Dealing with large predicates: exo-compilation in the WAM and in Mercury

Bart Demoen[&], Phuong-Lan Nguyen[@],
Vítor Santos Costa[#], and Zoltan Somogyi[*]

& Department of Computer Science, K.U.Leuven, Belgium
[@] Institut de Mathématiques Appliquées, UCO, Angers, France
LIACC/FCUP, Universidade do Porto, Portugal
[*] Department of Computer Science and Software Engineering,
University of Melbourne, Australia

bmd@cs.kuleuven.be, nguyen@ima.uco.fr, vsc@dcc.fc.up.pt, zs@csse.unimelb.edu.au

Abstract. Logic programming systems often need to deal with large but otherwise regular predicates, e.g. wide ground facts. Such predicates can be treated as any other predicate by the compiler, but there are good reasons to treat them specially, the most important being that separating the code from the data really pays off. We call the technique exo-compilation: it reduces the memory needed for the code to about one third of the normal WAM compilation schema without undue slowdown. As a bonus, queries with lots of void variables get a significantly better treatment. We first introduce the idea of exo-compilation by an example and present its implementation in hProlog. We show how other optimisations can be built on top of it, and evaluate how it performs in practice. We then show that the same ideas can also be applied (albeit in a slightly different form) to Mercury, whose implementation is based on a very different principles.

1 Introduction

In some applications, logic programming systems must run queries over predicates consisting of a large set of *wide facts*, i.e. facts where the arity of the predicate is large. As one example, such predicates appear in Machine Learning *datasets* ranging from gene expression databases, where the biological study may include a large number of activity observations for the same gene, to medical reports, where the doctor may annotate a large number of possible conditions and parameters of interest, to film databases, where communities may store a large number of fields of interest per film and performer, and so on. Wide predicates also have been used to represent properties input stream, or to represent tables for scanner and parser generators.

One key observation is that most often these facts are *typed and moded nicely*, i.e. the facts are ground and the arguments are all constants. Traditional, WAM-based Prolog implementations do not exploit this situation. Moreover, such predicates are often called with only a few arguments instantiated (typically, but not always, a key or sub-key of the relation), and most arguments are *void*. Note that in WAM parlance one says a variable is *void* if it is singleton in the query. The WAM does have limited support for void variables that can be recognised at compile-time. However, in this case, different queries will have different void variables. Therefore, the WAM must process void variables as it processes any other logical variable, forcing both the caller and the callee side to do an amount of work that is linear in the number of these variables. We would prefer to avoid that work altogether.

We show in Section 2 how both issues (wide facts and void variables) can be addressed by a technique where data and code are separated and which was named *exo-compilation* by the third author. We discuss next how exo-compilation was implemented in hProlog (see [6]): Section 3 contains an experimental evaluation thereof. Section 4 discuss how exo-compilation can be applied to other forms of regular code. The ideas of exo-compilation also apply to other logic programming systems. Section 5 shows how Mercury deals with large predicates: there is a great deal of similarity between the emulator and compiler approaches, but it is worth showing the details in both contexts. Section 6 discusses related work and concludes.

2 Exo-compilation

We first introduce some conventions that will be useful when showing abstract machine code.

- when an instruction refers to the i^{th} WAM argument register, we denote that by A(i), as in `getatom A(3), foo`
- the instruction *try* (and others) takes as argument a number that represents an arity, say 3 - we denote this as `try arity(3)`

Other operands are adorned in similarly way, to make clearer what they stand for.

When an atom (like foo) or a functor (bla/3) is used as an operand of an instruction - and in an exo-table (see later) - we actually mean the internal tagged representation of the atom or functor. Such a representation typically fits in a machine word.

We use $@x$ to denote an address labeled x.

