\!\!\rightarrow nf(t)$ and any $x : t$, $\mathsf{app}(r_1)(x) = \mathsf{app}(r_2)(x)$.* □

This theorem tell us that a new primitive `normal` can be added to OR-SML to give it adequate power to work with conceptual representations of objects:

$$\frac{}{\texttt{normal} :: t \to nf(t)}$$

The semantics of `normal` at type t is $\mathsf{app}(r)$ where $r : t \longrightarrow\!\!\!\!\!\rightarrow nf(t)$. Normalization of types and objects is represented in OR-SML by two functions: `normalize` of type `co_type -> co_type` and `normal` of type `co -> co`. 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
- normal x;
val it = <{(1, 2), (4, 5)}, {(1, 3), (4, 5)}, {(1, 2), (4, 6)}, {(1,3), (4, 6)}> : co
```

In section 5 we shall show how normalization can be used to solve the incomplete design problem from the introduction.

4 Additional features of the system

Arithmetic functions. OR-SML has integers as one of its base types. The following operations are available on integers: `plus`, `mult` and `monus` are addition, multiplication and modified subtraction respectively; they all have true type $int \times int \to int$. The semantics of modified subtraction is $\texttt{monus}(m, n) = \max\{0, m - n\}$. The function `gen` of true type $int \to \{int\}$ is given by $\texttt{gen}(n) = \{0, \ldots, n\}$. Finally, the summation construct `sum` takes in a function f of true type $s \to int$ 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.

There are several reasons why these constructs were chosen. First of all, these operations are precisely what must be added to a set language to endow it with the power of languages for nested *bags* as presented in [10, 19, 20]. Secondly, bag semantics is most often used for correct evaluation of aggregate functions like "total of column" etc. This now can easily be done in OR-SML. For example, `sum(fn x => mkintco(1))` is cardinality; `sum p2` is "add up all elements in the second column". If bags are represented as sets of "element–number of occurrences" pairs, all functions on bags from [19, 20] can now be modeled easily in OR-SML. For example, the difference of two bags can be implemented as follows:

```
fun bag_diff (x,y) = let
```

```
    fun equals_a  a = select (fn z => eq(p1(z),p1(a))) y
in Set.select (fn v => neg(eq(p2(v),mkintco(0))))
         (smap (fn z => mkprodco(p1(z),monus(p2(z),(sum p2 (equals_a z))))) x)
end;
val bag_diff = fn : co * co -> co
- val x = create "{('a',2),('b',4),('c',1)}";
val x = {('c', 1), ('a', 2), ('b', 4)} :: {string * int} : co
- val y = create "{('b',1),('b',2),('c',3),('d',1)}";
val y = {('d', 1), ('b', 1), ('b', 2), ('c', 3)} :: {string * int} : co
- bag_diff(x,y);
val it = {('b', 1), ('a', 2)} :: {string * int} : co
```

Primitives involving base types. Since the system allows user-defined base types, it must provide a way of making functions on those base types into functions that fit into the type system of OR-SML. For example, if the user-defined base type is `real`, there must be a way to have a function `addone_co : co -> co` whose semantics is addition of one to real numbers. Furthermore, there is a need for a mechanism to translate predicates on base types into predicates on complex objects that can be used with `cond` and `select`.

The solution to this problem is given by the function `apply` that takes a function `f : base list -> base` and returns a function from `co` to `co` representing the action of `f` on complex objects. For example, if `val f_co = apply f`, then `f_co` applied to a complex object $(r_1,(r_2,r_3))$ yields `f` $[r_1,r_2,r_3]$ in the form of a complex object.

In practice, most of the primitives on base types are unary or binary. Therefore, OR-SML has a special feature that allows you to apply binary and unary functions on base types by using functions `apply_unary` and `apply_binary`. For predicates, `apply_test` takes a function of type `(base -> bool)` and returns it in the form of a function on complex objects. For example,

```
- val addone_co = apply_unary ( fn x => x + 1.0);
val addone_co = fn : co -> co
- val x = create "{ @(2.5),@(4.5) }";
val x = {@(2.5), @(4.5)} :: {real} : co
- smap addone_co x;
val it = {@(3.5), @(5.5)} :: {real} : co
```

Structural recursion. Structural recursion on sets [2] is a very powerful programming tool for query languages. Unfortunately, it is too powerful because it is often unsafe. A function defined by structural recursion is not guaranteed to be well-defined, and well-definedness can not be generally checked by a compiler [4]. It is, however, often helpful in writing programs or changing types of big databases (rather than reinputting them), so we have decided to include structural recursion in OR-SML. Structural recursion on sets and or-sets is available to the user by means of 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 a function `SR.orsr`(e,f) of type $\langle s \rangle \to t$ respectively. 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 as follows:

