



Functional Programming

Week 12 – Cyclic Data Structures, Abstract Data Types

René Thiemann

Jonathan Bodemann

James Fox

Joshua Ocker

Daniel Rainer

Daniel Ranalter

Christian Sternagel

Department of Computer Science

Last Lecture – Evaluation Strategies

- evaluation strategies determine order of evaluation
- three kinds: innermost, outermost, and lazy evaluation (outermost + sharing)
- in pure functional languages the result does not depend on the evaluation strategy
 - consider non-pure language with function `uNum :: Int` that asks the user for a number and returns it

- what is result of evaluating

`f uNum where f x = x - x`

if the user will enter the two numbers 5 and 3?

- outermost (left-to-right): `f uNum = uNum - uNum = 5 - uNum = 5 - 3 = 2`
 - outermost (right-to-left): `f uNum = uNum - uNum = uNum - 5 = 3 - 5 = -2`
 - innermost: `f uNum = f 5 = 5 - 5 = 0`
- tail recursion in combination with innermost strategy can be implemented as loop
 - `seq a b` enforces evaluation of `a` to WHNF and then results in `b`
 - pitfall: in the following Haskell program, `seq` does not have the required effect

```
sumAux acc 0 = acc
```

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

```
-- correct: = let accN = acc + n in seq accN (sumAux accN (n - 1))
```

Last Lecture – Lazy Evaluation and Infinite Data Structures

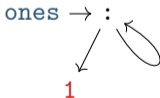
- it is possible to define infinite lists, trees, etc., e.g.,
`enumFrom x = x : enumFrom (x + 1)`
- finite parts of infinite lists can be accessed, e.g., via `take`, `takeWhile`, etc., and lazy evaluation will not enforce computation of whole infinite list
- benefit: natural definition of several algorithms without having to worry about bounds, lengths, etc.
- main algorithmic structure: guarded recursion so that new constructors are produced in each recursive evaluation step

Cyclic Data Structures

Cyclic Lists

- aim: direct definition of infinite lists which are implicitly computed on demand via lazy evaluation
- methodology: provide **start of cyclic list** and **remaining cyclic list**
- a first example: the infinite list of ones
 - starting element is 1
 - remaining list is the list of ones itself
 - Haskell definition

```
ones :: [Integer]
ones = 1 : ones
```
 - created cyclic data structure



Combination of Lists

- cyclic definitions may involve auxiliary functions such as `map`, `filter`, and `zipWith`
- example: the list of natural numbers: `nats`

- start is 0
- remainder is addition of the list of ones with natural numbers itself

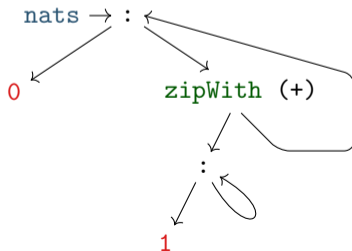
```
0 1 2 3 4 5 ...
+ 1 1 1 1 1 1 ...
= 1 2 3 4 5 6 ...   (= tail nats)
```

- in Haskell

```
nats :: [Integer]
```

```
nats = 0 : zipWith (+) nats ones
```

- created cyclic data structure:



Computing Fibonacci Numbers

- definition:
$$fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{otherwise} \end{cases}$$

- efficient computation of Fibonacci numbers via cyclic lists
- two starting elements: 0 and 1
- remainder is `tail(tail fibs) = fibs + tail fibs`

```
0 1 1 2 3 5 8 ... -- fibs
+ 1 1 2 3 5 8 13 ... -- tail fibs
= 1 2 3 5 8 13 21 ... -- tail (tail fibs)
```

- in Haskell

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

- remark: two starting elements, since otherwise `tail fibs` in rhs cannot be evaluated

