TLP **13** (1): 71–106, 2013. © Cambridge University Press 2011
doi:10.1017/S1471068411000421 First published online 9 August 2011

Representation sharing for Prolog

PHUONG-LAN NGUYEN
Institut de Mathématiques Appliquées, UCO, Angers, France
(*e-mail:* nguyen@ima.uco.fr)

BART DEMOEN
Department of Computer Science, K.U. Leuven, Belgium
(*e-mail:* bart.demoen@cs.kuleuven.be)

submitted 8 June 2010; revised 19 January 2011; accepted 30 May 2011

Abstract

Representation sharing can reduce the memory footprint of a program by sharing one representation between duplicate terms. The most common implementation of representation sharing in functional programming systems is known as hash-consing. In the context of Prolog, representation sharing has been given little attention. Some current techniques that deal with representation sharing are reviewed. The new contributions are (1) an easy implementation of *input sharing* for *findall/3*; and (2) a description of a *sharer* module that introduces representation sharing at runtime. Their realization is shown in the context of the Warren Abstract Machine (WAM) as implemented by hProlog. Both can be adapted to any WAM-like Prolog implementation. The sharer works independently of the garbage collector, but it can be made to cooperate with the garbage collector. Benchmark results show that the sharer has a cost comparable to the heap garbage collector, that its effectiveness is highly application-dependent, and that its policy must be tuned to the collector.

KEYWORDS: Prolog, WAM, memory management

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 contexts, e.g., by the *intern* method for Strings in Java, by hash-consing in functional languages (Goto 1974), and by data de-duplication during backup. In programming language implementation, hash-consing is probably the best-known representation sharing technique: hash-consing was invented by Ershov (1958) and used by Goto (1974) 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. Appel and Gonçalves (1993) explores the idea of using 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: for the first time, a representation sharing technique is cooperating with the garbage collector. Our approach is most closely related to Appel and Gonçalves's (1993) approach, but also has some important differences.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

Representation sharing has been given little explicit attention in the context of Prolog implementation. However, the issue pops up from time to time. Here are some historical highlights:

- In 1989, in his Diplomarbeit, Neumerkel (1989) mentioned how by applying DFA-minimization to Prolog terms, certain programs can run in linear space (instead of quadratic); there was no implementation; in Section 8.1, his example program is used as a benchmark.
- In 1991, Sahlin and Carlsson (1991) ends with the following sentence: "It still remains to be seen, however, what we meant by 'folding identical structures'; the current paper offers a solution to this mysterious sentence.
- In a 1995 comp.lang.prolog post, Grimley-Evans (1995) asked for more sharing in *findall/3*, i.e., he wanted the solution list of a call to *findall/3* to share with the generator; *input sharing* (Mariën and Demoen 1993) does exactly that; Section 3 describes input sharing more precisely and how it can be implemented efficiently.
- In a 2001 Logic Programming Pearl O'Keefe (2001) mentioned a *findall/3* query that could benefit from representation sharing in the answers; as for the previous bullet, Section 3 contains the solution.
- Demoen (2002) gave a fresh view on garbage collection for Prolog; it detailed 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, Neumerkel (1989) 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 used the term *factoring*; we prefer *representation sharing*; the current paper is the result of exploring its implementation issues.

The paper is organized as follows: Section 2 starts with describing what we mean by representation sharing. Section 2.1 lists a number of more or less popular forms of representation sharing in Prolog. Section 3 describes how to retain *input sharing* for *findall/3* and evaluates our implementation on a number of benchmarks.

Section 4 sets the scene for the focus of the rest of the paper: general sharing for Prolog. Section 5 forms the intuition on such sharing, while Section 6 introduces the notion of *absorption*: It shows when individual cells can share their representation and the approximation that works for us. It then lifts representation sharing from individual cells to compound terms and discusses some properties of our notion of representation sharing. Section 7 explains our implementation of representation sharing based on the earlier decisions. Section 8 discusses the benchmarks and the experimental results. Section 9 shows extensions of the basic implementation, variations, and related issues. Section 10 discusses related work, and we conclude in Section 11.

We have used hProlog 3.1.* as the Prolog engine to experiment with, but it is clear that everything can be ported to other Warren Abstract Machine (WAM)-like systems as well: we make that more explicit later on. hProlog is a descendant of dProlog as described by Demoen and Nguyen (2000). SICStus Prolog 4.1.1 serves as a yardstick to show that the hProlog time and space figures are close to a reliable state of the art system. All benchmarks were run on an Intel Core2 Duo Processor T8100 2.10 GHz.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

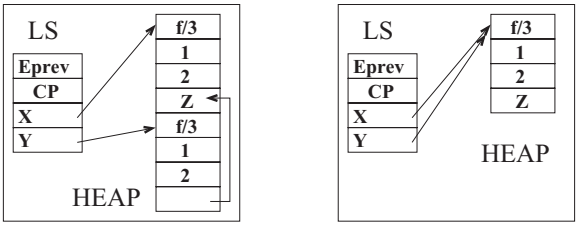


Fig. 1. Representation sharing.

We assume the reader to be familiar with the WAM (Warren 1983; Aït-Kaci 1991) and Prolog (Clocksin and Mellish 1984). We use the term *heap* when others use *global stack*, i.e., the place where compound terms are allocated. We use *local stack* and *environment stack* interchangeably and denote it by LS in pictures.

2 Representation sharing versus hash-consing

Consider the predicates main1 and main2 defined as

```
main1 :-                          main2 :-
       X = f(1,2,Z),                     X = f(1,2,Z),
       Y = f(1,2,Z),                     X = Y,
       use(X,Y).                         use(X,Y).
```

In a naive[1] implementation, the execution of ?- main1. just before the call to *use/2* results in a memory situation as in the left of Figure 1. In this figure, the heap cell with Z is a self-reference in the WAM. Clearly, the terms X and Y are exactly the same even after they have been created, and therefore they can share the same representation: that sharing can be seen in the right of Figure 1 and in the code for the predicate main2.

Hash-consing is usually associated with the technique that keeps a hash table of terms and during term creation checks whether a term is new or already exists in the hash table.

An implementation with hash-consing usually changes the representation of terms, and consequently the code that deals explicitly with this representation. For Prolog, the affected code would be general unification and built-in predicates. That is too intrusive for our aims: We intend our implementation of representation sharing to be easy to integrate in other Prolog systems and there should be no global impact. Therefore, we keep the usual (WAM) term representation and do not touch any part of the implementation, except for the *sharer* module that introduces representation sharing. Given the complexity of current Prolog systems, this seems to us the only way to make representation sharing acceptable by implementors.

2.1 Some forms of representation sharing for prolog

Prolog implementations already provide some specific representation sharing. Here are few examples:

[1] In other words, an implementation without compile time common subexpression elimination; however, note the danger of such optimization in the presence of destructive assignment: see Section 9.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

- In older implementations, the predicate *copy_term/2* copies ground terms; in newer implementations – starting probably with SICStus Prolog (Carlsson 1990) – *copy_term/2* avoids copying ground (sub)terms; this means that the second argument can have some representation sharing with the first argument; however, note that *mutable* ground terms must be copied by *copy_term/2*, because otherwise sharing would become observable at the program level; we discuss this issue further in Section 9.
- Some programs contain ground terms at the source level; a typical example is the second argument of a goal like *member(Assoc,[fx,fy,xfx,xfy,yfx,xf,yf])*; ECLiPSe Wallace *et al.* 1997) pre-allocates such ground terms and makes sure that any time such a fact or goal is called, the ground term is reused; Mercury performs this compile-time optimization as well.
- When two terms are unified, they can share a common representation in the forward execution; at various stages in its life, BinProlog (Tarau 1991) enforced such sharing by (in WAM speak) redirecting the S-tagged pointer of one of the two terms and (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; a similar technique was already used for strings only in the Logix implementation of Flat Concurrent Prolog (Hirsch *et al.* 1987)

In each of the above cases, the implementor of the Prolog system decided for more representation sharing than would be the case in a more straightforward implementation. Application programers and library developers usually take care as well to let their runtime data structures share common parts.

In the above, *copy_term/2* and unification are built-in predicates that have a chance to increase representation sharing. In Section 3, *findall/3* is added to this short list.

3 Input sharing for *findall/3*

Marien and Demoen (1993) introduced the notion of *input sharing* in the context of *findall/3*. Input sharing consists of a solution in the output from *findall/3* (its third argument) sharing with the input to *findall/3* (its second argument).

Later, in the Logic Programming Pearl (O'Keefe 2001) it is suggested that *findall/3* could repeatedly avoid copying the same terms over and over again: this would improve the space complexity of some queries that use *findall/3*, from $O(n^2)$ to $O(n)$. However, O'Keefe (2001) suggests that hash-consing should be used, with the consequence that the time complexity remains the same: our implementation of *input sharing* – which is exactly what is needed here – improves both time and space complexity. The example used by O'Keefe is rather complicated, so for now we use as an illustration a piece of simple Prolog code that was posted by Grimley-Evans (1995); we changed the names of the predicates and variables,

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

```
findall_tails(L,Tails) :- findall(Tail,is_tail(L,Tail),Tails).

is_tail(L,L).
is_tail([_|R],L) :- is_tail(R,L).

all_tails([],[[]]).
all_tails(L,[L|S]) :- L = [_|R], all_tails(R,S).
```

Clearly, goals of the form ?- findall_tails(L,Tails). and ?- all_tails(L,Tails). with a ground argument L succeed with the same answer Tails. For instance,

```
?- findall_tails([1,2,3],Tails).
Tails = [[1,2,3],[2,3],[3],[]]
```

The usual implementation of *findall/3* copies over and over again parts of the input list L, and this results in quadratic behavior (in the length of L) for *findall_tails/2*, while *all_tails/2* is linear, both in space and time! Clearly, with enough *input sharing*, the *findall_tails/2* query could be linear.

