

# Big facts, void variables, the WAM and exo-compilation

*Vitor Santos Costa*

*Bart Demoen*

*Phuong-Lan Nguyen*

*Report CW486, April 2007*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Big facts, void variables, the WAM and exo-compilation

*Vitor Santos Costa*

*Bart Demoen*

*Phuong-Lan Nguyen*

*Report CW 486, April 2007*

Department of Computer Science, K.U.Leuven

## **Abstract**

Prolog systems need to deal with large sets of wide facts, e.g. in the context of ILP. These facts are often regular in the sense that all arguments are atoms. We investigate a compilation schema (which we name exo-compilation) of such facts which reduces the memory needed for the code to about one third of the normal WAM compilation schema without undue slowdown. As a bonus, we get a significantly better treatment of queries with lots of void variables: this also occurs frequently in the ILP context. We discuss generalisations of the basic idea, we present an empirical evaluation in hProlog and we show how (dynamic) indexing can be integrated in the approach as well. All these issues are explained in the context of an emulator, but a compiler (to C or native code) can benefit from the idea as well.

# Big facts, void variables, the WAM and exo-compilation

Vitor Santos Costa<sup>#</sup>, Bart Demoen<sup>&</sup>, and Phuong-Lan Nguyen<sup>@</sup>

<sup>#</sup> COPPE, Universidade Federal do Rio de Janeiro, Brasil  
vitor@cos.ufrj.br

<sup>&</sup> Department of Computer Science, K.U.Leuven, Belgium  
bmd@cs.kuleuven.be

<sup>@</sup> Institut de Mathématiques Appliquées, UCO, Angers, France  
nguyen@ima.uco.fr

**Abstract.** Prolog systems need to deal with large sets of wide facts, e.g. in the context of ILP. These facts are often regular in the sense that all arguments are atoms. We investigate a compilation schema (which we name exo-compilation) of such facts which reduces the memory needed for the code to about one third of the normal WAM compilation schema without undue slowdown. As a bonus, we get a significantly better treatment of queries with lots of void variables: this also occurs frequently in the ILP context. We discuss generalisations of the basic idea, we present an empirical evaluation in hProlog and we show how (dynamic) indexing can be integrated in the approach as well. All these issues are explained in the context of an emulator, but a compiler (to C or native code) can benefit from the idea as well.

## 1 Introduction

In some applications, Prolog is faced with predicates consisting of a large set of wide facts (i.e. the arity of the predicate is large): this is the case in Machine Learning based on Inductive Logic Programming. Most often these facts are *typed and moded nicely*, i.e. the facts are ground and the arguments are all atoms. The WAM does not exploit this situation. Such predicates are often called with one argument instantiated (typically a key of the relation, but that is not important at this moment), and all other arguments are void<sup>1</sup> variables except for a few arguments which are either also instantiated or free variables which are not void. Some WAM instructions deal with void variables. However, both at the call and the callee side an amount of work linear in the number of void variables is performed, and it would be better to avoid that work altogether.

We show in Section 2 how both issues (wide facts and void variables) can be improved in the WAM by a technique where data and code are separated and which was named *exo-compilation* by the first author. Exo-compilation was

---

<sup>1</sup> This is WAM speak for saying that syntactically the same variable doesn't occur anywhere else in the query.

implemented in hProlog (see [3]): Section 3 contains an experimental evaluation thereof. Section 4 extends the initial exo-idea to more general facts, i.e. non-ground and structured facts, and even to general clauses. Section 6 discusses indexing issues, i.e. how to exploit the exo-idea in indexing code as well. Section 8 discusses related work and concludes.