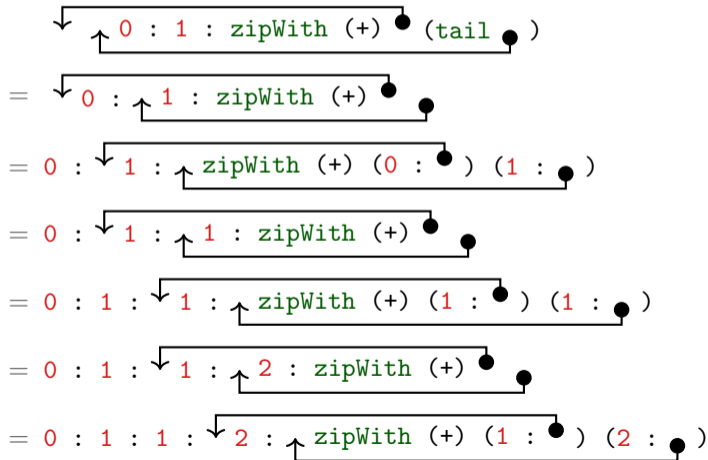
Fibonacci Numbers in Haskell

- implementation was given in first lecture (slide 18 of week 1)

```
fibs :: [Integer]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- cyclic definition of list, evaluation:



Infinite Data Structures Beyond Lists

- lists are not the only infinite data structure, e.g., there are also infinite trees (vertically and/or horizontally)
- also cyclic trees can be defined, e.g., consider a tree that represents all (finite and infinite) paths in the graph starting from node 1



- in Haskell we use a mutual recursive definition of four trees (`Paths`)

```
data Paths = Root Integer [Paths]
```

```
paths1 = Root 1 [paths2]
```

```
paths2 = Root 2 [paths1, paths3]
```

```
paths3 = Root 3 [paths2, paths4]
```

```
paths4 = Root 4 []
```

- access finite parts of infinite tree in the same way as for infinite lists;
example: analogy of “take first n elements of a potentially infinite list” would be a function for computing “all paths of length up to n of an potentially infinite tree”

Abstract Data Types

Concrete and Abstract Datatypes

- **concrete** datatypes
 - defined via `data` which defines **values** of that type
 - user defines own operations on this type via pattern matching
 - no need for primitive operations on that type
 - examples: `Rat`, `Person`, `Expr`, `Bool`, `[a]`, ...
- **abstract** datatypes
 - defined via their primitive **operations**
 - usually no access to internal structure of representation of values
 - pattern matching only via equality: `f 5 = ...` is equivalent to `f x = if x == 5 ...`
 - **abstraction barrier**: internal structure can be easily changed
 - meaning of operations usually specified
 - examples: `Char`, `Integer`, `Double`, ... which provide basic arithmetic operations and conversion to strings

Example Abstract Datatype: Queues

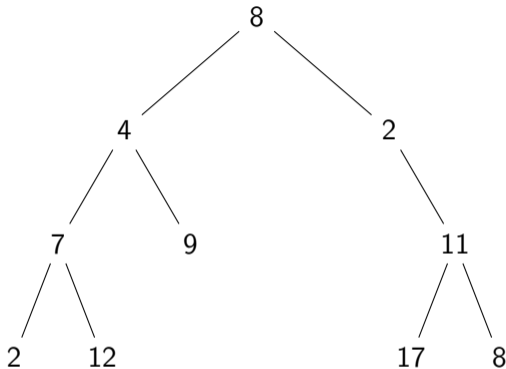
- queues are useful in computer science: printer (jobs), web-server (requests), ...
- queue provides the following operations
 - `empty :: Queue a` – the empty queue for elements of type `a`
 - `isEmpty :: Queue a -> Bool` – check whether queue is empty
 - `dequeue :: Queue a -> (a, Queue a)` – remove head of queue
 - `enqueue :: a -> Queue a -> Queue a` – add new element to end of queue

these operations in combination with their types are the **signature** of the abstract datatype `Queue a`

- signature only gives idea about operations; more information can be specified via **axiomatic specification** in the form of equations or formulas
 - `isEmpty empty`
 - `not $ isEmpty $ enqueue x q`
 - `dequeue (enqueue x empty) = (x, empty)`
 - `not $ isEmpty q -> dequeue q = (y, q') ->`
`dequeue (enqueue x q) = (y, enqueue x q')`

Example Application for Queues: Tree-Traversals

- consider binary tree



- tree-traversal: visit all nodes, e.g., to search for node, or convert nodes to list

- in-order [2,7,12,4,9,8,2,17,11,8]
- depth-first search, pre-order [8,4,7,2,12,9,2,11,17,8]
- breadth-first search [8,4,2,7,9,11,2,12,17,8]

Tree Traversals in Haskell

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

