Interfacing HOL90 with a Functional Database Query Language

Elsa L. Gunter[1] and Leonid Libkin[2]

[1] AT&T Bell Laboratories, Rm.#2A-432
600 Mountain Ave., Murray Hill, N.J. 07974, USA
phone: 1 908 582 5613 email: elsa@research.att.com
[2] AT&T Bell Laboratories, Rm.#2A-422
phone: 1 908 582 7647 email:libkin@research.att.com

Abstract. We describe a functional database language OR-SML for handling disjunctive information in database queries, its implementation in Standard ML [10], and its interface to HOL90. 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 an example of queries over the "database" of HOL90 theories which require disjunctive information and show how to use the language to answer these queries. Since the system is running 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. The language has been implemented as an HOL90-loadable library of modules in Standard ML.

1 Introduction

In this paper we describe a functional language, which we call OR-SML, for querying databases with incomplete and disjunctive information, and its application to querying HOL90 theories. Our language is based on the functional paradigm. Design of functional database query languages has been studied extensively in the past few years and proved very useful. (See, for example [1, 2, 11, 14, 9].) Functional query languages have clear syntax, they can be typechecked, their semantics is generally easy to define and they allow a limited form of polymorphism. We believe that such a powerful and general query language will greatly facilitate the user in interactively finding useful theorems in HOL90 theories, but will also allow for the development of tactics and other tools of proof development which make full use of stored ancestor theories.

The language we describe in this paper contains the nested relational algebra as a sublanguage. The nested relational algebra is a standard query language for database objects that freely combine values of base types, records and sets. Its standard presentations [4, 12, 13] have cumbersome syntax, so we have decided to follow the approach of [2] which gives a clean and simple language that has precisely the same expressive power.

In order to represent disjunctive information in our query language, we added a new type constructor for *or-sets* to the nested relational algebra. One of the problems addressed in the language is the difference between *structural queries* and *conceptual queries*. At the structural level, an *or-set* is a collection of objects, just as a set is. However, at the conceptual level, an or-set represents one element from the or-set, while the set continues to represent the whole collection. It was shown in [8, 7] that conceptually equivalent objects can be reduced in a canonical manner to the same object, called its *normal form*. The normal form is a disjunct of all usual objects (*i.e.* not involving disjunctive information) represented by the given object prior to normalization. Therefore, one can take the conceptual meaning of any object to be its normal form. 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. In section 5 we give an example where it is desirable to be able to make queries at both the structural level and at the conceptual level. We make use of queries at the structural level to distinguish between recursive datatypes and other kinds of types while we make use of the conceptual level to determine what are the possibilities for induction principles for any recursive datatype.

The system OR-SML includes a subsystem which is equivalent to the nested relational algebra, but the whole system contains much more. First, normalization is present as a primitive. OR-SML also allows programming with structural recursion on sets and or-sets. It provides a mechanism for converting any user-defined functions on base types (integers, strings, HOL90 types, terms, and theorems) into functions that fit into the type system of OR-SML. It also gives a "way out" of database objects into SML values. This is useful, for example, if you wish to incorporate a query into a tactic or a derived rule of inference.

2 The core language

The theoretical language upon which OR-SML is based was developed by Libkin and Wong in [8]. 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.

Types of database objects (also called *complex objects* in the database literature) 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 as interfaced to HOL90 consists of `int`, `string` and a datatype composed of `hol_type`, `term`, and `thm`); *unit* is a special type whose domain has a unique element denoted in OR-SML by (); *bool* is the type of booleans; $t \times s$ is the product type whose objects are pairs of objects of types t and s. The set type $\{t\}$ denotes finite sets of elements of t and the or-set type $\langle t \rangle$ denotes finite or-sets of elements of t. The core language of *or-$\mathcal{NRA}$* consists of a family of operators, implemented as SML functions which provides the basic interface to OR-SML. Their specific types as *or-$\mathcal{NRA}$* operators are given by the rules in Fig. 1. All occurrences of s, t and u in the table are object types.

