

Representation Sharing for Prolog

Bart Demoen
Phuong-Lan Nguyen

Report CW 571, November 2009



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

Representation Sharing for Prolog

Bart Demoen
Phuong-Lan Nguyen

Report CW571, November 2009

Department of Computer Science, K.U.Leuven

Abstract

Techniques to reduce the memory footprint of an application include representation sharing between multiple copies of some data. Representation sharing has been implemented through hash-consing in functional languages. Representation sharing for Prolog was not given that much attention in the past. Here, representation sharing is defined for Prolog implementations in a structure copying environment (basically the WAM), and the specific issues arising from the logical variable and backtracking are analyzed. A high-level algorithm is described, and its properties are derived. An implementation in hProlog is evaluated. The impact of representation sharing depends very much on the application. The cost seems acceptable and the memory gains can be large. Issues related to the representation sharing policy are discussed.

Representation Sharing for Prolog

Bart Demoen* and Phuong-Lan Nguyen[©]

* Department of Computer Science, K.U.Leuven, Belgium

[©] Institut de Mathématiques Appliquées, UCO, Angers, France

bmd@cs.kuleuven.be, nguyen@ima.uco.fr

Abstract. Techniques to reduce the memory footprint of an application include representation sharing between multiple copies of some data. Representation sharing has been implemented through hash-consing in functional languages. Representation sharing for Prolog was not given that much attention in the past. Here, representation sharing is defined for Prolog implementations in a structure copying environment (basically the WAM), and the specific issues arising from the logical variable and backtracking are analyzed. A high-level algorithm is described, and its properties are derived. An implementation in hProlog is evaluated. The impact of representation sharing depends very much on the application. The cost seems acceptable and the memory gains can be large. Issues related to the representation sharing policy are discussed.

1 Introduction

Data structures with the same value during the rest of their common life, can share the same representation. This is exploited in various programming language implementations, e.g. through hash-consing in functional languages [12] and by the *intern* method for Strings in Java. More recently, the *Flyweight pattern* offers a general way to share the representation of identical objects. Prolog implementations (as well as many other symbolic languages) perform such representation sharing at the micro-level by representing different occurrences of the same atom (or functor) by a shared representation in the atom (or functor) table: see for instance [1] how this is done. ECLiPSe [22] goes further: source code containing the same ground term several times, is compiled so that the term is represented only once and that representation can be reused many times. Whether achieved by a compile-time or a run-time technique, representation sharing can be crucial for the performance of an application. The representation sharing technique now known as hash-consing was invented by Ershov in [11] and used by Goto in [12] in an implementation of Lisp. Originally, hash-consing was performed during all term creations so that no duplicate terms occurred during the execution of a program. [3] explores the idea to use hash-consing only during generational garbage collection: the new generation contains non-hash-consed terms, and on promotion to the older generation, they are hash-consed.

The issue of representation sharing pops up from time to time in the Prolog context. Without an attempt to be complete, we mention:

- in 1989, the Diplomarbeit of Ulrich Neumerkel [13] mentions how by applying DFA-minimization to Prolog terms, certain programs can run in linear space (instead of quadratic)
- 1991: [17] ends with the sentence: *It still remains to be seen, however, what we meant by “folding identical structures”*
- a 1995 post in comp.lang.prolog from Edmund Grimley-Evans asks for more sharing in findall/3: see [14]
- in a 2001 Logic Programming Pearl [15], R. O’Keefe mentions a findall/3 query that could benefit from representation sharing in the answers
- in 2002, [8] gives a fresh view on garbage collection for Prolog and it details a number of desirable optimal properties of a garbage collector, one of which is the introduction of representation sharing (albeit naming it differently)
- in May 2009, Ulrich Neumerkel posted an excerpt of his Diplomarbeit in comp.lang.prolog and urged implementations to provide for more representation sharing, either during unification, or during garbage collection; he uses the term *factoring*; we prefer *sharing representation*

Representation sharing during findall/3 has been dealt with in [14], and we will here not get back to that.

AFAWK, BinProlog [19] is the only Prolog system that - depending on the release - introduces representation sharing during unification. The idea is that on unification of two terms, one is made to point to the other. However, Paul Tarau - author of BinProlog - has over the years switched between enabling and disabling that feature, because of its properties. We will not discuss this further.

Our aim is to describe the principles behind representation sharing for Prolog, a practical implementation and an evaluation.

There are two issues that make representation sharing in Prolog-like languages different from other languages: the logical variable and backtracking. Section 3 explores this by example and forms the intuition for the general concepts and definitions in Section 4. Section 5 shows when individual cells can share their representation and the approximation that works for us. Section 6 shortly discusses reference chains and the relation with variable shunting. Section 7 lifts the algorithm from Section 5 to compound terms. Section 8 discusses the properties of our notion of representation sharing. Section 9 discusses our implementation of representation sharing based on the concepts from the previous sections. Section 10 shortly discusses when representation sharing can be expected to be effective. Section 11 discusses the benchmarks and the experimental results.

While the section up to that point focus on representation sharing for structured terms, Section 12 shows extensions to the basic idea, variations and related issues. Section 13 discusses related work, and we conclude in Section 14.

One could try to adapt the ideas of [3], but this is not what we want to do: [3] adapts a copying collector to perform hash-consing for the data in the older generation. Prolog systems on the other hand typically have sliding collectors, the exceptions being hProlog and BinProlog. So we want to investigate representation sharing in a way that is independent of the details of the garbage

collector. [3] argues that garbage collection time is a good moment to perform hash-consing, but there is no inherent need to do it only then. Still, we agree in this with [3]: the effort that goes into representation sharing between terms that are dead is to be avoided.

In the sequel, we use as Prolog goals in the examples *share* and *gc*: the former performs representation sharing, the latter just performs garbage collection. By keeping the two separated, the issues become more clear. We make no assumptions on the workings of the garbage collector: both [4] or [6] will do, and also *look ahead* (see [8]) is fine.

In [5] it is pointed out that the combination of tabling and hash-consing is particularly powerful: the guaranteed uniqueness of the representation of a term can indeed be exploited by replacing calls to `==/2` by a single pointer comparison. However, hash-consing guarantees representation sharing all the time, while it will be clear from Section 3 that this is not our aim. It is a pity that [5] does not show experimental data for hash-consing alone.

We begin with a short overview of different forms of representation sharing in Prolog in Section 2. We expect the reader to be familiar with the WAM [23, 2].

2 Forms of Representation Sharing in Prolog

There are different forms of representation sharing in Prolog implementations, and some are more natural than others. Without an attempt to be exhaustive, we list a few of them:

- Prolog implementations usually have an atom table; it can contain information about the atom (e.g. its properties as an operator) and its representation as a sequence of characters; at run-time, only (tagged) pointers to this atom table are manipulated; the same can hold for function symbols
- in older implementations, the predicate *copy_term/2* also copies ground terms; in newer implementations - starting probably with SICStus Prolog - *copy_term/2* avoids copying ground (sub)terms; this means that the second argument can have some representation sharing with the first argument;
- some programs contain ground terms at the source level; e.g. a DFA emulator could contain a ground fact describing the transition table; another example is a goal like *member(Assoc,[fx,fy,xfx,xfy,yfx,xf,yf])*; ECLiPSe pre-allocates such ground terms, and makes sure that any time such a fact or goal is called, the ground term is re-used
- when two terms are unified, they can share a common representation in the forward execution; at various stages in its life, BinProlog enforced such sharing by (in WAM speak) redirecting the S-tagged pointer of one of the two terms, while (conditionally) trailing this change so that on backtracking it can be undone; if trailing is not needed, then the savings can be huge; otherwise, the locality of access can be improved, but memory and time savings can be negative; *le coeur de Paul Tarau* - the author of BinProlog - *balance entre les deux*: it is a mixed blessing

In each of these cases, the Prolog implementer explicitly caters for more representation sharing than naively would be the case. Application programmers also take care to share between terms. E.g. nobody likes the following definition of `append/3`

```

append([], [], []).
append([], [X|R], [X|S]) :- append([], R, S).
append([X|R], S, [X|T]) :- append(R, S, T).