2.1 The basic idea

We start from a predicate p/3 which consists of 4 facts and whose arguments are atoms:

```
p(a1,b1,c1).
p(a2,b2,c2).
p(a3,b3,c3).
p(a4,b4,c4).
```

We go through some steps before arriving at the final code we want to generate for it. For now, we ignore both indexing and instruction merging: they are orthogonal issues. The WAM compiles the above predicate to code such as can be seen in the left column below:

```
                                |       set_exo_pointer @t ------------------> a1 b1 c1
    try_me_else arity(3) @2     |       try_me_else_exo arity(3) @2            a2 b2 c2
    getatom A(1) a1             |       getatom_exo A(1)                       a3 b3 c3
    getatom A(2) b1             |       getatom_exo A(2)                       a4 b4 c4
    getatom A(3) c1             |       getatom_exo A(3)
    proceed                     |       proceed
@2: retry_me_else arity(3) @3   |   @2: retry_me_else_exo arity(3) @3
    getatom A(1) a2             |       getatom_exo A(1)
    getatom A(2) b2             |       getatom_exo A(2)
    getatom A(3) c2             |       getatom_exo A(3)
    proceed                     |       proceed
@3: retry_me_else arity(3) @4   |   @3: retry_me_else_exo arity(3) @4
    getatom A(1) a3             |       getatom_exo A(1)
    getatom A(2) b3             |       getatom_exo A(2)
    getatom A(3) c3             |       getatom_exo A(3)
    proceed                     |       proceed
@4: trust_me_else arity(3)      |   @4: trust_me_else_exo arity(3)
    getatom A(1) a4             |       getatom_exo A(1)
    getatom A(2) b4             |       getatom_exo A(2)
    getatom A(3) c4             |       getatom_exo A(3)
    proceed                     |       proceed
```

The code left above is very repetitive: we just scan each fact symbol by symbol. To show this idea clearly, we can re-arrange the code to separate walking through the arguments from the actual constants. The new code is shown in the right column. It relies on the following new WAM instructions:

- *set_exo_pointer* has one argument @t: it is a pointer to *a table* with the constants occurring in the facts. The table is nicely rectangular and compact. A WAM register *exo_pointer* is set to this pointer.
- *try_me_else_exo* acts as the *try_me_else* instruction in the WAM and also stores the current exo_pointer in its choice point.
- *retry_me_else_exo arity(N) @alt* fetches the exo_pointer from the choice point, adds `N` to it and stores that value in the choice point. The other WAM actions associated to *retry_me_else* are also performed.
- *trust_me_else_exo arity(N)* fetches the exo_pointer from the choice point and adds `N` to it. The other WAM actions associated to *trust_me_else* are also performed.
- *getatom_exo A(i)* fetches the i^{th} element from the current row in the exo-table (the exo_pointer points to that row now) and unifies it with Argument register i.

Note that the arity in the `(re)try/trust_me_else_exo` instruction is also the width of the exo table, so we could have denoted that operand as width(3). In fact, the width and the arity can be processed independently, and thus could be given separately.

At this point, have we gained anything? The amount of space needed for code+data has not decreased, and the instructions have a small extra overhead in fetching the constants from the table and manipulating the exo_pointer. On the other hand, the code for p/3 is now generic, i.e. it suffices to make the exo_pointer point to a different table - say

```
          u1 v1 w1
          u2 v2 w2
          u3 v3 w3
          u4 v4 w4
```

to see that all the code except for setting the exo_pointer can be reused for executing a different set of facts.

Clearly every fact consists of the same code: three `getatom_exo` instructions and a `proceed`. We exploit that by generating the following (final) code:

```
     try_exo arity(3) @a @e @t ----------> a1 b1 c1
 @a: keep_trying_exo arity(3)              a2 b2 c2
 @e: getatom_exo A(1)                      a3 b3 c3
     getatom_exo A(2)                      a4 b4 c4
     getatom_exo A(3)                      NULL
     proceed
```

We have added a NULL entry to the table as sentinel, so that we can check whether we have reached the end of the table. (We could have used a count of rows or entries; as we will see, Mercury uses the latter.)

