BFNP – Functional Programming

Lecture 7: Imperative Features and Efficiency

Niels Hallenberg

These slides are based on original slides by Michael R. Hansen, DTU. Thanks!!!

The original slides has been used at a course in functional programming at DTU.

Overview

- Imperative programming,

by simple examples.

What is this?

```
let  ...
   let rec visit u =
      color.[u] <- Gray ; time := !time + 1; d.[u] <- !time
      let rec h v = if color.[v] = White
                    then  pi.[v]  <- u
                          visit v
      List.iter h (adj.[u])
      color.[u] <- Black
      time      := !time + 1
      f.[u]     <- !time

   let mutable i = 0
   while i < V do
      if color.[i] = White
      then visit i
      i <- i + 1
   (d, f, pi);;
```

Depth-First Search of directed graphs

"Direct" translation of pseudocode from Corman, Leiserson, Rivest.

Remaining parts:

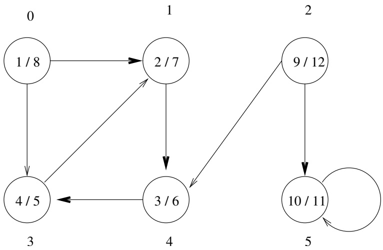
```
type color = White | Gray | Black;;

let dfs(V,adj: int list[]) =
   let color       = Array.create V White
   let pi          = Array.create V -1
   let d           = Array.create V -1
   let f           = Array.create V -1
   let time        = ref 0
   let rec visit u =
          ....

   let mutable i = 0
   while i < V do
      ....
   (d, f, pi);;
```

DFS – an example

```
val (d,f,pi) = dfs(g6);
```

d	:	Discovery times
f	:	Finishing times
pi	:	Predecessors

A node *i* is marked d_i/f_i



Elements of imperative F#

A *store* is a table associating values v_i with locations l_i:

$$\begin{bmatrix} l_1 & \mapsto & v_1 \\ l_2 & \mapsto & v_2 \\ & \cdots & \\ l_n & \mapsto & v_n \end{bmatrix}$$

Allocation of a new cell in the store

```
let mutable x = 1;;
val mutable x : int = 1

let mutable y = 3;;
val mutable y : int = 3
```

Results in the following environment and store:

Environment	Store
$\begin{bmatrix} \mathtt{x} & \mapsto & l_1 \\ \mathtt{y} & \mapsto & l_2 \end{bmatrix}$	$\begin{bmatrix} l_1 & \mapsto & 1 \\ l_2 & \mapsto & 3 \end{bmatrix}$

A similar effect is achieved by:

```
let x = ref 1;;
let y = ref 3;;
```

Value in a location in the store and Assignment

Given the following environment and store:

$$\text{Environment} \quad \begin{bmatrix} \mathtt{x} & \mapsto & l_1 \\ \mathtt{y} & \mapsto & l_2 \end{bmatrix} \qquad \text{Store} \quad \begin{bmatrix} l_1 & \mapsto & 1 \\ l_2 & \mapsto & 3 \end{bmatrix}$$

The assignment `x <- y+2` results in the new store:

$$\begin{bmatrix} l_1 & \mapsto & 5 \\ l_2 & \mapsto & 3 \end{bmatrix}$$

A similar effect is achieved by the assignment `x := !y + 2`

- The assignment `x := ...` is used
- The explicit “contentsOf” `!y` is necessary

when `let x = ref ...` and `let y = ref ...` are used

Default Values and Sequential Composition

Default values may be obtained using
`Unchecked.defaultof<`*type*`>`

```
> Unchecked.defaultof<int>;;
val it : int = 0
```

Two expressions are combined with `;` : $exp_1;exp_2$

Evaluate first exp_1 and then exp_2.

If exp_2 has type τ, then $exp_1;exp_2$ has type τ as well.
A warning is issued if exp_1 has a type different from unit.
You may use `ignore` to overrule the warning

```
let mutable x = 0;;
let y = (x;23);;
```

```
/Users/nielshallenberg/Dropbox/...: warning FS0020: This expression
should have type 'unit', but has type 'int'. Use
'ignore' to discard the result of the expression, or
'let' to bind the result to a name.
```

Arrays

- "'a [] is the type of one-dimensional, mutable, zero-based constant-time-access arrays with elements of type 'a."