```
inOrder :: Tree a -> [a]
```

```
inOrder Empty = []
```

```
inOrder (Node l n r) = inOrder l ++ [n] ++ inOrder r
```

```
-- preOrder is similar to inOrder
```

```
bfs :: Tree a -> [a]
```

```
bfs t = bfsMain (enqueue t empty) where
```

```
  bfsMain :: Queue (Tree a) -> [a]
```

```
  bfsMain q
```

```
    | isEmpty q = []
```

```
    | otherwise = let (t', q') = dequeue q in
```

```
      case t' of
```

```
        Empty -> bfsMain q'
```

```
        Node l n r -> n : (bfsMain $ enqueue r $ enqueue l $ q')
```

Implementing an Abstract Datatype

- implementation has to provide the desired operations and must satisfy the specification (informal text or axiomatic)
 - `empty :: Queue a`
 - `isEmpty :: Queue a -> Bool`
 - `dequeue :: Queue a -> (a, Queue a)`
 - `enqueue :: a -> Queue a -> Queue a`
 - `isEmpty empty`
 - `not $ isEmpty $ enqueue x q`
 - `dequeue (enqueue x empty) = (x, empty)`
 - `not $ isEmpty q -> dequeue q = (y, q') ->`
`dequeue (enqueue x q) = (y, enqueue x q')`
- any implementation can be used, e.g., a basic one in the beginning, which might be replaced by more efficient one later on
- if corner cases are not specified, implementation can choose freely, e.g., how dequeue should behave on empty queues
- modules can be used to hide internals

A Basic Implementation of Queues

```
module BasicQueue(Queue, empty, isEmpty, dequeue, enqueue) where
data Queue a = Empty | Enqueue a (Queue a)

empty = Empty
enqueue = Enqueue

isEmpty Empty = True
isEmpty (Enqueue x q) = False

dequeue (Enqueue x Empty) = (x, Empty)
dequeue (Enqueue x q) = (y, Enqueue x q') where
  (y, q') = dequeue q
dequeue Empty = error "dequeue on empty queue"
```

- implementation is rather direct translation of specification
- `empty` and `enqueue` are implemented as constructors of queues, and exported; still the constructors itself are not exported and so internal structure is not revealed, e.g., externally no pattern matching on queues is possible

Notes on the Basic Implementation of Queues

```
...  
data Queue a = Empty | Enqueue a (Queue a)  
isEmpty Empty = True  
isEmpty (Enqueue x q) = False  
dequeue (Enqueue x Empty) = (x, Empty)  
dequeue (Enqueue x q) = (y, Enqueue x q') where  
  (y, q') = dequeue q  
dequeue Empty = error "dequeue on empty queue"
```

- we did not **prove** that implementation meets the specification; will be covered in
 - program verification (bsc), or
 - interactive theorem proving (msc)
- implementation is inefficient, since first enqueueing n elements and then dequeuing n elements requires $\sim \frac{1}{2}n^2$ evaluation steps

Towards a More Efficient Implementation of Queues

- previous queue-type is essentially a list where the list head represents the end of the queue (queue = reversed list)
- assume customers 1, 2, 3 and 4 enqueue in that order, then the representation is [4, 3, 2, 1]
- enqueueing is efficient since it just adds element in front of list
- dequeueing is expensive since it traverses and rebuilds whole list
- new version: store queue as pair of two lists: (front, rear)
 - front part of queue (head of queue is head of list)
 - rear part of queue in reverse order (tail of queue is head of list)
 - invariant: whenever front part of queue is empty then whole queue is empty
- example queue with customers 1, 2, 3, 4 has multiple representations
 - ([1, 2, 3, 4], []) ✓
 - ([1, 2, 3], [4]) ✓
 - ([1], [4, 3, 2]) ✓
 - ([], [4, 3, 2, 1]) ✗
- advantage: often constant time access to both ends of queue

More Efficient Implementation of Queues

