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

Elsa Gunter[1] and **Leonid Libkin**[⋆2]

[1] AT&T Bell Laboratories, Rm.#2A-432
600 Mountain Ave., Murray Hill, NJ 07974, USA
email: elsa@research.att.com

[2] Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104, USA
email: 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 [20]. 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.

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 [13]. 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.

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

⋆ Part of this work was done when this author was visiting AT&T Bell Laboratories. Partial support was provided by NSF Grant IRI-90-04137 and AT&T Doctoral Fellowship.

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 [13, 17, 23]. 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.) 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. 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, 18, 22, 26, 27].) Moreover, there are theoretical foundations for studying such languages [4, 11]. 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, cf. [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 contains the nested relational algebra as a sublanguage. The standard presentations of the nested relational algebra [7, 24, 25] 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 [27, 28] and it was later generalized to other collection types [17, 18]. The language obtained from the nested relational algebra by adding appropriate primitives dealing with or-sets was called *or-*$\mathcal{NRA}$ in [17].

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 [17] 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. 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 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 [18]. We show how bags and certain aggregate functions can be encoded. OR-SML allows programming with structural recursion on sets and or-sets. It 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 [20]. 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 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. 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 thus easy to change without touching the rest of the system. In particular, implementation of *any* internal OR-SML feature can be changed without altering the end-user interface.

As of now, the system is suitable for querying small and medium size databases, 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. Due to the modularity of the implementation, any such changes can 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. In Section 4 we describe additional features of the language: 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 [17]. 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:

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

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.

General operators

$$\frac{g : u \to s \quad f : s \to t}{\texttt{comp}(f, g) : u \to t} \qquad \frac{c : bool \quad f : s \to t \quad g : s \to t}{\texttt{cond}(c, f, g) : s \to t} \qquad \frac{f : u \to s \quad g : u \to t}{\texttt{pair}(f, g) : u \to s \times t}$$

$$\frac{}{\texttt{p1} : s \times t \to s} \qquad \frac{}{\texttt{p2} : s \times t \to t} \qquad \frac{}{\texttt{bang} : t \to unit} \qquad \frac{}{\texttt{eq} : t \times t \to bool} \qquad \frac{}{\texttt{id} : t \to t}$$

Operators on sets

$$\frac{}{\texttt{emptyset} : unit \to \{t\}} \qquad \frac{}{\texttt{sng} : t \to \{t\}} \qquad \frac{}{\texttt{union} : \{t\} \times \{t\} \to \{t\}}$$

$$\frac{f : s \to t}{\texttt{smap} f : \{s\} \to \{t\}} \qquad \frac{}{\texttt{pairwith} : s \times \{t\} \to \{s \times t\}} \qquad \frac{}{\texttt{flat} : \{\{t\}\} \to \{t\}}$$

Operators on or-sets

$$\frac{}{\texttt{emptyorset} : unit \to \langle t \rangle} \qquad \frac{}{\texttt{orsng} : t \to \langle t \rangle} \qquad \frac{}{\texttt{orunion} : \langle t \rangle \times \langle t \rangle \to \langle t \rangle}$$

$$\frac{f : s \to t}{\texttt{orsmap}\ f : \langle s \rangle \to \langle t \rangle} \qquad \frac{}{\texttt{orpairwith} : s \times \langle t \rangle \to \langle s \times t \rangle} \qquad \frac{}{\texttt{orflat} : \langle\langle t \rangle\rangle \to \langle t \rangle}$$

Interaction of sets and or-sets

$$\frac{}{\texttt{alpha} : \{\langle t \rangle\} \to \langle\{t\}\rangle}$$

Fig. 1. *or-$\mathcal{NRA}$* Type Inference of OR-SML Terms

Let 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. `emptyset`() is the empty set. This constant also has name `empty`. `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`$(f)\{x_1, \dots, x_n\} = \{f(x_1), \dots, f(x_n)\}$. `pairwith` pairs the first component of its argument with every item in the second component: `pairwith`$(y, \{x_1, \dots, x_n\}) = \{(y, x_1), \dots, (y, x_n)\}$. Finally, `flat` is flattenning: `flat`$\{X_1, \dots, X_n\} = X_1 \cup \dots \cup X_n$. The semantics of the or-set constructs is similar.

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

In the sequel 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 the 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 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` (interactive creation). The function `make` is terminated by typing a special "end-of-input" marker, which in this case is "`.`". For example:

```
- 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
```

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.