## 2 Exo-compilation

We first introduce some conventions that come in handy 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 a similar way, so that it is more clear 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/5` which consists of 3 facts and whose arguments are atoms:

```
p(a1,b1,c1,d1,e1).
p(a2,b2,c2,d2,e2).
p(a3,b3,c3,d3,e3).
```

We go through some steps before arriving at the final code we want to generate for it. At this moment, we do not consider indexing; indexing is an orthogonal issue. The WAM compiles the above predicate to code such as:

```
try_me_else arity(5) @2
getatom A(1) a1
getatom A(2) b1
getatom A(3) c1
getatom A(4) d1
getatom A(5) e1
proceed
@2: retry_me_else arity(5) @3
getatom A(1) a2
```

```

    getatom A(2) b2
    getatom A(3) c2
    getatom A(4) d2
    getatom A(5) e2
    proceed
@3: trust_me_else arity(5)
    getatom A(1) a3
    getatom A(2) b3
    getatom A(3) c3
    getatom A(4) d3
    getatom A(5) e3
    proceed

```

Instruction merging would certainly improve that, but it is also an orthogonal issue. The above code is very repetitive and we can re-arrange it as follows:

```

    set_exo_pointer @t -----> a1 b1 c1 d1 e1
    try_me_else_exo arity(5) @2      a2 b2 c2 d2 e2
    getatom_exo A(1)                 a3 b3 c3 d3 e3
    getatom_exo A(2)
    getatom_exo A(3)
    getatom_exo A(4)
    getatom_exo A(5)
    proceed
@2: retry_me_else_exo arity(5) @3
    getatom_exo A(1)
    getatom_exo A(2)
    getatom_exo A(3)
    getatom_exo A(4)
    getatom_exo A(5)
    proceed
@3: trust_me_else_exo arity(5)
    getatom_exo A(1)
    getatom_exo A(2)
    getatom_exo A(3)
    getatom_exo A(4)
    getatom_exo A(5)
    proceed

```

First the new instructions are explained:

- *set\_exo\_pointer* has one argument @t: it is a pointer to a table with the atoms occurring in the facts. The table is nicely rectangular. A global variable *exo\_pointer* is set to this pointer.
- *try\_me\_else\_exo* acts like the *try\_me\_else* instruction in the WAM and also stores the current *exo\_pointer* in its choicepoint.
- *retry\_me\_else\_exo arity(N) @alt* fetches the *exo\_pointer* from the choicepoint, adds N to it and stores that value in the choicepoint. The other WAM actions associated to *retry\_me\_else* are also performed.

- *trust\_me\_else\_exo arity(N)* fetches the *exo\_pointer* from the choicepoint 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(5)*. Later, the width and the arity can become independent and are given separately.

At this point, we have not gained much: the amount of space needed (code+data) is probably higher, and the instructions have a little extra overhead in fetching the atoms from the table and manipulating the *exo\_pointer*. On the other hand, the code for *p/5* is now generic, i.e. it suffices to make the *exo\_pointer* point to a different table - say

```

u1 v1 w1 x1 y1
u2 v2 w2 x2 y2
u3 v3 w3 x3 y3

```

to see that all the code except for setting the *exo\_pointer* can be used for executing a different set of facts.

Also, it is clear that every fact consists of the same code: 5 *getatom\_exo* instructions and a *proceed*. We exploit that by generating the following (final) code:

```

try_exo arity(5) @a @e @t -----> a1 b1 c1 d1 e1
@a: keep_trying_exo arity(5)          a2 b2 c2 d2 e2
@e: getatom_exo A(1)                  a3 b3 c3 d3 e3
    getatom_exo A(2)                  NULL
    getatom_exo A(3)
    getatom_exo A(4)
    getatom_exo A(5)
    proceed

```

We have added to the table a sentinel NULL so that we can check whether we have reached the end of the table - other mechanisms can be envisaged.

The instructions acts as follows:

- *try\_exo N @a @e @t* performs the actions:
  - *exo\_pointer* is set to point to the table @t
  - a choicepoint is created: *exo\_pointer* is saved in it and its alternative is set to @a; it ends by transferring control to @e
- *keep\_trying\_exo N* fetches *exo\_pointer* from the choicepoint, adds N to it and stores that value in the choicepoint; the alternative in the choicepoint is not updated. If at this moment (*exo\_pointer+N*) points to the NULL sentinel, the choicepoint is discarded; this instruction restores the argument registers

Now we have made potentially huge memory savings because all the facts of p/5 have collapsed to the code for one fact - and of course the genericity of the code is still present. In Section 3, we show how big this memory savings can be in practice.

