Advanced Functional Programming

Chalmers & GU

2013

Patrik Jansson

(slides by Jansson, Norell & Bernardy)

This Course

- Advanced Programming Language Features
  - Type systems
  - Programming techniques
- In the context of Functional Programming
  - Haskell (and a touch of Agda)
- Applications
  - Signals, graphics, web programming
  - Domain Specific Languages

Self Study

- You need to read yourself
- Find out information yourself
- Solve problems yourself

- With a lot of help from us!
  - All information is on the web (soon;-)
  - Discussion board (afp2013 google group)
  - Office hours: Mon. 15-16, (Thu. 13-14)

Organization

- 2 Lectures per week
  - Except for one or two holes (to be determined)
- 3 Programming Assignments (labs)
  - Done in pairs
- 1 Written Exam

Getting Help

- Course Homepage
  - Should have all information
  - Complain if not!
- Discussion Board (afp2013 google groups)
  - Everyone should become a member
  - Discuss general topics
- e-mail teachers (Patrik + Jonas)
  - Organizational help, lectures, etc. (Patrik)
  - Specific help with programming labs (Jonas)
- Office Hours
  - 1-2 times a week, time: Mon. 15-16, (Thu. 13-14)

Recalling Haskell

- Purely Functional Language
  - Referential transparency
- Lazy Programming Language
  - Things are evaluated at most once
- Advanced Type System
  - Polymorphism
  - Type classes
  - ...

Functions vs. Instructions

Compare:

f :: String -> Int

Only the knowledge about the string is needed to understand the result…

vs

g :: String -> IO Int

Anything can be used to compute the result: Reading from files, randomness, user input…!

Functions vs. Instructions

Compare:

f :: String -> Int

But... The “Action” depends solely on the string

vs

g :: String -> IO Int

Moreover, *anything can be modified or changed!*

Programming with IO

```
hello :: IO ()
hello =
  do putStrLn ”Hello! What is your name?”
     name <- getLine
     putStrLn (”Hi, ” ++ name ++ ”!”)
```

Programming with IO

```
printTable :: [String] -> IO ()
printTable xs = prnt 1 xs
 where
  prnt i []     = return ()
  prnt i (x:xs) = do putStrLn (show i ++ ”:” ++ x)
                     prnt (i+1) xs
```

IO actions are *first class.*

```
printTable :: [String] -> IO ()
printTable xs =
  sequence_ [ putStrLn (show i ++ ":" ++ x)
            | (x,i) <- xs `zip` [1..length xs]
            ]
```

sequence_ :: [IO a] -> IO ()

A Function

```
fun :: Maybe Int -> Int
fun mx | mx == Nothing = 0
       | otherwise     = x + 3
 where
  x = fromJust mx
```

Could fail… What happens?

Another Function

```
expn :: Integer -> Integer
expn n | n <= 1    = 1
       | otherwise = expn (n-1) + expn (n-2)
```

```
choice :: Bool -> a -> a -> a
choice False  f  t  =  f
choice True   f  t  =  t
```

```
Main> choice False 17 (expn 99)
17
```

Without delay…

Laziness

- Haskell is a *lazy* language
  - Things are evaluated *at most once*
  - Things are only evaluated when they are needed
  - Things are never evaluated twice

(We will now explore what this means.)

Understanding Laziness

- Use error ”message” or undefined to see whether something is evaluated or not
  - choice False 17 undefined
  - head [3,undefined,17]
  - head (3:4:undefined)
  - head [undefined,17,13]
  - head undefined

Lazy Programming Style

- Separate
  - Where the computation of a value is defined
  - Where the computation of a value happens

Modularity!

