Odd Prolog benchmarking.

Bart Demoen, Phuong-Lan Nguyen

Report CW 312, June 2001

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

Odd Prolog benchmarking.

Bart Demoen, Phuong-Lan Nguyen

Report CW 312, June 2001

Department of Computer Science, K.U.Leuven

Abstract

In the course of developing hProlog - an alternative backend for HAL - we felt the need to check our implementation of certain features both for correctness and performance against other implementations. The first such feature was freeze/2 and later grew out in a set of benchmarking programs for catch&throw, findall, global variables, meta call (and variants), arithmetic, if/3 and copy_term. Some of these are not even ISO conforming, but they may be crucial for many applications. We discuss the problems in constructing benchmarks for such features, present the benchmark programs and the measurements in a number of relevant systems (B-Prolog, GNU Prolog, SICStus, Yap, hProlog, SWI-Prolog and ECLiPSe). Even though all these systems are (to some extent) WAM-based, we find huge differences and wherever we can, we try to explain them and/or indicate when an obvious explanation is wrong. As a side effect, we gain some insight in the overhead that ISO imposes.

Odd Prolog benchmarking.

Bart Demoen
Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
bmd@cs.kuleuven.ac.be

Phuong-Lan Nguyen
Institut de Mathematiques Appliquées
Université Catholique de l'Ouest
49000 Angers, France
nguyen@ima.uco.fr

Abstract

In the course of developing hProlog - an alternative backend for HAL - we felt the need to check our implementation of certain features both for correctness and performance against other implementations. The first such feature was freeze/2 and later grew out in a set of benchmarking programs for catch&throw, findall, global variables, meta call (and variants), arithmetic, if/3 and copy_term. Some of these are not even ISO conforming, but they may be crucial for many applications. We discuss the problems in constructing benchmarks for such features, present the benchmark programs and the measurements in a number of relevant systems (B-Prolog, GNU Prolog, SICStus, Yap, hProlog, SWI-Prolog and ECLiPSe). Even though all these systems are (to some extent) WAM-based, we find huge differences and wherever we can, we try to explain them and/or indicate when an obvious explanation is wrong. As a side effect, we gain some insight in the overhead that ISO imposes.

1 Introduction

We will assume working knowledge of Prolog and its implementation without further explanation. For a good introduction to Prolog see [16]; to WAM, see [1, 18]; for B-Prolog, see [19, 20]; the SICStus implementation is described in [3]. About Yap one can find implementation details in [5]; for GNU Prolog at [11] ; SWI Prolog can be found at [17]; ECLiPSe at [12]. hProlog has a predecessor [8] and is available at [13]. The ultimate documentation of these systems, their source code, is accessible, either for free or at a small cost for academia.

It is clear that only very rarely one would select a particular Prolog system because of its performance: features are more important than speed. Amongst appreciated features, ISO compliance and robustness score high, as so does user friendliness. However, performance is important as well, especially if performance does not come at the cost of the other more appreciated features. It is not always clear whether this is possible and especially when just a few people - or even one person - develop and maintain a complete system, performance might be the last thing one worries about.

About the performance of the most basic Prolog features - unification, backtracking, indexing, some built-in predicates - the community has developed an understanding. There remain however the less well known - say odd - predicates that sometimes aren't even standardized: different systems have implemented them in different ways and it is not always clear how systems perform neither what the reason is for a particular performance. Sometimes it is just believed that *tout est pour le mieux dans le meilleur des mondes* because a particular renowned implementation performs in a particular way, but we should be more inquisitive.

Even though users will not port easily their code to a new implementation because it implements feature X more efficiently, it is always worthwhile pointing out that feature X can be implemented in better ways - sometimes dramatically better. A comparative study of the performance of the odd features found in a series of systems can help achieving this goal. Such has been our interest for a long time, but only while we were building a new system - named hProlog (see section 13) - did we find the need to start this study: indeed, while adding features in hProlog, we needed to know whether our implementation was on the right track both performance wise and regarding the semantics.

The first such feature was freeze/2 (see e.g. [4]) and we ran into severe problems comparing our implementation with others: see 12. At first we compared hProlog with B-Prolog and SICStus and this for the obvious reasons: B-Prolog makes a point of implementing delayed goals very efficiently by means of a stack frame and stack frame freezing, while SICStus uses a more conventional heap based approach - which hProlog also uses. B-Prolog claims that its speed (of delayed goals) derives from its different implementation.

We also became interested in the performance of other systems for the same feature and we choose to include besides the above two mentioned systems also SWI-Prolog, Yap, ECLiPSe and GNU-Prolog for rather obvious reasons: SWI-Prolog is quite visible, has a wide user base and a robust reputation; Yap is reputedly and amazingly fast; ECLiPSe is one of the most complete but very old systems with an undeserved low profile and of course GNU-Prolog has done more for Prologs visibility in the real world than any other LP action since hyper resolution saw the light. All these systems have the advantage that they largely support the same set of features, even though they are not always based on exactly the same underlying architecture.

Freeze/2 was just the first feature we tested in different systems: the other features were findall/3 and exception handling by means of the predicates catch/3 and throw/1. Then a renewed discussion on call/n on comp.lang.prolog in February 2001 prompted us to do also some benchmarking of those predicates.

These features are implemented with an amazing difference in performance and we have also tried to identify the reason. Also if/3 is in this category.

All the above mentioned built-in predicates are meta predicates: they call a goal that was constructed as a term, basically by means of the predicate call/1 [1]. Therefore, we start our measurements with a benchmark that measures only call/1 in section 3.

Another feature that HAL relies on, is *global variables*: these can be mimicked in any ISO-conforming Prolog system, but some systems offer support at a better level. The outcome of the measurements was interesting. Global variables arouse interest periodically in news groups and their functionality might be subject to standardization in the future. Related to global variables - but also to findall/3 - is the performance of copy_term/2 (see section 5). Also the sort/2 was investigated further, especially since the compare/3 family of ISO bips does not work really on variables, so a user has to rely on system support at least sometimes.

We then also made a short attempt at benchmarking arithmetic: see section 11.

We have other feature on our wish-list-to-benchmark: interrupt handling and garbage collection for instance. These will be the subject of future investigations.

