ISRN SICS/R--91/07--SE

Variable Shunting for the WAM

by

Dan Sahlin and Mats Carlsson

SICS research report
R91:07
ISSN 0283-3638

ISRN SICS/R--91/07--SE

Variable Shunting for the WAM

Dan Sahlin and Mats Carlsson

March 1991

SICS Research Report R91:07
ISSN 0282-3638
Swedish Institute of Computer Science
Box 1263, S164 28 Kista, Sweden

Variable Shunting for the WAM

Dan Sahlin and Mats Carlsson

Abstract

This paper describes how to extend the garbage collection for WAM [ACHS88] so that it will shunt chains of bound variables if possible. Doing so has two advantages:

1. Space is saved by making it possible to deallocate the intermediate cells. This is particularly useful when those cells are associated with frozen goals.

2. Later dereferencing is speeded up by not having to follow long variable chains.

The main complication of this optimization is the treatment of the trailed variables. We claim that all possible chains of variables are shunted by this algorithm.

The algorithm has been implemented in SICStus Prolog, and benchmark results are presented in this paper.

This paper is a revised version of [Sahlin89] and is meant to be read in conjunction with [ACHS88] as the notation used is presented there and only briefly summarized here. The full source code for the shunting algorithm is given in this paper.

An example

Chains of variables are unavoidable in an ordinary WAM execution. For instance, the query `p(1000,s(A),B)` given to the program below, will create a variable chain starting in `B` having 1000 links.

```
p(0,s(Z),Z).
p(N,s(X),Z) :- N>0, N1 is N-1, p(N1,Y,Z), Y=s(X).
```

During WAM execution a number of value cells containing various instances of `s(X)` will be created first. Then the chain of `X`'s will be linked so that the last one will point to the next to last, the next to last will point to the third from the end, etc. Thus these variable chains may occur in normal WAM execution.

If these intermediate cells are "short-circuited" (shunted), subsequent dereferencing of the variable `B` in the query will not have to traverse the intermediate cells. Thus execution speed is improved.

If the intermediate cells are not referenced from anywhere else, they may also become deallocated. Thus more storage can be reclaimed.

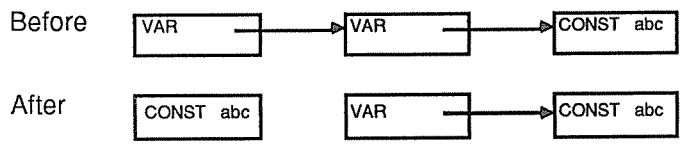
Invocation

Variable shunting is an independent algorithm, and may take place either before or after garbage collection. As the whole heap is traversed it seems advantageous to do the shunting *after* garbage collection, as the heap will only contain reachable value cells at that stage. The disadvantage is of course that the value cells which are no longer referred to after the shunting, will not be freed until the next garbage collection. Furthermore, if the segmented garbage collection optimization is used, it is pointless to perform shunting after garbage collection as this part of the memory will not be treated in the next garbage collection.

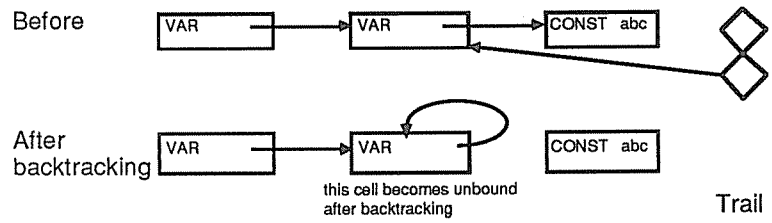
In the sequel we shall therefore assume that variable shunting is performed immediately prior to garbage collection.

The basic ideas

First some examples that will illustrate the basic ideas. When investigating the variable chain below, a natural idea is to replace the left-most VAR-cell with the contents of the right-most cell.

