universität innsbruck

WS 2026/2027



Advanced Functional Programming

Week 1 – Organisation and Introduction, Strict- and Lazy-Evaluation

René Thiemann

Department of Computer Science

Organization

Organization of Course

- LV-Number: 703139
- lecturer: René Thiemann
  consultation hours: Tuesday 10:15 – 11:15 in 3M09 (ICT building)
- time and place: Wednesday, 12:00 – 14:30 in RR 12
- course website: `https://cl-teaching.uibk.ac.at/26w/afp/`
- lecture will be in English
- slides are available online and contain links
- modus: VU 3
  - 3 hours per week
  - attendance is obligatory
  - VU: lecture and exercises combined
    - today: just lecture
    - from next week onwards: first, presentation of exercises; afterwards lecture

Schedule

- detailed schedule: see website
- special dates
  - today: just lecture
  - January 20, Q & A session, no new content
  - January 27: exam

Evaluation

- 50 % exercises + 50 % exam
- exam on January 27, closed book, no electronic devices
- exercises will be handed out every week
- mark solved exercises and upload Haskell sources in OLAT
- deadline in OLAT: Tuesday, 4pm
- definition of solved:
  - 100 % solutions are not required, but a significant part of tasks should have been solved
  - capability to explain your solution to everyone in this room
- positive evaluation: get in total at least 50 % of points

AI Policy for Coursework

"Riding a taxi is not a good way to learn how to drive."

Principles

- Students are responsible for all content they deliver (spoken, in writing, or code).
- The 'traffic light' system recognises that policies differ between courses. Its purpose is to provide students with certainty.
- Teachers have the authority to redefine any aspect of the policy and its enforcement.
- Failure to comply with the policy may result in failing the course.

Definition of the traffic light



AI use is prohibited.



AI is permitted, but must be declared and comply with other restrictions (specific details to be provided by the course teacher).



AI use is allowed.



AI use is recommended.

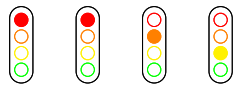
AI Policy for This Course

The traffic light applies to different types of AI use in the following way:



Creative tasks

- Ideation
- Text generation
- Creation of presentation slides
- Production of multimedia content



Presentation quality assurance

- Spell checking
- Text editing (without changing the meaning)
- Improvement/harmonisation of graphical elements

Professional core tasks

- Programming
- Data analysis
- Proof generation

Supporting tasks

- Literature search
- Text summarisation and translation (without directly adopting the output)
- Not allowed:Generation of solution hints
- Proof verification

Literature

- slides and exercises
  - no other topics will appear in exam . . .
  - . . . but topics need to be understood thoroughly
    - read and write functional programs
    - apply presented techniques on new examples
    - not only knowledge reproduction
- Bryan O'Sullivan, John Goerzen and Don Stewart. Real World Haskell, O'Reilly.
- . . . see slides

Prerequisites: Basic Knowledge of Functional Programming

- knowledge on lists, trees and other algebraic data types
- knowledge on recursive function definitions
- basic knowledge on type-classes (`Eq`, `Ord`, `Show`, `Num`)
- basic knowledge on programming with higher-order functions (`map`, `filter`, `foldr`, `.`, partial application, ...)
- basic knowledge on `IO` (separate pure from IO-computations, do-notation, ...)

Overview of Course

- today: programming
- next two weeks: theory to better understand Haskell's type system
- remaining course: programming
- details: see AFP website

Strict- and Lazy-Evaluation

Example

Consider

- program `square x = x * x`, and
- expression `square (3 + 2)`

Different Ways to Apply Equations

- strict/innermost: evaluate arguments before doing a function application

  `square (3 + 2) = square 5 = 5 * 5 = 25`

- non-strict/lazy: apply program equation as soon as possible

  `square (3 + 2) = (3 + 2) * (3 + 2) = 5 * 5 = 25`

  where the sub-expression 3+2 is shared and hence, only evaluated once

Values and Thunks

- value: a fully evaluated term, e.g., `5`, `"hello"`, `[1,2,3]`
- thunk: a term that needs further evaluation, e.g., `2 + 3`, `"hel" ++ "lo"`, ...
- strict/innermost: evaluate arguments to values before invoking function application
- non-strict/lazy: arguments can be passed as values or as thunks
- consequences
  - strict/innermost is easier to implement; takes less space per cell
  - non-strict/lazy includes overhead when working with thunks; admits new kinds of programming styles
- ML and OCaml use a strict/innermost evaluation strategy
- Haskell uses non-strict/lazy as default evaluation strategy; strict/innermost on demand
  - offer strict and lazy folding functions
  - offer strict and lazy arrays
  - offer strict and lazy dictionaries
  - enforce strictness via `seq`, via strict datatypes, ...

