



Functional Programming

Lecture 3

Cezary Kaliszyk Jonas Schöpf
Christian Sternagel Vincent van Oostrom

Department of Computer Science

Overview

- Module Basics
- Lists and Strings
- Recursive Functions
- Example – Printing a Calendar

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, **list comprehensions**, lists, **modules**, pattern matching, polymorphism, property-based testing, reasoning about functional programs, **recursive functions**, sets, **strings**, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Structuring Code into Modules

- **note:** separate namespaces for functions and types
- split source code into several files
- for each module `Module` create file `Module.hs`
- module names always start with uppercase letters
- start module by **module header** with optional **export list**
`module Module (...) where`
- export list is list of functions and types visible outside
- without export list all functions and types visible

Example

```
module Stack where
type Stack a = [a]
empty = []
push = (:)
pop s = (head s, tail s)
```

Type Synonyms

- `type Stack a = [a]` is a **type synonym**
- gives alternative name for `[a]` (no new types involved)
- afterwards, both names may be used interchangeably

Type Signatures

- every function `f` may be preceded by a **type signature** `f :: T`, stating that `f` is of type `T`
- good for documentation purposes

Example

```
push :: a -> Stack a -> Stack a
push = (:)
```

- note the partial application of `(:)`
- this is equivalent to `push x s = x : s`

Note – Imports

- `import M` imports **all** functions and types exported by module `M`
- we may restrict to `f1, ..., fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything **except** for functions `f1` to `fN`
- `import qualified M` allows us to access all functions exported by `M` using prefix “`M.`”
- `import qualified M as N`, similar to `import qualified M` but additionally rename `M` to `N`

Examples

- `import Stack`
- `import Stack (push, pop)`
- `import Stack hiding (pop)`
- `import qualified Stack`
- `import qualified Stack as S`

Strings are Lists

- type `String` is type synonym for `[Char]`
- that is, strings are lists of characters
- consequently, all list functions apply also to `Strings`

Some Implications

- `[]` is same as `""` for strings
- `['h','e','l','l','o']` is same as `"hello"` for strings

Useful Functions on Strings

- `lines :: String -> [String]` – breaks string at newlines
- `unlines :: [String] -> String` – concatenates strings, inserting newlines
- `words :: String -> [String]` – breaks string at white space
- `unwords :: [String] -> String` – concatenates strings, inserting spaces

Interlude – Function Composition

- in mathematics $f \circ g$ usually denotes applying f after g
- more precisely, $(f \circ g)(x) = f(g(x))$
- only possible if output of g compatible with input of f , that is, $f: B \rightarrow C$ and $g: A \rightarrow B$
- in Haskell: `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- try `":info (.)"` in GHCi

Examples

- `map (f . g) xs` – to every element of `xs`, first apply `g` and then `f`
- equivalent to `map f (map g xs)`
- what are the results of `unwords . words` and `words . unwords`?

List Comprehensions – Generators

- in mathematics **set comprehensions** can be used to construct new sets from existing sets
- e.g., $\{x^2 \mid x \in \{1, \dots, 5\}\}$ produces $\{1, 4, 9, 16, 25\}$
- in Haskell: `[x2 | x <- [1..5]]`
- here, `x <- [1..5]` is called **generator**
- there may be more than one generator, e.g.,
`[(x, y) | x <- xs, y <- xs]` (all pairs of elements from `xs`)
- order** is important: `[(x,y) | x <- [1,2], y <- ["a","b"]]`
`= [(1,"a"),(1,"b"),(2,"a"),(2,"b")]`

Examples

- `length xs = sum [1 | _ <- xs]`
- `firsts ps = [x | (x, _) <- ps]`
- `concat xss = [x | xs <- xss, x <- xs]`

Basic Concepts

- functions may be defined in terms of other functions
`factorial :: Int -> Int`
`factorial n = product [1..n]`
- or in terms of themselves (that is, recursively)
`factorial n`
`| n <= 1 = 1`
`| otherwise = n * factorial (n - 1)`
- note:** `factorial` does not loop forever, since at some point its argument will be `1` or smaller (its **termination condition**)
- recipe for defining recursive functions
 - define type (e.g., `product :: [Int] -> Int`)
 - enumerate cases (e.g., `[]` and `x:xs`)
 - define simple cases (e.g., `product [] = 1`)
 - define other cases (e.g., `product (x:xs) = x * product xs`)
 - generalize and simplify (e.g., `product :: Num a => [a] -> a` and `product = foldr (*) 1`)