## 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,_)`.<sup>2</sup> Exo-compilation suggests dealing with void variables by generating a new predicate p\_1.2\_11/3 whose code is

```

try_exo arity(3) @a @e @t(p/12) -----> table for p/12
@a: keep_trying_exo2 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_exo2` is a variant of `keep_trying_exo` in which the width of the exo-table is different from the arity of the choicepoint.

`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`.

The original query is replaced by `?- p_1.2_11(bruce,willis,Salary)`. The unnecessary overhead of the void variables (initialization, unification, [un]trailing and [re]storing in/from the choicepoint) is all avoided.

There remains a challenge in the context of ILP: the above goal would typically have been generated dynamically and as part of a conjunction. In that case, before executing the query, some small analysis should be performed (to detect the void variables) and the appropriate transformation can be done. Since other analyses/transformations are already performed on such queries (subsumption testing, once-transformation ...) this seems reasonable - see for instance [6]. The code above must then be generated at runtime. This is feasible, 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 specialization

It is clear that instruction merging can be applied: both in the original WAM in the exo-compilation approach, we can collapse easily 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 instruction like

<sup>2</sup> 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.

```
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 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. 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. We have also shortly played with an instruction `getatoms_exo N`, where `N` is the number of `getatom_exo` instructions (from consecutive argument registers starting at 1) to be compressed, but it was not a success performance wise.

Instruction specialization is also applicable: hProlog and Yap (as many other systems) have specialized versions of the `try/retry/trust-me_else` instructions for several arities. hProlog does this specialization up to arity 5, Yap up to arity 4. The same can be done for the analogous `exo`-instructions. We have not done this specialization 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 (and at startup generated) predicates with code (for arity equal to 5 for instance) of the form:

<pre>keep_trying_exo arity(5) getatom_exo A(1) getatom_exo A(2) getatom_exo A(3) getatom_exo A(4) getatom_exo A(5) proceed</pre>	<pre>keep_trying_exo arity(5) getatom3_exo A(1) A(2) A(3) getatom2_exo A(4) A(5) proceed</pre>
--	--

where the left half shows the code without instruction merging and the right with instruction merging.

This code acts as entry points for the code for an `exo`-compiled set of facts. This compilation is not yet integrated in the general compiler: one needs to call the `exo`-compiler separately on a file with the facts. 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 specializations is done by calling a new built-in predicate `create_void_specialization/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

E.g., `?- create_void_specialization(foo/5, gee/3, [2,4,5]).` generates the following code for `gee/3`:

```
try_exo arity(3) @a @e @t(foo/5)
@a: keep_trying_exo2 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 Pentium, 1.8GHz, and they 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		hProlog		SICStus
	merging	no merging	merging	no merging	
1	664	676	552	568	2720
2	664	1596	1880	1980	2970
3	948	2032	1464	1993	3480
4	1188	2352	2216	2324	4610
5	1620	2905	1752	2596	5490
6	1856	3392	2756	2896	6430
7	3040	3604	3053	3324	6560
8	2801	3904	3272	3624	7660
9	4284	4812	3904	4373	8230
10	4800	5541	4140	4676	9540
11	5085	5592	4613	4784	10130
12	5068	5685	4552	5325	11690
13	5884	6248	5024	5648	12580
14	6001	6428	5372	5916	13020
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 - we do not know at this point whether SICStus performs some instruction merging for these benchmarks.

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 exo merging	hProlog exo no merging	hProlog merging	hProlog no merging
1	1476	1412	552	568
2	1972	2312	1880	1980
3	2056	2372	1464	1993
4	2396	2552	2216	2324
5	2072	2868	1752	2596
6	2644	3168	2756	2896
7	3040	3433	3053	3324
8	3081	4020	3272	3624
9	3744	4816	3904	4373
10	4000	4540	4140	4676
11	4452	4889	4613	4784
12	4597	5160	4552	5325
13	4908	5656	5024	5648
14	5072	5777	5372	5916
15	5485	6364	5761	6457

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

Table 2 shows that for hProlog exo-compilation starts paying off from arity 6 (with merging) and from arity 11 (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 which has (Arity-3) void variables. 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 Yap would probably do.

Clearly, if facts are wide, this exo void-optimization pays off.

## 4 What about non-flat facts ?

Caveat: we have not implemented the ideas described in the following subsections. Read them as anything between wild speculation and an informed guess.

Arity	hProlog exo void	hProlog plain exo	hProlog exo void	hProlog plain exo
	no merging	no merging	merging	merging
3	2312	2328	1540	1564
4	2292	2792	1492	1852
5	2288	2964	1492	1820
6	2337	3260	1584	2700
7	2284	3672	1492	2805
8	2312	3973	1552	3388
9	2256	4284	1565	3760
10	2356	4540	1560	4272
11	2272	4725	1504	4605
12	2308	4920	1508	4484
13	2353	5528	1580	5188
14	2312	5825	1572	5253
15	2384	6156	1688	5724

Table 3. Optimizing queries with void variables

#### 4.1 Non-flat ground facts

It is possible that the information is presented in a form that does not lend itself directly to exo-compilation as described above. For instance, facts could be of the form<sup>3</sup>

