

Programmering og Problemløsning, 2017

Typer og Mønstergenkendelse – Part III

Martin Elsman

Datalogisk Institut
Københavns Universitet
DIKU

27. Oktober, 2017

- 1 Typer og Mønstergenkendelse – Part III
 - Opsamling på Rekursion
 - Stakke og Køer
 - Rekursive Sum-Typer

Opsamling på Rekursion, Stakke og Kører, Abstrakte typer og Introduktion til Rekursive Sum-Typer

Emner for i dag:

1 Opsamling på rekursion.

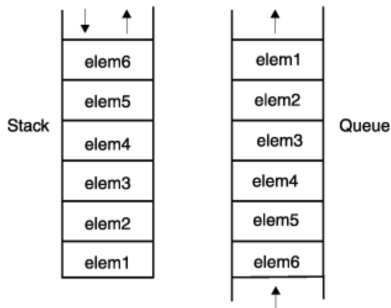
Eksempler på oversættelse af to matematiske definitioner til F# kode.

2 Stakke og køer.

To data-strukturer der let kan implementeres med lister og mønstergenkendelse og hvis implementation kan holdes abstrakt ved brug af **abstrakte modul typer**.

3 Introduktion til rekursive sum-typer.

Vi vil se på en simpel definition af en træ-struktur i F#.



Opsamling på Rekursion

Vi vil se på hvordan vi kan oversætte to rekursive matematiske formler til F# kode.

De to eksempler giver sammen mulighed for at beregne “Maximum Segment Sum” af et heltalsarray, hvor et *segment* er defineret som en vilkårlig sammenhængende del af arrayet.

Eksempel:

```
A = [| -2; 1; -3; 4; -1; 2; 1 |]; -5; 4 |] // MSS(A) = 6
```

Bemærk:

- Problemet er kun virkeligt interessant hvis arrayet indeholder negative værdier.
- Problemet er blandt andet relevant indenfor emner som gen-sekventering, billedgenkendelse og data-mining.

Maximum End-Segment Sum

Vi løser først et nemmere problem:

$$MESS_a(i) =$$

Find det største slut-segment i delarrayet $a[0]..a[i]$.

dvs: segmentet skal indeholde $a[i]$

Eksempel:

$$A = [-2; 1; -3; \boxed{4; -1}; 2; 1; -5; 4] \quad // \text{ MESS}_A(4) = 3$$

// 0 1 2 3 4

Rekursiv formel:

$$MESS_a(i) = \begin{cases} 0 & \text{if } i < 0 \\ \max \left\{ \begin{array}{l} a[i] \\ MESS_a(i-1) + a[i] \end{array} \right\} & \text{otherwise.} \end{cases}$$

Maximum End-Segment Sum – forsat

$$MESS_a(i) = \begin{cases} 0 & \text{if } i < 0 \\ \max \left\{ \begin{array}{l} a[i] \\ MESS_a(i-1) + a[i] \end{array} \right\} & \text{otherwise.} \end{cases}$$

F# kode:

```
let rec mess (a:int array) i =  
    if i < 0 then 0  
    else max (a.[i]) (mess a (i-1) + a.[i])  
  
let ex = [-2; 1; -3; 4; -1; 2; 1; -5; 4]  
do printfn "mess(ex)(4)=%A" (mess ex 4)
```

Maximum Segment Sum

Vi kan nu løse det lidt vanskeligere problem:

$MSS_a(i) =$

Find det største segment i en vilkårligt del af delarrayet $a[0]..a[i]$.

Eksempel:

$A = [|-2; 1; -3; \boxed{4}; -1; 2; 1; -5; 4|]$ // $MSS_A(4) = 4$
 // 0 1 2 3 4

Rekursiv formel:

$$MSS_a(i) = \begin{cases} 0 & \text{if } i < 0 \\ \max \left\{ \begin{array}{l} MSS_a(i) \\ MSS_a(i-1) \end{array} \right\} & \text{otherwise.} \end{cases}$$

Maximum Segment Sum – forsat

$$MSS_a(i) = \begin{cases} 0 & \text{if } i < 0 \\ \max \left\{ \begin{array}{l} MESS_a(i) \\ MSS_a(i-1) \end{array} \right\} & \text{otherwise.} \end{cases}$$

F# kode:

```

let rec mss (a:int array) i =
    if i < 0 then 0
    else max (mss a i)           // max segment at end
            (mss a (i-1))       // max segment somewhere before

let ex = [| -2; 1; -3; 4; -1; 2; 1; -5; 4 |]
do printfn "mss(ex)(|ex|-1)=%A" (mss ex (Array.length ex - 1))

```


Stakke

En *stak* er en data-struktur med et simpelt interface:

```
module Stack // content of stack.fsi

type 'a stack // LIFO
val empty : unit -> 'a stack
val push  : 'a stack -> 'a -> 'a stack
val pop   : 'a stack -> ('a * 'a stack) option
```

Spørgsmål:

- Hvordan implementers et stak-modul?
- Hvordan sikres det at man **KUN** kan tilgå værdier af type 'a stack med operationerne pop og push?



