Copying overlapping terms

Ulrich Neumerkel[1]

Technische Universität Wien
Institut für Computersprachen
A-1040 Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract. In this paper we discuss techniques to copy overlapping terms. Starting from the classical Cheney-style copying algorithm, we present several refinements that exploit compact terms where a structure's last argument overlaps with the following structure. Finally, a two pass algorithm is presented that uses less auxiliary space and that exploits all possible overlaps for finite terms. In spite of its using two passes, this algorithm is comparable in speed to single pass algorithms. The presented algorithms have been used and compared within BinProlog.[1]

Keywords: *implementation of Prolog, WAM, term representation, last argument overlapping*

1 Introduction

Usage of copying algorithms. Algorithms for copying terms are used for many purposes. In the case of a Prolog system copying is used for implementing various predicates like setof/3, copy_term/2 and lemmaing predicates as well as assert/1. These predicates require so called non-destructive copying algorithms which preserve the original term. Further, copying algorithms are often used for garbage collection. In this case the original term can be discarded and so called destructive copying can be used.

BinProlog. In this paper we consider adaptations of copying algorithms for the term representation used in the BinProlog system. BinProlog is a C-emulated Prolog system that uses a simplified WAM [7] called BinWAM [5] which executes Prolog in a continuation passing style. The BinWAM uses also a simplified term representation that has been adopted to support last argument overlapping [6].

Contents. In Section 2 presents the simple term representation used in the BinWAM. Section 3 discusses algorithms for copying overlapping terms. Section 4 presents performance results in BinProlog.

[1] URL: `ftp://clement.info.umoncton.ca/pub/BinProlog`

2 Term representation

We compare the simplified term representation of the BinWAM with the traditional structure copying representation used in the WAM. As the BinWAM uses a continuation passing style to implement goals, it allocates more terms than the traditional WAM. A compact representation of heap terms is therefore of particular interest for the BinWAM. This representation is also of interest for other structure copying Prolog systems.

Tag-on-pointer representation. Current structure copying Prolog machines like the WAM use several pointer types to represent terms. Usually, at least three pointer types are used. We call this representation *tag-on-pointer* representation.

1. reference or variable
2. pointer to a structure
3. pointer to a list, as an optimization for structure ./2

The term [f(X,a)] is represented as in Fig. 1.

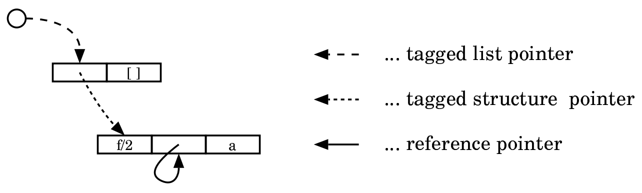

Fig. 1. Representation of the term [f(X,a)] in the WAM.

The specialized pointer type for lists is not strictly needed. Yet, most machines implement this optimization. Usually, references are tagged by word alignment. The other pointer tags are encoded in the lower bits. When creating a pointer, a dedicated tag is added to the address.

Tag-on-data representation. The BinWAM uses a *single* pointer type. Pointers are word aligned so no extra tagging is needed. Only data that does not contain pointers is tagged: functor blocks, atoms and integers.

1. reference, variable, or pointer to structure

In the case of the term [f(X,a)] the BinWAM requires 6 cells (Fig. 2), while the WAM requires only 5.

Term manipulation for this simplified representation is quite similar to the WAM, some differences must be observed for binding variables. By and large the efforts are comparable to the WAM.

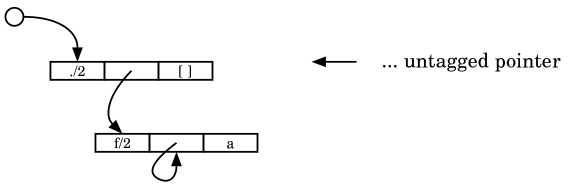

Fig. 2. Representation of the term [f(X,a)] in the BinWAM.

Last argument overlapping. While the BinWAM does not provide a specialized representation for lists, it allows a more general optimized representation useful for any structure. References to structures in the last argument of another structure can be replaced by the structure itself. The WAM represents a list of n elements by $2n$ memory cells. Fig. 3 shows the representation of the list [1,2,3] in the classical WAM.