This report clearly is not about assessing the general performance of a Prolog system neither does this report pretend to offer a suite for measuring some of the odd features of Prolog systems: it is quite difficult

- to measure exactly what one wants

[1]in B-Prolog, this is not true for freeze/2

- to measure something relevant
- to measure it in a uniform way across Prolog systems

Certainly, this report is not about showing off how well we did with hProlog, although we have not much to be ashamed off either. Indeed, hProlog turns out to be among the fastest (if not the fastest) for every single benchmark. Still, we are no better implementors than others, so obviously, the reason must be that we have made different basic choices or tradeoffs. When we can, we will indicate these choices. In particular, we will indicate - when possible - how the ISO Prolog standard seems to impose an overhead on certain features that is not always wanted neither needed.

So what is this report about ? It is about showing that without sacrificing robustness and ISO-compliance, better than currently available average performance is possible - sometimes much better. Let the user community prod their Prolog supplier for it.

The paper should contain enough information to understand the inner part of each benchmark. The full set will be obtainable from `http://www.cs.kuleuven.ac.be/~bmd/odd_benches` [2].

The version of hProlog with which the benchmarks were done, can also be obtained from the same place. It is currently only tested on Intel with Linux.

The experiments were performed on a Pentium III, 500MHz, 128 Mb - our findings might not carry over to other architectures, but we are actually convinced they do. Timings are reported in milliseconds. We used B-Prolog 4.0 #3, SICStus 3.8.5, Yap4.3.0, SWI-Prolog (Version 3.4.4), GNU Prolog 1.2.6, ECLiPSe Version 5.1.0 and the continuously evolving version 1.3 of hProlog. We have formerly done more extensive measurements comparing B-Prolog and hProlog: see [7]. The main reason for including B-Prolog in this report is its implementation of delay. Since B-Prolog is written in ANSI-C - while most other systems rely for speed partly on gcc specifics - we have in many tables included figures for a version of hProlog that was compiled with the -ansi option.

While running the benchmarks, we encountered regularly problems with particular systems - memory leaks, segmentation violations, inconsistencies between the manual and the implementation ... We have most often reported these to the author(s) of the systems and most problems have been fixed in the mean time - we were however not always organised well enough to run our benchmarks again with the fixed system. These occasional problems and also misunderstandings from our side have lead to some empty entries in some of the tables: these empty entries should be interpreted as *no information.*

Before presenting any benchmark results at all, we have to stress that we consider differences in implementation that are in the order of 50% or less rather uninteresting; a factor of two or more is what we find meaningful. Consequently, we will treat small percentages as noise and not comment on them.

2 Issues in benchmarking Prolog features

When benchmarking a particular feature, the benchmark needs to be repeated many times to give meaningful timings: we name this loop the *interesting* loop. It is then customary to subtract some sort of *empty* loop from the interesting loop, so that the loop overhead does not mask the feature that we want to measure. This is potentially deceiving. Let's take the example of a benchmark involving findall/3. This built-in predicate is composed of at least two ingredients: it (usually) meta calls the generator and it copies (by a method similar to copy_term) the template (perhaps even twice). If one wants to benchmark findall/3, should the empty loop do a meta call (of true/0 for

[2] if it is not, ask the first author

instance) or not ? For a system with a fast meta call, it does not matter. For a system with a slow meta call, it does. The argument in favour of doing a meta call in the empty loop is that after all findall/3 does a meta call that in practice can always be avoided: in most programs, the generator is indeed not a program variable. The argument against doing the meta call in the empty loop, is that it seems that no system avoids the meta call in findall/3, even if that is possible, maybe - but perhaps not justifiably - because of the code growth that could results. We have resolved this issue as follows: since we made a separate measurement of the meta call (see section 3), it makes sense to put a meta call in the empty loop of the other benchmarks whose interesting loop contain meta calls: in this way, we measure the issues separately. The measurements of freeze/2 are an exception to this rule, for reasons explained in section 12.

A second issue relates to ISO compatibility, or more generally, to the exact semantics of a feature in case it is abused: indeed, most often systems do agree on the semantics of features when they are used correctly. The most blatant example is again meta call: ISO requires that for a goal like *call((foo,7))*, the exception for the illegal goal 7 is raised before the goal *foo* is executed. So, ISO requires basically to make a pre-pass over the goal to be executed. Some systems don't do that - and thus do not conform to ISO - and they might unjustifiably show good results during the benchmarking. This means that just running a benchmark is not enough: one needs also to take into account the conformance to a certain semantics standard (ISO being the only more or less rigorous one). The other edge of the sword is that one might gain an insight in the *performance penalty* ISO is imposing: this issue has not been given much attention and probably is one of the reasons for more than one bad decision by the ISO committee. Section 3 will show more on this.

A third issue is related to supported features that are ISO compliant (as implementation defined or dependent features) but are not visible in a particular subset of ISO Prolog. A particular example is the support for rational trees and its impact on e.g. copy_term: one way to cater for rational trees is to short circuit the term that is being copied (similar to the way that unification would do). This affects the speed at which structured terms are copied, even if they are not circular. But it doesn't affect copying atoms or at least, it shouldn't. It means that hidden (and good !) features can have a negative impact on speed and without mentioning this, the benchmarks results become less meaningful.

In view of the above, we have tried to critically examine both the benchmarks and the context in which they are run. We cannot claim we have made a full discussion of every benchmark: examining just one built-in feature would require a full paper, and would interest few people.

Finally, we have made sure that during the benchmarks, garbage collection was never ran. Memory management in Prolog is still too little understood, so we could not allow this to blur the picture.

3 A key factor: call/1

The benchmarks in most of the following sections involve meta calling a goal: call/n (with $n > 1$), findall/3, exception handling, if/3 and freeze/2. So it is crucial to measure the performance of call/1 independently: when in the other benchmarks, meta call is part of the empty loop, the results in this section will complete the picture.

Before looking at the benchmarks, it is worth looking at the characteristics of the meta call in different systems: ISO requires two things from the meta call; the argument is interpreted as a body and

- if it contains a (runtime) variable X - at the moment of doing the meta call - it must mean the same as if it were actually call(X); we name this *var replacement.*

- if it contains a number as a goal, it must throw an exception before executing any part of its argument; this will be referred to as *precheck*

The meta call in SWI, SICStus, GNU and hProlog is by default ISO compliant. In ECLiPSe and Yap, it is by option. B-Prolog has only the non-ISO mode for call/1 and in hProlog we also have a non-ISO call/1 just for the sake of comparison. Also, we must admit that we added precheck and var replacement to hProlog while working on this paper: as a backend for HAL, none make sense and we will remove them later.