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,b)` even though the true types of `a` and `b` are $\{\langle \mathit{int} \rangle\}$ and $\mathit{int} \times \mathit{string}$. 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 connectives, membership and subset tests, difference, selection, cartesian product and their counterparts for or-sets, see [3, 17]. 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
- 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 `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 of 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. 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
```

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 [16].

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 [17]. Define the following rewrite system on types:

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

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 $s_1 \twoheadrightarrow s_n$. 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 [17]. Moreover, the following result was proved in [17]:

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` can be added to OR-SML to give it adequate power to work with conceptual representations of objects:

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

The semantics of `normal` at type t is $\mathsf{app}(r)$ where $r : t \twoheadrightarrow nf(t)$. 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
```

4 Additional features of the system

Arithmetic functions. Integers are a base type in OR-SML with the following supported operations: `plus`, `mult` and `monus` of true type $int \times int \to int$. `monus` is the modified subtraction: $\texttt{monus}(m, n) = \max\{0, m - n\}$. The function `gen` of true type $int \to \{int\}$ is given by $\texttt{gen}(n) = \{0, \ldots, n\}$. The summation construct `sum` takes 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.

These operations are precisely what must be added to a set language to endow it with the power of languages for nested *bags* as in [9, 18]. Bag semantics is most often used for correct evaluation of aggregate functions like "total of column" etc. This now can 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 operations from [18] can be modeled easily in OR-SML. For example, the difference of two bags is 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. 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`, there is a way to have a function `addone_co : co -> co` whose semantics is $\lambda x.x + 1.0$. The function `apply` takes a function `f : base list -> base` and returns a function from `co` to `co` representing the action of `f` on complex objects. 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 for applying binary and unary functions on base types by using functions `apply_unary` and `apply_binary`.

There is also a mechanism to translate predicates on base types into predicates on complex objects to be used with `cond` and `select`. The function `apply_test` translates predicates of type `(base -> bool)` into predicates of type `co -> co`. For example,

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

Structural recursion. Structural recursion on sets [2] is a very powerful programming tool for query languages. However, it is often unsafe: a function defined by structural recursion is not guaranteed to be well-defined, and well-definedness can not be checked by a compiler [4]. But it is often helpful in writing programs or changing types of big databases (rather than reinputting them), so we have decided to include it in OR-SML. Structural recursion on sets and or-sets is available by means of two constructs `SR.sr` and `SR.orsr` that take an object e of type t and a function f of type $s \times t \rightarrow t$ and return a function `SR.sr`(e, f) of type $\{s\} \rightarrow t$ or a function `SR.orsr`(e, f) of type $\langle s \rangle \rightarrow t$ 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:

```
- 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 I/O module must be used. However, binary I/O is an unsafe feature of SML.

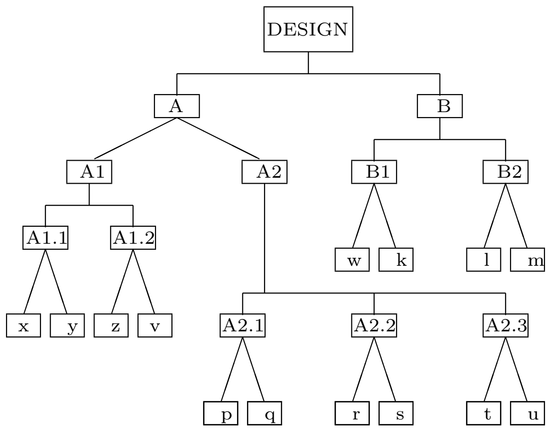
Deconstruction of complex objects. It may be the case that after evaluating a query, the user may need to write a program to deal with the result. Since all operations of OR-SML work with type `co`, there is a way out of complex objects to the usual SML types. The structure `DEST` contains some functions to deconstruct complex objects and obtain SML values. See the system manual [16] 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 below. 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 desing 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 each smallest subpart (a leaf of the tree) has two parameters: its cost (of type *int*) and its reliability (of type *real*). Then the tree above can be translated into an OR-SML object `design` 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 questions. First, is it possible to complete the design using $60? Second, which design has the best reliability? And third, what is the most reliable design that costs under $62?

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 whole design. To write the cost function, simply add up all occurrences of integers in a normalized design. Writing the realibility function, the type of connection – series of parallel – must be taken into account. We do not show these functions here since they can be written straighforwardly in OR-SML. In our example we assume parallel connection of $B1$ and $B2$ and series connection of A and B.

To answer the first query, we write

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

Thus, we have 48 completed designs. The price range is from $56 to $82, and hence the design can be completed with $60.

To find the one that has the best reliability, we write the query `is_best` that selects the design with the best reliability from a given collection, and then apply it to `nd`.