In the following sections we show how a traditional *findall/3* implementation in the context of the WAM can be easily adapted to cater for input sharing. An alternative copy-once implementation of *findall/3* is also shown.

Before going into the details, it is worth pointing out the limitations of input sharing. Clearly, if L is a list with non-ground elements, the two queries

```
?- findall_tails(L,Tails).          ?- all_tails(L,Tails).
```

yield different answers. The first query makes fresh variants of the variables in each of the solutions in Tails, while the second query does not. As an example:

```
?- findall_tails([X,Y,Z],L),          ?- all_tails([X,Y,Z],L),
   numbervars(L,0,_).                    numbervars(L,0,_).

                                      X = A   Y = B   Z = C
L = [[A,B,C],[D,E],[F],[]]            L = [[A,B,C],[B,C],[C],[]]
```

This means, we can use an input sharing version of *findall/3* when the arguments of the generator are either ground or free: the danger is only in terms containing variables.

3.1 The implementation of **findall/3**

The hProlog implementation of *findall/3* follows the same pattern as in many systems:

```
findall(Template,Generator,SolList) :-
        findall_init(Handle),
        (
          call(Generator),
          findall_add(Template,Handle),
          fail
        ;
          findall_get_solutions(SolList,Handle)
        ).
```

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

For simplicity, we have left out all error checking and error recovery codes. The predicate *findall_init/1* returns a handle so that the particular invocation of *findall/3* is identified: This is used for correct treatment of nested calls to *findall/3*. *findall_add/2* uses that handle, and copies the Template to a temporary zone. *findall_get_solutions/2* uses the handle as well: It retrieves the complete list of solutions from the temporary zone and unifies it with the third argument to *findall/3*.

The next section describes how to turn this code into code that shares the input.

3.2 *The basic idea of input sharing for* **findall/3**

The predicate *findall_add/2* in our implementation of *findall/3* is just a version of *copy_term/2*: at the implementation level, they both use the same C function for actual copying. The same is true for *findall_get_solutions/2*.

The first idea might be to use an implementation of *copy_term/2* that avoids copying ground terms. However, in the context of *findall/3*, groundness is not enough: the ground term must also be *old enough*, so that backtracking (over the Generator) cannot alter it. In order to be more precise, anything ground that survives backtracking over the Generator need not be copied by *findall_add/2*. Or to put in another way, anything ground before the call to findall (Template,Generator,SolList) need not be copied by *findall_add/2*.

When the arguments of the generator are ground or free, such terms can be recognized easily: their *root* resides in a heap segment that is not younger than the call to *findall/3*.

So we need to be able to identify the older heap part relevant to a particular call to *findall/3*. That is quite easy in the WAM: we just remember the relevant heap pointer!

3.3 **findall/3** *with input sharing: the implementation*

We use two new low-level built-in predicates:

- **current_heap_top(–)**: Unifies the argument with (an abstraction of) the current value of the heap pointer H.
- **set_copy_heap_barrier(+)**: Sets a global (C-)implementation variable (named *copy_heapbarrier*) to the heap pointer value corresponding to its argument.

The following code shows how the new built-ins are used:

```
sharing_findall(Template,Generator,SolList) :-
        current_heap_top(Barrier),
        findall_init(Handle),
        (
          call(Generator),
          set_copy_heap_barrier(Barrier),
          findall_add(Template,Handle),
          fail
        ;
          set_copy_heap_barrier(Barrier),
          findall_get_solutions(SolList,Handle)
        ).
```

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

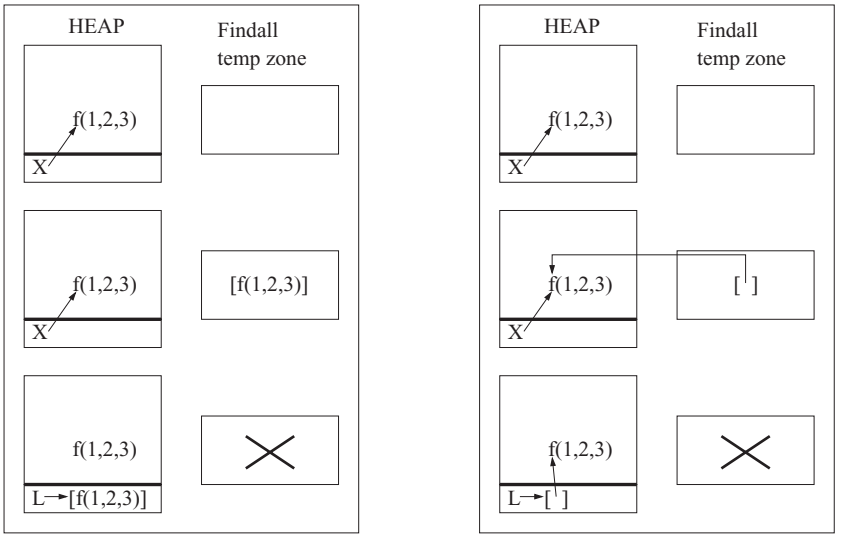


Fig. 2. Findall without and with input sharing.

An additional small change needs to be made to the implementation of *findall_add/2* (and *findall_get_solutions/2*) as well: when a term is about to be copied and it is older than *copy_heapbarrier*, only the root pointer to this term is copied. It amounts to adding a statement like

```
if (struct_addr < copy_heapbarrier) { *whereto = struct_addr; continue; }
```

at a few places in the C code of *copy_term/2*: this piece of code just copies the top pointer of the structured term instead of copying it recursively. The C variable *struct_addr* holds the address of the structure about to be copied.

For explanatory reasons, we have shown the implementation of *sharing_findall/3* as a variant of the basic implementation of *findall/3* using two new built-ins. However, one can also fold the functionality of these new built-ins into adapted versions of *findall_[init/1, add/2, get_solutions/2]*: the top-of-heap at the moment of calling *sharing_findall/3* is then stored in the data structures belonging to that particular call. This top-of-heap at the moment of calling *sharing_findall/3* must also be appropriately treated by the garbage collector.

3.4 An example

The heap and temporary findall zone are shown in Figure 2 for the very simple query

```
?- findall(X, X=f(1,2,3), L).
```

The left part of the picture shows three snapshots during the execution of the query without input sharing. The right part shows the corresponding snapshots with input sharing.

The snapshots are taken

- just before *findall_add/2* is executed: the temporary zone is still empty;

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

- just after *findall_add/2* is executed; at the left, the temporary zone contains a copy of the term f(1,2,3); at the right, there is a pointer to the term on the heap;
- just after *findall_get_solutions/2* is executed, the temporary zone can be discarded; at the left, the solution list contains a copy of f(1,2,3); at the right, there is just a pointer to the old term on the heap.

The space savings are clear.

3.5 *Copy-once* findall/3

The usual implementation of *findall/3* copies the solutions twice. BinProlog was probably the first implementation copying the solution only once, by means of a technique named *heap lifting* or more popularly a *bubble in the heap* (Tarau 1992). Currently, the BinProlog implementation (Tarau and Majumdar 2009) relies on *engines* for *findall/3*. Mercury also uses a copy-once findall (named *solutions/2*): As Mercury relies on the Boehm-collector (Boehm and Weiser 1988), there is no memory management hassle with a bubble in the heap.

It is rather easy to implement a copy-once *findall/3* in any Prolog system that has non-backtrackable destructive assignment (with *nb_setarg/3*) as in hProlog or SWI-Prolog (Wielemaker *et al.* 2008)[2]:

```
copy_once_findall(Template,Generator,SolList) :-
       Term = container([]),
       (
         call(Generator),
         Term = container(PartialSolList),
         copy_term(Template,Y),
         nb_setarg(1,Term,[Y|PartialSolList]),
         fail
       ;
         Term = container(FinalSolList),
         reverse(FinalSolList,SolList)
       ).
```

As before, this code can be enhanced with the newly introduced built-ins to yield a copy_once_sharing_findall. If copying the solutions dominates the execution, the copy-once *findall/3* is about twice as fast as the regular *findall/3*. However, its main drawback is that it consumes (for the benchmarks below) about three times as much heap space. The reason is that *nb_setarg/3* must freeze the heap if its third argument is a compound term. The heap-lifting technique (which we have not implemented in hProlog) does not have this drawback.

3.6 *Experimental evaluation*

Input sharing improves (sometimes) the complexity (space and time), and the constant overhead is really very small, as can be judged from the changes needed to implement

[2] Note that SWI-Prolog uses *nb_linkarg/3* as the name for hProlog's *nb_setarg/3*.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

it. One could therefore argue that benchmarks are not needed. Even so, we present two benchmarks: one is the *findall_tails/2* example (see Section 3.7). We start with a *findall/3* related query from O'Keefe (2001): This pearl is about tree construction and traversal. It contains the following text:

> *Query q1*[3] *requires at least* $O(n^2)$ *space to hold the result. If findall/3 copied terms using some kind of hash consing, the space cost could be reduced to* $O(n)$*, but not the time cost, because it would still be necessary to test whether a newly generated solution could share structure with the existing ones.*

Note that the n above is the number of nodes in the tree, not the tree depth: the number of nodes is roughly 4^{depth} where *depth* is the depth of the tree.

We needed to make a slight change to the program from O'Keefe (2001): in its original form it contains a *mk_tree/2* predicate defined as

```
mk_tree(D, node(D,C)) :-
        (   D > 0 ->
            D1 is D - 1,
            C = [T,T,T,T],
            mk_tree(D1, T)
        ;   C = []
        ).
```

Because of the conjunction *C = [T,T,T,T], mk_tree(D1, T)*, the heap representation of the constructed tree is linear in the first argument D, even though it has an exponential number of nodes: indeed, the constructed tree has a lot of internal sharing. Such internal sharing is retained by most reasonable implementations of *copy_term/2* and by *findall_add/2*.[4]

In order to test what O'Keefe (2001) really meant, we have changed the particular conjunction to

```
C = [T1,T2,T3,T4],
mk_tree(D1, T1), mk_tree(D1, T2),
mk_tree(D1, T3), mk_tree(D1, T4)
```

so that the size of the representation of the tree is linear in the number of nodes in the tree (and exponential in D). This code rewrite achieves the desired effect because Prolog systems typically don't perform the analysis needed to notice that T1, T2, T3, and T4 are declaratively the same value, and neither is this detected at runtime. See Appendix for all codes necessary to run the benchmark.