An ISO-compliant meta call also needs to deal with modules (meta expansion and module lookup) at runtime and SICStus even performs goal expansion. Following a discussion on comp.lang.prolog March 2001, we measured that for SICStus the overhead of modules is in the order of about 35% and 22% for goal expansion (when no goal expansion needs to be done). Note that e.g. SWI Prolog has the same module system as SICStus Prolog, but the figures below show that the cost there is much lower. The modules standard is quite recent, but SICStus has implemented modules quite close to ISO already for many years.

The most commonly meta called goals are just single goals, so our artificial benchmarks test those to start with: a goal with one argument, and a goal with 10 arguments. Meta calling these goals is contrasted with calling them directly. ISO conformance when meta calling a single goal should not impose any overhead at all. Then follow meta calls of conjunctions with 2 and 3 goals (all with arity 1). For the systems in which this is possible, we give the figures for these tests both in ISO conforming and non-conforming mode: in non-conforming mode, neither precheck nor var replacement is performed.[3] The goals are shown in the first column of table 1. The definitions of g/1 and g/10 are simply

```
g(_).            and            g(_,_,_,_,_,_,_,_,_,_).
```

	B-Prolog	hProlog (ansi)	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog
call(g(1))	310	140	170	700	4600	2600	360	100
call(g(1,2,3,4,5,6,7,8,9,0))	380	250	260	879	4700	2820	390	190
module call(g(1)) (**)					320			350
module call(g(1,...9,0)) (**)					430			370
g(1)	130	60	10	229	90	90	20	10
g(1,2,3,4,5,6,7,8,9,0)	240	130	120	380	180	130	120	90
ISO call((g(1),g(1)))		2610		2980	9640	4170	4810	1920
ISO call((g(1),g(1),g(1)))		4310		3569	14450	5660	(*)	2900
non ISO call(g(1),g(1))	1240	450	800				1790	410
non ISO call(g(1),g(1),g(1))	1940	770	1350				2380	660

Table 1: Call/1

(*) a bug needed fixing in Yap; it was detected by performance measurements :-)

The figures in the rows marked with (**) have only a SICStus and a hProlog column: the SICStus column was obtained by replacing the goal *call(X)* by the goal *prolog:call_module(X,user,[])*

[3]the module system of hProlog is atom based and has by nature zero overhead at runtime

which seems to be the basic call for a single goal once goal expansion and the module system is dealt with; in the hProlog column, essentially the same *call_module/2* goal was measured in the hProlog context: although hProlog never needs to make runtime decisions on which module to call a predicate in, the functionality to make a dynamic decision to call a certain predicate in a particular module exists anyway. The result shows that the basic meta call of SICStus Prolog is reasonable (and also in the same ball park as Yap for instance). The overhead by the runtime module call, is quite high (a factor of 3 and more) but the overhead resulting from the other features seems overly excessive.

The following things are noteworthy in our opinion:

1. the cost of meta calling a single goal is excessively high in SICStus Prolog and GNU Prolog

2. even in hProlog, whose meta call is much faster than in the other ISO conforming systems, the overhead of being ISO conforming is huge for conjunctions

3. for single goals - i.e. not a conjunction or another non-simple construct - the overhead due to being ISO conformance is actually non-existent in hProlog: hProlog uses for every meta call the basic one, and it is the meta called version of the ,/2 predicate that will pay some ISO price only; in contrast, SICStus uses (must use because of meta and goal expansion) the full blown meta call even for simple goals

There is a saying *Sometimes a cigar is just a cigar*, but we think that most often a meta call is just a meta call, not a demand for goal expansion, meta predicate expansion, module detection, error detection ... It seems unthoughtful to have a Prolog language with advertised and built-in meta programming capabilities, which puts undue overhead when such meta features are used.

4 Call/n with $n > 1$

Call/n is a recurring topic in news groups and FAQs: a user can define her own (faked_)call/(n+1) for instance as

```
faked_call(Goal,Arg1, ..., Argn) :-
        Goal =.. [Name|[Args],
        append(Args,[Arg1, ..., Argn],NewArgs),
        NewGoal =.. [Name|NewArgs],
        call(NewGoal).
```

SWI and hProlog provide a faster implementation of call/n [4]. Table 2 shows some measurements for call/2 in different systems, either as provided or as written by the user. The empty loop does not contain any meta calls in this case.

The figures indicate clearly that low level support of call/n, pays off. In terms of system implementation effort, call/n costs nearly zip.

5 Copy_term/2

Before measuring findall/3 in the next section, and since copying a term is part of most implementations of findall/3, we benchmark copy_term/3. This predicate is a dangerous built-in to

[4]hProlog provides other forms of fast meta calling as well, because it needs to support type classes

	B-Prolog	hProlog (ansi)	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog
call(a,a)	-	1190	-	2270	-	-	-	750
call(a(1,...,9),a)	-	1570	-	3179	-	-	-	1200
faked_call(a,a)	9030	6700	6460	20179	29530	16740	5670	4980
faked_call(a(1,...,9),a))	18030	16250	12000	73900	34920	25410	10480	9950

Table 2: Call/2 and faked call/2

benchmark: some implementations preserve the internal sharing in a term while copying it, while others don't. This is related to, but not the same as supporting rational trees or cyclic terms. Such support for sharing preservation usually affects the performance of copying a structured term, even if that term has no internal sharing. For copying atoms, there should be no cost. Table 3 has two parts, testing twice the copying of the same terms: results are given for a copy_term that does not preserve sharing and for a copy_term that does.

Yap and hProlog can be installed with or without preserving sharing [5]. SICStus has this feature by default.

Copy_term/3 is usually not written in Prolog, so one would expect performance of systems not to diverge much, but the differences are significant. The results for Yap and hProlog show that the overhead of preserving sharing is not high at all.

	B-Prolog	hProlog (ansi)	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog
destroy sharing								
a	300	140	510	400		170	110	90
a(0,...,9)	920	470	680	899		770	330	410
a(a(0),...,a(9))	1600	830	1250	1809		1240	900	770
preserve sharing								
a		150			480		120	90
a(0,...,9)		490			950		330	440
a(a(0),...,a(9))		1060			3060		1220	1000

Table 3: Copyterm/2

To check whether an implementation preserves sharing, observe its behaviour on the following program

```
test(N) :-
        mkterm(N,Term),
        copy_term(Term,NewTerm),
        fail,
        use(NewTerm).

mkterm(N,Term) :-
        (N > 0 ->
            M is N - 1,
```

[5]hProlog does not preserve sharing for lists, but it does so for other structured terms

```
                Term = f(Term1,Term1),
                mkterm(M,Term1)
            ;
                Term = end
            ).
```

for the query $?-test(N)$. and different values of N.

