Why functional programming is good ...

...when you like math - examples with Haskell

Jochen Schulz

Georg-August Universität Göttingen

Table of contents

Programming paradigm

- imperative (e.g. C)
- object-oriented (e.g. C++)
- functional (e.g. Haskell)
- (logic)
- (symbolc)

some languages have multiple paradigm

Side effects/pure functions

side effect

Besides the return value of a function it has one or more of the following

- modifies state.
- has observable interaction with outside world.

pure function

A pure function

- always returns the same results on the same input.
- has no side-effect.

also refered to as referential transparency

pure functions resemble mathematical functions.

Functional programming

- emphasizes pure functions
- higher order functions (partial function evaluation, currying)
- avoids state and mutable data (Haskell uses Monads)
- recursion is mostly used for loops
- algebraic type system
- strict/lazy evaluation (often lazy, as Haskell)
- describes more what is instead what you have to do

Table of contents

List comprehensions

Some Math:

$$S = \{x^2 | x \in \mathbb{N}, x^2 < 20\}$$

```
> [ x^2 | x <- [1..10] , x^2 < 20 ]
[1,4,9,16]
```

Ranges (and infinite ranges (don't do this now))

```
> a = [1..5], [1,3..8], ['a'..'z'], [1..]
[1,2,3,4,5], [1,3,5,7], "abcdefghijklmnopqrstuvwxyz"
```

usually no direct indexing (needed)

```
> (head a, tail a, take 2 a, a !! 2)
(1,[2,3,4,5],[1,2],3)
```

Functions: Types and Typeclasses

Types

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Typeclasses

```
factorial :: (Integral a) => a -> a
factorial n = product [1..n]
```

We also can define types and typeclasses and such form spaces.

Pattern matching and laziness

pattern matching defines variables through a pattern for given input data

```
divide :: (Eq a, Num a ,Fractional a) => (a,a) -> a
divide (_,0) = 0
divide (n,d) = n/d
```

_ matches all and drops it.

```
> sieve (x:xs) = x: sieve[ y | y <- xs, y `mod` x /= 0 ]
> primes = sieve [2..]
[2,3,5,7 ...
> take 20 (sieve [2..])
```

x:xs gets the head in x and the tail in xs.

x:y:xs gets the head in x, the second item y and the tail in xs.

and so on.

Higher order functions I

Higher order functions

Functions that have functions as input and/or output.

partial function evaluation

```
> addthree :: (Num a) => a -> a -> a -> a
> addthree a b c = a+b+c
> :t addthree
addthree :: Num a => a -> a -> a -> a
> :t addthree 1 2
addthree 1 2 :: Num a => a -> a
```

Higher order functions II

- map: applies a given function to every element of a list.

```
> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
> let m = map (+) [1,5,3,1,6]
> (m !! 1) 2
7
```

- filter : filters out all elements from a list, for which a given function returns False.

```
> filter (<4) [1,5,3,1,6]
[1,3,1]
```

- lambda functions are anonymous functions and start with \

```
> map (\x -> odd x) [1,5,3,1,6]
[True,True,True,True,False]
```

Higher order functions III

- folds like foldl applies a function to a list and accumlates the results.

```
> foldl (\acc x -> acc + x) 0 [12,4,8]
24
```

- scans like scanl are like foldl, only return the whole progression as a list.

```
> scanl (\acc x -> acc + x) 0 [12,4,8]
[0,12,16,24]
```

There are many more useful functions like this!

Where are the guards?

where: just like math.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

guards: nice syntatic sugar (similar to cases)

```
fibs2 = tailFibs 0 1 0
tailFibs prev1 prev2 start end
          | start == end = next
          | otherwise = tailFibs next prev1 (start + 1) end
          where next = prev1 + prev2
```

Functions: Recursion

```
facrec :: (Integral a) => a -> a
facrec 0 = 1
facrec n = n * facrec(n-1)
```

Tail recursion

When the last statement of a function call is the function itself

```
facrecT :: (Integral a) => a -> a
facrecT 0 = 1
facrecT n = tailfac n 1
  where tailfac 0 a = a
        tailfac n a = tailfac (n-1) (n*a)
```

Monads - or what to do with impurity

Monads...

- are like decorators to single commands: For every command they evaluate some additional code (there is even some similarity to decorators in python).
- are sometimes called *programmable semicolons*.
- enables the handling of side-effect in a controlled way.

Monads - Example

```haskell
mbint :: Int -> Int -> Maybe Int
mbint a b
  | c == 42 = Nothing
  | otherwise = Just c
  where c = a+b
```

- **Maybe**: is a Monad which can be
  - **Just** some Type (here **Int**).
  - **Nothing**.
- **Just**: puts a value in an **Maybe** construct.

```haskell
> mbint 20 1 >>= mbint 20
Just 41
> mbint 20 1 >>= mbint 20  >>= mbint 1 >>= mbint 20
Nothing
```

IO Monad

do-notation

```
mbint 20 1 >>= mbint 20  >>= mbint 1
```

```
donot = do
  d1 <- mbint 20 1
  d2 <- mbint 20 d1
  mbint 1 d2
```

All I/O is impure and Haskell puts it in the IO Monad.

```
hw = do
  putStrLn "Hello World! type your name!"
  name <- getLine
  putStrLn ("Hey " ++ name ++ ", Welcome to Haskell!")
```

Table of contents

Summary

- functional programming is very near to mathematics.
- it helps avoiding side-effects.
- avoids unecessary boilerplate code.

Remark: some languages have some features of functional programming.
So start using it there or directly with Haskell!

Literature

- Learn You a Haskell for Great Good!, M. Lipovača (`http://learnyouahaskell.com/`),
- Real World Haskell, B. O'Sullivan, D. Stewart, J. Goerzen (`http://book.realworldhaskell.org`),
- Prägnante Programmierung in Haskell (German), R. Grimm (`http://www.linux-magazin.de/Ausgaben/2011/06/Haskell`),