

Allocating WAM Environments/Choice Points on the Heap

Bart Demoen, Phuong-Lan Nguyen

Report CW455, July 2006



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

Allocating WAM Environments/Choice Points on the Heap

Bart Demoen, Phuong-Lan Nguyen

Report CW 455, July 2006

Department of Computer Science, K.U.Leuven

Abstract

The WAM keeps environments on a stack. As an alternative, we explore allocating them on the heap, in the context of hProlog. The changes needed to make this work are described, as well as the modified optimizations. Benchmarking shows that this is not a totally wacky idea. We do the same for WAM choice points.

Allocating WAM Environments/Choice Points on the Heap

Bart Demoen* and Phuong-Lan Nguyen†

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

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

Abstract. The WAM keeps environments on a stack. As an alternative, we explore allocating them on the heap, in the context of hProlog. The changes needed to make this work are described, as well as the modified optimizations. Benchmarking shows that this is not a totally wacky idea. We do the same for WAM choice points.

1 Introduction

Warning: this paper contains material that can be insulting to people not trained in the art of Prolog implementation: you should understand Prolog [4] and the WAM [1, 10], and know some things about concrete implementations. By reading further, you accept these conditions.

We are interested in exploring a particular alternative for the runtime data structures in the WAM. This interest stems from working towards an implementation of suspension frames (see [11]) in the WAM, by keeping the suspension frames on the heap: keeping them on the control stack as in B-Prolog complicates too much the implementation, and we want to keep it simple, as well as flexible. This paper reports on an experiment in which we got rid of the separate environment stack. Instead we allocate the environments on the heap. We started off from hProlog2.6 (referred to as the *original* later on) and modified it to a version where environments are kept on the heap. We refer to that version later as *env_on_heap*. After we finished the *env_on_heap* implementation, we performed the same experiment for choice points. This resulted in a version named *cp_on_heap*. We were able to achieve both variants with rather few changes to our WAM implementation. The changes for *env_on_heap* are described in Section 2, those for *cp_on_heap* in Section 5. Performance is quite good for *env_on_heap* and reported on in Section 4. Section 6 shows the performance of *cp_on_heap*. Section 3 discusses briefly last call optimization for *env_on_heap*. Related work can be found in Section 7. We conclude in Section 8. We start with a short terminology paragraph:

Terminology: We use *heap* where other sources use *global stack*: it is the area in which the WAM keeps runtime Prolog terms, lists etc. We use the terms environment stack and choice point stack: the WAM intertwines these two stacks in one consecutive area often referred to as the local stack. hProlog (see [5]) uses (just like SICStus Prolog, XSB ...) two separate stacks for the environments and for the choice points. We use the following abbreviations for particular WAM registers:

- E points to the start of the current environment
- B points to the start of the most recent choice point
- H points to the first free cell on the heap

2 Changes to our WAM for `env_on_heap`

2.1 Changes to the data structures

The most important change is of course the location of the environments. Even on an environment stack, the set of environments forms a tree in which children point to their parent. This tree is maintained, but is spread over the heap in `env_on_heap`. As a result, E and the E slot in a choice point, points to the heap.

hProlog has the top of the environment stack (TOES) as an extra WAM register; it is stored in choice points and restored on backtracking. TOES indicates the location at which the allocate instruction can start laying out a new environment. In `env_on_heap`, TOES is no longer needed. This saves machine instructions, and a slot in choice points.

2.2 Changes to the generated code

- The hProlog compiler moves the allocate instruction as far as possible down the instruction stream of a clause. E.g. for the clause `p([X]) :- foo, bar(X).` the normal code would be:

```
getlist A1
allocate Y3
unipvar Y2
unify_atom []
call foo/0 (user)
putpval Y2 A1
deallex bar/1 (user)
```

In this way, failure might happen before the allocate is executed (in case `p/1` is called with `[]` as first argument for instance). However, the above code is wrong in `env_on_heap`, because if `p/1` is called with a free argument, the `getlist` instruction is executed in write-mode, the S pointer is set to the top of the heap, and the allocate will put the environment there. This is clearly wrong. We have switched off the moving of the allocate in the compiler. This leads to suboptimal code but see next item.