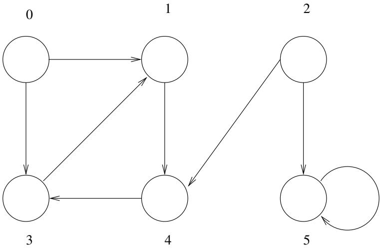
`Array.create` *n v* creates an array with *n* entries all containing *v*

Examples:

```
let a = Array.create 5 "a";;
val a : string [] = [|"a"; "a"; "a"; "a"; "a"|]

a.[2] <- "b";;
val it : unit = ()

a;;
val it : string [] = [|"a"; "a"; "b"; "a"; "a"|]

a.[0];;
val it : string = "a"
```

Graph representation: neighbour-list

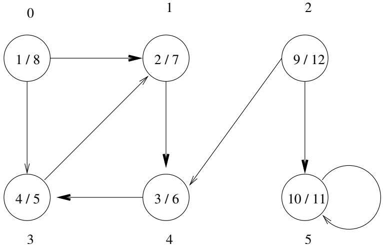
```
let adj =
   Array.ofList [ [1;3];
                  [4];
                  [4;5];
                  [1];
                  [3];
                  [5]]   ;;

let g6 = (6,adj);;


g6;;
val it : int * int list []
     = (6, [|[1; 3]; [4]; [4; 5]; [1]; [3]; [5]|])
```



Inspecting results

```
let (d,f,pi) = dfs(g6);;
```

```
d for Discovery times.
f for finishing times.
pi for predecessors.
```




```
>
val pi : int [] = [|-1; 0; -1; 4; 1; 2|]
val f : int [] = [|8; 7; 12; 5; 6; 11|]
val d : int [] = [|1; 2; 9; 4; 3; 10|]
```

Mutable record fields

You can also have a mutable record field

```
> type intRec = {mutable count : int } ;;

type intRec =
  {mutable count: int;}

> let r1 = {count = 0} ;;

val r1 : intRec = {count = 0;}

> let incr x =
  x.count <- x.count + 1
  x.count ;;

val incr : intRec -> int

> incr r1;;
val it : int = 1
> incr r1;;
val it : int = 2
```

Environment	Store
$[\ \texttt{r1} \ \mapsto \ \{count -> l_1\}\ ]$	$[\ l_1 \ \mapsto \ 2\ ]$

While loops

A while loop

`while` *b* `do` *e*

can be written as the function

`let rec wh () = if` *b* `then (`*e*`; wh()) else ()`

```
let mutable x = 0
while x < 10 do x<-x+1
let rec wh() = if x<10 then (x<-x+1;wh()) else ()
```

The compiler will generate the same code for the while loop and the tail-recursive `wh` function.

- F# is an excellent imperative language

- the combination of imperative and applicative constructs is powerful

In practice, only use imperative features when there is a good reason for it.

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)

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

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?

Append

The infix operator `@` (called 'append') joins two lists:

$$\begin{aligned}[x_1;x_2;\ldots;x_m]\ @\ [y_1;y_2;\ldots;y_n] \\ = [x_1;x_2;\ldots;x_m;y_1;y_2;\ldots;y_n]\end{aligned}$$

Properties

$$\begin{aligned}[\,]\ @\ ys &= ys \\ [x_1;x_2;\ldots;x_m]\ @\ ys &= x_1::([x_2;\ldots;x_m]\ @\ ys)\end{aligned}$$

Declaration

```
let rec (@) xs ys =
  match xs with
  | []       -> ys
  | x::xs'  -> x::(xs' @ ys);;
val (@) : 'a list -> 'a list -> 'a list
```

- Execution time is linear in the size of the first list

Another example: Naive reversal (I)

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

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

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

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:

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

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

Declaration of `factA`

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

An evaluation:

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

Space demand: constant.

Time demands: proportional to n

Declaration of `revA`

```
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. $facA(3, 20)$ 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, ...
```

