02157 Functional Programming

Lecture : Tail-recursive functions

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

Overview

Part I:

- Memory management: the stack and the heap

Overview

Part I:

- 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.

Overview

Part I:

- 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

Overview

Part I:

- 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

Part II: Continuation-based tail recursion

Three factorial functions

```
// using "plain" recursion
let rec fact = function
              | 0 -> 1
              | n -> n * fact(n-1)

// using "tail" recursion: "fact n = factA(n,1)"
let rec factA(n,m) = match n with
                    | 0 -> m
                    | _ -> factA(n-1,n*m)

// using imperative features (while, assignment)
let factW n = let ni = ref n
              let r  = ref 1
              while ni.Value>0 do
                 r.Value <- r.Value * ni.Value ;
                 ni.Value <- ni.Value-1
              r.Value
```

Three factorial functions: Benchmarks

- 13th Gen Intel(R) Core(TM) i7-1365U 1.80 GHz
  x64-based processor, Windows
- Native code is generated using CLI:
  dotnet publish -r win-x64 -c Release

Extract from benchmark report:

```
1.000000 computations of fact 16 are repeated 100 times.
          Mean = 15,13 ms.  ...

1.000000 computations of factA(16,1) are repeated 100 times
          Mean = 6,89 ms.  ...

1.000000 computations of factW 16 are repeated 100 times.
          Mean = 26,63 ms.  ...
```

Three factorial functions: Benchmarks

- 13th Gen Intel(R) Core(TM) i7-1365U 1.80 GHz
  x64-based processor, Windows
- Native code is generated using CLI:
  dotnet publish -r win-x64 -c Release

Extract from benchmark report:

```
1.000000 computations of fact 16 are repeated 100 times.
          Mean = 15,13 ms.  ...

1.000000 computations of factA(16,1) are repeated 100 times
          Mean = 6,89 ms.  ...

1.000000 computations of factW 16 are repeated 100 times.
          Mean = 26,63 ms.  ...
```

Program is uploaded to Learn

Two reversal functions with benchmarks

```
let rec naiveRev = function
                   | []    -> []
                   | x::xs -> naiveRev xs @ [x]

// "tail-recursive" function: revA(xs,[]) = naiveRev xs
let rec revA = function
               | ([], ys)    -> ys
               | (x::xs, ys) -> revA(xs, x::ys)
```

Extract from benchmark report:

```
naive reverse of a list of size 5.000
are repeated 100 times.
                         Mean = 88,72 ms.  ...


tail-recursive reverse of a list of size 500.000
are repeated 100 times.
                         Mean = 24,29 ms.  ...
```

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*)?

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.

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:

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

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
```

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

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*:

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

We have:

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

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:

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

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

```
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*

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. $facA(3, 20)$ and $revA([3], [2, 1])$

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. $facA(3, 20)$ and $revA([3], [2, 1])$
- 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, ...
```

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

Timing measured on an old system

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.

Timing measured on an old system

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; ...]
```

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.

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

Measured on the computer used in 2012

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

Measured on the computer used in 2012

Let's look at memory management

Memory management: stack and heap

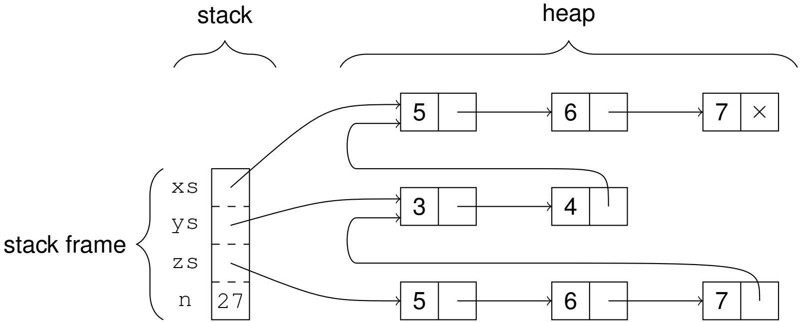
- Primitive values are allocated on the stack
- Composite values are allocated on the heap

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

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

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

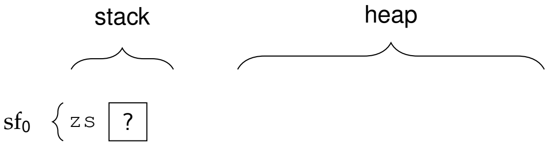
The running time of `@` is linear in the length of its first argument.

