

Input Sharing for findall/3

*Phuong-Lan Nguyen
Bart Demoen*

Report CW 553, June 2009



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

Input Sharing for findall/3

Phuong-Lan Nguyen
Bart Demoen

Report CW 553, June 2009

Department of Computer Science, K.U.Leuven

Abstract

A simple addition to *findall/3* achieves sharing between answers in the solution list and the input to *findall/3*. This reduces the time and space complexity of some queries. Its overhead is extremely small. It can be integrated in any WAM-like Prolog implementation.

Input Sharing for findall/3

Phuong-Lan Nguyen[®] and Bart Demoen^{*}

[®] Institut de Mathématiques Appliquées, UCO, Angers, France
Department of Computer Science, K.U.Leuven, Belgium

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

Abstract. A simple addition to *findall/3* achieves sharing between answers in the solution list and the input to *findall/3*. This reduces the time and space complexity of some queries. Its overhead is extremely small. It can be integrated in any WAM-like Prolog implementation.

1 Introduction

In [7] it is suggested that *findall/3* could avoid repeatedly copying the same terms over and over again. This would - in the programming pearl context at hand - improve the space complexity of some queries that use *findall/3*, from $O(n^2)$ to $O(n)$. However, the author suggests that hash-consing should be used, with the consequence that the time complexity remains the same. The example used in [7] is rather complicated, so we use as an illustration a piece of simple Prolog code that can be found already in [5] up to the names of the predicates.

```
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`. E.g.,

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

This can be proven by applying the transformations described in [10] or [6].

Current implementations of *findall/3* copy 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

sharing between the solutions of the output list of `findall/3`, and preferably even with the input to its generator, the `findall_tails/2` query would also be linear.

In [6], the notions *input sharing* and *solution sharing* were introduced: input sharing consists in a solution in the output from `findall/3` (its third argument) sharing with the input to `findall/3` (in its second argument). Solution sharing consists in different elements of the output list of `findall/3` sharing between them some data which is not part of the input. Solution sharing is exemplified by the following example:

```
t(N,L) :- findall(T,rep_mkterm(N,T),L).

rep_mkterm(N,T) :- between(1,N,_), T = foo(1,2,3,4,5,6,7,8,9,0).
```

Clearly, all the elements of the solution list can be shared, reducing the space by the constant factor `N` - but a more contrived example would show that space requirements could be reduced complexity wise.

We set out to provide a solution to input sharing only. 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. Alternative implementations of `findall/3` are also investigated and compared.

Before going into the details, it is worth pointing out the limitations of input sharing: clearly, if `L` is not ground, the queries

```
?- findall_tails(L,Tails).
and
?- 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), numbervars(L,0,_).

L = [[A,B,C],[D,E],[F],[ ]]

?- all_tails([X,Y,Z],L), numbervars(L,0,_).
X = A
Y = B
Z = C
L = [[A,B,C],[B,C],[C],[ ]]
```

That is why we only consider using the sharing version of `findall/3` when arguments of the generator are either ground or free.

We have used `hProlog` as the Prolog engine to experiment with, but it is clear that everything can be ported to other WAM-like systems as well: we make that more explicit later on. `hProlog` is a descendant of `dProlog` as described in [4].

We assume the reader to be familiar with the WAM [1,11] and Prolog [3].

2 The implementation of findall/3 in hProlog

The hProlog implementation of findall/3 follows the same pattern as in every system:

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

We have left out the checking of the arguments (required by ISO), and some error-recovery code. 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, 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 The basic Idea

The predicate findall_add/2 used in our definition of findall/3 is just a particular use of copy_term/2. At the implementation level, they both use the same C-function for the actual copying.

Some implementations (e.g. SICStus Prolog [2]) avoid copying ground sub-terms during copy_term/2. E.g. ?- X = foo(9), copy_term(X,Y). results in X and Y sharing the same term, and the heap consumption of that call to copy_term/2 is zero.

Groundness is not enough in the context of findall/3: the ground term must also be *old enough*, so that backtracking (over the Generator) cannot alter it. To be more precise, anything ground that survives backtracking over the Generator need not be copied by findall_add/2. Or put another way: anything ground before the call to findall(Template,Generator,SolList) need not be copied.

An old enough ground term can be easily recognized: its *root* resides in heap part¹ older 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: just remember a heap pointer !

¹ We avoid the name *heap segment* because we need a notion that is not related to choice points.

4 Sharing_findall/3: the Implementation

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

- **current_heap(-)**: 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 of its argument

The following code shows how it is used:

```
sharing_findall(Template,Generator,SolList) :-
    current_heap(Barrier),
    findall_init(Handle),
    (
        call(Generator),
        set_copy_heap_barrier(Barrier),
        findall_add(Template,Handle),
        set_copy_heap_barrier(0),
        fail
    );
    set_copy_heap_barrier(Barrier),
    findall_get_solutions(SolList,Handle),
    set_copy_heap_barrier(0)
).
```

An additional (small) change to the implementation of findall_add/2 (and findall_get_solutions/2 and copy_term/2) needs to be made as well: when a term is copied that is older than the value of the Barrier, just the root pointer to this term is copied. It amounts to adding a statement like

```
if ((copy_heapbarrier) && (term < copy_heapbarrier))
    { *whereto = make_struct_p(term); continue; }
```

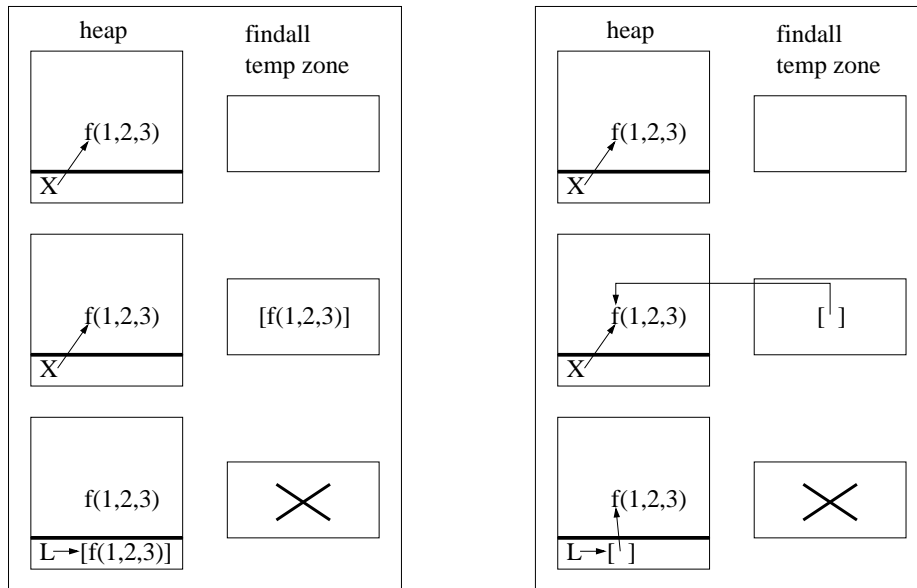
at a few places in the code.

It is clear that the functionality of the new low-level built-ins can be folded in adapted versions of findall_[init, add, get_solutions].

5 An Example

The heap and temporary findall zone is shown in the picture below 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 same with input sharing. The snapshots are taken

- just before `findall_add` is executed: the temporary zone is still empty
- just after `findall_add` 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` 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.

6 Copy-once findall/3

The usual implementation of `findall/3` copies the solutions twice. Bin-Prolog was probably the first implementation that copied the solution only once, by means of a technique named *heap lifting* or more popularly a *bubble in the heap* [8]. Currently, the Bin-Prolog implementation [9] relies on *engines* for `findall/3`. Mercury also uses a copy-once `findall` (named `solutions/2`): as Mercury relies on the Boehm-collector, there is no 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 [12]:

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

Before proceeding with the main issue of the paper, we point out some advantages and disadvantages of this particular copy-once implementation of findall/3:

- + the elements of the solution list are copied only once
- + there is no need for a bubble in the heap
- the spine of the solution list is copied twice, because the list needs to be reversed: a more complicated update of the argument of container/1 would do away with this need for reversing, but at the cost of more heap consumption and probably more time
- every destructive update *freezes* the heap, i.e. effectively prevents heap space to be reclaimed by backtracking (into the execution of Generator for instance)²; of course, the heap garbage collector can reclaim it at any point

As far as we are concerned, the latter disadvantage (freezing the heap) is a show-stopper, but we show anyway how to make the sharing work for the above implementation. The code is as follows:

² One needs to look into the implementation of nob_setarg/3 for understanding this to the full extent

