Generic Cut Actions for External Prolog Predicates

Tiago Soares, Ricardo Rocha, and Michel Ferreira

DCC-FC & LIACC
University of Porto, 4150-180 Porto, Portugal
Tel: (+351) 226078830, Fax: (+351) 226003654
{tiagosoares,ricroc,michel}@ncc.up.pt

Abstract. An interesting feature of current Prolog systems is the ability to define external Prolog predicates that can be written in other languages. However, an important drawback of these interfaces is the fact that they lack some important features necessary to improve both the efficiency and the transparent integration of the Prolog system and the external predicates. Such an example is the *cut operation*. This operation is used quite often in Prolog programs, both for efficiency and semantic preservation. However, its use after a call to an externally defined predicate can have undesired effects. For example, if we have a pending connection to another application, or if we have memory blocks allocated by an external predicate, when a cut operation occurs, we may not be able to perform generic destruct actions, such as closing the pending connection or freeing the unnecessary memory. In this work, we propose an extension of the interface architecture that allows to associate generic user-defined functions with external predicates, in such a way that the Prolog engine transparently executes them when a cut operation occurs. We describe the implementation details of our proposal in the context of the Yap Prolog system.

Keywords: Prolog Systems Implementation, External Modules, Pruning.

1 Introduction

Logic programming provides a high-level, declarative approach to programming. Arguably, Prolog is the most popular logic programming language. Throughout its history, Prolog has demonstrated the potential of logic programming in application areas such as Artificial Intelligence, Natural Language Processing, Knowledge Based Systems, Database Management, or Expert Systems.

Prolog's popularity was sparked by the success of the WAM [1] execution model that has proved to be highly efficient for common computer architectures. The success obtained with the WAM led to further improvements and extensions. Such an example is the foreign-interface to other languages. An interesting feature of this interface is the ability to define external Prolog predicates that can be written in other languages. These predicates can then be used to combine

Prolog with existing programs or libraries, thereby forming coupled systems; or to implement certain critical operations that can speed up the execution. Since most language implementations are linkable to C, a widely implemented interface by most Prolog systems is a foreign-interface to the C language.

However, an important drawback of these interfaces is the fact that they lack some important features necessary to improve both the efficiency and the transparent integration of the Prolog system and the external predicates. Consider, for example, the *cut operation*. This operation is used quite often in Prolog programs, both for efficiency and semantic preservation. However, its use after a call to an externally defined predicate can have undesired effects. For example, if we have a pending connection to another application, or if we have memory blocks allocated by the external predicate, when a cut operation occurs we may not be able to perform generic destruct actions, such as closing the pending connection or freeing the unnecessary memory.

In this work we focus on the transparent use of the cut operation over external Prolog predicates. The motivation for this work appeared from the undesired effects of the cut operation in the context of our previous work [2] on coupling the Yap Prolog system [3] with the MySQL RDBMS [4], in order to obtain a deductive database system that takes advantage of the YapTab [5] tabling mechanism in a similar manner to the XSB system [6]. The effect of the cut operation in this context is well-known and can be so significant that systems such as XSB clearly state in the programmers' manual that cut operations should be used very carefully with relationally defined predicates [7]:

> *"The XSB-ODBC interface is limited to using 100 open cursors. When XSB systems use database accesses in a complicated manner, management of open cursors can be a problem due to the tuple-at-a-time access of databases from Prolog, and due to leakage of cursors through cuts and throws."*

To solve the problem between cuts and external predicates we propose an extension of the interface architecture that allows to associate generic user-defined functions with external predicates, in such a way that the Prolog engine transparently executes them when a cut operation occurs. With this functionality we can thus use these generic functions to avoid the kind of problems discussed above.

The idea of handling cuts transparently by setting actions to be executed on cut is not completely new. Systems like Ciao Prolog [8] and SWI Prolog [9] also provide support for external predicates with the possibility to set actions to be executed on cut. Our approach innovates because it uses the choice point data structure to easily detect when a cut operation occurs on externally defined predicates, and thus from the user's point of view, pruning standard predicates or externally defined predicates is equivalent. We describe the implementation details of our approach in the context of the Yap Prolog system and we use the coupling interface between Yap and MySQL as a working example. As we shall see, our implementation requires minor changes to the Yap engine and interface. Despite the fact that we have chosen this particular system, we believe that our

approach can be easily incorporated in other Prolog systems that are also based on the WAM execution model.

