Funktionell programmering

DD1361

Generella egenskaper (funk prog)

- Variabler Nej!
- Tilldelning Nej!
- Sidoeffekter Nej!
- Pekare, referenser Nej!
- Skräpsamling (garbage collection) Ja!

Det funktionella dogmat

Genom att programmera funktionellt blir det:

- mindre att skriva
- färre buggar
- god modularisering
- god kodåtervinning

Haskell: egenskaper

- Strikt funktionellt. Inget ''fusk'' med tilldelning.
- Starkt typsystem. Hittar många fel tidigt, stödjer kodning.
- Typklasser. Överlagrade funktioner kodåtervinning, god abstraktion.
- Lat evaluering. Beräkna bara det som behövs.
- Funktioner är ''first class''. Behandlas som vilka värden som helst.
- Moduler. Storskalig programutveckling, goda bibliotek.

Utvikning: idiom

Wikipedia: An idiom is a low-level pattern that addresses a problem common in a particular programming language. Ett typiskt funktionellt idiom: Loopa över lista/sträng, inte över ett index.

Utvikning: Design patterns

'' Idiom på högnivå'' Wikipedia: A design pattern is a general reusable solution to a commonly occurring problem in software design.

Problemet med I/O

- I/O ej funktionellt
- getChar plockar bort ett tecken från en buffert
- putChar skriver in ett tecken i en buffert.
- Hur åstadkomma I/O utan sideffekter?
- **Lisp mfl:** Fuska! Använd sidoeffekter.

I/O Generellt

```
Pseudokod:
main =
    printStr("Rev: ")
    printStr(reverse(getLine))
```

- **I imperativt program:** Ordning och direkt tillgång till omvärlden.
- **I Haskell:** Ordning oklar, alla funktioner har värden, tillåter ej sidoeffekter (pure functions).

I/O i Haskell

- **Lat I/O:** Låtsas läsa in allt i början.
- **Monadisk I/O:** “Kapsla in världen på ett säkert sätt”
- Speciell notation för I/O.

interact: en god Unix-medborgare

```
interact :: (String -> String) -> IO ()

import Data.Char
main = interact (map toUpper)
```

Läs från `stdin`

Skriv till `stdout`

interact: en god Unix-medborgare

Vad gör detta?

```
module Main where
import Data.List
main =
   interact (concat . sort . lines)
```

interact: en god Unix-medborgare

Vad gör detta?

```
module Main where
import Data.List

newline str = str ++ "\n"

main =
 interact (newline . show .
           length . words)
```

De 10 vanligaste orden

```
module Main where
import Data.List (sortBy, sort, group)
import Data.Char (toLower)

countElems = map (\x ->
                    (head x, length x))

sortBySnd = sortBy (\x y ->
                  snd y ‘compare‘ snd x)

lower = map toLower
```

De 10 vanligaste orden

```
rankWords = sortBySnd . countElems .
            group . sort . words . lower

formatOutput = unlines . map
    (\(str, i) -> str ++ "\t" ++ show i)

main = interact (formatOutput .
                 (take 10) . rankWords)
```

Monadisk I/O

- Särskild notation som döljer problemen
- Monader: funktionellt idiom för
  - sekvensiella beroenden
  - att dölja parametrar
  - förenkla kod

Designprincip för I/O i Haskell

Världen	**Monadisk IO**	**Din kod**
Filer	getChar	myPreparations
Portar	getLine	computeItAll
stdin/stdout	openFile	myFilter
Grafik	readFile	
	isEOF	
	m.m.	

Ansats: Kapsla in världen

Vad vi vill ha:

```
type IO a = World -> (a, World)
```

IO-typer är handlingar: Eng: actions Exempel:

```
getChar :: IO Char
getChar :: World -> (Char, World)

putChar :: Char -> IO ()
putChar :: Char -> World -> ((), World)

isEOF :: IO Bool
getLine :: IO String
```

Till versaler igen

Låtsas läsa hela filen:

```
module Main where
import IO
import Data.Char
main = do {
           str <- getContents;
           putStrLn (map toUpper str);
}
```

Räkna ord igen

```
module Main where
import IO

main =do {
          input <- getContents;
          ws <- return (length
                        (words input));
          putStrLn (show ws);
}
```

Lat I/O: räkna ord i fil

```
module Main where
import IO
main = do { ih <- openFile
                    "input.txt" ReadMode;
            input <- hGetContents ih;
            ws <- return (length
                          (words input));
            putStrLn(show ws);
            hClose(ih)
}
```

Två nya operatorer i monadisk I/O

- `<-` plockar ut ett värde från IO-monaden. Kan skickas till “rena” funktioner utan IO-signatur.
- `return` betyder “sätt in ett värde i IO-monaden”. `return 'A'` skapar värde av typen `IO Char`.
- **Viktigt:** `return` avslutar ej ett `do`-uttryck!

Förenkla koden

blir

```
main = do {
     ih <- openFile "input.txt" ReadMode;
     input <- hGetContents ih;
     putStrLn (show (length
                          (words input)));
     hClose(ih);
}
```

Lat I/O: räkna ord i given fil

```
module Main where
import System.Environment (getArgs)
import IO
main = do { args <- getArgs;
            ih <- openFile (head args)
                                ReadMode;
            input <- hGetContents ih;
            putStrLn(show (length
                           (words input)));
            hClose(ih);
}
```

”Lat I/O farligt”, varför?

```
main = do {
     ih <- openFile "input.txt" ReadMode;
     input <- hGetContents ih;
     hClose(ih);
     putStrLn(show (length
                    (words input)));
}
```

Egen kod i IO-monaden: getLine

```
getLine :: IO [Char]
getLine = do { c <- getChar;
               if c == '\n' then
                  return []
               else
                  do { cs <- getLine;
                       return (c : cs)}
}
```

Paketera resultatet med return

Exempel: Räkna rader och tecken

Indata: Läs från stdin

Utdata: Skriv antalet rader och tecken till stdout

```
module Main where
import IO
main = do {
          (nLines, nChars) <- wc 0 0;
           putStrLn (show nLines ++
           "\t” ++ show nChars)
}
```

Exempel: Räkna tecken och rader

```
wc :: Int -> Int -> IO (Int, Int)
wc nLines nChars=
     do flag <- isEOF
        if flag then
           return (nLines, nChars)
        else
           consumeAndCount nLines nChars
```

Exempel: Räkna tecken och rader

```
consumeAndCount :: Int -> Int ->
                                IO (Int, Int)

consumeAndCount nl nc =
     do { c <- getChar;
          if (c == '\n') then
              wc (1 + nl) (1 + nc)
          else wc nl (1+nc)
}
```

Testa programmet

```
WC> wc 0 0
hubba
^D
WC> main
hubba
^D
1 6
WC>
```

I terminalen:

```
$ ghc -o minwc minwc.hs
$ ./minwc
hubba
bubba
^D
2 12
$
```

Lura Haskell... med unsafePerformIO

- Ett trick för att komma runt monadreglerna: `unsafePerformIO :: IO a -> a`
- Använd inte för F4 eller på tenta...
- Peyton-Jones:
  - Riktigt obekväm I/O, ”Once-per-run I/O”
  - Debugging:

```
visa :: String -> a -> a
visa s x =
    unsafePerformIO (putStrLns s >>
                                  return x)
```

”Haskell i verkligheten”

- `Data.ByteString` för strängar: 1 byte/bokstav istället för ca 12 byte/bokstav
- `Data.Map` för associativa listor. Operationer är O(log n).
- `Data.Array`, en oföränderlig array
- Vanlig (?) `Array` som monad: `Data.Array.ST`