# A fresh look at garbage collection for Prolog

*B. Demoen*

*Report CW 330, January 2002*

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

# A fresh look at garbage collection for Prolog

*B. Demoen*

*Report CW 330, January 2002*

Department of Computer Science, K.U.Leuven

**Abstract**

Garbage collection in the context of the WAM is reconsidered independent of garbage collection algorithms. A set of garbage collections compatible with the WAM is specified in two steps: the first step make the useful data for each continuation private and ensures that no garbage terms survive any collection. The second step completes garbage collection by extending the notion of folding of identical structures. The role of the trail in the folding process is crucial and shown for the ordinary WAM trail as well as for a value-trail. Particular requirements on the folding step lead to particular garbage collection algorithms, but new and unexpected opportunities for recovering memory are discovered to be compatible with this view of garbage collection. Even though this does not lead directly to new practical algorithms, it is a start for a better understanding of the usefulness logic in the WAM and shows new potential for compile time analysis that can improve run time memory management. An improvement to the treatment of value-trailed locations during marking is also given.

# A fresh look at garbage collection for Prolog

Bart Demoen

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

January 31, 2002

**Abstract**

Garbage collection in the context of the WAM is reconsidered independent of garbage collection algorithms. A set of garbage collections compatible with the WAM is specified in two steps: the first step make the useful data for each continuation private and ensures that no garbage terms survive any collection. The second step completes garbage collection by extending the notion of folding of identical structures. The role of the trail in the folding process is crucial and shown for the ordinary WAM trail as well as for a value-trail. Particular requirements on the folding step lead to particular garbage collection algorithms, but new and unexpected opportunities for recovering memory are discovered to be compatible with this view of garbage collection. Even though this does not lead directly to new practical algorithms, it is a start for a better understanding of the usefulness logic in the WAM and shows new potential for compile time analysis that can improve run time memory management. An improvement to the treatment of value-trailed locations during marking is also given.

## 1 Introduction

We assume knowledge of Prolog and its implementation. For a good introduction to the WAM, see [1, 19].

It might seem that garbage collection for Prolog and the WAM is understood completely especially since [3, 4] give such a thorough description. However, during the past years, several alternative implementations for garbage collection have been described, in which surprisingly new insights or small variations on existing algorithms are presented: [5, 11, 13, 8, 9, 6]. Also, in [10] we attempted to specify the garbage collection process (for a WAM heap) in logic and used Prolog so that the specification was even executable. We expressed there the hope that the application of existing techniques - and of some still to develop techniques - would be able to transform the specification to an efficient implementation of the specified principle, resulting in algorithms like Cheney copying or Morris' algorithm for

garbage collection. For some time, we have tried to reach that goal, but without much success. On one hand, we were challenged by advanced techniques like early reset and variable chain shunting which defied a compact efficiently executable specification [1]. On the other hand, we became more and more confused by the dichotomy between pointer-reachability which results in locations being considered *needed* for the future execution, and usufulness logic which at least in principle should deal with the necessity of keeping *values* (or terms, data structures ...) alive. We were unable to resolve these issues cleanly in the context of the executable specification. And the fine print of [8, 9] and [6] for instance, even though perhaps not important in practice, was difficult to integrate in a general frame work. In short, it seems as if garbage collection is understood through algorithms rather than through first principles.

Still, in principle it is quite clear what any garbage collection process attempts to achieve: loosely speaking, it minimizes the total amount of space needed while respecting some constraints, so that the computation can proceed undisturbed - see Section 2 for more details. In practice, quite involved algorithms approximate this and often the constraints are not explicit. The abundance of algorithms hinders basic understanding. So we will step away from algorithms: in Section 3 we describe how every continuation can be given explicitly its useful data and no more. This process involves a potential duplication of data, it is non-deterministic and it covers three known but non-trivial actions garbage collectors usually take. Section 4 then describes a set of compactions that are consistent with the view that garbage collection must minimize the space and we make the restrictions explicit. These two steps work within the context of binary programs and the usual WAM trailing. The former restriction is lifted in Section 10 while the latter is extended to include value trailing in Section 7. We will indicate opportunities for compile time analysis which can help in making better memory management decisions and we show a new algorithm for marking more precisely in the presence of value trail in Section 8.

Our wish to specify garbage collection instead of writing algorithms ([10]) was the main motivation for the current work, but three more issues were important:

1. a small example of a WAM heap layout that WAM code compilers usually do not achieve, but is reasonable: see Section 6
2. a sentence in the acknowledgement of [17]: see Section 4
3. the attention and support of an anonymous referee for a small section in [12]: see Section 12

# 2 What is garbage collection for the WAM ?

For simplicity, we will make a series of assumptions: these will make it possible to focus on the essence, but do not restrict in any way the generality of the discussion. The first assumption is that we deal only with binary Prolog programs as in [18]. This has the advantage that when executed in the WAM, the local stack is empty and we can exclude it

[1] in spite of being easy to implement in a formalism like C !

from our discussion. In section 10 we will shortly describe how we propose to deal with the environment stack.

In most Prolog systems garbage collection can occur only at specific points in the execution, say during the execution of a *call* instruction, at the entry point of a predicate or at an equivalently safe point during the execution of a built-in predicate. We will assume here that garbage collection is triggered at the moment of calling a predicate, and since we are dealing with binary Prolog programs, this means during the *execute* instruction: at that point, the heap, trail stack, choice point stack and argument registers contain all the information about the future *continuations* of the machine [2]. We usually distinguish the forward continuation from the failure continuations each of which is a (potential) forward continuation after the execution has backtracked one or more times. This distinction is not needed here and we will talk about a sequence of continuations $C_1, C_2, ...$ which are ordered by age: $C_i$ is executed earlier than $C_{i+1}$ - barring the execution of the Prolog !/0 [3]. We actually do not really need the argument registers either: many implementations of garbage collection in the WAM start by pushing an extra choice point in which the current argument registers are saved, so that the treatment becomes more uniform. We will do the same here too. The stacks (heap, trail, choice points) now determine together with the program completely the future of the computation and we denote them by $WAM_{bef}$. Garbage collection clearly attempts to transform this machine state in a $WAM_{after}$ which has the *same future* as $WAM_{bef}$. Although we are vague about what *same* exactly means, it is clear that none of the code pointers can be adapted and that $WAM_{after}$ has the same number of choice points as $WAM_{bef}$ (or more). In fact, garbage collection should not inspect the program code, i.e. garbage collection must base its actions only on $WAM_{bef}$ [4]. Another way to express this is that any program that can produce $WAM_{bef}$, should be able to continue with the same garbage collection created $WAM_{after}$. We will not get closer to a specification of what an acceptable $WAM_{after}$ is: when a more detailed specification is attempted – for instance, the same abstract machine instructions must be executed afterwards – one gets into terribly distracting details.

