

What if [a] really equals $.(a,[])$?

Bart Demoen Phuong-Lan Nguyen

Report CW 351, October 2002



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

What if [a] really equals .(a,[]) ?

Bart Demoen *Phuong-Lan Nguyen*

Report CW351, October 2002

Department of Computer Science, K.U.Leuven

Abstract

The WAM traditionally optimizes for lists in two ways: by a list-specialized instruction set (LSIS) and by a list-specialized data representation (LSDR). These are partly orthogonal and their relative merit is unknown. In this paper, we perform an experiment within hProlog, which compares the schemas (LSIS+LSDR), (LSIS) and (). If no LSDR is used, it means that lists are treated as the syntactic sugar they stand for: terms with principal functor ./2. By not using LSDR, the implementation becomes more uniform and thereby invites more generally applicable optimizations. This leads to almost no performance loss, except in a couple of artificial benchmarks. Anyone attempting to just implement Prolog from scratch, should seriously consider not using LSDR. LSIS was found to be less effective than anticipated.

What if [a] really equals .(a,[]) ?

Bart Demoen * Phuong-Lan Nguyen †

October 28, 2002

Abstract

The WAM traditionally optimizes for lists in two ways: by a list-specialized instruction set (LSIS) and by a list-specialized data representation (LSDR). These are partly orthogonal and their relative merit is unknown. In this paper, we perform an experiment within hProlog, which compares the schemas (LSIS+LSDR), (LSIS) and (). If no LSDR is used, it means that lists are treated as the syntactic sugar they stand for: terms with principal functor ./2. By not using LSDR, the implementation becomes more uniform and thereby invites more generally applicable optimizations. This leads to almost no performance loss, except in a couple of artificial benchmarks. Anyone attempting to just implement Prolog from scratch, should seriously consider not using LSDR. LSIS was found to be less effective than anticipated.

1 Introduction

The WAM (see [4] and [1]) optimizes lists in two ways: by having a set of instructions that are specialized for the list constructor (LSIS), and by having a special run time representation for lists (LSDR). Taken together, they improve both speed and memory consumption of Prolog programs that use lists a lot: there is some explicit encouragement to use lists because certain builtin predicates deliver lists as output (`sort/2`, `findall/3`, `=../2`, ...) or are defined on input that is a list (`sort/2`, `=../2`, ...). LSDR without LSIS is possible of course, but it seems a weird combination and it is probably not a good choice for performance. On the other hand, one could have LSIS without LSDR: BinProlog does so. Not everything improves by adopting LSDR, in particular, the tagging schema in the original WAM, as implemented on a 32-bit machine, uses 25% of the tags just for lists. The WAM tagging schema uses the two tag bits as follows

- 00 for a reference (including the self reference)
- 01 for a structure pointer
- 10 for a list pointer
- 11 for atomic data

*Dept. of Computer Science, K.U. Leuven, Belgium, bmd@cs.kuleuven.ac.be

†Inst. de Mathématiques Appliquées, UCO, Angers, France, nguyen@ima.uco.fr

Newer implementations of Prolog often provide other *types* as well: attributed variables, long integers, floating point numbers. Some of these are atomic from the point of view of Prolog, but at the implementation level they are not. For such implementations, more tag bits are needed and as a result often the range of addresses of heap (and local stack) words is limited. For instance in hProlog, the heap must fit in the lower half of the available 4Gb of memory. Other systems have even more stringent limitations. It then becomes clear that dedicating a tag to lists, conflicts with other interests: a tight memory limitation is a real problem for serious applications, and since the tag decoding logic becomes more complicated, performance might suffer. However, there is no evidence that the latter is true, so we have set ourselves the goal to show that a reasonably good WAM implementation like hProlog, does not need to suffer excessively by throwing away LSDR. Our experiments also show that LSIS is not very effective: we had expected more from it.

The setting in which we do this is hProlog 1.8: we make a new version which is derived from the old one, by suppressing the special treatment of the `./2` functor - which is syntactic sugar for the list notation - by the compiler and by the runtime system. This version is referred to later as *hProlog_nolist*. hProlog_nolist has gone through a series of versions itself and some of them are described in Section 3. Section 4 gives the results of the benchmarks, and Section 5 completes the picture with some cache profiling results.

In principle, one could transform any occurrence of a list in a program to a term with binary principal functor different from the dot and that would then give the effect of a system without LSDR (and without LSIS). However, this is impractical: some builtins return lists (`=./2`, `endall/3` ...) or rely on their input being a list (`sort/2` ...). So the source transformation approach works only for a very limited set of benchmarks. We have therefore chosen the approach to change the underlying implementation - which is also the only way to make a fair comparison.