3.6.1 The modified tree benchmark: results

Table 1 shows timings (when considered meaningful) and space consumption for queries *?- f1(Depth)* with different values of Depth. Time is reported in milliseconds, and space in bytes. We have chosen SICStus Prolog for comparison with another system because the SICStus Prolog implementation performed better and more reliably than the other

[3] f1(N) in the Appendix.
[4] A notable exception is Yap.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

Table 1. *Heap consumption (bytes) and time (msec) for the tree benchmark.*

	hProlog		hProlog input sharing		SICStus Prolog	
Depth	Time	Space	Time	Space	Time	Space
1		964		368		980
2		11,332		1,840		10,964
3		158,020		9,776		155,348
4		2,394,436		49,712		2,379,476
5	156	37,599,556		242,224	250	37,523,156
6	2616	598,030,660		1,143,344	6820	597,659,348
7				5272,112		
8			156	23,884,336		
9			652	106,721,840		
10			2,804	471,626,288		

systems that we have tried. Moreover, the trend of the measurements with other systems was basically the same.

The timings without sharing do not show anything interesting complexity-wise: neither of the implementations without input sharing can deal with more than about 5,000 nodes. The input sharing implementation, on the other hand, can go easily up to one million nodes. The heap consumption columns give a good picture of how the heap size grows: the non-input-sharing implementations show a quadratic dependency of heap consumption on the number of nodes. Only *hProlog input sharing* shows a linear dependency.

Table 1 clearly shows that our simple implementation to enforce input sharing is very effective and performs actually better than hoped for by O'Keefe (2001). Indeed, we achieve linear space *and* time complexity for the f1(Depth) query. Hash-consing would not be able to do that.

3.7 The tails benchmark

Table 2 shows the space consumption for the tails benchmark. The timings are meaninglessly small for the variants with sharing, and therefore only shown for the regular findall columns. The *Length/1000* column indicates the length of the ground input list L to queries of the form *?- all_tails(L,Tails)* and *?- [sharing_]findall(Tail,is_tail(L,Tail),Tails)*.

findall/3 with input sharing clearly beats the *findall/3* without input sharing. SICStus Prolog can do larger sizes with the ordinary *findall/3* implementation than hProlog: the latter runs out of memory earlier because of its different memory allocation and heap garbage collection policy.

We have tried to measure the overhead of our method, but it is too small to show up meaningfully in any of our experiments.

3.8 Conclusion on input sharing for **findall/3**

Already in Grimley-Evans (1995) there was a demand for sharing between the input to *findall/3* and its output. Also, O'Keefe (2001) points out that this would be beneficial

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

Table 2. *Heap consumption (KiB) and time (msec) for tails benchmarks.*

	hProlog				SICStus Prolog		
Length	Regular findall		Findall with input sharing	all_tails	Regular findall		all_tails
/1000	Time	Space	Space	Space	Time	Space	Space
1	112	26,034	8	8	120	26,034	8
2	444	104,068	16	16	490	104,068	16
3	1,028	234,102	24	24	1,160	234,102	24
4	1,920	416,136	32	32	2,130	416,136	32
5			40	40	3,650	650,170	40
6			48	48	5,080	936,204	48
7			56	56			56
8			64	64			64
9			72	72			72
10			80	80			80
100			800	800			800
1,000			8,000	8,000			8,000

to some programs. Optimal input sharing would attempt to share all (sub)terms that are ground just before the call to *findall/3*. Checking this at runtime can be involved and costly. Our implementation approximates that by just checking that the root of a term is old enough, and relying on the programer (or some other means) to use *sharing_findall/3* only when this simple check implies that the whole term was ground at the moment of the call to *sharing_findall/3*. This is, in particular, true in the common case that the generator of *findall/3* (its second argument) is a goal of which every argument is ground or free: that was the case for our benchmark *findall_tails/2*. That condition on the generator can be easily checked before calling *sharing_findall/3* and could also be derived by program analysis.

Our approach does not implement *solution sharing*: hash-consing, or maybe even better tries, could do the job. Sections 4 and later provide a more general and lightweight solution to representation sharing.

In O'Keefe (2001), one can also read the following:

> *One referee suggested that Mercury's "solutions/2" would be cleverer. A test in the 0.10 release showed that it is not yet clever enough.*

As Mercury (Somogyi *et al.* 1996) relies on the Boehm-allocator and -collector for its memory management, it is quite difficult to devise a simple dynamic test whether a (ground) term is old enough: On the whole, a Mercury implementation does indeed not benefit from keeping the address order of terms consistent with their age. On the other hand, in the WAM, such a test comes naturally with the needs of a strict heap allocation discipline and conditional trailing.

As a conclusion, we think we have succeeded in providing input sharing for *findall/3* with minimal change to the underlying Prolog execution engine: any Prolog implementation with a heap allocation strategy similar to the WAM can incorporate it easily. How to

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

present the functionality in a safe way to the user is a language design issue and as such beyond the scope of this paper.

4 General representation sharing for prolog

Appel and Gonçalves (1993) adapts a copying collector to perform hash-consing for the data in the older generation. Since we would like our implementation of representation sharer to be a model for other Prolog implementations, we cannot just copy that idea. Indeed, hash-consing requires a serious adaptation of the term representation, and, moreover, Prolog systems typically have sliding collectors, the exceptions being hProlog and BinProlog. Therefore, we want to investigate representation sharing in a way that does not require a change in term representation, and that is independent of the details of the garbage collector: This will make it easier for Prolog systems to implement their own sharing module based on our experience. Appel and Gonçalves (1993) 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 basically agree with Appel and Gonçalves (1993): it is better to avoid putting any effort in sharing with dead terms.

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 clearer, i.e., we make no assumptions on the workings of the garbage collector.

Baker (1992) shows that the combination of tabling and hash-consing is particularly powerful, as duplicate terms do not occur, equality of terms can be decided by a single pointer comparison instead of traversing the whole terms. However, in that context and in its original form, hash-consing guarantees representation sharing all the time, although that is not our aim. Unfortunately, Baker (1992) does not show experimental data for hash-consing without tabling.

5 Representation sharing in prolog: examples

Two issues make representation sharing in Prolog-like languages different from other languages: the logical variable and backtracking. Subsequent subsections show by example how these affect the possibilities for representation sharing.

5.1 Sharing within the same segment

The first example in Section 2 shows the simplest 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 identical ground terms in the same segment cannot always share their representation:

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

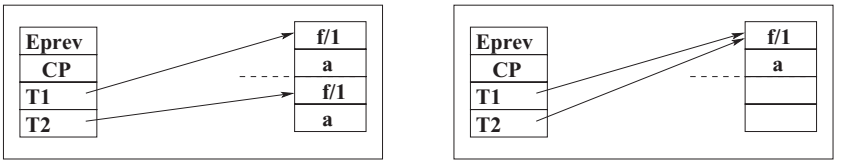


Fig. 3. Representation sharing of two terms in different segments.

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

While executing the query ?- main3, just before the execution of *share*, the terms T1 and T2 are identical, ground, and they are completely within the same segment. However, it would be wrong to make them share their representation, as in the failure continuation, they are no longer identical. Loosely speaking, the occurrence of trailed variables in a term makes the term unsuitable for representation sharing.

5.2 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. As we do not want to mix this issue with trailed heap locations, the example works with ground terms.

```
main4 :-
       T1 = f(a),
       (
        T2 = f(a), share, use(T1,T2)
       ;
        use(T1)
       ).
```

During the execution of the query ?- main4, T1 and T2 live in two different segments. T1 lives in the oldest segment, as seen in the left of Figure 3.[5] Since T1 is used after backtracking, the natural thing is to keep the representation of the oldest term, because it potentially lives longest. So the introduced sharing representation is as in the right of Figure 3. Alternatively, one could use as 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. We consider this a bad alternative, but a slight variation on the same example shows that the choice is not so clear cut:

[5] The dashed line indicates the heap segment barrier.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

```
main5 :-
       T1 = f(a),
       use(T1),
       (
        T2 = f(a), share, gc, use(T2)
       ;
        dontuseT1
       ).
```

```
main6 :-
       T1 = f(a),
       use(T1),
       (
        T2 = f(a), gc, use(T2)
       ;
        dontuseT1
       ).
```

The codes of main5 and main6 differ only in the call to *share* in main5.

- *With sharing in main5*: 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.
- *Without sharing in main6*: At the point gc kicks in, T1 is unreachable and its representation disappears; this means that after backtracking to dontuseT1, the heap is empty.

This example shows that representation sharing between terms in different segments can lead to a higher heap consumption, or more invocations of the garbage collection.

Finally, it is clear that mutable terms should not share their representation: It is in general impossible to know whether two mutable terms will be identical for the rest of their common lifetime. We deal with mutable terms in more detail in Section 9.

6 Sharable terms and absorption

The examples in the previous section give some intuition on what we mean by representation sharing, and also about its pitfalls. The examples have also indicated that we are working toward an implementation of a sharer that introduces sharing between two terms T1 and T2 by keeping the representation of one of them, say T1, and making T2 point to it. We coin this process *T1 absorbs T2*. This naturally leads to considering the notion *T1 can absorb T2*.

The most general definition of *T1 can absorb T2* would be that the sequence of solutions to the running program does not change by letting T1 absorb T2. That condition is, of course, not decidable, so we need a workable approximation to it.