The new instructions act as follows:

– *try_exo N @a @e @t* sets exo_pointer to point to the table @t, creates a choice point, saves exo_pointer in it and sets the alternative field to @a, and then transfers control to @e.
– *keep_trying_exo N* fetches exo_pointer from the choice point, adds N to it, and stores the resulting value in the choice point; the alternative in the choice point is not updated. If (exo_pointer+N) points to the NULL sentinel, the choice point is discarded. Either way, this instruction restores the argument registers.

These new instructions give us a major benefit of *exo-compilation*: they allow us to eliminate all but one copy of the fact handling code, which can mean a potentially huge memory saving, while preserving the genericity of the code.

2.2 Void Variables

In the context of ILP - but also in general in the database context - one is often confronted with wide facts that are queried by goals containing lots of void variables, i.e. fields in which one is not interested during a particular query. E.g. for a fact p/12, the query could be *?- p(bruce,willis,_,_,_,_,_,_,_,_,Salary,_)*.[1] Exo-compilation suggests dealing with void variables by generating a *specialised* predicate `p_1_2_11/3` whose code is:

[1] The point is that only the first, second and eleventh argument take part in the query, not that the first two arguments are instantiated or manifest.

```
    try_exo arity(3) @a @e @t(p/12) --------------> table for p/12
@a: keep_trying_select width(12) arity(3)
@e: getatom_exo A(1)
    getatom_exo A(2)
    getatom_exo_offset A(3), offset(11)
    proceed
```

The instruction *keep_trying_select* is a variant if *keep_trying_exo*, but where the width of the exo-table is different from the choice-point's arity.

getatom_exo_offset A(3), offset(11) unifies the third argument register with the atom to be found at offset 11 in the row currently pointed to by exo_pointer.

Replacing the original query by *?- p_1_2_11(bruce,willis.Salary)* eliminates all the unnecessary overhead of the void variables: initialisation, unification, trailing/untrailing, and storing in/restoring from the choice point.

In the context of ILP the above goal is typically generated dynamically and as part of a conjunction. In that case, before executing the conjunction, a void variable detection analysis can be performed and the appropriate transformation can be carried out. Since other analyses/transformations are already performed on such conjunctions (subsumption testing, once-transformation, ...) this seems reasonable - see for instance [9]. The code above must be generated at runtime. This is feasible as well, as other approaches have dealt with compiling (totally, partially, on the fly, and just in time) such code. See for instance [2].

2.3 Instruction merging and specialisation

The code for the facts can benefit from instruction merging: both in the original WAM and in the exo-compilation approach, we can easily collapse a sequence of get_atom instructions. We can even exploit the fact that the argument registers to be unified with table elements are consecutive and invent one new instruction like get5atoms_exo which needs no arguments at all, leading to a further reduction of the memory needed to represent the code and less argument fetching. Yap [10] performs such an instruction merging in ordinary WAM code: a sequence of up to 6 get_atom instructions from consecutive argument registers (and starting from 1) is compressed [8]. hProlog merges any sequence of up to 3 get_atom instructions, irrespective of the argument register they refer to. We have performed its analogue for the getatom_exo instruction.

Instruction specialisation is also applicable: hProlog and Yap (as many other systems) have specialised versions of the try/retry/trust_me_else instructions for several arities. hProlog does this specialisation up to arity 5, Yap up to arity 4. The same can be done for the analogous exo-instructions. We did not do this specialisation in our exo-compiler, because we are mainly interested in much higher arities.

3 Experiments in hProlog

Exo-compilation was implemented in hProlog as follows: for arities up to 15, there are predefined predicates (generated at startup) with code of the following form, which is for arity 5:

```
keep_trying_exo arity(5)            keep_trying_exo arity(5)
getatom_exo A(1)                    getatom3_exo A(1) A(2) A(3)
getatom_exo A(2)                    getatom2_exo A(4) A(5)
getatom_exo A(3)                    proceed
getatom_exo A(4)
getatom_exo A(5)
proceed
```

where the left half shows the code without instruction merging and the right with instruction merging. For larger arities, these would need to be generated on the fly.

