



Functional Programming

Week 11 – Lazy Evaluation, Infinite Lists

René Thiemann Philipp Anrain Marc Bußjäger Benedikt Dornauer Manuel Eberl
Christina Kohl Sandra Reitering Christian Sternagel

Department of Computer Science

Last Lecture

- `IO a` is type of I/O-actions with resulting type `a`
- `do`-blocks are used for sequential composition of I/O-actions
- clear separation between purely functional and I/O-code:
 - embed functional code into I/O: `return :: a -> IO a`
 - the other direction is not available: no function of type `IO a -> a`
- `ghc` compiles programs that provide `main :: IO ()` function in module `Main`
- example application: connect four
 - user-interface: I/O-code
 - game logic: purely functional

Evaluation Strategies

Pure Functions

- a function is **pure** if it always returns same result on same input
- pure functions are similar to mathematical functions
- examples of pure functions
 - addition
 - sort a list
 - ...
- examples of non-pure functions
 - roll a dice
 - current time
 - position of cursor
 - ...
- pure languages permit to define only pure functions
- **Haskell is a pure language**

Pure Functions and I/O

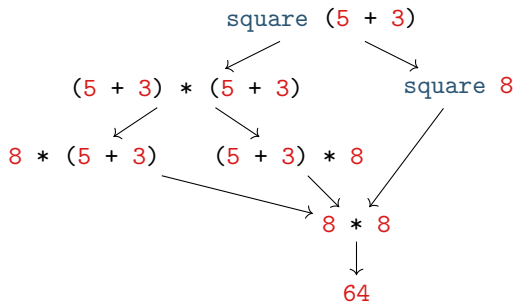
- even I/O is pure in Haskell
- consider `main = getLine >>= putStrLn . ("Hello " ++)`
- it seems that the result depends on user input, so is not pure
- however `main :: IO ()`, so the functional value of `main` is not what is entered and printed during execution, but the value is of type `IO ()`, i.e., a sequence of actions that are executed when running the program; and indeed this sequence is always the same:

first read some input i and then print the string `"Hello i "`

- alternative argumentation: interpret type `IO a` a state transformer on the outside world, e.g., as a function of type `RealWorld -> (RealWorld, a)`
- remark: in the remainder of this lecture we will only consider purely functional programs without I/O

Evaluation Order

- there are several ways to evaluate expressions, consider `square x = x * x`



- in pure languages, the evaluation order has no impact on resulting normal form
- normal form: an expression that cannot be evaluated further, a result

Theorem

Whenever there are two (different) ways to evaluate a Haskell expression to normal form, then the resulting normal forms are identical.

Standard Evaluation Strategies

- each functional language fixes the evaluation order via some evaluation strategy
- three prominent evaluation strategies (expressions represented as **trees** and **dags**)
 - **call-by-value** / **strict** / **innermost**: first evaluate arguments

$$\text{square } (5+3) = \begin{array}{c} \text{square} \\ | \\ + \\ / \quad \backslash \\ 5 \quad 3 \end{array} = \begin{array}{c} \text{square} \\ | \\ 8 \end{array} = \begin{array}{c} * \\ / \quad \backslash \\ 8 \quad 8 \end{array} = 64$$

- **call-by-name** / **non-strict** / **outermost**: directly replace function application by rhs

$$\text{square } (5+3) = \begin{array}{c} \text{square} \\ | \\ + \\ / \quad \backslash \\ 5 \quad 3 \end{array} = \begin{array}{c} * \\ / \quad \backslash \\ + \quad + \\ / \quad \backslash \quad / \quad \backslash \\ 5 \quad 3 \quad 5 \quad 3 \end{array} = \begin{array}{c} * \\ / \quad \backslash \\ + \quad 8 \\ / \quad \backslash \\ 5 \quad 3 \end{array} = \begin{array}{c} * \\ / \quad \backslash \\ 8 \quad 8 \end{array} = 64$$

