C Language interface to YAP

YAP provides the user with the necessary facilities for writing predicates in a language other than Prolog. We will describe the C language interface.

Before describing in full detail how to interface to C code, we will examine a brief example.

Assume the user requires a predicate my_process_id(Id) which succeeds when Id unifies with the number of the process under which YAP is running.

In this case we will create a my_process.c file containing the C-code described below.

```
#include "Yap/YapInterface.h"

static int my_process_id(void) {
      YAP_Term pid = YAP_MkIntTerm(getpid());
      YAP_Term out = YAP_ARG1;
      return(YAP_Unify(out,pid));
}
void init_my_predicates() {
      YAP_UserCPredicate("my_process_id", my_process_id, 1);
}
```

How to compile

The commands to compile the above file depend on the operating system. Under Linux (i386 and Alpha) you should use:

```
gcc -c -shared my_process.c
ld -shared -o my_process.so my_process.o
```

And could be loaded, under YAP, by executing the following prolog goal

```
load_foreign_files(['my_process'], [], init_my_predicates).
```

After loading that file the following prolog goal:

```
?- my_process_id(N)
```

would unify N with the number of the process under which Yap is running.

Terms - Primitives available to the C programmer

Several C typedefs are included in the header file yap/YapInterface.h to describe, in a portable way, the C representation of prolog terms. The user should write his programs using this macros to ensure portability of code across different versions of YAP.

The more important typedef is YAP_Term which is used to denote the type of a prolog term. Terms, from a point of view of the C-programmer, can be classified as follows:

- *uninstantiated variables*
- *instantiated variables*
- *integers*
- *floating-point numbers*
- *database references*
- *atoms*
- *pairs (lists)*
- *compound terms*

Manipulating Yap terms in C

The primitive

```
YAP_Bool YAP_IsVarTerm(YAP_Term t)
```

returns true iff its argument is an uninstantiated variable. Conversely the primitive

```
YAP_Bool YAP_NonVarTerm(YAP_Term t)
```

returns true iff its argument is not a variable.

The user can create a new uninstantiated variable using the primitive

```
YAP_Term YAP_MkVarTerm()
```

The following primitives can be used to discriminate among the different types of non-variable terms:

```
YAP_Bool YAP_IsIntTerm(YAP_Term t)
YAP_Bool YAP_IsFloatTerm(YAP_Term t)
```

```
YAP_Bool    YAP_IsDbRefTerm(YAP_Term        t)
YAP_Bool    YAP_IsAtomTerm(YAP_Term        t)
YAP_Bool    YAP_IsPairTerm(YAP_Term        t)
YAP_Bool    YAP_IsApplTerm(YAP_Term        t)
```

All the above primitives ensure that their result is dereferenced, i.e. that it is not a pointer to another term.

Manipulating Yap terms in C

The following primitives are provided for creating an integer term from an integer and to access the value of an integer term.

```
YAP_Term    YAP_MkIntTerm(YAP_Int        i)
YAP_Int     YAP_IntOfTerm(YAP_Term       t)
```

where YAP_Int is a typedef for the C integer type appropriate for the machine or compiler in question (normally a long integer). The size of the allowed integers is implementation dependent but is always greater or equal to 24 bits: usually 32 bits on 32 bit machines, and 64 on 64 bit machines.

The two following primitives play a similar role for floating-point terms

```
YAP_Term    YAP_MkFloatTerm(YAP_flt        double)
YAP_flt     YAP_FloatOfTerm(YAP_Term        t)
```

where flt is a typedef for the appropriate C floating point type, nowadays a double

Manipulating Yap terms in C - Atoms and compound terms

A special typedef YAP_Atom is provided to describe prolog atoms (symbolic constants). The two following primitives can be used to manipulate atom terms

```
YAP_Term   YAP_MkAtomTerm(YAP_Atom   at)
YAP_Atom   YAP_AtomOfTerm(YAP_Term   t)
```

The following primitives are available for associating atoms with their names

```
YAP_Atom   YAP_LookupAtom(char  * s)
char      *YAP_AtomName(YAP_Atom  t)
```

The function YAP_LookupAtom looks up an atom in the standard hash table. The functor YAP_AtomName returns a pointer to the string for the atom.