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:

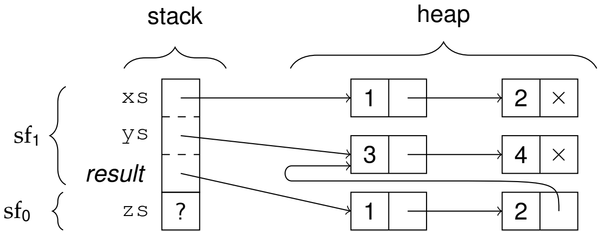


Operations on stack: Push

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:



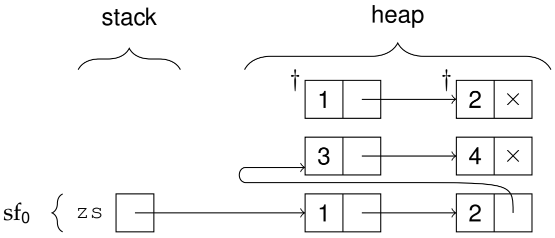
The auxiliary entry result refers to the value of the `let`-expression.

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.

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 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 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`.

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`.

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.

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 = ()
```

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

Characterizing tail-recursive functions syntactically

Tail position - briefly

Consider expressions of the form:

$\texttt{if}\ \ e_a\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2$

$e_1\ \ e_a$

$\texttt{fun}\ \ x\ \rightarrow\ e_1$

$\texttt{let}\ \ x\ =\ e_a\ \ \texttt{in}\ \ e_1$

$\texttt{match}\ \ e_a\ \ \texttt{with}\ \ |\ \ pat_1\ \rightarrow\ e_1\ \ \ldots\ \ |\ \ pat_n\ \rightarrow\ e_n$

$e_a\ +\ e_b$

primitive

Tail position - briefly

Consider expressions of the form:

$\texttt{if}\ \ e_a\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2$

$e_1\ \ e_a$

$\texttt{fun}\ \ x\ \rightarrow\ e_1$

$\texttt{let}\ \ x\ =\ e_a\ \ \texttt{in}\ \ e_1$

$\texttt{match}\ \ e_a\ \ \texttt{with}\ \ |\ \ pat_1\ \rightarrow\ e_1\ \ \ldots\ |\ \ pat_n\ \rightarrow\ e_n$

$e_a\ +\ e_b$

primitive

- $e_1, \ldots, e_n$ and *primitive* are in tail position, where *primitive* is a variable or a constant.
- e_a and e_b are not in tail position
- patterns are "binders" like $\texttt{fun}\ x \rightarrow \ldots$ and not in tail position.

Tail position - briefly

Consider expressions of the form:

$\texttt{if} \quad e_a \quad \texttt{then} \quad e_1 \quad \texttt{else} \quad e_2$

$e_1 \;\; e_a$

$\texttt{fun} \quad x \;\rightarrow\; e_1$

$\texttt{let} \quad x \;=\; e_a \quad \texttt{in} \quad e_1$

$\texttt{match} \quad e_a \quad \texttt{with} \quad | \quad pat_1 \;\rightarrow\; e_1 \quad \ldots \; | \quad pat_n \;\rightarrow\; e_n$

$e_a \;+\; e_b$

primitive

- $e_1, \ldots, e_n$ and *primitive* are in tail position, where *primitive* is a variable or a constant.
- e_a and e_b are not in tail position
- patterns are "binders" like $\texttt{fun}\ x \rightarrow \ldots$ and not in tail position.

Complete definition includes sub-expressions

Tail calls and tail recursion

- A function call in tail position is said to be a tail call

Calls to f are tail calls (calls to g, h are not) in:

```
f (3+4)

if x>0 then f(g(9)) else g 1 + h 6

fun x -> f (x+1)

match g(a+b) with
| []      -> f a
| x::xs   -> g(x)::h xs

let x = g y in f(if x=0 the g z else g 2)
```

Tail calls and tail recursion

- A function call in tail position is said to be a tail call

Calls to f are tail calls (calls to g, h are not) in:

```
f (3+4)

if x>0 then f(g(9)) else g 1 + h 6

fun x -> f (x+1)

match g(a+b) with
| []      -> f a
| x::xs   -> g(x)::h xs

let x = g y in f(if x=0 the g z else g 2)
```

- Tail calls can be implemented without adding a new stack frame

Tail calls and tail recursion

- A function call in tail position is said to be a tail call

Calls to f are tail calls (calls to g, h are not) in:

```
f (3+4)

if x>0 then f(g(9)) else g 1 + h 6

fun x -> f (x+1)

match g(a+b) with
| []     -> f a
| x::xs  -> g(x)::h xs

let x = g y in f(if x=0 the g z else g 2)
```

- Tail calls can be implemented without adding a new stack frame

- A tail recursive function is a recursive function where every recursive call is a tail call

A simple example

The function fact is not tail recursive:

```
let rec fact =
   function
   | 0 -> 1
   | n -> n * fact(n-1)
```

- The recursive call is not a tail call in `n * fact(n-1)`

A simple example

The function fact is not tail recursive:

```
let rec fact =
   function
   | 0 -> 1
   | n -> n * fact(n-1)
```

- The recursive call is not a tail call in `n * fact(n-1)`

But factA is tail recursive:

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

- The recursive call is a tail call in
  `(n,m) -> factA(n-1,n*m)`

Part 1: Summary

- Memory management: the stack and the heap

Part 1: 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.

Part 1: 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

Part 1: 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

A simple and useful technique to consider when bottlenecks are spotted

Part II
Continuation-based tail recursion

Limitation of accumulating parameters

Tail-recursive versions of recursive functions CANNOT be obtained using accumulating parameters in all cases.

Consider for example:

```
type BinTree<'a> = | Leaf
                   | Node of BinTree<'a>*'a*BinTree<'a>

let rec count = function
                | Leaf          -> 0
                | Node(tl,n,tr) -> count tl + count tr + 1
```

Limitation of accumulating parameters

Tail-recursive versions of recursive functions CANNOT be obtained using accumulating parameters in all cases.

Consider for example:

```
type BinTree<'a> = | Leaf
                   | Node of BinTree<'a>*'a*BinTree<'a>

let rec count = function
                | Leaf           -> 0
                | Node(tl,n,tr)  -> count tl + count tr + 1
```

A counting function:

```
countA: int -> BinTree<'a> -> int
```

using an accumulating parameter will not be tail-recursive due to the expression containing recursive calls on the left and right sub-trees. (Ex. 9.8)

Continuations

Continuation: A representation of the "rest" of the computation.

Continuations

Continuation: A representation of the “rest” of the computation.

Every functional program can automatically be turned into a program where every call is a tail call using continuations.

Continuations

Continuation: A representation of the “rest” of the computation.

Every functional program can automatically be turned into a program where every call is a tail call using continuations.

We take an example-based approach. Consider

```
let rec sum xs = match xs with
                 | []      -> 0
                 | x::tail -> let v = sum tail
                              x+v
```

Continuations

Continuation: A representation of the “rest” of the computation.

Every functional program can automatically be turned into a program where every call is a tail call using continuations.

We take an example-based approach. Consider

```
let rec sum xs = match xs with
                 | []       -> 0
                 | x::tail -> let v = sum tail
                              x+v
```

The continuation-based version of `sum` has a continuation

```
k: int -> int
```

as an extra argument. Determines what happens with the result.

Continuations

Continuation: A representation of the “rest” of the computation.

Every functional program can automatically be turned into a program where every call is a tail call using continuations.

We take an example-based approach. Consider

```
let rec sum xs = match xs with
                 | []       -> 0
                 | x::tail -> let v = sum tail
                              x+v
```

The continuation-based version of `sum` has a continuation

```
k: int -> int
```

as an extra argument. Determines what happens with the result.

```
let rec sumC xs  k =
   match xs with
   | []       -> k 0
   | x::tail -> sumC tail (fun v -> k(x+v))
```

Continuations

Continuation: A representation of the "rest" of the computation.

Every functional program can automatically be turned into a program where every call is a tail call using continuations.

We take an example-based approach. Consider

```
let rec sum xs = match xs with
                 | []       -> 0
                 | x::tail -> let v = sum tail
                              x+v
```

The continuation-based version of `sum` has a continuation

```
k: int -> int
```

as an extra argument. Determines what happens with the result.