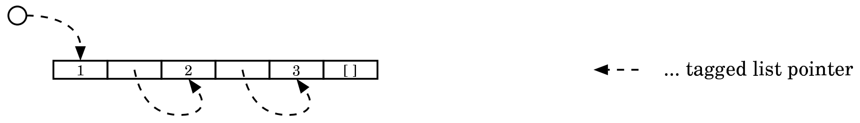

Fig. 3. Representation of the list [1,2,3] in the WAM.

Other structures of arity 2 are encoded with a separate functor block indicating the structures name and arity. To represent a list of n elements with an other functor $3n$ memory cells are required (Fig. 4).

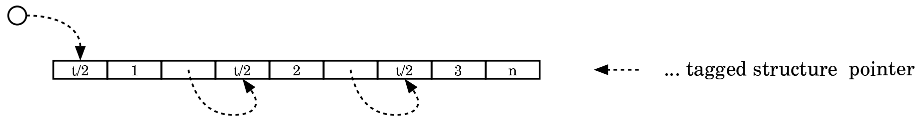

Fig. 4. Representation of the term t(1,t(2,t(3,n))) in the WAM.

Since pointers in the BinWAM are untagged the last argument of a structure can contain directly the next structure's functor block (Fig. 5).

The BinWAM is able to represent a list of n elements with $2n + 1$ cells. However, to exploit this representation cells must be allocated in appropriate order. Already allocated structures cannot exploit this optimization as show in Fig. 6. In the best case, last argument overlapping can half a term's size. For example the term s^n can be represented with only $n+1$ cells, whereas the WAM requires always $2n$ cells. Fig. 7 shows the case $n = 5$.

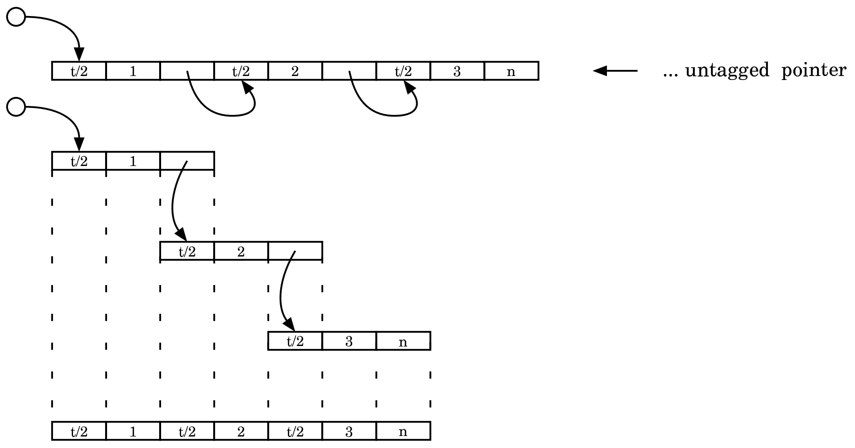

Fig. 5. Representations of the term t(1,t(2,t(3,n))) in the BinWAM.

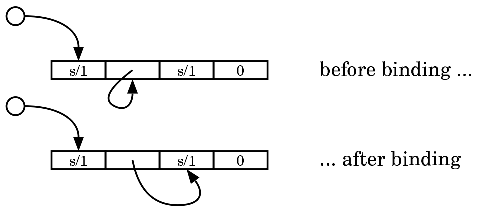

Fig. 6. Unexploited overlapping.

Adaptations for last argument overlapping. A few built-ins had to be modified in order to read the new compact representation. Instructions creating structures were modified to create overlapping structures if possible. Built-ins that copy terms have been adapted as will be shown in the next section.

Last argument overlapping versus CDR-coding. Techniques for eliminating pointers in data structures of symbolic programming languages are well known. In particular, CDR-coding has been used in LISP-systems. In such systems, a CONS-cell does not contain the CDR-pointer to the next CONS-cell.

