universität innsbruck

WS 2025/2026

Funktionale Programming

Woche 12 – Zyklische Datenstrukturen, Abstrakte Datentypen

René Thiemann
Philipp Dablander Joshua Ocker Michael Schaper Lilly Schönherr Adam Pescoller

Institut für Informatik

Letzte Vorlesung – Auswertungs-Strategien

- Auswertungs-Strategie bestimmt, wo in einem Ausdruck die Auswertung stattfindet
- drei Möglichkeiten: Innermost, Outermost, und Lazy Evaluation (Outermost mit Sharing)
- in rein funktionalen Sprachen hängt das Resultat nicht von der Auswertungs-Strategie ab
  - betrachten Sie nicht-reine Sprache mit Funktion `uNum :: Int`, die mittels I/O nach ein Zahl fragt und diese dann zurückgibt
  - Was ist das Resultat des Ausdrucks
    `let f x = x - x in f uNum`
    wenn die Zahlen 5 und 3 in dieser Reihenfolge eingegeben werden?
  - Outermost (links nach rechts): `f uNum = uNum - uNum = 5 - uNum = 5 - 3 = 2`
  - Outermost (rechts nach links): `f uNum = uNum - uNum = uNum - 5 = 3 - 5 = -2`
  - Innermost: `f uNum = f 5 = 5 - 5 = 0`
- Endrekursion bei Innermost Auswertung wird durch einfache Schleifen implementiert
- `seq a b` erzwingt Auswertung von `a` zu WHNF und gibt dann `b` zurück
  - Achtung: im folgenden Haskell Programm hat `seq` nicht den erwünschten Effekt

```
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))
```

Letzte Vorlesung – Lazy Evaluation und unendliche Datenstrukturen

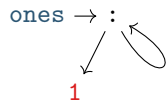
- es ist möglich, unendliche Listen, Bäume, usw. zu definieren, z.B. mittels `enumFrom x = x : enumFrom (x + 1)`
- extrahiere endliche Teile einer unendlichen Liste, z.B., mit `take`, `takeWhile`, etc., und durch Lazy Evaluation wird sichergestellt, dass nicht die ganze unendliche Liste berechnet wird
- Vorteil: einfache Definition vieler Algorithmen ohne Behandlung von Randfällen, ohne vorab-Berechnung von Längen, etc.
- Funktionen sollten Guarded Recursion nutzen, so dass neue Konstruktoren vor jedem rekursiven Aufruf erzeugt werden

Zyklische Datenstrukturen

Zyklische Listen

- Ziel: direkte Definition von unendlichen Listen, welche implizit durch Lazy Evaluation berechnet werden
- Methode: definiere Beginn der zyklischen Liste und Fortsetzung der zyklischen Liste
- Beispiel: die unendliche Liste der Einsen
  - erstes Element ist 1
  - Fortsetzung ist die Liste der Einsen
  - Haskell Definition

    ```
    ones :: [Integer]
    ones = 1 : ones
    ```
  - erstellte zyklische Datenstruktur



Kombination von Listen

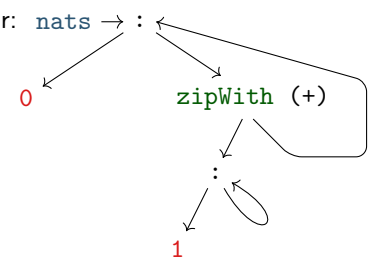
- Definition zyklischer Listen kann auf Funktionen wie `map`, `filter`, und `zipWith` basieren
- Beispiel: Liste der natürlichen Zahlen: `nats`
  - erstes Element ist 0
  - Fortsetzung ist die Addition der Liste der Einsen mit den natürlichen Zahlen

```
  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
```