```
let rec sumC xs  k =
   match xs with
   | []       -> k 0
   | x::tail -> sumC tail (fun v -> k(x+v))
```

- all calls of `sumC` and `k` are tail calls
- can be implemented without adding a new stack frame

An evaluation

```
   sumC [1;2;3] id
⇝  sumC [2;3] (fun v -> id(1+v))
⇝  sumC [3] (fun w -> (fun v -> id(1+v))(2+w))
⇝  sumC [] (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u))
⇝  (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u)) 0
⇝  (fun w -> (fun v -> id(1+v))(2+w)) 3
⇝  (fun v -> id(1+v)) 5
⇝  id 6
⇝  6
```

An evaluation

```
    sumC [1;2;3] id
⇝  sumC [2;3] (fun v -> id(1+v))
⇝  sumC [3] (fun w -> (fun v -> id(1+v))(2+w))
⇝  sumC [] (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u))
⇝  (fun u -> (fun w -> (fun v -> id(1+v))(2+w))(3+u)) 0
⇝  (fun w -> (fun v -> id(1+v))(2+w)) 3
⇝  (fun v -> id(1+v)) 5
⇝  id 6
⇝  6
```

Notice:

- Closures are allocated in the heap.
- Just one stack frame is needed due to tail calls.
- Stack is traded for heap.

A more efficient way

Consider the following version using an accumulating parameter:

```
let rec sumA xs n = match xs with
                    | []      -> n
                    | x::tail -> sumA tail (x+n)
```

- uses constant space and is much more efficient than

```
let rec sumC xs k = match xs with
                    | []     -> k 0
                    | x::tail-> sumC tail (fun v -> k(x+v))
```

Some relationships between `sum`, `sumA` and `sumC`:

A more efficient way

Consider the following version using an accumulating parameter:

```
let rec sumA xs n = match xs with
                    | []      -> n
                    | x::tail -> sumA tail (x+n)
```

- uses constant space and is much more efficient than

```
let rec sumC xs k = match xs with
                    | []     -> k 0
                    | x::tail-> sumC tail (fun v -> k(x+v))
```

Some relationships between `sum`, `sumA` and `sumC`:

1. $\texttt{sum}\ \mathit{xs} = \texttt{sumC}\ \mathit{xs}\ \mathit{id}$
2. $\texttt{sumA}\ \mathit{xs}\ 0 = \texttt{sumC}\ \mathit{xs}\ \texttt{id}$

A more efficient way

Consider the following version using an accumulating parameter:

```
let rec sumA xs n = match xs with
                    | []       -> n
                    | x::tail -> sumA tail (x+n)
```

- uses constant space and is much more efficient than

```
let rec sumC xs k = match xs with
                    | []      -> k 0
                    | x::tail-> sumC tail (fun v -> k(x+v))
```

Some relationships between `sum`, `sumA` and `sumC`:

1. $\texttt{sum}\ xs = \texttt{sumC}\ xs\ id$
2. $\texttt{sumA}\ xs\ 0 = \texttt{sumC}\ xs\ \texttt{id}$

Proof: Structural induction over lists
Validation: Use property-based testing

Continuations: Another example

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
```

The continuation-based version of `bigList` has a continuation

```
k: int list -> int list
```

as argument:

```
let rec bigListC n k =
   if n=0 then k []
   else bigListC (n-1) (fun res -> k(1::res));;
val bigListC : int -> (int list -> 'a) -> 'a
```

Continuations: Another example

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
```

The continuation-based version of `bigList` has a continuation

```
k: int list -> int list
```

as argument:

```
let rec bigListC n k =
   if n=0 then k []
   else bigListC (n-1) (fun res -> k(1::res));;
val bigListC : int -> (int list -> 'a) -> 'a
```

- Base case: “feed” the result into the continuation `k`.

Continuations: Another example

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
```

The continuation-based version of `bigList` has a continuation

```
k: int list -> int list
```

as argument:

```
let rec bigListC n k =
   if n=0 then k []
   else bigListC (n-1) (fun res -> k(1::res));;
val bigListC : int -> (int list -> 'a) -> 'a
```

- Base case: “feed” the result into the continuation `k`.
- Recursive case: The continuation after processing `(n-1)` is the function of the result `res` that

Continuations: Another example

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
```

The continuation-based version of `bigList` has a continuation

```
k: int list -> int list
```

as argument:

```
let rec bigListC n k =
   if n=0 then k []
   else bigListC (n-1) (fun res -> k(1::res));;
val bigListC : int -> (int list -> 'a) -> 'a
```

- Base case: “feed” the result into the continuation `k`.
- Recursive case: The continuation after processing `(n-1)` is the function of the result `res` that
  - builds the list `1::res` and
  - feeds that list into the continuation `k`.

Observations

- `bigListC` is a tail-recursive function, and

Observations

- `bigListC` is a tail-recursive function, and
- the calls of `k` are tail calls in the base case of `bigListC` and in the continuation: `fun res -> k(1::res)`.

Observations

- `bigListC` is a tail-recursive function, and
- the calls of `k` are tail calls in the base case of `bigListC` and in the continuation: `fun res -> k(1::res)`.

The stack will hence neither grow due to the evaluation of recursive calls of `bigListC` nor due to calls of the continuations that have been built in the heap:

```
bigListC 16000000 id;;
Real: 00:00:08.586, CPU: 00:00:08.314,
GC gen0: 80, gen1: 60, gen2: 3
val it : int list = [1; 1; 1; 1; 1;...]
```

- Slower than `bigList`
- Can generate longer lists than `bigList`

Towards a tail-recursive count version

Remember:

```
let rec count t = match t with
                  | Leaf          -> 0
                  | Node(tl,n,tr) -> let v1 = count tl
                                     let v2 = count tr
                                     1+v1+v2
```

Towards a tail-recursive count version

Remember:

```
let rec count t = match t with
                  | Leaf          -> 0
                  | Node(tl,n,tr) -> let v1 = count tl
                                     let v2 = count tr
                                     1+v1+v2
```

For a continuation- based version `countC` *t* *k*:

- *k* is the top-level continuation
- *k* is the continuation used for the base case: *k* 0

Towards a tail-recursive count version

Remember:

```
let rec count t = match t with
                  | Leaf          -> 0
                  | Node(tl,n,tr) -> let v1 = count tl
                                     let v2 = count tr
                                     1+v1+v2
```

For a continuation- based version `countC` *t k*:

- *k* is the top-level continuation
- *k* is the continuation used for the base case: *k* 0
- The continuation k2 for the second recursive call is

  ```
  fun v2 -> k(1+v1+v2)
  ```

Towards a tail-recursive count version

Remember:

```
let rec count t = match t with
                  | Leaf          -> 0
                  | Node(tl,n,tr) -> let v1 = count tl
                                     let v2 = count tr
                                     1+v1+v2
```

For a continuation- based version `countC` *t k*:

- *k* is the top-level continuation
- *k* is the continuation used for the base case: *k* 0
- The continuation k2 for the second recursive call is

  ```
  fun v2 -> k(1+v1+v2)
  ```

- The continuation k1 for the first recursive call is

  ```
  fun v1 -> countC tr k2
  ```

Towards a tail-recursive count version

Remember:

```
let rec count t = match t with
                  | Leaf          -> 0
                  | Node(tl,n,tr) -> let v1 = count tl
                                     let v2 = count tr
                                     1+v1+v2
```

For a continuation- based version `countC` ***t k***:

- ***k*** is the top-level continuation
- ***k*** is the continuation used for the base case: ***k*** 0
- The continuation k2 for the second recursive call is

  ```
  fun v2 -> k(1+v1+v2)
  ```

- The continuation k1 for the first recursive call is

  ```
  fun v1 -> countC tr k2
  ```

Hence, the recursive case is:

```
countC tl (fun v1 -> countC tr (fun v2 -> k(1+v1+v2)))
```

A tail-recursive count

Putting the pieces together:

```
let rec countC t k =
  match t with
  | Leaf          -> k 0
  | Node(tl,n,tr) ->
     countC tl (fun vl -> countC tr (fun vr -> k(1+vl+vr)))
```

A tail-recursive count

Putting the pieces together:

```
let rec countC t k =
  match t with
  | Leaf          -> k 0
  | Node(tl,n,tr) ->
     countC tl (fun vl -> countC tr (fun vr -> k(1+vl+vr)))
```

- Both calls of `countC` are tail calls
- The two calls of `k` are tail calls

Hence, the stack will not grow when evaluating `countC` *t k*; but the heap will.

A tail-recursive count

Putting the pieces together:

```
let rec countC t k =
  match t with
  | Leaf          -> k 0
  | Node(tl,n,tr) ->
     countC tl (fun vl -> countC tr (fun vr -> k(1+vl+vr)))
```

- Both calls of `countC` are tail calls
- The two calls of `k` are tail calls

Hence, the stack will not grow when evaluating `countC` *t k*; but the heap will.

- `countC` can handle bigger trees than `count`
- `count` is faster

Part II: Summary

On continuations

- a technique to turn arbitrary recursive functions into tail-recursive ones.
- trade stack for heap