Example (`foldl` and `foldl'`)

```
foldl, foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl f y [] = y
foldl f y (x : xs) =
  let z = f y x
  in foldl f z xs

foldl' f y [] = y
foldl' f y (x : xs) =
  let z = f y x
  in seq z $ foldl' f z xs
```

Remark

- `seq x y` returns `y` after evaluating `x` to weak-head normal form (WHNF), i.e., after outermost constructor has been computed
- example:
  `(let xs = take 2 [5..] in seq xs xs) = ... = 5 : take (2 - 1) [5 + 1..]`

Example (Lazy Evaluation via `foldl`)

```
foldl f y [] = y
foldl f y (x : xs) =
  let z = f y x
  in foldl f z xs

  foldl (+) 0 [1,2,3,4,5,6]
= let z1 = 0 + 1 in foldl (+) z1 [2,3,4,5,6]
= let z1 = 0 + 1 in let z2 = z1 + 2 in foldl (+) z2 [3,4,5,6] = ...
= let z1 = 0 + 1 in let z2 = z1 + 2 in let z3 = z2 + 3 in
  let z4 = z3 + 4 in let z5 = z4 + 5 in let z6 = z5 + 6 in foldl (+) z6 []
= let z1 = 0 + 1 in let z2 = z1 + 2 in let z3 = z2 + 3 in
  let z4 = z3 + 4 in let z5 = z4 + 5 in let z6 = z5 + 6 in z6
= ((((0 + 1) + 2) + 3) + 4) + 5) + 6
= ... = 21
```

Linear space requirement!

Example (Strict Evaluation via `foldl'`)

```
foldl' f y [] = y
foldl' f y (x : xs) =
  let z = f y x
  in seq z $ foldl' f z xs

  foldl' (+) 0 [1,2,3,4,5,6]
= let z1 = 0 + 1 in seq z1 $ foldl' (+) z1 [2,3,4,5,6]
= let z1 = 1 in seq z1 $ foldl' (+) z1 [2,3,4,5,6]
= foldl' (+) 1 [2,3,4,5,6]
= let z2 = 1 + 2 in seq z2 $ foldl' (+) z2 [3,4,5,6]
= let z2 = 3 in seq z2 $ foldl' (+) z2 [3,4,5,6]
= foldl' (+) 3 [3,4,5,6]
= ...
= foldl' (+) 21 []
= 21
```

Constant space requirement!

Example (Sometimes `foldl` is Preferable)

```
mulNS x 0 = 0
mulNS x y = x * y

-- compare
foldl mulNS 1 [3,6,undefined,0,7]
-- with
foldl' mulNS 1 [3,6,undefined,0,7]

-- result: only the former succeeds
```


Use `seq` Carefully

- `seq` forces only an evaluation, if `seq` itself is at a position which should be evaluated
- usually, put `seq` on the outside

```
f 0 y = ...
f x y = let z = ... in z `seq` f (x - 1) z   -- evaluation of z to WHNF
f x y = let z = ... in f (x - 1) (z `seq` z) -- no effect
f x y =
  let x1 = x - 1;
      z = ...
  in x1 `seq` z `seq` f x1 z      -- evaluate both x1 and z to WHNF
                                  -- here: useless for x1
```


Benefits from Lazy Evaluation: Modularity

- composing several programs can work out nicely with lazy evaluation, where strict evaluation is not performant
- example: compute the ten smallest elements in a list `xs`
- lazy approach: `take 10 (sort xs)`
  - approach can be efficient, since due to laziness, not all of `sort xs` has to be computed (efficiency depends on utilized sorting algorithm)
- strict approach
  - `take 10 (sort xs)` is inefficient to evaluate, if `xs` is long
  - writing separate program from scratch requires work