```
- val fact = SR.sr((create "1"),mult);
```

```
val fact = fn : co -> co
- fact (create "{1,2,3,4,5}");
val it = 120 :: int : co
```

File I/O. To support a form of persistence for databases, OR-SML provides means for writing lists of complex objects to files and reading such lists back in later. There are two modules for file I/O in OR-SML: one working with binary files and one with ASCII files. Working with ASCII files is relatively safe: if there is any problem with reading an object, an exception will be raised. (It is not safe from editing). However, it requires a parser for objects of base type, because strings read from a file are parsed to create complex objects.

If a parser for objects of base type was not provided, then the binary input-output module must be used. Since binary I/O is an unsafe feature of Standard ML, all binary files are required to have the extension ".db". If it is not used, OR-SML will add it and ask if the operation should be continued. It is also possible to obtain the list of all files with extension ".db" in the current directory.

Deconstruction of complex objects. It may be the case that after evaluating a query, the user may need to write some program to deal with the result. Since all operations of OR-SML work with type `co`, there is a need to have a way out of complex objects to the usual SML types. The structure `DEST` contains some functions to deconstruct complex objects and obtain SML values. We refer the reader to the system manual [12] for details.

5 Applications of OR-SML

5.1 Querying incomplete design database

In this section we show an application of normalization of databases. We start with a database containing an incomplete design and ask certain queries about possible completed designs. We then show how to write these queries using normalization.

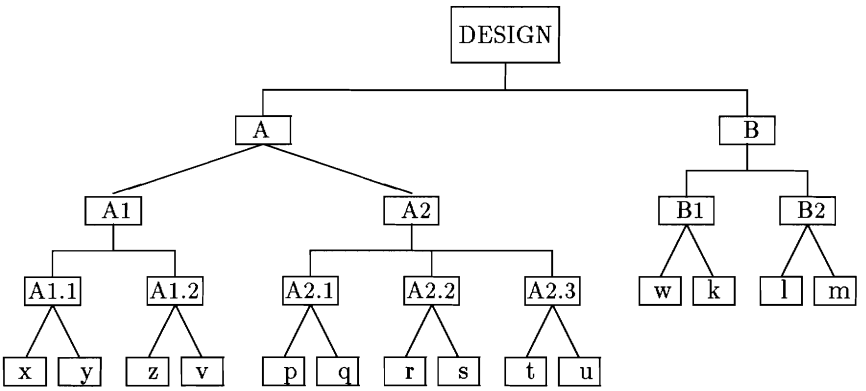
An example of an incomplete design is shown in figure 2. A part may consist of several subparts and each of them can be chosen from several possibilities with different parameters like price and reliability. In the picture, vertical lines indicate subparts that must be included, and the slopping lines indicate possible choices. For example, the whole design consists of parts A and B, where A is either $A1$ or $A2$ and B consists of $B1$ and $B2$. Further down the tree, $B1$ is either w or k and a $B2$ is either l or m.

Now assume that we know cost and reliability of each part that can make it into the completed designs (that is, for parts denoted by the lower case letters.) The incomplete design and costs and reliabilities of the individual parts are shown in figure 2.

We can create OR-SML values describing these smallest parts. For example,

```
val l = create "('l',(12,@(0.94)))";
val m = create "('m',(14,@(0.95)))";
```

Each part has true type *string* $\times$ (*int* $\times$ *real*). Now B and $A1$ can be created as



Part	Cost	Reliability
l	12	0.94
m	14	0.95
w	17	0.96
k	11	0.93
x	21	0.999
y	20	0.98
z	13	0.95
v	14	0.955
p	12	0.95
q	13	0.96
r	18	0.97
s	17	0.96
t	19	0.98
u	20	0.99

Figure 2: An incomplete design

```
- val B = mkprodco ((mkorsco [w,k]), (mkorsco [l,m]));
val B =
  (<('k', (11, @(0.93))), ('w', (17, @(0.96)))>,
   <('l', (12, @(0.94))), ('m', (14, @(0.95)))>) : co
- val A1 = mksetco [(mkorsco [x,y]), (mkorsco [z,v])];
val A1 =
  {<('z', (13, @(0.95))), ('v', (14, @(0.955)))>,
   <('y', (20, @(0.98))), ('x', (21, @(0.999)))>} : co
```

Parts $A2$ and A can be created in a similar fashion. Finally, the whole design can be created as

```
- val design = mkprodco (A,B);
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
```

Inferring type of `design` and normalizing it shows us the type of the database of completed designs.

