IKI30820 – Logic Programming
Programming Style and Techniques
Slide 09

Ari Saptawijaya (slides) Adila A. Krisnadhi (LaTeX adaptation)

Fakultas Ilmu Komputer
Universitas Indonesia

2009/2010 • Semester Gasal

Outline

1. Use of recursion
2. Improving efficiency
   - By difference list
   - By tail recursion

Judging how good your program

- Generally, the accepted criteria for judging how good a program is:
  1. Correctness
  2. User-friendliness
  3. Efficiency
  4. Readability
  5. Modifiability
  6. Robustness
  7. Documentation
- First **think** about the problem to be solved, then write the actual code.

Principle of stepwise refinement

- The Principle of Stepwise Refinement:
  the final program is developed through a sequence of transformations ("refinements") of the solution, from the top-level solution to the bottom-level solution.
- In the case of Logic Programming, we deal with the stepwise refinement of relations, in the form of predicates.

Use of recursion

Split the problem into cases belonging to two groups:

1. trivial (”boundary”) cases;
2. general cases where the solution is constructed from solutions of simpler versions of the original problem itself

One reason why recursion so naturally applies to defining relations in Logic Programming is that data objects themselves often have recursive structures; for example: lists and trees.

Example

Write a predicate for processing a list of items so that each item is transformed by the same transformation rule:

```
maplist(List, F, NewList)
```

where `List` is an original list, `F` is a transformation rule (a binary relation) and `NewList` is the list of all transformed items.

Example

```
cube(A,B) :- B is A*A*A.

?- maplist([1,2,3], cube, List).
List = [1, 8, 27]
Yes
```

Example

The problem can be split into two cases:

1. boundary case: `List = []`
   if `List = []` then `NewList = []`, regardless of `F`
2. general case: `List = [X|Tail]`
   transform the item `X` by rule `F` obtaining `NewX`, and transform the list `Tail` obtaining `NewTail`; the whole transformed list is `[NewX | NewTail]`.

Example

```
maplist([], _, []).
maplist([X|Tail], F, [NewX|NewTail]) :-
 G =..[F, X, NewX],
 call(G),
 maplist(Tail, F, NewTail).
```

Outline

1. Use of recursion

2. Improving efficiency
   - By difference list
   - By tail recursion

Improving efficiency of programs

- Ideas for improving the efficiency of a program usually come from a deeper understanding of the problem.
- A more efficient algorithm can, in general, be obtained from improvements of two kinds:
  - Improving search efficiency by avoiding unnecessary backtracking and stopping the execution of useless alternatives as soon as possible.
  - Using more suitable data structures to represent objects in the program, so that operations on objects can be implemented more efficiently.

Improving efficiency by difference list

- A list can be represented as the difference between a pair of lists.
- For example, the list `[a, b, c]` can be represented by each of the following lists:

```
[a,b,c,d,e]-[d,e]
[a,b,c] - []
[a,b,c|T] - T
```

Consider the `append/3` predicate

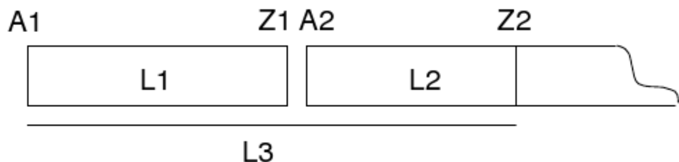
- The concatenation of lists has been programmed as:

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

- The program scans all of the first list, until the empty list is encountered.

Avoiding inefficiency of `append/3` by difference list

- This scanning can be avoided by using difference lists.




```
append2(A1-Z1, Z1-Z2, A1-Z2).

?- append2([a,b,c|T1]-T1, [d,e|T2]-T2, L).
T1 = [d,e|T2]
L = [a,b,c,d,e|T2]-T2

?- append2([a,b,c|T]-T, [d,e]-[], L-[]).
T = [d,e]
L = [a,b,c,d,e]
```

Improving efficiency by tail recursion

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

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.
  - If the goals preceding the recursive call are deterministic, then no backtracking occurs after this last call.
- Such recursion can be carried out simply as iteration.
- A Prolog system will typically notice such an opportunity of saving memory and realize tail recursion as iteration.
  - This is called **tail recursion optimization**.

Example

Let us consider the predicate for computing the sum of a list of numbers: `sumlist(List, Sum).`

```
?- sumlist([1,2,3,4], Sum).
Sum = 10
```

```
% This version is not tail recursive
sumlist([], 0).
sumlist([H|T], Sum) :-
  sumlist(T, Sum1),
  Sum is H + Sum1.
```

```
% This version is tail recursive.
% Use an accumulator.
sumlist(List, Sum) :- sumlist(List, 0, Sum).

% sumlist(List, PartialSum, TotalSum).
sumlist([], TotalSum, TotalSum).
sumlist([H|T], PartialSum, TotalSum) :-
  NewPartialSum is PartialSum + H,
  sumlist(T, NewPartialSum, TotalSum).
```

Exercise

Define tail recursive version of **length**/2

```
length([],0).
length([_|Xs],L) :-
  length(Xs,L1), L is L+1.
```

Define tail recursive version of `reverse/2`.

```
reverse([],[]).
reverse([X|Xs],Ys) :-
  reverse(Xs,Zs), append(Zs,[X],Ys).
```

Example: Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

```
% This version is not tail recursive.
% fib(N, F) is true if F is
% the N-th Fibonacci number.
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1,
    N1 is N-1, fib(N1, F1),
    N2 is N-2, fib(N2, F2),
    F is F1 + F2.
```

Fibonacci: using caching

```
fib2(0, 0).
fib2(1, 1).

fib2(N, F) :-
   N > 1,
   N1 is N-1, fib2(N1, F1),
   N2 is N-2, fib2(N2, F2),
   F is F1 + F2,
   asserta(fib2(N,F)).
```

Fibonacci: tail recursive

```
% This version is tail recursive. Use accumulators.

% The first two Fib numbers are 1.
fib(N, F) :- fibtr(1, N, 0, 1, F).

% fibtr(M, N, F1, F2, F) succeeds if F1 and F2 are
% the (M-1)st and Mth Fibonacci numbers, and F is
% the Nth Fibonacci number.
fibtr(M, N, F1, F2, F2) :-
    M >= N. % Nth Fibonacci number has been reached.
fibtr(M, N, F1, F2, F) :-
    M < N,
    NextM is M+1, NextF2 is F1 + F2,
    fibtr(NextM, N, F2, NextF2, F).
```