This code acts as entry points for the code for an exo-compiled set of facts. This compilation is currently integrated in the compiler as follows: when the *prolog_flag* named *exo_compiling* is on, and the predicate to be compiled contains only facts with an atom for each argument, the predicate is exo-compiled. The compiler constructs the exo_table. An exo-predicate is compiled to one try_exo instruction which sets the exo_pointer and then transfers control to the appropriate pre-defined predicate.

Instruction merging of the exo-instructions was made into a command line hProlog option, so it is easy to run the benchmarks with and without instruction merging.

Generating code for the void specialisations is done by calling a new built-in predicate create_void_specialisation/3 which takes as arguments

- the exo-predicate - so that the exo-table can be retrieved
- the name/arity of the new predicate
- a description of which arguments of the original exo-predicate need to be selected

Because of the application we have in mind (dynamically generated queries in ILP), the user needs to call this built-in, but nothing prevents the compiler to do so as well. E.g., *?- create_void_specialisation(foo/5, gee/3, [2,4,5]).* generates the following code for gee/3:

```
@gee_3: try_exo arity(3) @a @e @t(foo/5)
    @a: keep_trying_select width(5) arity(3)
    @e: getatom3_exo_offset A(1) offset(2) A(2) offset(4) A(3) offset(5)
        proceed
```

when instruction merging is on.

We use $@t(foo/5)$ to denote the address of the exo-table for foo/5: it is known at load/link time.

The timings in Tables 1, 2 and 3 were obtained on a PC with a 1.8 GHz Pentium 4 CPU running Debian. Times are given in milliseconds. We used hProlog 2.7, Yap 5.1.1 and SICStus 3.12.0.

We start with an experiment in which a set of predicates p/n, n=1..15, each with 150 facts and with all atom arguments is called with free, unshared arguments. We do this in Yap and in hProlog, both in a version with and without instruction merging. The table also contains the timings for SICStus. In this way, one gets a better view on the performance. Table 1 shows that without

Arity	Yap merging	Yap no merging	hProlog merging	hProlog no merging	SICStus merging
1	664	676	552	568	2720
3	948	2032	1464	1993	3480
5	1620	2905	1752	2596	5490
7	3040	3604	3053	3324	6560
9	4284	4812	3904	4373	8230
11	5085	5592	4613	4784	10130
13	5884	6248	5024	5648	12580
15	6304	6793	5761	6457	12860

Table 1. Performance on plain WAM code

instruction merging Yap and hProlog have close performance. With instruction merging, the figures show that up to arity 6, the Yap compression is superior, while the hProlog merging is better for larger arities. SICStus performs significantly worse on all arities. As far as we know, SICStus merges two consecutive get_atom instructions, as well as a (get_atom proceed) combination.

Even though the general trend (larger timings with larger arities) in Table 1 is clear, the columns with merging show some anomalies for which we have no explanation. Later tables exhibit similar anomalies.

Arity	hProlog no merging	hProlog exo no merging	hProlog merging	hProlog exo merging
1	568	1412	552	1476
3	1993	2372	1464	2056
5	2596	2868	1752	2072
7	3324	3433	3053	3040
9	4373	4816	3904	3744
11	4784	4889	4613	4452
13	5648	5656	5024	4908
15	6457	6364	5761	5485

Table 2. hProlog in plain WAM mode and in exo mode

Table 2 shows that for hProlog exo-compilation starts paying off from arity 7 (actually 6 with the full data) with merging, but only from arity 13 without merging. There is indeed an overhead in the getatom_exo instruction, probably because of the lack of registers: there is no spare register for the pointer to the exo-table.

In Table 3, we show hProlog on the same set of benchmarks, but with a query with only three non-void arguments. The *exo void* columns take advantage of that in the way described in Section 2.2, while the *plain exo* columns follow the plain exo-compilation schema. The difference is clear: the left column is close to constant (as it should be), while the right column's runtime increases linearly with the arity. It is nice to see that the break even point is close to three. All the code was generated beforehand, i.e. not dynamically as would be needed in an ILP context where the queries are not known in advance.