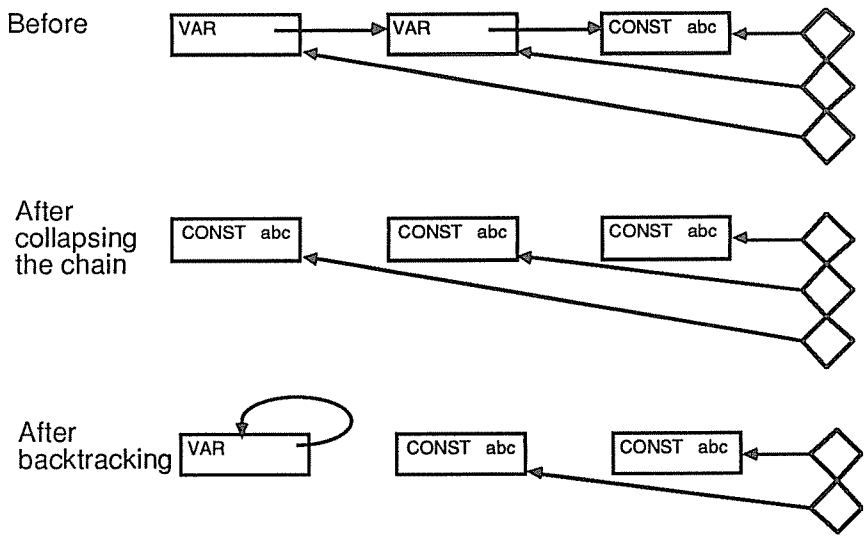


Unfortunately, this is not always a correct optimization, as some cells may be trailed, and thus reset on backtracking, as shown below. If the first cell had been changed to "CONST abc", as suggested above, an incorrect state would have been created.

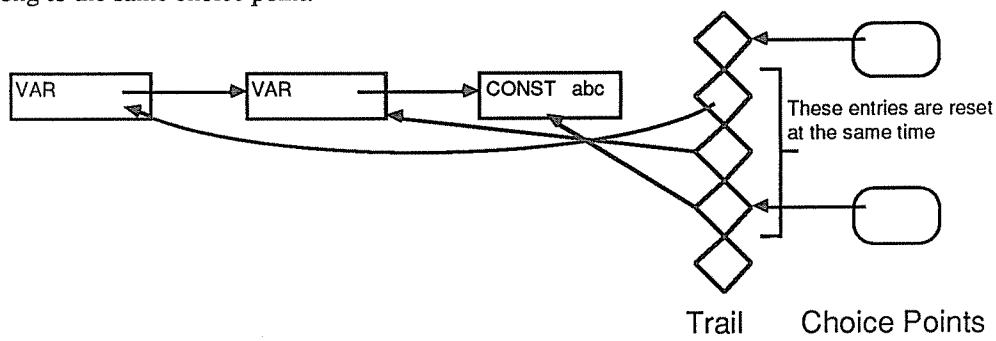


A natural solution is of course to stop following the chain as soon as a trailed cell is found. But we can do better than that!

In the example below, the left-most cell is trailed *later* than the other two cells. In this case, copying the value at the end of the chain is quite safe, as the first cell is reset *before* the other cells are reset. Thus there is no danger of the first cell containing a copy of a cell whose value is actually reset.



One more refinement of the rule above is possible. If two cells are trailed so they will be reset by the same backtracking point, then it is also safe to copy. The figure below shows a situation where the order of the trail entries is reversed compared to the example above, but where it is safe to shunt the chain, as all trail entries belong to the same choice point.

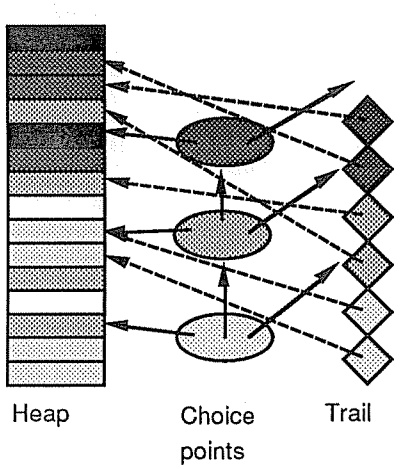


Not all value cells are trailed, so how are these cells to be treated? For a simple and uniform treatment of bound value cells the following definition is introduced:

DEFINITION:

The *binding age* of a bound value cell is

- for an untrailed cell: the age of the youngest choice point created before the cell.
- for a trailed cell: the age of the youngest choice point created before the trail entry.