```

because it does not exploit the chance to share some representation in the tails of argument 2 and 3.

[8] distinguishes between input sharing and output sharing: for the current paper, this is not an issue.

3 Representation Sharing in Prolog: Examples

A few examples show what we mean by representation sharing, and that it is different from hash-consing.

Consider the predicate `main1` and `main2` defined as

```

main1 :-
    X = f(1,2,3),
    Y = f(1,2,3),
    share,
    use(X,Y).

main2 :-
    X = f(1,2,3),
    X = Y,
    use(X,Y).

```

In a naive implementation, the execution of `?- main1.` just before the call to `share/0`, results in a memory situation as in the left of Figure 1. Clearly, since

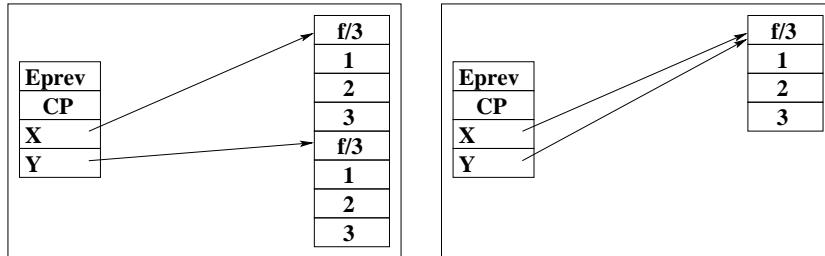


Fig. 1. Representation Sharing of two Terms in the same Segment

X and Y are exactly the same ever after their terms have been created, and so, they can share the same representation.

This is what we are aiming for: we do not want to change the representation of terms as hash-consing does, still we want to achieve the same amount of representation sharing.

The simple example above did not show any of the intricacies in the LP context: the terms are ground, no logical variables are unified, and no backtracking occurs. The next subsections will show how these issues affect the simple picture. In fact, the issues are all part of the same issue: the logical variable.

3.1 Sharing within the same Segment

The previous example showed the most simple case of sharing: the two terms are identical, in the same heap segment (as delimited by the HB pointers in the choicepoints) and ground at creation time.

The next example shows that ground terms in the same segment cannot always share their representation:

```
main3 :-
    T1 = f(a),
    T2 = f(X),
    ((X = a, foo ; write(T1 \== T2)).
```

Just before the execution of `foo`, the terms `T1` and `T2` are identical, and they are completely within the same segment. However, it would be wrong to make them share their representation, since in the failure continuation, they are no longer identical. Loosely speaking, the occurrence of trailed variables in a term, can make it unsuitable for representation sharing.

One could imagine that the introduction of representation sharing is implemented so that it is undone on backtracking. However, that is not our aim and we return to this issue in Section 12.9

3.2 Sharing between Segments

The previous examples dealt with representation sharing of terms that live in the same segment. The next example shows an issue with representation sharing of terms that live in different segments. Since we do not want to mix this issue with trailed variables, the example works with ground terms.

```
t :-
    T1 = f(a),
    (T2 = f(a), share, use(T1,T2)
    ;
    use(T1)
    ).
```

`T1` and `T2` live in two different segments. `T1` lives in the oldest segment, as seen in the left of Figure 2¹. Since `T1` is used after backtracking, the introduced sharing representation must be as in the right of Figure 2, i.e, the shared representation needs to be in the oldest segment. Alternatively, one could use as

¹ The dashed line indicates the heap segment barrier

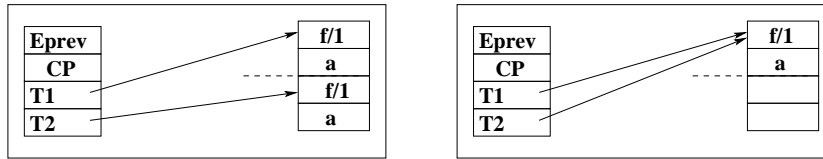


Fig. 2. Representation Sharing of two Terms in different Segments

shared representation the one in the younger segment, but then the heap should be frozen, so that on backtracking the value of T1 does not get lost.

A slight variation on the same example shows another issue:

```
t :-
    T1 = f(a),
    (T2 = f(a), share, gc, use(T2))
    ;
    dontuseT1
    ).
```

Consider two scenarios, one with sharing introduction and one without (i.e. in which the share is a nop).

- **no sharing:** at the point gc kicks in, T1 is unreachable and its representation disappears; this means that after backtracking to dontuseT1, the heap is empty
- **sharing:** share keeps one representation of f(a) and puts it in the oldest segment; gc cannot reclaim that representation, because T2 is not dead; after backtracking to dontuseT1, the f(a) term is still on the heap

This example shows that representation sharing between terms in different segments can lead to a higher heap consumption, or more need for gc.

4 Sharable Terms: Concepts

The examples in the previous section give some intuition on what we mean by representation sharing, and also about its pitfalls. Now it is time to turn to the basics. We will work under two assumptions:

- representation sharing is non-backtrackable
- there is no destructive assignment
-

Regarding the latter assumption, it is clear that if objects can be changed other than by unification, the safe approach is not to allow sharing. Most Prolog systems allow destructive updates on terms, so we make the weaker assumption

that mutable terms can be recognized: such terms are never subject to representation sharing. That assumption is correct in systems like SICStus, but not in hProlog or SWI-Prolog. However, see Section 11.4 for a way to deal with this.

Definition 1 (share): Terms T1 and T2 share their representation if T1 and T2 if their tagged topmost pointers are equal.

This definition is a bit vague about what sort of terms share. E.g. it is clear from the definition that in the context of the WAM two list terms could be sharing their representation. Extensions to the WAM in which e.g. bigints are represented by a new tagged pointer type are meant to be included in the definition.

We start with making more precise what it means that *terms T1 and T2 can share their representation*:

Definition 2 (can share): Terms T1 and T2 can share their representation (can share for short) if their sharing does not affect the answers computed by the process.

Whether T1 and T2 can share is *time*-dependent, as the following example shows:

```
t :-
    T1 = f(a),
    T2 = f(X), % now T1 and T2 cannot share yet
    something_deterministic,
    X = a,     % from now on, T1 and T2 can share
    ...
```

This means that we want to study whether two terms can share at a particular moment during the execution of a program: we will use the shorthand *at time t*, where time is supposed to increase during the execution.

Unfortunately, whether two terms can share at time t is not decidable. Here is a small example:

```
t(X) :- T1 = f(X), T2 = f(Y), g(X,Y), can_be_shared_or_not.

g(X,Y) :- some_computation, X = a, Y = b.
g(X,Y) :- some_other_computation, X = c, Y = c.
```

Depending on whether some_computation and some_other_computation succeed or fail, the terms T1 and T2 can share or not.

A rather weird consequence of definition 2 is that in some cases terms that are structurally different (e.g. have a different principal functor) can be shared. Here is an example:

```
t(T1,T2,L) :-
    T1 =.. [foo|L], T2 = [bla|L],
    here_T1_and_T2_can_share,
    use(L).
```

Also, any dead term can share with any live term, irrespective of its shape.

A more practical approximation of *can share* is clearly needed. To be explicit, we say T1 and T2 are *identical* if $T1 == T2$, where $==/2$ is the usual Prolog built-in.

Definition 3 (id-sharable): Two terms are id-sharable (at time t) if in the future of the computation (both forward and all backward continuations) they are identical as long as they are both needed.