6 Findall/3

Since findall/3 uses the meta call, every system with a slow implementation of call/1, will perform poorly on findall/3. To exclude this effect, we have defined the empty loop - which we subtract from the interesting loop with a findall goal - so that it performs as many meta calls as the findall loop: the figures in table 4 thus represent the overhead of findall related stuff excluding the meta call. It also means that to really know what a call to findall/3 costs, one should take table 1 into account.

The two rows findall-fail, findall-one and findall-ten are obtained by calling the goals $findall(a, fails, \_)$, $findall(a, succeeds\_once, \_)$ and $findall(a, succeeds\_ten, \_)$ repeatedly, where we have the definitions:

```
fails :- fail.              succeeds_once.

succeeds_ten. % 10 of these facts
```

But the implementation of findall/3 is more than meta call alone: usually an initialization of some data structures needs to be done at the beginning of the execution of findall/3, even before the goal is meta called. Answers must be copied - in most implementations even twice - and at the end of the findall/3 call, data structures might need to be cleaned up in some way or another. Moreover, the whole of findall/3 should be protected from being exited by an exception (thrown by the goal) without cleaning up. So the following structure of an implementation of findall/3 is not uncommon:

```
findall(Template,Goal,AnswerList) :-
        init_findall(Handle),
        catch(
                (
                        call(Goal),
                        copy_answer_out(Template,Handle),
                        fail
                ;
                        retrieve_answers_and_cleanup(AnswerList,Handle)
                ),
                AnyBall,
                (findall_cleanup(Handle), throw(AnyBall))
             ).
```

Some systems do not perform a cleanup when an exception is thrown inside the Goal: they typically crash or have a noticeable (by the UNIX top command) memory leak when this happens often enough. The code to test for the memory leak is simply:

```
a :-    findall([a,b,c,d,s,X,g,y,h,r,e],f(X),_), fail.

f(X) :- (X = 2 ; throw(out)).

go :-   (catch(a,_,true), fail ; go).
```

The table 4 has an indication of what happens in each implementation.

The copying of the template is also a source of discrepancy: some systems support cyclic terms to the extent that they can be in the answer list of findall/3. Such support has a price. We have discussed this in more detail in section 5, but note here that the answer list in our findall benchmarks only contain atoms, so the cyclic term support should not affect the results.

Finally, ISO requires two more checks: AnswerList must be a (partial) list, and Goal must be callable. Testing the latter can be done without extra overhead. Since the AnswerList is almost always a new variable, the former can be implemented with a usually negligible cost - but it is not always done that way. Some implementations perform these checks before starting the essence of findall/3: see table 4.

	B-Prolog	hProlog (ansi)	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog
findall-fail	(*)	630	1290	849	2840	240	450	470
findall-one	(*)	670	3040	2240	4290	570	760	520
findall-ten	(*)	2220	18430	7950	11050	1870	3020	1520
checks list	no	yes	yes	no	no	yes	yes	yes
goal check	?	yes	no	no	yes	yes	no	yes
cleanup on exception	crash	ok	ok	leak	ok	leak	leak	ok

Table 4: Findall/3

(*) The benchmark could not be run reliably in B-Prolog, as repeating it, resulted in a segmentation fault.

The results show that it is entirely possible to perform very well, while conforming to ISO. The leak in some systems can (should !) be closed easily at a reasonable cost.

7 Exception handling

Exception handling through the predicates catch/3 and throw/1 is usually implemented by meta calling the protected goal. So, also in this benchmark, the empty loop performs as many meta calls as the interesting loop. The last row in table 5 indicates whether the system performs last call optimization for clauses that call catch/3; of course this is only possible when the protected goal is deterministic; the system then has to detect this (at runtime for Prolog) and perform the same sort of LCO as in usual code. We used the following predicate to test this:

```
tailcatch(B) :- catch(succeed_once,B,true), tailcatch(B).
succeed_once.
```

We measured the cost of LCO in hProlog - by simply switching (LCO is done at the source level) it off for the catch-true row and found its cost about 240 of the 480 in the table: that is relatively high for hProlog and SWI, but it is (or would be) relatively low in the other systems; see table 5.

The rows in table 5 correspond to the following goals

```
catch(succeed_once,_,succeed_once)
catch(throw_direct,_,succeed_once)
catch(throw_after_10,_,succeed_once)
catch(catch_fail_10,_,succeed_once)
catch(backtrack_10,_,succeed_once)
```

with

```
succeed_once.

throw_direct :- throw(p).

throw_after_10 :- throw_after_9, just_for_alloc.
throw_after_9  :- throw_after_8, just_for_alloc.
...
throw_after_1  :- throw(p), just_for_alloc.

catch_fail_10 :- catch(catch_fail_9,_,succeed_once).
catch_fail_9 :- catch(catch_fail_8,_,succeed_once).
...
catch_fail_1 :- fail.

backtrack_10.
...
backtrack_10.
```

	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog
catch-true	1260	219	2500	2530	18540	480
catch-throw	9950	2899	20520	10110	20480	1790
catch-throw10	10960	7290	21700	10230	21040	2430
catch-fail	990	119	2320	2170	15460	560
catch-backtrack	3360	4730	7220	14440	61980	3050
LCO	yes	no	yes	yes	no	yes

Table 5: Catch/3 and throw/1

B-Prolog seems not to support catch&throw, hence its absence from table 5.

SWI-Prolog does not follow completely the ISO specification: it does not copy the Ball *before* trying to unify it with the Ball; this affects only slightly the figures in the throw and throw10 case, but the low cost of entering a catch (especially the catch-fail row) is unaffected and quite impressive.

Note that an implementation of catch&throw along the lines of the JVM is possible but a zero-cost entering of the protected region as in the try-catch construct in Java is impossible in Prolog because the standard requires unwinding of the stacks, meaning that a choice point is necessary.

Finally, note that Yap uses a choice point based implementation of catch&throw, as described for instance in [6], this in contrast with an environment based method used in other systems and which is also the natural one in a language like Java. The figures in table 5 seem to settle the question put forward in [6] which one is better: the choice point method is overall slow and even particularly slow when the protected goal is non-deterministic. Part of the slowness of Yap might come from using record instead of a proper global variable for setting the scope of a handler.

The ISO Prolog standard encourages the use of catch&throw based exception handling, since the abuse of built-in predicates results in the throwing of an exception. It is therefore very important to have an efficient implementation of this feature. The schemas around are mostly unsatisfactory and the SWI schema deserves more attention as it seems to be quite efficient in entering and exiting the scope; maybe a hybrid of the SWI and hProlog way will turn out to be universally acceptable and most efficient.