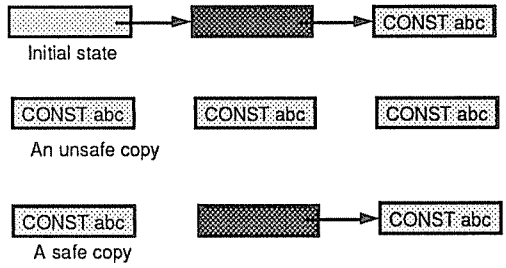
This figure illustrates the age of some value cells on the heap, where darker cells indicate older cells. For clarity, the choice points and the trail cells have also been given the corresponding shade. White cells indicate unbound value cells. These cells do not take part in the shunting.

With this definition of binding age we can state the following:

For a variable pointing to a value cell of the same binding age or older, it is always *safe* to copy the contents of the value cell to the variable. By *safe* we mean here that the value of the variable after dereferencing will during subsequent execution be the same as if no copying had been done.

As the variable cell is either reset (for trailed variables) or deallocated before or at the same time as the value cell it refers to, it is safe to copy the contents of the value cell into the variable cell.

If the value cell being copied is also a variable cell, the procedure above may be repeated until either a non-variable cell or a younger value cell is encountered. This principle is the basis for the algorithm.

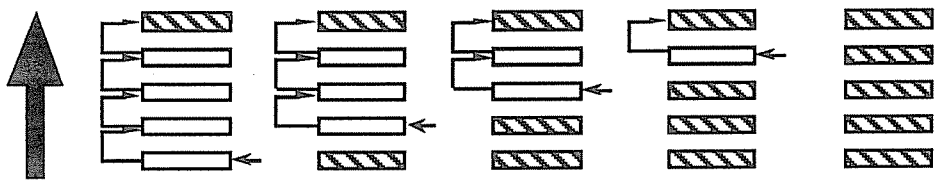


As illustrated by this figure, it may happen that it could be safe for the starting cell of a variable chain to be copied, whereas the intermediate cell may not be copied. Thus in general it is not possible to change all variable cells in a variable chain.

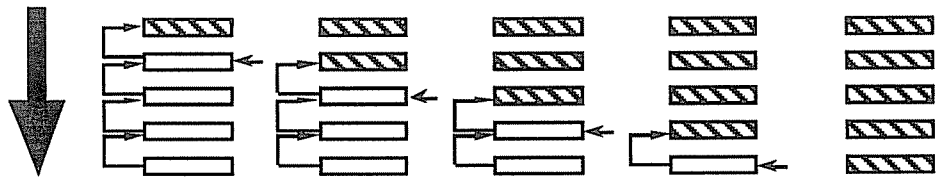
The algorithm

Let us call the parts of the stack and heap delimited by consecutive choice points a *segment*. (This is not to be confused with the two segments introduced by the segmented garbage collection.) Untrailed variables in the same segment have the same binding age, as defined above. The binding age of trailed variables is on the other hand determined by the choice point that caused them to be trailed. We do not need to know the exact binding age of variables, only their relative binding age.

In this algorithm each segment is shunted separately, but there is a need to have the value cells being trailed from younger segments (i.e. younger value cells) marked. This can either be done by starting with the youngest segment and ending with the oldest, or vice versa. The marking phase of the garbage collection algorithm [ACHS88] starts from the newest, but here we choose instead to start with the oldest. This seems slightly more advantageous as variable chains normally go in the opposite direction, i.e. from newer to older segments, as this is the preferred binding direction in WAM (to maximize the likelihood of not having to trail a value cell). Going in the opposite direction of a chain while shunting it, is computationally more efficient as shown by the example below.



Collapsing a variable chain by scanning it *along* its major direction



Collapsing a variable chain by scanning it *against* its major direction is more advantageous

The algorithm becomes slightly simpler to describe if we represent the principal WAM registers as a choice point `w`, which among other things contains pointers to the current environment (`E`), the continuation program counter (`CP`), the current choice point (`B`), the current top of trail (`TR`) and heap (`H`) and also all active argument registers (`A`).

Let `w0` be a pointer to a designated choice point, chosen so that `w0` and `w` delimit the area which is subject to variable shunting. Thus if the segmented garbage collection optimization has been used, `w0` points to the youngest choice point of the most recent GC, otherwise it points to a choice point containing the original settings of the relevant WAM registers.