A pair is a Prolog term which consists of a tuple of two prolog terms designated as the head and the tail of the term. Pairs are most often used to build lists. The following primitives can be used to manipulate pairs:

```
YAP_Term   YAP_MkPairTerm(YAP_Term   Head,  YAP_Term  Tail)
```

```
YAP_Term    YAP_MkNewPairTerm(void)
YAP_Term    YAP_HeadOfTerm(YAP_Term t)
YAP_Term    YAP_TailOfTerm(YAP_Term t)
```

One can construct a new pair from two terms, or one can just build a pair whose head and tail are new unbound variables. Finally, one can fetch the head or the tail.

A compound term consists of a functor and a sequence of terms with length equal to the arity of the functor. A functor, described in C by the typedef Functor, consists of an atom and of an integer. The following primitives were designed to manipulate compound terms and functors

```
YAP_Term     YAP_MkApplTerm(YAP_Functor f, unsigned long int n, YAP_Term[] args)
YAP_Term     YAP_MkNewApplTerm(YAP_Functor f, int n)
YAP_Term     YAP_ArgOfTerm(int argno,YAP_Term ts)
YAP_Functor  YAP_FunctorOfTerm(YAP_Term ts)
```

The YAP_MkApplTerm function constructs a new term, with functor f (of arity n), and using an array args of n terms with n equal to the arity of the functor. YAP_MkNewApplTerm builds up a compound term whose arguments are unbound variables. YAP_ArgOfTerm gives an argument to a compound term. argno should be greater or equal to 1 and less or equal to the arity of the functor.

YAP allows one to manipulate the functors of compound term. The function YAP_FunctorOfTerm allows one to obtain a variable of type YAP_Functor with the functor to a term. The following functions then allow one to construct functors, and to obtain their name and arity.

```
YAP_Functor    YAP_MkFunctor(YAP_Atom a,unsigned long int arity)
YAP_Atom       YAP_NameOfFunctor(YAP_Functor f)
YAP_Int        YAP_ArityOfFunctor(YAP_Functor f)
```

Unification

YAP provides a single routine to attempt the unification of two prolog terms. The routine may succeed or fail:

```
Int         YAP_Unify(YAP_Term      a, YAP_Term   b)
```

The routine attempts to unify the terms a and b returning TRUE if the unification succeeds and FALSE otherwise.

From C back to Prolog

Newer versions of YAP allow for calling the Prolog interpreter from C. One must first construct a goal G, and then it is sufficient to perform:

```
YAP_Bool YapCallProlog(YAP_Term G)
```

the result will be FALSE, if the goal failed, or TRUE, if the goal succeeded. In this case, the variables in G will store the values they have been unified with. Execution only proceeds until finding the first solution to the goal, but you can call findall/3 or friends if you need all the solutions.

Writing predicates in C

We will distinguish two kinds of predicates:

deterministic predicates which either fail or succeed but are not backtrackable, like my_process_id;

backtrackable predicates which can succeed more than once.

The first kind of predicates should be implemented as a C function with no arguments which should return zero if the predicate fails and a non-zero value otherwise. The predicate should be declared to YAP, in the initialization routine, with a call to

```
void YAP_UserCPredicate(char *name, YAP_Bool *fn(), unsigned long int arity);
```

where name is the name of the predicate, fn is the C function implementing the predicate and arity is its arity.

For the second kind of predicates we need two C functions. The first one which is called when the predicate is first activated, and the second one to be called on backtracking to provide (possibly) other solutions. Note also that we normally also need to preserve some information to find out the next solution.

An example of a backtrackable predicate

As an example we will consider implementing in C a predicate n100(N) which, when called with an instantiated argument should succeed if that argument is a numeral less or equal to 100, and, when called with an uninstantiated argument, should provide, by backtracking, all the positive integers less or equal to 100.