General operators

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

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

$$\frac{}{\texttt{eq} : t \times t \to bool} \qquad \frac{}{\texttt{id} : t \to t}$$

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. $\texttt{pair}(f, g)$ is pair formation: $\texttt{pair}(f, g)(x) =$

$(f(x), g(x))$. First and second projections are called `p1` and `p2`. `id` is the identity function. `bang` always returns the unique element of type *unit*, which has the name `unit_co`. `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 value also has the 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`$(y, \{x_1, \ldots, x_n\}) = \{(y, x_1), \ldots, (y, x_n)\}$. Finally, `flat` is flattening: `flat`$\{X_1, \ldots, X_n\} = X_1 \cup \ldots \cup X_n$. The semantics of the or-set constructs is similar.

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

These constructs are represented in SML as follows. Every complex object has SML type `co`. We shall refer to the type of an object or a function in *or-$\mathcal{NRA}$* as its *true type*. True types of complex objects can be inferred using the function `typeof`. (This is much the same as the situation with the types and terms of HOL90.) They are SML values having type `co_type`. When OR-SML prints a complex object 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`.

Besides integers, strings, and booleans, we may create complex objects from one other type, namely:

```
datatype hol_theory_data =
   Type of hol_type
 | Term of term
 | Thm of thm
 | Parent of {thy_name : string, parent : string}
 | TypeOp of {thy_name : string, tyop :{Name : string, Arity : int}}
 | Constant_tm of {thy_name : string, constant : term}
 | Infix_tm of {thy_name : string, constant : term}
 | Binder_tm of {thy_name : string, constant : term}
 | Axiom of {thy_name : string, theorem : (string * thm)}
 | Definition of {thy_name : string, theorem : (string * thm)}
 | StoredThm of {thy_name : string, theorem :(string * thm)}
```

The constructors `Type`, `Term`, and `Thm` are for injecting arbitrary HOL90 types, terms and theorems into the base type. The remaining constructors are intended to allow us to represent the components of HOL90 theories. The labels `parent`, `constant`, and `theorem` in the arguments to `StoredThm`, *etc.*, should not be confused with SML functions of the same name. To create complex objects from these types, the following basic functions are available:

```
val mkintco : int -> co
val mkboolco : bool -> co
```

```
val mkstringco : string -> co
val mkbaseco : hol_theory_data -> co
```

There are also some derived functions such as

```
val mksetint : int list -> co
val mkorsint : int list -> co
val mkprodco : co * co -> co
val mksetco : co list -> co
val mkorsco : co list -> co
val mk_theory_db : string -> co
val mk_all_theories_db : unit -> co
```

The string argument to `mk_theory_db` is the name of the HOL90 theory. It returns the set of entries from the theory. The function `mk_all_theories_db` creates a complex object that is the set of all the entires in all the theories available in the running HOL90 system. The reason for this being a function is that this is an extensible collection.

An example of the creation of some complex objects is as follows:

```
- val a = mksetint [1,3,5];
val a = {1, 3, 5} :: {int} : co
- val b = mkbaseco (StoredThm{thy_name = "prim_rec",
                              theorem = ("LESS_0",
                                         theorem "prim_rec" "LESS_0")});
val b =
  (StoredThm{theorem = ("LESS_0", |- !n. 0 < SUC n),
    thy_name = "prim_rec"}) :: hol_theory_data : co
- val c = itlist
          (fn x => (fn y => orunion(orsng(mkbaseco(Thm x)),y)))
          [EQ_SYM_EQ, TRUTH] orempty;
val c = <(Thm (|- T)), (Thm (|- !x y. (x = y) = y = x))
        > :: <hol_theory_data> : co
- val d = mkorsco (map (mkbaseco o Thm) [TRUTH, EQ_SYM_EQ]);
val d = <(Thm (|- !x y. (x = y) = y = x)), (Thm (|- T))
        > :: <hol_theory_data> : co
- eq(c,d);
val it = T :: bool : co
```

Output in the example above, as in the other examples in this paper was produced with a pretty-printer for the type `co` using the pretty-printer installation facility of the SML-NJ compiler. The pretty-printer for the type `co` is constructed from a pretty-printer for the type `hol_theory_data`, which has also been installed in SML-NJ.