```

sharing_copy_once_findall(Template,Generator,SolList) :-
    Term = container([],
    current_heap(Barrier),
    (
        Generator,
        Term = container(PartialSolList),
        set_copy_heap_barrier(Barrier),
        copy_term(Template,Y),
        set_copy_heap_barrier(0),
        nob_setarg(1,Term,[Y|PartialSolList]),
        fail
    );
    Term = container(FinalSolList),
    reverse(FinalSolList,[],SolList)
).

```

A small change to the C-code of `copy_term/2`, similar to the one for the `findall` related functions as in Section 4, completes this.

7 Experimental Evaluation

The current implementation of the sharing `findall/3` in hProlog is not yet robust for stack shifting or heap garbage collection, but it is dealing with nested calls to `findall/3`. Dealing with stack shifting and heap garbage collection does not affect the `findall/3` implementation at all.

In some sense, benchmarks are not necessary: sharing improves sometimes the complexity (space and time), and the constant overhead is really very small. Still, we did some benchmarking. We have used only two benchmarks: one is the `findall_tails/2` example from Section 1. The second one consists of some `findall/3` related queries from [7]: to quote from that paper

Query q1 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 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} .

The full program of [7] is included as an Appendix. The query `q1` mentioned above correspond to (our) predicate `f1/1` with increasing values as argument. Before doing the experiments, we first need to discuss the benchmark in more detail.

7.1 A closer Look at the [7] Benchmark

The `Mt_tree/2` predicate is defined as

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

The conjunction $C = [T, T, T, T]$, $Mt_tree(D1, T)$ is particularly interesting. Although the constructed tree has an size exponential in the first argument D , its heap representation is linear in D : indeed, the constructed tree has a lot of internal sharing. This can be easily observed in implementations like SICStus Prolog, Yap, SWI Prolog and hProlog, using their built-in statistics/2. The least one should require of a good findall/3 implementation is that it preserves this internal sharing. Of the above systems, only Yap does not retain the internal sharing. This can be easily found out by measuring the heap space used for goals