  - erzeugte zyklische Datenstruktur: `nats` → `:`



Berechnung der Fibonacci-Folge

- Definition: $fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{otherwise} \end{cases}$
- effiziente Berechnung der Folge der Fibonacci Zahlen mittels zyklischer Listen
- zwei Start-Elemente: 0 und 1
- Fortsetzung ist `tail(tail fibs)`, was genau `fibs + tail fibs` ist

```
  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

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

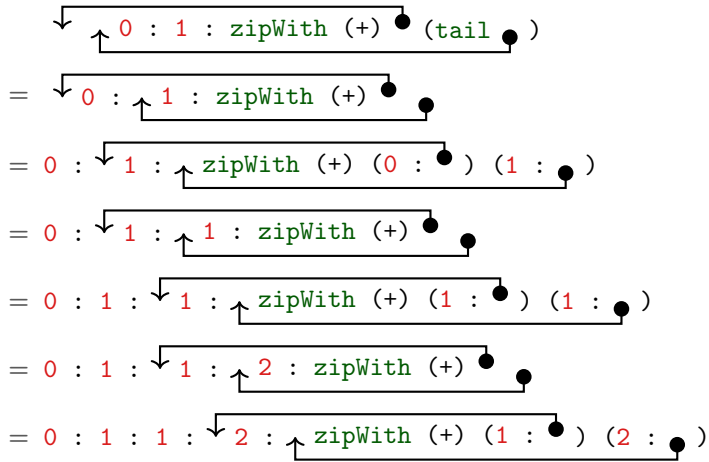
- Anmerkung: es benötigt zwei Start-Elemente, da sonst `tail fibs` in der rechten Seite nicht ausgewertet werden kann

Fibonacci Numbers in Haskell

- Implementatierung wurde in der ersten Vorlesung (Folie 19 von Woche 1) angegeben

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

- zyklische Definition der Liste, und Auswertung:



Weitere unendliche Datenstrukturen neben Listen

- Listen sind nicht die einzige unendliche Datenstruktur, z.B. gibt es auch unendliche Bäume (vertikal und/oder horizontal)
- auch zyklische Bäume sind möglich, z.B. könnte man den Baum aller (endlichen und unendlichen) Pfade in einem Graph definieren, wobei die Pfade in Knoten 1 beginnen

- in Haskell definieren wir vier verschränkt rekursive Bäume (`Paths`)

```
data Paths = Root Integer [Paths]