The above is just a correctness condition on $WAM_{after}$, but we usually want at least one more property for a garbage collector:

- $WAM_{after}$ must be *smaller* than the $WAM_{bef}$

Here, we are less concerned with this issue than usual.
Depending on other requirements or choices, we could also impose

- the order (in memory) of variables remains the same

- the size of the heap on backtracking starting from $WAM_{after}$ does not become larger than when starting from $WAM_{bef}$

[2]in its simplest form, the game of memory management does not allow to peek into the program code at garbage collection time to make memory management decisions

[3]we cannot take future cuts into account because of the rules of the game

[4]this can be relaxed to allow inspection of compiler generated stack maps

The former one excludes a plain copying collector. The latter imposes some form of segment order preservation.

We could also go for more weird requirements:

- no reference chain becomes longer (see [8] for more about this)
- the trail must not become larger (see Example 3 why this is perhaps unreasonable)
- $WAM_{after}$ must be minimal immediatly after gc
- the representation must be minimal at the start of each continuation that existed at the moment of garbage collection ; it is not clear whether this requirement can always be fulfilled

Even though sounding weirdly detailed, the first two are actually enforced by most collectors. And many people might believe the last two: these people err.

Without a formal statement and proof, it seems clear that finding a minimal $WAM_{after}$ is NP-complete. Still, all garbage collectors run linearly in the size of at most the heap and the root set [5]. It means that all garbage collectors for WAM are *just* implementing a good heuristic. The point is perhaps academic, but it is instructive find out what current garbage collectors miss.

We have now reduced the garbage collection problem to a minimization problem, but we do not have a complete grasp of the constraints: we have been unable to state explicitly what it means for two WAM states to have the same future. Still, we will create a setting which allows to reason in a more detailed way about minimizing a state and discover in this setting interesting and new opportunities for minimization.

# 3 Privatizing usefull data

In Figure 1 we describe informally a set of transformations of one $WAM_{bef}$ to a set of $WAM_{int}$ machines, which also have the same future, but which have an **empty** trail. Let $N$ equal the number of continuations. There are now also $N$ choice points (including the one just pushed) and $N$ trail segments - the oldest trail segment is empty.

```
for i = 1 upto N do
    copy the terms referred to from choice point CPi, while relocating the
      entries in CPi
    untrail from trail segment TRi
```

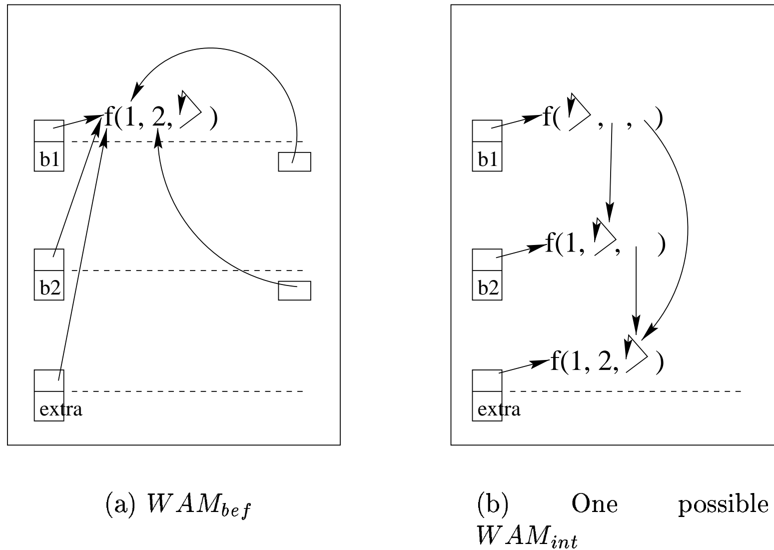
Figure 1: Informal algorithm for the construction of a $WAM_{int}$

We need to clarify what *copy* means and also what happens with the heap pointer in the choice points. We come back to these points later more formally and rely for now on some figures to make things clear with less effort. Consider the program

[5] or $NlogN$ in some cases - N is the size of the useful data

```
run :- A = f(X,Y,Z), b1(A).
b1(A) :- A = f(1,_,_), b2(A).
b1(A) :- use(A).
b2(A) :- A = f(_,2,_), ***(A).
b2(A) :- use(A).
```

and let the query be $? - run$. We have indicated the moment of doing the above transformation by the *** in the code: during the *execute* of in the first clause of b2/1. This notation will be used later on as well, possibly with a numeric index. The predicate *use/1* serves the purpose of keeping its argument alive.




(a) $WAM_{bef}$

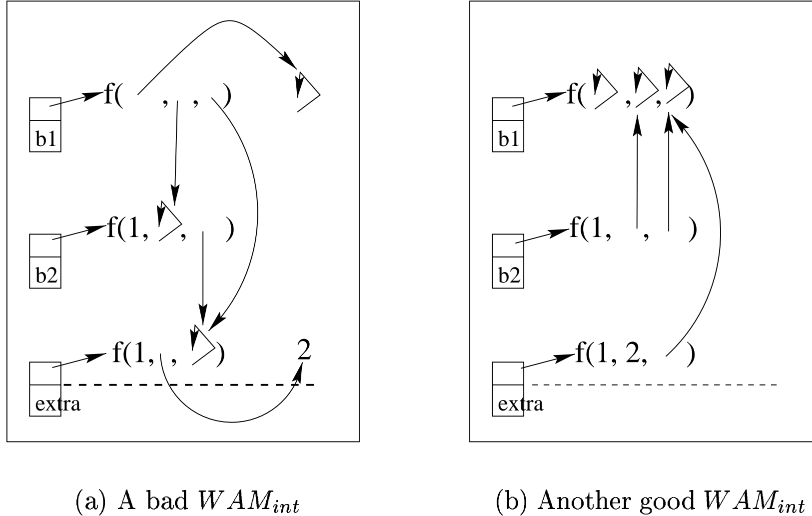
(b) One possible $WAM_{int}$

Figure 2: Illustration of the transformation

Figure 2(a) shows the situation just before the transformation: the dotted horizontal lines delimit segments; to the left are the choice points for b1 and b2 and the extra choice point we pushed for convenience. The trail entries are at the right of that subfigure. Figure 2(b) shows the situation just after the transformation: note that the variables shared by different continuations, remain shared after the transformation. But other terms - especially ground ones - are disconnected. Note that the trail in 2(b) is empty and each continuation could start executing by reinstalling the saved arguments from its choicepoint: execution of a continuation will of course need the trail again, but no part of the trail before the transformation. Still, the continuations cannot execute independently, because of the shared variables. It is certainly possible to present the issues also with completely disconnected continuations and in retrospect, this might have made the specification of the privatization step easier.