Stak-implementation

```
module Stack
```

```
type 'a stack = S of 'a list
let empty () = S []
let push (S s: 'a stack) v = S (v::s)
let pop (S s) : ('a * 'a stack) option =
  match s with
  | [] -> None
  | x::xs -> Some (x,S xs)
```

Bemærk:

- Enkel version ved brug af 'a list.
- Singleton “Sum-type” benyttes til at sikre **fuld abstraktion** (S konstruktør).
- Modul skal oversættes med både fsi-fil og fs-fil:

```
$ fsharpc -a stack.fsi stack.fs
$ fsharpi -r stack.dll
```

Køer

En *kø* er en data-struktur med et simpelt interface:

```
module Queue // content of queue.fsi

type 'a queue // FIFO
val empty : unit -> 'a queue
val insert : 'a queue -> 'a -> 'a queue
val remove : 'a queue -> ('a * 'a queue) option
```

Spørgsmål:

- Hvordan implementers et kø-modul?
- Hvordan sikres det at man **KUN** kan tilgå værdier af type 'a stack med operationerne insert og remove?



Kø-implementation – NOT GOOD – file queue_bad.fs

```
module Queue
```

```
type 'a queue = Q of 'a list           // BAD
let empty () = Q []                   // BAD
let insert (Q q: 'a queue) v = Q (v::s) // BAD
let remove (Q q) : ('a * 'a queue) option = // BAD
  match List.rev q with               // BAD
  | [] -> None                        // BAD
  | x::xs -> Some (x, Q (List.rev xs)) // BAD
```

Bemærk:

- Enkel version ved brug af 'a list.
- Singleton “Sum-type” benyttes til at sikre **fuld abstraktion** (Q konstruktør).

Spørgsmål:

- Hvad er problemet?

Kø-test – file `qtest.fs`

```
module Q = Queue
let q = List.fold (fun q v -> Q.insert q v) (Q.empty())
    [0..5000]
let rec loop q = match Q.remove q with
    | None -> 0
    | Some (v,q) -> v + loop q
let a = loop q
do printfn "sum(queue) = %d" a
```

Kørsel med `queue_bad.fs`

```
cp queue_bad.fs queue.fs
fsharp --nologo -a queue.fsi queue.fs
fsharp --nologo -r queue.dll qtest.fs
time mono qtest.exe
sum(queue) = 12502500
    3.60 real          4.12 user          0.14 sys
```

En bedre kø-implementation – file `queue_good.fs`

```
module Queue      // GOOD Queue Implementation

type 'a queue = Q of 'a list * 'a list
let empty () = Q ([],[])
let insert (Q (b,f)) v = Q (v::b,f)
let remove (Q (b,f)) : ('a * 'a queue) option =
  match f with
  | x :: xs -> Some (x,Q(b,xs))
  | [] ->
    match List.rev b with
    | [] -> None
    | x :: xs -> Some (x,Q([],xs))
```

Bemærk:

- To lister: en til “indsættelse” og en til “fjernelse”.
- Hvis listen til fjernelse er tom tages hele listen til indsættelse og indsættes i listen til fjernelse (efter at den er vendt om).

Kørsel af `qtest.fs` med `queue_good.fs`

```
cp queue_good.fs queue.fs
fsharpc --nologo -a queue.fsi queue.fs
fsharpc --nologo -r queue.dll qtest.fs
time mono qtest.exe
sum(queue) = 12502500
           0.06 real           0.04 user           0.00 sys
```

Bemærk:

- Vi har formået at ændre implementationen af kø-modulet uden at programmet `qtest.fs` kan “se forskel”.
- Applikationen virker stadig korrekt (hvilket kunne testes med black-box unit testing).
- Effekten er blot at programmet `qtest.fs` nu kører hurtigere! (Før 3 sekunder nu 60 millisekunder...)

Introduktion til rekursive sum-typer

Rekursive sum-typer er sum-typer der kan have konstruktører der tager argumenter hvis type refererer til sum-typen selv!

Eksempel:

```
type expr = Const of int                               // Expression trees
          | Add of expr * expr
          | Mul of expr * expr
```

Med simple rekursive funktioner er beregninger på sådanne sum-typer mulig:

Eksempel:

```
let rec evaluate (e:expr) : int =
  match e with
  | Const c -> c
  | Add (a,b) -> evaluate a + evaluate b
  | Mul (a,b) -> evaluate a * evaluate b

let x = Add(Mul(Const 3,Const 8),Const 8)
do printfn "evaluate(x)=%d" (evaluate x)
```