Programmering og Problemløsning, 2019

Træstrukturer — Part I

Martin Elsman

Datalogisk Institut
Københavns Universitet
DIKU

29. oktober, 2019

Rekursive Sum-Typer

Emner for i dag:

1. **Introduktion til rekursive sum-typer.**
   Vi vil se på et par simple definitioner af træstrukturer i F#.
2. **Generisk lighed.**
   F# har en indbygget operation til at bestemme **lighed** på værdier af den samme type, herunder på sum-type værdier.
3. **Forskellige typer af træer.**
   Træer med værdier i bladene (HTML)
   Træer med værdier i knuderne (mængder, binær søgning)
4. **Intro til Gennemløb af træstrukturer.**
   Mapping, foldninger, sletninger, indsættelser, balancering, ...

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!

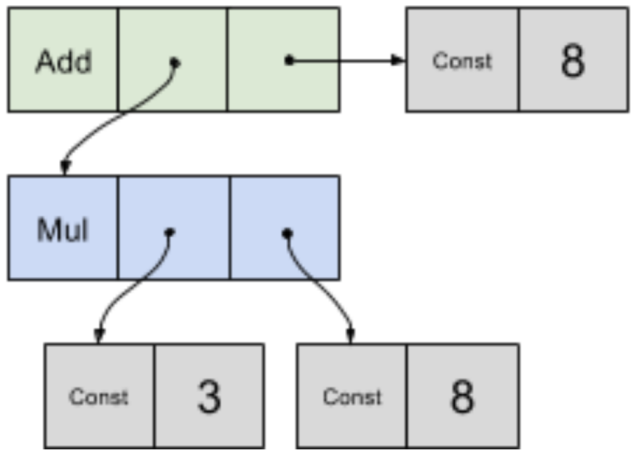
Eksempel:

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

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



Rekursive Sum-typer kan være type-generiske.

Rekursive sum-type definitioner kan (ligesom ordinære sum-typer) være *generiske* således at de er parameteriserede over en eller flere typer.

En generisk træ-type er et godt eksempel:

```
type 'a tree = Leaf of 'a | Tree of 'a tree * 'a tree
```

Vi kan nu skrive generisk (genbrugelig) kode:

```
let rec depth (t:'a tree) : int =
  match t with
  | Leaf _ -> 1
  | Tree (t1,t2) -> 1 + max (depth t1) (depth t2)
```

Her er en funktion som kun virker på specialiserede træer:

```
let rec sum (t:int tree) : int =
  match t with
  | Leaf x -> x
  | Tree (t1,t2) -> sum t1 + sum t2
```

Generisk lighed

F# har en indbygget "generisk funktion" til at undersøge lighed på data-strukturer:

```
val (=)  : 'a -> 'a -> bool   when 'a : equality
val (<>) : 'a -> 'a -> bool   when 'a : equality
```

Eksempler:

```
> 3 = 2;;
val it : bool = false
> (2,3.2) = (2,3.2);;
val it : bool = true
> [|2;3|] <> [|2;3|]
val it : bool = false
```

```
> 4.2 = 2.3;;
val it : bool = false
> [2;3] = [2;3]
val it : bool = true
> [2;3] = [2;4]
val it : bool = false
```

Bemærk:

- F# implementerer visse begrænsninger; det er ikke muligt at teste for lighed på værdier af funktionstype.

Generisk lighed på sum-typer — `eq.fs`

Generisk lighed virker også på sum-typer og er internt konceptuelt implementeret ved simpel rekursion og pattern-matching.

Eksempel:

```
let a = Add(Mul(Const 3,Const 8),Const 8)  // --> 24 + 8
let b = Add(Const 8,Mul(Const 6,Const 4))  // --> 8 + 24
do printfn "(a=b) = %A" (a=b)
do printfn "(eval a = eval b) = %A" (eval a = eval b)
```

Træer med værdier i bladene

I følgende definition af et træ er værdierne gemt i **bladene**:

```
type 'a tree = Leaf of 'a | Tree of 'a tree * 'a tree
```

En sådan definition kan være anvendelig f.eks. i forbindelse med at gøre streng-sammensætning effektiv. Hvad er f.eks. problemet med følgende kode:

```
let rec loop i =
  if i < 1 then "" else loop (i-1) + i.ToString()
in loop 50000
```

Problemet kan også ses i følgende kode:

```
let problematic = "hello" + " " + "world"
```

Vi kan i stedet opbygge et træ — `cstest.fs`:

```
let (++) x y = Tree (x,y)     // infix operator definition
let S s = Leaf s              // simple leaf construktor
let rec csloop i : string tree =
  if i < 1 then S"" else csloop (i-1) ++ S(i.ToString())
let cs = S"Numbers from 1 to 50000:\n" ++ csloop 50000
```

Konstruktion af den færdige streng:

```
let rec flatten (acc:'a list) (t:'a tree) : 'a list =
  match t with
  | Leaf s -> s :: acc
  | Tree (x,y) -> flatten (flatten acc y) x

let toString (x:string tree) : string =
  String.concat "" (flatten [] x)    // 50000x speedup!
```

