Programmering og Problemløsning, 2017

Højere-ordens Funktioner — Part I

Martin Elsman

Datalogisk Institut
Københavns Universitet
DIKU

20. November, 2017

1. Højere-ordens Funktioner — Part I
   - Funktionsbegrebet
   - Funktioner der tager funktioner som argument
   - Funktioner der returnerer funktioner
   - Funktioner som/i datastrukturer
   - Funktionelle billeder

Højere-ordens Funktioner

Emner for i dag:

1. **Brug af højere-ordens funktioner:**
   - Funktionsbegrebet genbesøgt.
     (closures)
   - Funktioner der tager funktioner som argument.
     (liste-operationer, generisk sortering, ...)
   - Funktioner der returnerer funktioner.
     (funktionssammensætning, currying, uncurryring)
   - Funktioner som/i datastrukturer.
     (afbildninger, pull arrays, funktionelle billeder, ...)
2. **Funktionelle Billeder.**

   Vi skal se hvordan vi ved at forstå et billede som en **funktion fra punkter i planen til farver** kan definere (og tegne) en lang række interessante billeder.

Closures

På køretid er en funktion repræsenteret ved en såkaldt *closure* der, abstrakt set, indeholder tre dele:

1. En definition af de *formelle parametre* til funktionen (læs: variabelnavne).
2. En *omgivelse* der indeholder værdier for de variabler der ikke er formelle parametre til funktionen.
3. Kode for *kroppen* af funktionen.

Eksempel F# funktion defineret med `let`:

```
let a = 5+3
let f x = x + a
```

På køretid er funktionen `f` repræsenteret som

$$\mathtt{f} \quad \mapsto \quad (\ \mathtt{x}\ ,\ \{\mathtt{a} \mapsto 8\}\ ,\ \mathtt{x} + \mathtt{a}\ )$$

Bemærk:

- Med denne repræsentation kan funktionen benyttes også fra steder i programmet hvor `a` ikke er kendt (f.eks. i et eksternt bibliotek).

Anonyme Funktioner

Den samme kode kunne skrives ved brug af en *anonym* funktion:

```
let a = 5+3
let f = (fun x -> x + a)
```

Repræsentation:

$$\mathtt{f} \mapsto (\mathtt{x}, \{\mathtt{a} \mapsto 8\}, \mathtt{x} + \mathtt{a})$$

Bemærk:

- Der er ingen forskel på repræsentationen af de to definitioner af `f`.
- Anonyme funktioner benyttes ofte når en funktion umiddelbart skal gives til en anden funktion som argument, når vi umiddelbart skal gemme en funktion i en datastruktur, eller når vi umiddelbart skal returnere funktionen fra en anden funktion.

Funktioner der tager funktions-argumenter

Vi har tidligere set eksempler på funktioner der tager funktioner som parametre.

Eksempel: `List` modulet

```
val map    : ('a -> 'b) -> 'a list -> 'b list
val fold   : ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b
val filter : ('a -> bool) -> 'a list -> 'a list
```

Det kan ofte være hensigtsmæssigt selv at konstruere funktioner der er parametiske over andre funktioner.

Et oplagt eksempel er sortering:

- Sammenligningsbaserede sorteringsrutiner afhænger normalt blot af en funktion, der definerer en total ordning på elementerne i mængden. Funktionen kunne passende være en “mindre-end” funktion ($<$) med type `'a->'a->bool`:
  1. $\forall x.\ x \not< x$ (irreflexivity)
  2. $\forall x, y.\ x < y \Rightarrow y \not< x$ (asymmetry)
  3. $\forall x, y, z.\ x < y \wedge y < z \Rightarrow x < z$ (transitivity)
  4. $\forall x, y.\ x < y \vee y < x \vee y = x$ (totality)

Generisk Mergesort

```
let rec sel lt m ys =                          // select smallest
  function [] -> (m,ys)                        // of (m::ys)
         | x::xs -> if lt x m then
                       sel lt x (m::ys) xs
                    else sel lt m (x::ys) xs

let rec ssort (lt:'a->'a->bool) : 'a list -> 'a list =
  function [] -> []
         | x::xs ->
           let (m,xs) = sel lt x [] xs
           in m :: ssort lt xs
```

Brug af funktionen `ssort`:

```
> ssort (>) ["Dog"; "Apple"; "Horse"; "Monkey"];;
val it : string list = ["Monkey"; "Horse"; "Dog"; "Apple"]
```

Funktioner der returnerer funktioner

Det er ofte anvendeligt at kunne skrive funktioner der selv returnerer funktioner.