- For a predicate like

```
p([]).  
p([X|R]) :- q(X), p(R).
```

we used to generate the following code:

```
switchonlist_skip A1 @var, @con, @lis  
@var: try_me_else arity = 1 @alt  
@con: get_atom A1 []  
      proceed  
@alt: trust_me_else arity = 1  
      getlist A1  
@lis: unitvar A1  
      allocate 3  
      unipvar Y2  
      call q/1 (user)  
      putpval Y2 A1  
      deallex p/1 (user)
```

switchonlist_skip performs the action of getlist if it is in read-mode, so it can be skipped. This relies on postponing the allocate, but as pointed out above, this is wrong. We have modified the compiler and instruction set so that we now generate:

```
switchonlist_skip_alloc A1 3 @var, @con, @lis  
@var: try_me_else arity = 1 @alt  
@con: get_atom A1 []  
      proceed  
@alt: trust_me_else arity = 1  
      allocate 3  
      getlist A1  
@lis: unitvar A1  
      unipvar Y2  
      call q/1 (user)  
      putpval Y2 A1  
      deallex p/1 (user)  
      end_clauses
```

The working of switchonlist_skip_alloc should be clear.

2.3 Changes to the instructions and instruction set

There is surprisingly little to change.

- allocate: TOES is no longer needed; its role is taken over by H; environment stack overflow needs no longer to be tested; the heap overflow test at allocate

is not needed, because it is done at the first call in the body: this is safe because the size of environments has an upper bound - even without an upper bound, the compiler can emit extra heap overflow checks when needed, as is the case with very big heads already

- deallocate: it needs no longer to recompute the TOES; some related instructions are changed in the same spirit
- call: since hProlog does not perform environment trimming, this instruction needs no change; otherwise, environment trimming should be switched off
- switchonlist_skip_alloc: a new instruction; see Section 2.2.

2.4 Changes for and to the garbage collector

In the original version of hProlog, we never needed the length of an environment (except when allocating it). In `env_on_heap` this is different: even though the collector only relocates environment cells that are reachable - they have a reliable contents - the whole environment needs to be moved as a block if the environment is still alive. Therefore the length of the environment needs to be known. The easiest method is to put this length on the heap just before pushing the environment: `E` points one cell further than the cell containing the length. In this way, only the garbage collector needs to be aware of this cell. The benchmarks were of course run with this extra cell. What follows is a loose description of changes to the gc code needed, but not actually performed: since a complete deployment of `env_on_heap` was not intended, there was no real need for that.

hProlog uses currently a mark© collector with preservation of segments as described in [9]. The mark bits are kept in arrays: one for the heap and one for the environment stack. Apart from the bit that indicates whether a cell is alive, we also keep information about whether a cell is trailed and whether the cell needs relocation on copying. We use actually a whole “mark” byte (instead of the 3 bits we strictly need) for each cell. We can use some of the other five bits now.

- one of these bits is used to indicate that the cell belongs to an environment; in this way, we can make sure that all slots of an environment are copied
- during the copying phase, the collector finds for a particular marked cell its enclosing marked block (see [3] for details); the bit just mentioned is taken into account at that moment
- one bit is used to indicate that the environment was marked before; this saves traversing already visited parts of the environment tree: this bit (as well as the mark bit for environment slots) was originally allocated in an array dedicated to the environment stack; in `env_on_heap`, it is integrated in the heap mark array

Since we haven't actually implemented all this, we can not give any empirical data, but we expect that the locality of reference improves.

2.5 Trailing

It might seem strange to consider trailing an issue worth mentioning. However, hProlog never needs to trail an environment cell, because free variables are always allocated on the heap. This means that hProlog doesn't need the `put_unsafe` instruction: a ref pointer never points into the environment stack.

Since in `env_on_heap` the environment is on the heap, we could reconsider this. Still, as long as we don't implement any form of LCO (see Section 3), we do not need the `put_unsafe` instruction. Another advantage of `env_on_heap` is that the trailing test for environment variables is exactly the same as for heap variables.

3 Last call optimization

It is clear that LCO is more difficult to achieve in `env_on_heap`. Also, it is not very often applicable, because an environment tends to become trapped by the heap terms the clause has produced. OTOH, very often it can be decided at compile time that LCO will be effective: this is the case when in a deterministic stretch of execution. From the space consumption point of view, it could be worth to put those environments on a stack. This would complicate the compiler, the engine and the garbage collector.

Another approach would be to reuse a deallocated environment even when it is trapped. Take for instance the following typical code

```
p([], []).
p([X|R], [Y|S]) :- process(X,Y), !, p(R,S).
```

We have put the cut explicitly, but if it is known that `process/2` does not leave a choice point, the cut could as well not be there.

With a little help from the compiler, code could be generated that allocates one environment (on the heap) and uses it for the whole execution of `p/2`. In fact, generating such code could (should) also be done in the plain WAM setting when all environments are on the stack.

4 Benchmarks for `env_on_heap`

The benchmarks were all run on a Pentium, 1.8GHz, under Debian, and while running the benchmarks, garbage collection was disabled by starting the systems with enough initial memory.