Definitions of the relevant data types are shown in the Appendix.

The algorithm, called `shunt_vars` below, is comprised of the following steps:

- All trailed value cells are marked.
- To simplify traversing the choice points from the oldest to the newest, the chain of choice points starting with the control area `w` is temporarily reversed.
- Then the actual variable shunting is performed, starting from the oldest segment working towards the newest. During this step all marked value cells also become unmarked again.
- The chain of choice points is restored by another reversal.

```
shunt_vars()
{
    mark_all_trailed(w0→TR,w→TR);

    w = reverse_choicepoints(w);
    shunt_choicepoints(w);
    w = reverse_choicepoints(w);
}
```

The procedure `mark_all_trailed` traverses the trail between `from` and `to` marking all trailed cells.

```
mark_all_trailed(from,to)
struct valuecell **from, **to;
{
     while(from ≤ to)
     {
          (*from)→m = TRUE;
          from = from + 1;
     }
}
```

For completeness, the simple procedure `reverse_choicepoints` is also explicitly given :

```
struct choicepoint *reverse_choicepoints(B)
struct choicepoint *B;
{
     struct choicepoint *temp, *B2;

     B2 = w0;
     while(B≠w0)
          temp = B→B, B→B = B2, B2 = B, B = temp;
     return B2;
}
```

In `shunt_choicepoints`, which is shown below, the chain of all choice points is traversed by the variable `cp`. A trailing pointer `cp_last` is maintained which points to the previous (older) choice point. For each choicepoint, the respective stack, heap and trail segments are shunted separately.

First all value cells being trailed between `cp_last` and `cp` are unmarked. This denotes that the values in these cells are created in this segment and thus safe to shunt. Notice that these cells all reside in stack and heap segments that are older than the segments defined by `cp_last` and `cp`.

Then the same cells (referenced from the trail segment) are shunted. As mentioned above, the values of these variables are considered to be created in this segment and should therefore be shunted at the same time.

Then the heap segment defined by `cp_last` and `cp` is traversed, shunting the variable chain for any encountered unmarked variables. A marked variable must not be followed, as this denotes that it received its value at a later choice point.

Similarly the chain of environments is followed as long as they are more recent than `cp_last`. Unmarked variables in these environments are shunted.

Finally, the saved argument registers of the choicepoint are shunted. Notice that it is advantageous to consider the various segments in the exact order described above, due to the directionality of variable bindings.

```
shunt_choicepoints(cp)
struct choicepoint *cp;
{
     struct choicepoint *cp_last = w0;

     while(cp ≠ w0)
     {
          unmark_trailed(cp_last→TR,cp→TR);
          shunt_trail(cp_last→TR,cp→TR);
          shunt_heap(cp_last→H,cp→H);
          shunt_environments(cp→E,env_size(cp→CP),cp_last);
          for (i=0; i<size; i++)
               shunt_variable(&cp→A[i]);
          cp_last = cp;
          cp = cp→B;
     }
}


shunt_environments(env,size,cp_last)
struct environment *env;
int size;
struct choicepoint *cp_last;
{
     while(env>cp_last)
     {
          for (i=0; i<size; i++)
               if (env→Y[i].m ≡ FALSE)
                    shunt_variable(&env→Y[i]);
          size = env_size(env→CP);
          env = env→CE;
     }
}


shunt_heap(from,to)
struct valuecell *from, *to;
{
     while(from≠to)
     {
          if (from→m ≡ FALSE)
               shunt_variable(from);
          from = from+1;
     }
}
```

```
shunt_trail(from,to)
struct valuecell **from, **to;
{
     while(from≠to)
     {
          shunt_variable(*from);
          from = from+1;
     }
}


unmark_trailed(from,to)
struct valuecell **from, **to;
{
     while(from≠to)
     {
          (*from)→m = FALSE;
          from = from+1;
     }
}
```

Finally the workhorse of this whole algorithm: `shunt_variable` which does the actual pointer shunting. It is very simple; it follows an unmarked chain of bound variables, copying for each link of the chain its tag and value to the start of the chain.

