Funktionell programmering

DD1361

Högre ordningens funktion

En funktion är en högre ordningens funktion om:
- den tar en eller flera funktioner som argument

eller
- returnerar en funktion

eller
- båda ovanstående

(Sebesta s. 695)

Funktionskomposition

- I matte: $f \circ g$ ”sammanfogar” f och g.
  Dvs $f \circ g(x) = f\,(g(x))$
- I Haskell: `(f . g) x`
- Kräver alltså att

```
g :: a -> b
f :: b -> c
```

Funktionskomposition, exempel

Dubbel tillämpning:

```
twice :: (a -> a) -> (a -> a)
twice f = f . f

kvadrat x = x*x

*Main> twice kvadrat 2
16
```

Lambda-kalkyl

Alonzo Church på 30-talet: formaliserade beräkningar med speciell notation kallad λ-kalkyl.

Delar av lambda-kalkylen har inspirerat den funktionella paradigmen. I scheme används lambda.

Allmänt: $\lambda x \rightarrow \text{expr}$ $\lambda x\, y \rightarrow \text{expr}$

Exempel: $\lambda\, x\, y \rightarrow 1/(x + y)$ blir i Haskell:

```
\ x y -> 1 / (x + y)
```

```
Prelude> (\x y -> 1 / (x + y)) 2 3
0.2
```

Lambda-kalkyl i Haskell

Man kan betrakta funktioner i Haskell som uppbyggda mhja λ-notation dvs

```
addera = \x y -> x + y

*Main> :t addera
addera :: Integer -> Integer -> Integer
```

Vilken är typen om man bara skickar med ett värde till addera?

```
*Main> :t (addera 2)
(addera 2) :: Integer -> Integer
```

Skapa funktioner av funktioner

På detta vis kan man bygga upp nya funktioner och som i detta fall kan man utgå från en generalisering och gå till en specialisering.

```
add2 = (addera 2)
```

Currying

Funktioner som tar ett argument och returnerar en ny funktion kallas Curried functions

Currying är en princip i Haskell:

Alla funktioner tar alltid en parameter och returnerar ett värde eller en ny funktion.

Haskell har fått namnet från Haskell B. Curry som är en av upphovsmännen till lambda-kalkylen. (Thompson s.236)

Syntaktiskt socker döljer λ-notation!

Med socker:

```
inc x = x + 1
```

eller

```
inc = (+) 1
```

Utan socker:

```
inc = \ x -> x + 1
```

Anonyma funktioner

Kan använda lambda-notationen för att skapa icke namngivna funktioner.

Exempel:

```
addOneList = map (\x -> x + 1) [1, 2, 3]
```

Kan man ange en typsignatur för en anonym funktion?

```
((\x -> x + 1) :: Int -> Int)
```

Några typiska grupper för funktioner

1) Mappning - applicera en funktion på alla element
2) Filtrera - välja element
3) Folding - Kombinera element
4) Bryta upp listor
5) Kombinationer
6) Primitiv rekursion och folding
7) Övrigt

Ett återkommande mönster

Scheme, schematiskt

```
(define funktionsnamn
  (lambda (lista indata)
    (if (listan är tom)
        returnera ett passande värde
        ((gör något med första elementet)
          sammanfogning
         (undersök resten av listan)))))
```

Ett återkommande mönster

Haskell, schematiskt

```
funknamn indata [] =
   returnera ett passande värde
funknamn indata
  (förstaElementet : restenAvListan) =
       ((gör något med förstaElementet)
        sammanfogning
        (undersök resten av listan))
```

map idé

```
map1 f [] = []
map1 f (x:xs) = (f x : (map1 f xs))

*Main> map1 kvadrat [1,2,3]
[1,4,9]

convertChars :: [Char] -> [Int]
convertChars str = map1 fromEnum str

*Main> convertChars "Ann"
```

Filter

Signatur:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Exempel 1:

```
Prelude> [1,2..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> filter odd [1,2..10]
[1,3,5,7,9]
```

Filter, forts

Exempel 2:

```
Prelude>filter even (map length
                ["Chalmers, "LiTH, "KTH"])
[8,4]
```

Exempel 3:

```
Prelude> filter ((\ x -> x <= 3).length)
             ["Tom","Oskar","Ann","Lisa"]
["Tom","Ann"]
```

Filter, idé

```
filter1 predikat [] = []
filter1 predikat (x:xs)
  | predikat x = x : filter1 predikat xs
  | otherwise = filter1 predikat xs


*Main> filter1 (>0) [-1,1,-2,2]
[1,2]

-- med listomfattning
filter2 predikat xs =
                [ x | x<-xs,   predikat x ]
```

Generalisering

```
length1 [] = 0
length1 (x:xs) = 1 + (length1 xs)

*Main> length1 [1,2,3]
3

sum1 [] = 0
sum1 (x:xs) = x + (sum1 xs)

*Main> sum1 [1,2,3]
6
```

Hitta det gemensamma mönstret!

Folding - kombinera element

Tanken är att man viker in en operator mellan listans element. Generaliserar mhja folding.

