Implementação de Sistemas de Bases de Dados Dedutivas

Existem duas abordagens principais:

• **Sistemas integrados**: o sistema de programação lógica e o sistema de gestão de bases de dados são desenvolvidos em conjunto de forma monolítica e integrada.

• **Sistemas acoplados**: existem dois sistemas separados, um de programação lógica e um de gestão de bases de dados que são interligados via uma interface de comunicação.

Nos sistemas integrados as consultas escritas em linguagem lógica podem ser implementadas directamente, sem necessidade de tradução para uma linguagem que seja entendida por um sistema de bases de dados SQL.

Ao contrário, nos sistemas acoplados as consultas escritas em linguagem lógica têm primeiramente que ser traduzidas para SQL, que é depois enviado para o SGBD relacional.

Implementação de Sistemas *Acoplados* de Bases de Dados Dedutivas

Ferramentas:

- Um sistema lógico (Yap Prolog).
- Um tradutor de Prolog para SQL (`SQLCompiler.pl` escrito por Cristoph Draxler).
- Um sistema SQL (MySQL).

A ideia geral da implementação é usar a API C do MySQL e a API C do Yap para construir um programa em C que implemente a interface entre os dois sistemas.

O compilador de Prolog para SQL, escrito em Prolog, é usado ao nível do programa Prolog a compilar, traduzindo as consultas lógicas a bases de dados para SQL.

A API C do Yap

Assim como muitos outros sistemas Prolog, o Yap fornece uma interface para escrever predicados noutras linguagens de programação, tais como C, como *módulos externos*.

Exemplo:

Assuma que pretende escrever um predicado `my_random(N)` que unifique `N` com um número aleatório. Para isto temos de criar um módulo `my_rand.c` com o seguinte código C:

```
#include "Yap/YapInterface.h"   // header file for the Yap interface to C

void init_predicates() {
   YAP_UserCPredicate("my_random",c_my_random,1);
}

int c_my_random(void) {
   YAP_Term number = YAP_MkIntTerm(rand());
   return(YAP_Unify(YAP_ARG1,number));
}
```

Exemplo de um módulo externo em C

O módulo deve ser compilado como *shared object* e carregado dentro do Yap através da seguinte directiva:

```
:- load_foreign_files([my_rand],[],init_predicates).
```

Depois disto a chamada `?- my_random(N).` unifica `N` com um número aleatório.

A declaração de `include` disponibiliza as macros para fazer a interface com o Yap.

A função `init_predicates()` indica ao Yap, através da chamada a `YAP_UserCPredicate()`, os predicados que são definidos no módulo.

A função `c_my_random()` é a implementação em C do predicado em causa.

De notar que esta função não tem argumentos apesar do predicado que ela define ter um. Os argumentos de um predicado Prolog escrito em C são acedidos via as macros `YAP_ARG1,...,YAP_ARG16`.

Primitivas para Manipulação de Termos Yap em C

Termo	Teste	Construtor	De-construtor
uninst var	YAP_IsVarTerm()	YAP_MkVarTerm()	(nenhum)
inst var	YAP_NonVarTerm()		
integer	YAP_IsIntTerm()	YAP_MkIntTerm()	YAP_IntOfTerm()
float	YAP_IsFloatTerm()	YAP_MkFloatTerm()	YAP_FloatOfTerm()
atom	YAP_IsAtomTerm()	YAP_MkAtomTerm() YAP_LookupAtom()	YAP_AtomOfTerm() YAP_AtomName()
pair	YAP_IsPairTerm()	YAP_MkNewPairTerm() YAP_MkPairTerm()	YAP_HeadOfTerm() YAP_TailOfTerm()
compound term	YAP_IsApplTerm()	YAP_MkNewApplTerm() YAP_MkApplTerm()	YAP_ArgOfTerm() YAP_FunctorOfTerm()
		YAP_MkFunctor()	YAP_NameOfFunctor() YAP_ArityOfFunctor()

Tipos de predicados definidos em módulos externos em C

O Yap distingue dois tipos de predicados que podem ser definidos em módulos externos em C:

- *predicados determinísticos*
- *predicados backtrackable*.

Os predicados determinísticos podem suceder ou falhar, mas nunca invocam backtracking em caso de falha.

Os predicados backtrackable podem suceder várias vezes, invocando backtracking para fornecer novas soluções.

O exemplo `my_random(N)` é um exemplo de um predicado determinístico, que é implementado por uma única função em C.

Os predicado backtrackable necessitam de duas funções para a sua definição, uma para a primeira vez em que o predicado é chamado e uma para as chamadas seguintes via backtracking.