This definition renders the T1 and T2 in the example of Section 3.1 not-id-sharable. Clearly

- if two terms are id-sharable, they can be shared
- if two terms are id-sharable at some point during the execution, then they are id-sharable at any later point
- id-sharable is a symmetric relation

Id-sharable is not decidable, and even worse, id-sharable is not transitive, i.e. if Term1 id-sharable Term2 (at time t) and Term1 id-sharable Term3 (at time t), then Term2 is not necessarily id-sharable Term3 (at time t). As an example: T1 could live from time 1 to 10, T2 from 1 to 20 and T3 from 1 to 30, and they can all be identical in the time interval 5 to 10. Then T1 and T2 are id-sharable (at time 5) and T1 and T2 are id-sharable (at time 5), but T2 and T3 could be different between 10 to 20, so T2 and T3 cannot be id-sharable at time 5.

A non-transitive notion is evil, because when sharing is implemented based on that notion, the order in which sharing is performed can be crucial, and detecting the optimal order is likely to be NP-hard.

If representation sharing between two cells c1 and c2 is allowed, actual sharing can be enforced. This results in a situation in which one of the cells (say c1) remains and the other (c2) disappears, because there are no more pointers to c2. We coin this *c1 absorbs c2*. Since the phrase *c1 and c2 can share their representation* is non-committal as to which cell absorbs the other, we need the notion *can absorb*. Ultimately, we want to implement an *absorber*.

As we will see later, the relation *can absorb* is not symmetric.

We do not define the notion *can absorb* further here: the next section explores representation sharing for individual heap cells and it will be clear what *can absorb* means.

5 Representation Sharing for Individual Heap Cells

It pays off to study the most basic representation sharing of all: between two individual heap cells.

Clearly, when two cells have a different contents, neither of them can absorb the other. And when the two cells have identical addresses, they have absorbed each other already. So, we are left with the possibilities that

- c1 and c2 are in the same heap segment or not

- c1 and/or c2 is trailed or not

To break some symmetry, we assume that c1 is older than c2.

This results in the 8 combinations, shown in Figure 3: a trailed cell is shaded. The contents of the two cells at the moment of the snapshot is the same, but shaded cells will be set to *free* (a self-reference in the WAM) on backtracking to the appropriate choicepoint. The horizontal dashed line indicates a heap segment separation. The vertical lines just separate the different cases.

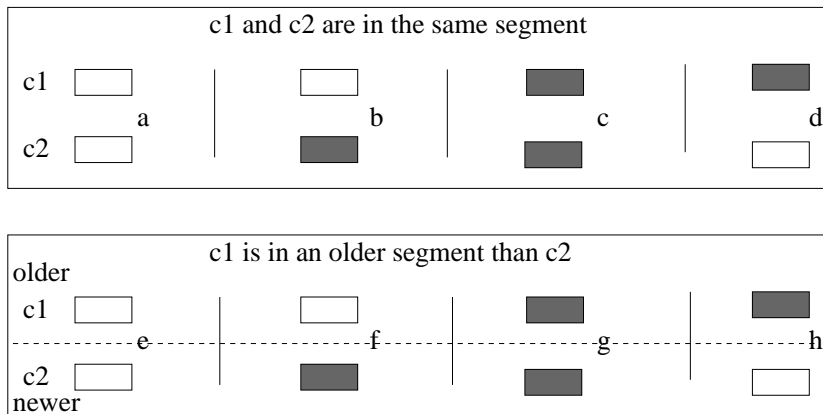


Fig. 3. The 8 combinations of two cells

- a:** c1 can absorb c2 and also vice versa, because the two cells have an identical contents, and that will remain so in the forward and in the backward computation
- bcd:** in the forward computation, the two cells remain identical, but not after backtracking; so no representation sharing can take place, and neither can absorb the other
- e:** on backtracking, c2 *dies* before the older cell c1, but for the duration of their common life, the two cells are identical, so representation sharing is allowed: c1 can absorb c2, but not the other way around, unless the segment of c2 is not reclaimed on backtracking
- fg:** these have in common that the newer cell is trailed: backtracking results in setting the newer cell to *free*, and both cells are still live at that moment, and they have a different contents; therefore representation sharing is not allowed; neither can absorb the other
- h:** there are two possibilities now:
 - (a) at the moment the older cell is untrailed, backtracking also recovers the segment in which the newer cell resides; this means that the newer cell dies, so the fact that the older cell is set to *free* does not prevent representation sharing; so c1 can absorb c2 (and not the other way around);

this happens if *c1* was trailed before the segment of *c2* is final, or to put it differently: if the moment of trailing *c1* is not after the segment of *c2* is closed by a choicepoint

- (b) otherwise, representation sharing is disallowed; neither cell can absorb the other

The only liveness information we are allowed to rely on is that a cell in an older segment lives longer than a cell in newer segment. We have used that in case h.

The above suggests that it is in the interest of maximizing the chances for representation sharing to keep the trail *tidy*: this is in many Prolog systems done at the moment a cut (!/0) is executed. Also during garbage collection, the trail can be tidied.

The above analysis can be used to specify a decision procedure on whether one cell can absorb another. Already anticipating an implementation, we notice that it is important to be able to check quickly whether a cell is trailed, and even *in which segment* a cell resides or is trailed. One bit is enough for checking whether a cell is trailed: that bit could be in the heap cells themselves, or it could be allocated in an array parallel to the heap. Retrieving the heap segment number from a heap address can be done with a binary tree as in our segment preserving copying garbage collector. Retrieving the heap segment number in which a cell was trailed can be done similarly: one just preprocesses the trail into an appropriate tree. Still, this might be an overkill², and since the number of heap segments in all our benchmarks is one (except for chess) we have decided in our implementation to disallow representation sharing also in case h(a), i.e. we follow a more restrictive definition of *can absorb*. This leads to a simpler decision procedure: *c1* and *c2* are pointers to heap cells.

```
boolean can_absorb(cell *c1, cell *c2)
{
    if (c1 == c2) return(FALSE);
    if (*c1 != *c2) return(FALSE);
    if (trailed(c1)) return(FALSE);
    if (trailed(c2)) return(FALSE);
    return(TRUE);
}
```

If cell *c1* can absorb cell *c2*, every (tagged) pointer to *c2* can be changed into a (tagged) pointer to *c1*: this change does not affect the outcome of the execution. Note that it is immaterial whether the cell containing the (tagged) pointer to *c2* is trailed or not.

Note the similarity of the above analysis with the one for variable shunting in [17].

² We come back to this in the Appendix

6 Reference Chains

A consequence of the WAM term representation is that chains of references can be created, notably when free variables are unified. Often, such chains can be collapsed, as described in [17] and this is typically performed at garbage collection time. Our representation sharing module (in the future named *sharer* for short) does not deal explicitly with such chains as we are primarily interested in sharing compound terms. However, two remarks need to be made

- it can be beneficial to run a variable shunting pass, and as far as representation sharing goes, it is never a bad thing
- it is clear that trailed reference chain cells prevent sharing up to the trailed point, and allow it after the last trailed cell, but our algorithms will not pursue this; in particular, our algorithm is only concerned with absorbing the bodies of compound structures³

7 Representation Sharing for Compound Terms

The representation of a compound term with principal functor `foo/n` in the WAM is an S-tagged pointer to an array of $(n+1)$ contiguous heap cells, the first of which contains `foo/n`, and the next n cells contain one cell of the representation of one argument each. We name this array of $(n+1)$ heap cells the *body* of the term.

The idea of one term absorbing the other, is that after absorption, there is only one body instead of two - but there are still two cells with an S-tagged pointer pointing to it. See Figure 4.