Figure 3(a) shows an alternative intermediate WAM: it show two *bad* cells. The first bad cell is an *undef* [6] which is referred to from inside a structured term, but which is not a cell

[6] a self-reference




(a) A bad $WAM_{int}$

(b) Another good $WAM_{int}$

Figure 3: More illustration of the transformation

of a structured term itself. The other bad cell contains the number 2, and it is also referred to from inside a structured term while not being a cell of a structured term. The copy we make with the procedure in Figure 1 cannot result in such cells. To be more precise: every cell on the heap containing a reference, must point directly to an undef cell; every undef cell must either belong directly to a structured term, or it must not be referred to from another cell on the heap.

This still leaves some freedom for the copy: indeed, Figure 3(b) is also a good alternative intermediate WAM. This means that there is more than one *correct* result form the transformation.

In the figures with an intermediate WAM, we have ommitted the segmentation of the (new) heap and we have indicated only its top: we still need to make a decision on where the heap pointers from the choice points need to point to. Since the WAM will reclaim heap on backtracking, the only requirement is that the heap pointer of a choice point $CP_i$ points at least as high in the heap as any of terms choice point $CP_i$ refers to: we will not pay any more attention to this issue.

Taking all this into account, the result of the transformation leaves room for the following variations in its outcome:

- the end of a variable chain can vary
- the heap pointers in choice points can vary
- the order of copied terms can vary
- cells containing atomic values or LIST or STRUCT tagged pointers are duplicated, but a block of cells referred to by the LIST or STRUCT tagged pointer can be eiterh duplicated or shared

The above informal description must be intuitively clear by now and it can be coded quite easily. But that would be besides the point because it is by no means the intention to produce a $WAM_{int}$ as an intermediate stage in a real garbage collection .

Lest the reader be misled: two different $WAM_{bef}$ can result in the same $WAM_{int}$ as the following example shows. Consider the programs

**Example 1** *Consider the following two programs:*

```
program 1                          program 2

run :- Y = f(_), b(Y).             run :- Y = f(_), b(Y).
b(Y) :- Y = f(a), ***(Y).          b(_) :- ***(f(a)).
b(Y) :- use(Y).                    b(Y) :- use(Y).
```

*Both result in a $WAM_{int}$ as in Figure 4*

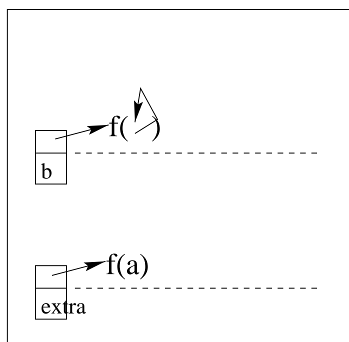



Figure 4: Illustration of the same privatization resulting from different $WAM_{bef}$

The total amount of memory needed after the transformation can larger than what was in use just before making it. In fact, in the example in Figure 2 it has doubled. So it is not immediatly clear what the use is of this step in garbage collection. Note that in [3] a similar forking of machines at choice points was considered in the conclusion, for the sake of making an (informal) argument in what sense the proposed garbage collection schema is optimal. The difference is mainly that we explicitly want different continuations to share variables.

Even though the transformation does not diminish the memory requirements yet, the situation after the transformation exhibits already a number of interesting properties and they are all related to the emptyness of the trail. In particular: variable shunting, early reset and rejuvenation of future garbage is dealt with already. And of course cleaning up the trail during garbage collection is achieved as well. We will discuss in a bit more detail the first three issues.

## 3.1 Variable shunting

Variable shunting consists in making reference chains shorter and was introduced by [17]. It is clear that after performing the privatizing transformation, no reference chain has length

more than one. So it covers completely the principle of variable shunting and it is in fact stronger than [17] and the variant of shunting described in [6]. Also the tiny extension to [17] introduced in [8] is covered. So, the known algorithms for shunting cannot be considered optimal, unless one takes into account the whole set of constraints one wants garbage collection to satisfy: shunting is only perfect, if one allows duplication of terms.

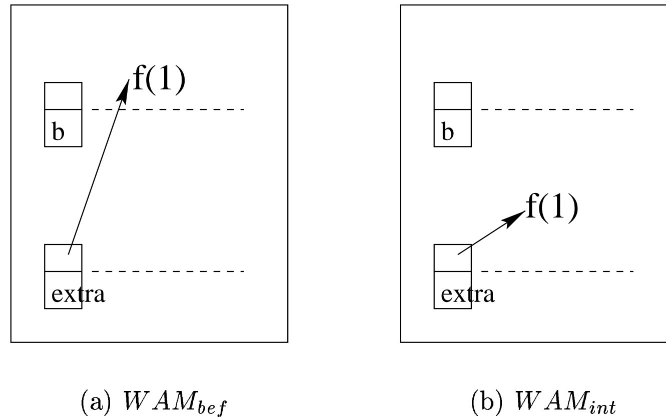
It is possible that in some continuations a chain that used to have length zero has length one after the transformation: some chains can be made shorter at the expense of others. This does however not violate the basic principle of varable shunting. Note that this effect was also described in [8].

## 3.2 Rejuvation of future garbage

In [11] we have shown how a copying phase from old to new root pointers moves terms to the oldest segment in which they are accessible without any extra cost, i.e. terms can move to a strictly younger segment. In $WAM_{int}$ each continutation has **only** access to its own data, and a natural division of this data into segments, achieves the same migration as in [11]. The following code and figure exemplifies rejuvation of future garbage:

```
run :- X = f(1), b(X).
b(X) :- ***(X).
b(_) :- ...
```

Note that rejuvation of future garbage relies on a compiler optimization named *choice point trimming* (see e.g. [15] and references therein) which is not implemented by many systems. For the example it means that the choice point for b/1 does not contain a reference to the term $f(1)$, as can be seen in figure 5(a).