The remainder of the paper is organized as follows. First, we briefly describe the cut semantics of Prolog and the problem arising from its use to prune external predicates. Next, we present the C language interface of Yap and its implementation details. Then we describe the needed extension of the interface architecture in order to deal with pruning for external predicates. At the end we discuss some experimental results on the specific case of relational database interfaces and outline some conclusions.

2 Pruning External Predicates

Cut is a system built-in predicate that is represented by the symbol '**!**'. Its execution results in pruning all the branches to the right of the *cut scope branch*. The cut scope branch starts at the current node and finishes at the node corresponding to the predicate containing the cut.

Figure 1 gives a general overview of cut semantics by illustrating the left to right execution of an example with cuts. The query goal `a(X)` leads the computation to the first alternative of predicate `a/1`, where `!(a)` means a cut with scope node `a`. If `!(a)` gets executed, all the right branches until the node corresponding to predicate `a`, inclusively, should be pruned. Predicate `b(X)` is then called and suppose that it succeeds with its first alternative. Next, `!(a)` gets executed and all the remaining alternatives for predicates `a` and `b` are pruned. As a consequence, the nodes for `a` and `b` can be removed.

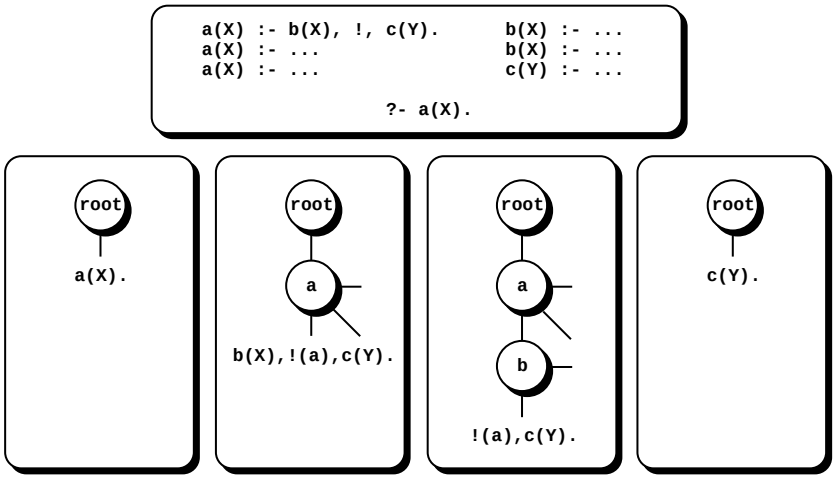

Fig. 1. Cut semantics

Consider now the coupling interface between a logic system and a database system. Logic goals are usually translated into SQL queries, which are then sent to the database system. The database system receives the query, processes it, and sends the resulting tuples back to the logic system. The usual method of accessing the tuples in the result set is to use the Prolog backtracking mechanism, which iteratively increments the result set pointer (*cursor*) and fetches the current tuple. Using this tuple-at-a-time access, the deallocation of the data structure holding the result set, whether on the server or on the client side, is only performed when the last tuple in the result set has been reached.

The problem is when, during the tuple-at-a-time navigation, a cut operation occurs before reaching the last tuple. If this happens, the result set cannot be deallocated. This can cause a lack of cursors and, more important, a lack of memory due to a number of very large non-deallocated data structures. Consider the example described in Fig. 1 and assume now that predicate `b/1` is a database predicate. The execution of `b(X)` will query the database for the corresponding relation and bind `X` with the first tuple that matches the query. Next, we execute `!(a)` and, as mentioned before, it will disable the action of backtracking for node `b`. The result set with the facts for `b` will remain in memory, although it will never be used.

3 The C Language Interface to Yap Prolog

Like other Prolog Systems, Yap provides an interface for writing predicates in other programming languages, such as C, as external modules. An important feature of this interface is how we can define predicates. Yap distinguishes two kinds of predicates: *deterministic predicates*, which either fail or succeed but are not backtrackable, and *backtrackable predicates*, which can succeed more than once.

Deterministic predicates are implemented as C functions with no arguments which should return zero if the predicate fails and a non-zero value otherwise. They are declared with a call to `YAP_UserCPredicate()`, where the first argument is the name of the predicate, the second the name of the C function implementing the predicate, and the third is the arity of the predicate.