It is clearly a necessary condition for such representation sharing is that the two bodies have the same contents. Moreover, since each body belongs to a segment, the condition worked out for absorption for two individual cells must hold for each pair of corresponding body elements. Figure 4 shows two bodies that fulfill the conditions.

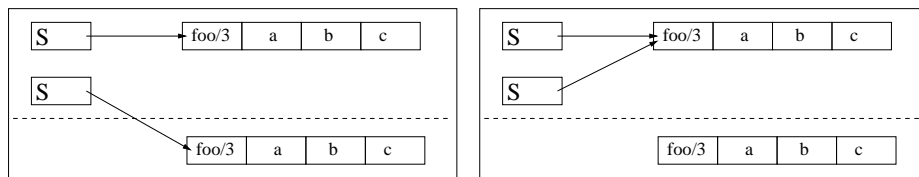


Fig. 4. Left: the bodies fulfill the conditions for representation sharing. Right: sharing has been performed

³ Our implementation also performs representation sharing for doubles, strings and bigints: those follow the same principle

An algorithm that given two terms decides whether one can absorb the other is now easily constructed. However, the naive use of this algorithm would be very inefficient - see the Appendix for more on this.

8 Properties of *can absorb*

Before going to the implementation of representation sharing, it is good to understand some properties of the *can absorb* relation: the optimality (if any) of our algorithms depends crucially on those properties.

It is clear that *can absorb* is not symmetric: a newer term can not absorb an older term in a different segment.

Neither is *can absorb* anti-symmetric: case a in Section 5 shows that.

In the sequel we denote by *absorbed(x,y)* the fact that term x has absorbed term y - of course under the condition that x can absorb y.

With the more restricted definition of *can absorb* (i.e. no sharing for case h(a) in Section 5, the most important property of *can absorb* is

`if (can absorb(p,q)) then no cell of q is trailed`

That means that candidates for absorbing another term can be recognized without knowing the other term. From this property it follows that

`absorbed(absorbed(x,y),z) == absorbed(x,absorbed(y,z))`

under the condition that x can absorb y and y can absorb z.

This is particularly important for our implementation, because it allows to scan the heap while treating each term only once. See the Appendix for a discussion on how to treat the finer notion of *can absorb*.

9 Implementation of Representation Sharing

We have taken hProlog as the platform for an implementation of representation sharing. hProlog resembles strongly the WAM [23, 2] with a few differences:

- the choicepoint stack and environment stack are not interleaved as in the WAM
- free variables only reside on the heap; i.e. there are no self-references in the environment stack
- hProlog supports some more native types like char, string and bigint; it also uses an unusual representation for attributed variables

hProlog employs a mark-copy type of garbage collector, with its roots in [6], but preserving segment order as described in [21]. Most other systems use a sliding collector based on [4]. hProlog does not implement variable shunting. The main reason is that its representation of frozen variables does not lead to

reference chains (see [9] for more info on this) and for plain Prolog (i.e. without delayed goals) in practice, reference chains are short [20].

hProlog is a direct descendant of dProlog [10]. Its purpose is to offer a platform for experiments in WAM-like Prolog implementation. Its high performance gives the experiments an extra dimension of credibility.

The implementation uses two data structures: they can be seen in Figure 5. Because of lack of inspiration, we named them INFOHEAP and INFOTABLE.

- INFOHEAP: this is an array the size of the WAM heap (or global stack) and can be thought of as parallel to the heap; its entries contain information about the corresponding heap cells; the information is one of the following three:
 - **no-info**: the corresponding heap cell has not been *treated* yet
 - **impossible**: the corresponding heap cell cannot participate in representation sharing; see Section 9.3 for more on this
 - a pointer to the INFOTABLE: the corresponding cell has been treated, and its sharing information can be found by following the pointer
- INFOTABLE: this data structure contains records with two fields; suppose a pointer in the INFOHEAP points to a record in the INFOTABLE, and the corresponding heap cell A is the entry point of term T, then
 - the *hashvalue* field in the record is the hash value of term T
 - the *term* field in the record is a (possibly tagged) pointer to a heap cell B that is the entry point of a term S that can absorb T; our implementation make sure that the heap cell B is as old as possible

Treating a heap cell consists in filling out the corresponding cell in the INFOHEAP and possibly the INFOTABLE.

The implementation of the INFOTABLE is actually as a hash table: the hashvalue of a term modulo the size of the hash table is used for determining the place in the INFOTABLE, and a linked list of buckets is used to resolve collisions.

Our first description of the algorithm only tries to introduce sharing between structure (not lists). Therefore, for now, INFOTABLE pointers can only appear in cells corresponding to a heap cell containing a functor descriptor.

The main algorithm consists of two phases:

- **build**: building the INFOHEAP and INFOTABLE - in this phase nothing is changed to the heap
- **absorb**: perform all absorption possible by using the INFOHEAP and INFOTABLE

In the algorithms below, we use *beginheap* and *endheap* for the pointers to the first (oldest) cell in the heap and the last (newest). We assume no cell is trailed, and come back to this point later.

9.1 Phase I: building the INFOHEAP and INFOTABLE

The build phase performs the action *compute_hash* for each cell in the heap. The effect is that the corresponding cell in the INFOHEAP is set to one of *impossible* or to a pointer to the INFOTABLE:

```
foreach p in [beginheap, endheap] and is_functor(*p)
    compute_hash(make_struct(p)); // ignore return value

int compute_hash(p)
{
    switch tag(p)
    {
        case ATOMIC:
            return(p);

        case STRUCT:
            p = get_struct_pointer(p);
            if (already_computed(p)) return(already_computed_hash(p));
            hashvalue = *p;
            foreach argument of structure p do
                hashvalue += compute_hash(argument);
            save_hash(hashvalue,p);
    }
}
```

The particular hash value computed above is not relevant for our discussion: in practice, there are better (more complicated) ways to compute hash values of terms.

The function call *already_computed(p)* checks whether the corresponding element in the INFOHEAP points to the INFOTABLE. *already_computed_hash(p)* returns the hash value previously computed (for the term starting at p) from the INFOTABLE entry corresponding to p: in this way, re-computation (and re-traversal of the same term) is avoided.

In the next *save_hash* function, we have left out collision handling - just assume that the hashing is perfect.

```

save_hash(hashvalue,p)
{
  index = hashvalue % length(INFOHEAP);

  INFOHEAP[p-beginheap] = INFOTABLE + index;

  if (empty(INFOTABLE[index]))
  { INFOTABLE[index].term = p;
    INFOTABLE[index].hashvalue = hashvalue;
    return;
  }

  // a non-empty entry might need to be adapted
  if newer(INFOTABLE[index].term,p) INFOTABLE[index].term = p;
}

```

The last line in `save_hash` makes sure that the term pointed at in an `INFOTABLE` entry is as old as possible. The reason is that it is generally safe to let an older term absorb a younger one.

Figure 5 shows how three equal terms are treated by `compute_hash` and the effect thereof on the `INFOHEAP` and `INFOTABLE`.