An alternative implementation would be to postpone copying the tag and value until the end of the chain is reached, but since the length of a chain rarely exceeds 1, the current formulation is more efficient.

```
shunt_variable(start)
struct valuecell *start;
{
     while(start→tag ≡ VARIABLE AND
           start→value→m ≡ FALSE AND
           start→value ≠ start)   /* check for unbound variables */
          start→tag = start→value→tag,
          start→value = start→value→value;
}
```

Discussion

It would have been preferable to incorporate variable shunting into the marking phase of the garbage collection. Unfortunately, with the pointer reversal technique used during marking we found that this was not feasible.

Alternatively one would like to extend the ordinary execution of Prolog (i.e. the unification) to handle variable shunting. To some extent this is already done, but generally dereferencing during run-time is only done when necessary, following the principle of not doing anything unnecessary as the branch may fail anyway. The run-time system is however incapable of shunting variable chains originating in value cells created

before the latest choice point, as no run-time information is available in value cells regarding whether or when they have been trailed.

Finally, the most important question: Is this extra work trying to shunt variable chains worthwhile? Data from [Tick87] and [Touati87] suggest that most variable chains are very short. In the larger benchmarks (in [Touati87]) only 1.1%–36.9% of the references go from a variable cell to a constant cell. Longer reference chains account for less that 1% of all references. As these are dynamic and not static data, they are not directly related to how much memory could be saved by shunting. Also, the report does not mention how many of these cells were trailed in a way that makes shunting impossible. Nevertheless, those data suggest that variable shunting is of little importance in most applications.

Examples, like the one given in the first section, can be constructed where an unbounded amount of memory is consumed if no variable shunting is done. Thus, these programs would not be executable without shunting. However, this is not so interesting, as similar arguments can be made for almost any memory optimization.

In SICStus Prolog there are some indications that a particular application of variable shunting could be very important for some programs. This is due to the way *freeze* is implemented [Carlsson87]. A frozen unbound variable also refers to some goals that have to be executed when the variable becomes bound. Those goals are not removed when they have executed, as they are still potentially reachable if backtracking should occur and the variable should become rebound. But in precisely those circumstances when variable shunting is safe, the goals can safely be removed. This application seems to be the major advantage of using this algorithm, as garbage collection alone is incapable of removing those goals. Data from running some programs indicate that enormous amounts of memory can be consumed by frozen goals, causing the system to run out of space soon.

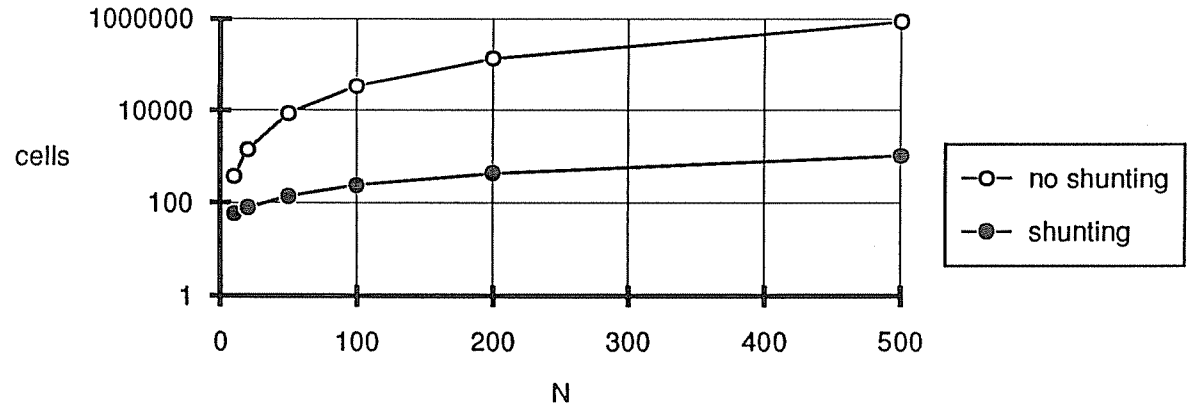
The same point is made in a paper by Serge Le Huitouze [Huitouze90], where variable shunting in the context of the MALI abstract machine is described (without algorithms). MALI uses essentially the same *freeze* mechanism as SICStus. The paper gives an example of a program generating prime numbers which allegedly consumes enormous amounts of memory unless variable shunting is used. To our surprise, the program ran in linear space in SICStus, *without* variable shunting, probably because the program was designed to simply print prime numbers instead of computing a list of them, and so the garbage emanating from the already computed prime numbers could be reclaimed by the usual mechanism. We did, however, reach results similar to [Huitouze90] with the program shown below, which computes a list of prime numbers:

```
nprimes(N, List) :- length(List, N), primesfrom(List, 2).

primesfrom([], _).
primesfrom([I|Ns], I) :- nomultall(Ns, I), J is I+1, !, primesfrom(Ns, J).
primesfrom(L, I) :- J is I+1, primesfrom(L, J).

nomultall([], _).
nomultall([N|Ns], I) :- freeze(N, N mod I > 0), nomultall(Ns, I).
```

