10 Difference Lists. Side Effects

A prolog list is accessed through its head and its tail. The setback of this way of viewing the list is that when we have to access the n^{th} element, we must access all the elements before it first. If, for example, we need to add an element at the end of the list, we must go through all the elements in the list to reach that element.

```
add(X, [H|T], [H|R]) :- add(X, T, R).
add(X, [], [X]).
```

There is an alternative technique of representing lists in prolog that lets us access the end of a list easier. A difference list is represented by two parts, the start of the list S and the end of the list E:

```
S: [1,2,3,4]
E: [3,4]
S-E: [1,2]
```

S-E(the difference list) represents the list obtained by removing part E from part S:

There are no advantages when using difference lists like in the previous example, but when combined with the concepts of free variables and unification, difference lists become a powerful tool. For example, list [1,2] can be represented by the difference list [1,2|X]-X, where X is a free variable. We can write the add predicate with difference lists in the following way:

```
add(X,LS,LE,RS,RE):-RS=LS,LE=[X|RE].
```

We test it in Prolog by asking the following query:

```
?- LS=[1,2,3,4|LE],add(5,LS,LE,RS,RE).
LE = [5|RE],
LS = [1,2,3,4,5|RE],
RS = [1,2,3,4,5|RE] ?
yes
```

To better understand the way the add predicate works, we can imagine the list is represented by two “pointers”, one pointing to the start of the list (LS) and the second one to the end of the list (LE), a variable without an assigned value.

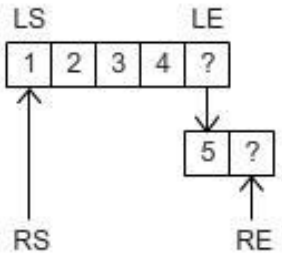

Figure 10.1 – Adding an element at the end of a difference list

The result is also represented by two “pointers”. The resulting list will be the input list with the new element inserted at the end. The beginning of the input and result lists is the same so we can unify the result list start variable with the input list start variable (RS=LS).

The result list must end, just like the input list, in a variable (RE), but we must, somehow, modify the input list to add the new element at the end. Because the end of the input list is a free variable, we can unify it with the list beginning with the new element followed by a new variable, the new end of the list (LE=[X|RE]). After the predicate finished execution we can see that the input list LS and result list RS have the same values, but the end of the input list is no longer a variable (LE=[5|RE]).

10.1 Tree traversal

The tree traversal predicates are used to extract the elements in a tree to a list in a specific order. The computation intensive part of these predicates is not the traversing itself but the combination of the result lists to obtain the final result. Although hidden from us, prolog will go through the same elements of a list many times to form the result list. We can save a lot of work by changing regular lists to difference lists.

The inorder predicate using a regular list to store the result:

```
inorder(t(K,L,R),List):- inorder(L,ListL), inorder(R,ListR), append1(ListL,[K|ListR],List).
inorder (nil,[]).
```

By executing a query trace on the inorder predicate we can easily observe the amount of work performed by the append predicate. It is also visible that the append predicate will access the same elements in the result list more than once as the intermediary results are appended to obtain the final result:

```
[. . .]
 8     3 Exit: inorder(t(5,nil,nil),[5]) ?
12     3 Call: append1([2],[4,5],_1594) ?
13     4 Call: append1([],[4,5],_10465) ?
13     4 Exit: append1([],[4,5],[4,5]) ?
12     3 Exit: append1([2],[4,5],[2,4,5]) ?
[. . .]
22     2 Call: append1([2,4,5],[6,7,9],_440) ?
23     3 Call: append1([4,5],[6,7,9],_20633) ?
24     4 Call: append1([5],[6,7,9],_21109) ?
25     5 Call: append1([],[6,7,9],_21585) ?
25     5 Exit: append1([],[6,7,9],[6,7,9]) ?
24     4 Exit: append1([5],[6,7,9],[5,6,7,9]) ?
23     3 Exit: append1([4,5],[6,7,9],[4,5,6,7,9]) ?
22     2 Exit: append1([2,4,5],[6,7,9],[2,4,5,6,7,9]) ?
[. . .]
```

We can improve the efficiency of the inorder predicate by rewriting it using difference lists. The inorder_dl predicate has 3 parameters: the tree node it is currently processing, the start of the result list and the end of the result list:

```
/* when we reached the end of the tree we unify the beggining and end of the partial result list –
representing an empty list as a difference list */
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-
/* obtain the start and end of the lists for the left and right subtrees */
inorder_dl(L,LSL,LEL),
inorder_dl(R,LSR,LER),
/* the start of the result list is the start of the left subtree list */
LS=LSL,
/* insert the key between the end of the left subtree list and start of the right subtree list */
LEL=[K|LSR],
/* the end of the result list is the end of the right subtree list */
LE=LER.
```