The next sections explore the notion *can absorb* further, first by focussing on representation sharing for individual heap cells and then by considering compound terms.

6.1 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, say c1 and c2, have different contents (and are live), neither of them can absorb the other. Moreover, when the two cells have identical addresses, they have already absorbed each other. 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.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

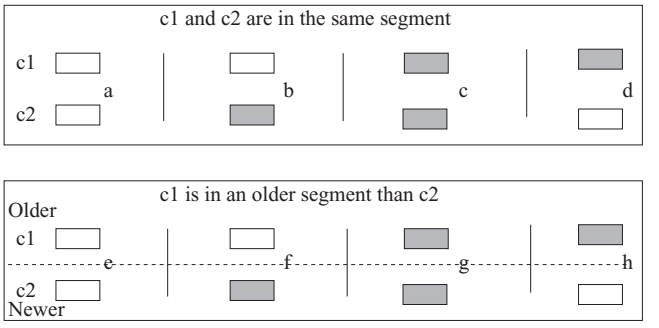


Fig. 4. Eight combinations of two cells.

Without loss of generality, we assume that c1 is older than c2.

This results in eight combinations shown in Figure 4(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 lines now indicate one or more heap segment separations. The vertical lines just separate different cases.

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 backward computations.

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.

fg: These cases are similar to cases bf bc above: as soon as one of the trailed cells is untrailed by backtracking, the contents of c1 and c2 differ; therefore representation sharing is not allowed; neither can absorb the other.

h: Now there are two possibilities:

(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, i.e., if c2 dies not later than c1 is untrailed, c1 can absorb c2.

(b) Otherwise, representation sharing is disallowed; neither cell can absorb the other.

Anticipating an implementation, we notice that it is important to be able to check quickly whether a cell is trailed. One bit – appropriately placed – is enough for that: that bit could be in the heap cells themselves, or it could be allocated in an array parallel to the heap. This would make cases **a** and **e** easy to identify.

In order to detect case **h(a)**, however, we also need to quickly retrieve from a heap address, the heap segment number in which it was trailed. That requires more setup, and it would slow down the sharer. We think the expected gain in space is too small to make this

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

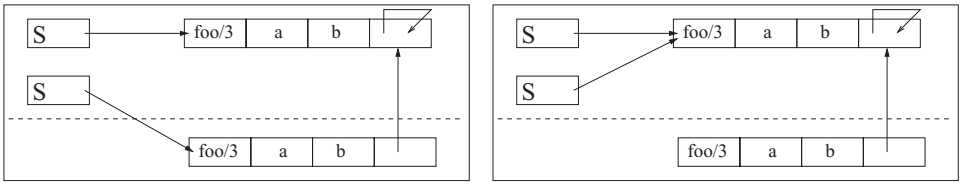


Fig. 5. (Left) the bodies fulfill the conditions for representation sharing. (Right) Sharing has been performed.

worthwhile. Instead, we went for disallowing sharing in case **h(a)**, so that our notion of *can absorb* becomes quite simple and leads to a simple decision procedure. In the following piece of code, pc1 and pc2 are pointers to heap cells c1 and c2:

```
boolean can_absorb(cell *pc1, cell *pc2)
{
  if (*pc1 != *pc2) return(FALSE);
  if (trailed(pc1)) return(FALSE);
  if (trailed(pc2)) return(FALSE);
  return(pc1 < pc2);
}
```

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.

As trailing prevents a cell from being able to absorb, or being absorbed, 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.

6.2 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 5.

Clearly, a necessary condition for such representation sharing is that the two bodies have the same contents. Moreover, as the term body always belongs to a single segment, the condition worked out for absorption for two individual cells that must hold for each pair of corresponding body elements. We arrive at the following:

Definition: Term T1 can absorb term T2 if T1 is older than T2, T1 == T2 and neither T1 nor T2 contain trailed cells.

Figure 5 shows two bodies that fulfill the conditions.

Note that the example exhibits a situation that we have not yet described: a variable chain. De-referencing must be stopped when a trailed cell is found.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

Note the similarity of the above analysis with the one for variable shunting in Sahlin and Carlsson (1991).

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.

6.3 Properties of our notion 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 crucially depends on those properties.

It is clear that *can absorb* is not symmetric: a newer term cannot absorb an older term in a different segment. Neither is *can absorb* anti-symmetric: case **a** in Section 6.1 shows that.

We denote by *absorbed(x,y)* the result of letting term *x* absorb term *y*; of course, under the condition that *x* can absorb *y*.

An important part of our definition of *can absorb* is that the terms do not contain trailed cells: it implies that a candidate term for absorbing or being absorbed can be recognized without knowing the other term, i.e., one checks whether it contains trailed cells or not and by keeping information about visited terms, one can assure that this information about the terms can be gathered in time proportional to the heap.

From the definition, it also follows that *can absorb* is transitive:

```
(x can absorb y) && (y can absorb z) ==> x can absorb z
```

Finally, the absorption process is also associative, i.e.,

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

(of course under the condition that *x* can absorb *y* and *y* can absorb *z*). This means that the order in which absorption takes place is immaterial: the end result is the same.

Together, these properties allow for a basically linear sharing algorithm, on condition that term hashing is perfect. With a less than perfect hash function, the algorithm might need to traverse some terms more than once.

7 Implementation of representation sharing

We have taken hProlog as the platform for an implementation of representation sharing. It is based on the WAM (Warren 1983; Aït-Kaci 1991) with a few differences:

- The choicepoint stack and environment stack are not interleaved as in the WAM, but separate stacks as in SICStus Prolog.
- Free variables only reside on the heap; i.e., there are no self-references in the environment stack, just as in Aquarius Prolog (Van Roy and Despain 1992)
- hProlog supports some more native types like char, string, and bigint; it also has attributed variables.

hProlog employs a mark-and-copy type of garbage collector, with its roots as in Bevemyr and Lindgren (1994) and preserves segment order as described in Vandeginste *et al.* (2002).

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

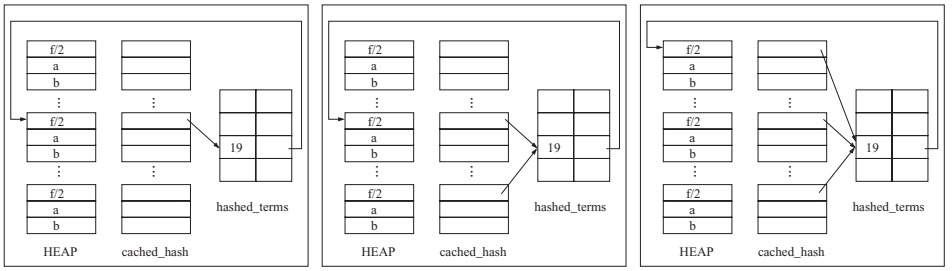


Fig. 6. Three identical terms are treated during the build phase. (a) After treating middle f(a,b); (b) after treating younger f(a,b); and (c) after treating older f(a,b).

Most other systems use a sliding collector based on Appleby *et al.* (1988). hProlog does not implement variable shunting.

hProlog is a direct descendant of dProlog (Demoen and Nguyen 2000). Its purpose is to offer a platform for experiments in the WAM-like Prolog implementation. Its high performance gives the experiments an extra dimension of credibility.

The implementation uses two data structures: these can be seen in Figure 6. We name them *cached_hash* table and *hashed_terms* table. Together they form the sharer tables.

- cached_hash: This is an array of 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 7.3 for more on this.
  - a pointer to the hashed_terms table: The corresponding heap cell has been treated, and its sharing information can be found by following the pointer
- hashed_terms: This data structure contains records with two fields: *hashvalue* and *term*; suppose a pointer in the cached_hash points to a record in the hashed_terms, and the corresponding heap cell A is the entry point of term TermA, then
  - the *hashvalue* field in the record is the hash value of term TermA.
  - the *term* field in the record is a pointer to a heap cell B that is the entry point of term TermB that can absorb TermA (provided A and B are the not same cell); our implementation makes sure that the heap cell B is as old as possible, i.e., B is equal to A or older than A.

Treating a heap cell consists of filling out the corresponding cell in the cached_hash table and possibly the hashed_terms table.

The implementation of the hashed_terms table is actually as that of a hash table: the hashvalue of a term modulo the size of the hash table is used for determining the place in the hashed_terms, and a linked list of buckets is used to resolve collisions. Many other implementations of this hashed_terms table would also be fine.

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

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

The main algorithm consists of two phases:

- **build**: It builds the cached_hash and hashed_terms tables; during this phase nothing is changed to the heap; this phase *treats* all heap cells.
- **absorb**: It performs all absorption possible by using the cached_hash and hashed_terms tables.

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

7.1 Phase I: building the cached_hash and hashed_terms tables

The build phase performs the action *compute_hash* for each cell in the heap: the corresponding cell in the cached_hash is set to either *impossible* or to a pointer to the hashed_terms. The function *compute_hash* is always called with a tagged term as argument. In the code below, we use *STRUCT* as the tag of a pointer pointing to the functor cell of a compound term. In figures, this tag simply shows as *S*. The function call *tag(p,STRUCT)* returns such a STRUCT tagged pointer; the function *untag* has the opposite effect. A function call like *tag(term)* returns the tag to its argument.

Note that the following code ignores certain issues like checking whether a cell is trailed, and LISTS. We deal with them in Section 7.3.

```
foreach p in [beginheap, endheap] && is_functor(*p)
     compute_hash(tag(p,STRUCT));  // ignore return value

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

    case STRUCT:
         p = untag(p,STRUCT);
         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);
         return(hashvalue);
  }
}
```

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 cached_hash table points to the hashed_terms table. *already_computed_hash(p)* returns

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

the previously computed hash value (for the term starting at p) from the hashed_terms entry corresponding to p: in this way, re-computation (and re-traversal of the same term) is avoided.