List Comprehensions – Guards

- filter values before generating result
- e.g., $\{x^2 \mid x \in \mathbb{N}, x > 5\}$
- in Haskell: `[x2 | x <- xs, x > 5]`; square every number in `xs` that is greater than `5`

Examples

- `[x | x <- [1..10], even x]`
- `findAll k t = [v | (k', v) <- t, k == k']`
- `factors n = [x | x <- [1..n], n `mod` x == 0]`
- `primes = [n | n <- [1..], factors n == [1,n]]`

Example – drop

- define type: `drop :: Int -> [a] -> [a]`
- enumerate cases:
`drop 0 [] = []`
`drop 0 (x:xs) = x : xs`
`drop n [] = []`
`drop n (x:xs) = drop (n - 1) xs`
- define simple cases:
`drop 0 [] = []`
`drop 0 (x:xs) = x : xs`
`drop n [] = []`
- define other cases:
`drop n (x:xs) = drop (n - 1) xs`
- generalize and simplify:
`drop :: Integer -> [a] -> [a]`
`drop n xs | n <= 0 = xs`
`drop _ [] = []`
`drop n (_:xs) = drop (n - 1) xs`

Example – init

- define type: `init :: [a] -> [a]`
- enumerate cases:
`init (x:xs) =`
- define simple cases:
`init (x:xs) | null xs = []`
- define other cases:
`| otherwise = x : init xs`
- generalize and simplify:
`init :: [a] -> [a]`
`init [] = []`
`init (x:xs) = x : init xs`

The Picture Analogon

pictures:

- atomic part: **pixel**
- **height** and **width**
- **white** pixel

strings:

- atomic part: **character**
- number of **rows** and **columns**
- **blank** character

Auxiliary Types

```
type Height = Int
type Width  = Int
type Picture = (Height, Width, [[Char]])
```

- consider `(h, w, rs)`
- `rs :: [[Char]]` – “list of rows”
- invariant 1: length of `rs` is height `h`
- invariant 2: all rows (that is, lists in `rs`) have length `w`

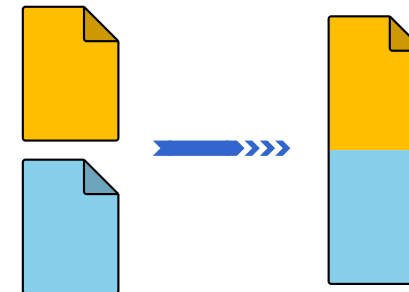
Printing a Calendar

- given a month and a year, print the corresponding calendar
- separate construction phase (computation of days, leap year, ... in file `Calendar.hs`) from printing
- we concentrate on printing, assuming machinery for construction

Example – November 2018

```
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

Stacking 2 Pictures Above Each Other



above

```
above :: Picture -> Picture -> Picture
(h, w, css) `above` (h', w', css')
  | w == w'      = (h + h', w, css ++ css')
  | otherwise     = error "above: different widths"
```

Stacking Several Pictures Above Each Other

```
stack :: [Picture] -> Picture
stack = foldr1 above
```

Notes

- `error :: String -> a` indicates a runtime error given as string
- `foldr1` – special version of `foldr`, without base value (does not work on empty list)


```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = x `f` foldr1 f xs
```

Combining 2 Lists via a Function

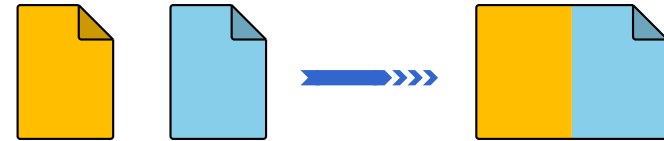
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `zipWith f [x1, ..., xm] [y1, ..., yn] = [x1 `f` y1, ..., xmin{m,n} `f` ymin{m,n}]`
- specialization `zip :: [a] -> [b] -> [(a, b)]`

```
zip = zipWith (,)
```

Examples

- `zip [1,2,3] ['a','b'] = [(1,'a'),(2,'b')]`
- `zipWith (*) [1,2] [3,4,5] = [1*3,2*4] = [3,8]`
- `zipWith drop [1,0] ["a","b"] = [drop 1 "a",drop 0 "b"] = ["","b"]`

Spreading 2 Pictures Beside Each Other



beside

```
beside :: Picture -> Picture -> Picture
(h, w, css) `beside` (h', w', css')
  | h == h' = (h, w + w', zipWith (++) css css')
  | otherwise = error "beside: different heights"
```

Spreading Several Pictures Beside Each Other