paths1 = Root 1 [paths2]
paths2 = Root 2 [paths1, paths3]
paths3 = Root 3 [paths2, paths4]
paths4 = Root 4 []
```

- Zugriff auf endliche Teile des unendlichen Baums wird analog zu Listen durchgeführt; Beispiel: die Analogie von "nehme die ersten n Elemente einer (unendlichen) Liste" würde zu einer Funktion "nehme alle Pfade bis zur Länge n eines (unendlichen) Baums"

Abstrakte Datentypen

Konkrete und Abstrakte Datentypen

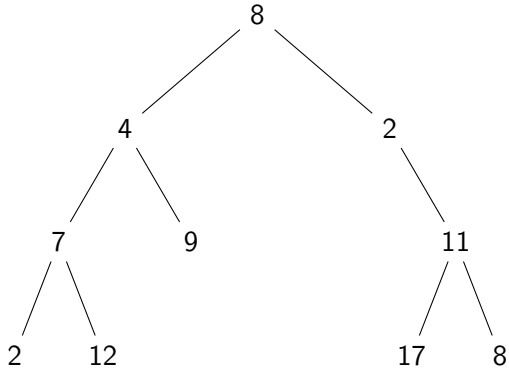
- konkrete Datentypen
  - werden mittels `data` definiert, und man sieht die möglichen Werte des Typs
  - Anwender können eigene Funktionen für diesen Typ mittels Pattern Matching definieren
  - es ist nicht notwendig, vordefinierte Funktionen für den Typ bereitzustellen
  - Beispiele: `Rat`, `Person`, `Expr`, `Bool`, `[a]`, ...
- abstrakte Datentypen
  - werden durch die vordefinierten Funktionen genutzt
  - oft hat man keinen Zugriff auf die interne Repräsentation der Werte
  - Pattern Matching funktioniert nur über Gleichheit: `f 5 = ...` ist äquivalent zu `f x = if x == 5 ...`
  - Abstraktions-Barriere: interne Repräsentation kann einfach geändert werden
  - Bedeutung der vordefinierten Operationen ist meistens klar spezifiziert
  - Beispiel: `Char`, `Integer`, `Double`, ... bieten arithmetische Funktionen, Vergleiche, sowie die Konvertierung zu Strings

Beispiel eines abstrakten Datentyps: Warteschlange (Queues)

- Warteschlange haben viele Anwendungen: Zugriff auf Drucker, Anfragen an Webserver, ...
- Warteschlangen bieten folgende Funktionalität
  - `empty :: Queue a` — die leere Warteschlange mit Elementen von Typ `a`
  - `isEmpty :: Queue a -> Bool` — prüfe, ob Warteschlange leer ist
  - `dequeue :: Queue a -> (a, Queue a)` — liefere und entferne erstes Element
  - `enqueue :: a -> Queue a -> Queue a` — füge Element am Ende hinzu
- diese Funktionen in Kombination mit ihren Typen bilden die Signatur des abstrakten Datentyps `Queue a`
- die Signatur bietet nur eine vage Idee der Funktionalität; weitere Informationen können durch eine axiomatische Spezifikation gegeben werden, in der Formeln das Verhalten beschreiben
  - `isEmpty empty`
  - `not $ isEmpty $ enqueue x q`
  - `dequeue (enqueue x empty) = (x, empty)`
  - `not $ isEmpty q` $\longrightarrow$ `dequeue q = (y, q')` $\longrightarrow$
    `dequeue (enqueue x q) = (y, enqueue x q')`

Beispiel Anwendung für Warteschlangen: Baum-Traversierungen

- betrachte Binärbaum



- Baum-Traversierung: besuche alle Knoten, z.B., um Baum in Liste zu verwandeln
  - In-Order [2,7,12,4,9,8,2,17,11,8]
  - Tiefensuche, Pre-Order [8,4,7,2,12,9,2,11,17,8]
  - Breitensuche [8,4,2,7,9,11,2,12,17,8]

Baumtraversierungen in Haskell

```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')
```

Implementierung eines abstrakten Datentyps

- Implementierung muss alle Funktionen der Signatur bereitstellen, und die (informelle oder axiomatische) Spezifikation erfüllen
  - `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')`
- jede Implementierung kann genutzt werden, z.B., eine einfache zu Beginn, die dann später durch eine effizientere ersetzt wird
- wenn Randfälle nicht genau spezifiziert sind, kann die Implementierung sich beliebig verhalten; Beispiel: `dequeue` auf leerer Warteschlange ist nicht spezifiert
- Module werden genutzt, um die internen Details der Implementierung zu verbergen

Einfache Implementierung von Warteschlangen

```
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"
```

- Implementierung ist eine direkte Umsetzung der Spezifikation
- `empty` und `enqueue` werden als Konstruktoren implementiert und exportiert; beachten Sie, dass die Konstruktoren selber nicht exportiert werden, damit die interne Struktur nicht öffentlich gemacht wird

Anmerkungen zur einfachen Implementierung