In the second part of the example above, we give a sample derivation of `mkorsco` from primitives. Notice that `eq` returns `T` for `c` and `d` even though they don't print out in the same order. This is, of course, because we are dealing with or-sets, and order is disregarded.

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 [2, 8]. These functions are included in OR-SML in the form of a structure called `Set`.

```
- val x1 = mksetint [1,2];
val x1 = {1, 2} :: {int} : co
- smap (pair(id,id)) x1;
val it = {(1, 1), (2, 2)} :: {(int * int)} : co
- val x2 = mksetint [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
```

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' is derived from t by erasing all or-set brackets from t. 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 [8] and later refined in [7].

Definition: The *normal form* of a complex object is the or-set of all *conceptual representations* of the complex object, where

– for any x of base type, y is a conceptual representation of x if and only if $y = x$;
– (x', y') conceptually represents (x, y) if and only if x' conceptually represents x and y' conceptually represents y;
– x conceptually represents $\langle x_1, \ldots, x_n\rangle$ if and only if x conceptually represents one of $x_1, \ldots, x_n$;
– $\{x'_1, \ldots, x'_k\}$ conceptually represents $\{x_1, \ldots, x_n\}$ if and only if each $x' \in \{x'_1, \ldots, x'_k\}$ conceptually represents some $x \in \{x_1, \ldots, x_n\}$ and each $x \in \{x_1, \ldots, x_n\}$ is conceptually represented by some $x' \in \{x'_1, \ldots, x'_k\}$.

Normalization is supported in OR-SML by the addition of two new functions: `normalize` of SML type `co_type -> co_type` and `normal` of SML type `co -> co`. These two functions are sufficient to give OR-SML adequate power to work with conceptual representations of objects. The function `normalize`, when applied to a type t, returns the type of the or-set of conceptual representatives of objects of type t. As a primitive operator of OR-SML, `normal` has true type

$$\overline{\texttt{normal} : t \to \texttt{normalize}(t)}$$

The semantics of `normal x` is the or-set of conceptual representatives of `x`. For example, if we construct a pair `x` of an HOL90 term coupled with selections of atomic subtypes of the term's type, this is conceptually the same as a selection of pairs of the term coupled with one of its atomic subtypes.

```
- val x = mkprodco(mkbaseco (Term(--`f:ind -> 'a`--)),
                   mkorsco (map (mkbaseco o Type)
                                [(==`:ind`==),(==`:'a`==)]));
val x =
  ((Term ((--`f`--))),
   <(Type ((==`:'a`==))), (Type ((==`:ind`==)))
    >) :: (hol_theory_data * <hol_theory_data>) : co
- normalize (typeof x);
val it = <(hol_theory_data * hol_theory_data)> : co_type
- normal x;
val it =
  <((Term ((--`f`--))), (Type ((==`:ind`==)))),
   ((Term ((--`f`--))), (Type ((==`:'a`==))))
   > :: <(hol_theory_data * hol_theory_data)> : co
```

Many conceptual queries do not require that all objects in the normal form be constructed at once, before the query can be asked. Instead, processing elements of the normal forms one-by-one will often suffice. For example, in order to select normal form entries that satisfy a given criterion, one need not construct the entire normal form first. This suggest a different evaluation scheme for conceptual queries, that processes entries in the normal form one-by-one, while accumulating the result. OR-SML provides a number of functions that implement this query evaluation mechanism. We do not discuss this feature of OR-SML here. The interested reader may consult [7]. Notice that such a scheme greatly reduces the space usage, which is important for large size databases.

4 Additional features of the system

4.1 Primitives involving `hol_theory_data`

So far we have seen ways to create objects of type `co` out of objects of type `hol_theory_data`, and how to perform various generic constructs, like pairing and union on the resulting objects. However, the system must also provide a way of making functions on `hol_theory_data` into functions that fit into the type system of OR-SML. For example, we want to be able to write a function `thm_of_co : co -> co` that extracts the theorem component (as a complex object) from complex object consisting of the information about a stored theorem. Furthermore, there is a need for a mechanism to translate predicates on `hol_theory_data` into predicates on complex objects which can be used with operators like `cond` and `Set.select`.