When is a Value ”Needed”?

```
strange :: Bool -> Integer
strange False = 17
strange True  = 17
```

```
Main> strange undefined
Program error: undefined
```

An argument is evaluated when a *pattern match* occurs

But also *primitive functions* evaluate their arguments

At Most Once?

```
foo :: Integer -> Integer
foo x = f x + f x
```

f x is evaluated twice

```
bar :: Integer -> Integer -> Integer
bar x y = f 17 + x + y

Main> bar 1 2 + bar 3 4
310
```

Quiz: How to avoid recomputation?

f 17 is evaluated twice

At Most Once!

```
foo :: Integer -> Integer
foo x = fx + fx
 where
  fx = f x
```

So... *bindings* are evaluated at most once...

```
bar :: Integer -> Integer -> Integer
bar x y = f17 + x + y

f17 :: Integer
f17 = f 17
```

... in their scope. So, top level ones are really evaluated at most once!

Infinite Lists

- Because of laziness, values in Haskell can be *infinite*
- Do not compute them completely!
- Instead, only use parts of them

Examples

- Uses of infinite lists
  - take n [3..]
  - xs `zip` [1..]

Example: PrintTable

```
printTable :: [String] -> IO ()
printTable xs =
  sequence_ [ putStrLn (show i ++ ":" ++ x)
            | (x,i) <- xs `zip` [1..]
            ]
```

lengths adapt to each other

Iterate

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
Main> iterate (2*) 1
[1,2,4,8,16,32,64,128,256,512,1024,...
```

Other Handy Functions

```
repeat :: a -> [a]
repeat x = x : repeat x

cycle :: [a] -> [a]
cycle xs = xs ++ cycle xs
```

Quiz: How to define repeat with iterate?

Alternative Definitions

```
repeat :: a -> [a]
repeat x = iterate id x

cycle :: [a] -> [a]
cycle xs = concat (repeat xs)
```

Problem: Replicate

```
replicate :: Int -> a -> [a]
replicate = ?

Main> replicate 5 'a'
"aaaaa"
```

Problem: Replicate

```
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]
group = ?

Main> group 3 "apabepacepa!"
["apa","bep","ace","pa!"]
```

Problem: Grouping List Elements

```
group :: Int -> [a] -> [[a]]
group n = takeWhile (not . null)
        . map (take n)
        . iterate (drop n)
```

. connects "stages" --- like Unix pipe symbol **|**

Problem: Prime Numbers

```
primes :: [Integer]
primes = ?

Main> take 4 primes
[2,3,5,7]
```

Problem: Prime Numbers

```
primes :: [Integer]
primes = sieve [2..]
 where
  sieve (p:xs) = p : sieve [ y | y <- xs, y `mod` p /= 0 ]
```

Commonly mistaken for *Eratosthenes' sieve*[1]

[1] Melissa E. O'Neill, The Genuine Sieve of Eratosthenes. JFP 2008.

Infinite Datastructures

```
data Labyrinth
  = Crossroad
  { what  :: String
  , left  :: Labyrinth
  , right :: Labyrinth
  }
```

How to make an interesting labyrinth?

Infinite Datastructures

```
labyrinth :: Labyrinth
labyrinth = start
 where
  start  = Crossroad "start"  forest town
  town   = Crossroad "town"   start  forest
  forest = Crossroad "forest" town   exit
  exit   = Crossroad "exit"   exit   exit
```

What happens when we print this structure?

Laziness Conclusion

- Laziness
  – Evaluated at most once
  – Programming style
- Do not have to use it
  – But powerful tool!
- Can make programs more ”modular”

Type Classes

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
class Eq a => Ord a where
  (<=) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
```

```
instance Eq Int where ...
instance Eq a => Eq [a] where ...
```

```
sort :: Ord a => [a] -> [a]
```

Type Classes

```
class Finite a where
  domain :: [a]
```

What types could be an instance of this class?

Can you make functions instance of Eq now?

Focus of This Course

- Libraries ~= Little Languages
  – Express and solve a *problem*
  – in a *problem domain*
- Programming Languages
  – General purpose
  – *Domain-specific*
    - *Description languages*
- Embedded Language
  – A little language implemented as a library

E.g. JavaScript

E.g. HTML, PostScript

Little languages

Typical Embedded Language

- Modelling elements in problem domain
- Functions for *creating* elements
  - *Constructor functions*
- Functions for *modifying* or *combining*
  - *Combinators*
- Functions for observing elements
  - *Run functions*