

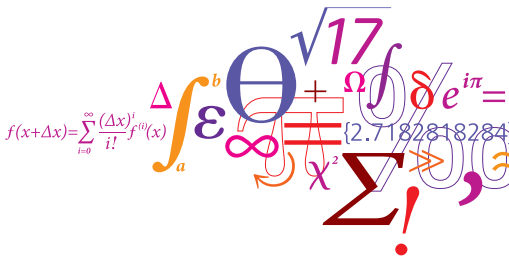
02157 Functional Programming

Lecture 11:

Imperative, Asynchronous, Parallel and Monadic Programming

A short story

Michael R. Hansen



DTU Informatics

Department of Informatics and Mathematical Modelling

- Imperative programming,
- asynchronous programming,
- parallel programming, and
- monadic 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);;
```

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

Remaining parts:

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

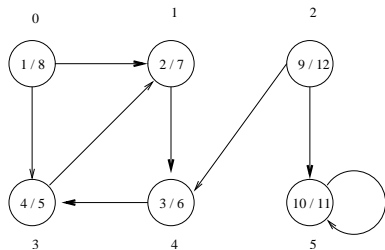
let dfs(V,adj: int list[]) =
  let color      = Array.create V White
  let pi        = Array.create V -1
  let d         = Array.create V -1
  let f         = Array.create V -1
  let time      = ref 0
  let rec visit u =
    ....

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

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

d : Discovery times
 f : Finishing times
 pi : Predecessors

A node i is marked d_i/f_i



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

$$\left[\begin{array}{lcl} l_1 & \mapsto & v_1 \\ l_2 & \mapsto & v_2 \\ & \dots & \\ l_n & \mapsto & v_n \end{array} \right]$$

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
$\left[\begin{array}{l} x \mapsto l_1 \\ y \mapsto l_2 \end{array} \right]$	$\left[\begin{array}{l} l_1 \mapsto 1 \\ l_2 \mapsto 3 \end{array} \right]$

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:

Environment	Store
$\left[\begin{array}{l} x \mapsto h_1 \\ y \mapsto h_2 \end{array} \right]$	$\left[\begin{array}{l} h_1 \mapsto 1 \\ h_2 \mapsto 3 \end{array} \right]$

The assignment $x \leftarrow y+2$ results in the new store:

$$\left[\begin{array}{l} h_1 \mapsto 5 \\ h_2 \mapsto 3 \end{array} \right]$$

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

- The assignment $x := \dots$ is used
- The explicit “contentsOf” $!y$ is necessary

when $\text{let } x = \text{ref } \dots$ and $\text{let } y = \text{ref } \dots$ are used

- `"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:

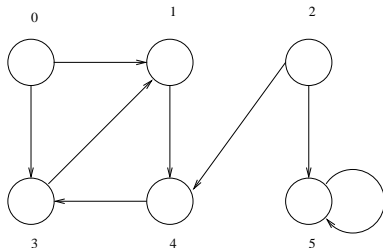
```
let a = Array.create 5 "a";;  
val a : string [] = [|"a"; "a"; "a"; "a"; "a"|]  
  
a.[2] <- "b";;  
val it : unit = ()  
  
a i;  
val it : string [] = [|"a"; "a"; "b"; "a"; "a"|]  
  
a.[0];;  
val it : string = "a"
```

```
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]/])
```

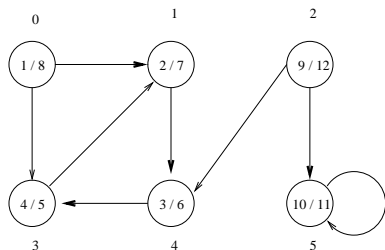


```
val (d,f,pi) = dfs(g6);

d;; (* Discovery times *)
val it : int []
    = [|1; 2; 9; 4; 3; 10|]

f;; (* Finishing times *)
val it : int []
    = [|8; 7; 12; 5; 6; 11|]

pi;; (* Predecessors *)
val it : int []
    = [| -1; 0; -1; 4; 1; 2|]
```



- F# is an excellent imperative language
- the combination of imperative and applicative constructs is powerful

Asynchronous computations by example

```
open System;;
open System.Net;;
let downloadDTUcomp =
  async {
    let webCl = new WebClient()
    let! html = webCl.AsyncDownloadString(
      Uri "http://www.dtu.dk")
    return html} ;;
val downloadDTUcomp : Async<string>
```

- 1 Create a `WebClient` object.
- 2 Initiate the download using `AsyncDownloadString`. This function makes the task an wait item and will eventually terminate when the download has completed. It uses no thread while waiting.
- 3 At termination the rest of the computation is re-started with the identifier `html` bound to the result.
- 4 The expression `return html` returns the value bound to `html`, that is, the result of the download.

Parallel downloads of web pages