8 If/3

If/3 is the logical if-then-else: in contrast with the Prolog if-then-else, it backtracks over the alternative solutions of the conditions. It is straightforward to implement if/3 in Prolog, as long as one can get some unique identifiers. Here is one possibility:

```
if(Cond,Then,Else) :-
        gensym(Sym),
        remember(Sym,nothing),
        (
           call(Cond),
           remember(Sym,cond_succeeded),
           call(Then)
        ;
           remembered(Sym,nothing),
           call(Else)
        ).
```

The implementation gensym/1, remember/2 and remembered/2 can be done with assert or global variables. Apart from being slow (we measured it between 5 and 10 times slower in hProlog than the version that is supported at the low level) this implementation has at least two more flaws:

1. the disjunction choice point is not cut away in case the condition succeeds
2. the *memory* used to remember whether Cond succeeded or not is not recovered

Also, a system might run out of symbols (or some other resource) quickly with this implementation !

HAL supports the logical if-then-else (as does Mercury) and therefore it was important for hProlog to do the same. Both points above are easy to remedy in an implementation with a bit of support for cutting away or invalidating choice points that are not most recent. Here is the hProlog way [6]:

[6]the knowledgeable reader might recognize some XSB heritage in the naming

```
if(A,B,C) :-
        if(A,B,C,0).

if(A,B,C,_) :-
        '_$savecp'(CPbefore),
        call(A),
        smash_cp(CPbefore,4,1),    % overwrite the 4th argument of the choice point with
        '_$savecp'(CPafter),
        (CPbefore = CPafter ->
            !                      % LCO
        ;
            true
        ),
        call(B).
if(A,B,C,0) :-
        call(C).
```

Since only hProlog and SICStus support if/3, the following table is restricted to these systems.

goal	SICStus	hProlog
if(true,true,true)	8850	1440 (3580)
if(fail,true,true)	18130	760 (2670)
if(backtrack10,true,true)	5820	870 (1090)

Table 6: If/3

Note that hProlog seems to implement a failing condition much more efficiently than a succeeding one, while SICStus does the opposite. SICStus checks the arguments of if/3 before starting to execute the condition: as a backend to HAL, this makes no sense for hProlog. So the default is that there is no pre-checking. Between brackets are the timings with pre-check included for hProlog.

For this benchmark, the empty loop did not contain any meta calls, but it is entirely possible to avoid meta calls in if/3. Both hProlog and SICStus Prolog do LCO for if/3.

IBM-Prolog [7] used to support if/3 as the construct

```
Cond some Then none Else
```

9 Sort/2

Sorting is done in the context of setof/3. It is important that the built-in sort/2 is efficient: if the user wants a slower one, she will find it easy to write it. The following table shows the timings for sorting all permutations of a list of 10 different integers.

The Prolog sort/2 is a copy of the SICStus Prolog implementation of sort/2 (which is in Prolog) attributed partly to R. O'Keefe. The difference in the SICStus column between the figure for the built-in sort and the Prolog sort is due to a different implementation of must_be/4 (which checks that the first argument of sort/2 is a list), which we specialized a bit: the difference shows a tradeoff that SICS has made between implementation effort and performance !

[7] no, it is not a predecessor of HAL-Prolog :-)

	B-Prolog	hProlog (ansi)	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog
built-in sort	373700	26340	18850	57070	175100	26610	16230	25710
Prolog sort	243870	134670	243910	607109	118110	123730	117790	91870
qsort	286960	110140	198500	464400	101880	99100	101160	65260

Table 7: Sorting

The row with qsort uses a straightforward qsort/2 with accumulating parameter and if-then-else. Funnily enough, it performs better in almost [8] all implementations than the very crafty one by O'Keefe. Of course, a 10-element list is not long enough to draw final conclusions.

One notable issue here: sort/2 written in Prolog relies on compare/3 or on one of the variants of @¿/2. An implementation usually chooses either one as *basic*. There are always circumstances where such a choice is the bad one and for overall uniform good performance, **both** must be implemented at a lower level: that's the choice made in hProlog.

10 Global variables

ISO-Prolog does not acknowledge the need of global variables, instead it stuck to dynamic predicates that can be used to mimick them and moreover, the record-family of predicates became obsolete. Still, many systems provide global variables which differ from dynamic predicates or recorded information in mainly three aspects:

- global variables have only one value at any moment (some systems can only store atomic values, others also structured terms)
- they can be backtrackable or not (not all systems offer both)
- these variables are associated to atoms (sometimes to other types as well)

GNU-Prolog and hProlog offer such global variables.

One of the important usages of global variables is for keeping a counter. So our benchmark just stores and retrieves integers by means of the mechanisms offered by each implementation. There is no arithmetic performed during this benchmark.

We also did the benchmark while storing and retrieving a small compound term (f(1,2,3)). The issue of preserving sharing (or supporting cyclic terms) is present here as well, but we have seen earlier that this is cheap.

The primitives we used for non-backtrackable global variables are

- **SWI:** flag/3
- **SICStus:** bb_put/2 and bb_get/2
- **GNU:** g_assign/2 and g_read/2
- **Yap:** set_value/2 and get_value/2
- **ECLiPSe:** setval/2 and getval/2 (together with a local variable declaration)

[8]B-Prolog is an exception probably because it doesn't do the usual optimizations on the Prolog if-then-else

stored term	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog
1						
assert/retract	7040	4720	12510			
record*		3910	6740			
nb-globvars	1060	1720	3100	350	320	200
f(1,2,3)						
assert/retract	9520	6179	12860			
record*		4979	7120			
nb-globvars	1520		3530	910		830

Table 8: Non-backtrackable global variables

- **hProlog:** nb_setval/2 and nb_getval/2

hProlog, Yap and GNU-Prolog seem to have something in common in the implementation that gives them a speed advantage over the other implementations.

11 Arithmetic

The performance of arithmetic is quite difficult to assess. First of all, one should consider integer and floating point arithmetic separately of course. For integer arithmetic, implementations all seem to have a positive MAXINT and negative MININT. Integers within the range $[MININT...MAXINT]$ have an efficient representation and either

1. overflow is detected and a different representation is used for integers outside this range or
2. overflow is not detected at all

The form: overflow is detected and throws an exception seems not present.

It is clear that an interpretation of a benchmark result without taking into account these implementation decisions, cannot provide a good insight. Some systems (e.g. Yap) allow their installation with and without support for bignums. Others (like SICStus) have only this mode of installation.

