

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 3



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, implementing a type checker, 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

Overview

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

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**
- just gives an alternative name for `[a]`
- 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`

Strings are Lists

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

Some Implications

- `[]` is the same as `""` for strings
- `['h','e','l','l','o']` is the 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, separated by 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 is compatible with input of f , that is, $f: B \rightarrow C$ and $g: A \rightarrow B$
- in Haskell: $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
- try “:info (.)” in GHCi

Examples

- `map (f . g) xs` – on 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 a **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: rightmost generators are evaluated first

Examples

- `length xs = sum [1 | _ <- xs]`
- `firsts ps = [x | (x, _) <- ps]`
- `flatMap f xs = [y | x <- xs, y <- f x]`

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]`
- `find 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]]`

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`)

Example – drop

- define type: `drop :: Int -> [a] -> [a]`

- enumerate cases:

`drop 0 [] =`

`drop 0 (x:xs) =`

`drop n [] =`

`drop n (x: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
```

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 – October 2017

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	31				

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

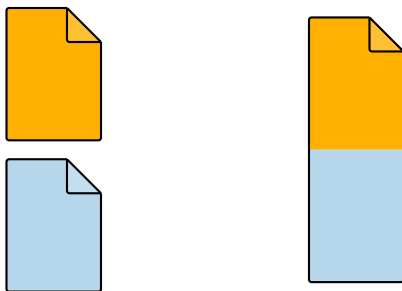
```
type Height = Int
```

```
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**

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

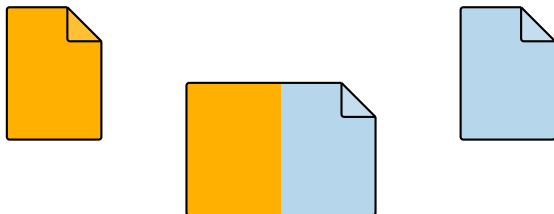
- `error :: String -> a`, indicates a runtime error, given as string
- `foldr1` – special version of `foldr`, without base value (does not work on empty lists)

```
foldr1 :: (a -> a -> a) -> [a] -> a
```

```
foldr1 f [x]      = x
```

```
foldr1 f (x:xs) = x `f` foldr1 f xs
```


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
```

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"]`

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 ' ')
```

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 10 2017 = (0, 31)`, meaning that the first weekday of October 2017 is a Sunday and the month has 31 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 $ group 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 = replicate (n - length xs) ' ' ++ xs
```

- `group` – split list into sublists of given length

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

```
group n xs =
```

```
  if null ys then []
```

```
  else ys : group n zs
```

```
  where
```

```
    (ys, zs) = splitAt n xs
```

- `tile` – tile a list of lists of pictures

```
tile :: [[Picture]] -> Picture
```

```
tile = stack . map spread
```

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 10 2017`

Exercise Preparation – Caesar Cipher

- a **Caesar cipher** encodes text by replacing each letter by another one, some fixed positions (the **key**) down the alphabet
- for example, encoding `hello` with a key of 2, yields `jgnnq`
- in the following we restrict to lowercase letters
- approximate letter frequency list for English

```
tableEn = [8.2,1.5,2.8,4.3,12.7,2.2,2.0,6.1,7.0,
           0.2,0.8,4.0,2.4,6.7,7.5,1.9,0.1,6.0,
           6.3,9.1,2.8,1.0,2.4,0.2,2.0,0.1]
```

- chi-square statistic

$$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

- where *os* is list of observed frequencies
- and *es* list of expected frequencies (e.g., `tableEn` for English)
- the lower chi-square, the better the match between *os* and *es*

Exercises (for November 3rd)

1. Read Chapter 3 of [Real World Haskell](#).
2. Implement a function `rotate :: Int -> [a] -> [a]` that rotates the elements of a list to the left (wrapping around at the start of the list). For example, `rotate 3 [1,2,3,4,5] = [4,5,1,2,3]`.
3. Implement a function `encode :: Int -> String -> String` that applies a Caesar cipher, e.g.,
`encode 10 "hello world" = "rovvy gybvn"`. (Note that decoding is just encoding with the negated key.)
4. Implement a function `freqs :: String -> [Double]` that produces a frequency list for the 26 lowercase letters. For example,
`freqs "aaab" = [75.0,25.0,0.0,...,0.0]`.
5. Implement the chi-square statistic by a function `chisqr :: [Double] -> [Double] -> Double`, taking two frequency lists.
6. Implement a function `crack :: String -> String` that is able to break the ciphertext `"zosk lux xkgr ktixevzout"`. You may use all the previous functions and `tableEn`.

Hints

- function `f` from module `M` is available through `M.f`
- in order to use `f` without qualifier `import M` at start of file
- converting between integers and characters:
 - `Data.Char.chr :: Int -> Char`
 - `Data.Char.ord :: Char -> Int`
- converting from integer to float: `fromIntegral`