```
car(0, class("fill up only"),
  attributes(["state/province" - s("WA"), "tag" - s("IPD050"),
    "year" - i(2004), "month" - i(12), "day" - i(26),
    "hour" - i(15), "minute" - i(42),
    "station" - s("mobil"), "fill-ups" - i(4),
    "record history (days)" - i(27), "weekdays" - i(3),
    "weekends" - i(1), "preferred day" - s("tuesday"),
    "day preference %" - i(25),
    "mean time of visit (minutes after midnight)" - i(195),
    "standard deviation from mean" - i(293),
    "different locations" - i(2),
    "preferred station" - s("mobil"),
    "preferred station %" - i(50),
    "% pay cashier (instead of at pump)" - i(25),
    "payment method" - s("credit"),
    "preferred pump" - s("17LG"), "preferred pump %" - i(25)
  ])).
```

Such facts show also a great deal of similarity and exploiting it is almost mandatory: first of all, unification factoring can be done. Some systems (XSB is one of them) perform this. If all heads have as arguments a structure (or atom) of the form `foo/n` (with `n` possibly 0) then this common part is factored out. After

<sup>3</sup> This fact comes from a large data set with `car/3` facts, we obtained through Zoltan Somogyi from Douglas M. Auclair - thanks for permission to reproduce it here.

this is done, one can end up with flat, regular facts, or possibly the difference is just in the name of some functors - and not in their arity. E.g. the following two facts are similar enough for our purposes:

```
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 `get_atom_atom` 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_exo2 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. It is also clear how to deal with void variables in a call to non-flat facts.

#### 4.2 General facts

The final step is towards dealing in the exo-style with general facts: we now allow also variables, as long as two different facts are *similar*. It is time to define similarity more precisely:

- let the exo-generalisation of a term T be the term S which one obtains by replacing every function symbol (any arity) by *foo* except the list constructor
- two facts are exo-similar if the facts obtained by exo-generalising their arguments, are variants (i.e. variable renamed terms)

The compiled code of each exo-generalised fact from a set of exo-similar facts is exactly the same, and we use it to construct the body of one exo-clause, where the instructions have been replaced by the appropriate exo-variant with offset. As an example, consider the predicate

```
p(a(f, [A|p], A)).
p(b(g, [B|q], B)).
p(c(h, [C|r], C)).
p(d(i, [D|s], D)).
p(e(j, [E|t], E)).
```

This would be compiled to

```
try_exo arity(1) @a @e @t -----> a/3 f p
@a: keep_trying_exo arity(1)          b/3 g q
@e: get_struct_exo_offset A(1) offset(1) c/3 h r
    unify_atom_exo_offset offset(2)    d/3 i s
    uni_tvar A(1)                       e/3 j t
    uni_tvar A(2)
    getlist A(1)
    unitval A(2)
    unify_atom_exo_offset offset(3)
    proceed
```

The list instructions don't need an exo variant: an exo instruction always refers to an exo-table for finding the operand on which to operate. The list instructions have this operand encoded in themselves.