Programming with Lazy Evaluation

- task
  - replace all elements in a non-empty list by the minimum in the list ...
  - ... with only one list-traversal
- solution

```
findMinRepl :: Ord b => a -> [b] -> (b, [a])
findMinRepl r [x] = (x, [r])
findMinRepl r (x : xs) = case findMinRepl r xs of
  (m, ys) -> (min m x, r : ys)

replAllByMin :: Ord a => [a] -> [a]
replAllByMin xs =
  let (m, ys) = findMinRepl m xs
  in ys
```

- trick: `m` is evaluated lazily in `replAllByMin`

Programming with Lazy Evaluation

```
findMinRepl r [x] = (x, [r])
findMinRepl r (x : xs) = case findMinRepl r xs of
  (m, ys) -> (min m x, r : ys)
replAllByMin xs = let (m, ys) = findMinRepl m xs in ys

  replABM [2,6,1]
= let (m, ys) = fMR m [2,6,1] in ys
= let (m, ys) = case fMR m [6,1] of (m1, ys1) -> (min m1 2, m : ys1) in ys
= let (m, ys) = case (case fMR m [1] of (m2, ys2) -> (min m2 6, m : ys2))
     of (m1, ys1) -> (min m1 2, m : ys1) in ys
= let (m, ys) = case (case (1, [m]) of (m2, ys2) -> (min m2 6, m : ys2))
     of (m1, ys1) -> (min m1 2, m : ys1) in ys
= let (m, ys) = case (min 1 6, [m, m])
     of (m1, ys1) -> (min m1 2, m : ys1) in ys
= let (m, ys) = (min (min 1 6) 2, [m, m, m]) in ys
= [min (min 1 6) 2, min (min 1 6) 2, min (min 1 6) 2] = ... = [1, 1, 1]
```


Programming with Lazy Evaluation – Lazy Arrays

- several container data structures (arrays, dictionaries, ...) are provided both in a strict and in a lazy variant in Haskell libraries
- advantage of strict versions
  - no overhead from working with thunks
  - less memory consumption, no boxing and unboxing of values
- advantage of lazy versions
  - lazy initialization becomes possible:
    already consume parts of array during construction (similar to `m` in previous example)
- documentation
  - https://hackage.haskell.org/package/array/docs/Data-Array-IArray.html
  - https://hackage.haskell.org/package/array/docs/Data-Array-Unboxed.html

Example with Lazy Initialization

```
import qualified Data.Array.IArray as L  -- lazy, boxed, immutable arrays

fibsLazyArray :: Int -> [Integer]
fibsLazyArray n =
  let a :: L.Array Int Integer
      a = L.genArray (0,n)
         (\ i -> if i <= 1 then 1 else a L.! (i - 1) + a L.! (i - 2))
  in L.elems a

-- lazy approach: in order to construct array a, we already use it

-- index types Ix might be Int, Integer, Char, (Int, Int), ...
-- L.genArray :: (IArray a e, Ix i) => (i, i) -> (i -> e) -> a i e
-- (L.!) :: (IArray a e, Ix i) => a i e -> i -> e
-- L.elems :: (IArray a e, Ix i) => a i e -> [e]
```


Lazy Initialization does Not Work with Strict Arrays

```
import Data.Array.Unboxed as S -- strict, unboxed arrays
-- UArray can store elements of type Int, Word32, ...,
--   but not Integer, String, ...
import Data.Word (Word64)

fibsStrictArray :: Int -> [Word64]
fibsStrictArray n =
  let a :: S.UArray Int Word64
      a = S.genArray (0,n)
         (\ i -> if i <= 1 then 1 else a S.! (i - 1) + a S.! (i - 2))
  in S.elems a

-- computation of fibsStrictArray 10 does not succeed

-- similar interface in comparison to lazy arrays
-- S.genArray :: (S.IArray a e, S.Ix i) => (i, i) -> (i -> e) -> a i e
```