4.1 Time

A classical set of benchmark programs was run with both systems. We report on the time to run the benchmarks and the extra column in the table gives the percentage extra time needed by `env_on_heap`, i.e. $((time_{env_on_heap} -$

$time_{original})/time_{original}) * 100$. A priori we didn't expect to see a big neither a systematic difference. Table 1 seems to indicate a slight edge for the env_on_heap system: negative numbers are good for env_on_heap. We think that percentages between -5 and 5 should be interpreted as noise.

benchmark	original	env_on_heap	% extra
boyer	3898	3980	2
browse	4066	3710	-8
cal	3875	3882	0
chat	2742	2731	0
crypt	4021	3686	-8
ham	4005	3895	-2
meta_qsort	4053	3847	-5
nrev	2144	2224	3
poly_10	4275	3823	-10
queens_16	5340	5028	-5
queens10	5517	5313	-3
reducer	3301	3415	3
send	2996	3133	4
tak	3094	3022	-2
zebra	3521	3558	1

Table 1. Timings in milliseconds

4.2 Space

Table 2 shows for the original system and the env_on_heap system and for each benchmark, the high watermark space usage in words (of 4 bytes) of the heap, the trail, the choice point stack, the environment stack and the sum of those. For ease of comparison, there is column showing the percentage extra time needed.

We expect to see the following trends:

- the heap space increases
- the trail usage does not change
- the choice points take up less space in env_on_heap
- the environment stack is empty in env_on_heap
- the total memory usage is higher

Table 2 basically confirms this. There is one outlier: boyer consumes about 5 times as much memory in env_on_heap. The reason is in the recursive clause:

```
rewrite_args(N,Old,Mid) :-
  arg(N,Old,OldArg),
  arg(N,Mid,MidArg),
  rewrite(OldArg,MidArg),
```

N1 is N-1,
rewrite_args(N1,Old,Mid).

The call to rewrite/2 produces heap terms which block the environment: this phenomenon occurs in many benchmarks, but is most pronounced in boyer. It is clear that boyer would benefit a lot from some form of LCO (see Section 3).

Some benchmarks consume a little less: this difference is due to the smaller choice point size in env_on_heap.

benchmark	heap	trail	choice points	env stack	total	% extra space	system
boyer	183234	2	130	386	183752		original
	1094258	2	112	0	1094372	495	env_on_heap
browse	11360	1210	4829	1328	18727		original
	24584	1210	4426	0	30220	61	env_on_heap
cal	5	4	56	15	80		original
	24	4	50	0	78	-2	env_on_heap
chat	1191	610	1044	858	3703		original
	2427	610	965	0	4002	8	env_on_heap
crypt	61	10	51	23	145		original
	86	10	45	0	141	-2	env_on_heap
ham	579	80	387	146	1192		original
	839	80	346	0	1265	6	env_on_heap
meta_qsort	5238	1134	2492	2050	10914		original
	13158	1134	2215	0	16507	51	env_on_heap
nrev	1022	1	7	153	1183		original
	1206	1	6	0	1213	2	env_on_heap
poly_10	57460	2	27	108	57597		original
	112219	2	24	0	112245	94	env_on_heap
queens_16	202	66	177	101	546		original
	318	66	159	0	543	0	env_on_heap
queens10	160	62	125	89	436		original
	503	62	112	0	677	55	env_on_heap
reducer	22731	60	259	605	23655		original
	51436	60	233	0	51729	118	env_on_heap
send	11	14	70	15	110		original
	28	14	61	0	103	-6	env_on_heap
tak	47707	95414	524784	143123	811028		original
	206734	95414	477076	0	779224	-3	env_on_heap
zebra	149	178	137	5	469		original
	156	178	122	0	456	-2	env_on_heap

Table 2. The size of the stacks in words

Note that there is a little (space) overhead in the benchmark test suite which explains for instance the small choice point consumption for nrev. Also note

that the benchmarks with small memory consumption should not be taken too seriously.

5 Changes to our WAM for `cp_on_heap`

5.1 Changes to the data structures

The most important change is of course the location of the choice points: any time a choice point is made, it is put at the top of the heap. B now points to the heap.

5.2 Changes to the instructions

There is surprisingly little to change.