### 4.3 General clauses

By lifting the definition of exo-similar facts to clauses in an appropriate way, one can clearly also apply the idea of exo-compilation to predicates with exo-similar clauses. It is wild speculation whether this is worthwhile and whether this is encountered in practice.

## 5 Relaxing exo-similarity for lists

We consider two lists exo-similar if they are both ground and have atoms as elements. For example, [a,b,c], [x] and [] are all exo-similar to each other. Assume now two facts for p/2:

```
p([], [e,f]).
p([w,x,y,z], []).
```

These facts are exo-similar according to our relaxed definition, and it is possible to apply the exo-idea as follows: we make an exo-table with the following items

```
5 0 2 e f
7 4 w x y z 0
```

The first number in each line indicates the number of items in that line. The indication **4 w x y z** means: the list to unify with has 4 elements - and then the elements follow. **0** means a zero length list.

The accompanying code is

```

try_exo_len arity(2) @a @e @t -----> 5 0 2 e f
@a: keep_trying_exo_len arity(2)      7 4 w x y z 0
@e: get_exo_closed_list_advance A(1)   NULL
    get_exo_closed_list_advance A(2)
    proceed

```

## 6 Exo-indexing

We have described exo-code for predicates which were supposed to be called with free arguments. In practice, this is seldom true and optimizing the case where some arguments are instantiated is important. In the WAM this happens with indexing instructions. In the exo-approach, the situation is not very different, but the exo context allows for some nice extras, in particular, the dynamic generation of indexing code as in Yap (from version 5.1.1 on) can be performed more easily, it can be combined with the optimization for void variables and it allows skipping arguments which indexing has already determined. We focus on flat ground facts: it will be clear how things generalize to the other situations. We also focus on dynamic generation of indexing code with the void optimization.

The example has to be small, but keep in mind that the exo-principle is meant to be applied to large databases.

Suppose we have a set of facts p/3 like

```

p(a1,a,a2). % 1
p(b1,b,b2). % 2
p(b3,b,b4). % 3
p(a3,a,a4). % 4

```

and there is a call with as first argument a void variable, the second argument is instantiated and the third argument is free (e.g. `?- p(_,b,X).`), then the following actions deal with this call:

1. divide the facts in groups with the same second argument (the argument we are indexing on) and put their numbers in different arrays - for the example, we end up with arrays `@a={1,4}` and `@b={2,3}`
2. generate the following code:

```

select arg=2 a=@a, b=@b
move_areg A(3) A(1)
try_some_exo arity(1) @x @y @t -----> a1 a a2
@x: keep_trying_some_exo arity(1)      b1 b b2
@y: getatom_exo_offset A(1) offset(3)  b3 b b4
proceed                                a3 a a4
                                         NULL

```

First note that (1) is easy because we have the *exo*-table: in the approach of [5], the atoms are arguments of the instructions and one must interpret these instructions.

The instructions that need some explanation are:

- *select arg=2 a=@a, b=@b*: if argument 2 is the atom *a*, set a global variable *to\_try* to *@a*; otherwise if argument 2 is the atom *b* to *@b*; otherwise fail
- *try\_some\_exo 1, @x, @y, @t*: make a choicepoint for one argument only, but also save the global variable *to\_try* in a slot with the same name; also save *@t* as we did in the *try\_exo* instruction; set the alternative slot to *@x* and jump to *@y*
- *keep\_trying\_some\_exo 1*: restore one argument from the choicepoint; compute the next value of the *exo\_pointer* using the *to\_try* slot; advance the *to\_try* pointer; discard choicepoint if appropriate

Note that we have also avoided generating code for the incoming second argument which the indexing instructions have already decided about that it will match any fact that is tried: this is an extra benefit of dynamically generating the indexing code and the specialized code for the selected facts. It amounts to a form of dynamic unification factoring.

The above is a bit sketchy, and alternatives exist: the main point was to show that indexing can be integrated with *exo*-compilation.

