TENTAMEN I DVA 229 FUNKTIONELL PROGRAMMERING MED F#

Tisdagen den 7 juni 2022, kl 14:30 – 18:30

LÖSNINGSFÖRSLAG

UPPGIFT 1 (6 POÄNG)

a) En enkel rekursion genom listan där värdet på listhuvudet avgör om dess värde + 1 ska läggas till i resultatlistan eller ej:

```
let rec f l =
  match l with
  | [] -> []
  | x::xs when x < 0 -> f xs
  | x::xs -> (x + 1) :: f xs
```

b) Vi använder `List.filter` för att plocka ut de icke-negativa elementen och `List.map` för att öka på dem med 1:

```
let f' = List.filter (fun x -> x >= 0) >> List.map (fun x -> x + 1)
```

UPPGIFT 2 (4 POÄNG)

Summeringen sker i den inre funktionen `localsum`. `l` och `u` bildar ett intervall: summeringen sker över det intervallet genom att dela det i två och rekursivt summera delarna. I och med att vi använder heltal för att peka ut vilken del av arrayen att summera sker ingen kostsam kopiering av sub-arrayer.

```
let balsum (a : int []) =
  let rec localsum l u =
    if l = u then a.[l]
             else localsum l ((l + u)/2) +
                  localsum ((l + u)/2 + 1) u
  in localsum 0 (Array.length a - 1)
```

(`a` måste typas explicit för att typinferensen ska fungera. Inget avdrag har gjorts om denna typning saknas.)

UPPGIFT 3 (2 POÄNG)

F# använder call-by-value för `List.take`. Det innebär att hela listan (10^7 element) räknas ut först innan det första elementet returneras. Denna uträkning tar en del tid.

För sekvenser använder F# däremot call-by-need, bara så mycket som för tillfället behövs av en sekvens räknas ut. I exemplet räkknas bara en liten del av sekvensen ut innan det första elementet returneras. Detta går betydligt fortare.

UPPGIFT 4 (4 POÄNG)

Vi gör en lösning där vi först sätter upp en muterbar referenscell `acc` och sen rekurserar genom listan. För varje element `x` där `p x` är sant inkrementeras `acc` med ett och när listan gåtts igenom returneras nuvarande värde som är lagrat i `acc`:

```
let numElem p l =
  let acc = ref 0
  let rec local l =
    match l with
    | []               -> !acc
    | x :: xs when p x -> acc := !acc + 1; local xs
    | x :: xs          -> local xs
  in local l
```

OBS att referenscellen måste deklareras och initieras utanför den rekursiva inre funktionen `local`. Om inte så hade en ny referenscell skapats för varje nytt rekursivt anrop och då hade inte ackumuleringen fungerat.

UPPGIFT 5 (2 POÄNG)

`s.[i]` är det i'te tecknet i strängen `s` och har typen `char`. `s.[i..i]` däremot är en sträng med ett tecken som alltså innehåller samma tecken som `s.[i]` men har typen `string`.

UPPGIFT 6 (2 POÄNG)

Ett deklarerat värde är inte synligt förrän efter den punkt där det är deklarerat. Så i deklarationen av `g` är inte `f` synlig ännu och kompilatorn kommer att säga att `f` inte är deklarerad.

UPPGIFT 7 (6 POÄNG)

a) En rättfram deklaration med två fall, för löv och för intern nod:

```
type Tree<'a> = Leaf of 'a | Node of Tree<'a> []
```

b) En lösning där vi rekursivt summerar delträden ända tills vi kommmer ned till löven. Vad som är lite speciellt är att vi rekursivt anropar funktionen på alla delträd i arrayen med hjälp av `Array.map` för att sen summera resultaten med `List.sum`. Vi hade också kunnat deklarera en inre hjälpfunktion för dessa ändamål.

```
let rec sumTree (t : Tree<int>) =
  match t with
  | Leaf n -> n
  | Node a -> Array.sum (Array.map sumTree a)
```

(`t` måste typas explicit för att typinferensen ska fungera. Inget avdrag har gjorts om denna typning saknas.)

UPPGIFT 8 (4 POÄNG)

Vi vet:

```
1 : int
(+) : 'n -> 'n -> 'n, 'n numerisk typ
```

Vi antar nu, för variablerna i deklarationen:

```
f : 'a
x : 'b
y : 'c
```

Vi kollar nu att alla uttryck är typkorrekta samt att typen på vänsterledet (VL) är lika med typen på högerledet (HL). Under den processen kommer villkoren på typen för `f` successivt att skärpas. När processen är klar kommer dessa villkor att ge den mest generella typen för `f`.

Först kollar vi VL. Där appliceras `f` på `x`. Detta är typkorrekt endast om `f` är funktionstypad med typen för `x` som argumenttyp, dvs.

```
'a = 'b -> 'd
```

för någon typvariabel `'d`. Vidare har VL typen `'d`. Låt oss nu gå igenom högerledet (HL), som är ett let-uttryck. Från deklarationen av `y` har vi direkt att `y` måste ha typen `int`, dvs.

```
'c = int
```

För att uttrycket `y + f x` ska vara vältypat måste då gälla att

```
'n = int
```

och vidare att returtypen för `f` måste ha typen `int`, dvs.

```
'd = int
```

vilket medför

```
'a = 'b -> int
```

Vi noterar vidare att delutttrycket `f x` är vältypat eftersom `x : 'b` och `f : 'b -> int`. Typen för HL är samma som för `y + f x`, dvs. `int`. Detta är samma typ som för VL. Vi har därmed gått igenom hela deklarationen och svaret är

```
f : 'b -> int
```

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

Lycka till! *Björn*