In the *save_hash* function that follows, we have left out collision handling: for the sake of presentation, we assume perfect hashing.

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

  cached_hash[p-beginheap] = hashed_terms + index;

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

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

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

Figure 6 shows how three equal terms are treated by compute_hash and the effect thereof on the cached_hash and hashed_terms tables.

7.2 Phase II: absorbing

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

```
foreach cell c in the heap
               in the local stack
               in the choicepoint stack
               in the argument registers do
      let p be the contents of c;
      if (tag(p) == STRUCT)
         {
           q = untag(p,STRUCT);
           if (cached_hash[q-beginheap] points to hashed_terms)
                  replace c by tag(cached_hash[q-beginheap]->term,STRUCT);
         }
```

Figure 7 shows how the older term absorbs the two identical younger terms.

7.3 Comments on the code

The code in Section 7.1 ignores certain issues:

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

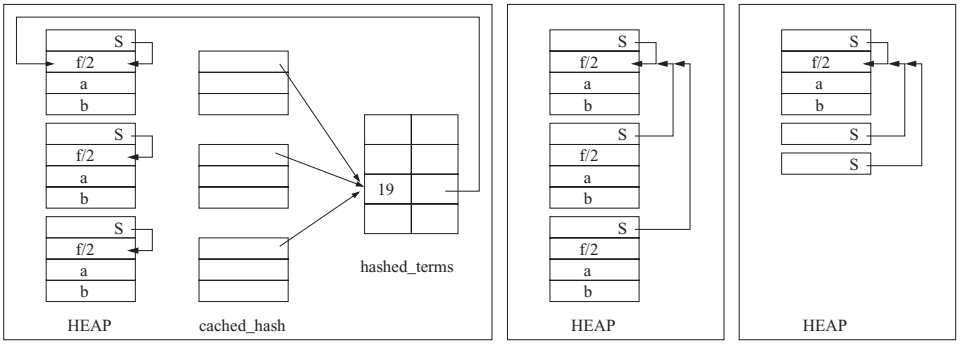


Fig. 7. Absorption and GC in action. (a) Just before absorbing, (b) after absorbing, and (c) after one more GC.

- *Checking whether a heap cell is trailed*: During the initialization of the build phase, the cached_hash table entries corresponding to trailed heap entries are initialized to *impossible*; this requires traversing the trail once and it makes checking whether a cell is trailed for constant time; the checks whether a heap cell is trailed are required during the dereferencing loop; when a trailed cell is encountered, the computation of the hash value is stopped and the corresponding cached_hash table entries of the term containing the trailed cell are also set to *impossible.*
- *Other datatypes*: The code takes into account only non-listed structured terms, atoms and variables; it is easy to extend it to other types that occupy a single cell; for other *atomic* types (real, string, bigint) we have followed the same principle as for unlisted 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 7.4).
- *foreach*: Our implementation uses a linear scan for the *foreach* constructs: this is possible for all the stacks in hProlog; if this is not the case, then one can traverse the live data starting from the root set as the garbage collector does (e.g., during its marking phase).

The code implementing the above is less than 700 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 hashed_terms table: the cached_hash table has exactly the same size as the collector needs for performing its collector duties.

7.4 Representation sharing of lists

In the WAM, lists have no header like other compound terms. A list is represented by an L-tagged pointer 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 hashed_terms pointer in the cell corresponding to the list-pointer. Figure 8 shows an example with just lists.

Note that functor cells can only appear on the heap, while list pointers can occur also in environments, choicepoints, and the argument registers. As a result, with just a

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

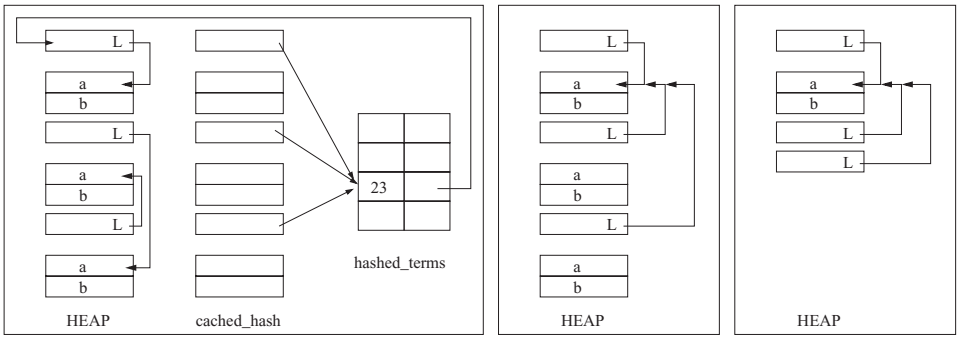


Fig. 8. Absorption and GC in action. (a) Just before absorbing, (b) after absorbing, and (c) after one more GC.

hashed_terms pointer array parallel to the heap, some representation sharing in the other stacks can get lost for lists. A similar hashed_terms pointer array parallel to other stacks can solve this problem: our implementation does not do that. Another solution consists in using the cell of the first element of a list for keeping the corresponding hashed_terms information. We have not explored that alternative.

7.5 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 most easily when the GC starts with a marking phase: the build phase of the sharer can indeed be integrated in the marking phase of the collector. The absorb phase can be run before the next GC phase, or be integrated with it. That would lead to a (mark + build) & (copy + absorb) collector for hProlog. 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 the beneficial effect of the sharer can be seen only after the next GC. 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 switch from one policy to another could be based on that.

8 The benchmarks and the results

As Appel and Gonçalves (1993) is the closest to our representation sharing, we are inclined to use the same benchmarks. However, Appel and Gonçalves (1993) show overall very

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

little impact of *hash-consing* and unfortunately the benchmarks were not analyzed so as to explain why hash-consing is not effective on them. On the other hand, one cannot *a priori* assume that our sharer will show the same behavior because of the differences between our respective implementations, and even the language:

- hProlog only performs major collections, while SML/NJ has a generational collector (with two generations).
- Our sharer does not alter the representation of terms, while Appel and Gonçalves (1993) perform hash-consing (which entails a representation change) on the old generation only.
- SML/NJ is a deterministic language and a boolean SML/NJ function is like a semi-det predicate in Prolog; however, in a typical Prolog implementation, the data it creates is (on failure) backtracked over in Prolog and the WAM recovers its space: this can have a huge impact on some benchmarks (the mandelbrot benchmark is an example).
- In Appel and Gonçalves (1993) hash-consing was inseparably tied to the (generational) collector; in contrast, we have explicitly aimed at keeping the collector and the sharer separated (we have stated its reason in Section 4); this has an impact on the efficiency of the sharing process.

So, it seems worthwhile to redo some of the benchmarks of Appel and Gonçalves (1993). The following section describes those benchmarks as well as some others not appearing in Appel and Gonçalves (1993).

8.1 The benchmarks

8.1.1 Boyer

Boyer is a famous benchmark initially proposed in (Gabriel 1985) 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, Baker (1992). Anyway, in Appel and Gonçalves (1993), this benchmark shows the best results for hash-consing. The inherent reason is that terms are rewritten to a canonical form and thus many initially different terms end up the same. We measured that the final result of the rewriting process needs 39,834 heap cells without representation sharing, and only about 200 with representation sharing.

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

Note that the boyer benchmark also benefits a lot from tabling (Chen and Warren 1996). This means that repeated computations are going on, which also explains the high amount of representation sharing. However, while tabling does avoid the repetition of duplicate computations, as usually implemented, it does not avoid the creation of duplicate terms on the heap. It is possible to add enough information to the tries so that ground terms need be copied only once to the heap as long as this copy is not backtracked.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

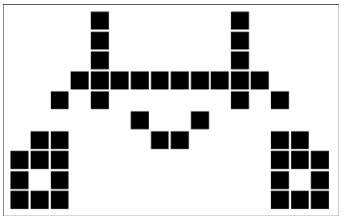


Fig. 9. The weekender.

8.1.2 Life

Appel and Gonçalves (1993) also uses the well-known *Game of Life* as a benchmark. We have written a version in Prolog following Reade's (1989) ideas, just as done by Appel and Gonçalves (1993). A (live) cell is represented as a tuple in coordinate form (X,Y). A generation is a list of live cells. The program keeps a list of the first 1,000 generations, starting from the *The Weekender*,[6] which is a glider, i.e., a pattern that repeats itself after a few generations (seven in this case) translated a few cells (two in this case). If just the most recent generation is kept alive, one expects little from running the sharer immediately after a major collection, as the just rewritten generation is garbage. Our benchmark still shows some 50% memory improvement because it keeps all computed generations in a list so that the existing overlap between generations is shared.

Appel and Gonçalves (1993) show little gain from hash consing for this benchmark, but we could not retrieve the initial generation(s) on which the benchmark was run.

8.1.3 Mandelbrot

This benchmark was also used by Appel and Gonçalves (1993): it computes (actually outputs) a bitmap of a Mandelbrot set of a given dimension. As 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 Glendon Holst's version. Mandelbrot uses quite a bit of heap and as such appears a good memory benchmark. However, one can quickly see that literally *all* memory used by mandelbrot is by floating point numbers: computed floating point numbers have the tendency to be different and therefore representation sharing might not have much effect. We have indeed checked that half of the generated floating point numbers are unique during the benchmark.

Almost all the floating point numbers are generated during a ground call to *mandel/5*, a semidet predicate called as the condition in an if-then-else as follows:

```
(mandel(Height, Width, Y, X, 50) ->
        ByteOut1 is (ByteOut0 << 1)
;
        ByteOut1 is (ByteOut0 << 1) \/ 0x1
)
```

[6] See http://fano.ics.uci.edu/ca/rules/b3s23/g10.html

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

In the setting of Appel and Gonçalves (1993) (generational collection + hash-consing), the Mandelbrot benchmark 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 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.