- **call-by-need** / **lazy evaluation**: like call-by-name + sharing (dags = directed acyclic graphs)

$$\text{square } (5+3) = \begin{array}{c} \text{square} \\ | \\ + \\ / \quad \backslash \\ 5 \quad 3 \end{array} = \begin{array}{c} * \\ (\quad) \\ + \\ / \quad \backslash \\ 5 \quad 3 \end{array} = \begin{array}{c} * \\ (\quad) \\ 8 \end{array} = 64$$

Evaluation Strategy of Haskell

- Haskell uses lazy evaluation with left-to-right argument order
- sharing is applied whenever a variable occurs multiple times
- example: consider definition $f\ x = g\ x + g\ (3 + 5) + x$
 - when evaluating $f\ (1 + 2) = g\ (1 + 2) + g\ (3 + 5) + (1 + 2)$ the two occurrences of $1 + 2$ are shared: they use the same variable x
 - when evaluating $f\ (3 + 5) = g\ (3 + 5) + g\ (3 + 5) + (3 + 5)$ the two occurrences of $g\ (3 + 5)$ are not shared: it was a coincidence that x was substituted by $3 + 5$ and this equality is not detected at runtime
- there might be further sharing (depending on the compiler), e.g. sharing common subexpressions such as the expression $g\ x$ in a function definition $f\ x = g\ x + h\ (g\ x)$
- argument evaluation within function invocation $f\ expr1 \dots exprN$ is mainly triggered by pattern matching, i.e., the process of finding the suitable defining equation $f\ pat1 \dots patN = expr$, cf. slides 12 and 14 of week 3
- many builtin arithmetic functions will trigger evaluation of all arguments, e.g., $(0 :: Integer) * undefined$ will result in error, and not in 0

Evaluation Strategy and Termination

- consider the following Haskell script

```
three :: Integer -> Integer
```

```
three x = 3
```

```
inf :: Integer
```

```
inf = 1 + inf
```

- strict evaluation does not terminate, i.e., it will evaluate forever

```
three inf = three (1 + inf) = three (1 + (1 + inf)) = ...
```

- non-strict and lazy evaluation are immediately done

```
three inf = 3
```

Theorem

- if the evaluation of an expression terminates for some evaluation strategy, then it terminates using non-strict or lazy evaluation
- if the evaluation of an expression terminates using strict evaluation, then it terminates for every evaluation strategy

Comparison of Evaluation Strategies

- call-by-value
 - easy to understand
 - easy to implement
 - overhead in evaluating non-required expressions
 - used in many functional programming languages
- lazy evaluation
 - harder to understand
 - single evaluation step is more complicated to implement:
pass arguments that are unevaluated expressions (thunks) instead of just values
 - overhead in computing with thunks
 - allows programmers to naturally define and work with infinite data
 - used in Haskell

Tail Recursion and Strict Evaluation

Different Kinds of Recursion

- a function calling itself is **recursive**
- functions that mutually call each other are **mutually recursive**

```
even n | n == 0    = True
      | otherwise = odd  (n - 1)
odd  n | n == 0    = False
      | otherwise = even (n - 1)
```

- **nested recursion**: recursive calls inside recursive calls

```
ack n m | n == 0 = m + 1
        | m == 0 = ack (n - 1) 1
        | otherwise = ack (n - 1) (ack n (m - 1))
```

- **linear recursion**: at most one recursive call (per if-then-else branch)

- `fib n | n >= 2 = fib (n - 1) + fib (n - 2)`
- `length (x : xs) = 1 + length xs`
- `f x = if even x then f (x `div` 2) else f (3 * x + 1)`



✗

✓

✓

- **tail recursion** and **guarded recursion** will be discussed in more detail

Tail Recursion

- tail recursion is special form of linear recursion
- additional requirement
 - recursive function calls happen at the outermost level
 - however, they can be within an if-then-else
- examples
 - `length (x : xs) = 1 + length xs` 
 - `f x = if even x then f (x `div` 2) else f (3 * x + 1)` 
- advantage of tail recursion
 - no dangling function calls
 - can be evaluated as loop
 - space efficient