## 7 Memory

An easy calculation shows the following: let the arity of the predicate be *A*, the number of facts be *F*. Full WAM code without instruction merging and without indexing, generates  $A \cdot F$  *get\_atom* instructions, *F* *proceed* instructions, one *try\_me\_else*, one *trust\_me\_else* and  $(F-2)$  *retry\_me\_else*. This amounts to  $3 \cdot A \cdot F + F + 3 \cdot F - 1$  words (instructions are aligned on a word boundary).

For *exo*-code, this becomes:  $A \cdot F$  words for the *exo*-table, one *try\_exo*, one *keep\_trying\_exo*, one *proceed* and *A* *get\_atom\_exo* instructions, which amounts to  $A \cdot F + 3 + 2 + A \cdot (2+1)$ .

For large *A* and *F*, the gain in memory is about factor 3, not counting the space that the atoms need in the atom table that is common to both models and which varies with the length of the atoms.

For the benchmark set we used up to now, we measured 252024 bytes for the WAM code, and 73912 bytes for the *exo*-code (which includes the *exo*-tables of course).

### 7.1 Cache measurements

Table 4 shows a summary of the cache simulations<sup>4</sup>: the *normal* row shows the figures for executing the benchmarks as compiled in ordinary WAM, while the

	I refs	D refs	D1 misses	L2 refs
normal	107,004,188,651	57,337,005,491	325,196,416	325,280,558
exo	113,616,319,238	60,208,012,211	67,685,660	67,769,855

**Table 4.** Summary of Cache Simulation

*exo* row shows the benchmarks compiled with the *exo*-compiler. We have left out the misses that were too close to be informative.

As expected, the number of I and D refs is larger for *exo*: about 6%. The D1 misses are much higher for *normal*: a factor of 5. The same holds for the L2 refs. It seems clear that *exo*-execution profits from better locality of data.

## 8 Discussion, related work, conclusion

The average Prolog application probably does not benefit from *exo*-compilation. In the ILP context there is a nice application: even if the facts representing one example cannot be considered big, they can be similar and there can be thousands of examples. *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. In the *exo* setting, one can just switch between the *exo*-tables instead. Since an *exo*-table is typically only one third in size of full code, this could lead to performance gain, and/or a significant increase of the datasets that can be handled by the system.

We have shown how highly regular code, as in big facts whose arguments are all atoms, allows for large code space reductions by separating the instructions from their operands: the technique is coined *exo*-compilation. *Exo*-compilation increases in principle the number of data accesses, because the data is accessed through an extra indirection. Also, it can break locality of reference, because the data is moved away from the code. But it improves locality by eliminating redundant copies of WAM instructions and by putting the the arguments of each fact closer together. In practice, the instruction overhead seems small and the improved locality of reference has a positive impact on performance. Also, *exo*-compilation invites optimizations that are more difficult to implement without separating the operands from the code. One optimization consists in removing the overhead of void variables in queries: this can be achieved to some extent by the Vienna Abstract Machine (see [4], or a tagging schema that caters for void variables (as in Beer [1] which caters for uninitialised variables), but these techniques still perform actions linear in the number of void variables, while our technique does not. Also dynamic indexing fits in nicely with *exo*-compilation.

Application of the *exo*-compilation principles to realistic contexts like ILP will eventually show whether the idea works in practice.

---

<sup>4</sup> They were obtained with the *cachegrind* option of *valgrind-3.2.0-Debian*

## Epilogue

While exploring exo-compilation, we became aware that Mercury [7] uses this technique as well. We thank Zoltan Somogyi for explaining the Mercury implementation of exo-compilation in extenso to us, and for useful comments on the current report. We intend to cooperate in writing down the experience of exo-compilation in Mercury, Yap<sup>5</sup> and hProlog.

## 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. B. Demoen. hProlog. <http://www.cs.kuleuven.be/bmd/hProlog/>.
4. A. Krall. The Vienna Abstract Machine. *Journal of Logic Programming*, 29(1-3):85–106, 1996.
5. V. Santos Costa, K. Sagonas, and R. Lopes. Demand-driven Indexing of Prolog Clauses. 2007 - Submitted to ICLP'07.
6. 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.
7. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

---

<sup>5</sup> Hopefully coming soon !