Compilador de Prolog para SQL

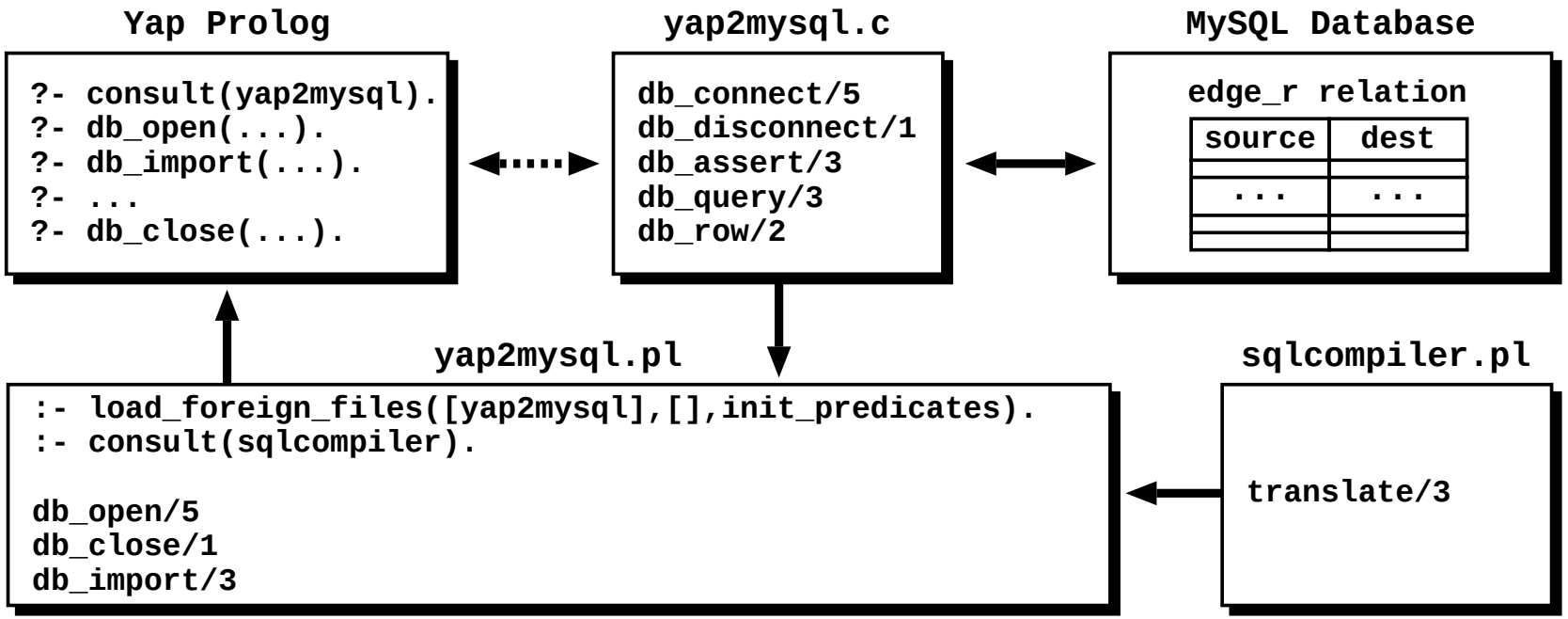
O compilador genérico de Prolog para SQL escrito por Christoph Draxler disponibiliza um predicado `translate/3` que recebe como primeiro argumento um termo de projecção, como segundo argumento um golo de base de dados (query) e retorna no terceiro argumento a representação do SQL equivalente a esse golo.

Exemplo:

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

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

Arquitectura Genérica da Implementação



Predicado de Bases de Dados Implementados

• `db_open/5` - inicializa uma conexão com o servidor de base de dados.

• `db_import/3` - associa um predicado Prolog a uma relação numa base de dados através de uma conexão.

• `db_close/1` - fecha uma ligação ao servidor de base de dados.

```
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).
```

Implementação do db_connect/5 através da API C

```
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;
}
```

Implementação do `db_disconnect/1` através da API C

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

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

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

Implementação do `db_import/3`

Duas alternativas:

- Fazer `assert` dos tuplos como factos Prolog.
- Aceder aos tuplos tuplo-a-tuplo via backtracking.

A primeira abordagem é a mais simples. Logo no início os tuplos são lidos e carregados dinâmicamento como factos Prolog. A partir daí não existe diferença entre esse predicados associados a relações em bases de dados e os restantes predicados.

A segunda abordagem faz com que sempre que um predicado associado a uma relação seja encontrado, se gere uma consulta SQL, sendo os respectivos tuplos disponibilizados como soluções ao Prolog via backtracking.

Esta segunda abordagem pode ainda ser melhorada tentando gerar consultas SQL que seleccionem apenas os tuplos que satisfazem as instanciações correntes das variáveis Prolog. Pode ser possível ainda *juntar* golos de bases de dados de forma a que seja o SGBD a calcular a junção ao invés do sistema Prolog (que é mais lento).

Abordagem de `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;}
```

Abordagem de 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();
}
```

Transferência da Unificação para o SGBD

```
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]).
```

Junção de Golos de base de dados:

```
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]).
```