Arity	hProlog exo void no merging	hProlog plain exo no merging	hProlog exo void merging	hProlog plain exo merging
3	2312	2328	1540	1564
5	2288	2964	1492	1820
7	2284	3672	1492	2805
9	2256	4284	1565	3760
11	2272	4725	1504	4605
13	2353	5528	1580	5188
15	2384	6156	1688	5724

Table 3. Optimising queries with void variables

4 Generalising exo-compilation

Exo-compilation as described above exploits a specific form of regularity of code. The generalisation to other atomic types besides atoms is straightforward. An example shows how facts with structured (ground) terms can be dealt with:

```
p(foo(a,b(c))).      p(gee(x,y(z))).
```

These facts have enough in common to treat them by exo-compilation, especially if there are many thousands of them. The generalisation of the getatom_exo instruction to functors is

```
get_struct_exo_offset A(i), offset(j)
```

with obvious meaning. Other instructions need an exo version as well.

The above p/1 would be translated to

```
    try_exo arity(1) @a @e @t ----------------> foo/2 a b/1 c
@a: keep_trying_select arity(1) width(4)        gee/2 x y/1 z
@e: get_struct_exo_offset A(1), offset(1)       NULL
    unify_atom_exo_offset offset(2)
    unify_struct_exo_offset offset(3)
    unify_atom_exo_offset offset(4)
    proceed
```

The meaning of the instructions should be clear, and dealing with void variables in a call to such non-flat facts is clearly feasible. The next steps in generalising exo-compilations are: allow variables, allow lists (of different length), allow clauses with similar bodies, and allow arbitrary ground terms; the first few of those are described in more detail in [4]. The same reference also contains some preliminary ideas about combining indexing with exo-compilation in the context of the WAM.

5 How Mercury handles large predicates

5.1 Large predicates without indexing

In Mercury, every predicate (or function) has one or more modes. Each mode specifies, for each argument, whether that argument is input or output in that mode, and also specifies a *determinism*, which specifies upper and lower bounds on the number of solutions expected in that mode. For example, *append(in,in,out)* has determinism *det*, meaning calls to *append* in that mode will have exactly one solution, while *append(out,out,in)* has determinism *multi*, meaning calls to *append* in that mode will have one or more solutions.

The Mercury compiler generates a separate piece of code for each mode: each mode of a predicate is called a *procedure*. The easiest modes to generate good code for when the predicate body is defined by a large set of facts are the modes in which all arguments are output (and whose determinism is therefore *multi*). The C code generated by the compiler for this mode of the predicate p/3 with four facts from Section 2.1, will look something like this, after discarding some irrelevant details:

```
p_3_0:
    mkframe(1, local_label_2);
    temp1 = &common_table_0[0];
    framevar1 = 3;
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed();

local_label_2:
    r4 = framevar1;
    if (r4 >= 9) goto local_label_3;
    framevar1 = framevar1 + 3;
    temp1 = &common_table_0[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed();

local_label_3:
    temp1 = &common_table_0[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed_discard();
```

The first block of the code handles the first solution, the last block of code handles the last solution, and the middle block handles all the others in between, rather like a try/retry/trust chain in the WAM in which all retry's are collapsed into one block of code.

The *mkframe* macro allocates a frame on the nondet stack; this frame functions as a combination choice point and environment. The environment part holds one slot (containing the equivalent of the WAM's exo-pointer) and the backtrack point is *local_label_2*. The common_table_0 is Mercury's internal name for what was named exo table in Section 2.1, and it stores the data which in this case is the 12 atoms. The Mercury calling convention for *multi* procedures requires argument n to be returned in register r_n, so the assignments to r_1, r_2 and r_3 pick up the value of each argument of the first solution from a table containing all the solutions. The *succeed* macro then returns to the return address recorded in the stack frame by the *mkframe* macro.

After backtracking causes execution to reach *local_label_2*, the code checks whether the next solution is the last one. If not, the stack frame is updated to show that this solution was returned but otherwise it is left intact. If the last solution is reached, execution branches to code that uses the *succeed_discard* variant of the *succeed* macro: it discards the stack frame after picking up the return address from it.

