TENTAMEN I DVA 229 FUNKTIONELL PROGRAMMERING MED F#

Fredagen den 3 juni 2016, kl 14:10 – 18:30

LÖSNINGSFÖRSLAG

UPPGIFT 1 (5 POÄNG)

a) Eftersom bara de element i den representerade arrayen som är skilda från noll bidrar till summan räcker det att gå igenom listan av par som representerar arrayen, för varje element fiska upp värdet samt att lägga detta till summan som successivt beräknas. Med lite pattern-matching går detta galant:

```
let rec sumarr l =
  match l with
  | (i,x)::xs -> x + sumarr xs
  | [] -> 0.0
```

b) med `List.map` gör vi först en lista av de lagrade värdena som sen summeras med `List.fold`. Vi kopplar ihop dem med funktionskomposition (`>>`):

```
let sumarr = List.map (fun (i : int,x) -> x) >> List.fold (+) 0.0
```

(Pga. begränsningar i typsystemet måste vi ha en explicit typannotering på argumentet `i`. Jag har inte dragit några poäng om den fattas.)

UPPGIFT 2 (2 POÄNG)

`f` kommer att få typen `int -> int` beroende på att dess formella argument `x` adderas med konstanten `17` som har typen int. Detta medför att både `x` (argumentet till `f`) och `x + 17` (returvärdet) måste ha typen `int`.

För `g` däremot blir det ett typfel, beroende på att argumentet till `f` pga. typannoteringen har typ `float` men `f` förväntar sig en `int`.

UPPGIFT 3 (5 POÄNG)

Den här uppgiften går att lösa på flera sätt. Jag har valt en metod där man går igenom siffrorna från mindre till mer signifikanta positioner, med två ackumulerande argument: en som successivt samlar på sig summan av bidragen från siffrorna och en som håller tiopotensen för den aktuella siffran. Lösningen är strukturerad i en funktion som avkodar siffrorna (tecknen) till heltal och en funktion som gör själva konverteringen. I den senare finns en lokal funktion, med de ackumulerande argumenten, som utför själva beräkningen.

```
let c2int c =
  match c with
  | '0' -> 0
  | '1' -> 1
  | '2' -> 2
  | '3' -> 3
  | '4' -> 4
  | '5' -> 5
  | '6' -> 6
  | '7' -> 7
  | '8' -> 8
  | '9' -> 9
```

```
  | _   -> failwith "Non-numeric character"

let string2int (s : string) =
  let rec s2i sum power i =
    if i < 0 then sum
             else s2i (sum + power*c2int (s.[i])) (power*10) (i-1)
  in s2i 0 1 (String.length s - 1)
```

(Återigen behövs en explicit typannotering på argumentet. Typinferensen kan inte från koden avgöra om `s` ska ha typen `string` eller `char []` och eftersom dessa är olika typer måste konflikten lösas upp manuellt. Jag har inte dragit några poäng om annoteringen fattas.)

UPPGIFT 4 (3 POÄNG)

a) När anropet till `g` sker kommer systemet att försöka räkna ut `f -1` vilket resulterar i en oändlig rekursion.

b) Uträkningen av `f -1` kommer att skjutas upp tills dess att dess värde verkligen behövs. Vad som händer vid anropet av `g` är att `0` returneras utan att `f -1` någonsin behöver räknas ut.

c) Ivrig evaluering.

UPPGIFT 5 (4 POÄNG)

Den muterbara referenscellen `m` ackumulerar summan. Eftersom `m` är en lokal variabel i `maxarray` syns den inte utåt, vilket medför att `maxarray` blir en ren funktion trots att den internt använder sidoeffekter. Arrayen `a` traverseras i den inre, rekursiva funktionen `max_local`. När summeringen är klar returneras `m`:s värde, `!m`.

```
let maxarray (a : float []) =
    let m = ref a.[0]
    let n = Array.length a
    let rec max_local i =
        if i = n then !m
                 else m := if a.[i] > !m then a.[i] else !m;
                      max_local (i+1)
    in max_local 1
```

UPPGIFT 6 (6 POÄNG)

a) En rättfram typdeklaration. (För att snofsa till det lite och göra deklarationen mer läsbar introducerar vi två typsynonymer "`radius`" och "`coordinate`", men man kan också klara sig utan dem.)

```
type radius = float
type coordinate = float * float * float
type STree = Tree of (STree * STree) | Sphere of radius * coordinate
```

b) En enkel trädtraversering där sfärerenas volymer summeras rekursivt:

```
let rec volume t =
  let pi = 3.141592654 in
    match t with
    | Sphere (r,_) -> 4.0*pi*r/3.0
    | Tree (t1,t2) -> volume t1 + volume t2
```

UPPGIFT 7 (5 POÄNG)

a) den tar en lista och returnerar en lista av par där varje listelement `x` har ersatts av ett par `(x,x)`.

b) 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)
```

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:

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

I vänsterledet (VL) appliceras `pair` på `l`. Detta är typkorrekt endast om `pair` är funktionstypad, 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 `[]` 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 `pair : 'c list -> 'k`. Låt oss nu totta på de uttryck som returneras från matchuttrycket. För att `(x,x)` ska vara vältypat krävs att typen på `x` är samma som argumenttyperna för `(,)`, dvs

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

Detta ger att typen för `(x,x)` är `'c * 'c`. För att `pair xs` ska vara vältypat krävs att argumenttypen för `pair` är samma som typen för `xs`, dvs.

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

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

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

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

```
'b list = 'd list
```

som går ihop sig om `'b = 'd`.

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

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

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