The effect of hash-consing or representation sharing is expected to be very small for the mandelbrot benchmark.

Our test runs of the mandelbrot benchmark indeed show zero gain from representation sharing.

8.1.4 One more classical Prolog benchmark: tsp

We were unable to retrieve more benchmarks from Appel and Gonçalves (1993), so we tried different benchmarks from the established general Prolog benchmark suite. None showed any benefit from representation sharing. We report only on *tsp*: just like *mandelbrot* and the other benchmarks showing no benefit, it is mainly good for showing the overhead of the useless sharer.

8.1.5 blid/1

The next program was slightly altered from what Neumerkel (1989) had posted in comp.lang.prolog; it also appeared in his Diplomarbeit.

```
blid(N) :-                          blam([]).
       length(L, N),                blam([L|L]) :-
       blam(L),                            blam(L).
       id(L,K),
       use(K).                      id([], []).
                                    id([L1|R1], [L2|R2]) :-
                                           id(L1,L2), % L1 = L2
use(_).                                    id(R1,R2). % R1 = R2
```

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 *after GC* policy, the following happens:
  - a1: the first GC finds that 99% (or more) of the data are live, and decides to expand the heap.
  - a2: The sharer shares most data.
  - a3: The next triggered GC finds that about half of the heap is live, so does not expand.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

a4: The following sharer shares most of the data.
a5: Points a3 and a4 are repeated.

- With the *between GC* policy, the following happens:

b1: The first GC finds that 99% (or more) of the data are live, and decides to expand the heap.
b2: The sharer shares most of the data.
b3: the second GC collects almost all the data.
b4: points b1, b2, and b3 are repeated.

The first GC in a1 and b1 is triggered by lack of space, the second GC (in b3) is there by policy. A GC can decide to expand the heap (in hProlog when the occupancy is more than 75%: 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.

If hProlog also had a heap shrinking policy, the *between GC* policy would after its second collection shrink the heap, and this would amount to almost the same effect as the *after GC* policy.

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 unifications, and with unification factoring these would also introduce the sharing needed to run in O(N) heap (of course, always with the aid of GC).

8.1.6 Four applications

The next four benchmarks provide some insight in what to expect from the sharer in a some typical applications of Prolog: there is little impact on memory and performance.

Treelearner. This realistic benchmark consists of the *best-first relational regression tree-learner* written by Gutmann (Gutmann and Kersting 2006). The program is about 900 LOC. It works on a data set of 350-K facts.

Emul. Emul is a BAM emulator (Van Roy and Despain 1992) written by Van Roy in Prolog. The benchmark consists in executing the BAM code for the famous *SEND+MORE=MONEY* problem. It is about 1-K LOC.

An XSB compiler. xsbcomp is an old version of the XSB compiler (Sagonas *et al.* 1994) and a run of the benchmark consists in compiling itself. The XSB compiler is about 5-K LOC and also uses the hProlog or SICStus Prolog reader, which are also in Prolog.

The hProlog compiler. In this benchmark, the hProlog compiler compiles itself. It uses *setarg/3* heavily, so we need a *mutable/1* (see Section 9) declaration for seven functors. This benchmark cannot be run by SICStus. The hProlog compiler (which is a version of the hipP compiler (Blockeel *et al.* 2000) written by Henk Vandecasteele) plus all other codes it needs (reader, optimizer ...) total more than 10-K LOC.

8.1.7 Worst and best cases

The best and the worst cases for our sharer is not clear: 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

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

the hashed_terms table, and in some sense this is both the best and the worst, because the least time is lost in collisions etc., but also no sharing can be performed. So we choose the following as the best *versus* the worst case: a large complete binary tree in which every node is of the form *node(tree,tree,number)*. In what we consider as the best case, the number is always the same (and thus resembles a bit the blid data structure in Section 8.1.5). This leads to a very sparse hashed_terms table, and a large amount of sharing. In the worst case, the number is different in all nodes: as a result, the hashed_terms becomes quite full, and no sharing is possible at all. The main reason for this benchmark is to find out how the build and the absorb phases contribute to the total time of the sharer.

8.2 The benchmark results for representation sharing

The results are shown in Table 3. Time is in milliseconds. Space is in Mib or Kib, as indicated in the table.

The first two columns denote the benchmark and system used (with the sharing option for hProlog). Then follow the total time taken by heap garbage collection (including the stack shifter), the total time taken by the sharing module, the total execution time, and the number of garbage collections. Then follow four columns related to space: the initial heap size, the final heap size, and the amount of space collected by the garbage collections are given in megabytes. Finally, there is the heap high water mark at the end of the benchmark given in KiB instead of Mib because the figures vary widely. It measures the size of the result computed by the benchmark and includes a small system specific overhead from the toplevel: for *mandelbrot*, the figure is just that overhead.

Table 3 sometimes shows a large difference between the memory consumption of SICStus Prolog and hProlog. Also, the time spent in garbage collection, and the number of collections can be very different. The reason is that although both systems are based on the WAM, they differ in a number of other design decisions. In particular, their heap expansion policy differs, their garbage collectors differ (the SICStus Prolog one is generational and compacting, while the hProlog one is non-generational and copying), they have a different approach to floating point arithmetic, and hProlog does not allocate free variables in the local stack.

In addition to the results given in Table 3, we can also mention that the build phase takes between 8.3 (for blid) and 2.6 (for worst) times as long as the absorb phase: the absorb phase is indeed much simpler.

8.3 Conclusions from the benchmarks

By and large our results confirm the findings of Appel and Gonçalves (1993): most benchmarks hardly benefit from representation sharing, and sometimes the space and time performance becomes 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 Appel and Gonçalves (1993). We have not been able to pinpoint why: the benchmarks used by Appel and Gonçalves are not even available anymore, let alone the *queries*. The fact is that generational collection retains terms longer than an only-major-collections strategy might plays a role. Still, our result is in line with the (confluent) rewriting character of boyer.

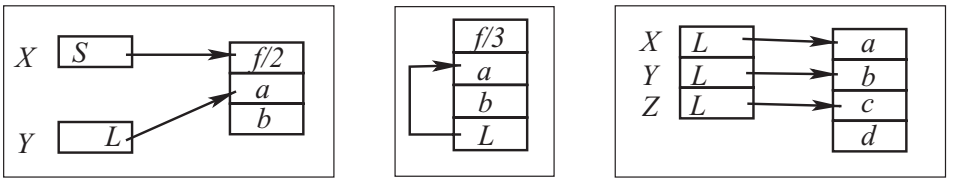
https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

Table 3. *The Sharer and the collector.*

Bench	System	GC time (msec)	Share time (msec)	Total runtime (msec)	#GC	Initial heap (Mib)	Final heap (Mib)	Collected (Mib)	At end (Kib)
boyer	hProlog -r0	920	0	3,130	19	6.10	24.42	96.75	6953.96
boyer	hProlog -r1	280	380	2,820	24	6.10	6.10	111.60	287.76
boyer	hProlog -r2	260	380	2,750	36	6.10	6.10	109.74	0.91
boyer	SICStus	1440	–	6,170	83	6.10	26.01	240.76	6954.23
life	hProlog -r0	2,570	0	28,960	100	3.76	15.04	17.70	8896.39
life	hProlog -r1	1,400	3,060	30,860	100	3.76	7.52	22.23	4211.53
life	hProlog -r2	2,820	2,910	32,200	200	3.76	7.52	22.27	4164.63
life	SICStus	4,730	–	69,250	100	3.76	9.91	22.20	8896.60
mandelbrot	hProlog -r0	10	0	29,370	360	16.04	16.04	5774.14	0.09
mandelbrot	hProlog -r1	50	0	29,460	360	16.04	16.04	5774.14	0.09
mandelbrot	hProlog -r2	20	10	29,390	720	16.04	16.04	5774.14	0.09
mandelbrot	SICStus	2,090	–	219,070	171	16.04	16.02	2656.25	0.36
tspgc	hProlog -r0	1,190	0	47,960	1,625	3.76	3.76	5684.49	258.23
tspgc	hProlog -r1	1,070	1,610	52,030	1,625	3.76	3.76	5684.49	258.23
tspgc	hProlog -r2	2,730	1,770	55,540	3,250	3.76	3.76	5684.49	258.23
tspgc	SICStus	3,760	–	96,430	1,304	3.76	3.77	2867.55	258.43
blid	hProlog -r0	1,860	0	2,480	6	3.76	240.64	0.01	131072.08
blid	hProlog -r1	970	1,910	3,520	34	3.76	7.52	123.94	301.76
blid	hProlog -r2	860	1,750	3,250	10	3.76	120.32	116.52	0.26
blid	SICStus	5,660	–	7,050	15	3.76	179.22	0.00	131072.33
worst	hProlog -r0	40	0	80	1	16.04	16.04	0.01	8192.06
worst	hProlog -r1	70	110	240	1	16.04	16.04	0.01	8192.06
worst	hProlog -r2	120	100	280	2	16.04	16.04	0.01	8192.06
worst	SICStus	90	–	240	1	16.04	16.05	0.00	8192.33
best	hProlog -r0	40	0	80	1	16.04	16.04	0.07	8192.06
best	hProlog -r1	70	100	220	1	16.04	16.04	0.01	0.37
best	hProlog -r2	70	110	240	2	16.04	16.04	8.01	0.37
best	SICStus	90	–	230	1	16.04	16.05	0.00	8192.33
treelearner	hProlog -r0	990	0	41380	88	35.19	35.19	2936.31	0.08
treelearner	hProlog -r1	920	760	39900	88	35.19	35.19	2936.50	0.08
treelearner	hProlog -r2	1980	750	40570	176	35.19	35.19	2936.75	0.08
treelearner	SICStus	5340	–	95860	75	42.13	42.08	2947.34	0.35
emul	hProlog -r0	30	0	4,610	70	3.76	3.76	258.28	0.08
emul	hProlog -r1	60	20	4,500	70	3.76	3.76	260.04	0.08
emul	hProlog -r2	80	20	4,310	140	3.76	3.76	260.08	0.08
emul	SICStus	740	–	8,330	90	3.76	3.72	318.77	0.36
xsbcomp	hProlog -r0	20	0	390	3	3.76	3.76	10.02	0.07
xsbcomp	hProlog -r1	20	0	360	3	3.76	3.76	10.48	0.07
xsbcomp	hProlog -r2	20	0	410	6	3.76	3.76	10.72	0.07
xsbcomp	SICStus	30	–	850	4	3.76	3.77	12.96	0.35
dpcomp	hProlog -r0	190	0	850	13	3.81	3.81	29.76	0.07
dpcomp	hProlog -r1	130	110	900	11	3.81	3.81	29.62	0.07
dpcomp	hProlog -r2	240	120	1,010	22	3.81	3.81	30.30	0.07