The solution to this problem is given by the function `apply` which takes a function `f : hol_theory_data list -> hol_theory_data` 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. If `f_co` is applied to a complex object which is not a tuple of `hol_theory_data`, then it raises the exception `Cannotapply`. Bundling the arguments to a function in a list allows us to apply functions of arbitrarily many arguments over `hol_theory_data` to the corresponding complex objects.

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

```
- fun thm_of (Axiom {theorem = (_,thm),...}) = Thm thm
    | thm_of (Definition {theorem = (_,thm),...}) = Thm thm
    | thm_of (StoredThm {theorem = (_,thm),...}) = Thm thm
    | thm_of (x as Thm _) = x
    | thm_of _ = raise HOL_ERR{origin_structure = "top",
                               origin_function = "thm_of",
                               message = "Has no theorem component"};
val thm_of = fn : hol_theory_data -> hol_theory_data
```

The function `can` when applied to a function `f` and then to an argument `x` returns `true` if `(f x)` does not raise an exception, and `false` if it does. Using `can` with `thm_of` gives us a test for whether an `hol_theory_data` object has a theorem component. With this we are able to write a query to extract all the theorems from the complex object representing an HOL90 theory as follows:

```
- val thm_of_co = apply_unary thm_of;
val thm_of_co = fn : co -> co
- val has_thm_component_co = apply_test (can thm_of);
val has_thm_component_co = fn : co -> co
- val one_db = mk_theory_db "one";
val one_db =
  {(StoredThm{theorem = ("one_axiom", |- !f g. f = g),
     thy_name = "one"}),
   (StoredThm{theorem = ("one", |- !v. v = one), thy_name = "one"}),
   (StoredThm{theorem = ("one_Axiom", |- !e. ?!fn. fn one = e),
     thy_name = "one"}),
   (Definition{theorem = ("one_TY_DEF",
     |- ?rep. TYPE_DEFINITION (\b. b) rep),
     thy_name = "one"}),
   (Definition{theorem = ("one_DEF", |- one = (@x. T)),
     thy_name = "one"}),
   (Constant_tm{constant = (--`one--)`, thy_name = "one"}),
   (TypeOp{tyop = {Name = "one", Arity = 0}, thy_name = "one"}),
   (Parent {parent = "bool", thy_name = "one"})
  } :: {hol_theory_data} : co
- val one_stored_thms =
      smap thm_of_co (Set.select has_thm_component_co one_db);
val one_stored_thms =
```

```
{(Thm (|- one = (@x. T))),
 (Thm (|- ?rep. TYPE_DEFINITION (\b. b) rep)),
 (Thm (|- !e. ?!fn. fn one = e)), (Thm (|- !v. v = one)),
 (Thm (|- !f g. f = g))} :: {hol_theory_data} : co
```

4.2 Structural recursion

Structural recursion on sets [1] is a very powerful programming tool for query languages. Unfortunately, it is too powerful because it is often unsafe as an operation on sets. A function defined by structural recursion is not guaranteed to be well-defined as a function on sets or or-sets (*i.e.* two different presentations of the same set or or-set may yield unequal results), and well-definedness can not generally be checked by a compiler [3]. It is, however, often helpful in writing programs, 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` and `orsr` that take an object e of type t and a function f of type $s \times t \to t$ and return a function $\texttt{sr}(e, f)$ of type $\{s\} \to t$ or a function $\texttt{orsr}(e, f)$ of type $\langle s \rangle \to t$ respectively. Their semantics is as follows: $\texttt{sr}(e,f)\{x_1, \ldots, x_n\} = f(x_1, f(x_2, f(x_3, \ldots f(x_n, e) \ldots)))$ and similarly for `orsr`. That is, `sr` and `orsr` behave on sets and or-sets in much the same way as `fold` or `itlist` behaves on lists. The two functions implementing structural recursion are contained in the SML structure `SR`. As an example of the use of structural recursion, functions for converting sets to or-sets and vice versa can be defined as follows:

```
- val set_to_or = SR.sr(orempty, (fn (x,y) => orunion(orsng x, y)));
val set_to_or = fn : co -> co
- val or_to_set = SR.orsr(empty, (fn (x,y) => union(sng x, y)));
val or_to_set = fn : co -> co
- val a = mksetint [1,2,3,4,5];
val a = {1, 2, 3, 4, 5} :: {int} : co
- val b = set_to_or a;
val b = <1, 2, 3, 4, 5> :: <int> : co
- eq (a, or_to_set b);
val it = T :: bool : co
```