To summarize, SICStus, SWI, ECLiPSe are of the first kind, Yap, GNU of the second and hProlog can be made to check overflow, and then just throws an exception.

Another issue that is relevant to benchmarking, is the value of MININT and MAXINT: if a benchmark for one system goes over them, but not for another system, the results will be unreliable, or at least difficult to interpret. So, before benchmarking, we have made sure that the integer arithmetic benchmarks do not use integers outside of the range $[MININT...MAXINT]$ for any system. We have three benchmarks: one for adding integers, one for adding floating point numbers and one for multiplying integers.

It is worth noting that none of the tested systems does the kind of optimizations commonly found in compilers for classical languages. This is fortunate during benchmarking. E.g. one of the benchmarks contains the clause:

```
add1([_|R],A,B) :-
```

	B-Prolog	hProlog (ansi)	ECLiPSe	SWI	SICStus	GNU	Yap	hProlog check/nocheck
integer + 1	1040	430	1730	18270	940	1690	430	500/480
integer + integer	640	800	1870	19140	1050	2140	630	610/830(*)
integer * integer	1140	520	2340	20019	2680	1980	650	1200/560
float + float	2820	1800	3770	25139	3710	3380	5280	- /1910
float * float	14270	1430	3770	24539	3870	3380	5280	- /1520

Table 9: Some arithmetic

```
        Z1 is A + 1,  Z2 is Z1 + 1, Z3 is Z2 + 1, Z4 is Z3 + 1, Z5 is Z4 + 1,
        Z6 is Z5 + 1, Z7 is Z6 + 1, Z8 is Z7 + 1, Z9 is Z8 + 1, Z0 is Z9 + 1,
        f(Z0),
        add1(R,A,B).
```

which could be optimized to

```
add1([_|R],A,B) :-
        Z0 is A + 10,
        f(Z0),
        add1(R,A,B).
```

But it is unfortunate for the state of affairs in Prolog compilers: they are largely responsible for the vicious circle that goes like "Prolog compilers are no good at compiling arithmetic, so nobody uses Prolog for arithmetic; since nobody uses Prolog for arithmetic, Prolog compilers tend to ignore arithmetic".

The above measurements were particularly unsatisfying: large unexplained fluctuations were observed. In particular the entry marked with (*) is weird. However, it is clear that most systems should take arithmetic more serious.

12 Freeze/2

Benchmarking freeze/2 is particularly difficult: there is no standard semantics for melting frozen goals and it is extremely difficult to assess conformance between two systems, even if for some wide range of programs, these two systems produce the same answers for a problem. The problems with the semantics of freeze/2 are well-known: when exactly is the wake-up performed, and in what order are woken goals activated ? Some of these problems can be exemplified by:

- in the query ?- freeze(X,g1), freeze(Y,g2), X = Y, X = 1. which of g1 and g2 is executed first ? is there a reason for choosing the *oldest* goal first ? or for choosing the goal from the originally oldest variable first ? and what would oldest mean exactly - given ISO's stand on the comparison of variables ?
- in the query ?- freeze(X,g1), freeze(X,g2), X = 1. we have a similar problem
- in the query ?- freeze(X,g1), freeze(Y,g2), f(Y,X) = f(1,2). we have again the same problem, but now, one might argue it depends on the (non-standardized) order of unification ...

It is clear that an application (or a benchmark) that is written with a particular wake-up model in mind, might perform better in this model than in others, but there is no guarantee that this is true. It is even possible that in one wake-up model the program works and in another model it results in runtime errors; take the queries:

```
?- freeze(X,arg(X,f(2),Y)), freeze(Z,arg(Y,f(a,b,c),T)), f(X,Z) = f(1,p).
```

and

```
?- freeze(X,arg(X,f(2),Y)), freeze(Z,arg(Y,f(a,b,c),T)), f(Z,X) = f(p,1).
```

In SICStus Prolog, the first one succeeds, but the second one throws an exception.

One might argue that programs that rely for *correctness* on a particular wakeup model are erroneous, and one might be lucky to discover such reliance by running the same program in different systems. However, a program might rely on a particular model for its *performance*, and although this can be considered equally erroneous, this is much more difficult to prove or disprove.

One more problem is fail/0. At the toplevel of Prolog systems, the query ?- freeze(X, write(pok)), X = 1, fail. writes pok before failing in B-Prolog, SICStus Prolog and hProlog; in Yap it fails without output.

On the other hand, a file with the clause a :- freeze(X, write(pok)), X = 1, fail. when consulted, and queried with ?- a. will produce output in SICStus, but not when the file was compiled; same in B-Prolog; in hProlog and Yap, output is never produced [9].

Since we wanted to make as fair a comparison as possible, we have spend quite a bit of time - actually most of the time spend on implementing freeze/2 in hProlog - just trying to make the order in which goals are melted, exactly as in SICStus Prolog - which happens to be the same as in B-Prolog as far as we know - but different from Yap's. Still, any optimization that affects the order in which arguments of structured terms are unified, could break this immediately.

We got one supposedly non-toy benchmark for freeze/2 from Neng-Fa Zhou; we name it later *shirai* after its implementor Yasuyuki Shirai from Mitsubishi Research Institute, Inc. This benchmark ran only under B-Prolog, SICStus Prolog and hProlog (under Yap 4.3.0 it bombed; we didn't investigate long enough ECLiPSe).

We have also added some artificial benchmarks, just because for these, there should be no disagreement between systems on when and in which order wake-up is performed.

There is one more problem with running the same benchmarks under hProlog as under other systems: hProlog does only a wakeup test at the calls to Prolog predicates; e.g. not just before a cut, nor in between two goals that are inline built-in calls (arg/3 for instance), nor before entering an explicit disjunction (;/2). As far as being a backend for HAL is concerned, this is no problem, as the pre-compiler is responsible for adding checks for wake up [10]. As for the benchmarks: it is necessary (both for correctness and for fairness) to add extra calls at specific program points (see above). So, the shirai benchmark is run as shirai2 in hProlog: goals *check_wakeup* were added at all necessary places (and probably a superset of them). The predicate *check_wakeup* is simply defined as the fact check_wakeup. [11].

[9] hProlog does not distinguish between compile and consult

[10] and because !/0 does not exist in HAL

[11] Another problem with shirai is that it uses the record database; hProlog simulates that with its global variables, which are more efficient but the fraction of the benchmark's time that is spend in these predicates is less than 1%

The tables will also mention the number of calls to freeze/2 with a free first argument, with a bound argument, the number of wake up events (that is a checkpoint at which a series of goals was woken) and the number of goals actually woken. This was measured for hProlog.