For backtrackable predicates we need two C functions: one to be executed when the predicate is first called, and other to be executed on backtracking to provide (possibly) other solutions. Backtrackable predicates are declared with a call to `YAP_UserBackCPredicate()`. When returning the last solution, we should use `YAP_cut_fail()` to denote failure, and `YAP_cut_succeed()` to denote success. The reason for using `YAP_cut_fail()` and `YAP_cut_succeed()` instead of just returning a zero or non-zero value, is that otherwise, when backtracking, our function would be indefinitely called. For a more exhaustive description on how to interface C with Yap please refer to [10].

3.1 Writing Backtrackable Predicates in C

To explain how the C interface works for backtrackable predicates we will use a small example from the interface between Yap and MySQL. We present the `db_row(+ResultSet,?ListOfArgs)` predicate, which given a previously generated query result set, the `ResultSet` argument, fetches the tuples in the result set, tuple-at-a-time through backtracking, and, for each tuple, unifies the attribute values with the variables in the `ListOfArgs` argument. The code for the `db_row/2` predicate is shown next in Fig. 2.

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

void init_predicates() {
  YAP_UserBackCPredicate("db_row", c_db_row, c_db_row, 2, 0);
}

int c_db_row(void) {                    // db_row: ResultSet -> ListOfArgs
  int i, arity;
  YAP_Term arg_result_set, arg_list_args, head;
  MYSQL_ROW row;
  MYSQL_RES *result_set;

  arg_result_set = YAP_ARG1;
  arg_list_args = YAP_ARG2;
  result_set = (MYSQL_RES *) YAP_IntOfTerm(arg_result_set);
  arity = mysql_num_fields(result_set);
  if ((row = mysql_fetch_row(result_set)) != NULL) {    // get next tuple
    for (i = 0; i < arity; i++) {
      head = YAP_HeadOfTerm(arg_list_args);
      arg_list_args = YAP_TailOfTerm(arg_list_args);
      YAP_Unify(head, YAP_MkAtomTerm(YAP_LookupAtom(row[i])));
    }
    return TRUE;
  } else {                                                // no more tuples
    mysql_free_result(result_set);
    YAP_cut_fail();
    return FALSE;
  }
}
```

Fig. 2. The C code for the `db_row/2` predicate

Figure 2 shows some of the key aspects about the Yap interface. The include statement makes available the macros for interfacing with the Yap engine. The `init_predicates()` procedure tells Yap, by calling `YAP_UserBackCPredicate()`, the predicate defined in the module. The function `c_db_row()` is the implementation in C of the desired predicate. We can define a function for the first time the predicate is called and another for calls via backtracking. In this example the same function is used for both calls. Note that this function has no arguments even though the predicate being defined has two. In fact the ar-

guments of a Prolog predicate written in C are accessed through the macros YAP_ARG1, ..., YAP_ARG16 or with YAP_A(N) where N is the argument number.

The c_db_row() function starts by converting the first argument (YAP_ARG1) to the corresponding pointer to the query result set (MYSQL_RES *). The conversion is done by the YAP_IntOfTerm() macro. It then fetches a tuple from this result set, through mysql_fetch_row(), and checks if the last tuple as been already reached. If not, it calls YAP_Unify() to unify the values in each attribute of the tuple (row[i]) with the respective elements in arg_list_args and returns TRUE. On the other hand, if the last tuple has been already reached, it deallocates the result set, as mentioned before, calls YAP_cut_fail() and returns FALSE.

For simplicity of presentation, we omitted type checking procedures over MySQL attributes that must be done to convert each attribute to the appropriate term in Yap. For some predicates it is also useful to preserve some data structures across backtracking. This can be done by calling YAP_PRESERVE_DATA() to associate such space and by calling YAP_PRESERVED_DATA() to get access to it later. With these two macros we can easily share information between backtracking steps. This example does not need this preservation, as the cursor is maintained in the result set structure.

3.2 The Yap Implementation of Backtrackable Predicates

In Yap a backtrackable predicate is compiled using two WAM-like instructions, try_userc and retry_userc, as follows:

```
try_userc c_first arity extra_space
retry_userc c_back arity extra_space
```

Both instructions have three arguments: the c_first and c_back arguments are pointers to the C functions associated with the backtrackable predicate, arity is the arity of the predicate, and extra_space is the memory space used by the YAP_PRESERVE_DATA() and YAP_PRESERVED_DATA() macros.

When Yap executes a try_userc instruction it uses the choice point stack to reserve as much memory as given by the extra_space argument, next it allocates and initializes a new choice point (see Fig. 3), and then it executes the C function pointed by the c_first argument. Later, if the computation backtracks to such choice point, the retry_userc instruction gets loaded from the CP_AP choice point field and the C function pointed by the c_back argument is then executed.

In order to repeatedly execute the same c_back function when backtracking to this choice point, the retry_userc instruction maintains the CP_AP field pointing to itself. This is the reason why we should use YAP_cut_succeed() or YAP_cut_fail() when returning the last solution for the predicate, as otherwise the choice point will remain in the choice point stack and the c_back function will be indefinitely called.

The execution of the YAP_PRESERVE_DATA() and the YAP_PRESERVED_DATA() macros in the C functions corresponds to calculate the starting position of the reserved space associated with the extra_space argument. For both functions, this is the address of the current choice point pointer plus its size and the arity of the predicate.

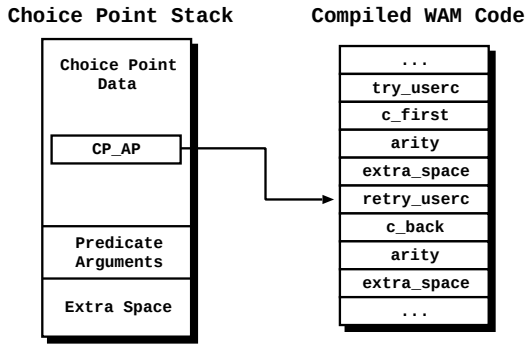

Fig. 3. The Yap implementation of backtrackable predicates

4 Generic Cut Actions for External Predicates

In this section, we discuss how we can solve the problem of pruning external predicates. We discuss two different approaches: a first approach where the user explicitly calls special predicates that perform the required action; and a second approach, which is our original proposal, where the system transparently executes a generic cut action when a cut operation occurs. To support our discussion, we will consider again the coupling interface between Yap and MySQL. However and as we shall see, our approach can be generalised to handle not only database predicates, but also any external predicate that requires a generic action over a cut.

4.1 Handling Cuts Explicitly

In this approach the user has to explicitly call a special predicate to be executed when a cut operation is performed over external predicates. For instance, for the interface between Yap and MySQL, the idea is as follows: before executing a cut operation that potentially prunes over databases predicates, the user must explicitly call a predicate that releases beforehand the result sets for the predicates to be pruned.

If we consider again the example from Fig. 1 and assume that `b(X)` is a database predicate, we might think of a simple solution: every time a database predicate is first called, we store the pointer for its result set in an auxiliary stack frame (we can extend the `c_db_row()` function to implement that). Then we could implement a new C predicate, `db_free_result/0` for example, that deallocates the result set in the top of the stack. Having this, we could adapt the code for the first clause of predicate `a/1` to:

```
a(X) :- b(X), db_free_result, !, c(X).
```

To use this approach, the user must be careful to always include a call to `db_free_result/0` before a cut operator. A problem with this `db_free_result/0`

predicate occurs if we call more than one database predicate before a cut. Consider the following definition for the predicate `a/1`, where `b1/1` and `b2/1` are database predicates.

```
a(X) :- b1(X), b2(Y), db_free_result, !, c(X).
```

The `db_free_result/0` will only deallocate the result set for `b2/1`, leaving the result set for `b1/1` pending. A possible solution for this problem is to *mark* beforehand the range of predicates to consider. We can thus implement a new C predicate, `db_cut_mark/0` for example, that *marks* where to cut to, and change the `db_free_result/0` predicate to free all the result sets within the mark left by the last `db_cut_mark/0` predicate.