```
- 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 (fn x => mkprodco ((cost x), (reliability x))) 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. 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 first select those designs from `nd` and then find the best one of those using the above query. In our example, the most reliable design under $62 costs $60, so 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 being used successfully is with the theorem prover HOL90 [8]. The precise nature of the theorem prover is not important to the discussion of how OR-SML is used to enhance it functionality, but the following are some aspects. HOL90 is an interactive theorem prover written in SML. It has a pre-existing notion of a "theory database" of previously defined constants and previously proven theorems. It is an "open system" in that it is SML with an environment enriched by a collection of datatypes, data structures, and procedures, the same way OR-SML is. 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, such as definitions and theorems, that are stored in the given database. 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.

The power of the combination of OR-SML and HOL90 can be seen with an example of proof planning. For the particular example we describe here, a few more details of HOL90 are necessary. The language of HOL90 is has a notion of type, and types may be parametrized by other types. Users may define new types. An important class of user-defined types is those of inductive datatypes, including nested mutually inductive datatypes. These types are characterized by having stored in the a theory an "initiality theorem" stating that a function over the datatype may be uniquely defined by cases over the constructors for the datatype. An inductive datatype may or may not have a principle of structural induction already stored in the theory database. However, given the initiality theorem, one may test if it is present, and if it is not it may be derived.

A common method of solving a goal is by structural induction. Therefore, given a goal, we would like to know all relevant principles of induction for all types over which the goal is universally quantified. However, a given type may or may not be an inductive datatype. Moreover, a polymorphic datatype may have instances which are components of several inductive datatypes. An example of this is the following:

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

These datatypes provide us with the following two principles of induction:

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

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

The first principle says that a property P holds for all lists provided it holds for the Nil list, and, if it holds for the tail of a list, then it still holds when the head is put on. The second principle is similar, but for properties over trees and tree lists jointly. To prove a fact holds for all objects of type σ tree list, we could use structural induction over lists, *or* mutual structural induction over both tree lists and trees. Our query for finding such information must allow for multiple choices, and hence disjunctive information.

Assume we have the following:

- `all_theories_db : co` is the OR-SML version of the theories database for HOL90
- `universal_types : term -> hol_type list` is a function returning the list of the types of the leading universally quantified variables of a given term;
- `is_initial_theorem_for : hol_type -> base -> bool` tests whether a given theorem is an initiality theorem for a given type;

- `is_induction_theorem_for : base -> base -> bool` tests if the second argument is the induction theorem of an initiality theorem given as the first argument.

To apply the testing functions to complex objects of OR-SML, we compose them with `apply_test`. A complex object which has true type `base`, may be converted 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 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_induction init_thm_co =
    let val initial_thm = DEST.co_to_base init_thm_co
        val induct_co =
            Set.select (apply_test (is_induction_for initial_thm))
                       all_theories_db
    in  mkprodco (init_thms_co, info)  end;
fun mk_full_induction_options ty =
    orsmap gather_induction (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 the definition of `fold_induction_options`. (The `::` cons-es an element onto the front of a list in SML.) 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 did not admit induction. This allows us to switch to the conceptual level using normalization to acquire of all possible sequences of induction information, using the empty tuple when induction is not appropriate.

Our experience with 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, runnig the query `goal_induction_options` on a goal with two universally quantified variables, each admitting induction, on a database containing 759 entries took approximately 7 seconds on a Sparcstation 2, which we believe to be tolerable in an interactive environment.

5.3 Querying independent databases

The general problem of querying independent databases is the following: given a set of databases $D_1, \dots, D_n$ 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 $D_1, \dots, D_n$. These problems have been investigated and they gave rise to a number of theoretical models [5, 10, 21, 15]. 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).

Here we demonstarte how OR-SML can be used to solve a simple problem of querying independent databases. Consider the following problem. Suppose the university database has two relations, Employees and CS1 (for teaching the course CS1):

Employees

Name	Salary
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. For the purpose of this paper, we make an assumption that the Name field is a key. For more complicated solutions that do not use this assumption, see the full paper [16].