(a) $WAM_{bef}$ (b) $WAM_{int}$

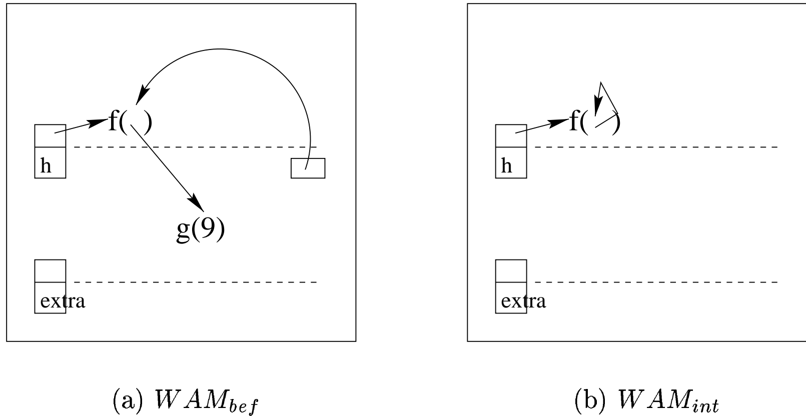
Figure 5: Illustration of rejuvenated future garbage

During the transformation, the term $f(1)$ moved from the oldest heap segment to the youngest one. This is potentially advantageous for the future computation because the term will be popped off the heap by backtracking earlier.

### 3.3 Early reset

Finally, also early reset is dealt with already: everything that could be reset, has been reset. This follows easily from the fact that the trail is empty, but a small example also shows this clearly; see figure 6.

```
run :- X = f(Y), h(X).
h(f(g(9)) :-  *** .  % none of the input is used here
h(X) :- use(X).
```




(a) $WAM_{bef}$ (b) $WAM_{int}$

Figure 6: Illustration of early reset

# 4 Sharing the private data or folding of identical structures

The total amount of memory needed after performing this first step can be larger than what was in use just before taking it. The reason is of course that sharing of terms between continuations (and possibly even within one continuation) is lost. In the next step we will try to (re-)introduce sharing among terms. This process seems quite obvious at first and it is expressed already in [3]: the idea seems to ressemble hash-consing as in [2], but is not further elaborated on in [3]. Later in the acknowledgement of [17], it is commented on by the sentence: *It remains to be seen, however, what we meant by "folding identical structures".*

Hash-consing can be performed at garbage collection time as described for instance in [2] for ML. In order to take full advantage of it, the abstract machine should also take it into account. The results in [2] are however not completely in favour of the technique. Moreover, in the context of a logic language with backtracking and logical variables, hash-consing takes on a different dimension: given the analogy between lazy datastructures and logical variables, it is no coincidence that hash-consing was studied in a strict functional language. Also, the phrase "identical structures" suggests an equivalence relation. We will show however that the right notion - until some point in the paper ! - is subsumption of terms.

We start demonstrating this by some examples: the Prolog code in the examples will contain manifest terms, but one should consider these as if undetectable by the compiler and constructed at run time.

**Example 2** *For the code*

```
run :- X = f(a), b(X).
b(X) :- Y = f(a), ***(X,Y).
b(X).
```

*the resulting* $WAM_{after}$ *is shown Figure 7.*

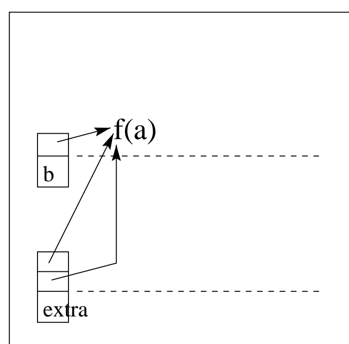



Figure 7: Illustration of folding of identical structures

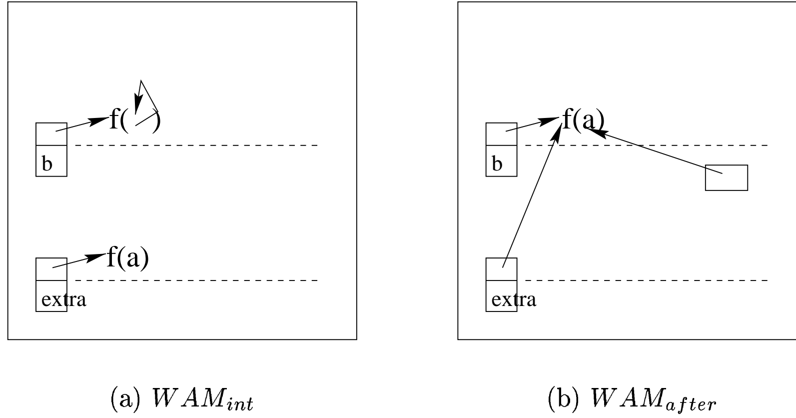
Example 2 shows the folding of identical structures that were accessible from different continuations, into one structure. Identical has the usual meaning of the Prolog built-in predicate == /2. After such folding, the term must belong to a segment at least as old as the oldest continuation involved in the folding: indeed, in the example only two continuations are involved, but in general, any number of continuations (even just one) could be involved. This kind of folding is similar to what hash-consing achieves. The example shows such folding for ground terms, but it is easily extended to identical (as for == /2) non-ground ==-indentical terms.

But we can also fold non-identical terms, as Example 3 shows.

**Example 3** *For the* $WAM_{int}$ *in Figure 8(a), a possible* $WAM_{after}$ *is shown in Figure 8(b).*

Example 3 shows the folding of non-identical structures. They can be folded even though they are not identical, because the two terms were accessible from different continuations and the older term subsumes the newer term: the latter is reflected by the fact that we have introduced a trail entry. Example 3 is very important, as it shows that

- the naive notion of *identical* structures is insufficient to allow useful kinds of folding
- the introduction of WAM trail entries gives a powerful means to extend the applicability of folding




(a) $WAM_{int}$ (b) $WAM_{after}$

Figure 8: Illustration of folding of non-identical structures

The naive notion of identical structures would entail that we are looking for an equivalence relation. However, consider the following situation: let $T_1 = f(a, b)$, $T_2 = f(a, _)$ and $T_3 = f(_, b)$, and let $T_1$ be accessible from continuation $C_1$ and the other two from $C_2$. Then $T_1$ and $T_2$ can be folded (by introducing a trail entry) or $T_1$ and $T_3$ can be folded (also introducing a trail entry), but $T_2$ and $T_3$ can not be folded: terms belonging to the same continuation, must be ==-identical for allowing folding. But when terms belong to different continuations, it is enough that one term subsumes the other.

We also note that in Figure 8(b) the space requirements of the machine are now minimal, in the sense that all heap segment lines are mininal.