```
a(X) :- db_cut_mark, b1(X), b2(Y), db_free_result, !, c(X).
```

A more intelligent and transparent solution is implemented in Ciao Prolog system [8]. The user still has to explicitly call a special predicate, the '`!!`'/0 predicate, which will execute the cut goal specified using a special primitive `det_try(Goal,OnCutGoal,OnFailGoal)`. For our `db_row/2` example this would be `det_try(db_row(ResultSet,_),db_free_result(ResultSet),fail)`, and now if a `db_row/2` choice point is pruned using a '`!!`', the associated result set is deallocated by `db_free_result/1`.

This solution solves the problem of cursor leaks and memory deallocation when pruning database predicates. However, because it relies on calling special predicates, the transparency in the use of relationally defined predicates is lost. Moreover, if the user happens to mistakenly use a '`!`' instead of a '`!!`', incorrect behaviour may happen the next time a '`!!`' is used [8].

4.2 Handling Cuts Transparently

We next present our approach to handle cuts transparently. As we shall see, this requires minor changes to the Yap engine and interface. First, we extended the procedure used to declare backtrackable predicates, `YAP_UserBackCPredicate()`, to include an extra C function. Remember that for backtrackable predicates we used two C functions: one to be executed when the predicate is first called, and another to be executed upon backtracking. The extra function is where the user should declare the function to be executed in case of a cut, which for database predicates will involve the deallocation of the result set. Declaring and implementing this extra function is the only thing the user needs to do to take advantage of our approach. Thus, from the user's point of view, pruning standard predicates or relationally defined predicates is then equivalent.

With this extra C function, the compiled code for a backtrackable predicate now includes a new WAM-like instruction, `cut_userc`, which is used to store the pointer to the extra C function, the `c_cut` argument.

```
try_userc c_first arity extra_space
retry_userc c_back arity extra_space
cut_userc c_cut arity extra_space
```

When now Yap executes a `try_userc` instruction, it also allocates space for a cut frame data structure (see Fig. 4). This data structure includes two fields: `CF_inst` is a pointer to the `cut_userc` instruction in the compiled code for the predicate and `CF_previous` is a pointer to the previous cut frame on stack. A top cut frame global variable, `TOP_CF`, always points to the youngest cut frame on stack. Frames form a linked list through the `CF_previous` field.

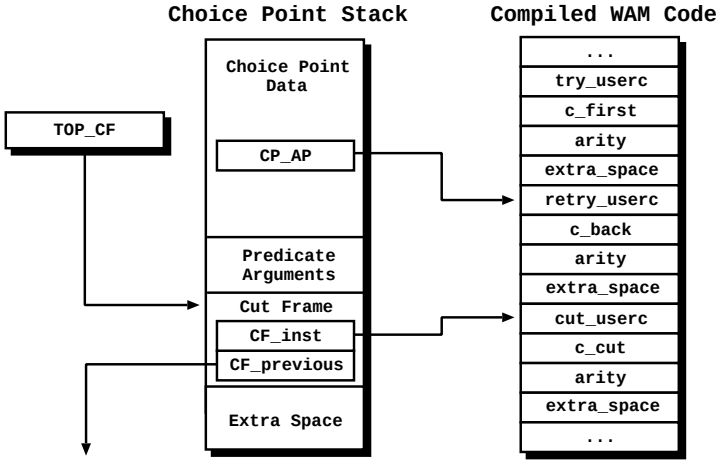

Fig. 4. The Yap support for generic cut actions

By putting the cut frame data structure below the associated choice point, we can easily detect the external predicates being pruned when a cut operation occurs. To do so, we extended the implementation of the cut operation to start by executing a new `userc_cut_check()` procedure (see Fig. 5). Remember that a cut operation receives as argument the choice point to cut to. Thus, starting from the `TOP_CF` variable and going through the cut frames, we can check if a cut frame will be pruned. If so, we load the `cut_userc` instruction stored in the corresponding `CF_inst` field in order to execute the cut function pointed by the `c_cut` argument. The `userc_cut_check()` procedure is like a handler in other programming languages that throws an exception, that is, executes the `cut_userc` instruction for the cut frames being pruned, when it encounters an abnormal condition that it can not handle itself, in this case, a cut operation over an externally defined predicate.

The process described above is done before executing the original code for the cut instruction, that is, before updating the global registry `B` (pointer to the current choice point on stack). This is important to prevent the following situation. If the cut function executes a `YapCallProlog()` macro to call the Prolog engine from C, this might have the side-effect of allocating new choice points on stack. Thus, if we had updated the `B` register beforehand, we will