Eksempel: Funktionssammensætning

```
// compose f g = f ∘ g
let compose (f: 'a->'b) (g:'c->'a) : 'c->'b =
  fun x -> f(g x)
```

Denne funktion er direkte tilgængelig i F# som infix-funktionen <<:

```
> ((fun x->x+1) << (fun x->x*2)) 5;;
val it : int = 11
```

Bemærk:

- Funktionssammensætning (`f << g`) i F# svarer til matematisk funktionssammensætning, som i $f \circ g$, hvor $(f \circ g)(x) = f(g(x))$.
- Omvendt funktionssammensætning er i F# defineret ved `f >> g = g << f`.

Currying

Currying henviser til følgende indsigt:

1. En funktion `f:'a*'b->'c` der tager et par som argument kan omskrives til en funktion `g:'a->'b->'c` der tager to argumenter.
2. En funktion `g:'a->'b->'c` der tager to argumenter kan omskrives til en funktion `f:'a*'b->'c` der tager et par som argument (purity antaget).

Omskrivningerne kan realiseres med følgende to funktioner:

```
let curry (f:'a*'b->'c) : 'a->'b->'c =
  fun a -> fun b -> f(a,b)

let uncurry (f:'a->'b->'c) : 'a*'b->'c =
  fun (a,b) -> f a b
```

Eksempel:

```
> List.map (uncurry (+)) [(2,5);(8,1);(7,6)];;
val it : int list = [7; 9; 13]
```

Delvist anvendte funktioner

Det kan ofte være anvendeligt at anvende en funktion delvist for derved at skabe en ny funktion der passer i en sammenhæng.

Eksempel — pretty printing med "point-free" notation:

```
let rec padl n s = if String.length s > n then s
                   else padl (n-1) (" " + s)
let pp n = String.concat "\n"
           << List.map (String.concat " "
                        << List.map (padl n << string))
do printfn "%s" (pp 3 [[1;2;5];[12;3;25];[7;32;1]])
```

Kørsel:

```
  1   2   5    // (<<)   : ('a->'b) -> ('c->'a) -> ('c->'b)
 12   3  25    // padl   : int -> string -> string
  7  32   1    // concat : string -> string list -> string
               // map    : ('a -> 'b) -> 'a list -> 'b list
               // string : int -> string
```

Funktioner som objekter i datastrukturer

Funktioner (dvs. closures) kan gemmes i datastrukturer:

```
let rec loop i : (int->int) list =
  if i < 0 then []
  else (fun x -> i * x) :: loop (i-1)    // i is caught here!

let fs = loop 200
let xs = List.map (fun f -> f 3) fs
```

Resultatet af kaldet `loop 200`:

$$[(\,x\,,\,\{i \mapsto 200\}\,,\,i * x\,);\ (\,x\,,\,\{i \mapsto 199\}\,,\,i * x\,);\ \ldots;\ (\,x\,,\,\{i \mapsto 0\}\,,\,i * x\,)]$$

Indholdet af `xs`:

$$[600;\ 597;\ \ldots;\ 0]$$

Eksempler på funktioner brugt i datastrukturer

Funktioner kan tit med fordel blive brugt som/i datastrukturer.

Eksempler:

- Endelige afbildninger (lookup tables).

```
type 'a m = string -> 'a option
val lookup : 'a m -> string -> 'a option
val empty  : unit -> 'a m
val insert : string -> 'a -> 'a m -> 'a m
```

- Pull arrays (arrays der ikke nødvendigvis materialiseres i lageret)

```
type 'a parray = int * (int -> 'a)
val iota : int -> int parray       // [0; 1; ... ; n-1]
val map  : ('a -> 'b) -> 'a parray -> 'b parray
...
```

- Uendelige (dovne strømme af værdier).
- Funktionelle billeder (efter pausen).