4.3 Deconstruction of complex objects

While we can use the system as described so far to query the HOL90 theories interactively to find theorems that might be useful in solving goals, we really want to be able to incorporate the results of such queries into further computations, such as tactics and conversions. Since all operations of OR-SML described so far produce elements of type `co`, there is a need to have a way out of complex objects to the usual SML types. The structure `DEST` contains the following functions to deconstruct complex objects and obtain ordinary SML values.

```
exception Cannotdestroy
val co_to_base : co -> hol_theory_data
val co_to_bool : co -> bool
val co_to_int : co -> int
```

```
val co_to_list : co -> co list
val co_to_pair : co -> co * co
val co_to_string : co -> string
```

It should be noted that because `DEST.co_to_list` takes an object whose elements are supposed to be treated as unordered and orders them, deconstruction of complex objects is inherently as unsafe (in the sense of allowing ill-defined functions over sets and or-sets) as structural recursion is.

4.4 Derived functions for HOL90 queries

To support some of the kinds of queries that users are most likely to perform when browsing the HOL90 theories, we have provided a structure `Hol_queries` containing the following functions:

```
val mk_theory_db : string -> co
val mk_all_theories_db : unit -> co
val type_test : (hol_type -> bool) -> hol_theory_data -> bool
val term_test : (term -> bool) -> hol_theory_data -> bool
val thm_test : (thm -> bool) -> hol_theory_data -> bool
val data_to_type : hol_theory_data -> hol_type
val data_to_term : hol_theory_data -> term
val data_to_thm : hol_theory_data -> thm
val db_find_thms : {test:thm -> bool, theory:string} -> co
val db_find_all_thms : (thm -> bool) -> co
val seek : {pattern:term, theory:string} -> co
val seek_all : term -> co
```

The functions `mk_theory_db` and `mk_all_theories_db` were mentioned earlier. Care is taken with these two functions to memoize their results on any proper ancestor theories to avoid their subsequent recomputation. The functions `type_test`, `term_test` and `thm_test` convert predicates on HOL90 types, terms and theorems respectively into predicates over `hol_theory_data`. They can be composed with `apply_test` to create predicates over the corresponding complex objects. The functions `data_to_type`, `data_to_term` and `data_to_thm` extract the type, term or theorem from the given `hol_theory_data`. The function `db_find_thms` returns the set of all theorems satisfying the given test in the named theory, while `db_find_all_thms` looks in all currently available theories. The function `seek` returns the set of all theorems in the named theory that contain a subterm that match the given pattern. `seek_all` is the corresponding function for looking in all the currently available theories. The functions `db_find_thms`, `db_find_all_thms`, `seek`, and `seek_all` all return complex objects so that further queries may be performed on the result. These functions are not sufficient for complex queries, but will handle the simple lookups, and they can be the starting point of more complex queries.

5 Using OR-SML and HOL90 together — an example

The main use to which the OR-SML extension of HOL90 has been put so far is browsing the theories for theorems which 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.

A very important class of user-defined types in HOL90 are those of recursive datatypes, including nested mutually recursive datatypes. Structural induction over these datatypes is often an important step in solving goals. Part of the process of defining a recursive datatype involves proving an "initiality theorem" (or pair of theorems) which states that a function over the datatype may be uniquely defined by cases over the constructors for the datatype. If a recursive datatype was defined by one of the automatic procedures for creating recursive datatypes, then such theorems have been stored in the theory database. 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 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 theorems.