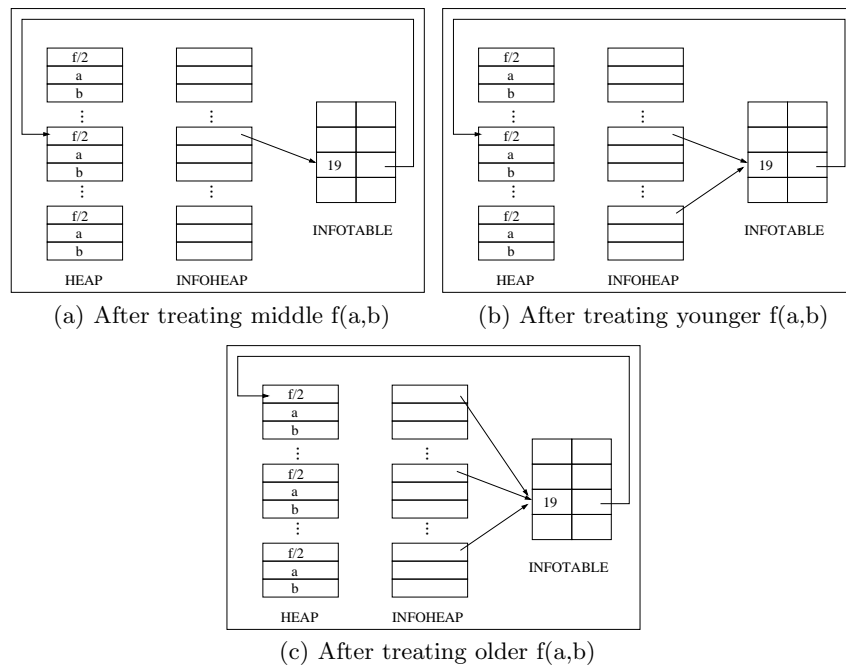


Fig. 5. Three identical terms are treated

9.2 Phase II: Absorbing

The absorb phase performs the actual representation sharing: a S-tagged pointer is redirected to the oldest term body that can absorb it. The code is very simple:

```

foreach struct pointer p in heap
    in local stack
    in choicepoint stack
    in argument registers do
{
  q = get_funct_pointer(p);
  replace p by struct_tag(hash_heap[q-beginheap]->term);
}

```

Figure 6 shows how the older term absorbs the two identical younger terms.

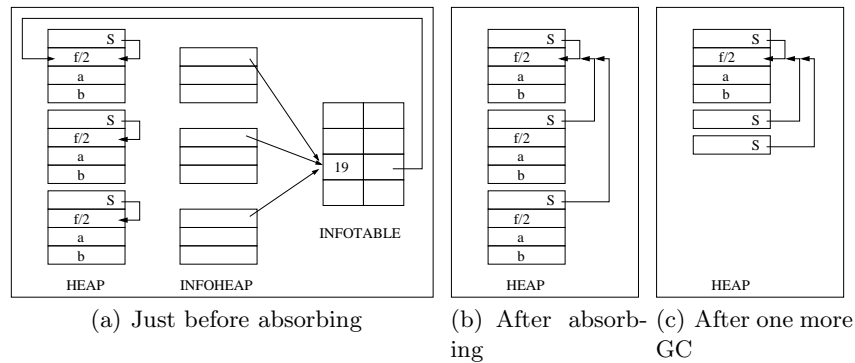


Fig. 6. Absorption and GC in action

9.3 Comments on the Code

- we have omitted the check whether a heap cell is trailed; those checks can be inserted easily and whenever a trailed cell is found, the computation of a hash value is stopped; in practice, we use in the INFOHEAP one value (*impossible*) to indicate that the corresponding cell is trailed; that makes the check constant time
- we have omitted any dereferencing from the code, but it should be clear that it is needed in a number of places; during dereferencing, the check whether any intermediate cell is trailed needs to be made
- the code takes into account only non-list structured terms and atoms; it is easy to extend it for variables and other types that occupy a single cell; for some of the other types (real, string, bigint in casu) we have followed the same principle as for non-list structured terms: those types are implemented

roughly like such terms, i.e. with a tagged pointer to a header on the heap which is followed by the actual value that can span several heap cells; for lists, we have a different solution: see Section 9.4

- our actual implementation uses a linear scan for the *foreach* constructs: this is possible for all the stacks in hProlog; if this is not the case, one can traverse the life data starting from the root set as the garbage collector does

The code implementing the above is less than 600 lines of plain C that reuses very little previously existing code.

Note that in the context of our copying collector, the extra space needed for representation sharing is just the INFOTABLE: the INFOHEAP has exactly the same size as the collector needs for performing its collector duties.

9.4 Representation Sharing of Lists

In the WAM, lists have no header like other terms: an L-tagged pointer points to two consecutive heap cells containing the first element of the list and its tail respectively. Clearly, we cannot deal with lists as in the previous algorithm. The change is however small: we keep the INFOTABLE pointer in the cell corresponding to the list-pointer. Figure 7 shows an example with just lists.

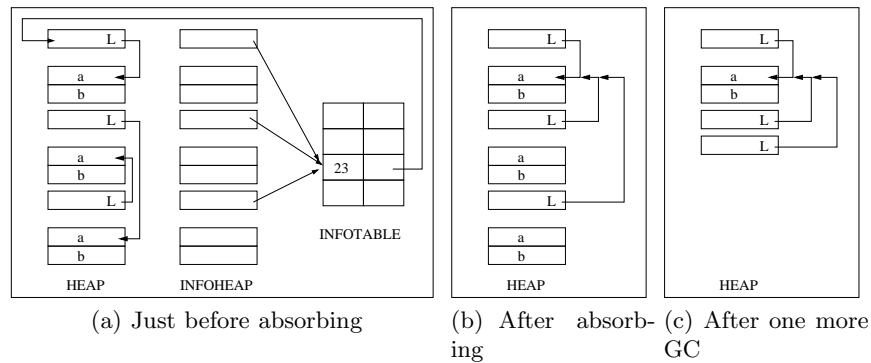


Fig. 7. Absorption and GC in action

Note that functor-cells can only appear on the heap, while list-pointers can occur also in the environment and choicepoint stack. As a result, with just a INFOTABLE pointer array parallel to the heap, some representation sharing in the other stacks can get lost. A similar INFOTABLE pointer array parallel to the other stacks can solve this problem: our implementation does not do that.

9.5 When to run the Sharer

It seems obvious that the sharer must be run either during GC, or just after GC. Our sharer can be adapted to run during GC because hProlog has a copying collector, so the build phase of the sharer can be integrated in either the marking

phase of the collector. The absorb phase can be run before the GC copy phase, or be integrated with it. That would lead to a (mark+build)&(copy+absorb) collector. In a sliding GC context, this would become (mark+build)&(compact+absorb). Still, we choose from the beginning to run the sharer as an independent module that could actually be run at any time. Just after GC seems the best, because at that moment, the heap has minimal size. We name that policy *after GC*.

There is one snag in this: the space freed by the sharer cannot be used immediately, and only after the next GC the beneficial effect of the sharer can be seen. Therefore, it feels like immediately after the sharer, another GC should be done. We name that policy *between GC*.

We have therefore added an option to hProlog:

- -r0: no sharing
- -r1: sharer with policy after GC
- -r2: sharer with policy between GC

Note that the absorb phase could estimate the amount of space it has freed, and the decision to run the next GC immediately can be based on that.

10 The Effectiveness of Representation Sharing

Why does a term occur more than once in the heap? The most obvious reason is that the same computation is performed more than once (and in conjunction with each other). If this is the case, tabling [7] can be used: it avoids the duplicate computation, but as usually implemented, it does not avoid duplicate terms. Duplicate terms also occur when the computation produces the same term along different conjunctive branches without doing the same computation. A typical example is the benchmark *boyer* (see Section 11.1): it is a term-rewriting program where formulas are rewritten to a canonical form involving many times the functor *if/3*. Such a rewriting has a good chance to result in the same *if/3* structure occurring many times in the final result. Indeed, we have not come across a program that benefits from representation sharing unless it is of that type. It is ironic that such rewriting programs typically also benefit from tabling.

life (see Section 11.3) is another type of rewriting program: a term is rewritten repeatedly, but not every initial term leads to a fix-point, i.e. the process keeps going, perhaps repeating itself, but it does not converge. In that case, one expects little from representation sharing immediately after a major collection, because the just rewritten terms are garbage.

11 The Benchmarks and the Results

The work most closely related to ours is [3]. It employs a number of benchmarks, and of course, we were inclined to use the same. What follows is a short discussion of some of the benchmarks in [3], and some other benchmarks we used additionally.

