TENTAMEN I DVA 201/229 FUNKTIONELL PROGRAMMERING MED F#

Onsdagen den 4 januari 2017, kl 14:10 – 18:30

LÖSNINGSFÖRSLAG

UPPGIFT 1 (6 POÄNG)

a) En enkel rekursion genom listan där vi adderar 1 varje gång vi påträffar ett positivt element:

```
let rec num_pos l =
  match l with
  | [] -> 0
  | x::xs when x > 0 -> 1 + num_pos xs
  | x::xs            -> num_pos xs
```

b) Vi komponerar en funktion som först filtrerar ut de positiva elementen till en lista och sedan räknar ut längden av listan:

```
let num_pos = List.filter (fun x -> x > 0) >> List.length
```

UPPGIFT 2 (3 POÄNG)

a) Vid funktionsanrop så räknas alla argument ut helt innan anropet görs.

b) Vid funktionsanrop så räknas ett argument ut först när dess värde verkligen behövs.

c) Det finns fall där lat evaluering terminerar men där ivrig evaluering hamnar i en oändlig rekursion. Ett exempel: vi deklarerar

```
let rec f x = f x
let g x = 17
```

Betrakta nu uttrycket `g (f 0)`. Med ivrig evaluering kommer `f 0` att försöka räknas ut först vilket ger en oändlig rekursion. Med lat evaluering, däremot, returneras direkt konstanten 17 och `f 0` räknas aldrig ut.

UPPGIFT 3 (6 POÄNG)

a) En enkel sökning genom elementen i arrayen. Sökningen utförs i den inre, rekursiva funktionen:

```
let find_array x (a : 'a []) =
  let n = a.Length - 1
  in
    let rec find_local i =
      if i > n then None
               else if a.[i] = x then Some i
                                 else find_local (i+1)
    in find_local 0
```

Notera att pga. ofullkomligheter i typinferensen hos F# krävs typdeklarationen på argumentet `a` för att tala om att det är en polymorf array. Jag har inte dragit några poäng om den deklarationen fattas.

b) Vi testar på om svaret från `find_array` är av formen `Some i` eller `None` och levererar den önskade utskriften i respektive fall:

```
let present_result y =
  match y with
  | Some i -> printf "x found at position: %d\n" i
  | None -> printf "x not found\n"
```

UPPGIFT 4 (4 POÄNG)

En enkel rekursion genom listan, där vi för varje element skriver över dess innheåll med funktionsvärdet och sen går vidare med resten av listan. `mutmap` returnerar unit-värdet `()`, det som är intressant är sidoeffekten på listan den får som argument:

```
let rec mutmap f l =
  match l with
  | [] -> ()
  | r :: rs -> r := f (!r); mutmap2 f rs
```

UPPGIFT 5 (6 POÄNG)

a)

```
type BTree = True | False | Not of BTree | Or of BTree * BTree
             | And of BTree * BTree
```

b) En enkel rekursion ned genom trädet, där man applicerar rätt boolesk operator på resultatet av evalueringen av deluttrycken beroende på vilken konstruktor man har:

```
let rec beval bt =
  match bt with
  | True -> true
  | False -> false
  | Not bt1 -> not (beval bt1)
  | Or (bt1,bt2) -> beval bt1 || beval bt2
  | And (bt1,bt2) -> beval bt1 && beval bt2
```

UPPGIFT 6 (5 POÄNG)

Vi vet:

```
[] : 'a list (första förekomsten)
[] : 'b list (andra förekomsten)
(::) : 'c -> 'c list -> 'c list (första förekomsten)
(::) : 'd -> 'd list -> 'd list (andra förekomsten)
(,) : 'e -> 'f -> ('e * 'f)
(+) : 'n -> 'n -> 'n där 'n är en numerisk typ
1 : int
```

Notera att för tomma listan och cons måste vi anta att de kan ha olika typ på de olika ställen där de förekommer. Därför ger vi dem en typ per förekomst, med olika typvariabler. Vi antar nu, för variablerna i deklarationen:

```
f : 'g
l : 'h
x : 'i
xs : 'j
```

I vänsterledet (VL) appliceras `f` på `l`. Detta är typkorrekt endast om `f` är funktionstypad med typen för `f` som argumenttyp, dvs.

```
'g = 'h -> 'k
```

för någon typvariabel `'k`. Vidare har VL typen `'k`. Låt oss nu gå igenom högerledet (HL), som är ett matchuttryck. För sådana gäller att (1) alla mönster måste ha samma typ som uttrycket man matchar på, (2) alla returnerade uttryck måste ha samma typ samt (3) typen på själva matchuttrycket måste vara samma som typen på de returnerade uttrycken. Vi börjar med uttrycket `x::xs`. För att det ska vara vältypat krävs att

```
'i = 'c
'j = 'c list
```

Dessutom måste typen på `x::xs` vara `'c list`. (1) ger att typerna för `[]` (första förekomsten) och `x::xs` måste vara lika, vilket medför

```
'a list = 'c list
```

som kan gälla endast om

```
'a = 'c
```

Vidare måste `l` ha samma typ, vilket medför att

```
'h = 'c list
```

Så vi vet nu att `f : 'c list -> 'k`. Låt oss nu titta på de uttryck som returneras från matchuttrycket. För att `(x,x+1)` ska vara vältypat krävs att typerna på `x` och `x+1` är samma som argumenttyperna för `(,)`, dvs

```
'c = 'e
'l = 'f
```

där `'l` är typen för `x+1`. Pga. typningen för `(+)` är

```
'l = 'n
```

Vidare har vi

```
'n = 'c (x argument till (+))
'n = int (1 argument till (+))
```

Ur detta drar vi slutsatsen att även `'f = 'l = 'c = int`. Vi erhåller

```
(x,x+1) : int * int
```

För att `f xs` ska vara vältypat krävs att argumenttypen för `f` är samma som typen för `xs`, dvs.

```
'c list = 'j = 'c list
```

vilket gäller trivialt. Då får `f xs` typen `'k`. Slutligen, för att `(x,x+1) :: f xs` ska vara vältypat krävs att typerna för `(x,x+1)` och `f xs` är samma som för respektive argumenttyp för `(::)` (andra förekomsten). vilket ger

```
int * int = 'd
'k = 'd list = (int * int) list
```

ur vilket vi drar slutsatsen att `f : 'c list -> ('c * 'c) list`. Vidare blir typen för `(x,x+1) :: f xs` lika med `('c * 'c) list`. Denna typ ska vara samma som typen för `[]` (andra förekomsten), vilket ger

```
'c * 'c = 'd
'k = 'd list
```

ur vilket vi frar slutsatsen att

```
f: 'c list -> ('c * 'c) list
```

vilket, eftersom `'c = int`, medför

```
f: int list -> (int * int) list
```

Vidare blir typen för `(x,x+1) :: f xs` lika med `(int * int) list`. Denna typ ska vara samma som typen för `[]` (andra förekomsten), vilket ger

```
'b list = (int * int) list
```

som går ihop sig om `'b = (int * int)`.

Slutligen kollar vi att typerna för VL och HL är lika. Typen för HL är `(int * int) list`. Typen för VL är lika med resultattypen för `f`, vilken är samma. Vi har då checkat alla typningar och kommit fram till att de går ihop sig om

```
f : int list -> (int * int) list
```

Att detta är den mest generella typen följer av att vi i varje steg gjort minimala antaganden om typningen på de olika deluttrycken.