Given a goal to be proved, one often wants to proceed by structural induction over any recursive datatypes over which the goal is universally quantified. Thus, we would like to know all principles of induction stored in the HOL90 theory database that are relevant to a given goal. However, a given type may or may not be a recursive datatype. If the type is a recursive datatype, initiality may be stated as one or two theorems, one for existence of the function and the other for uniqueness. Moreover, a polymorphic datatype may have instances which are components of several mutually recursive datatypes. To see this, consider the two datatype specifications:

$$\sigma\ \mathsf{list}\ =\ \mathsf{Nil}\ \mid\ \mathsf{Cons\ of}\ \sigma\ \ (\sigma\ \mathsf{list})$$

and

$$\sigma\ \mathsf{Tree}\ =\ \mathsf{Lf\ of}\ \sigma\ \mid\ \mathsf{Nd\ of}\ (\sigma\ \mathsf{Tree})\ \mathsf{list}$$

The type (σ Tree) list is an instance of the type σ list and is a component of the mutual recursion defining the type σ Tree. They provide us with the following two principles of induction:

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

and

$$\begin{array}{l} \forall R\ P.\ \ (\ (\forall\ n.\ R\ (\mathsf{Lf}\ n))\ \wedge\ (\forall\ l.\ P\ l\ \Longrightarrow\ R\ (\mathsf{Nd}\ l))\ \wedge \\ \qquad\qquad P\ \mathsf{Nil}\ \wedge\ (\forall\ t\ l.\ (R\ t\ \wedge\ P\ l)\ \Longrightarrow\ P\ (\mathsf{Cons}\ t\ l))\ )\ \Longrightarrow \\ \qquad\quad (\forall\ t.\ R\ t)\ \wedge\ (\forall\ l.\ P\ l) \end{array}$$

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 that 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 the set of entries from each ancestor theory, including the current theory;
`universal_types : term -> hol_type list` returns the list of the types of the leading universally quantified variables of a given term;
`is_initial_theorem_for : hol_type -> hol_theory_data -> bool` tests whether a given theorem is an initiality theorem for a given type;
`is_existential_for : hol_type -> hol_theory_data -> bool` tests whether a given theorem is the existential half of a statement of initiality for the given type;
`is_uniqueness_for : hol_theory_data -> hol_theory_data -> bool` when applied to the existential half of a statement of initiality, tests whether the second argument is the the corresponding uniqueness theorem.
`is_induction_for : hol_theory_data -> hol_theory_data -> bool` takes an initiality theorem, or the existential half, as the first argument and tests whether the second argument is the corresponding induction theorem.

The testing functions given above may be converted into ones which work with OR-SML by composing them with `apply_test`. When we wish to apply a function over `hol_theory_data` to a complex object (*i.e.* an OR-SML object) that we know is the equivalent of an object of `hol_theory_data` type, we may accomplish this by composing the original function with `DEST.co_to_base`.

```
fun is_initial_co_for ty = apply_test (is_initial_theorem_for ty)
fun is_existential_co_for ty = apply_test (is_existential_for ty)
fun is_uniqueness_co_for thm_co =
    apply_test (is_uniqueness_for (DEST.co_to_base thm_co))
fun is_induction_co_for thm_co =
    apply_test (is_induction_for (DEST.co_to_base thm_co))
```

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. We will gather each statement of initiality as a pair where either the first component is a theorem of initiality and the second component is the empty set, or the first component is the existential half of initiality and the second component is the set containing the uniqueness half. (Remember that OR-SML is a typed language, so we need to use the same type of representation in each case.)

```
fun mk_initial_co_for ty =
    comp (smap (fn thm_co => mkprodco(thm_co, empty)),
```

```
              Set.select (is_initial_co_for ty))

fun mk_exist_uniq_co_for ty theory_db =
    smap (fn existential_co =>
           mkprodco(existential_co,
                    Set.select (is_uniqueness_co_for existential_co)
                               theory_db))
         (Set.select (is_existential_co_for ty) theory_db)

fun mk_initiality_options ty =
    set_to_or (union (mk_initial_co_for ty all_theories_db,
                      mk_exist_uniq_co_for ty all_theories_db))
```