To do that we first declare a structure, which can only consist of prolog terms, containing the information to be preserved on backtracking and a pointer variable to a structure of that type.

```
typedef struct {
    YAP_Term  next_solution;     /* the next solution */
  } n100_data_type;

n100_data_type    *n100_data;
```

The n100/1 predicate

We now write the C function to handle the first call:

```
static int start_n100()
{
        YAP_Term t = ARG1;
        YAP_PRESERVE_DATA(n100_data,n100_data_type);
        if(YAP_IsVarTerm(t)) {
            n100_data->next_solution = YAP_MkIntTerm(0);
            return(continue_n100());
         }
        if(!YAP_IsIntTerm(t) || YAP_IntOfTerm(t)<0 || YAP_IntOfTerm(t)>100) {
            YAP_cut_fail();
        } else {
            YAP_cut_succeed();
        }
}
```

The n100/1 predicate

The routine starts by getting the dereference value of the argument. The call to YAP_PRESERVE_DATA is used to initialize the memory which will hold the information to be preserved across backtracking. The first argument is the variable we shall use, and the second its type. Note that we can only use YAP_PRESERVE_DATA once, so often we will want the variable to be a structure.

If the argument of the predicate is a variable, the routine initializes the structure to be preserved across backtracking with the information required to provide the next solution, and exits by calling continue_n100 to provide that solution.

If the argument was not a variable, the routine then checks if it was an integer, and if so, if its value is positive and less than 100. In that case it exits, denoting success, with YAP_cut_succeed, or otherwise exits with YAP_cut_fail denoting failure.

The reason for using for using the functions YAP_cut_succeed and YAP_cut_fail instead of just returning a non-zero value in the first case, and zero in the second case, is that otherwise, if backtracking occurred later, the routine continue_n100 would be called to provide additional solutions.

The n100/1 predicate

The code required for the second function is

```
static int continue_n100()
{
       int n;
       YAP_Term t;
       YAP_Term sol = ARG1;
       YAP_PRESERVED_DATA(n100_data,n100_data_type);
       n = YAP_IntOfTerm(n100_data->next_solution);
       if( n == 100) {
            t = YAP_MkIntTerm(n);
            YAP_Unify(&sol,&t);
            YAP_cut_succeed();
         }
        else {
            YAP_Unify(&sol,&(n100_data->next_solution));
            n100_data->next_solution = YAP_MkIntTerm(n+1);
            return(TRUE);
         }
}
```

The n100/1 predicate

Note that again the macro YAP_PRESERVED_DATA is used at the beginning of the function to access the data preserved from the previous solution. Then it checks if the last solution was found and in that case exits with YAP_cut_succeed in order to cut any further backtracking. If this is not the last solution then we save the value for the next solution in the data structure and exit normally with 1 denoting success. Note also that in any of the two cases we use the function YAP_unify to bind the argument of the call to the value saved in n100_state-¿next_solution.

Note also that the only correct way to signal failure in a backtrackable predicate is to use the YAP_cut_fail macro.

Backtrackable predicates should be declared to YAP, in a way similar to what happened with deterministic ones, but using instead a call to

```
void YAP_UserBackCPredicate(char *name,
        int *init(), int *cont(),
        unsigned long int arity, unsigned int sizeof);
```

where name is a string with the name of the predicate, init and cont are the C functions used to start and continue the execution of the predicate, arity is the predicate arity, and sizeof is the size of the data to be preserved in the stack.

Writing predicates to connect to MySQL

To build a couple deductive database system we need:

- A logic system (Yap Prolog).
- A Prolog to SQL translator (`SQLCompiler.pl` written by Cristoph Draxler).
- A SQL system (MySQL).

We will use the C API of MySQL and of Yap to build a C program that implements the interface between the two systems.

The Prolog to SQL compiler is used at the Prolog level, compiled together with the Prolog program accessing a database, and translates the database goals into SQL.