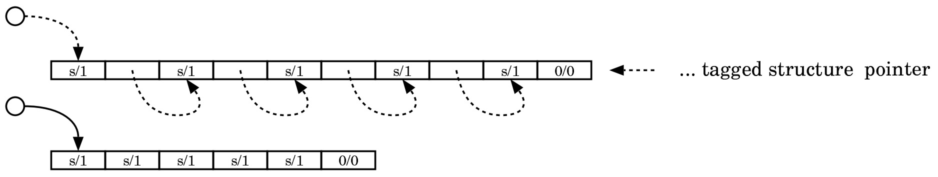

Fig. 7. Representations of the term s(s(s(s(s(0))))) in WAM and BinWAM.

Instead, a special tag in the CAR indicates that the next CONS-cell is located at the place of the CDR. CDR-coding has been investigated for Prolog in [3]. Since this technique requires additional effort for decoding pointers it is a good candidate for hardware implementations.

In comparison to CDR-coding last argument overlapping is much simpler. On the other hand the typical reduction of terms is also lower: CDR-coding reduces a list of $2n$ cells to approximately n cells. The typical storage reductions of last argument overlapping are smaller, typically less than $3n$ to $2n$. For Prolog systems, last argument overlapping is a more general optimization that applies to any structure and requires no special tags. In particular, continuations profit from our optimization. Optimizations of CDR-coding, like the preallocation of storage for the simultaneous creation of several CDR-coded lists [4] does not seem to be feasible for our technique. Modifications to copy-collection for linearizing lists were already discussed by Baker [1] in the context of CDR-coding in LISP-systems.

3 Copying algorithms

3.1 Breadth first copying

The classical algorithm of Cheney [2] has been adapted for the BinWAM's terms (Fig. 15). During copying, structures and variables are marked with a forward references. Since all pointers in the BinWAM are untagged, following a reference chain is performed implicitly with the dereferencing operation DEREF2. When the original term is no more required as in the case of garbage collection, Cheney's algorithm uses no auxiliary space. For non-destructive copying the forwarding pointers in the original term must be trailed (CT_TRAIL_IT) such that they can be removed after copying (UNWIND_TRAIL). Non-destructive copying requires thus auxiliary space equal to the number of forwarding pointers.

Worst case of breadth first copying. The term t(s^n,s^n) illustrates the deficiencies of breadth first copying. This term requires in the best case $2+2(n+1) = 2n+4$ memory cells: $n+1$ for each term s^n and 2 cells for the structure t/2 which overlaps with its second subterm. Copying this term breadth first expands the term to $3+2(2n) = 4n+3$ memory cells yielding an expansion ratio of $1:2$. Thus, in the worst case, a simple Cheney copying algorithm doubles the size of the copied term.

3.2 Last argument first copying

The adaptations of the simple Cheney style copying algorithm are very few (cf. the bold parts in Fig. 15). When a structure is copied, the last argument is checked for another uncopied structure. Only if there is none, the algorithm resumes to the main loop, otherwise the algorithm continues copying with the

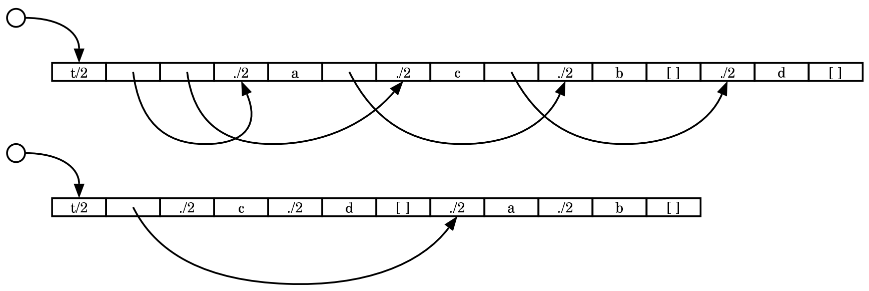

Fig. 8. Breadth first copying vs. Last argument first and breadth first for t([a,b],[c,d]).

last argument. Many structures are now copied in an optimal manner, as e.g. the term t([a,b],[c,d]) (Fig. 8).

This algorithm exploits all possibilities for last argument overlapping as long as there are no shared subterms, that are referred to from both a last argument position and another position. Overlaps of structures that are referred to from both a last argument position and another position are only copied optimally, if the reference out of a last argument position happens to be copied first. Otherwise, the overlap is not detected.