2 Fringe benefits of abandoning LSDR

We just mention without motivation the fringe benefits of abandoning LSDR: it is clear that a whole lot of experiments can be done to confirm (or disprove) some of these claims.

- one tag becomes available for more interesting purposes
- support for rational trees becomes easier
- garbage collection becomes more *safe* (see [2])

A more uniform instruction set (obtained by suppressing also LSIS) makes some instruction merging more attractive and more widely effective. However, this turned out not to outweigh the effect of specialization of the instruction set for the functor `./2`.

3 Different versions of hProlog_nolist

3.1 Making hProlog_nolist

The first step was to change the compiler so that no list-related instructions are generated. This was easy, as in the first phase of the compiler, it transforms the parse tree into an abstract syntax tree, which makes lists explicit: suppressing this was enough in the compiler.

The second step consisted in adapting the builtins that either expect lists or return lists. This involved changing the define for some macro's. E.g.

```
#define is_list(p) tag(p) == LIST
```

became

```
#define is_list(p) \
    is_struct(p) && *(get_struct_pointer(p)) == dot_2
```

and

```
#define make_list(p) (((dlong)(p) << 1) | LIST) (*)
```

in a C fragment like: `*x = make_list(h);` into

```
#define make_list(p) ((dlong)(p) | STRUCT)
```

with corresponding fragment:

```
*x = make_list(h); *h = dot_2; h++;
```

Such changes are a bit tedious, but easy to get right. The left shift in (*) has disappeared: it was there because hProlog1.8 needs an extra tag since it supports floating point numbers and frozen variables which have their own tag.

These changes resulted in hProlog_nolist: it does not have LSIS neither LSDR.

3.2 Introducing LSIS

Introducing LSIS was done at two levels: the loader turns some *structure* instructions with as structure dot_2 into the corresponding *list* instruction - we did this for the get_structure and put_structure instructions. This results in code that is close to what the original compiler generates¹. There are small differences because the compiler knows about the size of a cons cell. The main difference however was that the original compiler generates an instruction switchonlist_skip - which speeds up nrev quite a bit - and there was no switchonterm_skip yet. So we had to introduce it - by a peephole pass - and then let it be transformed to the switchonlist_skip. The result is named hProlog_nolist(LSIS).

3.3 One more little trick

We have in the code above used *dot_2*: at the C-level, one can think of a (global) variable that is initialized (at startup time) to the appropriate handle for *dot_2*. Implementing *dot_2* as a variable results in a memory access every time *dot_2* is used and also in longer assembler instructions. One can alternatively make sure that the value of *dot_2* is known at compile time, e.g. by running the system and printing it out and then using this value in subsequent runs. Then code will look like `*(get_struct_pointer(p)) == 179` if 179 happens to be the value of *dot_2*. For this to work, the

¹the other structure instructions occur quite infrequently

handle for `.l2` should be constant across runs of course. One has an interest in keeping this value small. We have chosen to make `.l2` the first functor that is ever put in the functor table, resulting in the value 19, which further reduces the length of the assembler instructions needed to implement operations referring to `dot.l2`. We do not report on the effect of this trick here: it is just part of the LSIS schema.

4 Benchmark results

We are interested in time and space for a set of benchmarks: these were classified roughly and a priori into benchmarks that manipulate lists a lot, benchmarks that manipulate lists very little and benchmarks that are more a mixture ². Performance wise, one expects to see the biggest (negative) impact of not having LSDR in the first category, none (or a small positive one) in the second and a small impact in the third category. Space wise, similar comments apply. For `hProlog_nolist` and `hProlog_nolist(LSIS)`, we just give the relative difference to the original `hProlog` (which includes LSDR and LSIS) in %, i.e.

$$100 * (time(hProlog_nolist) - time(original_hProlog)) / time(original_hProlog).$$

Each benchmark was run with enough initial heap, trail ... space, so that no garbage collection or expansions were triggered. The experimental evaluation was performed on a Pentium III 266MHz with 256Mb RAM. The code size also is influenced by the choice between the three systems. The code in `original_hProlog` and `hProlog_nolist(LSIS)` is virtually the same ³. The code size in `hProlog_nolist(LSIS)` is about 1.3% smaller than in `hProlog_nolist` measured over the whole benchmark set.

