universität innsbruck

WS 2018/2019

Functional Programming

Lecture 11

Cezary Kaliszyk Jonas Schöpf
Christian Sternagel Vincent van Oostrom

Department of Computer Science

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

Overview

- Laziness and Infinite Data Structures
- Examples of (Infinite) Laziness
- Recommendations

Laziness – Motivation

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

Example

- in the program

```
main = do
  let f1 x = x + 1
      f2 x = f2 x  -- a nonterminating function
  input <- getLine
  let i = read input
  print $ head [f1 i, f2 i]
```

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

Laziness and Infinite Data Structures Facilitate Modularity

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

Find Index of First List Element Satisfying Property

- function `firstIndex :: (a -> Bool) -> [a] -> Int`
- in Haskell (only using Prelude functions!) `firstIndex p =`
  `fst . head . filter (p . snd) . zip [0..]`
- (lazy) evaluation

```
fst . head . filter ((==1) . snd) $ zip [0..] [1..9]
  = ... $ (0,1) : zip [1..] [2..9]
  = fst . head $ (0,1) : filter ((==1) . snd) (...)
  = fst (0,1)
  = 0
```

- without laziness (ignoring nontermination of `[0..]`) three list traversals (generation of `[0 .. length xs]`, `zip`, and `filter`)

Watch out for Memory Leaks

with laziness 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
```

Example

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

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

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

Sequence

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

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	...	

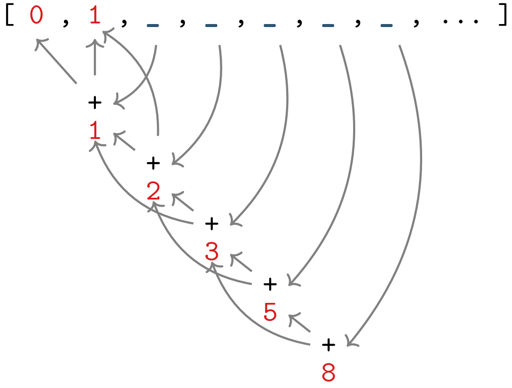
Ingredients

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

Fibonacci Numbers in Haskell

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

Resulting Data Dependencies



Example 2 – A Variation of the “Sieve of Eratosthenes”

Goal

generate list of all prime numbers

Procedure

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

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

Example 3 – Dropping n Elements from Tail of List

- naive solution: `dropEnd n xs = take (length xs - n) xs`
- problem: does not work on infinite lists
  (nontermination even if only finite prefix of result is used)
- another attempt: `dropEnd n = reverse . drop n . reverse`
- problem: does not work on infinite lists and requires three traversals
- solution: `dropEnd n xs = zipWith const xs (drop n xs)`

Why does this work?

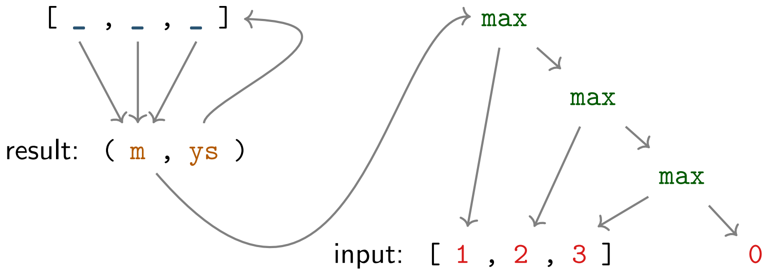
- result of `zipWith f xs ys` has length of shorter of `xs` and `ys`
- `zipWith const xs ys` ignores elements of `ys`
- `zipWith const` $[x_1, \ldots, x_m]$ $[y_1, \ldots, y_n] =$
  $[x_1, \ldots, x_{\min\{m,n\}}]$

Example 4 – 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]



Courses Related to FP in SS2019

- Term Rewriting (VO2 + PS1)

Bachelor Projects Related to FP

- Fast Multiset-Comparison
- Domain Specific Language for Machine Learning Theorem Proving
- Intelligent search-time proof strategy selector for Isabelle/HOL
- Meta-Tool Based Proof Search in Coq
- Alternative Approaches to Termination Analysis
- Maximal Interpretations
- A Certified Decision Procedure for Termination of Right-Ground Term Rewrite Systems
- Commutation of Term Rewrite Systems
- Unification in Booltool
- ...