```
void userc_cut_check(choiceptr cp_to_cut_to) {
  while (TOP_CF < cp_to_cut_to) {
    execute_cut_userc(TOP_CF->CF_inst);
    TOP_CF = TOP_CF->CF_previous;
  }
  return;
}
```

Fig. 5. The pseudo-code for the `userc_cut_check()` procedure

potentially overwrite the cut frames stored in the pruned choice points and avoid the possibility of executing the corresponding cut functions.

As a final remark, note that we can also call the `YAP_PRESERVED_DATA()` macro from the cut function to access the data store in the extra space. We thus need to access the extra space from the cut frames. This is why we store the cut frames above the extra space. The starting address of the extra space is thus obtained by adding its size to the pointer of the current cut frame.

To show how the extended interface can be used to handle cuts transparently, we next present in Fig. 6 the code for generalising the `db_row/2` predicate to perform the cursor closing upon a cut.

```
void init_predicates() {
  YAP_UserBackCPredicate("db_row", c_db_row_first, c_db_row,
                         c_db_row_cut, 2, sizeof(MYSQL_RES *));
}

int c_db_row_first(void) {
  MYSQL_RES **extra_space;
  ...                              // the same as for the c_db_row function
  result_set = (MYSQL_RES *) YAP_IntOfTerm(arg_result_set);
  YAP_PRESERVE_DATA(extra_space, MYSQL_RES *);  // initialize extra space
  *extra_space = result_set;       // store the pointer to the result set
  ...                              // the same as for the c_db_row function
}

int c_db_row(void) {
  ...                                                  // the same as before
}

void c_db_row_cut(void) {
  MYSQL_RES **extra_space, *result_set;

  YAP_PRESERVED_DATA(extra_space, MYSQL_RES *);
  result_set = *extra_space;         // get the pointer to the result set
  mysql_free_result(result_set);
  return;
}
```

Fig. 6. Extending the `db_row/2` predicate to handle cuts transparently

First, we need to define the function to be executed when a cut operation occurs. An important observation is that this function will be called from the cut instruction, and thus it will not be able to access the Prolog arguments, `YAP_ARG1` and `YAP_ARG2`, as described for the `c_db_row()` function. However, we need to access the pointer to the corresponding result set in order to deallocate it. To solve this, we can use the `YAP_PRESERVE_DATA()` macro to preserve the pointer to the result set. As this only needs to be done when the predicate is first called, we defined a different function for this case. The `YAP_UserBackCPredicate()` macro was thus changed to include a cut function, `c_db_row_cut()`, and to use a different function when the predicate is first called, `c_db_row_first()`. The `c_db_row()` function is the same as before (see Fig. 2). The last argument of the `YAP_UserBackCPredicate()` macro defines the size of the extra space for the `YAP_PRESERVE_DATA()` and `YAP_PRESERVED_DATA()` macros.

The `c_db_row_first()` function is an extension of the `c_db_row()` function. The only difference is that it uses the `YAP_PRESERVE_DATA()` macro to store the pointer to the given result set in the extra space for the current choice point. On the other hand, the `c_db_row_cut()` function uses the `YAP_PRESERVED_DATA()` macro to be able to deallocate the result set when a cut operation occurs. With these two small extensions, the `db_row/2` predicate is now protected against cuts and can be safely pruned by further cut operations.

5 Experimental Results

In this section we evaluate the performance of our generic cut action mechanism on the problem we have been addressing of relational queries result sets deallocation. As we mentioned in the introduction, the XSB manual recommends the careful use of *cut* with relationally defined predicates and recommends the following solution:

> *"When XSB systems use database accesses in a complicated manner, management of open cursors can be a problem due to the tuple-at-a-time access of databases from Prolog, and due to leakage of cursors through cuts and throws.* ***Often, it is more efficient to call the database through set-at-a-time predicates such as findall/3, and then to backtrack through the returned information.****"*

Using this solution it is clear that a *cut* operation will provide the correct pruning over database tuples, as tuples are now stored on the WAM heap, but at the sacrifice of execution time, as we will show.

The existing literature also lacks a comparative performance evaluation of the coupling interfaces between a logic system and a relational database system and, in this section, we also try to contribute to the benchmarking of such systems. We compare the performance of XSB 2.7.1, Ciao 1.10 and Yap 4.5.7, accessing a relational database. Yap has an ODBC interface and a native interface to MySQL. XSB has an ODBC interface and Ciao has a native interface to MySQL.

We used MySQL Server 4.1.11, both for the native and ODBC interfaces, running on the same machine, an AMD Athlon 1000 with 512 Mbytes of RAM.

This performance evaluation is directed to the cut treatment on the different systems. For this purpose, we created a relation in MySQL using the following SQL declaration:

```
CREATE TABLE table1 (
   num1    INT NOT NULL,
   PRIMARY KEY (num1));
```

and populated it with $1,000$, $100,000$ and $1,000,000$ tuples. This relation was imported as the `db_relation/1` predicate. To evaluate cut treatment over this predicate we created the two queries presented in Fig. 7.

```
query1 :- db_relation(Tuple),
          test(Tuple),
          !.

query2 :- findall(Element,db_relation(Element),List),
          member(Tuple,List),
          test(Tuple),
          !.

% test predicate for 1,000 tuples
test(500).

% test predicate for 100,000 tuples
test(50000).

% test predicate for 1,000,000 tuples
test(500000).
```

Fig. 7. Queries evaluated

Query 1 represents the typical backtracking search as is trivially implemented in Prolog. We want to find a particular value among the tuples of the database relation, testing each one with the `test/1` predicate, and execute a `cut` after finding it. On average we should go through half of the tuples, and so our `test/1` predicate succeeds when this middle tuple has been reached.

Query 2 follows the approach recommended on the XSB manual. Tuples are stored in a Prolog list [11], which is kept on the WAM heap data structure and thus works properly with cuts. Backtracking goes through the elements of the list using the `member/2` predicate. The same `test/1` predicate is used. Note that, after running the SQL query on the database server, the result set is stored completely on the client side both in query 1 and query 2. The main difference between query 1 and query 2 is that the native result set structure is used in query 1, while in query 2 navigation is performed on a Prolog list structure.

Table 1 presents the execution time for query 1 and the difference in allocated memory at the end of the query, for Yap with (Yap^+) and without (Yap^-)

our generic cut action mechanism. Table 2 presents the execution time and the difference in allocated memory for Yap$^+$, XSB and Ciao (for Ciao we used '`!!`' instead of '`!`') in both queries. We measured the execution time using `walltime`. The memory values are retrieved by consulting the `status` file of the system being tested in the `/proc` directory. All of the time results are in seconds, and the memory results are in Kbytes. Queries 1 and 2 are run for a $1,000$, $100,000$ and $1,000,000$ tuples.

Interface	System	Memory (Kb)			Running Time (s)		
		1K	100K	1M	1K	100K	1M
ODBC	Yap$^+$	0	72	72	0.005	0.481	4.677
	Yap$^-$	0	3324	31628	0.005	0.474	4.625
MySQL	Yap$^+$	0	60	60	0.004	0.404	4.153
	Yap$^-$	0	3316	31616	0.005	0.423	4.564

Table 1. Query 1 results for Yap with and without our generic cut action mechanism

There is one straightforward observation that can be made from Table 1. As expected, memory comparison for query 1 translates the fact that Yap$^-$ cannot deallocate the result set pruned by a cut. As a result, memory size will grow proportionally to the size of the result sets and to the number of queries pruned.

Interface	System	Query	Memory (Kb)			Running Time (s)		
			1K	100K	1M	1K	100K	1M
ODBC	Yap$^+$	Query 1	0	72	72	0.005	0.481	4.677
		Query 2	0	8040	73512	0.008	0.720	7.046
	XSB	Query 1	0	3284	31584	0.008	0.735	7.068
		Query 2	0	4088	39548	0.012	1.191	11.967
MySQL	Yap$^+$	Query 1	0	60	60	0.004	0.404	4.153
		Query 2	0	8028	73500	0.006	0.640	6.474
	Ciao	Query 1 (!!)	176	204	204	0.015	1.645	16.467
		Query 2 (!!)	36	13708	n.a.	0.036	3.820	n.a.

Table 2. Queries results for Yap, XSB and Ciao

Regarding Table 2 there are some interesting comparisons that can be made. According to the approach suggested on the XSB manual to deal with cuts, query 2 can be seen to be around 1.5 to 2 times slower than query 1 for all interface/system combinations. These results thus confirm our observation that this approach sacrifices the execution time. Memory results for query 2 are not so relevant because at the end of query 2 no memory is left pending. The results obtained in Table 2 simply reflect the fact that, during evaluation, the execution stacks were expanded to be able to store the huge Prolog list constructed by the

`findall/3` predicate. For Ciao we were not able to run query 2 for the list term with 1,000,000 tuples.