Worst case of last argument first copying. To construct the worst case we take a term with the best compression rate (s^n) and refer to each subterm from a different position. Consider e.g. the term f(s^1, s^2, ..., s^{n-1}, s^n, 0) where all common subterms are shared. The optimal representation requires $2n + 3$ memory cells for all $n > 0 : n + 2$ for the functor $\mathrm{f}/n + 1$ and $n + 1$ for the term s^n. In this case, all terms s^i $0 < i < n$ are represented as pointers to subterms of s^n (Fig. 9).

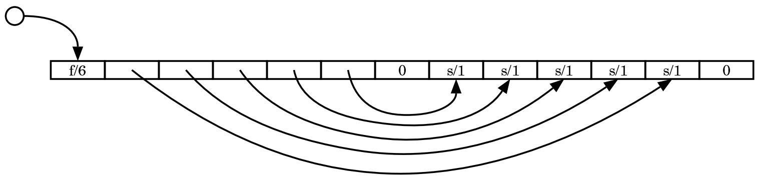

Fig. 9. Optimal representation of f(s^1, ..., s^n, 0) for $n = 6$.

Every structure s/1 up to one can be accessed twice: once from the structure $\mathrm{f}/n{+}1$ and once from another structure s/1. In order to perform optimal copying, the topmost structure representing s^n should be copied first. However, a left-to-right scan first copies s^1, then s^2 etc. All last argument overlaps are hence destroyed. The resulting copy (Fig. 10) requires $(n{+}2) + 2n = 3n{+}2$ cells. Thus yielding an expansion ration of 1 : 2. Fig. 10 illustrates the case $n = 5$. Not that changing the direction of the scan only causes a different term to be the worst case.

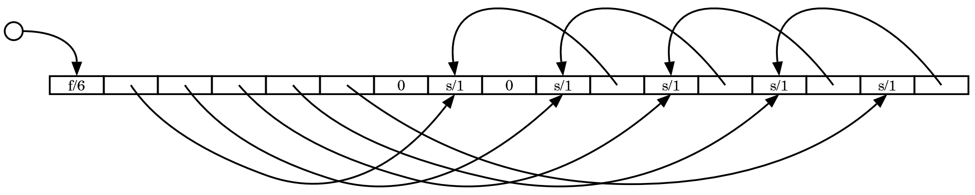

Fig. 10. Worst case after copying of f(s^1, ..., s^n, 0) for $n = 6$.

To avoid the premature copying of overlapping structures it would be best to start copying non overlapping structures. Unfortunately such an approach seems to be very costly since it seems to require considerable auxiliary space. Instead, the algorithm in the next part simply avoids copying overlapping structures at inappropriate places.

3.3 A mark & copy algorithm

Our algorithm is outlined in Fig. 11. The major part of our algorithm consists of a marking and a copying phase. These phases detect overlapping structures and delay their being copied by storing a reference to them. After copying the delayed terms are reconsidered. In case of finite terms, all of the delayed terms will be copied. In case of infinite terms a delayed structure is copied and the marking and copying phase resumes.

```
term mc_copy_term(h,t,from,to,wam)
  ...
{ do
    { Phase 1: mark term
      Phase 2: copy term
      update delayed references
      find uncopied references
    }
  while (there is an uncopied reference, choose an uncopied reference);
  untrail forward references
}
```

Fig. 11. Outline of mark&copy

Phase 1: Marking. The primary purpose of the marking phase is to identify candidates for overlaps. Note that not all structures that are referenced from a last argument are overlaps. E.g. the term X = s(X) will be marked as a candidate, yet an overlap cannot be realized.

Since a separate marking pass approximately doubles the execution time of copying, we extended the simple marking algorithm to detect more information about structures. In addition to detecting possible candidates for last argument overlapping the number of references to a cell are counted as well. A two bit reference count is used to distinguish between the states "unmarked" "marked once" and "marked more than once".