We ran the above program with the query '`?- nprimes(N,L), garbage_collect.`' for different values of N and observed the amount of cells marked by the garbage collector, with and without variable shunting. The results are displayed in the table below.



Conclusions

We have designed and implemented algorithms for shunting chains of variable bindings in the WAM. We do not expect the technique to cause improved execution speed due to faster dereferencing except in exceptional cases. Nor do we expect the gain in execution speed for better memory reclamation to outweigh the overhead of variable shunting for most programs. However, we have demonstrated that unless this technique is used, some programs using the *freeze* mechanism will soon run out of memory.

Acknowledgements

We are grateful for the valuable comments from Karen Appleby, Gunnar Blomberg, and Per Brand.

Special thanks go to Irvin Shizgal who asked us what we meant by "collapsing variable chains" in [ACHS88], and this question prompted us to clarify our thoughts and write this paper. It still remains to be seen, however, what we meant by "folding identical structures".

Appendix: Data Types

This Appendix contains pseudo C declarations for the principal data types used.

An important data type is the *value cell*. A value cell stores a Prolog term and consists of a *tag*, a *value*, and two *bits* used for garbage collection:

```
struct valuecell {
      int tag;
      struct valuecell *value;
      bool m;
      bool f;
      }
```

A *choicepoint* consists of saved pointers into the various stacks, an alternative program pointer, saved environment and continuation pointers, and saved argument registers:

```
struct choicepoint {
      struct code *P;          /* alt. program pointer */
      struct code *CP;         /* continuation program pointer */
      struct environment *E;   /* environment pointer */
      struct choicepoint *B;   /* previous choicepoint */
      struct valuecell **TR;   /* trail pointer */
      struct valuecell *H;     /* heap pointer */
      struct valuecell A[m];   /* argument registers */
      }
```

An *environment* consists of a continuation program pointer, a continuation environment, and local variables:

```
struct environment {
      struct code *CP;         /* continuation program pointer */
      struct environment *CE;  /* continuation environment pointer */
      struct valuecell Y[m];   /* local variables */
      }
```

References

[ACHS88] ..... Appleby, Carlsson, Haridi and Sahlin, *Garbage Collection for Prolog Based on WAM*, Communications of the ACM, June 1988, Volume 31, Number 6.

[Carlsson87] ..... Mats Carlsson, *Freeze, Indexing, and Other Implementation Issues in the WAM*, In Logic Programming: Proceedings of the Fourth International Conference, MIT Press 1987.

[Huitouze90] ..... Serge Le Huitouze, *A new data structure for implementing extensions to Prolog*, Proc. International Workshop on Programming Language Implementation and Logic Programming (PLILP'90), Lecture Notes in Computer Science 456, pp. 136–150, 1990.

[Sahlin87] ..... Dan Sahlin, *Making the garbage collection independent of the amount of garbage*, Res. Rep. R87008, SICS, Kista, Sweden, ISSN 0283-3638.

[Sahlin89] ..... Dan Sahlin, *Collapsing Variable Chains in Prolog*, Res. Rep. R89003, SICS, Kista, Sweden, ISSN 0283-3638

[Tick87] ..... Evan Tick, *Studies in Prolog Architectures*, Ph.D. Thesis, T.R. CSL-TR-87-329 Stanford University, 1987.

[Touati87] ..... Hervé Touati and Alvin Despain, *An Empirical Study of the Warren Abstract Machine*, In Proceedings 1987 Symposium on Logic Programming, IEEE.