- `try`: Allocation of a new choice point happens on the top of the heap instead of on the choice point stack: as in `env_on_heap`, overflow checking is not needed, as it is done by the first call instruction and because the length of choice points is limited (and with the usual cop out). It is important to get the H pointer in the choice point correct: the choice point is conceptually still in the old segment, so the H pointer in the choice point must point just after the choice point.
- `trust`: This is specific to hProlog where some terms (values of global variables) survive backtracking: `trust` (and `tetry`) instructions did cater for this already, but the `trust` instruction can recover the heap space occupied by the choice point if such global variables are not preventing this recovery. The `trust` instruction needs new code for that.
- `cut`: Similar to `trust`, the `cut` might recover the space of a (cut away) choice point, but it needs to check whether the choice point is on top of the heap: apart from the hProlog global variables, this recovery could be prevented by non-steadfast code (see Section 6.2).

6 Benchmarks for `cp_on_heap`

6.1 Time

A classical set of benchmark programs was run with both systems. We report on the time to run the benchmarks and the extra column in the table gives the performance extra of `cp_on_heap`, i.e. $((time_{cp_on_heap} - time_{original}) / time_{original}) * 100$. Not taking into account differences that are too small, the trend seems to be that `cp_on_heap` is slower.

benchmark	original	cp_on_heap	% extra
boyer	3898	4056	4
browse	4066	4584	12
cal	3875	3986	2
chat	2742	2787	1
crypt	4021	3713	-7
ham	4005	3912	-2
meta_qsort	4053	4143	2
nrev	2144	2169	1
poly_10	4275	4295	0
queens_16	5340	5894	10
queens10	5517	5511	0
reducer	3301	3417	3
send	2996	2860	-4
tak	3094	3063	-1
zebra	3521	3483	-1

Table 3. Timings in milliseconds

6.2 Space

Table 4 shows for the original system and the cp_on_heap system and for each benchmark, the high watermark space usage in words (of 4 bytes) of the heap, the trail, the choice point stack, the environment stack and the sum of those. For ease of comparison, there is column showing the ratio of these totals as a percentage: more than 100 means cp_on_heap uses more than the original.

We expect to see the following trends:

- the heap space increases
- the trail stack usage does not change
- the environment stack usage does not change
- the choice points take up less space in cp_on_heap
- the total memory usage can go either way

Table 4 confirms all expectations.

There are three outliers: poly_10 and reducer show a much higher memory footprint; queens_16 uses (slightly) less space. We explain the latter first.

Take the predicate

```
p :- useheap(<n>).
p :- useheap(<m>).
```

with $\langle n \rangle$ and $\langle m \rangle$ some fixed numbers. Useheap/1 consumes as many heap cells as its argument. The size of a choice point is 7 (+ the number of arguments of the predicate). The amount of space used in original is $7 + \max(n, m)$ while in cp_on_heap it is $\max(7 + n, m)$. For $n = 1, m = 10$, original consumes more space than cp_on_heap. This explains the figures for queens_16.

benchmark	heap	trail	choice points	env stack	total	% extra space	system
boyer	183234	2	130	386	183752	3	original
	190059	2	0	386	190447		cp_on_heap
browse	11360	1210	4829	1328	18727	6	original
	17313	1210	0	1328	19851		cp_on_heap
cal	5	4	56	15	80	0	original
	61	4	0	15	80		cp_on_heap
chat	1191	610	1044	858	3703	0	original
	2235	610	0	858	3703		cp_on_heap
crypt	61	10	51	23	145	0	original
	112	10	0	23	145		cp_on_heap
ham	579	80	387	146	1192	0	original
	966	80	0	146	1192		cp_on_heap
meta_qsor	5238	1134	2492	2050	10914	0	original
	7730	1134	0	2050	10914		cp_on_heap
nrev	1022	1	7	153	1183	0	original
	1029	1	0	153	1183		cp_on_heap
poly_10	57460	2	27	108	57597	96	original
	112940	2	0	108	113050		cp_on_heap
queens_16	202	66	177	101	546	-1	original
	372	66	0	101	539		cp_on_heap
queens10	160	62	125	89	436	0	original
	285	62	0	89	436		cp_on_heap
reducer	22731	60	259	605	23655	74	original
	40559	60	0	605	41224		cp_on_heap
send	11	14	70	15	110	0	original
	81	14	0	15	110		cp_on_heap
tak	47707	95414	524784	143123	811028	0	original
	572491	95414	0	143123	811028		cp_on_heap
zebra	149	178	137	5	469	0	original
	286	178	0	5	469		cp_on_heap

Table 4. The size of the stacks in words

We noted already that a cut can trap an unreachable choice point when heap has been used since the choice point was layed out. This happens in particular in non-steadfast code, e.g.:

```
p(X,term(1,2,3)) :- check(X), !.  
p(X,foo).
```

when called in mode (+,-) and with a first argument that satisfies the check, consumes more heap in `cp_on_heap` than

```
p(X,Out) :- check(X), !, Out = term(1,2,3).  
p(X,foo).
```

Such non-steadfast code is omnipresent in `poly_10` and `reducer`. We have rewritten `poly_10` so that its predicates become steadfast, and the extra space needed drops to zero. The positive (but small) extra space needed in `boyer` (and some other benchmarks) is also due to this fenomemon, but the extent is smaller because the non-steadfast predicates do not form the bulk of the execution.