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

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press



$X = f(a,b)$, $Y = [a|b]$ f(a,b,[a|b]) $X = [a|b], Y = [b|c], Z = [c|d]$

Fig. 10. Unusual representation sharing.

complicated, as one does not need to take into account the liveness of the locations anymore and less copying is going on. In our application benchmarks, the sharer always takes less time than the garbage collection. It is clear that a better policy, and improvements to our implementation code, can make the sharer even more efficient. Our sharer does not depend on the efficiency of the underlying Prolog system, neither its garbage collector, so we feel it is safe to say that our sharer can be implemented with the same (or better) performance in other WAM-like systems.

9 Variations, extensions, and related issues

Unusual sharing. In Demoen (2002), the rather unusual representation sharings, as depicted in Figure 10, are described.

Our current representation sharing implementation does not achieve the above sharings. Still, all ingredients are present and while the expected gains are small, it is nice that the above unusual sharing can be achieved in time linear in the size of the heap (assuming perfect hashing).

Cyclic terms. Appel and Gonçalves (1993) deals with cyclic terms by excluding them from hash-consing. It is easy to do the same in our implementation as follows:

(1) Besides the special values **no-info** and **impossible**, cached_hash entry can also have the value **busy**.
(2) When a functor cell is visited for the first time, that corresponding cached_hash entry is set to **busy**.
(3) When a functor cell is visited recursively, a check on the corresponding cached_hash entry detects that there is a cycle: the field is set to **impossible**.
(4) As usual, when a term is visited completely, its corresponding field is set to an appropriate value, i.e., **impossible** or a pointer to the hashed_terms table.

However, one can do better: A variation in the above-stated point 3 yields a procedure that can perform representation sharing also for cyclic terms.

(3') When a functor cell is visited recursively, a check on the corresponding cached_hash entry detects that there is a cycle: a fixed value (say 17) is returned as the hash value of this term; the corresponding cached_hash entry is not updated at this time: this happens when the visit has returned to the point where the entry was set to **busy**.

The procedure for testing equality of terms must also be adapted to correctly deal with cycles: this is a common practice now in most Prolog systems.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

Note that it does not matter which value is chosen in point (3') above. What only matters is that the hash value of terms that can share their representation is the same. Still, our procedure can attach a different hash value to cyclic terms that are equal (in the sense of *==/2*) and could share their representation. This results in no representation sharing for those cyclic terms. As an example

```
test :- X = f(1,f(1,X)), share, use(X).
```

does not result in the same heap representation as

```
test :- X = f(1,X), use(X).
```

The procedure based on minimization of finite automata described by Neumerkel (1989) does this.

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

However, just as a ground mutable term must be copied by *copy_term/2*, a mutable term itself is not allowed to absorb or be absorbed. This means that mutable terms should be recognizable during the build phase. In SICStus Prolog this is the case (*$mutable/2* is reserved for this), but not so in other systems (e.g., SWI Prolog, Yap, hProlog ...). In hProlog we have resolved that problem by introducing a declaration: *:- mutable foo/3*, which declares that the arguments of any *foo/3* term can be destructively updated, and effectively prevents sharing of *foo/3* terms. We use one bit in the functor table and the overhead during the build phase is unnoticeable. Note that the *:- mutable* declaration does not readily work across modules.

Cooperation between collector and sharing. We have implemented the representation sharing module independent of the garbage collector module. The advantage is less dependency and a higher potential that the sharer can be integrated in other systems. The disadvantage is that some information that the garbage collector has computed, needs to be recomputed by the sharing module. For instance, the collector might leave behind information on the cells that are trailed, and the cells that contain sharable information. This would speed up the sharer and in particular the build phase.

What if representation sharing does not work. The benchmark programs show that representation sharing is not always effective: It indeed depends highly on the type of program. The non-working of representation sharing can be noticed during a run of the representation sharing module by observing the hashed_terms. 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, as no changes to the WAM run-time data structures are made until the absorb phase in which structure (or list ...) pointers are updated, and even the absorb phase can be stopped before finishing. Also, if representation sharing is run from time to time only – as suggested by Neumerkel (1989) – then the frequency of running it can take into account shows the effectiveness of representation

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

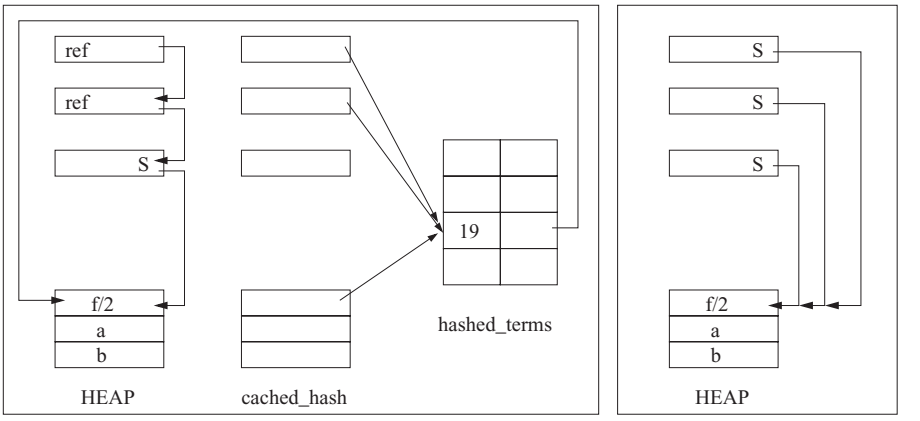


Fig. 11. Absorption for chains of references in action. (a) Just before absorbing, and (b) after absorbing.

sharing up to that moment. Such tuning could also depend on the relative performance of the garbage collector and the representation sharing module.

Parallelization. During the scanning phase, the stacks (heap, local stack ...) are read-only, while the cached_hash and the hashed_terms can be read and written by different workers. During the absorb phase, the cached_hash and hashed_terms are read-only, and only the stacks are written.

By giving different workers a different part of the heap to start working on, duplicate work might be avoided and synchronization slowdown could be kept low in the scanning phase. During the absorb phase, giving different workers different parts of the stacks makes their actions completely independent.

Variable chains. We have not treated variable chains in much detail, 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 by Sahlin and Carlsson (1991), but is not as *complete* as the method described by them. Figure 11 shows an example of how a chain of references is transformed.

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, regardless of whether they have trailed cells or not. The change made (to a LIST or STRUCT-tagged pointer) by the absorb phase is now conditionally (and value) trailed. This, of course, costs extra trail space. On cut, the trail can be tidied, so in case the computation becomes eventually deterministic, the amount of sharing can be arbitrarily larger than without this form of backtrackable representation sharing. However, suppose that all sharing are trailed, then it is possible that an immediately following GC would not be able to recover anything. Moreover, if the computation eventually becomes deterministic, running the sharer will do the same job as was done in the case of the backtrackable representation sharing, only later – which might be even better, because the earlier sharing

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

could have been unnecessary as backtracking has destroyed it. All in all, our feeling is that backtrackable representation sharing is not worth its while.

Partial sharing. Partial sharing refers to running the sharer in an incomplete way, i.e., it achieves part of its potential effect, but maybe not all.

Partial sharing can result, for instance, from restricting the part of the heap in which duplicate terms are identified, i.e., restricting the scan phase to part of the heap. Another possibility is to restrict sharing to certain terms, e.g., just for lists, or to certain parts of other stacks. It is one of the strengths of our implementation approach that all such variations can be incorporated rather easily.

Incremental sharing. The notion of incremental sharing refers to the possibility to perform a partial sharer pass, e.g., on part of the heap, and continue that pass later on, eventually obtaining the same effect as running the sharer completely. The ability to perform partial sharing is certainly needed, but there is more: Information must be passed from one partial run to the other, and the user program and the sharer must be able to run in an interleaved way. This immediately raises the question of completeness, but efficiency is also at stake.

The issues with incremental sharing are similar to the ones with generational sharing in next paragraph and we do not discuss further incremental sharing separately.

Generational sharing. The notion of generational sharing refers to the possibility to avoid performing sharing on the part of the heap on which it was performed earlier. In analogy with generational garbage collection, there is a rationale for performing generational sharing: for generational garbage collection, the rationale is that new objects tend to die quickly. For generational sharing the rationale is that redoing sharing on old data (on which sharing was performed earlier) does not pay off.

Our strategy to non-generational sharing is to recompute the cached_hash and hashed_terms tables from scratch every time after a new garbage collection. With generational sharing, one would like to reuse the part of the tables corresponding to the older generation.

We reason about forward computation first. The information on terms in the older generation that were ground at the previous run of the sharer and eligible for sharing at that moment is still valid. The same is generally not true for a non-ground term: It can now contain cells that are trailed, and in that case the information about the term is discarded from the table, or at least not used. Since it is not straightforward to keep track of the information that in tables is no longer valid because of this reason, it might be best to restrict a generational sharer to ground terms only.