For each initiality statement we find for a given type, we want to find an induction theorem, if it exists. Again, we will use sets to allow for the possibility that none exists.

```
fun get_induct_thm_co init_thms_co =
    let val init_co = p1 init_thms_co  (* Either the initiality theorem,*)
                                       (* or the existential half.     *)
        val induct_co =
    Set.select (is_induction_co_for init_co) all_theories_db
    in mkprodco (init_thms_co, induct_co)
    end

fun mk_induction_options ty =
    orsmap get_induct_thm_co (mk_initiality_options ty)
```

For any one given type `ty`, the query `mk_induction_options ty` returns the or-set of pairs of initiality theorems and possible induction theorems. We need to accumulate this information over the list of types over which the goal is universally quantified. We wish to preserve the order in which the information is gathered, to match it with the order in which the types are universally quantified. Therefore, the result we return will be a tuple. We would like each entry in the tuple to be the or-set of possibilities for viewing the type as a recursive datatype. However, some types simply are not recursive datatypes. Therefore, we take advantage of the structural level of OR-SML to replace the empty or-set by the empty tuple, that is, `unit_co`, the unique element of unit type, to represent that the type of the universally quantified variable does 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.

```
fun fold_induction_options [] = unit_co
  | fold_induction_options (hd_ty :: tl_tys) =
    let val new_options = mk_induction_options hd_ty
    in cond(eq(new_options, orempty),
            (fn rem_co => mkprodco(unit_co,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))
```

Using a package for making nested recursive datatype definitions, we added to HOL90 the definition of the type `Tree` as given above. An example of finding the possible induction information for a goal over `Tree` in this setting is as follows: (The output has been abbreviated for the sake of space.)

```
- val poss_ind = goal_induction_options
    (--`!(n:num) l. ((Nd (CONS (Lf n) l)) = Nd (APPEND l [(Lf n)])) =
               (EVERY (\x. x = Lf n) l)`--);
val poss_ind =
  <((((StoredThm{theorem = ("num_Axiom", (* ... the theorem ... *)),
        thy_name = "prim_rec"}),
      {}),
     {(StoredThm{theorem = ("INDUCTION", (* ... the theorem ... *)),
        thy_name = "num"})}),
    ((((StoredThm{theorem = ("list_Axiom", (* ... the theorem ... *)),
       {}),
      {(StoredThm{theorem = ("list_INDUCT", (* ... the theorem ... *),
     ())),
   (  . . . (* same first tuple as above *) . . . ,
    ((((StoredThm{theorem = ("Tree_existence",
         |- !Lf_case Nd_case Tree_NIL_Tree_case Tree_CONS_Tree_case.
              ?y y'.
                (!x1. y (Lf x1) = Lf_case x1) /\
                (!x1. y (Nd x1) = Nd_case (y' x1) x1) /\
                (y' [] = Tree_NIL_Tree_case) /\
                (!x1 x2.
                  y' (CONS x1 x2) =
                  Tree_CONS_Tree_case (y x1) (y' x2) x1 x2)),
         thy_name = "Tree"}),
       {(StoredThm{theorem = ("Tree_unique",
          . . . (* the uniqueness theorem corresponding *) . . .
          . . . (* to the above existence theorem       *) . . . ),
          thy_name = "Tree"})}),
      {(StoredThm{theorem = ("Tree_induct",
         |- !Tree_Prop Tree_list_Tree_Prop.
              (!y. Tree_Prop (Lf y)) /\
              (!y. Tree_list_Tree_Prop y ==> Tree_Prop (Nd y)) /\
              Tree_list_Tree_Prop [] /\
              (!y y'.
                Tree_Prop y /\ Tree_list_Tree_Prop y' ==>
                Tree_list_Tree_Prop (CONS y y')) ==>
              (!x1. Tree_Prop x1) /\ (!x2. Tree_list_Tree_Prop x2)),
         thy_name = "Tree"})}),
     ()))
   > :: <(((hol_theory_data * {'a}) * {hol_theory_data}) *
```

```
          (((hol_theory_data * {hol_theory_data}) * {hol_theory_data}) *
           unit))> : co
- val number_of_options = length (DEST.co_to_list poss_ind);
val number_of_options = 2 : int
```

