Tail recursion: some motivating examples

Marek Sergot
Department of Computing
Imperial College, London

Autumn 2012

Here are a couple of little example programs to demonstrate the advantages of tail recursion.

Tail recursion is a feature of all recursive programming languages, not just Prolog. In the case of Prolog, very roughly: "tail recursion is recursion in which the recursive call is the last subgoal of the last clause." (Though that is not quite right.) You can see the effect by tracing the computation of some typical tail recursive and non-tail recursive programs. Here are a couple of such examples.

I did not want to go into too much detail, for lack of time, but there is a a little more to it. If there are no choice points (backtracking points) left when the recursive call is made, that is, when the computation is essentially deterministic, then the Prolog compiler can optimise the code very efficiently. (See 'Last Call Optimization' in the 'Programming Tips and Examples' section of the Sicstus Prolog manual.)

Note (1): "The memory saved by tail recursion can be very important when a recursive loop has to go through thousands or millions of cycles. When the recursion depth is measured in dozens or less, the memory saved by tail recursion is not important."

Note (2): Not all computations can easily be made tail recursive. Even if they can, the resulting code can often become obscure and hard to read, and therefore also difficult to maintain.

So: don't be afraid of writing non-tail recursive programs if you need them or if they are more natural. With experience you will find you can write tail-recursive programs more or less automatically without thinking about it, for simple common cases at least.

(The quotes above are from a recent document *Coding guidelines for Prolog* by Covington, Bagnar, O'Keefe, Wielemaker, and Price. It is available at `http://arxiv.org/abs/0911.2899`. I don't suggest reading it unless you are reasonably experienced with Prolog already.)

1 Example 1

`factorial(N, X)` – `X` is the factorial of integer `N`.

(The program assumes `N` is given, and is an integer.)

```
factorial(0,1).
factorial(N, FactN) :-
   N > 0,
   M is N - 1,
   factorial(M, FactM),
   FactN is N*FactM.
```

Trace the computation:

```
?-  factorial(20, X).
|
?-  factorial(19, X1), X is 20*X1.
|
?-  factorial(18, X2), X1 is 19*X2, X is 20*X1.
|
?-  factorial(17, X3), X2 is 18*X3, X1 is 19*X2, X is 20*X1.
      .
      .
      .
```

To get a tail-recursive version, introduce an extra argument to act as a kind of 'accumulator' to record the partial answer computed so far. At the end of the recursion (the base case) the final answer will be the value of this accumulator.

```
tr_factorial(0, F, F).
tr_factorial(N, Acc, FactN) :-
   N > 0,
   M is N - 1,
   NewAcc is Acc*N,    % no choice points here
   tr_factorial(M, NewAcc, FactN).

tr_factorial(N, FactN) :-
   integer(N), N >= 0,     % we might as well include this test
   tr_factorial(N, 1, FactN).
```

Trace the computation!

`sumlist(List, Total)` in section 5 of the Introduction to Prolog notes is similar.

Example 2

The following program for 'naive reverse' is in the Introduction to Prolog notes.

```
naive_reverse([], []).
naive_reverse([U|X], Res) :-
  naive_reverse(X, Y),
  append(Y, [U], Res).
```

Try it out using the following:

```
tnr(N, M) :-
  length(X, N),
  naive_reverse(X, Rev),
  length(Rev, M).
```

`append/3` and `length/2` are built-in programs in Sicstus version 4. `length/2` finds the length of a given list, or, given an integer N, constructs a list of length N. I use this to avoid having to type long lists when testing the program. The second argument of `tnr/2` and the second call to `length/2` don't really do anything. I call `tnr(N,M)` with `N` given and `M` a variable.

Tail-recursive reverse:

```
tail_reverse([], X, X).
tail_reverse([U|X], Y, Result) :-
  tail_reverse(X, [U|Y], Result).

tail_reverse(X, Result) :- tail_reverse(X, [], Result).


ttr(N, M) :-
  length(X, N),
  tail_reverse(X, Rev),
  length(Rev, M).
```

Here, the extra (middle) argument of `tail_reverse/3` again acts as a kind of 'accumulator'.

Try out the two versions on some largish examples.

```
?- tnr(N, M).
```

with increasing values of `N`: `N = 100, 1000, 10000, ....`

You will find that `tnr` (naive reverse) gets pretty slow around `N = 100000`. It depends on your computer of course. Now try the tail recursive version:

```
?- ttr(N, M).
```

You might find it hard to believe.