Example: Advantage of Tail Recursion

- linear but not tail recursive variant

```
sumRec 0 = 0
```

```
sumRec n = n + sumRec (n - 1)
```

```
sumRec 5 = 5 + sumRec (5 - 1)
= 5 + sumRec 4 = 5 + (4 + sumRec (4 - 1))
= 5 + (4 + sumRec 3) = 5 + (4 + (3 + sumRec (3 - 1))) = ...
= 5 + (4 + (3 + (2 + (1 + 0)))) = ... = 15    -- linear space
```

- tail recursive variant using **accumulator** to store intermediate results

```
sumTr n = aux 0 n where
```

```
aux acc 0 = acc
```

```
aux acc n = aux (acc + n) (n - 1)
```

```
sumTr 5
= aux 0 5 = aux (0 + 5) (5 - 1)
= aux 5 4 = aux (5 + 4) (4 - 1)
= aux 9 3 = ... = 15
```

-- constant space, implement as loop with two variables: acc and n

Problem of Tail Recursion using Lazy Evaluation

```
sumTr n = aux 0 n where
```

```
  aux acc 0 = acc
```

```
  aux acc n = aux (acc + n) (n - 1)
```

- example evaluation of `sumTr` on previous slide used call-by-value
- in lazy evaluation `acc` and `n` are only evaluated on demand
- causes linear memory consumption in `sumTr`

```
sumTr 5                                -- with lazy evaluation
= aux 0 5
= aux (0 + 5) (5 - 1)
= aux (0 + 5) 4
= aux ((0 + 5) + 4) (4 - 1)
= ...
= aux (((((0 + 5) + 4) + 3) + 2) + 1) 0
= (((((0 + 5) + 4) + 3) + 2) + 1) = ... = 15
```

Enforcing Evaluation

- Haskell function to enforce evaluation: `seq :: a -> b -> b`
- evaluation of `seq x y` first evaluates `x` to WHNF and then returns `y`
- **WHNF**: weak head normal form
- expression `e` is in WHNF iff it has one of the following three shapes
 - `e = C expr1 ... exprN` for some constructor `C` (constructor application)
 - `e = f expr1 ... exprN` if the defining equations of `f` have `M > N` arguments, i.e., they are of the form `f pat1 ... patM = expr` (too few arguments)
 - `e = \ pat1 ... patN -> expr` (λ -abstraction)
- examples
 - in WHNF: `True`, `7.1`, `(5+1)` : `[1] ++ [2]`, `(:)`, `undefined` : `undefined`, `(++)`, `(++ undefined)`, `\ x -> undefined`
 - not in WHNF: `[1] ++ [2]`, `(\ x -> x + 1) (1 + 2)`, `undefined ++ undefined`
 - evaluation: `let x = 1 + 2 in seq x (f x)`
 - `= seq (1 + 2) (f (1 + 2))` -- with 1 + 2 shared
 - `= seq 3 (f 3)` -- seq enforced evaluation of argument
 - `= f 3 = ...` -- evaluation of f 3 continues

Example Application using seq

- solve memory problem in tail recursion by enforcing evaluation of accumulator

```
sumTrSeq n = aux 0 n where
```

```
  aux acc 0 = acc
```

```
  aux acc n = let accN = acc + n in seq accN (aux accN (n - 1))
```

```
sumTrSeq 5
```

```
= aux 0 5
```

```
= let accN = 0 + 5 in seq accN (aux accN (5 - 1))
```

```
= seq (0 + 5) (aux (0 + 5) (5 - 1))
```

-- 0 + 5 is shared

```
= seq 5 (aux 5 (5 - 1))
```

-- and evaluated

```
= aux 5 (5 - 1)
```

```
= aux 5 4
```

-- pattern matching triggers evaluation

```
= let accN = 5 + 4 in seq accN (aux accN (4 - 1))
```

```
= seq (5 + 4) (aux (5 + 4) (4 - 1))
```

-- 5 + 4 is shared

```
= seq 9 (aux 9 (4 - 1))
```

-- and evaluated

```
= aux 9 (4 - 1)
```

-- same structure as above

```
= ... = 15
```

-- constant space

Enforcing Strict Evaluation ... Continued

- besides `seq`, there are other options to enforce strict evaluation
- **strict library functions** like a strict version of `foldl`:

```
Data.List.foldl' :: (b -> a -> b) -> b -> [a] -> b
```

```
import Data.List
length = foldl' (\ x _ -> x + 1) 0
```