Now suppose that backtracking has taken place between two activations of the sharer: generally, this invalidates entries in the sharer tables because terms have disappeared. It is easy to adapt the cached_hash table (it shrinks with the heap on backtracking), but the hashed_terms table also needs to be adapted. By keeping high and low water marks of the top of heap pointer, this can also be achieved. However,the cost of adapting the tables might be larger than the cost of rebuilding them.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

10 Related work

Appel and Gonçalves (1993) describe how hash-consing can be performed during garbage collection in the 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 longer. The reported performance and space gains are disappointing: half of the benchmarks lose performance (up to 25%) and the gain is up to 10% (for boyer). The space improvement is even smaller: on most benchmarks less than 1%. Also for space, 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 collected infrequently. As such, these numbers do not give full insight in the potential of hash-consing. Still, Appel and Gonçalves (1993) 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 just survived the collection.

Mercury (Somogyi *et al.* 1996) is basically a functional language, and the issue of trailing does not enter. In August 1999, in the developers' mailing list the issue of hash consing was raised with 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 *:- mutable* declaration was 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.

Last but not least, Neumerkel (1989) provides the example *blid/1* and gives a high-level outline of an algorithm for the minimization of heap terms seen as deterministic finite automata (DFAs). Our implementation can be seen as a concrete version of that algorithm. However, our *minimization* mostly shows similarities with Ershov (1958), in which he uses (for the first time in the published history of computer science) hashing to detect common subtrees in a given tree.

11 Conclusion

Without the questions raised by Neumerkel (1989) on comp.lang.prolog, we would not have worked on this topic and we are grateful for his insistence that Prolog systems should have a sharer. 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. We have also shown that input sharing for *findall/3* is easy to implement.

Acknowledgements

We thank the anonymous referees, as their suggestions have considerably improved the paper. The second author acknowledges support from project "GOA/08/008 Probabilistic

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

Logic Learning," and from the Research Foundation Flanders (FWO) through projects "WOG: Declarative Methods in Computer Science" and "G.0221.07 Platform independent analysis and implementation of Constraint Handling Rules."

Appendix
The relevant part from the program in O'Keefe (2001)

```
tree_children(node(_,Children), Children).

top_pointer(Tree, ptr(Tree,[],[],no_ptr)).

up_down(P, ptr(T,L,R,A)) :-
        (   var(P) ->
            A = ptr(_,_,_,_), % not no_ptr, that is.
            P = A
        ;   A = P,
            P = ptr(Tree,_,_,_),
            tree_children(Tree, Children),
            % split Children++[] into reverse(L)++[T]++R
            split_children(Children, [], L, T, R)
        ).

split_children([T|R], L, L, T, R).
split_children([X|S], L0, L, T, R) :-
        split_children(S, [X|L0], L, T, R).

f1(N) :- mk_tree(N,T),
         top_pointer(T,P),
         findall(Q, up_down_star(P, Q), _L).

up_down_star(A, D) :-
        (   A = D
        ;   up_down_plus(A, D)
        ).

up_down_plus(A, D) :-
        (   var(A) ->
            up_down(X, D),
            up_down_star(A, X)
        ;   up_down(A, X),
            up_down_star(X, D)
        ).

mk_tree(D, node(D,C)) :-
        (   D > 0 ->
            D1 is D - 1,
            C = [T1,T2,T3,T4],
            mk_tree(D1, T1),
            mk_tree(D1, T2),
            mk_tree(D1, T3),
            mk_tree(D1, T4)
        ;   C = []
        ).
```

References

AÏT-KACI, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA.

APPEL, A. W. AND GONÇALVES, M. J. R. 1993 (Feb). *Hash-consing Garbage Collection*. Technical Report CS-TR-412-93, Com- puterr Science Department, Princeton University.

APPLEBY, K., CARLSSON, M., HARIDI, S. AND SAHLIN, D. 1988. Garbage collection for Prolog based on WAM. *Communications of the ACM 31,* 6 (June), 719–741.

BAKER, H. G. 1992. The Boyer Benchmark at warp Speed. *SIGPLAN Lisp Pointers V,* 3 (Jul–Sep), 13–14.

BEVEMYR, J. AND LINDGREN, T. 1994. A simple and efficient copying garbage collector for Prolog. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, M. Hermenegildo and J. Penjam, Eds. Lecture Notes in Computer Science, no. 844. Springer-Verlag, New York, 88–101.

BLOCKEEL, H., DEHASPE, L., DEMOEN, B., JANSSENS, G., RAMON, J. AND VANDECASTEELE, H. 2000. Executing query packs in ILP. In *Proceedings of Inductive Logic Programming, 10th International Conference, ILP2000, London, UK, July*, J. Cussens and A. Frisch, Eds. Lecture Notes in Artificial Intelligence, vol. 1866. Springer, New York, 60–77.

BOEHM, H.-J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience 18*, 807–820.

CARLSSON, M. 1990. *Design and Implementation of an Or-Parallel Prolog Engine*. Ph.D. thesis, The Royal Institute of Technology (KTH), Stokholm, Sweden. See also http://www.sics.se/isl/sicstus.html

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

CHEN, W. AND WARREN, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43*, 1 (January), 20–74.

CLOCKSIN, W. AND MELLISH, C. 1984. *Programming in Prolog*. Springer-Verlag, New York.

DEMOEN, B. 2002. A different look at garbage collection for the WAM. In *Proceedings of ICLP2002 – International Conference on Logic Programming*, P. Stuckey, Ed. Lecture Notes in Computer Science no. 2401. ALP, Springer-Verlag, Copenhagen, Denmark, 179–193.

DEMOEN, B. AND NGUYEN, P.-L. 2000. So many WAM variations, so little time. In *Computational Logic – CL2000, Proceedings of First International Conference, London, UK, July*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv and P. J. Stuckey, Eds. Lecture Notes in Artificial Intelligence, vol. 1861. ALP, Springer, New York, 1240–1254.

ERSHOV, A. P. 1958. On programming of arithmetic operations. *Commun. ACM 1*, 8, 3–6.

GABRIEL, R. P. 1985. *Performance and Evaluation of Lisp Systems.* MIT Press.

GOTO, E. 1974. *Monocopy and Associative Algorithms in Extended Lisp*. Technical Report TR 74-03, University of Tokyo, Tokyo, Japan.

GRIMLEY-EVANS, E. 1995. Findall/3 without copying? [online]. Accessed 14 July 2011.

http://www.logicprogramming.org/newsletter/archive_93_96/net/meta-level/findall1.html

GUTMANN, B. AND KERSTING, K. 2006. Tildecrf: Conditional random fields for logical sequences. In *Proceedings of the 15th European Conference on Machine Learning (ECML-2006)*, J.Fürnkranz, J. Scheffer and T. Spiliopoulou, Eds. Springer, New York, 174–185.

HIRSCH, M., SILVERMAN, W. AND SHAPIRO, E. 1987. Computation control and protection in the logix system. In *Concurrent Prolog: Collected Papers,* vol. 2, E. Shapiro, Ed. MIT Press, Cambridge, MA, 28–45.

MARIËN, A. AND DEMOEN, B. 1993. Findall without findall/3. In *Proceedings of the Tenth International Conference on Logic Programming*, D. S. Warren, Ed. MIT Press, Cambridge, MA, 408–423.

NEUMERKEL, U. 1989. *Speicherbereinigung für Prologsysteme*. M.S. thesis, Institut für Praktische Informatik der Technischen Universität Wien, Austria.

O'KEEFE, R. 2001. O(1) reversible tree navigation without cycles. *Theory and Practice of Logic Programming 1*, 5 , 617–630.

READE, C. 1989. *Elements of Functional Programming*. Addison-Wesley, Massachusetts.

SAGONAS, K., SWIFT, T. AND WARREN, D. S. 1994. XSB as an efficient deductive database engine. In *Proceedings of SIGMOD 1994 Conference*, vol. 23:2. ACM, New York, 442–453.

SAHLIN, D. AND CARLSSON, M. 1991. *Variable Shunting for the WAM*. Technical Report SICS/R-91/9107, SICS, Stockholm, Sweden.

SOMOGYI, Z., HENDERSON, F. AND CONWAY, T. 1996. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29*, 1–3, 17–64.

TARAU, P. 1991. A simplified abstract machine for the execution of binary metaprograms. In *Proceedings of the Logic Programming Conference'91*. ICOT, Tokyo, 119–128.

TARAU, P. 1992. Ecological memory management in a continuation passing prolog engine. In *Proceedings of IWMM'92: International Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds. Lecture Notes in Computer Science, no. 637. Springer-Verlag, New York, 344–356.

TARAU, P. AND MAJUMDAR, A. 2009. Interoperating logic engines. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009,* Savannah, GA, Jan 19–20. LNCS 5418, Springer, New York, 137–151.

VAN ROY, P. AND DESPAIN, A. 1992. High-performance logic programming with the aquarius prolog compiler. *IEEE Computer 25*, 1, 54–68.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press

VANDEGINSTE, R., SAGONAS, K. AND DEMOEN, B. 2002. Segment order preserving and generational garbage collection for Prolog. In *Proceedings of Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002*, S. Krishnamurthi and C. Ramakrishnan, Eds. Lecture Notes in Computer Science, vol. 2257. Springer, New York, 299–317.

WALLACE, M., NOVELLO, S. AND SCHIMPF, J. 1997. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal 12*, 1 (May), 159–200.

WARREN, D. H. D. 1983. *An Abstract Prolog Instruction Set*. Technical Report 309, SRI, California.

WIELEMAKER, J., HUANG, Z. AND VAN DER MEIJ, L. 2008. SWI-Prolog and the web. *Theory and Practice of Logic Programming 8*, 3, 363–392.

https://doi.org/10.1017/S1471068411000421 Published online by Cambridge University Press