```
spread :: [Picture] -> Picture
spread = foldr1 beside
```

Tiling Several Pictures

```
tile :: [[Picture]] -> Picture
tile = stack . map spread
```

Creating Pictures

- single 'pixels'


```
pixel :: Char -> Picture
pixel c = (1, 1, [[c]])
```
- rows


```
row :: String -> Picture
row r = (1, length r, [r])
```
- blank


```
blank :: Height -> Width -> Picture
blank h w = (h, w, blanks)
  where
    blanks = replicate h (replicate w ' ')
```

Note

`replicate :: Int -> a -> [a]` – replicates single element given number of times

Constructing a Month

- assume function `monthInfo :: Int -> Int -> (Int, Int)`, returning the first weekday of the month together with the number of days for the month
- where days are 0 (Sunday), 1 (Monday), ...
- e.g., `monthInfo 11 2018 = (4, 30)`, meaning that the first weekday of November 2018 is a Thursday and the month has 30 days

```
daysOfMonth :: Int -> Int -> [Picture]
```

```
daysOfMonth m y =
  map (row . rjustify 3 . pic) [1 - d .. 42 - d]
  where
    (d, t) = monthInfo m y
    pic n = if 1 <= n && n <= t then show n else ""
```

```
month :: Int -> Int -> Picture
```

```
month m y = tile $ groupsOfSize 7 $ daysOfMonth m y
```

variant of function application with lowest priority to avoid parentheses

Missing Functions

- `rjustify` – right-justify given text inside box of given width


```
rjustify :: Int -> String -> String
rjustify n xs
  | l <= n = replicate (n - l) ' ' ++ xs
  | otherwise = error ("text of length " ++ show l ++
    " does not fit in box of width " ++ show n)
  where l = length xs
```
- `groupsOfSize` – split list into sublists of given length


```
groupsOfSize :: Int -> [a] -> [[a]]
groupsOfSize n xs =
  if null ys then []
  else ys : groupsOfSize n zs
  where
    (ys, zs) = splitAt n xs
```
- `splitAt` :: `Int -> [a] -> ([a], [a])` – split list at given position

Printing a Month

- transform a `Picture` into a `String`

```
showPic (_, _, css) = unlines css
```
- print result of `month m y`

```
printMonth m y =
  putStr $ showPic $ above weekdays $ month m y
  where
    weekdays = row " Su Mo Tu We Th Fr Sa"
```
- putting it all together in `Cal.hs`:


```
module Main where
import System.Environment -- for getArgs
...
main = do
  args <- getArgs
  case args of
    [m, y] -> printMonth (read m) (read y)
    _       -> error "expecting month and year"
```
- compile: `ghc --make Cal` run: `./Cal 11 2018`

Homework (for November 16th)

1. Read Chapter 3 of *Real World Haskell*.
2. Implement a module `Queue` for FIFO queues with interface:


```
empty :: Queue a
isEmpty :: Queue a -> Bool
enqueue :: a -> Queue a -> Queue a
dequeue :: Queue a -> (a, Queue a)
```
3. Use list comprehension to implement a function that generates the list of all triples $(x, y, z) :: (Int, Int, Int)$ such that $x^2 + y^2 = z^2$ and $1 \leq x \leq y < z \leq n$ for given n .
4. Implement two recursive functions


```
prefixes, suffixes :: [a] -> [[a]]
```

 that compute all prefixes and suffixes of a given list, respectively.

Examples:

- `prefixes [1,2,3] = [[], [1], [1,2], [1,2,3]]`
- `suffixes [1,2,3] = [[1,2,3], [2,3], [3], []]`

Homework (for November 16th, continued)

5. Implement a recursive function `sublists :: [a] -> [[a]]` that computes all sublists of a given list in increasing order of length.

Example: `sublists [1,2,3] =`

`[[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]]`

6. Use the `Picture` module to implement a program `Year` that, given a year, prints a calendar for the whole year where months are distributed over a given number of columns. For example, the first row for the year 2018 with 3 columns should look similar to:

2018

January							February							March						
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6					1	2	3					1	2	3
7	8	9	10	11	12	13	4	5	6	7	8	9	10	4	5	6	7	8	9	10
14	15	16	17	18	19	20	11	12	13	14	15	16	17	11	12	13	14	15	16	17
21	22	23	24	25	26	27	18	19	20	21	22	23	24	18	19	20	21	22	23	24
28	29	30	31				25	26	27	28				25	26	27	28	29	30	31