Thus there are two possible ways to proceed by induction. In each case, there is only one way to proceed by induction over the natural numbers. However, we have two different options for how to proceed by induction over lists of trees given to us by the second component of each tuple in `poss_ind`. We may either proceed by induction over lists, or by mutual induction over trees and lists of trees. Once having gathered this information in normal form, we may continue with further queries, such as which of the possibilities are missing the induction theorem and need to have it derived, or how many different ways are there to proceed.

In the above example, no further heap increases were required after the code from the library `nested_rec` was loaded. The heap at that time was 23 megabytes in HOL90 built using version 93 of SML-NJ. All database operations (including the calculation of `all_theories_db`) took place after the last major garbage collection. The time to run the query for `poss_ind` took 12 seconds of wall-clock time on a Sun Sparcstation 2. While more effort is needed to make queries more efficient in general, we feel that this is already acceptable performance for OR-SML to be a usable tool with HOL90 on moderately complicated queries.

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, it is our belief that the ability to make queries involving conjunctive and disjunctive information using OR-SML within the theorem prover HOL90 will enhance the end-user's ability to gather information appropriate for planning the proof of goals.

6 Conclusion

We describe a functional database language built on top of Standard ML and interfaced to HOL90. The set part of the core language (*i.e.* the primitives not involving or-sets) 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 at the structural level and at the conceptual level. 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 this paper we have described OR-SML as it connects to HOL90. OR-SML is also capable of being built as a stand-alone system, and as such, has certain features (for example file I/O, and the ability to handle multisets) that were not relevant to this setting and were not described here. A more complete description of the full system can be found in [6].

By interfacing HOL90 to a powerful, general purpose, database query language with good theoretical properties, such as a clear semantics, we believe we have provided a good tool for theory browsing as well as a solid platform for future work in constructing theorem-proving methodologies that make full use of previously developed theories.

References

1. 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.
2. 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.
3. 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.
4. L. Colby, A recursive algebra for nested relations, *Inform. Systems* 15 (1990), 567–582.
5. S. Grumbach, T. Milo, Towards tractable algebras for bags, *Proc. 12th Symposium on Principles of Database Systems*, Washington DC, 1993, pages 49–58.
6. E. Gunter and L. Libkin, OR-SML: A functional database programming language for disjunctive information and its applications, In: D. Karagiannis ed., *Proc. 5th Internat. Conf. on Database and Expert Systems Applications (DEXA'94),* Athens, Greece, September 1994. Springer-Verlag LNCS vol. 856, 1994, pp. 641–650.
7. L. Libkin. Normalizing incomplete databases. In *Proceedings of the 14th Symp. on Principles of Database Systems*, San Jose CA, 1995, pages 219–230.
8. L. Libkin and L. Wong, Semantic representations and query languages for or-sets, *Proceedings of the 12th Symp. on Principles of Database Systems*, Washington DC, 1993, pages 37–48.
9. 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.
10. R. Milner, M. Tofte, R. Harper, *"The Definition of Standard ML"*, The MIT Press, Cambridge, Mass, 1990.
11. 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.
12. H.-J. Schek and M. Scholl, The relational model with relation-valued attributes, *Inform. Systems* 11 (1986), 137–147.
13. S.J. Thomas and P. Fischer, Nested relational structures, in P. Kanellakis ed., *Advances in Computing Research: The Theory of Databases*, pages 269–307, JAI Press, 1986.
14. P.W. Trinder, Comprehension: A query notation for DBPLs, In *Proc. 3rd Int. Workshop on Database Progr. Languages*, 1991, pages 49–62, Morgan Kaufmann.

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