- pattern matching with **bang patterns** to enforce evaluation, e.g.,
`aux acc n = let !accN = acc + n in aux accN (n - 1)`
- **strict datatypes**
- see https://downloads.haskell.org/~ghc/8.10.7/docs/html/users_guide/glasgow_exts.html#bang-patterns-and-strict-haskell for further details

Lazy Evaluation and Infinite Lists

Guarded Recursion

- every recursive call is inside (“guarded by”) a constructor
- also known as “tail recursion modulo cons”
- more important than tail recursion in Haskell
- allows the result to be consumed lazily – tail recursion provides the result only at the end
- examples

- `map f [] = []`

- `map f (x:xs) = f x : map f xs`



- `reverse xs = revAux xs [] where`

- `revAux [] ys = ys`



- `revAux (x : xs) ys = revAux xs (x : ys)`

- `enumFrom x = x : enumFrom (x + 1)`



- remarks on `enumFrom`

- above definition is simplified, actual definition works for members of type class `Enum`, e.g., `Int`, `Char`, `Integer`, `Double`, ... and prevents overflows
- syntactic sugar: `[x..] = enumFrom x`

Infinite Lists

- infinite lists \sim **sequences** of elements (also known as streams)
- programming with infinite lists: producing and consuming elements of sequences one after another (e.g., with guarded recursion)
- example: `[x..] = x : [x + 1 ..]` generates infinite list
- in combination with lazy evaluation, infinite lists do not always cause non-termination
- examples

```
take 2 [7..]
= take 2 (7 : [8..])
= 7 : take 1 [8..]
= 7 : 8 : take 0 [9..]
= [7, 8]
```

```
takeWhile (< 95) $ map (\ x -> x * x) [0..]
= ... = [0,1,4,9,16,25,36,49,64,81]
```

```
filter (< 100) $ map (\ x -> x * x) [0..]
= ... = [0,1,4,9,16,25,36,49,64,81 -- interrupted
```

Laziness and Infinite Data Structures Facilitate Modularity

- separation of concerns
 - write small functions with specific tasks
 - use potentially infinite data structures
- example: find index of first list element satisfying predicate
 - function `firstIndex :: (a -> Bool) -> [a] -> Int`
 - in Haskell

```
firstIndex p = fst . head . filter (p . snd) . zip [0..]
```
 - (lazy) evaluation (without showing expansion of `(.)` and `($)`)

```
firstIndex (== 1) [1..9]
= fst . head . filter ((== 1) . snd) $ zip [0..] [1..9]
= fst . head . filter ((== 1) . snd) $ (0,1) : zip [1..] [2..9]
= fst . head $ (0,1) : filter ((== 1) . snd) $ zip [1..] [2..9]
= fst (0,1)
= 0
```
 - without laziness several **complete** list traversals are required when using library functions (e.g., computation of length and addition of indices)
 - remark: `firstIndex` works for arbitrary lists as input: finite and infinite

Sieve of Eratosthenes

- goal: generate list of **all** prime numbers
- algorithm
 1. start with list of all natural numbers (from 2 on)
 2. mark first element x as prime
 3. remove all multiples of x
 4. go to Step 2
- in Haskell

```
primes :: [Integer]
primes = sieve [2..] where
    sieve (x : xs) = x : sieve (filter (\ y -> y `mod` x /= 0) xs)
```

```
> take 1000 primes           -- the first 1000 primes
> takeWhile (< 1000) primes -- all primes below 1000
```

Summary

- in pure functional languages such as Haskell the result does not depend on the evaluation strategy
- different kinds of recursion
- **tail recursion** is usually efficient as it can be implemented as loop
- **seq** can be used to **enforce strict evaluation** (in particular of accumulators)
- **lazy evaluation** allows modeling of infinite lists
- **guarded recursion** is important for algorithms on infinite lists
- **infinite lists** permit to naturally formulate several algorithms (without having to take care about boundary conditions)