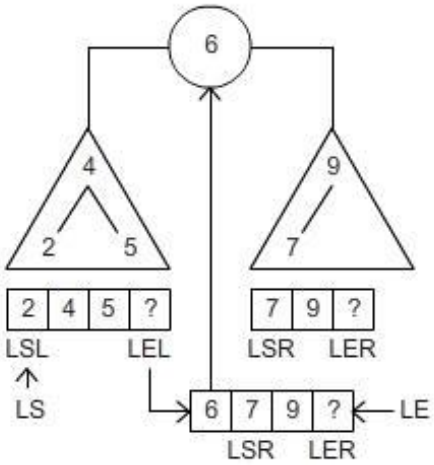

Figure 10.2 – Appending two difference lists

The predicate can be simplified by replacing the explicit unifications with implicit unifications:

```
inorder_dl(nil,L,L).
inorder_dl(t(K,L,R),LS,LE):-inorder_dl(L,LS,[K|LT]), inorder_dl(R,LT,LE).
```

Exercise 10.1: Study the execution of the following queries:

a. ?- tree1(T), inorder_dl(T,L,[]).
b. ?- tree1(T), inorder_dl(T,L,_).

Exercise 10.2: Implement the preorder_dl *tree traversal predicate using difference lists.*

Exercise 10.3: Implement the postorder_dl *tree traversal predicate using difference lists.*

10.2 Sorting – quicksort

Remember the *quicksort* algorithm (and predicate): the input sequence is divided in two parts – the sequence of elements smaller or equal to the pivot and the sequence of elements larger than the pivot; the procedure is called recursively on each partition, and the resulting sorted sequences are appended together with the pivot to generate the sorted sequence:

```
quicksort([H|T], R):-
        partition(H, T, Sm, Lg),
        quicksort(Sm, SmS),
        quicksort(Lg, LgS),
        append(SmS, [H|LgS], R).
quicksort([], []).
```

Just as for the `inorder` predicate, the `quicksort` predicate will waste a lot of execution time to `append` the results of the recursive calls. To avoid this we can apply difference lists again:

```
quicksort_dl([H|T],S,E):-
        partition(H,T,Sm,Lg),
        quicksort_dl(Sm,S,[H|L]),
        quicksort_dl(Lg,L,E).
quicksort_dl([],L,L).
```

The `partition` predicate is the same as the one for the old `quicksort` predicate, its purpose being to divide the list in two by comparing each element with the pivot. All elements in the list must be accessed for this operation so we cannot improve the performance for the partition predicate.

The `quicksort` predicate works in the same way as before: divides the list in elements larger and smaller than the pivot element and applies `quicksort` recursively on the each partition. The difference from the original version is the result list, represented by two elements, the start and the end of the list, and, consequently, the way in which the results of the two recursive calls are put together with the pivot (figure 10.3 below).

Exercise 10.4: Study the execution of the following queries:

a. `?- quicksort([4,2,5,1,3],L,[]).`
b. `?- quicksort([4,2,5,1,3],L,_).`

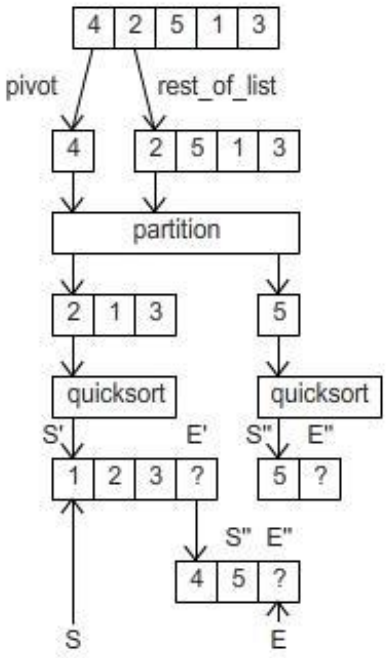

Figure 10.3 – Quicksort with difference lists

10.3 Side Effects

Side effects refer to a series of Prolog predicates which allow the dynamic manipulation of the predicate base:

- assert/1 (or assertz/1) - adds the predicate clause given as argument as last clause
- asserta/1 - adds the predicate clause given as argument as first clause
- retract/1 - tries to unify the argument with the first unifying fact or clause in the database. The matching fact or clause is then removed from the database

Predicates which can be manipulated via assert/retract statements at run-time are called dynamic predicates, as opposed to the static predicates that we have seen so far. Dynamic predicates should be declared as such. However, when the interpreter sees an assert statement on a new predicate, it implicitly declares it as dynamic.

The important aspects you need to know when working with side effects:

- Their **effect is maintained in the presence of backtracking**: i.e. once a clause has been asserted, it remains on the predicate base until it is explicitly retracted, even if the node corresponding to the assert call is deleted from the execution tree (e.g. because of backtracking).
- **assert - always succeeds; doesn't backtrack**
- **retract - may fail; backtracks;** retract respects the *logical update view* - it succeeds for all clauses that match the argument when the predicate was ***called***.

Exercise 10.5: To *understand how it works, try to execute the following query*[1]:

```
?- assert(insect(ant)),
assert(insect(bee)),
(retract(insect(I)),
  writeln(I),
  retract(insect(II)),
  fail
  ;                                                    true
).
```

You have probably observed that this query will output:

```
ant
bee
true
```

This is because even if the second call to retract will also delete the clause `insect(bee)`, when backtracking reaches the first retract call, the clause is still present in its logical view - it doesn't see that the clause has been deleted by the second `retract` call.