	B-Prolog shirai	hProlog (ansi) shirai2	hProlog shirai2	SICStus shirai	# freeze calls var	# freeze calls bound	# wake up events	# executed woken goals
run(8)	30	40	30	60	621	1506	361	884
run(9)	70	70	40	100	871	1857	582	1474
run(10)	270	250	180	320	1180	3102	2497	7130
run(11)	1060	1030	750	1250	1554	6876	8223	28145
run(12)	2540	2380	1670	2730	1999	9888	16612	58683
run(13)	145030	133760	93640	149430	2521	461132	974016	3168216

Table 10: The shirai benchmark

Although shirai is considered a non-tiy benchmark, from the table 10, the best thing one can conclude is that it does not measure adequately the performance of freeze/2: freeze/2 is called relatively little, and comparing with table 11 which contains some artificial freeze/2 benchmarks, one sees that doing in the order of 1M of wakeups, takes in the order of seconds, and since run(13) takes about 150 secs, less than 10% of the runtime is spend in freeze/2 related actions.

The artificial benchmarks are more interesting, because they try to split out the issues in delaying and reactivating a goal: these are

- freezing + triggering (by unification) and doing a wakeup once for each frozen goal
- freezing (but failing before a wakeup is triggered)
- freezing + triggering and doing a wakeup often for each frozen goal

The table 11 contains the results for a goal that is an atom, and a goal that has arity 4. M stands for Mega, K for Kilo.

delayed goal	B-Prolog	hProlog (ansi)	hProlog	Yap	SICStus				
g									
freeze+uni+melt	1940	3010	1810	4070	11470	1M	0	1M	1M
freeze	1260	910	430	1460	850	1M	0	0	0
freeze+(uni+melt)*	1130	2280	1400	2870	10680	10K	0	1M	1M
g(1,2,3,4)									
freeze+uni+melt	2010	3150	1940	4120	12750	1M	0	1M	1M
freeze	1250	910	430	1480	860	1M	0	0	0
freeze+(uni+melt)*	1190	2430	1560	2920	12050	10K	0	1M	1M

Table 11: Artificial benchmarks with freeze

It shows that B-Prolog is quite good at melting a goal - about twice as efficient as ANSI hProlog, but slightly slower in doing the actual freeze. The former conforms to the claim that B-Prolog, by avoiding reinstalling argument registers repeatedly, is more efficient for reinstalling a goal. Note that this is not meta call related: B-Prolog has a slower meta call than hProlog, but that does not matter here, since B-Prolog does not need to meta call a woken goal. That is also the reason why the empty loops in the artificial benchmarks do not contain meta call. As we have pointed out in [7], WAM can easily incorporate the basic idea of freezing a goal like B-Prolog does, without having to adhere to B-Prolog's argument passing schema, but the price is a freeze register that blocks stacks in a similar way as in XSB.

It seems no coincidence that the sum of the rows labeled *freeze* and *freeze+(uni+melt)** is very close to the row labeled *freeze+uni+melt.* We can also get some ball park figures for B-Prolog: it can do about 1M freezes per second and 1M melts (of individual goal) per second. For ansi hProlog the former is almost equal, the latter only half of it. Given that B-Prolog and ansi hProlog are on ordinary programs (not using freeze/2) of almost equal performance, this can help in assessing the measurements in table 12: it contains the benchmark results for programs from the distribution of B-Prolog.

	B-Prolog	hProlog (ansi)	hProlog	Yap	SICStus				
queens_freeze 10	820	1210	760	1980	11930	304K	0	110K	811K
queens_freeze 11	4430	6510	4060	10510	64640	1604K	0	546K	4432K
queens_freeze 12	25790	37280	23220	59980	374740	8991K	0	2915K	25769K
sort_freeze (17)	1880	4110	2640	3930	36510	1114K	0	1114K	2228K
nreverse_freeze (320)	940	1540	980	SEGV	6500	510K	3K	510K	510K
sendmoney_freeze	1350	2500	1590	4330	20600	105K	382K	627K	1041K

Table 12: Small benchmarks with freeze

Let's look at the queens_freeze benchmark first: the timings for B-Prolog correlate directly with the figures in the last column and according to 1M/sec. However for the sort_freeze and sendmoney_freeze, this *invariant* seems broken and also the timing differences with ansi hProlog are larger than expected, so a closer look at the benchmarks is in order.

It is well known that the following two goals are equivalent

```
freeze(A,freeze(B,goal(A,B)))              freeze(B,freeze(A,goal(A,B)))
```

so a compiler is allowed to transform one into the other: if it is known that B will be instantiated later than A, it is actually a good idea to transform the first into the second. However, in the absence of such information, one might expect the compiler not to do this. Still, B-Prolog does something like that, i.e. if the source program contains the first goal, then hProlog, Yap, SICStus also perform effectively the delay first on variable A. In B-Prolog on the other hand, the delay is put on the variable B. This is confirmed by inspection of generated abstract machine code and by benchmarking. As it turns out, queens_freeze and sort_freeze contain such doubly nested freezes and also sendmoney_freeze contains even deeper nested freezes [12]. We have done the measurements for queens_freeze and sort_freeze while reversing also the nesting of the freezes: the hProlog figure improves by about 15% and B-Prolog becomes slower by about 25%, thereby restoring the confidence in figures :-)

Still, it means that one has to be veeeery careful when benchmarking programs with freeze/2: we had instrumented the benchmarks at first, so as to be sure that there was the same number of freezes,

[12]the latter partly because we needed to transform the benchmark slightly since hProlog does not support X = 3+1, Y is X - 2, type of arithmetic

wakeups etc, but instrumenting a goal like $freeze(A, freeze(B, goal(A, B)))$ was done by replacing it by $myfreeze(A, myfreeze(B, goal(A, B)))$ with an appropriate definition of $myfreeze/2$; but this of course prevents B-Prolog from seeing the nesting and changing the order of delaying.

We end this section with three more remarks

1. Apart from time, freeze/2 also uses space. As B-Prolog freezes goals through stack frames, its memory requirements are different than those of systems like SICStus or hProlog: a melted goal can get trapped on the stack and without a stack collector, it cannot be re-used until failure. But Prolog implementations usually have a heap garbage collector: then, a melted goal can be garbage collected. This seems a good argument for using the heap based approach.