Unlike most Prolog implementations, the Mercury compiler does not convert procedure bodies to disjunctive normal form. The goal form representing a table facts (a disjunction in which each disjunct is a conjunction of unifications that create static terms) may therefore appear inside other goals, e.g. an if-then-else, either in the program as written or after inlining. In such cases, the generated code is very similar code to the code above, the main difference being that it won't have to create a stack frame (the surrounding code having already created one), and the mechanism used to record the next alternative will be slightly different [3]. The reasons why the names of the tables don't refer to procedure names is that a single procedure may refer to more than one of these tables, while a table may be referred to from more than one procedure if the predicate whose data is contained in the table is inlined at more than once call site. This works because Mercury does not support dynamic predicates.

In Mercury, the exo tables can contain any type of ground term. For ground terms, there is never any need to copy them: the exo table contains the term exactly as it would be laid out in the heap, so all what needs to be returned as a result is a pointer to the term. The same technique can be used in the WAM, but would require minor changes in the garbage collector (which Mercury's use of a conservative collector renders moot). In fact, ECLiPSe has implemented static terms at least since 1990.

5.2 Large predicates with indexing

Mercury uses indexing quite aggressively, i.e., not just on arguments in the head of a predicate, but also in explicit disjunctions. In the past, the implementation of indexing was based on using a table (e.g. a hash table) to map the value of the switched-on variable to a code address, with the code address giving the start of the code for handling a particular switch arm. Remember that Mercury generates C code. This indexing schema works fine for small and even medium-sized code. But when the size of the predicate increases, and as a consequence the size of the generated C code also increases, this becomes problematic: it exposes quadratic behaviour in the C optimiser, as well as limits in the Mercury compiler's own low level optimiser. Exo-compilation avoids the problem of generated code size explosion: only the size of the generated C data arrays increases.

Indexing kicks in when enough input is available to reduce the alternatives that can return answers. In the context of exo-compilation, and for simplicity of explanation, we focus on a predicate consisting of facts, and for which there is one input argument that is an integer. We assume also that the range of this integer is dense in some range starting from 0, so that the input can be used as in index in an array without further manipulation.[2] We work by example, and distinguish three cases, namely that the procedure is (1) det, (2) semidet and (3) nondet or multi.

A det procedure: This corresponds to a predicate like

```
:- pred p(int:in, term1:out, term2:out).
p(0,a0,b0).
p(1,a1,b1).
p(2,a2,b2).
p(3,a3,b3).
p(4,a4,b4).
```

It is clear that a two dimensional table of the form

```
a0 b0
a1 b1
a2 b2
a3 b3
a4 b4
```

can be addressed directly with the input argument to retrieve the necessary output. The generated code's size is independent of the number of facts: this will be true for the other cases as well.

A semidet procedure: This occurs when there is no solution for one or more input values, as in

```
:- pred p(int:in, term1:out, term2:out).
p(0,a0,b0).
p(2,a2,b2).
p(3,a3,b3).
p(4,a4,b4).
```

The compiler generates a bit vector, which at place i indicates whether the corresponding input argument corresponds to a solution. A successful test causes a jump to the same code as in the det case; an unsuccessful test causes failure.

[2] Mercury uses appropriate generalisations of these conditions.

A nondet or multi procedure: This occurs for instance in the following predicate:

```
:- pred p(int:in, term1:out, term2:out).
p(0,a0,b0).
p(0,c0,d0).
p(0,e0,f0).
p(0,g0,h0).
p(2,a2,b2).
p(3,a3,b3).
p(4,a4,b4).
p(4,c4,d4).
p(4,e4,e4).
```

The compiler generates two tables, the first one of which is addressed by using the input argument as an index. The table contains $n+2$ columns if there are n output arguments - in our example $n=2$ and the table is shown in Figure 1. The first entry in each column contains an indication of whether the input value