11.1 Boyer

Boyer is a famous benchmark initially conceived by R. Gabriel for Lisp, and later used in other functional and logic contexts. Essentially, it rewrites a term to a canonical form. Boyer has been the subject of many studies, and in particular for proving that it is not a good benchmark: see for instance [5]. Anyway, in [3], this benchmark shows the best results for hash-consing. A closer look at boyer is in order. The rewriting is performed depth-first:

```
rewrite(Old,New) :-
    (atomic(Old) ->
        New = Old
    ;
        functor(Old,F,N),
        functor(Mid,F,N),
        rewrite_args(N,Old,Mid),
        ( equal(Mid,Next) ->
            rewrite(Next,New)
        ;
            New=Mid
        )
    ).
```

A small adaptation of this code can already get rid of a lot of duplicate terms:

```
rewrite(Old,New) :-
    (atomic(Old) ->
        New = Old
    ;
        (
            functor(Old,F,N),
            functor(Mid,F,N),
            rewrite_args(N,Old,Mid),
            Old \== Mid ->
                true
        ;
            Old = Mid
        ),
        ( equal(Mid,Next) ->
            rewrite(Next,New)
        ;
            New=Mid
        )
    ).
```

This small change reduces the total heap high water mark for boyer (without garbage collection) from 184.232 to 15.957 entries (hProlog specific). This reduction seems huge, but even more impressive is that with representation sharing the number of heap cells of the final result is less than 200, while without it,

39.834 heap cells are needed to represent it. The inherent reason is that terms are rewritten to a canonical form, and originally different terms end up the same.

This makes boyer close to an optimal benchmark for showing the effectiveness of representation sharing.

11.2 Mandelbrot

This benchmark computes - actually outputs - a bitmap of a Mandelbrot set of a given dimension. Since the output does not play a role in the heap usage, we have removed the code for the output. We took the version from the *Computer Language Benchmarks Game* (<http://shootout.alioth.debian.org/>) written for Mercury and based on a version by Glendon Holst. Mandelbrot uses quite a bit of heap and as such appears a good memory benchmark. However, one can see quickly that literally **all** memory used by Mandelbrot is by floating point numbers: floating point numbers have the tendency to be different and therefore representation sharing might not have much effect. We have indeed checked that of the first 88708427 generated floating point numbers, at least 48156432 are different.

Even worse is that this benchmark only produces those floating point numbers as the result of a *test*. To be more precise, the predicate `mandel/5` is called with ground integer arguments, is `semidet`, and it performs all the floating point computation. Wrapping it in a double negation has exactly the same semantics, and results in a benchmark that runs in less than 10K heap cells, instead of needing about 75 garbage collections (collecting 1.499.849.443 cells) when given initially 20M heap cells ! Clearly, the improved version makes no sense as a benchmark as it uses almost no space, so we discuss further only the original version.

In the setting of [3] (generational collection + hash-consing) the original version has the following characteristic: if the garbage collector runs during the test (`mandel/5`) then a few floats are copied to the older generation, otherwise, no float from the new generation survives the collection. So, not even all computed floats end up in the zone subject to hash-consing.

In our setting (only major collections + representation sharing), at each collection, only some floats in the test are alive. Exactly at that moment, the chance for duplicates is very small, and representation sharing is expected to be nil.

Our test runs of the mandelbrot benchmark show indeed zero gain from representation sharing. As an extra piece of information: the live data at any moment during the execution of the program, is less than 30 heap entries.

11.3 Life

The Game Of Life is well known and also [3] uses it as a benchmark. We have written a version in Prolog following the ideas of Chris Reade [16], just as [3] did. A (live) cell is represented as a tuple in coordinate form (X,Y) . Since cells

can survive from one generation to the next, it is clear that one generation can share with the next. However, this sharing is already performed by the original program, and more representation sharing cannot be expected.

In a generational system, there could be some gain from representation sharing, because some terms in the old generation could be dead, but still share with terms that are added later to the old generation. Still, [3] shows little gain from hash consing for this benchmark. Our measurements show no gain at all.

11.4 Knuth-Bendix

[3] uses an ML/NJ version of an implementation of the Knuth-Bendix algorithm written by Xavier Leroy⁴. Xavier send us his OCaml version and we started translating it to Prolog. We expected a pure functional program, but it uses the impure *ref* feature, and to mimic that, we planned to use `setarg/3`, which in other Prolog systems is replaced by the concept of mutable variables. Since an updatable data structure cannot share its representation, we have introduced in hProlog a new declaration to this effect. E.g., the declaration

```
:- norepshar(foo(_,_,_)).
```

prevents the representation sharing code to introduce sharing between terms with the functor `foo/3` and any terms containing such terms. Testing this introduces a negligible overhead during representation sharing. Our translation of the OCaml code to Prolog was not finished at the time of writing this report, but the `norepshar` declaration remained.

11.5 tsp, chess, serial, dnamatch, browse, emul

The benchmarks mentioned in the section title all come from established general or garbage collection benchmark suites. None of them shows any benefit from representation sharing.

11.6 Worst and best Case

It is not clear what the best and worst case for our sharer is: if the heap were just one huge flat term (say of the form `f(1,2,3,...)`) then only one hash value would have to be saved in the INFOTABLE, and in some sense that is both best and worst, because the least time is lost in collisions etc, but also no sharing can be performed. So we choose the following as best-versus-worst case: a large complete binary tree in which every node is of the form `node(tree,tree,number)`. In what we consider the best case, the number is always the same (and thus resembles a bit the `blid` data structure in Section 11.7). This leads to a very sparse INFOTABLE, and a large amount of sharing. In the worst case, the number is different in all nodes: as a result, the INFOTABLE becomes quite

⁴ Personal Communication, October 2009

full, but no sharing is possible at all. The main reason for this benchmark is to find out how the build and the absorb phase contribute to the total time of the sharer.

11.7 blid/1

The following program was altered slightly from what Ulrich Neumerkel posted in comp.lang.prolog.

```
blid(N) :-
    length(L, N),
    blam(L),
    id(L,K),
    use(K).

blam([]).
blam([L|L]) :-
    blam(L).

id([], []).
id([L1|R1], [L2|R2]) :-
    id(L1,L2), % L1 = L2
    id(R1,R2). % R1 = R2

use(_).

?- blid(24).
```

His question was *Are there systems, that execute a goal blid(N) in space proportional to N? Say blid(24)*. At first we expected that with our representation sharing, space would be indeed linear in N. However, the expansion policy and order in which events (garbage collection and representation sharing) take place is also crucial.

- with the *between GC* policy, the following happens:
 1. the first GC finds that 99% (or more) of the data is live, and decides to expand the heap
 2. the sharer shares most data
 3. the second GC collects almost all data
 4. points 1,2 and 3 are repeated
- with the *after GC* policy, the following happens:
 1. the first GC finds that 99% (or more) of the data is live, and decides to expand the heap
 2. the sharer shares most data
 3. the next triggered GC finds that about half of the heap is live, so does not expand
 4. the following sharer shares most of the data
 5. points 3 and 4 are repeated

The first GC is triggered by lack of space, the second GC is there by policy. A GC can decide to expand the heap (in hProlog when the occupancy is less than 70%: this is known after marking). So one sees that in the case of the *between GC* policy, the heap is repeatedly expanded, even though the program could run in constant space (with the aid of the sharer). With the *after GC*, we do not get into this repeated expansion.

This shows that the combination of a reasonable heap expansion policy and a reasonable sharer policy can result in an overall bad policy. More work could be done on this.

Note that in its original form, id/2 also contains the two commented out unifications, and that these would with unification factoring also introduce the sharing needed to run in $O(N)$ heap (with the aid of GC of course).