Functor blocks are marked using the following bits:

marked: true, if structure is referred to by at least one reference
marked multiply: true, if structure is referred to by more than one reference
lastarg referenced: true, if structure is referred to by a last argument

Phase 2: Copying. Copying uses the information collected in the marking phase as follows:

1. For structures that are referred to only once ("marked multiply" is false) no forward reference is created. Therefore there is also no need to trail this forward reference, and untrail it after copying.
2. Structures are no more tested to be in old- or newspace. A structure in newspace is not marked at all, whereas a structure in oldspace is always marked.
3. Structures marked as "lastarg referenced" are not copied if copying would not exploit overlapping. Instead the reference to this structure is stored on the trail for later copying or updating.

Auxiliary space. The stack required in the marking phase is allocated in the newspace, so auxiliary space is only created for forward references and delayed references. The algorithm creates forward references for structures only if they are referenced at least twice. For many small terms (as e.g. when using setof/3) no auxiliary space is needed.

While mark & copy is optimal for finite structures, infinite structures are still handled suboptimally. Argument overlaps remain unexploited if the only access to the subterm is via an overlapping structure as in Fig. 12. In this case the algorithm copies the structure potentially losing the possibility for overlaps.

Worst case for mark & copy. The worst case ratio of the previous algorithm $(2:3)$ is not reduced. Fig. 13 shows such a structure for $n = 4$. There are n cyclic terms T_i of the form $\mathrm{T}_{i+1} = \mathrm{s}(\mathrm{f}(\mathrm{T}_{i+1},\mathrm{T}_i))$. The innermost term T_0=a. Therefore, except of T_n all terms are subterms of other terms. All functors f/2 and all but one functor s/1 are marked as overlaps. All n structures f/2 are referred to from n external places. The term consists in the best case of $(1+n)+(3n+1) = 4n+2$

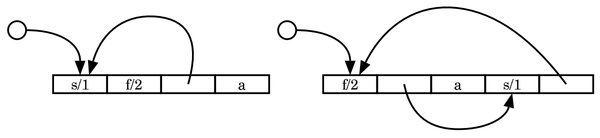

Fig. 12. The simplest case of unexploited overlapping.

cells. Since all terms f/2 seen from outside are marked as overlaps, the first round of the copying algorithm delays all copies. In the next round a term is chosen to be copied in spite its being marked as an overlap. In case of an unfortunate choice (all terms appear equal from outside) all overlaps are destroyed (Fig. 13). The term now occupies $(1 + n) + 5n = 6n + 1$ cells.