Note that two terms that belong to the same continuation and that are not ==-identical can be folded when more is known on the future use of the terms. For instance for the program

```
run :- b***(f(a,b),f(c,d)).
b(f(X,_),f(_,Y)) :- use(X,Y).
```

it is clear that the terms in the WAM state at b*** could be folded as if the run/0 clause actually were

```
run :- Folded = f(a,d), b***(Folded,Folded).
```

but given the rules of the game, such folding is not allowed.

Note also that we can now fold a variable in one continuation with a different variable in another continuation, because clearly, one subsumes the other.

Up to now, there are two allowed actions:

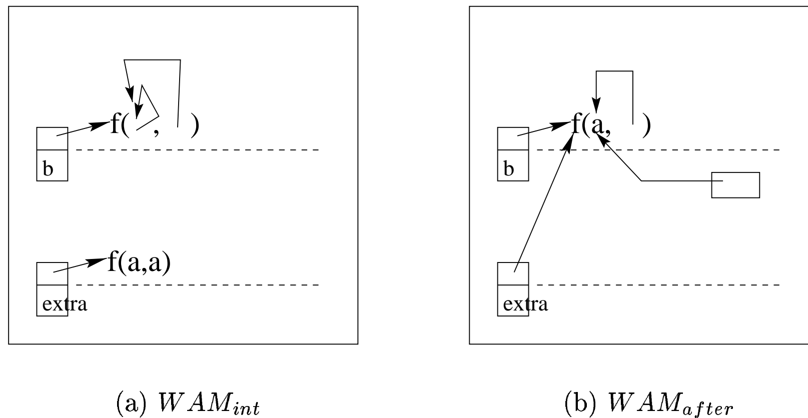
- fold identical structures
- fold structures that differ by one or more UNDEFs - where these UNDEFs must be in the strictly older structure - by introducing the appropriate trail entries

The next example show the need for one more rule.

**Example 4** *Consider the program:*

```
run :- X = f(A,A), a(X).

a(X) :- X = f(p,_), ***(X).
a(X) :- ...
```

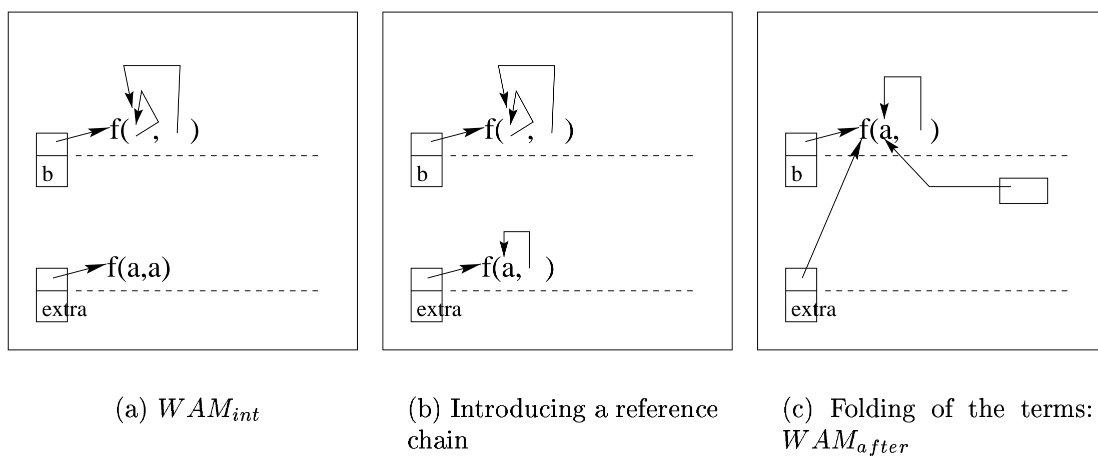



(a) $WAM_{int}$ (b) $WAM_{after}$

Figure 9: Illustration of the need for another rule

*The* $WAM_{int}$ *is shown in Figure 9(a). We clearly want to arrive at the* $WAM_{after}$ *in Figure 9(b), because it needs the least memory.*

The extra - and final - rule is that it is allowed to introduce reference chains. The succession of figures in Figure 10 shows how this is useful.




(a) $WAM_{int}$ (b) Introducing a reference chain (c) Folding of the terms: $WAM_{after}$

Figure 10: Introducing a chain and then folding

The example 4 shows that after the folding it is possible that $WAM_{bef} == WAM_{after}$, so the question remains: what did we gain in this process ? Apart from understanding [7] we note that the rule that allows to introduce trail entries allows for some freedom as to which segment the trail entry must be introduced in. This is the subject of the Section 5.

# 5 Rejuvenating trail entries

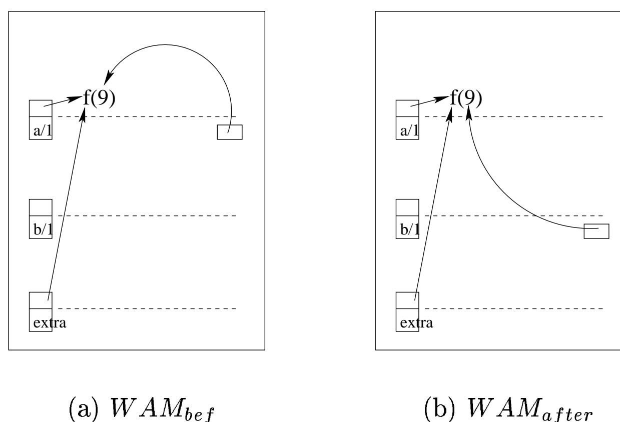
**Example 5** *Consider the program*

```
run :- a(f(X)).

a(Z) :- Z = f(9), b(Z).
a(Z) :- use(Z).

b(Z) :- ***(Z).
b(_) :- ...
```

*and the Figure 11.*




(a) $WAM_{bef}$ (b) $WAM_{after}$

Figure 11: A trail entry became younger

*Since the term $f(9)$ is not used in the in the second clause of b/1, the trail entry that trails the binding of X to the number 9, can be moved to the segment younger than the choice point for b/1.*

The example shows that during garbage collection *future* garbage can be made available for collection earlier by manipulating the trail. This would be useful if instead of the number 9, a large data structure were bound to X and if during the execution of the second clause of b/1, another garbage collection would occur.

[7] hopefully at least this !

One might think that the same effect can be obtained by moving (at the source level) the unification $Z = f(9)$ down to the body of b/1: however, in general this might lead to multiple executions of the unification and also, the above program is of course *too manifest.*

Note that X in Example 5 is not subject to early reset. The example shows that early reset is in fact *just* a special case of a trail entry being rejuvenated to the segment that is younger than the extra choice point, in which case it can be used for resetting immediately.