```
module BetterQueue(Queue, empty, isEmpty, dequeue, enqueue) where
type Queue a = ([a], [a])
empty :: Queue a
empty = ([], [])
isEmpty :: Queue a -> Bool
isEmpty (front, _) = null front
enqueue :: a -> Queue a -> Queue a
enqueue x (front, rear) = maybeMtf (front, x : rear)
dequeue :: Queue a -> (a, Queue a)
dequeue ([], _) = error "dequeue on empty queue"
dequeue (x : front, rear) = (x, maybeMtf (front, rear))
maybeMtf ([], rear) = (reverse rear, [])
maybeMtf q = q
```

Efficiency of More Efficient Implementation

```
dequeue ([], _) = error "dequeue on empty queue"  
dequeue (x : front, rear) = (x, maybeMtf (front, rear))
```

```
maybeMtf ([], rear) = (reverse rear, [])
```

```
maybeMtf q = q
```

- move-to-front operation required when `front` is empty (obey invariant)
- single move-to-front operation may be expensive, but these operations are rare
- efficiency: n queue operations require at most $2n$ evaluation steps
- proving technique: **amortized cost analysis**, will be covered in course algorithms and data-structures

Abstraction Barrier of More Efficient Implementation

```
module BetterQueue(Queue, empty, isEmpty, dequeue, enqueue) where
```

```
type Queue a = ([a], [a])
```

```
...
```

```
empty :: Queue a
```

```
...
```

- since `type` is just an abbreviation:

```
empty :: ([a], [a])
```

- since pairs and lists are visible, external users can completely inspect internal structure and create queues which are not permitted, e.g., `isEmpty ([], [4,3,2,1])` evaluates to `True`
 - since `type` is just an abbreviation, in particular `Queue`'s are instances of `Eq`, `Show`, and `Ord`, which might not be intended
 - simple solution: hide representation in new datatype
- ```
data Queue a = Queue ([a], [a])
```

## Implementation with Separate Datatype

```
module DataQueue(Queue, empty, isEmpty, dequeue, enqueue) where
data Queue a = Queue ([a], [a]) -- new datatype
empty :: Queue a
empty = Queue ([], []) -- wrap Queue constructor around
isEmpty :: Queue a -> Bool
isEmpty (Queue (f, _)) = null f -- unwrap Queue constructor
queue = Queue . maybeMtf
enqueue :: a -> Queue a -> Queue a
enqueue x (Queue (f, r)) = queue (f, x : r)
dequeue :: Queue a -> (a, Queue a)
dequeue (Queue ([], _)) = error "dequeue on empty queue"
dequeue (Queue (x : f, r)) = (x, queue (f, r))
maybeMtf ([], r) = (reverse r, [])
maybeMtf q = q
```

# Newtype

```
data Queue a = Queue ([a], [a])
```

```
queue = Queue . maybeMtf
```

```
enqueue :: a -> Queue a -> Queue a
```

```
enqueue x (Queue (f, r)) = queue (f, x : r)
```

...

- always wrapping and unwrapping the `Queue` constructor has some efficiency penalty
- more efficient version to hide an implementation type: **newtype**
- syntax: **newtype** TName tvars = CName typ
  - only **one** constructor (CName) allowed
  - this constructor must have exactly one argument type
  - nearly equivalent to `data TName tvars = CName typ`,  
one difference: `newtype` is faster (CName won't be created at runtime)
- minimal change in implementation of queues
  - **newtype** `Queue a = Queue ([a], [a])` instead of  
`data Queue a = Queue ([a], [a])`

## Summary

- **cyclic lists**
  - implicit definition of infinite lists
  - can be used to elegantly and efficiently implement some functions (Fibonacci)
- **abstract datatypes**: specify operations with their properties; introduces **abstraction barriers** that permit change of implementations
- example: different implementations of **queues**
- **newtype** is efficient variant of **data** in case there is only one constructor with one argument
- example abstract datatypes
  - known: `Queue`, `Double`, `Char`, `Integer`, ...
  - further examples: sets (`Data.Set`), stacks (`Data.Stack`), dictionaries (`Data.Map`), ...