```
Mt_tree(N,T), findall(T,true,L).
```

with increasing N .

This would be a good reason to exclude Yap from further tests. However, this internal sharing and its preservation by the other systems also renders the statement *Query q1 requires at least $O(n^2)$ space to hold the result.* false for the other three systems. This can be observed by measuring the heap consumption of goals $f1(N)^3$ with increasing N .

So we took the liberty to change the particular conjunction into

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

so that the size of the representation of the tree is linear in the number of nodes in the tree. This *trick* only works because Prolog systems typically don't perform the little analysis needed to notice that $T1$, $T2$, $T3$ and $T4$ are declaratively the same value, and neither is this detected at runtime.

³ See the Appendix

7.2 The modified [7] Benchmark: the Results

Tables 1 and 2 show show timings and space consumption for queries $?- f1(Depth)$ with different values of Depth.

Depth	hProlog	hProlog sharing	SICStus Prolog	Yap	SWI Prolog
1	0	0	0	0	0
2	0	0	0	0	0
3	4	0	0	4	10
4	24	0	30	68	60
5	156	0	250	360	
6	2616	8	6820		
7		36			
8		156			
9		652			
10		2804			

Table 1. Timings in msec - tree benchmark

Table 1 actually doesn't show much in terms of complexity: the implementations without input sharing (all except the column for *hProlog sharing*) just can't deal with more than about 5000 nodes. But the sharing implementation can go easily up to one million nodes. The heap consumption Table 2 is much more clear.

Depth	hProlog	hProlog sharing	SICStus Prolog	Yap	SWI Prolog
1	964	368	980	1200	1108
2	11332	1840	10964	19568	13060
3	158020	9776	155348	329328	186180
4	2394436	49712	2379476	5380720	2854468
5	37599556	242224	37523156	86710908	
6	598030660	1143344	597659348		
7		5272112			
8		23884336			
9		106721840			
10		471626288			

Table 2. Heap in bytes - tree benchmark

All non-input-sharing implementations (i.e. all except *hProlog sharing*) show a quadratic dependency of the heap consumption on the number of nodes. Only *hProlog sharing* shows a linear dependency.

7.3 The *tails* Benchmark

The Tables 3 and 4 show timings and space consumption for the tails benchmark. The *Length* 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)*. The column *no findall* means the former query. Table 3 shows timings only for hProlog.

Length	hProlog findall	hProlog findall sharing	hProlog no findall	hProlog copy-once	hProlog copy-once sharing
1000	112	0	0	56	0
2000	444	0	0	232	0
3000	1028	0	0	520	0
4000	1920	0	0	932	0
5000		0	0		0
6000		0	0		0
7000		4	0		0
8000		4	0		4
9000		4	0		4
10000		4	0		4
100000		12	0		4

Table 3. Timings in msec - tails benchmark

When input sharing is retained, the timings are meaninglessly small, but *findall* with input sharing clearly beats the *findall* without sharing.

Length	hProlog findall	hProlog findall sharing	hProlog no findall	hProlog copy-once	hProlog copy-once sharing	SICStus Prolog findall	SICStus Prolog no findall
1000	26034072	8072	8048	26050092	24096	26034148	8056
2000	104068072	16072	16048	104100092	48096	104068148	16056
3000	234102072	24072	24048	234150092	72096	234102148	24056
4000	416136072	32072	32048	416200092	96096	416136148	32056
5000		40072	40048		120096	650170148	40056
6000		48072	48048		144096	936204148	48056
7000		56072	56048		168096		56056
8000		64072	64048		192096		64056
9000		72072	72048		216096		72056
10000		80072	80048		240096		80056
100000		800072	800048		2400096		800056

Table 4. Heap in bytes - tails benchmark

Table 4 shows the heap consumption for the same queries as in Table 3, but now also for SICStus Prolog. 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 policy and its different heap garbage collection method.

Also Table 4 shows clearly that our simple implementation to enforce input sharing is very effective and performs actually better than hoped for in [7]. Indeed, we achieve constant time space **and** time complexity for the `q1` query. Hash-consing would indeed not be able to do that.

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

8 Conclusion

There has been a demand for sharing between the input to `findall/3` and its output already in [5]. Also [7] points out that this would be beneficial to some programs. Although not possible in general, it seems like the usual case is that the input is ground. This case can be dealt with cheaply and effectively. It can also be done safely, i.e. there is no need to let the programmer decide on whether to call `findall/3` or `sharing_findall/3`: at runtime, it can be checked that the arguments to the Generator are either ground or a free variable. That is enough to allow `sharing_findall/3` to be called. However, this test can affect adversely calls to `findall/3` that are not linear in their input, or do not reproduce parts of the input in their output. Note also that this condition can be relaxed, but then might impose more runtime overhead.

As mentioned before, our approach does not solve the problem of solution sharing: hash-consing, or maybe even better tries, would be necessary. We are currently investigating this deeper. The sharing of terms that eventually become ground is even trickier and could lead to a degradation of the complexity properties one is used to. For instance, constant time unification between a free variable and an atom could become arbitrarily expensive complexity wise.

In [7], one can also read:

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 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 natural with the needs of a strict heap allocation discipline and conditional trailing.

We have succeeded in providing input sharing for `findall/3` with little change to the underlying Prolog execution engine: any Prolog implementation with a heap allocation strategy similar to the WAM can incorporate it easily.

Acknowledgements

Part of this work was while the second author enjoyed the hospitality of the Institut Xe Mathématiques Appliquées of the Université Catholique de l'Ouest in Angers, France. We thank Henk Vandecasteele for letting us use his hipP compiler.

References

1. H. Aït-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.
2. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stockholm, Sweden, Mar. 1990. See also: <http://www.sics.se/isl/sicstus.html>.
3. W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.
4. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *CL2000: Proceedings of the 1st International Conference on Computational Logic*, volume 1861 of *LNAI*, pages 1240–1254, London, UK, July 2000. ALP, Springer Verlag.
5. E. Grimley-Evans. Findall/3 without copying?, 1995. http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/archive_93_96/net/meta-level/findall1.html.
6. A. Mariën and B. Demoen. Findall without findall/3. In D. W. , editor, *Proceedings of the Tenth Int. Conf. on Logic Programming*, pages 408–423. MIT-Press, 1993.
7. R. O’Keefe. O(1) reversible tree navigation without cycles. *Theory and Practice of Logic Programming*, vol. 1, no. 5, pp 617 - 630, 2001.
8. P. Tarau. Ecological Memory Management in a Continuation Passing Prolog Engine. In Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM’92: International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 344–356. Springer-Verlag, Sept. 1992.
9. P. Tarau and A. Majumdar. Interoperating Logic Engines. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009*, pages 137–151, Savannah, Georgia, Jan. 2009. Springer, LNCS 5418.
10. K. Ueda. Making exhaustive search programs deterministic. In *ICLP*, pages 270–282, 1986.
11. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.
12. J. Wielemaker. SWI-Prolog release 5.4.0, 2004. <http://www.swi-prolog.org/>.

Appendix: the Program from [7]