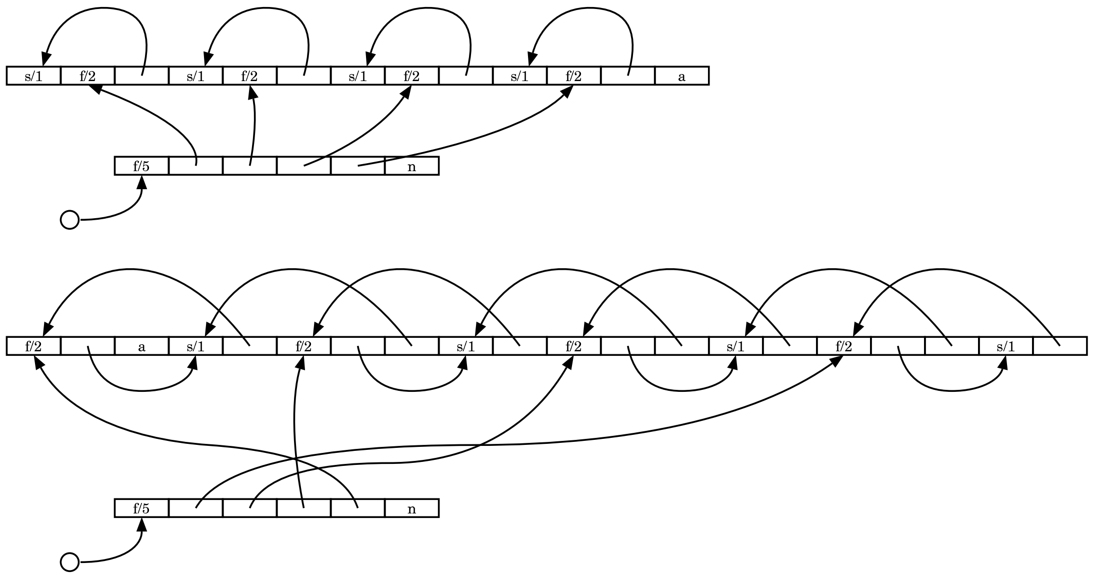

Fig. 13. Worst case for mark & copy for $n = 4$

4 Performance evaluation

The table Fig. 14 shows the times for copying 100 000 times the indicated term on a on Sparc 10/20. Benchmark "nil" copies just the atom [] and shows the calling overheads incurred. "list_a" measures a list of atoms of given length. "list_a 16" corresponds to the average size of a term to be copied. "s1" is an s(X)-term of indicated length. "f3" is a linear term with functor term3/3. When the term's arity increases our new algorithm deteriorates, since marking visits all cells.

data	*n*	*Cheney orig.*	*Cheney overl.*	*Mark & Copy*
nil	0	520 1.00	520 1.00	520 1.00
list_a	8	1440 1.00	1210 0.84	1420 0.99
list_a	16	2270 1.00	1770 0.78	2240 0.99
list_a	1024	119700 1.00	80170 0.67	102400 0.86
s1	8	1320 1.00	1070 0.81	1190 0.90
s1	16	2030 1.00	1490 0.73	1700 0.84
s1	1024	98640 1.00	56510 0.57	65930 0.67
term3	8	1560 1.00	1380 0.88	1720 1.10
term3	16	2480 1.00	2020 0.81	2810 1.13
term3	1024	134560 1.00	100390 0.75	149730 1.11

Fig. 14. Execution times (ms) for copying 100 000 times on Sparc 10/20

Acknowledgments. We thank Paul Tarau for providing BinProlog and many discussions. This work was done within project INTAS-93-1702 Efficient symbolic computing. The Jubiläumsfond der Stadt Wien supported this work.

References

1. H. G. J. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, Apr. 1978.
2. C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of ACM*, 11(13):677–678, Nov. 1970.
3. T. Dobry. *A High Performance Architecture for Prolog.* Phd thesis, University of California at Berkley, 1987.
4. K. Li and P. Hudak. A new list compaction method. *Software—Practice and Experience*, 16(2):145–163, Feb. 1986.
5. P. Tarau. A simplified abstract machine for the execution of binary metaprograms. In *Proceedings of the Logic Programming Conference'91*, pages 119–128. ICOT, Tokyo, Sept. 1991.
6. P. Tarau and U. Neumerkel. A novel term compression scheme and data representation in the BinWAM. In *Programming Language Implementation and Logic Programming, International Symposium, PLILP'94*, Lecture Notes in Computer Science. Springer-Verlag, 1994.
7. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Oct. 1983.

This article was processed using the LaTeX macro package with LLNCS style

```
term copy_term(h,t,from,to,wam)                                          a
   term h,t,from,to; stack wam;
{  term ct = h; term *TR=TR_TOP;
   SETREF(h++,t);
   do
      {  term t; cell val_t;
         t = ct; DEREF2(t,val_t);
         if (t == ct)
            {}
         else if (ATOMIC(val_t)) SETCELL(ct,val_t);
         else if (INSPACE(t,from,h)) SETREF(ct,t);
         else if (VAR(val_t))
            {  SETREF(ct,ct);
               CT_TRAIL_IT(t); SETREF(t,ct); /* Forward reference */
            }
         else
            {  SETREF(ct,h);
               do
                  {  cell arity = GETARITY(val_t);
                     SETCELL(h,val_t);
                     CT_TRAIL_IT(t); SETREF(t,h); /* Forward reference */
                     if(h>to) OVERFLOW;
                     COPY_CELLS(h,t,arity-1);
                     h += arity; t += arity;
                     DEREF2(t,val_t);
                  }
               while (COMPOUND(val_t) && !INSPACE(t,from,h));
               if (COMPOUND(val_t)) SETREF(h++,t);
               else SETCELL(h++,val_t);
            }
         ct++;
      }
   while (ct < h);
   UNWIND_TRAIL(TR,TR_TOP);
   return h;
}
```

Fig. 15. Cheney style copying. **Bold** parts are adaptations for last argument overlapping