Eksempel: HTML generering

```
module Html
type html
val S        : string -> html
val tag      : string -> html -> html
val (++)     : html -> html -> html
val toString : html -> string
```

Implementation:

```
module Html
type html = string tree
let S s = Leaf s
let (++) x y = Tree (x,y)    // infix operator definition
let tag t e = S("<"+t+">") ++ e ++ S("</"+t+">")
let toString (x:html) : string =
  String.concat "" (flatten [] x)
```

Eksempel: HTML generering — brug af bibliotek

```
> toString(tag "h2" (S"Nice world"));;
val it : string = "<h2>Nice world</h2>"
```

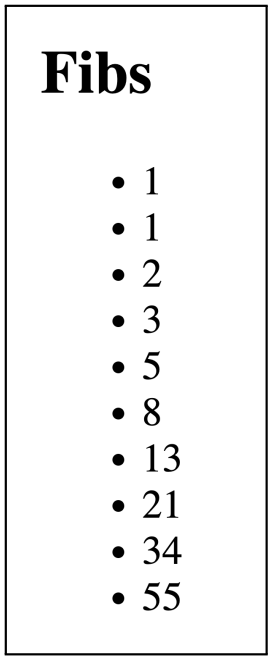
Mere interessant kode:

```
let flat (xs:html list) : html =
  List.foldBack (fun x acc -> x ++ acc) xs (S"")

let intitems (xs:int list) : html =
  let es = List.map (fun x -> tag "li" (S(x.ToString()))) xs
  tag "ul" (flat es)

let rec fib n = if n <= 2 then 1 else fib (n-1) + fib (n-2)

let doc =
  let fibs = List.map fib [1..10]
  in tag "html" (tag "body" (tag "h2" (S"Fibs") ++
                             intitems fibs))
```

Eksempel: HTML generering — output — `html.fs`:

`toString doc` giver følgende output:

```
<html>
  <body>
    <h2>Fibs</h2>
    <ul><li>1</li><li>1</li><li>2</li>
        <li>3</li><li>5</li><li>8</li>
        <li>13</li><li>21</li><li>34</li>
        <li>55</li>
    </ul>
  </body>
</html>
```



Bemærk:

- Det er let at konstruere nye interessante kombinatorer der kan bygge tabeller, etc.
- Vi skal senere se hvordan vi kan gemme den genererede HTML-kode i en HTML-fil.
- Teknikken kan også bruges i en **web-server** der serverer HTML-kode eller anden XML-formateret kode til klienter.

Træer med værdier i forgreningerne

I følgende definition af et træ er værdierne gemt i **forgreningerne**:

```
type 'a t = L | T of 'a t * 'a * 'a t
```

Funktion til opbygning af balanceret træ:

```
let rec build (l:'a list) : 'a t =
  match List.splitAt (List.length l/2) l with
  | ([],[]) -> L
  | (l1,x::l2) -> T(build l1,x,build l2)
  | _ -> failwith "impossible"
```

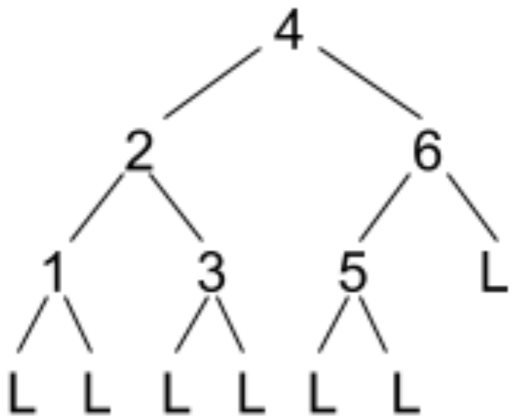
Spørgsmål:

- Hvad kan et sådan balanceret træ bruges til?

Balancerede træer kan fint bruges til søgetræer

Her er en funktion til at afgøre om et element er i træet:

```
let rec mem (t:int t) (v:int) : bool =
  match t with
  | L -> false
  | T(left,x,right) ->
   if v < x then mem left v
   else if v > x then mem right v
   else true
```



Bemærk:

- Der skal højst benyttes $O(\log N)$ operationer til at afgøre om et element er i træet.
- Balancerede træer kan således fint benyttes til repræsentation af mængder.
- Hvilke begrænsninger har den simple sum-type vi har givet?

Gennemløb af træstrukturer

Vi så i HTML-eksemplet hvordan vi kunne etablere en liste indeholdende informationen i alle bladene i et træ.

Andre træ-operationer

- map : omform data i bladene (eller knuderne)
- fold(Back) : akkumulér data i bladene eller knuderne (forfra eller bagfra)
  (her er der mange muligheder, afhængigt af i hvilken rækkefølge knuder skal processeres)
- indsætning
- sletning af element
- (re)balancering
- pretty-printing