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; ...]
```

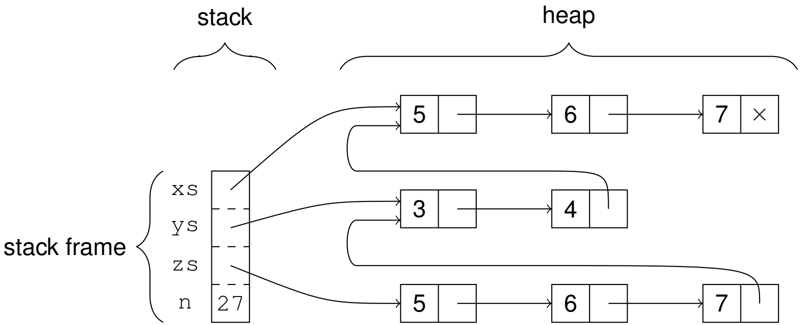
- The naive version takes 7.624 seconds - the iterative just 1 ms.
- The use of append (`@`) has been reduced to a use of cons (`::`). This has a dramatic effect of the garbage collection:
  - No object is reclaimed when `revA` is used
  - 825+253 obsolete objects were reclaimed using the naive version

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.

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:

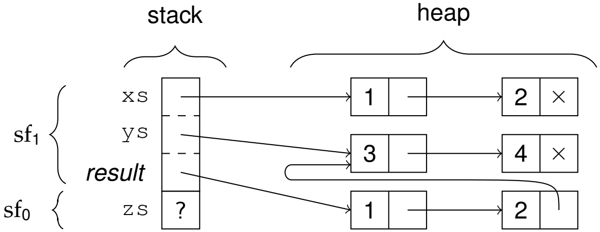
stack heap

sf_0 $\left\{ \texttt{zs} \; \boxed{?} \right.$

Operations on stack: Push

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

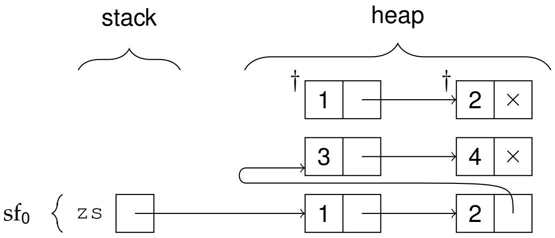
Evaluation of the local declarations initiated by pushing a new stack frame onto the stack:



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

Operations on stack: Pop

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 '†'
They will be reclaimed by the next GC.

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.
Stack size depends on setup - you may have to use much larger lists.
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;;
```

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 $p : \tau \texttt{->}$ `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 $g\ v$:

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

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.

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

Example: Fibonacci numbers (I)

A declaration based directly on the mathematical definition:

```
let rec fib = function
  | 0 -> 0
  | 1 -> 1
  | n -> fib(n-1) + fib(n-2);;
val fib : int -> int
```

is highly inefficient. For example:

```
     fib 4
⇝    fib 3 + fib 2
⇝    (fib 2 + fib 1) + fib 2
⇝    ((fib 1 + fib 0) + fib 1) + fib 2
⇝    ··· ⇝ 2 + (fib 1 + fib 0)
⇝    ···
```

Ex: `fib 44` requires around 10^9 evaluations of base cases.

Example: Fibonacci numbers (II)

An iterative solution gives high efficiency:

```
fun rec itfib(n,a,b) = if n <> 0
                       then itfib(n-1,a+b,a)
                       else a;;
```

The expression $\texttt{itfib}(n, 0, 1)$ evaluates to F_n, for any $n \geq 0$:

- Case $n = 0$: $\texttt{itfib}(0, 0, 1) \rightsquigarrow 0\ (= F_0)$
- Case $n > 0$:

$$
\begin{array}{ll}
 & \texttt{itfib}(n, 0, 1) \\
\rightsquigarrow & \texttt{itfib}(n-1,\ 1,\ 0) = \texttt{itfib}(n-1,\ F_1,\ F_0) \\
\rightsquigarrow & \texttt{itfib}(n-2,\ F_1 + F_0,\ F_1) \\
\rightsquigarrow & \texttt{itfib}(n-2,\ F_2,\ F_1) \\
\vdots & \\
\rightsquigarrow & \texttt{itfib}(0, F_n, F_{n-1}) \\
\rightsquigarrow & F_n
\end{array}
$$

Limits of accumulating parameters

Accumulating parameters are not sufficient to achieve a tail-recursive version for arbitrary recursive functions.

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 function for the “rest” of the computation.

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

```
c: int list -> int list
```

as argument:

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

- Base case: “feed” the result of `bigList` into the continuation `c`.
- Recursive case: let `res` denote the value of `bigList(n-1)`:
  - The rest of the computation of `bigList n` is `1::res`.
  - The continuation of `bigListC(n-1)` is

    ```
    fun res -> c(1::res)
    ```

Observations

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

Example: Tail-recursive count

```
let rec countC t c =
  match t with
  | Leaf          -> c 0
  | Node(tl,n,tr) ->
    countC tl (fun vl -> countC tr (fun vr -> c(vl+vr+1)))
val countC : BinTree<'a> -> (int -> 'b) -> 'b

countC (Node(Node(Leaf,1,Leaf),2,Node(Leaf,3,Leaf))) id;;
val it : int = 3
```

- Both calls of `countC` are tail calls
- The calls of the `c` is tail call

Hence, the stack will not grow when evaluating `countC` *t c*.

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

Summary and recommendations

- Loops in imperative languages corresponds to a *special case* of recursive function called tail recursive functions.
- Have iterative functions in mind when dealing with efficiency, e.g.
  - to avoid evaluations with a huge amount of pending operations
  - to avoid inadequate use of `@` in recursive declarations.
- Memory management: stack, heap, garbage collection
- Continuations – provide a technique to turn arbitrary recursive functions into tail-recursive ones.

  trades stack for heap

Note: Iterative function does not replace algorithmic idea and the use of good algorithms and datastructure.