Endelige afbildninger

```
type 'a m = string -> 'a option
let empty () : 'a m =
  fun _ -> None
let insert s v (m:'a m) : 'a m =
  fun x -> if x = s then Some v else m x
let lookup (m:'a m) s : 'a option =
  m s
```

Brug af funktionerne:

```
> let m = insert "b" 8 (insert "a" 5 (empty()));;
val m : int m
> lookup m "c";;
val it : int option = None
> lookup m "a";;
val it : int option = Some 5
```

Funktionelle Billeder

Funktionelle billeder illustrerer hvordan vi kan forstå et billede som en funktion fra et punkt i planen til, f.eks., en farve:

```
type point = float * float          // Points in the plane
type 'a image = point -> 'a         // Generic image

type frac = float                   // floats in [0;1]
type fcolor = frac*frac*frac*frac   // alpha,red,green,blue

type region = bool image            // region (b/w)
type cimage = fcolor image          // color images
```

Bemærk

- Repræsentationen fortæller ikke for hvilke værdier i planen vi skal forstå billedet.
- Når vi skal “tegne billedet” skal vi derfor vælge intervallet (for x og y) **og** antallet af pixels (vidde og højde) for det resulterende bitmap.

Farver

Vi har brug for hjælpefunktioner til at konvertere farver og til at omregne fra booleans til farver (sort/hvid).

Fra funktionelle farver til system-farver

```
let toColor ((a,r,g,b):fcolor) : System.Drawing.Color =
  ImgUtil.fromArgb (int(255.0*a),int(255.0*r),
                    int(255.0*g),int(255.0*b))

let boolToFColor (b:bool) : fcolor =     // black & white
  if b then (1.0,0.0,0.0,0.0)
  else (1.0,1.0,1.0,1.0)

let fracToFColor (f:frac) : fcolor =     // grey-scale
  (1.0,f,f,f)
```

Tegning af Funktionelle Billeder

Funktion nedenfor kan benyttes til at omdanne et vilkårligt funktionelt farve-billede til et bitmap som kan vises eller gemmes som png-fil.

Funktionen benytter sig af funktionalitet i `ImgUtil` biblioteket (`ImgUtil.mk` samt `ImgUtil.setPixel`):

```
let toBitmap (width:float) (img:cimage) w h : bitmap =
  let bmp = ImgUtil.mk w h
  for x in [0..w-1] do
    for y in [0..h-1] do
      let p_x = width * float (x - w/2) / float w
      let p_y = width * float (y - h/2) / float h
      let fc = img (p_x,p_y)   // function call!
      let c = toColor fc       // convert color
      in ImgUtil.setPixel c (x,y) bmp
  bmp
```

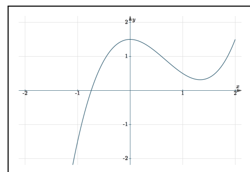
Tegning af Simple Funktioner

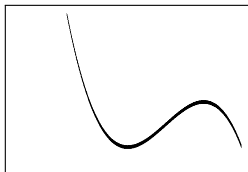
Vi kan let repræsentere en simpel funktion som et funktionelt billede.

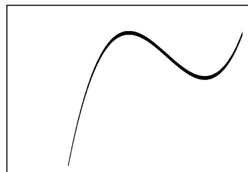
```
let (<<>>) : float -> float -> bool =
  fun (x:float) y -> abs(x-y) < 0.05
let f : region =                          // f(x) = x^3 - 2x^2 + 1
  fun (x,y) -> y <<>> (x*x*x-2.0*x*x+1.5)
let invY (f : 'a image) : 'a image =      // change y-direction
  fun (x,y) -> f (x,-y)
```

Generering af PNG:

```
let save600x400 f img = toPngFile f (toBitmap 4.0 img 600 400)
do save600x400 "f.png" (boolToFColor << f)
do save600x400 "finv.png" (invY(boolToFColor << f))
```







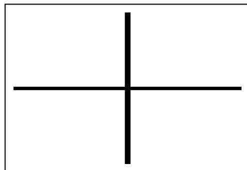
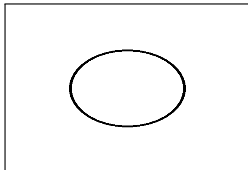
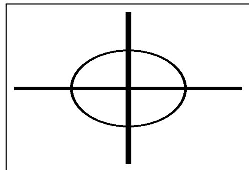
Tegning af Simple Relationer