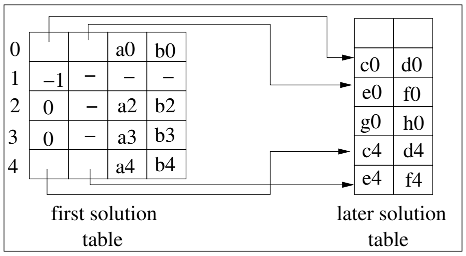

Fig. 1. First and later solution table for a nondet procedure

corresponds to a switch arm with no solution (-1), exactly one solution (0) or more than one solution (any value above 0). Code tests this value and branches to the appropriate continuation, which consists in failure in the first case.

If the selected switch arm has one solution, the indexing code will pull the values of the output variables out of the last n columns of the selected row, put them where the code after the switch (which may be the procedure epilogue) expects them, and then jump to that code.

If the selected switch arm has more than solution, the indexing code will also pull the values of the output variables out of the last n columns and put them where they are expected, but before jumping to the code after the switch, it will set up the return of the later solutions on backtracking.

Part of this involves saving the values of the first and second columns in the current stack frame (as *framevar1* and *framevar2* respectively in the example code below). Each switch arm that has more than one solution stores all those solutions except the first in a contiguous region of the second table. The first column of the first-solution table row points to (contains the index of) the start of the first of these solutions in the later-solution table, while the second column points to the start of the last of these solutions.

The other part is directing execution to code that uses these saved values. How this is done depends on the context. The required code may be as expensive as pushing a temporary frame (effectively a mini choicepoint) on the nondet stack, as cheap as simply updating the backtrack code pointer of the current nondet stack frame, or something in between (see [3] for more info, through some details have changed since then).

If the switch is the entirety of a procedure body, the code executed on backtracking will look like the following, which is a generalised version of the code at local labels 2 and 3 above (*common_table_1* refers to the later solutions table).

```
local_label_5:
    r4 = framevar1;
    if (r4 >= framevar2) goto local_label_6;
    framevar1 = framevar1 + 3;
    temp1 = &common_table_1[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed();

local_label_6:
    temp1 = &common_table_1[r4];
    r1 = temp1[0];
    r2 = temp1[1];
    r3 = temp1[2];
    succeed_discard();
```

5.3 Semantic analysis of large predicates

Code generation, is not the only area in which large predicates pose challenges. Many semantic analysis algorithms (which Prolog does not need but Mercury does) have behavior that is quadratic (or worse) in the number of clauses, in the sizes of terms, or both. The Mercury compiler certainly contained many such algorithms. We had to find and try to fix them one by one (unfortunately, some fixes require more work than we have funding for). The most ubiquitous problem was algorithms that added something to the end of a list after each disjunct; doing $O(n)$ work at disjunct n yields a quadratic algorithm. We changed these to instead return a list of those somethings (which differ from analysis to analysis), and then used code patterned after balanced mergesort to generate the final output.

6 Discussion, related work, and conclusion

Large datasets occur in a wide range of applications. Our experience includes

- knowledge bases processed by an inductive logic programming system;
- tables generated by scanner and parser generators; and
- large databases of structured patterns to look for in an input stream.

There are doubtless many more. While each one of these may be only a niche, put together they are significant enough to be worthy of attention.

Most current logic programming systems do not have any support for large datasets. The reason for this is mostly historical: such systems have traditionally been called deductive databases. As a result, the handling of large datasets by most logic programming systems has left a lot to be desired. Each of us (the authors) bumped into these problems in our daily work, and decided to fix the situation.

As it happened, though we started working on different systems (Yap, hProlog and Mercury) and got our driving motivations from different application areas, we ended up with techniques that are quite similar. [3] Thought both Prolog and Mercury compilers traditionally generate code for each part of the program, we (Santos Costa and Somogyi) independently decided that the efficient implementation of large tables of facts requires breaking this rule, and generating generic table lookup code instead, with all the information specific to the program confined to the tables. Although this is a fairly standard programming technique (being used in databases, parsers and in many other programs), it *is* unusual to find it in code generated automatically by a compiler.