```
let downloadComp url =
    let webCl = new WebClient()
    async {let! html = webCl.AsyncDownloadString(Uri url)
           return html};;
```

A computation for parallel downloads:

```
let downlArrayComp (urlArr: string[]) =
    Async.Parallel (Array.map downloadComp urlArr);;
val downlArrayComp : string [] -> Async<string []>
```

Activation of the computation:

```
let paralDTUandMScomp =
    downlArrayComp
        [| "http://www.dtu.dk"; "http://www.microsoft.com" |];;

Array.map (fun (s:string) -> s.Length)
    (Async.RunSynchronously paralDTUandMScomp);;
val it : int [] = [|45199; 1020|]
Real: 00:00:02.235, CPU: 00:00:00.046
```

Uses limited CPU time.

Parallel computation – exploiting multiple cores

```
type BinTree<'a> = | Leaf
                  | Node of BinTree<'a>*'a*BinTree<'a>;;
let rec exists p t =
  match t with
  | Leaf                -> false
  | Node(_,v,_) when p v -> true
  | Node(tl,_,tr)       -> exists p tl || exists p tr;;
```

Sequential search in big trees:

```
let rec genTree n range =
  if n=0 then Leaf
  else let tl = genTree (n-1) range
        let tr = genTree (n-1) range
        Node(tl, gen range, tr);;
let t = genTree 25 10000;;

exists (fun n -> isPrime n && n>10000) t;;
Real: 00:01:22.818, CPU: 00:01:22.727
val it : bool = false
```

Parallel search in big trees

```
open System.Threading.Tasks;;
let rec parExistsDepth p t n =
  if n=0 then exists p t else
  match t with
  | Leaf                -> false
  | Node(_,v,_) when p v -> true
  | Node(tl,_,tr)       ->
    let b1 = Task.Factory.StartNew(
      fun () -> parExistsDepth p tl (n-1))
    let b2 = Task.Factory.StartNew(
      fun () -> parExistsDepth p tr (n-1))
    b1.Result || b2.Result;;
val parExistsDepth: ('a->bool)->BinTree<'a>->int->bool
```

Experiments show that the best result is obtained using depth 4:

```
parExistsDepth (fun n -> isPrime n && n>10000) t 4;;
Real: 00:00:35.303, CPU: 00:02:18.669
```

The speedup is approximately 2.3.

Defining computation expressions

also called workflows or monads.

Purpose: hide low-level details in a builder class.

Expression evaluation with error handling:

```

let I e env =
  let rec eval = function
    | Num i      -> Some i
    | Var x      -> Map.tryFind x env
    | Add(e1,e2) -> match (eval e1, eval e2) with
                    | (Some v1, Some v2) -> Some(v1+v2)
                    | _                 -> None
    | Div(e1,e2) -> match (eval e1, eval e2) with
                    | (_ , Some 0)      -> None
                    | (Some v1, Some v2) -> Some(v1/v2)
                    | _                 -> None

  eval e;;
  
```

How can the Some/None manipulations be hidden?

Declaring a computation builder object

Define the computation type:

```
type maybe<'a> = option<'a>;;
```

Define a computation builder class:

```
type MaybeClass() =  
    member bld.Bind(m:maybe<'a>,f:'a->maybe<'b>):maybe<'b>  
        match m with | None    -> None  
                    | Some a  -> f a  
    member bld.Return a:maybe<'a> = Some a  
    member bld.ReturnFrom m:maybe<'a> = m  
    member bld.Zero():maybe<'a> = None;;
```

Declare a computation builder object:

```
let maybe = MaybeClass();;
```

Many improvements are possible, e.g. to delay computations

The Some/None manipulations are now hidden

```
let I e env =
  let rec eval = function
    | Num i      -> maybe {return i}
    | Var x      -> maybe {return! Map.tryFind x env}
    | Add(e1,e2) -> maybe {let! v1 = eval e1
                          let! v2 = eval e2
                          return v1+v2}
    | Div(e1,e2) -> maybe {let! v2 = eval e2
                          if v2<>0 then
                            let! v1 = eval e1
                            return v1/v2}

    eval e;;
  val I : expr -> Map<string,int> -> maybe<int>
```

F# supports a rich collection of different programming paradigms