

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 12



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

- Lazyness and Infinite Data Structures
- Examples of (Infinite) Lazyness

Lazyness - Motivation

- only compute values needed for final result
- avoid computing same value twice (memoization)

Example

- in the program

```
f1 x = x + 1
```

```
f2 x = f2 x -- nonterminating
```

```
main = do
```

```
  input <- getLine
```

```
  let i = read input
```

```
  print (head [f1 i, f2 i])
```

- value of `f2 i` not needed
- however, without lazyness program would not terminate

Lazyness and Infinite Data Structures Facilitate Modularity

- separation of concerns
- use potentially infinite data structures
- write small functions with specific tasks

Find Index of First Number in Range Satisfying Property

- in Haskell (only using Prelude functions!)

```
snd . head . filter (p . fst) $ zip [m..n] [0..]
```

- (lazy) evaluation

```
snd . head . filter ((==1) . fst) $ zip [1..n] [0..]
= ... $ (1,0) : zip [2..n] [1..]
= snd . head $ (1,0) : filter ((==1) . fst) (...)
= snd (1,0)
= 0
```

- without lazyness (ignoring nontermination of `[0..]`) three list traversals

Watch out for Memory Leaks

with lazyness even tail recursive programs may run out of memory

Function (Tail Recursive)

```
length' acc []      = acc
length' acc (_:xs) = length' (acc + 1) xs
```

Evaluation

```
length' 0 [1,2,3,4]
= length' (0+1) [2,3,4]
= length' (0+1+1) [3,4]
= length' (0+1+1+1) [4]
= length' (0+1+1+1+1) []
= (0+1+1+1+1)
```

Being Strict – The seq Function

- type `seq :: a -> b -> b`
- `x `seq` y` reduces `x` to WHNF and returns `y`

Function (still Tail Recursive)

```
length' acc [] = acc
length' acc (_:xs) = acc `seq` length' (acc + 1) xs
```

Strict Folding of Lists – Data.List.foldl'

```
foldl' _ b [] = b
foldl' f b (x:xs) = c `seq` foldl' f c xs
  where
    c = f b x in
```

Example

```
length' = foldl' (const . (+1))
```

Example 1 – Fibonacci Numbers (this time starting from 0)

$$\text{fib}(i) = \begin{cases} 0 & \text{if } i \leq 0 \\ 1 & \text{if } i = 1 \\ \text{fib}(i - 1) + \text{fib}(i - 2) & \text{otherwise} \end{cases}$$

Sequence

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 ...

One Way of Computing Fibonacci Numbers

starting at 0		0	1	1	2	3	5	8	13	21	...
starting at 1		1	1	2	3	5	8	13	21	...	
(+)		1	2	3	5	8	13	21	34	...	

Ingredients

- function to shift sequence to left
- function to add two sequences

Fibonacci Numbers in Haskell

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Example 2 – The Sieve of Eratosthenes

start with list of all natural numbers (from 2 on)

1. mark first element x as prime
2. remove all multiples of x
3. go to Step 1

The Sieve in Haskell

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

Example 3 - Lazy Shenanigans

- task – replace all elements of list by its maximum in single traversal
- in Haskell

```
replaceAllByMax xs = ys
```

```
  where
```

```
    (m, ys) = maxAndReplaceAll m xs
```

```
    maxAndReplaceAll c =
```

```
      foldr (\x (m, ys) -> (max x m, c : ys)) (0, [])
```

Resulting Data Dependencies for [1,2,3]

