Overview

- Memory management: the stack and the heap
- Iterative (tail-recursive) functions is a simple technique to deal with efficiency in certain situations, e.g.
  - 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
- The notion: continuations, provides a general applicable approach

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:

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

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` $[x_1, x_2, \ldots, x_n]$:

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

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*:

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

We have:

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

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

Declaration of `factA`

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

An evaluation:

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

Space demand: constant.

Time demands: proportional to *n*

Declaration of `revA`

```
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. $itfac(3, 20)$ and $revA([3], [2, 1])$
- only *one set* of bindings for argument identifiers is needed during the evaluation

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