02157 Functional Programming

Lecture 8: Tail-recursive (iterative) functions (I)

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Memory management: the stack and the heap
- Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, for example, in order
  - to avoid evaluations with a huge amount of pending operations, e.g.

$$7+(6+(5\cdots+f\ 2\cdots))$$

  - to avoid inadequate use of `@` in recursive declarations.
- Iterative functions with accumulating parameters correspond to while-loops

An example: Factorial function (I)

Consider the following declaration:

```
let rec fact =
   function
   | 0 -> 1
   | n -> n * fact(n-1);;
val fact : int -> int
```

- What resources are needed to compute `fact`(N)?

Considerations:

- Computation time: number of individual computation steps.
- Space: the maximal memory needed during the computation to represent expressions and bindings.

An example: Factorial function (II)

Evaluation:

$$
\begin{array}{ll}
 & \texttt{fact}(N) \\
\rightsquigarrow & (\texttt{n * fact(n-1)}\ ,\ [\texttt{n} \mapsto N]) \\
\rightsquigarrow & N * \texttt{fact}(N-1) \\
\rightsquigarrow & N * (\texttt{n * fact(n-1)}\ ,\ [\texttt{n} \mapsto N-1]) \\
\rightsquigarrow & N * ((N-1) * \texttt{fact}(N-2)) \\
\vdots & \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * (2 * 1))) \cdots))) \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * 2)) \cdots))) \\
\vdots & \\
\rightsquigarrow & N!
\end{array}
$$

Time and space demands: proportional to N Is this satisfactory?

Another example: Naive reversal (I)

```
let rec naiveRev =
   function
   | []    -> []
   | x::xs -> naiveRev xs @ [x];;
val naiveRev : 'a list -> 'a list
```

Evaluation of `naiveRev` $[x_1, x_2, \ldots, x_n]$:

$$
\begin{array}{ll}
 & \texttt{naiveRev}\,[x_1, x_2, \ldots, x_n] \\
\rightsquigarrow & \texttt{naiveRev}\,[x_2, \ldots, x_n]\,@\,[x_1] \\
\rightsquigarrow & (\texttt{naiveRev}\,[x_3, \ldots, x_n]\,@\,[x_2])\,@\,[x_1] \\
\vdots & \\
\rightsquigarrow & ((\cdots(([\,]\,@\,[x_n])\,@\,[x_{n-1}])\,@\cdots@\,[x_2])\,@\,[x_1])
\end{array}
$$

Space demands: proportional to n — satisfactory

Time demands: proportional to n^2 — not satisfactory

Examples: Accumulating parameters

Efficient solutions are obtained by using *more general functions*:

$$\begin{array}{lcl} \texttt{factA}(n, m) & = & n! \cdot m, \text{ for } n \geq 0 \\ \texttt{revA}(\texttt{[}x_1, \ldots, x_n\texttt{]}, ys) & = & \texttt{[}x_n, \ldots, x_1\texttt{]} \texttt{ @ } ys \end{array}$$

We have:

$$\begin{array}{lcl} n! & = & \texttt{factA}(n, 1) \\ \texttt{rev [}x_1, \ldots, x_n\texttt{]} & = & \texttt{revA}(\texttt{[}x_1, \ldots, x_n\texttt{]}, \texttt{[ ]}) \end{array}$$

m and *ys* are called *accumulating parameters*. They are used to hold the temporary result during the evaluation.

Declaration of `factA`

Property: $\texttt{factA}(n, m) = n! \cdot m$, for $n \geq 0$

```
let rec factA =
   function
   | (0,m) -> m
   | (n,m) -> factA(n-1,n*m) ;;
```

An evaluation:

$$
\begin{array}{ll}
 & \texttt{factA(5,1)} \\
\rightsquigarrow & (\texttt{factA(n-1,n*m)},\ [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow & \texttt{factA(4,5)} \\
\rightsquigarrow & (\texttt{factA(n-1,n*m)},\ [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow & \texttt{factA(3,20)} \\
\rightsquigarrow & \ldots \\
\rightsquigarrow & \texttt{factA(0,120)} \rightsquigarrow (\texttt{m},\ [\texttt{m} \mapsto 120]) \rightsquigarrow 120
\end{array}
$$

Space demand: constant.