One challenge [8] is to generalize example 5 and characterize the situation in terms of a marking phase so that garbage collection can really use it. Static analysis might even derive enough information to help the garbage collector achieve such rejuvenation.

# 6 Beyond folding of identical structures

In 1988, we were challenged by Marleen D'Hondt [9] by the following statement: *any legal heap layout in WAM is the result of some compiler generated sequence of WAM instructions.* The following is clearly a counter example: the Prolog code is the query $?-X = [a \mid b], Y = f(a,b)$. and the layout as in figure 12 could not be the result of executing WAM instructions *as generated by the compiler.*

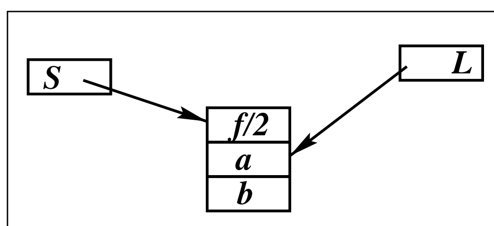



Figure 12: Absurd CSE ?

The particular heap layout can be seen as the result of a form of *common subexpression elimination* pushed to the almost absurd. The heap layout can be the result of a sequence of WAM instructions:

```
get_struct Y, f/2
get_list   X
unify_atom a
unify_atom b
```

Another example of such weird layout is provided by the Prolog code $X = f(a, b, [a \mid b])$ and $X = [a \mid b], Y = [b \mid c], Z = [c \mid d]$ with corresponding figure 13.

In the above examples, the (common subexpression elimination) optimization could be performed at compile time, because the common subexpressions were manifest. In general, such a common subexpression elimination can only be performed at run time and garbage collection time is a good opportunity because it has a global view on the data in the program.

[8] perhaps not leading to a practical result

[9] one of the ProLog-by-BIM team members

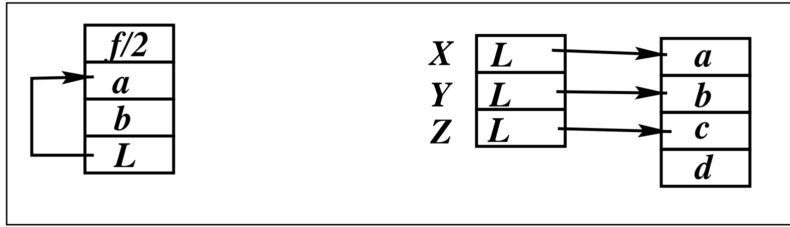



Figure 13: More absurd CSE ?

It therefore fits in well with the view of finding a minimal representation for the set of terms needed for all the continuations. It is however beyond the original intention of folding identical structures.

# 7 The value trail and folding of structures

Many Prolog implementations these days employ a value trail: it allows to destructively update a data structure - known to the user as an mutable variable - on the heap, so that on backtracking the old value can be re-installed. But a value trail is also used for re-installing forward bindings, as in the XSB execution model ([16]), and in parallel execution mechanisms as in [14]. In any case, a value trail is popular enough these days to warrant attention. Moreover, it gives a new dimension to the possibility that different processes can have a different view in the same location. Indeed, multiple values in one location occur in WAM for instance when a location - a heap cell - is bound to an integer say and it is trailed: in forward execution, the cell contains the integer. In the alternative forward execution - that is, after some backtracking - the cell contains again a free variable. But in plain WAM, there are at most two views on a cell: bound or undef. With a value trail, the number of views is bounded only by the number of continuations - note that for instance [15] describes a technique for avoiding multiple value trailing of the same location within the same segment. In section 8 we will describe an algorithm that deals more precisely with marking in the presence of value trail entries. We start by showing with a few examples how introducing value trail entries during garbage collection is useful in minimizing the heap consumption. Figures 14, 15 and 16 show this clearly. The minimality problem of garbage collection becomes even harder in the presence of the value trail.

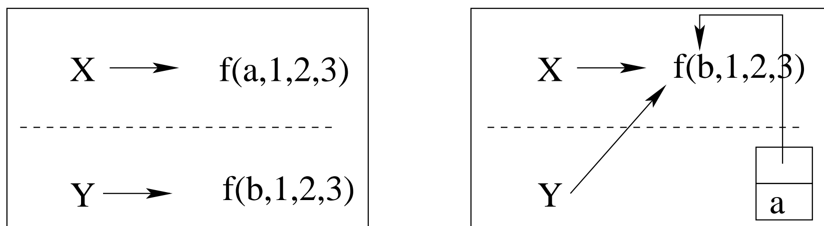



Figure 14: Folding while introducing an ordinary value trail entry

Figure 14 shows how an introduced value trail entry lowers the total memory consump-

tion: even though this might lead in practice to a longer execution later on, because future backtracking will need a non-void untrailing action.

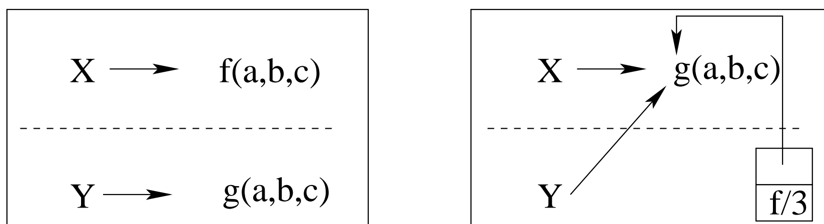



Figure 15: Introducing an unusual value trail entry

Figure 15 shows how value trailing can be done with an unusual value: normally, the values in the value trail are restricted to anything that can end a dereference action in the WAM – atomic values and a STRUCT or LIST tagged pointer. Here, we have put a FUNCT tagged value on the trail.

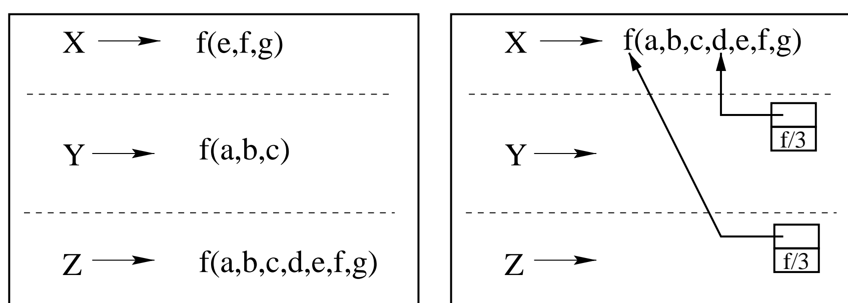



Figure 16: Introducing other unusual value trail entries

Figure 16 is another illustration of unusual value trail entries introduced for minimizing the WAM state.