For query 1, we can compare Yap with XSB interfacing MySQL through an equivalent ODBC driver and Yap with Ciao interfacing MySQL through the MySQL C API. In terms of memory comparison for query 1, the results obtained for XSB translate the fact that it cannot deallocate the result set pruned by a cut (remember that for Ciao we used '`!!`' to correctly deal with cuts). In terms of execution time for query 1, Yap is a little faster than XSB and around 4 times faster than Ciao. This is probably due to the fact that Ciao negotiates the connection with MySQL server at each Prolog goal, which causes important slow-downs.

We should mention that in order to use the '`!!`' approach of Ciao we modified the Ciao source code to use the special primitive `det_try/3` as described in subsection 4.1 to correctly deallocate the pending result sets when a '`!!`' occurs. By default, Ciao uses a different approach to deal with pending results sets. It uses the *tuple-at-a-time* functionality of the MySQL C API and negotiates the connection with MySQL server at each Prolog goal.

MySQL offers two alternatives for sending tuples to the client program that generated the query: **(i)** store the set of tuples on a data structure on the database server side and send each tuple *tuple-at-a-time* to the client program; or **(ii)** store the set of tuples on a data structure on the client side sending the *set of tuples at once*. Note that with a tuple-at-a-time transfer between server and client, the problem of pruned result sets also happens, though leaving the allocated result set in the MySQL Server process and not in the Prolog process. However, with the negotiation of connections at the goal level, Ciao can overcome this problem, by closing connection with MySQL and therefore deallocating result sets, but at the price of execution time, as is clear from Table 2.

6 Concluding Remarks

We discussed the problem of pruning externally defined Prolog predicates and we proposed a completely transparent approach where the Prolog engine executes a user-defined function when a cut operation occurs. We implemented our proposal in the context of the Yap Prolog system with minor changes to the Yap engine and interface. Our approach is applicable to any predicate that requires a generic action over a cut.

We evaluated the performance of our generic cut action mechanism on the problem we have been addressing of relational queries result sets deallocation, using the Yap, XSB and Ciao Prolog systems interfacing MySQL for two queries where a cut was executed over a large database extensional predicate. We observed that, when using a typical backtracking search mechanism without support for deallocation of result sets over a cut, this can cause memory to grow arbitrarily due to the very large number of non-deallocated data structures. Alternatively, if we follow an approach like the one recommended on the XSB

manual to avoid this problem or if we maintain the result sets on the MySQL server as in Ciao, we observed that we need to sacrifice the execution time.

Acknowledgements

This work has been partially supported by MYDDAS (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Pluri-anual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International (1983)
2. Ferreira, M., Rocha, R., Silva, S.: Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases. In: Colloquium on Implementation of Constraint and LOgic Programming Systems. (2004) 71–82
3. Santos Costa, V.: Optimising Bytecode Emulation for Prolog. In: Principles and Practice of Declarative Programming. Number 1702 in LNCS, Springer-Verlag (1999) 261–267
4. Widenius, M., Axmark, D.: MySQL Reference Manual: Documentation from the Source. O'Reilly Community Press (2002)
5. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
6. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM SIGMOD International Conference on the Management of Data, ACM Press (1994) 442–453
7. Sagonas, K., Warren, D.S., Swift, T., Rao, P., Dawson, S., Freire, J., Johnson, E., Cui, B., Kifer, M., Demoen, B., Castro, L.F.: (XSB Programmers' Manual) Available from `http://xsb.sourceforge.net`.
8. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López, P., Puebla, G.: (Ciao Prolog System Manual) Available from `http://clip.dia.fi.upm.es/Software/Ciao`.
9. Wielemaker, J.: (SWI-Prolog Reference Manual) Available from `http://www.swi-prolog.org`.
10. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: (YAP User's Manual) Available from `http://www.ncc.up.pt/~vsc/Yap`.
11. Draxler, C.: Accessing Relational and NF^2 Databases Through Database Set Predicates. In: UK Annual Conference on Logic Programming. Workshops in Computing, Springer Verlag (1992) 156–173