```
- val ndt = normalize (typeof design);
val ndt =
  <({(string * (int * real))} *
    ((string * (int * real)) * (string * (int * real))))> : co_type
```

Hence, one can write the cost function which is the sum of costs of all parts. In this particular case it is

```
- fun cost X =
```

```
      let fun cost1 X = sum (fn z => p1(p2(z))) (p1 X)
          fun cost2 X =  p1(p2(p1(p2(X))))
          fun cost3 X =  p1(p2(p2(p2(X))))
  in plus(cost1(X), plus(cost2(X),cost3(X))) end;
val cost = fn : co -> co
```

Calculating reliability may be a bit harder because it depends on how different parts are connected. In the case of parallel connection of two parts with individual reliabilities r_1 and r_2, the reliability is calculated as $r_1 + r_2 - r_1 \cdot r_2$, whereas for the series connection it is $r_1 \cdot r_2$. To be able to operate with these functions, we must have them as functions from complex objects to complex objects. For example,

```
- val rmult = apply_op2 (fn (r1:real,r2:real) => r1 * r2);
val rmult = fn : co * co -> co
- val par_rel = apply_op2 (fn (r1:real,r2:real) => r1 + r2 - (r1 * r2));
val par_rel = fn : co * co -> co
```

Now we can calculate individual reliabilities for A, $B1$ and $B2$ and then, assuming parallel connection of $B1$ and $B2$ and series connection of A and B, calculate the realibility of a completed design as

```
- fun reliability X = rmult(relA(X), par_rel(relB1(X),relB2(X)));
val reliability = fn : co -> co
```

Now assume that we want to answer the following conceptual queries:

- How many completed designs are there?
- Which completed design has the best reliability?
- Which completed design that costs less than n dollars has the best reliability?

To answer these queries, we first *normalize* `design`, creating the or-set of all possible completed designs:

```
val nd = normal design; (* output omitted *)
```

Now it is possible to get all information about reliabilities and costs of completed designs by saying `orsmap cr nd` where `cr` is the function `fn x => mkprodco ((cost x), (reliability x))`. To answer our queries, we write

```
- orsum (fn z => mkintco(1)) nd;
val it = 48 : co
```

Hence, there are 48 completed designs. To find the one that has the best reliability, we write the following query

```
- fun is_better(x,y) = apply_test (fn (z:real) => z > 0.0) (rminus(x,y));
val is_better = fn : co * co -> co
- fun is_best (x,obj) = eq(orempty,
    (Set.orselect (fn y => is_better(reliability(y), reliability(x))) obj));
val is_best = fn : co * co -> co
- val select_best = Set.orselect (fn y => is_best(y,nd)) nd;
val select_best =
  <({('v', (14, @(0.955))), ('x', (21, @(0.999)))},
    (('w', (17, @(0.96))), ('m', (14, @(0.95)))))> : co
- orsmap cr select_best;
val it = <(66, @(0.95213691))> : co
```

Thus, we see that the design with the best reliability costs only \$66, even though the cost varies from \$56 to \$82, as we know from mapping `cr` over `nd`. So, as it often happens, one does not have to buy the most expensive thing to get the best quality.

Finally, to select the design with the best reliability that costs under n dollars, we write a function `bestunder` and find the most reliable design under \$62:

```
- 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
- bestunder 62;
val it =
  <({('v', (14, @(0.955))), ('x', (21, @(0.999)))},
    (('k', (11, @(0.93))), ('m', (14, @(0.95)))))> : co
- orsmap cr it;
val it = <(60, @(0.9507058425))> : co
```

Again, it is not necessary to get the most expensive design for 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.

5.2 Connecting OR-SML to a theorem prover

An example where OR-SML is currently being put to successful use is with the theorem prover HOL90 [9]. The precise nature of the theorem prover is not important to the discussion here of how OR-SML is used to enhance it functionality. Some of the aspects that are relevant are the following. HOL90 is an interactive theorem prover written in SML. It has a pre-existing notion of a "theory database" for permanently recording information about previously defined constants and previously proven theorems. It is an "open system" in the sense that it is SML with its environment enriched by a collection of datatypes, data structures, and procedures. (OR-SML is an "open system" in the same sense.) This allows us to incorporate the query language of OR-SML specialized to HOL90 theory databases without having to recompile the theorem prover. The main task in interfacing OR-SML to HOL90 is defining a type `base` that describes the different kinds of information that are stored

in the given database. In our case, most of this information is just the theorems that have been proved for the theory associated with the theory-database. (Other information includes the names and types of constants that have been introduced in the theory.) Were HOL90 a "closed" system with its own read-evaluate-print loop (or other user interface), the task of incorporating OR-SML queries into it would be somewhat more complicated. In addition to defining the appropriate `base` datatype (and accompanying functions), the implementor would need to decide how to expose the additional functionality provided by OR-SML to the end user, and would need to modify the user-interface accordingly. Because we are merging two "open" systems, we are able to add OR-SML to HOL90 as a library loadable at the users request.

The main use to which the OR-SML extension of HOL90 has been put so far is browsing the theories for theorems that might be relevant to the theorem proving task at hand. The power of the combination of OR-SML and HOL90 can be seen, however, with an example involving proof planning. For the particular example we describe here, a few more details of HOL90 will be necessary. The language of HOL90 is a weakly polymorphic version of the simply-typed lambda calculus [7]. That is, it has a notion of type, and types may be parametrized by other types. Users may define new types. A very important class of user-defined types are those of inductive, or recursive, datatypes, including nested mutually recursive datatypes. Part of the process of defining a recursive datatype involves proving an "initiality theorem" which states that a function over the datatype may be uniquely defined by cases over the constructors for the datatype. A type is a recursive datatype if and only if it has an initiality theorem stored in the theory database of the theory where the type was defined. Given a recursive datatype there may or may not be a principle of structural induction for that type already stored in the theory database. However, one may readily test if a theorem is the principle of induction for a type that corresponds to a given initiality theorem. Moreover, if the principle of structural induction is not present, it may be automatically derived from the initiality theorem. Other useful facts which may be present, or can be proved include a principle for reasoning by cases and the fact that all the constructors are distinct. Now, given a goal to be proved, one often wants to proceed by structural induction over any datatypes admitting induction over which the goal is universally quantified. Given a goal, we would like to know all relevant principles of induction and any other theorems of the kind just mentioned that have been stored in the HOL90 theory database. However, a given type may or may not be an inductive datatype. Moreover, a polymorphic datatype may have instances which are components of several recursive datatypes. To see this, consider the to datatype specifications:

$$\sigma\ \mathsf{list} \;=\; \mathsf{Nil} \mid \mathsf{Cons}\ \sigma\ (\sigma\ \mathsf{list}) \qquad \text{and} \qquad \sigma\ \mathsf{tree} \;=\; \mathsf{Leaf}\ \sigma \mid \mathsf{Node}\ (\sigma\ \mathsf{tree})\ \mathsf{list}$$

They provide us with the following two principles of induction:

$$\forall\, P.\, (P\ \mathsf{Nil} \wedge \forall\, h\ t.\, P\ t \Longrightarrow P\ (\mathsf{Cons}\ h\ t)) \Longrightarrow \forall\, l.P\ l \qquad \text{and}$$

$$\begin{aligned}\forall R\ P.\ (\ &(\forall\, n.\ R\ (\mathsf{Node}\ n)) \wedge (\forall\, l.\ P\ l \Longrightarrow R\ (\mathsf{Node}\ l)) \wedge \\ &P\ \mathsf{Nil} \wedge (\forall\, t\ l.\, (R\ t \wedge P\ l) \Longrightarrow P\ (\mathsf{Cons}\ t\ l))\ ) \Longrightarrow (\forall\, t.\ R\ t) \wedge (\forall\, l.\ P\ l)\end{aligned}$$

The first principle says that to prove that any property P holds for all lists, it suffices to show that it holds for the Nil list, and that if it holds for the tail of a list then it still holds when the head is put on. The second principle provides a similar reduction, but for proving properties over trees and tree lists jointly. If we are trying to prove a fact holds for all objects of type σ tree list, we could proceed by structural induction over lists, *or* we could proceed by mutual structural induction over both tree lists and trees. Our query for finding such information needs to be sensitive to the possibility of multiple choices, and thus to disjunctive information.

Assume we have the following:

`all_theories_db` : `co` is the OR-SML version of the theories database for HOL90 (which is essentially a set of entries of `base` type);

`universal_types` : `term -> hol_type list` is a function (procedure) which returns the list of the types of the leading universally quantified variables of a given term;

`is_initial_theorem_for` : `hol_type -> base -> bool` is a function which tests whether a given theorem is an initiality theorem for a given type;

`is_induction_theorem_for` : `base -> base -> bool`
`is_cases_theorem_for` : `base -> base -> bool` and
`is_distinct_constructors_theorem_for` : `base -> base -> bool` are functions which take an initiality theorem as the first argument and then test whether the second argument is the corresponding induction theorem for the first function, the theorem supporting reasoning by cases for the second function, and for the third, the theorem stating that all the constructors have distinct ranges.

The function `is_initial_theorem_for` is of a different type than the last three functions, because it is driven by the type which might be inductive, and definitely determines whether the type is or not. The other three functions allow us to gather other possible information once we have the results of `is_initial_theorem_for`. All of these testing functions may be converted into ones which work with OR-SML by composing them with `apply_test`. When we know we have a complex object (*i.e.* an OR-SML object) which is the equivalent of an object of `base` type, we may convert it into the corresponding `base` object using `co_to_base`. Using these functions together with some of the functions from OR-SML described previously, we may incrementally define the query for finding all possible sequences of relevant induction information as follows:

```
fun is_initial_co_for ty = apply_test (is_initial_theorem_for ty)

fun mk_initiality_options ty = set_to_or (Set.select (is_initial_co_for ty) all_theories_db)

fun gather_ind_and_cases co_initial_thm =
    let val initial_thm = DEST.co_to_base co_initial_thm
    in  mkprodco
        (co_initial_thm,
         mkprodco
         (Set.select (apply_test (is_induction_theorem_for initial_thm)) all_theories_db,
          union (Set.select
                 (apply_test (is_cases_theorem_for initial_thm)) all_theories_db,
                 Set.select
                 (apply_test (is_distinct_constructors_theorem_for initial_thm))
                             all_theories_db)))
    end;

fun mk_full_induction_options ty = orsmap gather_ind_and_cases (mk_initiality_options ty)

fun fold_induction_options [] = bang empty
```

```
  | fold_induction_options (hd_ty :: tl_tys) =
    let val new_options = mk_full_induction_options hd_ty
    in cond(eq(new_options, orempty),
            (fn rem_co => mkprodco(bang empty,rem_co)),
            (fn rem_co => mkprodco(new_options,rem_co)))
           (fold_induction_options tl_tys)
    end

fun goal_induction_options goal = normal (fold_induction_options (universal_types goal))
```

The only thing out of the ordinary with the above is definition of `fold_induction_options`. (Note that in its definition the `::` is for cons-ing an element onto the front of a list in SML, and should not be confused with its use in pretty-printing "true types".) As we are building the sequence of possibilities for induction, we take advantage of the structural level of OR-SML to replace the empty or-set by the empty tuple (`bang empty`) to represent that the type of the universally quantified variable did not admit induction. This allows us to switch to the conceptual level using normalization to acquire the collection of all possible sequences consisting of induction information when appropriate and a place holder of the empty tuple when induction is not appropriate.

In the above we have described a particular example of creating a query to find all possible principles of structural induction and related information relevant to a particular goal to be proved. Other examples exist which involve finding all possible sequences of equations and conditional equations for rewriting a goal towards a particular form. Our experience with using OR-SML in HOL90 is still limited. However, we have found its performance to be acceptable for the size of database with which we are dealing and the nature of query we are apt to put. For example, to run the query `goal_induction_option` on a goal with two universally quantified variables, each admitting induction, and on a database containing 759 entries took approximately 7 seconds on a Sparcstation 2. It is our belief that the ability to make queries involving conjunctive and disjunctive information using OR-SML within the theorem prover HOL90 will considerably enhance the end-users ability to gather information appropriate for planning the proof of goals.

5.3 Querying independent databases

The general problem of querying independent databases is the following: given a set of databases $D_1, \ldots, D_n$ and a query q that can not be answered by using information from one of D_i's, approximate the answer to q by using information from all $D_1, \ldots, D_n$. These problems have been investigated and they gave rise to a number of theoretical models [5, 11, 22, 17]. Intuitively, given a query q, the databases are divided into two groups, one giving the upper approximation to the answer to q (that corresponds to possible information) and the other giving the lower approximation (that corresponds to the definite information). It has been shown in [11, 17] that the approximation constructs enjoy nice theoretical properties that allow defining structural recursion over them. However, a large number of preconditions to be verified [17] makes programming by structural recursion rather inconvenient, and it was argued in [17] that, from the semantical point of view, the approximation constructs correspond to using both sets and or-sets in a certain way. We leave the general treatment of this problem, as well as the formal definition of structural recursion on approximations in OR-SML for a future paper, and here demonstrate an OR-SML solution to one of the most typical examples of querying independent databases.