11.8 Timings for Representation Sharing

We have run all the benchmarks previously mentioned (with the exception of Knuth-Bendix), and the qualitative results are mentioned in Section 11. We show just a few quantitative results.

The columns denote the number of GCs, the total time taken by GC (in milliseconds), the total time taken by the sharer (the sum of the two phases), the number of bytes the GCs collected, the heap high water mark at the end of the benchmark run, and the highest water mark during the run.

Below, = means that for that same benchmark, the figure is the same (and otherwise irrelevant).

11.9 Space and relative Cost of the Phases

We also want to give an idea about the space required to run the sharer. The *fixed* cost is the INFOHEAP: its size is the size of the live data. The variable cost (INFOTABLE) depends on the amount of actual possible sharing and the amount of terms (lists, ...) in the heap. The table gives also some insight in the relative (time) cost of the build and absorb phase in the sharer.

The size of the INFOTABLE can determine the performance of the sharer. This became apparent when we devised the worst/best case artificial benchmark: the worst case improved by a factor of 70 by choosing an INFOTABLE size depending on the size of the heap instead of a naively fixed small size. Even better would be to let the size dynamically grow on need.

11.10 Conclusions from the Benchmarks

By and large, our results confirm the findings of [3]: most benchmarks hardly benefit from representation sharing, and sometimes the space and time performance gets worse. Apart from the artificial benchmark blid/1, only for boyer do we find a much larger - huge in fact - benefit from representation sharing than in [3]. We have not been able to pinpoint why: the benchmarks used in [3]

	#gc	time gc	time sharer	bytes collected	final memory	high mark
tsp -r0	38	0	-	=	=	=
tsp -r1	38	10	0	=	=	=
tsp -r2	76	10	10	=	=	=
boyer -r0	20	20	-	1484288	39726	74576
boyer -r1	20	10	0	1642368	206	28829
boyer -r2	40	10	10	1642480	178	22340
blid -r0	6(6)	2100	-	1228	33554482	33554482
blid -r1	33(1)	1172	2140	127610900	1652064	1994002
blid -r2	10(5)	1000	2068	123541892	2669316	15937994
mandel -r0	303	10	-	=	=	=
mandel -r1	303	30	0	=	=	=
mandel -r2	606	50	0	=	=	=
life -r0	10000	480	-	=	=	=
life -r1	10000	390	150	=	=	=
life -r2	20000	790	150	=	=	=
worst -r0	1	56	-	1740	2097163	2097592
worst -r1	1	52	352	1740	2097163	2097592
worst -r2	2	120	352	1740	2097163	2097592
best -r0	1	60	-	1692	2097163	2097580
best -r1	1	60	116	1692	2097163	2097580
best -r2	2	60	112	8389980	91	2097580

Table 1. Time and Space of GC and Sharer

are not even available anymore, let alone the *queries*. The fact that generational collection retains terms longer than an only major collections strategy might play a role. Still, given the rewriting character of boyer, our result is not totally unexpected.

The time taken by our implementation of representation sharing is reasonable: the algorithm is linear in the size of the heap, localstack, choicepointstack and trail, so complexity wise not worse than an actual garbage collection. The traversal of the stacks is however less complicated, since one does not need to take into account the liveness of the locations anymore and less copying is going on. In the blid/1 benchmark, the sharer takes about twice the time of the GCs, in life it is less than half and in our worst case, up to about 7 times as much. That might seem huge, but the answer to that is threefold: a good policy is necessary; our garbage collection times are about 3 to 4 times lower than in systems like SICStus, Yap or SWI, mainly due to its copying nature; finally, there is surely room for improvement of our code. Other systems would probably hardly be slowed down by the adoption of our sharer.

	allocated bytes INFOTABLE	allocated bytes INFOHEAP	ratio build/absorb
tsp -r1	7657	42732	-
boyer -r1	5196	18348	-
blid -r1	200658	3988283	8.3
mandel -r1	4063	78	-
life -r1	4716	602	-
worst -r1	6491552	8388628	2.6
best -r1	200336	8388628	4.5

Table 2. Space and relative Speed of Sharer

12 Variations, Extensions and related Issues

12.1 List Pointers not in the Heap

The observant reader has certainly noticed that list pointers in the environments, in choicepoints and in the argument registers are not adapted in our implementation of the sharer. To do this, it suffices to introduce an array parallel to those regions similar to the INFOHEAP parallel array: the space needs become higher, and the gain can be arbitrary. A more complicated way to do this, is by letting the INFOHEAP cell corresponding to a first element of a list, play a double role, namely the role of itself as a term, and also the role of the list body: this complicates a little the management of INFOHEAP cells, but we think it is a good solution and plan to explore it in the near future.

12.2 Unusual Sharing

In [8], the following rather unusual representation sharings are described:

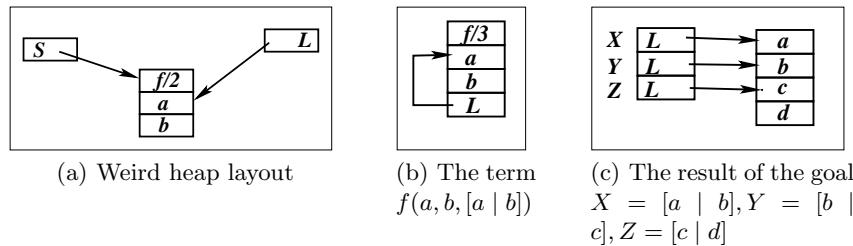


Fig. 8. Unusual representation sharing

The representation sharing implementation we described earlier does not achieve the above sharings. Still, all ingredients are present and can be used as

follows: for convenience define a two-cell as two consecutive cells in the heap; clearly, every list-tagged pointer points to a two-cell.

```
foreach two-cell, find the oldest occurrence of a sharable two-cell
[that is at the same time part of the arguments of a structured term
(and not a list)]; this is done by computing a combined hash value and
further using similar techniques as before

finally, redirect each list-tagged pointer to the oldest two-cell
sharable with its arguments
```

This clearly needs quite a bit of space for the hash table, and the expected gains are small. But it is nice to see that it can be achieved in (expected/average) time linear in the size of the heap.

12.3 Cyclic Terms

[3] describes how to deal with cyclic terms: we have not catered for that, but our implementation could be adapted accordingly.

12.4 Value Trail Entries

Prolog systems supporting destructive update - setarg/3 like or through mutable terms - often do this using a trail in which each entry keeps the old value: clearly, these old values can point to sharable terms and they can be updated accordingly in the final absorb phase.

12.5 Cooperation between Collector and Sharing

We have implemented the representation sharing module independent of the garbage collector module. The advantage is less dependency. The disadvantage is that some information that the garbage collector has computed, needs to be recomputed by the sharing module. For instance, the collector could leave behind information on which cells are trailed, and which cells contain sharable information. This would speed up the sharer.

12.6 When Representation Sharing does not work

The benchmark programs show that representation sharing is not always effective: it depends indeed highly on the type of program. When representation sharing does not work, this can be noticed during a run of the representation sharing module by observing the INFOTABLE. If it keeps growing, it means that lots of different terms are found. This in turn gives an indication that representation sharing is not effective. An important advantage of our implementation is that the representation sharing process can be abandoned at any time since no changes to the WAM run-time data structures are made until the final phase

in which structure (or list ...) pointers are updated. Similarly, if representation sharing is run from time to time - as suggested by Ulrich Neumerkel - then the frequency of running it can take into account the effectiveness of representation sharing up to that moment. Such tuning must depend also on the relative performance of the garbage collector and the representation sharing module.

12.7 Parallelization

It seems that a quite naive parallelization of the sharer is possible: the heap is read-only for all workers, the INFOHEAP and the INFOTABLE are (entry-wise) of the type many readers, one writer. By giving different workers a different part of the heap to start working on, duplicate work might be avoided and synchronization slowdown kept low.