Time demands: proportional to n

Declaration of `revA`

Property: $\texttt{revA}(\texttt{[}x_1,\ldots,x_n\texttt{]},ys) = \texttt{[}x_n,\ldots,x_1\texttt{]}\ \texttt{@}\,ys$

```
let rec revA =
   function
   | ([], ys)     -> ys
   | (x::xs, ys) -> revA(xs, x::ys) ;;
```

An evaluation:

```
      revA([1,2,3],[])
⇝    revA([2,3],1::[])
⇝    revA([3],2::[1])
⇝    revA([3],[2,1])
⇝    revA([],3::[2,1])
⇝    revA([],[3,2,1])
⇝    [3,2,1]
```

Space and time demands:

proportional to n (the length of the first list)

Iterative (tail-recursive) functions (I)

The declarations of `factA` and `revA` are *tail-recursive functions*

- the recursive call is the *last function application* to be evaluated in the body of the declaration e.g. $\mathit{facA}(3, 20)$ and $\mathit{revA}([3], [2, 1])$
- only *one set* of bindings for argument identifiers is needed during the evaluation

Example

```
let rec factA =
    function
    | (0,m) -> m
    | (n,m) -> factA(n-1,n*m)
               (* recursive "tail-call" *)
```

- only one set of bindings for argument identifiers is needed during the evaluation

$$
\begin{array}{ll}
 & \texttt{factA(5,1)} \\
\rightsquigarrow & (\texttt{factA(n,m)},\ [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow & (\texttt{factA(n-1,n*m)},\ [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow & \texttt{factA(4,5)} \\
\rightsquigarrow & (\texttt{factA(n,m)},\ [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow & (\texttt{factA(n-1,n*m)},\ [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow & \ldots \\
\rightsquigarrow & \texttt{factA(0,120)} \rightsquigarrow (\texttt{m},\ [\texttt{m} \mapsto 120]) \rightsquigarrow 120
\end{array}
$$

Concrete resource measurements: factorial functions

```
let xs16 = List.init 1000000 (fun i -> 16);;
val xs16 : int list = [16; 16; 16; 16; 16; ...]

#time;; // a toggle in the interactive environment

for i in xs16 do let _ = fact i in ();;
Real: 00:00:00.051, CPU: 00:00:00.046, ...

for i in xs16 do let _ = factA(i,1) in ();;
Real: 00:00:00.024, CPU: 00:00:00.031, ...
```

The performance gain of `factA` is much better than the indicated factor 2 because the `for` construct alone uses about 12 ms:

```
for i in xs16 do let _ = () in ();;
Real: 00:00:00.012, CPU: 00:00:00.015, ...
```

Real: time elapsed by the execution.   CPU: time spent by all cores.

Concrete resource measurements: reverse functions

```
let xs20000 = [1 .. 20000];;

naiveRev xs20000;;
Real: 00:00:07.624, CPU: 00:00:07.597,
GC gen0: 825, gen1: 253, gen2: 0
val it : int list = [20000; 19999; 19998; ...]

revA(xs20000,[]);;
Real: 00:00:00.001, CPU: 00:00:00.000,
GC gen0: 0, gen1: 0, gen2: 0
val it : int list = [20000; 19999; 19998; ...]
```

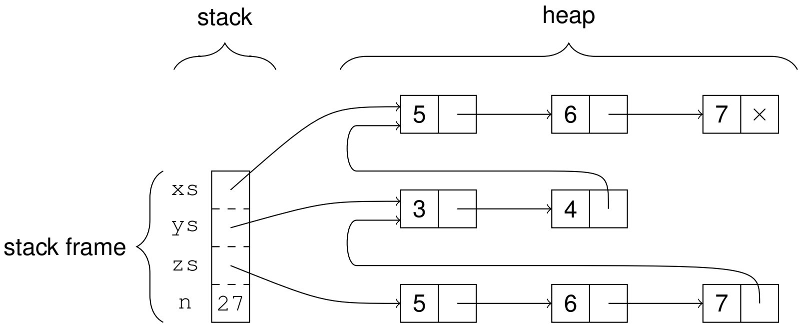
- The naive version takes 7.624 seconds - the iterative just 1 ms.
- The use of append (`@`) has been reduced to a use of cons (`::`). This has a dramatic effect of the garbage collection:
  - No object is reclaimed when `revA` is used
  - 825+253 obsolete objects were reclaimed using the naive version