Many papers dealing with the problem of querying independent databases make certain assumptions about the existence of a key (sometimes implicitly). Such key assumptions can be excessively restrictive. Below we give an example and use it to explain some of the problems we have to deal with when key assumptions are made. We also explain that dropping key assumptions inevitably leads to using or-sets. We then proceed to demonstrate how some of the mentioned problems can be solved in OR-SML.

Consider the following problem. Suppose the university database has two relations, Employees and CS1 (for teaching the course CS1):

Employees

Name	Salary
John	10K
John	15K
Mary	12K
Sally	17K

CS1

Name	Room
John	076
Jim	320
Sally	120

Assume that our query asks to compute the set TA of teaching assistants. We further assume that only TAs can teach CS1 and every TA is a university employee.

Now let us look at the problems we are to solve in order to answer the TA query. First, the databases are inconsistent. Jim teaches CS1 and hence he is a TA and an employee, but there is no record for Jim in the Employees relation. To get rid of this anomaly, we must decide if we believe CS1 or Employees. If the former is the case, then the problem is solved by adding Jim to Employees (with a null salary until one is acquired). In the case where we believe the Employee relation, the problem is solved by removing Jim from the CS1 relation. When there are no inconsistencies in the relations, we have to find an approximation of the set of TAs, that is, we have to 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 $\perp$ (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 meet of records. For example,

$$\langle \text{John}, 15\text{K}, \perp \rangle \vee \langle \text{John}, \perp, 076 \rangle = \langle \text{John}, 15\text{K}, 076 \rangle$$

$$\langle \text{John}, 15\text{K}, \perp \rangle \wedge \langle \text{John}, \perp, 076 \rangle = \langle \text{John}, \perp, \perp \rangle$$

Notice that the join of two records is not necessarily defined. The theory of partial information conveyed by means of partial orders was worked out in [5, 6, 14, 16]. For instance, for ordering collections the following two extensions of partial orders have been considered:

$$X \leq^{\flat} Y \Leftrightarrow \forall x \in X\ \exists y \in Y : x \leq y \qquad \text{and} \qquad X \leq^{\sharp} Y \Leftrightarrow \forall y \in Y\ \exists x \in X : x \leq y$$

In [18] it was argued that $\leq^{\flat}$ is better suited for ordering sets while $\leq^{\sharp}$ is better suited for ordering or-sets. Using the above notation and our assumptions, we can now say that the best information about TAs that can be obtained from the given relations is CS1 $\leq^{\flat}$ TA $^{\sharp}\geq$ Employees.

Now we show how a query "approximate the set of TAs" can be done in OR-SML. First, OR-SML has a library of domain theoretic orderings, which are $\leq^{\flat}$ for sets and $\leq^{\sharp}$ for or-sets (function `leqdom`) and

corresponding functions `meet, join` : $s \times s \to \langle s \rangle$ (the empty or-set is used to indicate a non-existent join or meet; otherwise a singleton or-set is produced). Using these functions, it is easy to write a test whether two records have a join.

Since we treat Employees as a relation of possible upper bounds for TAs, we make it an or-set. All elements of CS1 are TAs, so CS1 is a set. We now represent the data as follows:

```
- val emp = make();
<('John', ({@(10.00)}, {})), ('John', ({@(15.00)}, {})),
 ('Mary', ({@(12.00)}, {})), ('Sally', ({@(17.00)}, {}))>!
- val cs1 = create "{('John', ({}, {76})), ('Jim', ({}, {320})), ('Sally', ({}, {120}))}";
```

The first problem we face is getting rid of inconsistencies in the database. In our particular example, Jim is in CS1 but not in the Employees. Assuming we believe the Employees relation, we remove this anomaly as follows (`compatible` is a function that tests whether the join of two elements is defined):