```

tree_datum(node(Datum,_), Datum).

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

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

pointer_tree(ptr(Tree,_,_,_), Tree).

pointer_datum(ptr(Tree,_,_,_), Datum) :-
    tree_datum(Tree, Datum).

at_left(ptr(_,[],_,_)).

at_right(ptr(_,_,[],_)).

at_top(ptr(_,_,_,no_ptr)).

at_bottom(ptr(Tree,_,_,_)) :-
    tree_children(Tree, []).

left_right(ptr(T,L,[N|R],A), ptr(N,[T|L],R,A)).

up_down_first(P, ptr(T,[],R,P)) :-
    % P = ptr(tree(_,[T|R],_,_,_)).
    P = ptr(Tree,_,_,_),
    tree_children(Tree, [T|R]).

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).

siblings_before_after(ptr(T1,L1,R1,A), ptr(T2,L2,R2,A)) :-
    right_move([T1|L1], R1, L2, [T2|R2], L2).

right_move(L, R, L, R, _).
right_move(L1, [X|R1], L2, R2, [_|B]) :-
    right_move([X|L1], R1, L2, R2, B).

left_right_star(L, R) :-
    ( L = R
    ; left_right_plus(L, R)
    ).

left_right_plus(L, R) :-
    ( var(L) ->
        left_right(X, R),
        left_right_star(L, X)
    ; left_right(L, X),
        left_right_star(X, R)
    ).

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 = [T,T,T,T],
        mk_tree(D1, T)
    ; C = []
    ).

direct_datum(node(D,_), D).
direct_datum(node(_,C), D) :-
    member(N, C),
    direct_datum(N, D).

any_pointer_datum(T, D) :-
    top_pointer(T, P),
    up_down_star(P, N),
    pointer_datum(N, D).

labels(Tree, Labels) :-
    labels(Tree, Labels, []).

labels(node(Label,Children) -->
    [Label], labels_list(Children).

labels_list([]) --> [].
labels_list([Tree|Trees]) -->
    labels(Tree), labels_list(Trees).

collect(Tree, Labels) :-
    top_pointer(Tree, Ptr),
    collect(Ptr, Labels, []).

collect(Ptr, [Datum|Labels1], Labels) :-
    pointer_datum(Ptr, Datum),
    ( at_bottom(Ptr) -> Labels1 = Labels2
    ; up_down_first(Ptr, Child),
        collect(Child, Labels1, Labels2)
    ),
    ( at_right(Ptr) -> Labels2 = Labels
    ; left_right(Ptr, Sibling),
        collect(Sibling, Labels2, Labels)
    ).

q1(P, L) :- findall(Q, p1(P, Q), L).

p1(P, Q) :- up_down_star(P, Q).

f1(N) :- mk_tree(N,T), top_pointer(T,P), q1(P,_L).

```