# 8 Marking when locations have more than one value

In [12] we have pointed at a problem with marking in the presence of the value trail - or more generally in the presence of multiple views on the same cell. An anonymous (sic) referee pointed out that the *multiple view phenomenon* deserves more attention. We showed that that the usual marking process which associates one mark bit with each heap cell, is unable to treat precisely such a situation and concluded that each cell should have as many mark bits as there are continuations. The other conclusion was that with just one mark bit, the value from the value trail entry that points to a marked location, must be marked in order to be safe. This is indeed correct, but also conservative, i.e. too many values might be considered useful, as the following example shows:

```
a :- b(f([1])).
b(X) :- setarg(1,X,gee), gc, use(X).
b(_) :- ...
```

The question is whether a fixed and finite number of mark bits is enough to detect this in the marking phase. The following schema uses two bits for each heap cell. Our algorithm performs a (modified) marking phase for each continuation in the order from youngest to older continuation, so in the explanation of the meaning of the bits, we can refer to the current and a previous marking phase. Bits are associated to locations, i.e. given a location, we can find the value of its bits without having to inspect its contents.

- the **current value needed bit** indicates that the *value* in this location was in this location during the marking of the current or a previous continuation
- the **location needed bit** indicates that the cell at some point during the complete marking had its current value needed bit on

All bits are zero initially. The complete algorithm has two parts: a marking for all continuations followed by a treatment of the trail.

```
for i = 1 to numberofcontinuations do
      mark continuation i
      % treat trail segment i  -  for i == 1, this is a void operation
      for every trail entry in segment i do
         if the trail entry points to something with CVN off -> early reset
         else swap the value in the cell and the trail
              tag the heap pointer in the trail entry
              CVN set to off
      reset all CVN bits
```

Marking continuation i means that starting from the root set of continuation i, we traverse all data accessible from the root set. This marking could benefit from an extra bit - which is zero at the start of each continuation marking, so that the same data structure is not traversed repeatedly in case the terms are DAGs or to prevent looping in case the terms are cyclic, but this bit serves a completely different purpose than the other two and is therefore left out of the discussion.

Now comes the postpass treatment of the trail:

```
for each trail entry from old to young do
   if it is tagged
      swap the values
      untag it
      if it points to something unmarked
         mark the heap cell
         remove the trail entry
```

# 9 Dealing with generational garbage collection

Generational garbage collection basically considers the heap as divided in two parts: the part whose live data must be collected - the new generation - , and the part which is not inspected - the old generation. For this to work, all references from the old to the new generation must be known, because they keep data alive. In the WAM, the trail does so in a natural way. We can fit generational garbage collection to our schema, by only making the data private for the continuations that are in the new generation: this will not empty the trail, but after the privatization the trail contains only the pointers to the old generation. Folding of structures must now be restricted to terms in the new generation. One must take care to relocate pointers from old to new whenever necessary.

# 10 Dealing with the local stack

Up to now we restricted the discussion to binary programs for simplicity. Indeed, at first sight the environments complicate the picture a lot. However, using binarization of a non-binary program, applying the ideas above to the binary version and translating back to environments, will give insight. The crucial realisation is the relation between the term on the heap used for the forward continuation and an environment. The following example shows this.

```
original clause                    binary clause

a(X,Y) :- b(X,Z), c(Z,Y).          a(X,Y,C) :- b(X,Z,c(Z,Y,C)).
```

When executing a/2, its environment contains the variables Z and Y, because they have to survive the call to b/2. Analoguously, the variables Z and Y survive the execution of b/3, because they are in the continuation term c(Z,Y,C). One can easily see that the continuation term is never subject to choice point trimming: indeed, binarization makes sure that the incoming continuation is always used in the body. This means that up to the last clause for b/3 (included) the continuation term is live and this corresponds to the stack map that is associated to the call to c/2 in the body of a/2, which keeps exactly Z and Y alive – and the bare environment frame – as long as b/2 has not finished. The main difference between the continuation terms in the binary clauses and the environment for the orginal clause, is that continuation terms can never have a variable that is not initialized as this is not compatible with the WAM.

The following example shows that at the moment that b/2 is executing, only B and Z need be kept alive, while in the binary form, the corresponding continuation term c(B,C, d(C,Z, Cont)) also keeps C alive.

```
a(A,Z) :- b(A,B), c(B,C), d(C,Z).

a(A,Z,Cont) :- b(A,B, c(B,C, d(C,Z, Cont))).
```

Note that techniques like in [7] try to get rid of variables like C which in a binary clause only occurs in the body.

Still, the analogy is strong enough to make clear that environments act like heap terms, in that they can in the first step (see section 3) be replicated for different continuations and with a different set of active variables. In fact, one should imagine this first step taking into account any stack maps. In step two, environments can be collapsed, possibly with the introduction of trail entries. Collapsing was not always beneficial for heap terms, but in the case of stack frames it is: first note that even if there are no live variables in a stack frame, it has size 2 because the continuation pointer CP must be present and the previous environment pointer (Eprev). The next notable point is that variables cannot be moved in an environment (in an attempt to compact it), because WAM code was generated for fixed locations. It follows that two replicas of the same original environment will occupy always less space when collapsed because one needs to introduce trail cells for at most the variable slots which are common to both frames. Whether collapsing frames while introducing trail cells is also good for performance, is of course a different matter.

Finally, note that in the process of steps 1 and 2, one can trim frames: in the WAM this is named environment trimming and implemented as a mutator action on the basis of compile time information which is a crude approximation of stack maps. It is clear that environment trimming can be left to the collector as well.

# 11 Stack maps and choice point trimming

Reintroducing the local stack also offers the opportunity to point at the relation between choice point trimming and (precise) live variable maps (also named stack maps) when disjunction in the body is inline compiled. This relation is shown with an example; the clause for run1/0 has the same meaning as the clauses for run2/0 and new/2.

```
run1 :- a1(X), (b1(X), ***1 ; d1).

run2 :- a2(X), new(X).

new(X) :- b2(X), ***2.
new(_) :- d2.
```

At the moment of ***1 the stack map reflects the fact that X is dead. The corresponding point is ***2: without choice point trimming, X is saved in the choice point and considered live.