```
- fun compatible (x,y) = neg(eq(join(x,y),orempty));
- 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 [5]. 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, the promotion of x adds all information about x from Employees.) The solution proposed by them is to take all promotions of elements in CS1 as "sure TAs" and elements of Employees not consistent with those promotions as "possible TAs". However, this solution is contingent upon the condition that the Name field is a key. With this condition, we can easily program the solution of [5] using a function `promote` and a new relation `emp1`.

```
- 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 emp1 = make();
<('John', ({@(10.00)}, {})), ('Mary',({@(12.00)}, {})), ('Sally', ({@(17.00)}, {}))>!

- val promoted_cs1 = promote compatible (emp1,new_cs1);
val promoted_cs1 = <{('John', ({@(10.0)}, {76})), ('Sally', ({@(17.0)}, {120}))}> : co

- val res = divide_all compatible (emp1,promoted_cs1);
val res = <(<('Mary', ({@(12.0)}, {}))>,
    {('John', ({@(10.0)}, {76})), ('Sally', ({@(17.0)}, {120}))})> : co
```

Here `big_meet` calculates the meet of a family and `divide_all` separates sure TAs from possible TAs.

Now it is possible to separate sure TAs from possible TAs:

```
fun divide compat (R,S) = let
    fun compat_to_set (X,x) = member(mkboolco(true),
                                     (smap (fn z => compat(z,x)) X))
 in (orselect (fn z => neg(compat_to_set (S,z))) R, S) end;

fun divide_all compat (R,S) = orsmap (fn z => mkprodco(
                                        divide compat (p1(z),p2(z))))
                                     (orpairwith(R,S));

- val res = divide_all compatible (R2,S2);
val res = <(<('Mary', ({12}, {}))>, {('John', ({10}, {76}))})> : co
```

Therefore, John from office 76 and with salary 10K is *definitely* a TA and Mary with salary 12K and not known office *may be* a TA.

However, if the name field is not a key, this solution will not work. For example, both Johns from Employees 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. In particular, we take all possible joins $x \vee y_1, \ldots, x \vee y_n$ and make them into an or-set, which now plays the role of the promotion of x. Then, taking the or-set brackets outside, we obtain the or-set with all possible answers to the TA query.

```
fun solution compat (R,S) = let fun get_R_a a = orselect (fn z => compat(z,a)) R
  in orpairwith(R, alpha(smap get_R_a S)) end;
- val result = solution compatible (R1,S1);
val result =
  <(<('John', ({10}, {})), ('Mary', ({12}, {})), ('John', ({15}, {}))>,
    {('John', ({10}, {}))}),
   (<('John', ({10}, {})), ('Mary', ({12}, {})), ('John', ({15}, {}))>,
    {('John', ({15}, {}))})> : co
```

We now see that there are two possible answers to the TA query: both say that Mary could be a TA, and one says that John making 10K is a TA and the other says that John making 15K is a TA.

Summing up, we have seen that one of the canonical problems of querying independent databases can be solved by OR-SML. Moreover, using or-sets gives us a correct answer even if the key constraints do not hold, something that the solution of [5] falls short of doing.

6 Conclusion and further research

We have described a functional database language built on top of Standard ML. The set part of the core language is precisely the nested relational algebra. It is then extended with or-sets which are used to deal with disjunctive information. Normalization of objects, when added as a primitive, allows querying databases with certain kinds of incomplete information (for example, the databases of incomplete designs). Or-sets are also useful in querying independent databases, for which we have shown how to extend known methods of querying which usually rely on certain assumptions about keys. The language has adequate power to handle multisets and aggregate functions. It is extensible

with new base types, and can be interfaced to other systems written in Standard ML. Moreover, representing objects as a single SML type allows the user to write queries using higher-order functions which are typically not present in query languages.

There are two kinds of problems we need to address in the future. First, the language could be extended with variant types, true records and perhaps recursive values. The second and more important kind of problems deals with speeding up the evaluation of conceptual queries. Presently, normal forms are created before any conceptual queries can be asked. The process of normalization can be rather costly. In fact, tight upper bounds have been found by one of the authors in [18], and they show that current approach is not applicable to large databases.

We see three ways that could make evaluation of conceptual queries faster. First, for many queries there is no need to normalize all the way up to the normal form in order to answer a query. However, then we need an analog of the coherence theorem for partial normalization. Recently, some progress has been made in identifying classes of types for which a partial coherence result would hold. Second, many queries on normal forms ask about existence of objects with certain parameters (like a cheap reliable design.) We are currently working on designing a lazy evaluation strategy that would produce objects in the normal form one by one, thus possibly speeding up answering existential queries. Finally, on very large objects one may want to settle for approximate solutions. Finding such solutions (e.g. a design whose reliability is at least 95% of the optimal) is an intriguing problem we would like to look at.