Prolog also has a `retractall/1` predicate, with the following behavior: it deletes all predicate clauses matching the argument. In some versions of Prolog, `retractall` may fail if there is nothing to retract. To get around this, you may choose to assert a dummy clause of the right type. In SWI Prolog, however, `retractall` succeeds even for a call with no matching facts/rules.

Dynamic database manipulation via assert/retract can be used for storing computation results, such that they are not destroyed by backtracking. Therefore, if the same question is asked in the future, the answer is retrieved without having to recompute it. This technique is called memoisation, or caching, and in some applications it can greatly increase efficiency. However, it can also be used to change the behaviour of predicates at run-time (meta-programming). This generally leads to dirty, difficult to understand code. In the presence of heavy backtracking, it gets even worse. Therefore, this non-declarative feature of Prolog should be used with caution.

An example of memoisation with side effects is the following predicate which computes the nth number in the *fibonacci sequence* (you have already seen the less efficient version in the second lab session):

```
:-dynamic memo_fib/2

fib(N,F):-memo_fib(N,F),!.
fib(N,F):- N>1,
```

[1] http://www.swi-prolog.org/pldoc/doc_for?object=retract/1

```
        N1 is N-1,
        N2 is N-2,
        fib(N1,F1),
        fib(N2,F2),
        F is F1+F2,
        assertz(memo_fib(N,F)).
fib(0,1).
fib(1,1).
```

Exercise 10.6: Consult the predicate specification above, and run the following queries, sequentially:

```
?- listing(memo_fib/2).
?- fib(4,F).
?-listing(memo_fib/2).
?-fib(10,F).
?-listing(memo_fib/2).
?-fib(10,F).
```

What do you notice?

Failure-driven loops

Whenever you want to collect all the answers that you have stored on your predicate base via `assert` statements, you can use failure driven loops: *force Prolog to backtrack* until there are no more possibilities left. The pattern for a failure driven loop which reads all stored clauses - say for predicate `memo_fib/2` above - and prints all the *fibonacci numbers* already computed:

```
print_all:-memo_fib(N,F),
        write(N),
        write(' - '),
        write(F),
        nl,
        fail.
print_all.
```

Exercise 10.7: Study the execution of the following queries:

```
?-print_all.
?-retractall(memo_fib(_,_)).
?-print_all.
```

Question 10.1: Can you collect all the values in a list, instead of writing them on the screen? How? Can you do that without modifying the predicate base? (After you answer these questions, search for `findall/3` *in the SWI manual).*

Exercise 10.7: What are the answers you should get on the following queries (try to solve them on paper first, then check the answers in the Prolog engine):

```
?- findall(X,append(X,_,[1,2,3,4]),List).
?- findall(lists(X,Y), append(X,Y,[1,2,3,4]), List).
?-findall(X, member(X,[1,2,3]),List).
```

Let's now see an example of using *side effects* to get all the possible answers to a query: let's write a predicate which computes all the permutations of a list, and returns them in a separate list. We want the query on the predicate to behave like this:

```
?- all_perm([1,2,3],L).
L=[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]];
no.
```

Assuming that you already know how to implement the perm/2 predicate - the predicate which generates a permutation of the input list (see the document on *Sorting Methods* if not) - the all_perm/2 predicate specification is:

```
all_perm(L,_):-perm(L,L1),
                assertz(p(L1)),
                fail.
all_perm(_,R):-collect_perms(R).

collect_perms([L1|R]):-retract(p(L1)),
                       !,
                       collect_perms(R).
collect_perms([]).
```

Exercise 10.8: Study the execution of the following queries:

```
?-retractall(p(_)), all_perm([1,2],R).
?-listing(p/1)).
?-retractall(p(_)),all_perm([1,2,3],R).
```

Questions:

10.2: Why do I need a retractall call before calling all_perm/2?

10.3: Why do I need a ! after the retract call in the first clause of collect_perms/1*?*

10.4: What kind of recursion is used on collect_perms/1*? Can you do the collect using the other type of recursion? Which is the order of the permutations in that case?*

10.5: Does collect_perms/1 *destroy the results stored on the predicate base, or does it only read them?*

10.4 Quiz exercises

10.4.1 Write a predicate which transforms an incomplete list into a difference list (and one which makes the opposite transformation).

10.4.2. Write a predicate which transforms a complete list into a difference list (and one which makes the opposite transformation).

10.4.3. Write a predicate which generates a list with all the possible decompositions of a

list into 2 lists, without using findall. Example query:

```
?- all_decompositions([1,2,3], List).
List = [ [[], [1,2,3]], [[1], [2,3]], [[1,2], [3]], [[1,2,3], []] ];
no.
```

10.5 Problems

10.5.1 Write a predicate which flattens a deep list using difference lists instead of append.

10.5.2 Write a predicate which collects all even keys in a binary tree, using difference lists.

10.5.3. Write a predicate which collects, from a binary **incomplete** search tree, all keys between K1 and K2, using difference lists.