Vi kan også repræsentere en simpel relation som et funktionelt billede:

```
let circ : region =                              // 1 = x^2 + y^2
  fun (x,y) -> 1.0 <<>> (x*x + y*y)
let coord : region =                             // +
  fun (x,y) -> x <<>> 0.0 || y <<>> 0.0
let (<||>) : region -> region -> region =        // combine regions
  fun r1 r2 -> fun p -> r1 p || r2 p
```

Generering af PNG:

```
do save600x400 "ccirc.png" (boolToFColor << (coord <||> circ))
```

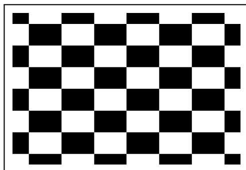




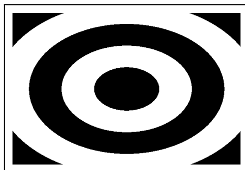
Skakbræt og Ringe

Det er muligt at definere andre relationer såsom et skakbræt eller alternerende ringe:

```
let even x = x % 2 = 0
let floori x = int(floor x)
let checker : region =
  fun (x,y) -> even(floori x + floori y)

let distO (x,y) = sqrt(x*x+y*y)
let altRings : region =
  even << floori << distO
```



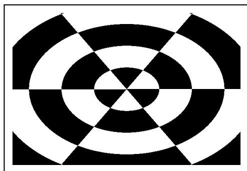


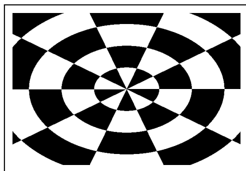
Polære Koordinater

I det polære koordinatsystem bestemmes et punkt *p* ved afstandet til centrum samt vinklen, i forhold til x-retningen, for linien der gennemgår centrum samt punktet.

```
type polar_point = float * float
let toPolar ((x,y):point) : polar_point =
  (dist0 (x,y), atan2 y x)

let polarChecker n : region =
  let sc (r,t) = (r, t * float n / pi)
  in checker << sc << toPolar
```





Gråskalabilleder

```
let wavDist : frac image =
  fun p -> (1.0 + cos (pi * dist0 p)) / 2.0

let save600x400s sz f img =
  toPngFile f (toBitmap sz img 600 400)

do save600x400s 7.0 "wavDist7.png" (fracToFColor << wavDist)
do save600x400s 9.0 "wavDist9.png" (fracToFColor << wavDist)
```

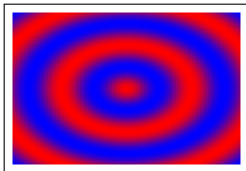
Billedinterpolation og Farver

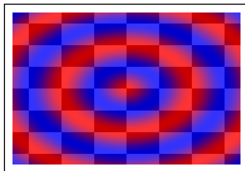
To farver kan "interpoleres" med vægt $w \in [0; 1]$ som følger:

```
let interpolC w (r1,g1,b1,a1) (r2,g2,b2,a2) : fcolor =
  let h x1 x2 = w * x1 + (1.0-w)*x2
  in (h r1 r2, h g1 g2, h b1 b2, h a1 a2)
let blueI : cimage = fun _ -> (1.0,0.0,0.0,1.0)
let redI  : cimage = fun _ -> (1.0,1.0,0.0,0.0)
```

Interpolation kan løftes til billeder:

```
let interpolI : frac image -> cimage -> cimage -> cimage =
  fun w a b p -> interpolC (w p) (a p) (b p)
let rbRings : cimage = interpolI wavDist redI blueI
let mystique : cimage =
  interpolI (fun _ -> 0.2) (boolToFColor<<checker) rbRings
```





Flere Muligheder

Læs mere i

> *Conal Elliott. Functional Images. Chapter in "The Fun of Programming". Book 2003. See* `http://conal.net/papers/functional-images/`

Brug `ImgUtil.runApp` til at styre parametre på køretid ved brug af tastaturinput.