The relation is even deeper than the above example suggests. Consider the following example:

```
a :- f(X,Y), b(X), c(Y).

b(alfa) :- ...
b(beta) :- ...
```

Assume that the variable X is in environment slot 3 and Y in 4. With stack maps there would be associated to the program point $PP$ just after the call to b/1, the information that only 4 is alive. Now consider the transformed clauses

```
a(...) :- f(X,Y,3,4), b(X,3,4), c(Y,3,4).

b(alfa,_,_) :- ...
b(beta,_,_) :- ...
b(_,_,Z) :- use(Z), fail.
```

The program still has the same meaning. The extra clause for $b$ has only a named variable in the place corresponding to the variable(s) still needed after $PP$. With choice point trimming on the extra arguments, the choice point for b/3 will not contain the number 3. So from this choice point one can retrieve the live variable map during marking as follows: when a combination (E,CP) is considered for marking, find the choice point which has E and CP as saved register values. From this choice point retrieve the saved arguments that denote a live variable.

It is clear that this is not a practical method to have accurate information on live variables in environments: the program transformation is horrible and leads to inefficient code, it is not robust for cuts, the adaptation to the garbage collector is ugly ... but the main point is that live variable maps and choice point trimming are related.

Few systems implement choice point trimming, but we feel that our use of choice point trimming is justified in the previous sections, exactly because of the relatonship between choice point trimming and stack maps which are used by several systems.

# 12 Conclusion

In the past the understanding of usefulness logic was linked directly at describing algorithms for exploiting this understanding in a garbage colleciton context. Since the attention then shifts naturally to efficiency and practicality, it is no surprise that some consequences of the usefulness logic are never discovered. Another issue is that one is inclined to preserve WAM invariants and since these have not been studied fully, one tends to inprete these in a conservative way. For instance, efficiency considerations seem to prevent thinking about making changes to the trail, except for compacting it. Still, other changes to the trail are possible and compatible with the strict WAM invariants. And they can improve overall memory management. An example of a (false) WAM invariant that seems broken by rejuvanation of future garbage, is *terms can reside in a heap segment that is not younger than the segment they were created in.* The abundant absence of some compiler optimizations (like choice point trimming) also enforces this false intuition for WAM invariants.

By breaking away from these traditional views, we get both a better understanding of the usefulness logic of the WAM and of the chances for better memory management we have missed in the past. Practice might indicate that some of these are not worth considering. But that is perhaps the subject of another paper.

The value of this work lies in offering a clearer understanding of what usefulness logic for the WAM is and in providing a framework which seems to cover all existing techniques which up to now seemed ad hoc and unrelated, and even a few novel ones which are equally ad hoc and unrelated. We believe we are now in a better position to reason about the correctness of specific garbage collection processes by making them fit into the framework. Moreover, our interpretation of what *folding identical structures* could mean allows definitely to recover more garbage than the one in [3] and goes beyond plain hash-consing at garbage collection as proposed in [2]. Rejuvenation of trail entries was discovered while exploring the frame work, as it fits in neatly with the constraints.

We want to stress once more: none of the garbage collection opportunities described might be worth persuing in practice and this paper is not about the practically issue at all. We merely wanted to point out some consequences of a clearer definition of garbage collection in Prolog and the fact that they are unexpected: it means that the understanding of garbage collection lags behind actual implementation. Any formal specification of garbage collection for Prolog must take into account these issues at the risk of not covering the issue in full.

We now also realise that the way we tried to fulfill the original intention in [10] - finding executable specifications of garbage collection in the hope to find classical algorithms - leads away from the goal because the setting was too narrow.

Backtracking and the trail lead to memory cells whose contents depend on the continuation. Such multiple view issue seems also to exist in other circonstances. It remains to be seen whether any of the above is useful there too.

# Acknowledgements

We are grateful to Olivier Ridoux for stressing the importance of understanding multiple views on the same contents. Part of this work was conducted while the 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.

# 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] A. W. Appel and M. J. R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Feb. 1993.

[3] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.

[4] Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Y. Bekkers and J. Cohen, editors, *Proceedings of*

*IWMM'92: International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 82–102. Springer-Verlag, Sept. 1992.

[5] J. Bevemyr 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.

[6] L. F. Castro and V. S. Costa. Understanding memory management in prolog systems. In P. Codognet, editor, *Proceedings of the 17th International Conference on Logic Programming, ICLP'2001*, number 2237 in Lecture Notes in Computer Science, pages 11–26. Springer-Verlag, jul 2001.

[7] B. Demoen. On the transformation of a Prolog program to a more efficient binary program. In *Proceedings of the LOPSTR'92 Workshop, Manchester, July 1992*, 1992.

[8] B. Demoen. Early reset and reference counting improve variable shunting in the WAM. Report CW 298, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Aug. 2000.

[9] B. Demoen. Marking in the presence of destructive assignment is suboptimal. Report CW 302, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Oct. 2000.

[10] B. Demoen. Prolog and abduction 4 writing garbage collectors. In K.-K. Lau, editor, *Pre-Proceedings of Tenth International Workshop on Logic-based Program Syntthesis and Transformation, 2000*, pages 128–135. University of Manchester, 2000. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1, URL : http://www.cs.man.ac.uk/cstechrep/titles00.html.

[11] B. Demoen, G. Engels, and P. Tarau. Segment order preserving copying garbage collection for WAM based Prolog. In *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 380–386. ACM Press, Feb. 1996.

[12] B. Demoen and K. Sagonas. Heap memory management in Prolog with tabling: Principles and practice. *Journal of Functional and Logic Programming*, 2001(9):1–56, Oct. 2001.

[13] 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.

[14] E. Lusk, R. Butler, R. O. T. Disz, R. Overbeek, R. Stevens, D. H. D. Warren, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The aurora or-parallel prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[15] J. Noye. *Elagage de contexte, retour arriere superficiel, modifications reversibles et autres: une etude approfondie de la WAM.* PhD thesis, Universite de Rennes I, Nov. 1994.

[16] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. of SIGMOD 1994 Conference.* ACM, 1994.

[17] D. Sahlin and M. Carlsson. Variable shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.

[18] 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.

[19] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

# Appendix

The next two examples show that a term can be accessible from continuation $C_i$ and non-accessible from continuation $C_j$ for any $i \neq j$.

**Example 6** *Take the code*

```
run :- X = f(9), a(X).
a(X) :- ***.
a(X) :- c(X).
```

*Clearly, the term $f(9)$ is accessible from $C_2$ (that is the first failure continuation) and not from $C_1$ (which is the forward continuation).*

**Example 7** *Take the code*

```
run :- a(f(9)).
a(X) :- gc, b(X).
a(_) :- ...
```

*Clearly, the term $f(9)$ is accessible from $C_1$ and not from $C_2$.*

The reason for these explicit examples is to take away any doubt about the possibility that some examples in the main text can really occur.