”Veckla ut” definitionen:

$(f\ e_1\ (f\ e_2\ (f\ e_3\ \ldots(f\ e_x\ \text{startvärde})\ \ldots)))$

```
foldr _ start [] = start
foldr f start (e : rest) =
              (f e (foldr f start rest))
```

Användning av foldr

Summera alla tal i en lista

```
summa = foldr (+) 0 [1,2,3]
```

Listlängd:

```
length lst =
       foldr (\el acc -> acc + 1) 0 lst
```

Listlängd med Currying:

```
length = foldr (\el acc -> acc + 1) 0
```

Map:

```
map f =
    foldr (\el rest -> (f el : rest)) []
```

Hur kan vi vända en lista, reverse ?

```
reverse1 xs = foldr snoc [] xs

-- snoc; cons  baklänges!
snoc x xs = xs ++ [x]

*Main> reverse1 [1,2,3]
[3,2,1]
```

Inte alltid rätt att jobba ”från höger”.

- Inte alltid effektivt att jobba ”från höger”.
- Upprepade ”append” med ++!
- Varför kan vi inte använda (:)?

```
foldl _ start [] = start
foldl op start (e : rest) =
          (foldl op (op start e) rest))

reverse = foldl (\xs x -> x : xs) []
```

Funktioner

- Har samma ”rättigheter” som andra värden.
- Behöver inte namnges, anonym
- Nya funktioner från gamla, currying
- Kan ges som argument
- Kan returneras som värde
- Kan lagras i datastrukturer

Funktioner i lista

```
fkn = [(+),(-)]

*Main> (head fkn) 2 1
3

*Main> (head (tail fkn)) 2 1
1
```

Strikt evaluering

Def: Strikt evaluering innebär att alla parametrars värde är kända när en operator eller funktion anropas.

Ex: Funktionsanrop i Java, C, etc

Icke-strikt evaluering

Hur beräknas

- `if jobbigt(17) && tungt(4711) then ...`?

Vanligen: Man ”kortsluter” jämförelsen.

Implementeras för `and` och `or`

Vad händer här?

```
int main(int argc, char** argv) {
    if (hej() || hopp()) {
       printf("du glade\n");}
    else {
       printf("nix pix!\n");}}
int hej() {
    printf("hej ");
    return 1;}
int hopp() {
    printf("hopp ");
    return 1;}
```

Lat evaluering

Hur beräknas

- `head [1..10]`?
- `head (qsort longlist)`?
- Lat evaluering i Haskell:

  ”Beräkna bara det som behövs”
- Varje uttryck lagras som ett löfte om evaluering — vid behov.

Se Hutton eller be en kursare förklara sina anteckningar.

Testa latheten!

Vad händer med

`head (qsort [4,5,4,1,2,9])?`

- Instrumentera koden:

```
import Hugs.Observe
-- Ej standardmodul
qsort [] = []
qsort (x : xs) =
 qsort [(observe ’’e1’’ e) | e<-xs, e<x]
 ++ [(observe ’’x’’ x)]
 ++ qsort [(observe ’’e2’’ e) |
                              e<-xs, e>=x]
```

Testa latheten!

```
QS> head (qsort [4, 5, 4, 3, 1, 2,9])
1
>>>>>>> Observations <<<<<<
e1
1
3
2
2
1
x
1
QS>
```

Inga värden på `e2` — det uttrycket har aldrig behövt evalueras

Vad händer med...

- ones = (1 : ones)?

```
Prelude> ones
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...∞
```

- take 10 ones ?

```
Prelude> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

- ex = (1 : map ((+) 1) ex)?

```
Prelude> take 10 ex
[1,2,3,4,5,6,7,8,9,10]
```

Strömmar: Oändliga listor

Typexemplet: Fibonaccitalen

Antag att `fibs` är listan av Fibonaccital.

```
      1 1 2 3  5  8 13 21 ... fibs
    + 1 2 3 5  8 13 21 34 ... tail fibs
      2 3 5 8 13 21 34 55 ... tail(tail fibs)
```

```
fibs = 1:1:(elementWiseAdd fibs
                           (tail fibs))
    where elementWiseAdd = zipWith (+)
Prelude> take 10 fibs
[1,1,2,3,5,8,13,21,34,55]
```

Plus och minus med lat evaluering

Nackdelar:
- Kan slöa ner ett program
  (använd då explicit strikt evaluering!)
- Kan slösa på minne: ”löftet” tar plats
- Överraskande för den ovetande?

Fördelar:
- Kan snabba upp ett program
- Undviker onödiga beräkningar.
- Automatiskt!
- Erbjuder smarta uttryckssätt.

Valfri strikthet

- Operatorn (`$!`). Ersätta `f x` med `f $! x`
  tvingar `x` att evalueras först.
- Funktionen `seq :: a -> b -> b`

```
  let x = fkn1
      y = fkn2
  in y ‘seq‘ fkn3 x y
```

- Anpassade funktioner, t.ex. `foldl’`.
- Strikta datatyper, t.ex.

```
  data StrictColor = SRGB !r !g !b
```

  lagrar inga löften