Let's look at memory management

Memory management: stack and heap

- Primitive values are allocated on the stack
- Composite values are allocated on the heap

```
let xs = [5;6;7];;
let ys = 3::4::xs;;
let zs = xs @ ys;;
let n = 27;;
```



Observations

No unnecessary copying is done:

1. The linked lists for *ys* is not copied when building a linked list for $y :: ys$.
2. Fresh cons cells are made for the elements of *xs* only when building a linked list for *xs* `@` *ys*.

since a list is a functional (immutable) data structure

The running time of `@` is linear in the length of its first argument.

Operations on stack and heap

Example:

```
let zs = let xs = [1;2]
         let ys = [3;4]
         xs@ys;;
```

Initial stack and heap prior to the evaluation of the local declarations:

stack heap

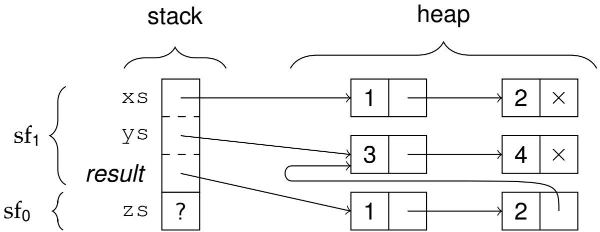
$\mathrm{sf}_0 \left\{ \texttt{zs} \ \boxed{?} \right.$

Operations on stack: Push

Example:

```
let zs = let xs = [1;2]
         let ys = [3;4]
         xs@ys;;
```

Evaluation of the local declarations initiated by pushing a new stack frame onto the stack:



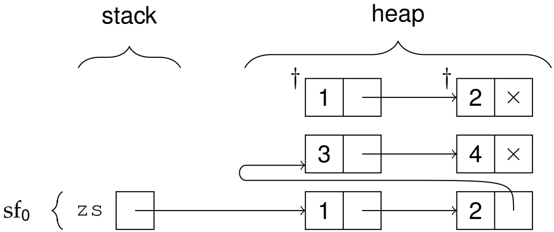
The auxiliary entry result refers to the value of the `let`-expression.

Operations on stack: Pop

Example:

```
let zs = let xs = [1;2]
         let ys = [3;4]
         xs@ys;;
```

The top stack frame is popped from the stack when the evaluation of the `let`-expression is completed:



The resulting heap contains two obsolete cells marked with '†'

Operations on the heap: Garbage collection

The memory management system uses a *garbage collector* to reclaim obsolete cells in the heap behind the scene.

The garbage collector manages the heap as partitioned into three groups or *generations*: `gen0`, `gen1` and `gen2`, according to their age. The objects in `gen0` are the youngest while the objects in `gen2` are the oldest.

The typical situation is that objects die young and the garbage collector is designed for that situation.

Example:

```
naiveRev xs20000;;
Real: 00:00:07.624, CPU: 00:00:07.597,
GC gen0: 825, gen1: 253, gen2: 0
val it : int list = [20000; 19999; 19998; ...]
```

The limits of the stack and the heap

The stack is big:

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
bigList 120000;;
val it : int list = [1; 1; 1; 1; 1; 1; 1; 1;...]
bigList 130000;;
Process is terminated due to StackOverflowException.
```

More than $1.2 \cdot 10^5$ stack frames are pushed in recursive calls.

The heap is much bigger:

```
let rec bigListA n xs = if n=0 then xs
                        else bigListA (n-1) (1::xs);;
let xsVeryBig = bigListA 12000000 [];;
val xsVeryBig : int list = [1; 1; 1; 1; 1; 1;...]
let xsTooBig = bigListA 13000000 [];;
System.OutOfMemoryException: ...
```

A list with more than $1.2 \cdot 10^7$ elements can be created.

The iterative `bigListA` function does not exhaust the stack. WHY?

Iterative (tail-recursive) functions (II)

Tail-recursive functions are also called *iterative functions*.

- The function $f(n, m) = (n - 1, n * m)$ is iterated during evaluations for `factA`.
- The function $g(x :: xs, ys) = (xs, x :: ys)$ is iterated during evaluations for `revA`.

The correspondence between tail-recursive functions and while loops is established in the textbook.

An example:

```
let factW n =
    let ni = ref n
    let r  = ref 1
    while !ni>0 do
          r := !r * !ni ; ni := !ni-1
    !r;;
```

Iteration vs While loops

Iterative functions are executed efficiently:

```
#time;;

for i in 1 .. 1000000 do let _ = factA(16,1) in ();;
Real: 00:00:00.024, CPU: 00:00:00.031,
GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()

for i in 1 .. 1000000 do let _ = factW 16 in ();;
Real: 00:00:00.048, CPU: 00:00:00.046,
GC gen0: 9, gen1: 0, gen2: 0
val it : unit = ()
```

- the tail-recursive function actually is faster than the imperative while-loop based version

Iterative functions (III)

A function $g : \tau \texttt{ -> } \tau'$ is an *iteration of* $f : \tau \texttt{ -> } \tau$ if it is an instance of:

let rec g z = if p z then g(f z) else h z

for suitable predicate $p : \tau \texttt{ -> } \texttt{bool}$ and function $h : \tau \texttt{ -> } \tau'$.

The function g is called an *iterative (or tail-recursive) function*.

Examples: `factA` and `revA` are easily declared in the above form:

```
let rec factA(n,m) =
   if n<>0 then factA(n-1,n*m) else m;;
```

```
let rec revA(xs,ys) =
     if not (List.isEmpty xs)
     then revA(List.tail xs, (List.head xs)::ys)
     else ys;;
```

Iterative functions: evaluations (I)

Consider: `let rec` g `z = if` p `z then` g`(`f `z) else` h `z`

Evaluation of the $g\ v$:

$$
\begin{array}{ll}
 & g\ v \\
\rightsquigarrow & (\texttt{if}\ p\ \texttt{z}\ \texttt{then}\ g(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}\ ,\ [\texttt{z} \mapsto v]) \\
\rightsquigarrow & (g(f\ \texttt{z}), [\texttt{z} \mapsto v]) \\
\rightsquigarrow & g(f^1 v) \\
\rightsquigarrow & (\texttt{if}\ p\ \texttt{z}\ \texttt{then}\ g(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}\ ,\ [\texttt{z} \mapsto f^1 v]) \\
\rightsquigarrow & (g(f\ \texttt{z}), [\texttt{z} \mapsto f^1 v]) \\
\rightsquigarrow & g(f^2 v) \\
\rightsquigarrow & \ldots \\
\rightsquigarrow & (\texttt{if}\ p\ \texttt{z}\ \texttt{then}\ g(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}\ ,\ [\texttt{z} \mapsto f^n v]) \\
\rightsquigarrow & (h\ \texttt{z}, [\texttt{z} \mapsto f^n v]) \qquad \text{suppose } p(f^n v) \rightsquigarrow \mathrm{false} \\
\rightsquigarrow & h(f^n v)
\end{array}
$$

Iterative functions: evaluations (II)

Observe two desirable properties:

- there are *n* recursive calls of *g*,
- at most *one binding* for the argument pattern `z` is 'active' at any stage in the evaluation, and
- the iterative functions require one stack frame only.

Summary

- Memory management: the stack and the heap
- Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, for example, in order
  - to avoid evaluations with a huge amount of pending operations, e.g.

    $$7+(6+(5\cdots+f\ 2\cdots))$$

  - to avoid inadequate use of `@` in recursive declarations.
- Iterative functions with accumulating parameters correspond to while-loops