The most obvious benefit of this approach, which we call exo-compilation, is of course the large reduction in the amount of code required. However, it also yields speedups, which is a more important benefit, though it is also much less obvious. For some people, it may even be counter-intuitive, because exo-compilation actually *increases* the number of memory accesses, due to accesses to data in tables requiring an extra level of indirection. Though this extra indirection reduces locality a bit, eliminating redundant copies of WAM instructions and putting the arguments of each fact closer together improves locality by far more, and it is this latter effect that dominates in our applications. Our experiments show performance improvements, and detailed cache simulations (performed with the cachegrind option of `valgrind-3.2.0-Debian`) have revealed that D1 misses and L2 references can drop by a factor of 5, even though overall I and D references increase by about 6%: these figures were obtained for running the totality of the benchmarks in Tables 1 and 2.

Exo-compilation also brings other benefits. In the ILP context often each example is represented by some predicates each of which consists of a few small facts only. Treating an individual example by exo-compilation might seem senseless. However, there are often too many examples to keep them all in memory and ILP tools need to switch between examples frequently. For that reason, hipP (an ILP dedicated cousin of hProlog) has an intricate module for switching between examples, where each example is pre-compiled fully to WAM code. Applying exo-compilation to the examples as a whole, would result in generic code that can be used for each example and switching between examples would consist of switching between exo-tables. Since these are smaller than fully compiled code, it allows more examples simultaneously in memory, it reduces the amount of memory traffic when an example needs to be (re)loaded, and as a result it increases performance and the size of the datasets that can be handled by

[3] Our different choices were partly dictated by differences in the languages (e.g. Prolog supports dynamic predicates and dynamic loading of predicates while Mercury doesn't, and likewise for variables in facts), and partly motivated by differences in the existing technology bases (the Mercury abstract machine and the WAM are quite dissimilar, and the WAM is usually interpreted while Mercury compiles to C).

the system. These considerations were in fact a major motivation for exploring exo-compilation in the context of Prolog.

Exo-compilation also serves as the basis for further optimisations. For example, dynamic indexing fits in nicely with exo-compilation, and it would be more difficult to implement without separating the operands from the code. Or consider the optimisation we presented in Section 2.2 which was intended to remove the overhead of void variables in queries. This overhead can be reduced by the Vienna Abstract Machine (see [7], or by a tagging schema that caters for void variables (as in Beer [1] which caters for uninitialised variables), but these techniques still perform actions that are linear in the number of void variables, while our technique does not.

We have just started using exo-compilation in live applications such as ILP. The future will show how well the idea actually works in practice, but the initial indications are quite promising.

Acknowledgements

We thank Research Foundation-Flanders (FWO-Vlaanderen) for supporting Bart Demoen.

References

1. J. Beer. The occur-check problem revisited. *J. Log. Program.*, 5(3):243–261, 1988.
2. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
3. T. Conway, F. Henderson, and Z. Somogyi. Code generation for mercury. In *ILPS*, pages 242–256, 1995.
4. V. Costa, B. Demoen, and P.-L. Nguyen. Big facts, void variables, the WAM and exo-compilation. Report CW 486, Department of Computer Science, K.U.Leuven, Leuven, Belgium, April 2007. URL: http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW486.abs.html.
5. L. Damas, V. Santos Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1989.
6. B. Demoen. hProlog. http://www.cs.kuleuven.be/ bmd/hProlog/.
7. A. Krall. The Vienna Abstract Machine. *Journal of Logic Programming*, 29(1-3):85–106, 1996.
8. H. Nässén, M. Carlsson, and K. Sagonas. Instruction merging and specialization in the SICStus Prolog virtual machine. In *Principles and Practice of Declarative Programming (PPDP01)*, 2001.
9. V. Santos Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. Van Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 4(Aug):465–491, August 2003.
10. V. Santos Costa. *Prolog Performance on Larger Datasets*, In *PADL07*, 2007.