Another Example for Lazy Containers: Dynamic Programming

- bracketing problem
  - given is list of $n-1$ compatible matrices $A_0\ A_1\ \ldots A_{n-2}$
  - in fact, only the dimensions of A_i are given: $[a_0,\ldots,a_{n-1}]$, A_i has dimension $a_i \times a_{i+1}$
  - task: figure out cheapest way to multiply all matrices, e.g., $(A_0A_1)(A_2(A_3A_4))$
  - algorithm computes optimal costs to multiply $A_i \ldots A_j$
  - $cost(i,i) = 0$
  - $cost(i,j) = \min\{cost(i,k) + cost(k+1,j) + \underbrace{a_i a_{k+1} a_{j+1}}_{\text{matrix-multiplication}} \mid i \le k < j\}$ if $i < j$

$$A_i \ldots A_j = \underbrace{(A_i \ldots A_k)}_{a_i \times a_{k+1}} \underbrace{(A_{k+1} \ldots A_j)}_{a_{k+1} \times a_{j+1}}$$

- naive recursive computation of $cost$ results in exponential algorithm
- solution: dynamic programming
  - compute values of $cost(i,j)$ for increasing differences of i and j – without recomputation

Lazy Maps and Sets

- `Data.Map.Lazy` provides lazy dictionaries (or: maps) in Haskell
- multiple construction possibilities
  - `empty :: Map k v`
  - `insert :: Ord k => k -> v -> Map k v -> Map k v`
  - `unionWith :: Ord k => (v -> v -> v) -> Map k v -> Map k v -> Map k v`
  - `fromList :: Ord k => [(k, v)] -> Map k v`
- querying single keys
  - `lookup :: Ord k => k -> Map k v -> Maybe v` (optional value)
  - `! :: Ord k => Map k v -> k -> v` (might throw error)
- implemented as balanced trees
- `Data.Set` has similar functionality to represent sets
- documentation
  - https://hackage.haskell.org/package/containers/docs/Data-Map-Lazy.html
  - https://hackage.haskell.org/package/containers/docs/Data-Map-Strict.html
  - https://hackage.haskell.org/package/containers/docs/Data-Set.html

Implementation of Bracketing Problem in Haskell via Lazy Maps

```
import qualified Data.Array.IArray as L
import qualified Data.Map.Lazy as M -- lazy dictionaries

optBracketCosts :: [Integer] -> Integer
optBracketCosts xs =
  let n = length xs - 1
      a = L.listArray (0,n) xs :: L.Array Int Integer
      m = M.fromList [((i,j),cost i j) | i <- [0..n - 1], j <- [i..n-1]]
      cost i j
        | i == j = 0
        | otherwise = minimum [costSplit k | k <- [i .. j - 1]] where
           costSplit k =
             let c1 = m M.! (i,k)
                 c2 = m M.! (k+1,j)
             in c1 + c2 + a L.! i * a L.! (k + 1) * a L.! (j + 1)
  in cost 0 (n-1)
```


Analysis of `optBracketCosts`

- no explicit sequence is given, in which dictionary is filled
- instead, an over-approximation of required values `(i,j)` is used:
  `i <- [0..n - 1], j <- [i..n-1]`
- recursion is done implicitly: from `(i,j)` with `i <= k <= j - 1`
  invoke both `(i,k)` and `(k+1,j)`
- input list `xs` is converted to array `a` for efficient element access
- the array might be changed to strict version (if input would be `[Int]`),
  but the dictionary must be lazy

Comparison of Maps and Immutable Arrays in Haskell

- lookup is logarithmic for maps, but constant time for arrays
- keys are arbitrary ordered objects, whereas type of array indices is restricted
- keys can have arbitrary gaps, whereas indices in arrays are dense
- maps also support deletion and change of key-value pairs
- both are available in strict and lazy version
- several variants of maps are available
  `https://haskell-containers.readthedocs.io/en/latest/map.html`

A Note on the Haskell Sources

- the demos and exercises are provided as a cabal package
- make sure to have ghc and cabal installed (via package manager or via ghcup);
  all programs are tested with ghc version 9.10.3
- download and extract the sources from the AFP website
- change directory into afp_01 (where `afp.cabal` is located)
- workflow for exercise sheet 1
  - `cabal repl` (run cabal project interactively)
  - `:m Exercise01` (load module Exercise01.hs)
  - `do {testsBrackets; testsEmb}` (run tests)
  - (edit `src/Exercise01.hs`)
  - `:r` (reload program after changes)
  - note: on first run, lean-check and other packages might be installed
  - just upload updated version of `Exercise01.hs` in OLAT

Literature

- Real World Haskell, pages 32–33, 108–110, 270–274, 289–292