Prolog to SQL compiler

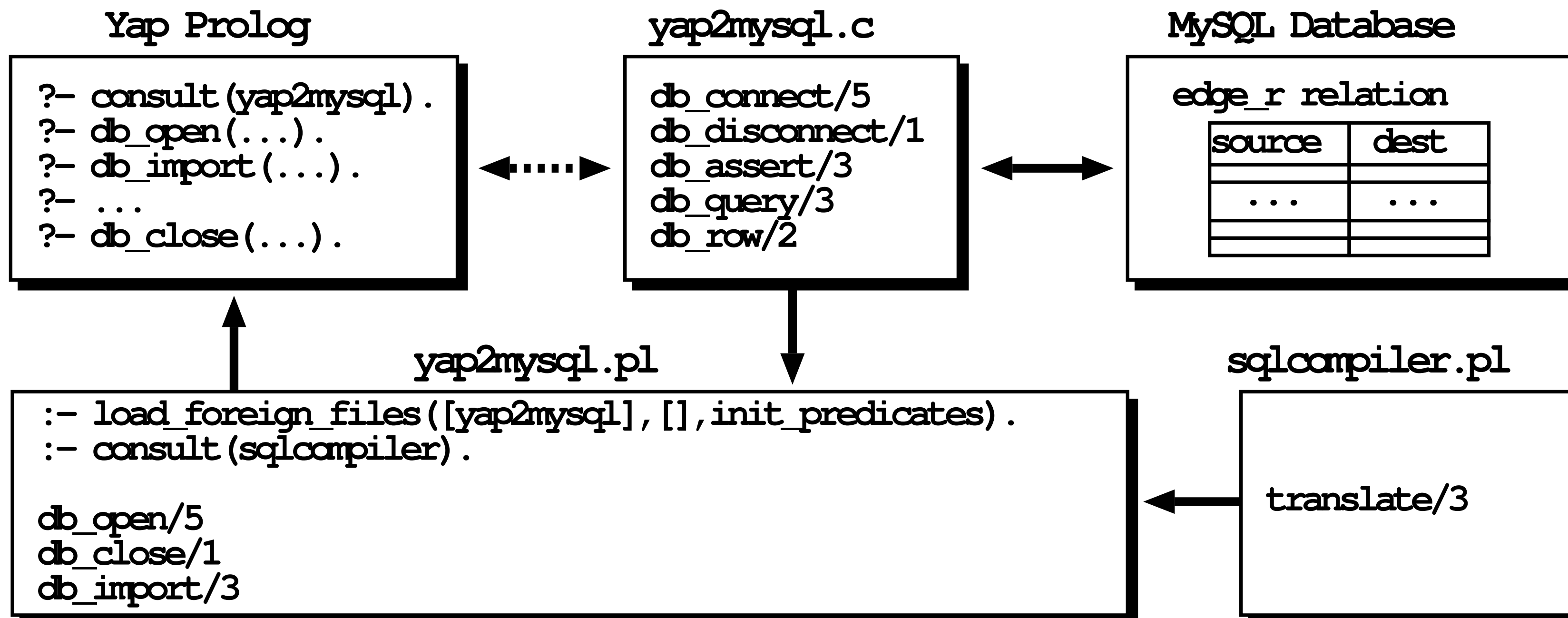
The generic Prolog to SQL compiler written by Draxler provides a `translate/3` predicate that receives as first argument a projection term, as second argument a database goal (query) and binds third argument with the SQL expression equivalent to the goal.

Example:

```
?- translate(edge(A,B),(edge(A,B),edge(B,A)),        RSQL),
   queries_atom(RSQL,SQL),
   write(SQL).

SELECT A.source,A.dest FROM edge A,edge B
WHERE B.source=A.dest AND B.dest=A.source;
```

Implementation architecture



Implemented Database predicates

- `db_open/5` - initializes a connection with the database server.
- `db_import/3` - associates a Prolog predicate with one database relation through an open connection.
- `db_close/1` - closes a connection with a database server.

```
db_open(Host,User,Passwd,Database,ConnName)            :-
    db_connect(Host,User,Passwd,Database,ConnHandler),
    set_value(ConnName,ConnHandler).

db_close(ConnName)        :-
    get_value(ConnName,ConnHandler),
    db_disconnect(ConnHandler).
```

Implementing `db_connect/5` through the Yap C interface

```
int c_db_connect(void) {
  YAP_Term arg_host = YAP_ARG1;
  YAP_Term arg_user = YAP_ARG2;
  YAP_Term arg_passwd = YAP_ARG3;
  YAP_Term arg_database = YAP_ARG4;
  YAP_Term arg_conn = YAP_ARG5;
  MYSQL *conn;
  char *host = YAP_AtomName(YAP_AtomOfTerm(arg_host));
  char *user = YAP_AtomName(YAP_AtomOfTerm(arg_user));
  char *passwd = YAP_AtomName(YAP_AtomOfTerm(arg_passwd));
  char *database = YAP_AtomName(YAP_AtomOfTerm(arg_database));
  conn = mysql_init(NULL);
  if (conn == NULL) {
    printf("erro no init\n");
    return FALSE;
  }

  if (mysql_real_connect(conn, host, user, passwd, database, 0, NULL, 0) == NULL) {
    printf("erro no connect\n");
    return FALSE;
  }
  if (!YAP_Unify(arg_conn, YAP_MkIntTerm((int) conn)))
    return FALSE;
  return TRUE;
}
```