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 Employees 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 $\bot$ (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, $\boxed{\text{John}|\text{15K}|\bot} \vee \boxed{\text{John}|\bot|\text{076}} = \boxed{\text{John}|\text{15K}|\text{076}}$ and $\boxed{\text{John}|\text{15K}|\bot} \wedge \boxed{\text{John}|\bot|\text{076}} = \boxed{\text{John}|\bot|\bot}$. 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, 12, 14]. For instance, for ordering collections the following two extensions of partial orders have been considered: $X \leq^{\flat} Y \Leftrightarrow \forall x \in X\ \exists y \in Y:\ x \leq y$ and $X \leq^{\sharp} Y \Leftrightarrow \forall y \in Y\ \exists x \in X:\ x \leq y$. In [17] 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 $\text{CS1} \ \leq^{\flat}\ \text{TA}\ {}^{\sharp}\!\geq\ \text{Employees}$.

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 $\texttt{meet}, \texttt{join} : s \times s \to \langle s \rangle$ (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 (called `compatible`) 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 below, and remove anomaly (Jim) using the test for a jon `compat` as a parameter:

```
- val emp = make();
```

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

- 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 $x \in \mathrm{CS1}$, 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".

```
- 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
- val res = divide_all compatible (emp1,promoted_cs1);
val res = <(<('Mary', ({@(12.0)}, {}))>,
    {('John', ({@(15.0)}, {76})), ('Sally', ({@(17.0)}, {120}))})> : co
```

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

If the name field is not a key, this solution will not work: 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. See [16].

6 Conclusion

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.

In the future, we plan to extend the language with variant types and true records. We also plan to show how it can be used for more complicated queries to which only approximate answers can be found.

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, pp. 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-92*, pages 140–154. Springer, 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, *JCSS* 43:170–218, 1991.
6. P. Buneman, A. Jung, A. Ohori, Using powerdomains to generalize relational databases, *Theoret. Comp. Sci.* 91:23–55, 1991.
7. L. Colby, A recursive algebra for nested relations, *Inf. Syst.* 15:567–582, 1990.
8. M. J. C. Gordon and T. F. Melham. *Introduction to HOL.* Cambridge University Press, Cambridge, Great Britain, 1993.
9. S. Grumbach, T. Milo, Towards tractable algebras for bags, *Proceedings of the 12th PODS*, Washington DC, 1993, pages 49–58.
10. C. Gunter, The mixed powerdomain, *Theoret. Comp. Sci.* 103:311–334, 1992.
11. 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.
12. T. Imielinski, W. Lipski. Incomplete information in relational databases. *JACM* 31:761–791, 1984.
13. 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.*
14. 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.
15. L. Libkin, Algebraic characterization of edible powerdomains, Technical Report MS-CIS-93-70/L&C 71, University of Pennsylvania, 1993.
16. L. Libkin and E. Gunter, A functional database programming language with support for disjunctive information, AT&T Technical Memo, 1993.
17. L. Libkin and L. Wong, Semantic representations and query languages for or-sets, *Proceedings of the 12th PODS*, Washington DC, 1993, pages 37–48.
18. 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.

19. L. Libkin and L. Wong, Aggregate functions, conservative extension and linear orders, In *Proceedings of the 4th International Workshop on Database Programming Languages, September 1993*, Springer Verlag, 1994, pages 282–294.
20. R. Milner, M. Tofte, R. Harper, *"The Definition of Standard ML"*, The MIT Press, Cambridge, Mass, 1990.
21. T.-H. Ngair. Convex Spaces as an Order-theoretic Basis for Problem Solving, Technical Report MS-CIS-92-60, University of Pennsylvania, 1992.
22. 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.
23. B. Rounds, Situation-theoretic aspects of databases, In *Proc. Conf. on Situation Theory and Applications*, CSLI vol. 26, 1991, pages 229–256.
24. H.-J. Schek and M. Scholl, The relational model with relation-valued attributes, *Inform. Systems* 11 (1986), 137–147.
25. S.J. Thomas and P. Fischer, Nested relational structures, in P. Kanellakis editor, *Advances in Computing Research: The Theory of Databases*, pp. 269–307, 1986.
26. P.W. Trinder, Comprehension: A query notation for DBPLs, In *Proc. of the 3rd DBPL*, August 1991, pages 49–62, Morgan Kaufmann.
27. 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.
28. P. Wadler, Comprehending monads, In *Proceedings of ACM Conference on Lisp and Functional Programming*, Nice, June 1990.

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