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

References

[1] S. Abiteboul and S. Grumbach, COL: a logic-based language for complex objects, In *Advances in Database Programming Languages*, ACM Press, 1990, pages 271–293.

[2] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. of 3rd Int. Workshop on Database Programming Languages*, pages 9–19, Naphlion, Greece, August 1991.

[3] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proc. ICDT, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 92.

[4] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *LNCS 510: Proc. of ICALP-1991*, Springer Verlag, 1991, pages 60–75.

[5] P. Buneman, S. Davidson, A. Watters, A semantics for complex objects and approximate answers, *Journal of Computer and System Sciences* 43(1991), 170–218.

[6] P. Buneman, A. Jung, A. Ohori, Using powerdomains to generalize relational databases, *Theoretical Computer Science* 91(1991), 23–55.

[7] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[8] L. Colby, A recursive algebra for nested relations, *Inform. Systems* 15 (1990), 567–582.

[9] M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, Cambridge, Great Britain, 1993.

[10] S. Grumbach, T. Milo, Towards tractable algebras for bags, *Proceedings of the 12th Conference on Principles of Database Systems*, Washington DC, 1993, pages 49–58.

[11] C. Gunter, The mixed powerdomain, *Theoretical Computer Science* 103 (1992), 311–334.

[12] E. Gunter and L. Libkin, A functional database programming language with support for disjunctive information, AT&T Technical Memo No. BL0112610-931203-47, 1993.

[13] G. G. Hillebrand, P. C. Kanellakis, and H. G. Mairson. Database query languages embedded in the typed lambda calculus. In *Proc. of LICS-93*, pages 332–343.

[14] T. Imielinski, W. Lipski. Incomplete information in relational databases. *JACM* 31(1984), 761–791.

[15] T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *Proc. of ACM-SIGMOD, Denver, Colorado, May 1991.*

[16] L. Libkin, A relational algebra for complex objects based on partial information, In *LNCS 495: Proceedings of Symp. on Math. Fund. of Database Systems–91*, Springer-Verlag, 1991, pages 36–41.

[17] L. Libkin, Approximation in Databases, Technical Report MS-CIS-94-21/L&C 79, University of Pennsylvania, 1994.

[18] L. Libkin and L. Wong, Semantic representations and query languages for or-sets, *Proceedings of the 12th Conference on Principles of Database Systems*, Washington DC, 1993, pages 37–48.

[19] L. Libkin and L. Wong, Some properties of query languages for bags, In *Proceedings of the 4th International Workshop on Database Programming Languages, September 1993*, Springer Verlag, 1994, pages 97–114.

[20] L. Libkin and L. Wong, New techniques for studying set languages, bag languages and aggregate functions, In *Proceedings of the 13th Conference on Principles of Database Systems*, Minneapolis, 1994, pages 155–166.

[21] R. Milner, M. Tofte, R. Harper, *"The Definition of Standard ML"*, The MIT Press, Cambridge, Mass, 1990.

[22] T.-H. Ngair. Convex Spaces as an Order-theoretic Basis for Problem Solving, Technical Report MS-CIS-92-60, University of Pennsylvania, 1992.

[23] A. Ohori, V. Breazu-Tannen and P. Buneman, Database programming in Machiavelli: a polymorphic language with static type inference, In *SIGMOD 89*, pages 46–57.

[24] B. Rounds, Situation-theoretic aspects of databases, In *Proc. Conf. on Situation Theory and Applications*, CSLI vol. 26, 1991, pages 229–256.

[25] H.-J. Schek and M. Scholl, The relational model with relation-valued attributes, *Inform. Systems* 11 (1986), 137–147.

[26] S.J. Thomas and P. Fischer, Nested relational structures, in P. Kanellakis editor, *Advances in Computing Research: The Theory of Databases*, pages 269–307, JAI Press, 1986.

[27] P.W. Trinder, Comprehension: A query notation for DBPLs, In *Proceedings of the 3rd International Workshop on Database Programming Languages*, August 1991, pages 49–62, Morgan Kaufmann.

[28] 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, University of Glasgow.

[29] P. Wadler, Comprehending monads, In *Proceedings of ACM Conference on Lisp and Functional Programming*, Nice, June 1990.