Implementing `db_disconnect/1` through the Yap C interface

```
int
c_db_disconnect(void)    {
  YAP_Term  arg_conn  = YAP_ARG1;

  MYSQL  *conn  = (MYSQL  *) YAP_IntOfTerm(arg_conn);

  mysql_close(conn);
  return  TRUE;
}
```

Implementing `db_import/3`

Two alternatives:

- `Assert` tuples as Prolog facts.
- Access tuples one-at-a-time through backtracking.

Doing assert

```
db_import(RelationName,PredName,ConnName):-
    get_value(ConnName,ConnHandler),
    db_assert(ConnHandler,RelationName, PredName).

int c_db_assert(void) {
    ...                                     // declare auxiliary variables
    YAP_Functor f_pred, f_assert;
    YAP_Term t_pred, *t_args, t_assert;
    MYSQL *conn = (MYSQL *) YAP_IntOfTerm(YAP_ARG1);
    sprintf(query,"SELECT * FROM %s",YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2)) );
    mysql_query(conn, query);
    res_set = mysql_store_result(conn);
    arity = mysql_num_fields(res_set);       // get the number of column fields
    f_pred = YAP_MkFunctor(YAP_AtomOfTerm(YAP_ARG3), arity);
    f_assert = YAP_MkFunctor(YAP_LookupAtom("assert") , 1);
    while ((row = mysql_fetch_row(res_set)) != NULL) {
        for (i = 0; i < arity; i++) {   // test each column data type to...
            ...; t_args[i] = YAP_Mk...(row[i]);    ...;  // ...construct the...
        }                                          // ...appropriate term
        t_pred = YAP_MkApplTerm(f_pred, arity, t_args);
        t_assert = YAP_MkApplTerm(f_assert, 1, &t_pred);
        YAP_CallProlog(t_assert);             // assert the row as a Prolog fact
    }
    mysql_free_result(res_set);
    return TRUE;}
```

Through backtracking

```
edge(A,B) :-
    get_value(my_conn,ConnHandler),
    db_query(ConnHandler,'SELECT * FROM edge_r',ResultSet),
    db_row(ResultSet,[A,B]).

int c_db_query(void) {
    ...
    MYSQL *conn = (MYSQL *) YAP_IntOfTerm(YAP_ARG1);
    char *query = YAP_AtomName(YAP_AtomOfTerm(YAP_ARG2));
    mysql_query(conn, query);
    res_set = mysql_store_result(conn);
    return(YAP_Unify(YAP_ARG3, YAP_MkIntTerm((int) res_set)));
}
```

db_row/2

```
int c_db_row(void) {
  ...
  MYSQL_RES *res_set = (MYSQL_RES *) YAP_IntOfTerm(YAP_ARG1);
  if ((row = mysql_fetch_row(res_set)) != NULL) {
    YAP_Term head, list = YAP_ARG2;
    for (i = 0; i < arity; i++) {
      head = YAP_HeadOfTerm(list);
      list = YAP_TailOfTerm(list);
      if (!YAP_Unify(head, YAP_Mk...(row[i]))) return FALSE;
     }
    return TRUE;
  }
  mysql_free_result(res_set);
  YAP_cut_fail();
}
```

Transferring unification to the DBMS

```
edge(A,B)    :-
    get_value(my_conn,ConnHandler),
    translate(proj_term(A,B),       edge_r(A,B), QueryString),
    db_query(ConnHandler,QueryString, ResultSet),
    db_row(ResultSet,[A,B]).
```

Executing joins on the DBMS:

```
direct_cycle(A,B)    :- edge(A,B),   edge(B,A).

direct_cycle(A,B)    :-
    get_value(my_conn,ConnHandler),
    translate(proj_term(A,B),      (edge_r(A,B),edge_r(B,A)), SqlQuery),
    db_query(ConnHandler,SqlQuery,ResultSet),
    db_row(ResultSet,[A,B]).
```