Note that the changes to `trust` and `cut` are essential for most benchmarks to consume no extra space: we have observed that before adding the code for the recovery of choice points on top of the heap, `boyer` consumes about 10 times more space in `cp_on_heap`.

7 Related work

The BinWAM [8] doesn't have environments at all: their equivalent is stored as heap terms representing a continuation and meta-called. Still BinProlog has a good performance but it needs to rely on garbage collection. The similarity between the BinWAM and `env_on_heap` is clear. The BinWam, `env_on_heap` and `cp_on_heap` are all very much in the spirit of early work by A.W. Appel [2] which defends allocating objects with a FILO lifetime on a garbage collected heap instead of on a stack. At least one benchmark (`boyer`) suggest that it might be worth exploring a mixture of environments on the stack allocated and heap allocated environments: if analysis can prove that LCO is effective (i.e. that the last goal is reached without alternatives pending in the body) than stack allocation could be better. However, overall implementation would become more complicated. There is a relation with the `det` and `nondet` stack of Mercury here, but we leave it unexplored at the moment.

We have experimented earlier with an alternatives for E (and B) in the WAM in [7]. However, those alternatives only changed the layout of those frames and kept them on the stack.

In [6], a merged heap/stack architecture for Prolog is described: the heap contains all the control frames - in WAM terminology the environments and the choice points. Since the implementation mentioned in that paper comes as a whole package (with a different term representation, memory management, instruction set ...), it does not give insight in the effect of just moving environments or choice points from the stack to the heap: our experiment isolates exactly this while keeping all other things equal.

8 Discussion

We were nicely surprised to find out that the changes needed to put environments/choice points on the heap were localized and easy. We found the necessity for some changes by debugging of course, but even that was not too painful. The most difficult changes were actually in the heap printing routine and the setup of the memory benchmark suite.

One could explore new optimization opportunities that are the consequence of putting environments or choice points on the heap. We have mentioned that within `env_on_heap`, we could change the policy of never having an `undef` in an environment. Also `undefs` in choice points could be allowed if choice points are on the heap. Allocating structured terms in an environment also becomes an option when environments are in the heap: one no longer needs to be afraid of pointers from the heap to an environment. This is just scratching the surface and we hope someone will explore this further.

Time wise, keeping environments on the heap seems a better decision than keeping the choice points on the heap: `cp_on_heap` suffers from extra code to be executed in order to recover (heap) topmost redundant choice points. On the other hand, `cp_on_heap` cannot lose space wise against original for well written Prolog code. Still, our main objective was to gain experience with putting WAM environments on the heap. In view of our aim, the experiments reported on in this paper were very helpful.

Acknowledgements

Part of this work was conducted while the first author was a guest at the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest in Angers, France. Sincere thanks for this hospitality. We also thank Henk Vandecasteele for maintaining the hipP compiler which we use within hProlog.

References

1. H. Aït-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.
2. A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.
3. J. Beveymyr and T. Lindgren. A simple and efficient copying Garbage Collector for Prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 88–101. Springer-Verlag, Sept. 1994.
4. W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.
5. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *CL2000: Proceedings of the 1st International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 1240–1254, Londong, UK, July 2000. ALP, Springer Verlag.

6. X. Li. Efficient Memory Management in a merged Heap/Stack Prolog Machine. In *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 245–256. ACM Press, 2000.
7. A. Mariën and B. Demoen. On the management of choicepoint and environment frames in the WAM. In *Logic Programming, Proceedings of the North American Conference 1989*, pages 1030–1047. MIT Press, Oct. 1989. URL = <http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ.info.pl?id=27817>.
8. P. Tarau. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In A. Voronkov, editor, *Russian Conference on Logic Programming*, number 592 in *Lecture Notes in Artificial Intelligence*, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag.
9. R. Vandeginste, K. Sagonas, and B. Demoen. Segment order preserving and generational garbage collection for Prolog. In S. Krishnamurthi and C. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Proceedings*, volume 2257 of *Lecture Notes in Computer Science*, pages 299–317. Springer, 2002.
10. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.
11. N.-F. Zhou. A Novel Implementation Method for Delay. In *Joint Internatinal Conference and Symposium on Logic Programming*, pages 97–111. MIT Press, 1996.