Table 1 shows that the impact of abandoning LSDR is very pronounced for the `list` and `nrev` benchmark - this is as expected. In most of the other benchmarks that we judged a priori to be list-intensive (and consequently expected a big slowdown), we found a slowdown of 10% up to almost none.

	list	nrev	poly_10	browse	crypt	ham	queens	reducer	zebra
nolist	28	20	14	8	7	4	3	3	2
lsis	25	18	9	9	7	5	1	3	1

Table 1: Performance difference on the list-intensive benchmarks

Table 2 shows that abandoning LSDR and in the course of doing so introducing some reasonable instruction specialization and merging, we get equally high performance *gains* for some benchmarks, but most often it is a break even. We find the figure for the benchmark `comp` most significant: it is the only real application (still of smaller than medium size) and there was no noticeable performance loss by giving up LSDR.

²or about which we have no idea

³indexing is influenced in unpredictable ways

	snrev	struct	boyer	cal	send	chat	meta_qsort	sdda	comp
lsls	-21	-10	1	-1	0	-2	-1	-1	0
lsls	-21	-10	0	-1	0	-2	-1	-1	0

Table 2: Performance difference on the non list-intensive benchmarks

It is clear that even with a specialized instruction set for lists, abandoning LSDR must lead to a higher memory foot print, i.e. a higher heap consumption. We have measured this to be between almost 50% (as expected for list) to 0% for boyer - for comp it was about 20%, but for comp with little effect on performance.

5 The cache

In order to get some insight in the difference between (LSDR+LSIS) and nolist, we measured the cache behavior for the one larger benchmark *comp*. This was done with *cachegrind version 1.0.3*, which is Copyright (C) 2000-2002 Julian Seward⁴. We simply ran

```
cachegrind ./hProlog
vg_annotate --auto=yes
```

Table 3 gives the instruction reads (Ir), the data reads (Dr) and the data writes (Dw), always with their level 1 and 2 misses. They are given for the total of the run, and for the part of the execution spend in the emulator (named machine) and the unify procedure (which has a prelude and a recursive part). As expected, more instructions are needed in nolist and also more data reads and writes. But the number of cache misses is not accordingly higher. This might explain partly why the 20% higher memory foot print for comp does not degrade performance; also, the emulator overhead (with its many unpredictable jumps) remains the same.

	Ir/10 ³	I1mr	I2mr	Dr/10 ³	D1mr	D2mr	Dw/10 ³	D1mw	D2mw
nolist									
total	1,910,348	559,866	5,350	824,715	7,323,052	80,078	355,105	834,635	226,589
machine	1,391,272	432,256	2,199	667,816	5,551,320	59,428	289,325	704,472	209,892
unify	178,501	22,006	93	42,227	672,131	2,598	23,133	9,465	1,236
unify_rec	108,141	13,293	46	21,431	222,405	79	11,842	6,140	443
original									
total	1,777,909	325,872	3,450	682,613	6,738,702	72,343	283,805	631,126	194,628
machine	1,288,350	217,534	1,355	542,329	4,839,922	53,211	222,637	562,601	178,024
unify	175,201	10,922	59	37,753	991,881	3,295	23,241	10,489	1,130
unify_rec	107,092	18,269	38	23,468	226,426	276	11,960	7,448	506

Table 3: Cache

⁴<http://developer.kde.org/~sewardj/>

6 Final comments

Changing a tagging schema in an existing system is usually not an option - although it was quite easy in the case of hProlog. So the value of this work is clearly not in persuading people to throw away LSDR. Also, in a native code implementation, the performance difference can be expected to be larger: since there is no emulator overhead, the extra instructions executed and the cache misses in nolist most certainly will show up. But it shows that by ignoring lists during the development of a Prolog system, no unreasonable penalty needs to be incurred.

The most important result for us was that on the more or less realistic benchmark - the compiler compiling itself - there is no meaningful difference in the speed between a system with LSIS+LSDR and a system without, even though the compiler uses lists itself ⁵: typical non-trivial applications indeed use lists and other terms in a rather balanced way. The other benchmarks seem to indicate that LSIS is not so effective as LSDR. This research was a necessary step for us in understanding how to use type information in hProlog: this type information is present in HAL programs. Mercury [3] has already shown the way of course, with a specialized data representation for each type. We needed to gain experience with these issues in the emulator context.

Acknowledgements

Part of this work was performed while the first author was a guest at the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest of Angers, France. Sincere thanks for this hospitality. We also thank the Henk Vandecasteele for his work on the ilProlog compiler used within hProlog.

References

- [1] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990 See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] B. Demoen, P.-L. Nguyen, and R. Vandeginste. Copying garbage collection for the WAM: to mark or not to mark ? In P. Stuckey, editor, *Proceedings of ICLP2002 - International Conference on Logic Programming*, number 2401 in Lecture Notes in Computer Science, pages 194–208, Copenhagen, July 2002. ALP, Springer-Verlag.
- [3] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.
- [4] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

⁵it is a version of the XSB compiler that was also used in dProlog