2. Attributed variables ([14]) are commonly used for implementing delayed goals: being frozen or not is just one possible attribute. Support for attributed variables can slow down delay. hProlog does not support attributed variables.

3. The bad performance of SICStus on the freeze benchmarks is mainly due to the slow meta call of SICStus: when benchmarks are rewritten so as to use the block declaration rather than freeze, SICStus behaves closer to hProlog. hProlog did not support block declarations when we started doing these tests and still does not fully so; however, preliminary (hand) testing indicates that there is little performance gain for block in hProlog, exactly because of its fast meta call.

13 hProlog and future work

hProlog is (just like ilProlog [2]) a descendant of dProlog ([8, 9]); the differences with dProlog are

- it only supports the heap_vars version
- the tagging schema has been changed (mainly because we needed an extra tag for delaying goals on the instantiation of variables)
- it no longer uses the XSB compiler for the generation of abstract machine code; instead, it uses the compiler written by Henk Vandecasteele, but with a few extra instruction compressions (some of which are described in [7])
- the double opcode schema of [5] has been definitely abandoned

hProlog was initially meant as an emulator back end to HAL ([]). This does not require hProlog to be a full Prolog system, e.g. it uses modules, but not the predicate based modules system of ISO Prolog; it doesn't need the whole I/O and many of the built-in predicates of the ISO standard. On the other hand, its own compiler is written in (h)Prolog and so hProlog is reasonably complete, definitely by Clocksin-Mellish standards.

As a backend to HAL, hProlog needs to support type classes and that is one reason to do meta calls (in different forms) well. HAL supports - like Mercury - modes, types and determinism declarations of predicates, but as a backend, hProlog can ignore these. Still, future research and implementation effort will go into exploiting such declarations for improved efficiency. For an early description of how to exploit types in a WAM context, see [15].

14 Conclusion

Most often benchmarks are used to prove superiority of one system over others. This was not our aim and even if hProlog turns out to be faster than to other systems for lots of benchmarks, we prefer the following message: within a context - e.g. being ISO - there are possibilities to improve [13] a given implementation; sometimes by a huge factor. Such an improvement might be *local* in the sense that a 10-fold speed-up of a particular feature will not result in a *global* 10-fold speedup of an application, but it is important nevertheless because the feature will become more attractive to use and so will be logic programming. Also, we might gain insight in the cost of a particular standard: the decisions in the ISO committee were often made without concern for efficiency, but also without information about the efficiency. We believe that a revision of the standard taking into account implementation experience is in order, in particular concerning meta predicates and modules.

Acknowledgements

Part of this work was conducted while the first author was a guest at the Institut de Mathematiques Appliquées of the Université Catholique de l'Ouest of Angers, France. Sincere thanks for this hospitality. We also thank the (main) authors/implementors of Prolog systems that come with sources: Joachim Schimpf (ECLiPSe), David S. Warren (XSB), Neng-Fa Zhou (B-Prolog), Daniel Diaz (GNU Prolog), Mats Carlsson (SICStus Prolog), Vitor S. Costa (Yap) and Jan Wielemaker (SWI-Prolog) and their teams, as well as Henk Vandecasteele and PharmaDM for letting us use the ilProlog compiler within hProlog.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[2] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, H. Vandecasteele. Executing Query Packs in ILP Proceedings of ILP2000 - 10th International Conference on Inductive Logic Programming, July 2000, London

[3] M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine.* PhD thesis, The Royal Institute of Technology (KTH), Stokholm, Sweden, Mar. 1990. See also: http://www.sics.se/isl/sicstus.html

[4] M. Carlsson. *Freeze, Indexing and Other Implementation Issues in the WAM.* Proceedings of the 4th International Conference on Logic Programming, Melbourne, 1987, pp 40-58

[5] V. S. Costa. *Optimising Bytecode Emulation for Prolog.* Proceedings of PPDP'99, LNCS 1702, Springer-Verlag, 261-277, September, 1999. See also http://www.ncc.up.pt/~vsc/Yap/.

[6] Bart Demoen *A 20' Implementation of Catch and Throw in WAM.* March 1989, CW report 96, K.U.Leuven

[7] B. Demoen, P.-L. Nguyen. 'On the impact of argument passing on the performance of the WAM and B-Prolog' K.U.Leuven, CW report 300, September 2000

[13] read: speed up

[8] B. Demoen, P.-L. Nguyen. *So many WAM variations, so little time.* Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings (V. John Lloyd, ed.), Lecture Notes in Artificial Intelligence, vol. 1861, Springer, 2000, pp. 1240-1254.

[9] B. Demoen, P.-L. Nguyen. *Experiments in WAM emulators and term representations.* CW report 283, Januari 2000 http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW283.abs.html

[10] Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, Peter Stuckey. "An Overview of HAL" Proceedings of the International Conference on Principles and Practice of Constraint Programming, Oct. 1999, Virginia, USA, pages 174–188, Springer Verlag

[11] D. Diaz and P. Codognet. *GNU Prolog: beyond compiling Prolog to C* Proceedings of the Second International Workshop, PADL 2000, Boston, MA, USA, January 2000. LNCS 1753, pp. 81-92 See also http://gprolog.inria.fr

[12] ECLiPSe: see http://www.icparc.ic.ac.uk/eclipse

[13] hProlog: see http://www.cs.kuleuven.ac.be/~bmd/hProlog

[14] Serge Le Huitouze. A new data structure for implementing extensions to Prolog. In Pierre Deransart and Jan Maluszynski, editors, *Programming Language Implementation and Logic Programming, 2nd International Workshop PLILP'90*, number 456 in LNCS, pages 136–150. Springer-Verlag, August 1990.

[15] Phuong-Lan Nguyen. *Optimisation du Code produit par un Compilateur Prolog.* Rapport de DEA, ENSIMAG, Laboratoire de Génie Informatique, Grenoble, 1988

[16] Sterling Shapiro The Art of Prolog, The MIT Press, 1986.

[17] SWI-Prolog: see http://www.swi.psy.uva.nl/projects/SWI-Prolog

[18] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

[19] N.-F. Zhou. On the Scheme of Passing Arguments in Stack Frames for Prolog" Proc. Eleventh International Conference on Logic Programming, MIT Press, pp.159-174, 1994 See also http://www.sci.brooklyn.cuny.edu/~zhou/bprolog.html

[20] N.-F. Zhou. A Novel Implementation Method for Delay Joint Internatinal Conference and Symposium on Logic Programming, pp.97-111, MIT Press, 1996