12.8 Variable Chains once more

We have not treated variable chains at all, as we were mostly interested in sharing between the bodies of compound data. However, a slight extension of the code for the build phase can also call `save_hash` for all reference cells. That results in a similar effect as variable shunting as described in [17], but is not as *complete* as the method described there. Figure 9 shows an example of how a chain of references is transformed.

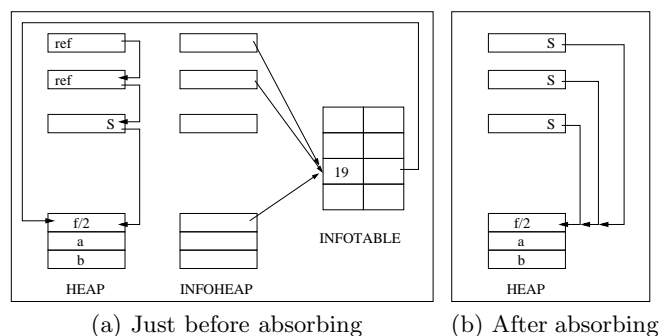


Fig. 9. Absorption for chains of references in action

12.9 Backtrackable Representation Sharing

Backtrackable representation sharing would follow the principle that when two terms are identical (as for `==/2`) then one can absorb the other: there is no need to take into account what is trailed. The change made (to a LIST or STRUCT-tagged pointer) is conditionally (and value) trailed. On cut, the trail is tidied, so

in case the computation becomes eventually deterministic, the amount of sharing can be arbitrarily larger than without backtrackable representation sharing. However, suppose that all sharing were trailed, then an immediately following GC would not be able to recover anything. And if the computation becomes deterministic eventually, running the sharer will do the same job as was done in the case of the backtrackable representation sharing, but later - which might be even better, because the earlier sharing could have been unnecessary because backtracking has destroyed it. All in all, our feeling is that backtrackable representation sharing is not worth its while, but a more thorough evaluation is needed.

13 Related Work

[3] describes how hash-consing can be performed during garbage collection in an implementation of Standard ML (SML/NJ). The collector is generational, and the data structures in the old generation are hash-consed. In this way, the operation of hash-consing is restricted to data structures that are expected to live long. The reported performance and space gains are disappointing: half of the benchmarks lose performance (up to 25%) and the gain is maximally 10% (for boyer). The space improvement is even smaller: on most benchmarks less than 1%. Also here, boyer is the exception with about 15%. Note however, that these space figures are about the amount of data copied to the older generation, i.e. the data that is hash-consed, and which is collected infrequently. As such, these numbers do not give full insight in the power of hash-consing. Still, [3] is most closely related to our implementation of representation sharing for Prolog: our strategy is to perform representation sharing after a (major) garbage collection, so we introduce sharing only for data that survived a collection.

Mercury [18] is basically a functional language, and the issue of trailing does not enter. In the developers mailing list in August 1999, the issue of hash consing is raised and a proposal for an implementation as well as how to present it to the user. It is interesting that at some point, the opposite of our `norepshar/1` declaration is proposed. As an example `:- pragma hash_cons(foo/3)` tells the compiler to hash-cons the constructors of type `foo/3`. As far as we know, the proposals were not implemented.

There is also a 1995 post in `comp.lang.prolog` from Edmund Grimley- Evans asking for more sharing in `findall/3`: see [14] for more on this.

Last but not least, [13] provides the example `blid/1`, and gives a high-level outline of an algorithm for minimization of heap terms seen as DFAs. Our implementation can be seen as a concrete version of that algorithm. However, our *minimization* shows mostly similarities with [11] in which Ershov uses (for the first time in the published history of computer science) hashing to detect common subtrees in in a given tree.

14 Conclusion

Without the questions by Ulrich Neumerkel on `comp.lang.prolog`, we would not have worked on this topic. We have provided a practical and efficient implementation of representation sharing, that can be incorporated without problems in most WAM based systems. Our implementation has the advantage that it does not rely on a particular garbage collection strategy or implementation. On the other hand, a tighter integration of the garbage collector with the representation sharing module can make the latter more efficient. Still, representation sharing is not effective for all programs, so it must not be applied indiscriminately, i.e. it needs its own policy.

References

1. H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.
2. H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
3. A. W. Appel and M. J. R. Goncalves. Hash-consing Garbage Collection. Technical Report CS-TR-412-93, Princeton University, Feb. 1993.
4. 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.
5. H. G. Baker. The boyer benchmark at warp speed. *SIGPLAN Lisp Pointers*, V(3):13–14, 1992.
6. J. Bevenmyr 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.
7. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
8. B. Demoen. A different look at garbage collection for the WAM. In P. Stuckey, editor, *Proceedings of ICLP2002 - International Conference on Logic Programming*, number 2401 in Lecture Notes in Computer Science, pages 179–193, Copenhagen, July 2002. ALP, Springer-Verlag.
9. B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, October 2002.
10. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.
11. A. P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–6, 1958.
12. E. Goto. Monocopy and associative Algorithms in extended Lisp. Technical Report TR 74-03, University of Tokyo, 1974.
13. U. Neumerkel. *Speicherbereinigung fur Prologsysteme*. PhD thesis, DIPLOMARBEIT, Institut fur Praktische Informatik der Technischen Universitat Wien, 1989.

14. P.-L. Nguyen and B. Demoen. Input sharing for findall/3. CW Reports CW553, Department of Computer Science, K.U.Leuven, June 2009.
15. R. O’Keefe. $O(1)$ reversible tree navigation without cycles. *Theory and Practice of Logic Programming*, vol. 1, no. 5, pp 617 - 630, 2001.
16. C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
17. D. Sahlin and M. Carlsson. Variable Shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.
18. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.
19. 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, 7 1991.
20. H. J. Touati and A. M. Despain. An empirical study of the warren abstract machine. In *SLP*, pages 114–124, 1987.
21. R. Vandeginste, K. Sagonas, and B. Demoen. Segment order preserving and generational garbage collection for Prolog. In S. Krishnamurthi and C. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Proceedings*, volume 2257 of *Lecture Notes in Computer Science*, pages 299–317. Springer, 2002.
22. M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, May 1997.
23. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

Appendix: the *finer* can absorb

We explore here shortly the finer notion of *can absorb*, i.e. the one that we absorb also in the case $h(a)$ discussed in Section 5. We only consider implementation issues.

The *can absorb* relation is a relation between two terms. The approximation we have used until now has the advantage that a term that is a definite candidate to absorb, can be determined independently of which term it could absorb. When also case $h(a)$ are included in *can absorb*, then one really needs to establish this relation. That means that basically, one needs to implement a doubly nested loop over the heap as in

```
for each p pointing into the heap
  for each q pointing into the heap
    if (*p can absorb *q) save that result
```

This leads to an algorithm that is quadratic in the size of the heap. Our first implementation did just that (but based on the approximate notion of *can absorb*) and was clearly suffering from the bad complexity to the point of being not usable.

Two more issues remain. For the finer *can absorb*

- one needs to be able to efficiently test in which heap segment a cell resides: this functionality is also present in our garbage collector and does in practice not pose a problem, because $O(\log(n))$ where n is the number of heap segments
- one needs to be able to efficiently test in which heap segment a cell was trailed: this requires a bit more preparation of the information in the trail before the build phase starts; this also seems rather straightforward

To avoid the quadratic traversal of the heap, one could as a reasonable approximation also use two extra pieces of term info item in the INFOTABLE: the age of the youngest trailed cell and the age of the oldest cell. When trying to find a term that can absorb a newly found term, these need to be taken into account. we have no evidence that such an approximation is in practice better than the coarser approximation we have used in our implementation.