CT331
Programming Paradigms

Week 11 Lecture 1

Prolog: Further Examples and Tail Recursion

Deletion

- Representation: del(X, L, L1)
  - Delete X from list L resulting in L1

- For example, if del is suitably defined:
  - `?- del(a, [c, d, a, f], R).`

Deletion Steps

- Base Case:
- If X is head of L then result of deleting X is the tail of L

- Reduce:
- add head of L to Res and delete X from tail of L

Deletion Steps

- delete_one(Term, [Term | Tail], Tail).
- delete_one(Term, [Head | Tail], [Head | Result]):-
  - delete_one(Term, Tail, Result).

Question: What happens if the element is not in the list?

- How can this be fixed?

- Add extra clause at start:
- delete_one(_, [ ], [ ]).

Question: Will this delete multiple occurrences of X?

No, stops when/if match found

To delete multiple occurrences:

- Base Cases:
  - If L is empty list then result is [ ]
- Reduce:
  - If X is head of L then delete X from tail of L.
  - If X is not head of L, add head of L to Res and delete X from tail of L

Question: Will this delete multiple occurrences of X?

```
delall( _, [ ], [ ] ).

delall(Term, [Term | Tail], Res) :-
    delall(Term, Tail, Res).

delall(Term, [Head | Tail] , [Head | Res]):-
    delall(Term,Tail,Res).
```

Question: More deletion … Remove Duplicates from a List

- Representation: `deldups(L, Res)`
- Delete duplicate occurrences of all elements from list L resulting in list `Res`
- E.g. if `deldups` suitable defined:

```
?- deldups([a, b, a, c, d, c], Res).
Res = [b, a, c, d]
```

Steps to remove duplicates from a list

- Base Case:
  - If L is the empty list the result is the empty list.
- Reduction:
  - If the first element in the list **is** a member of the tail of the list, remove it and check the tail of the list for more duplicates.
  - Otherwise, add it to the result and check the tail of the list.

Deleting Duplicates in a List

```
deldups([ ], [ ]).

deldups([H|T], Res1):-
    membr(H, T), deldups(T, Res1).

deldups([H|T], [H|Res1]) :-
    deldups(T, Res1).
```

Concatenation of Lists

- **Representation:** conc(L1, L2, L3)
  - L1 and L2 are two lists; L3 is their concatenation
- For example, if conc is suitably defined:
  - `?- conc([a, b], [c, d], Res).`

    `Res = [a, b, c, d]`
  - `?- conc([a, b], [c, d], [a, b, a, c, d]).`

    `What is the result?`
- So, general rule?

Concatenation/Merging of Two Lists

- Base Case:
  - If L1 is empty the result of merging L1 and L2 is?
- Reduce (recursive step):
  - keep adding head of L1 to L3 until L1 is empty (i.e. we reach the base case).

```
conc([ ], L, L).
conc([X|L1], L2, [X|L3]):-
        conc(L1, L2, L3).
```

Tail Recursion

- Recursive calls normally take up memory space which is only freed after the return from the call.
- In special cases, it is possible to execute nested recursive calls without requiring extra memory
- In such a case a recursive procedure has a special form called *tail recursion*.

- A tail recursive procedure only has **one** recursive call and this call appears as:
  - The *last goal* of the *last clause* in the procedure

Tail Recursion

- The goals preceding the recursive call must be deterministic so that no backtracking occurs after the last call
- In the case of tail recursive procedures, no information is needed upon the return from a call
- Such recursion can be carried out simply as iteration in which a next cycle in the loop does not require additional memory
- When memory efficiency is critical, tail recursive formulations of procedures help

Reverse Items in a List

- Reverse items in a list (top-level) such that:

- ?- reverselist([a, b, c], R).

  - `R = [c, b, a]`

Steps: reverselist([a,b,c], R).

- Base case: If list is empty, result is empty list:
  - reverse([ ], [ ]).
- Reduce:
  - Reduce to empty list, by reversing tail of list
    - reverse(T,L)
  - Add head of list to a new list using merge (conc) already defined
    - conc(L, [H], R).

Using previously defined conc

- reverselist([ ],[ ]).

- reverselist([H|T], R) :-

  reverselist(T, L), conc(L, [H], R).

Try writing tail recursive version `reversetr(L,R)`

- Use temporary list (`Temp`) to add each successive head value of `L` to.
- Use helper function to call `reverse2` with empty list as current value of the temporary list
- Base Case:
  - L is empty, return R
- Reduce:
  - Add Head to temporary list for each call of reverse2

Tail recursive version of reverselist

```
reversetr(L, R):-
  reverse2(L, [], R).

reverse2([ ],R,R).

reverse2([H|T], Temp, R) :-
   reverse2(T, [H|Temp], R).
```