```
...
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"
```

- es wurde nicht bewiesen, dass die Implementierung die Spezifikation erfüllt; Methoden dazu werden in anderen Vorlesungen vorgestellt
  - Programm Verifikation (Bachelor), oder
  - Interaktives Theorem Beweisen (Master)
- Implementierung ist nicht effizient, da das Einfügen von n Elementen und anschließendes Entfernen von n Elementen $\sim \frac{1}{2}n^2$ Schritte benötigt

Idee für eine effizientere Implementierung von Warteschlangen

- bisheriger `Queue`-Typ ist im Wesentlichen eine Liste, wo der Listen-Anfang das Warteschlangen-Ende ist (Warteschlange ist umgedrehte Liste)
- angenommen Kunden 1, 2, 3 und 4 betreten die Schlange in dieser Reihenfolge, dann ist die Repräsentation `[4, 3, 2, 1]`
- Einfügen ist effizient, da man nur ein weiteres Element am Listen-Anfang hinzufügen muss
- Entfernen ist teuer, da man ganze Liste durchgehen muss
- neue Version: repräsentiere Warteschlange mit zwei Listen: (vorne, hinten)
  - vorderer Teil wird in normaler Reihenfolge gedeutet
  - hinterer Teil ist in umgekehrter Reihenfolge (Ende der Schlange ist vorne in der Liste)
  - Invariante: immer wenn vorderer Teil leer ist, dann ist auch der hintere Teil leer
- Beispiel mit Kunden 1, 2, 3, 4 hat mehrere Repräsentationen
  - `([1,2,3,4], [])` ✔
  - `([1,2,3], [4])` ✔
  - `([1], [4,3,2])` ✔
  - `([], [4,3,2,1])` ✘
- Vorteil: oft braucht man nur am Anfang einer der Listen etwas zu ändern

Effizientere Implementierung von Warteschlangen

```
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
```

Analyse der effizienteren Implementierung von Warteschlangen

```
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 wird benötigt, wenn vorderer Teil leer wird (Invariante)
- eine einzelne move-to-front Operation kann teuer sein, aber das tritt selten auf
- Laufzeit: n Warteschlangen Operationen benötigen maximal $2n$ Auswertungs-Schritte
- Beweismethode: amortisierte Kostenanalyse, siehe Vorlesung Algorithmen und Datenstrukturen

Abstraktions-Barriere der effizienten Implementierung

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

type Queue a = ([a], [a])
...
empty :: Queue a
...
```

- weil `type` nur eine Abkürzung einführt, gilt:
  `empty :: ([a], [a])`
- da Paare und Listen bekannt sind, ist es kein Problem, Warteschlangen zu bauen, die die interne Invariante verletzen, z.B. wertet `isEmpty ([], [4,3,2,1])` zu `True` aus
- weil `type` nur eine Abkürzung ist, sind `Queue`s Instanzen von `Eq`, `Show`, und `Ord`, was nicht beabsichtigt sein kein; z.B. gilt `([1,2],[]) /= (([1],[2]) :: Queue Int)`
- einfache Lösung: man verbirgt die Repräsentation in neuem Datentyp
  `data Queue a = Queue ([a], [a])`

Implementierung mit neuem Datentyp

```
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)
...
```

- das Hinzufügen und Entfernen des `Queue`-Konstruktors zur Laufzeit kostet etwas Zeit
- effizientere Version, um eine Implementierung zu verstecken: newtype
- Syntax: `newtype TName tvars = CName argType`
  - nur ein Konstruktor (`CName`) ist erlaubt – keine Alternativen
  - dieser Konstruktor muss genau ein Argument haben (`argType`)
  - logisch äquivalent zu `data TName tvars = CName argType` mit einem Unterschied: `newtype` ist effizienter, da Konstruktor `CName` während der Laufzeit nicht erzeugt oder entfernt wird
- minimale Änderung der bestehenden Implementierung der Warteschlangen
  - `newtype Queue a = Queue ([a], [a])` statt `data Queue a = Queue ([a], [a])`

Zusammenfassung

- zyklische Listen
  - implizite Definition von unendlichen Listen
  - diese können genutzt werden, um auf elegante und effiziente Art einige Funktionen zu definieren
- abstrakte Datentypen: Spezifikation von Funktionen mit entsprechenden Eigenschaften; Abstraktions-Barriere erlaubt es, nachträglich die Implementierung zu ändern
- Beispiel: unterschiedliche Implementierungen von Warteschlangen
- newtype ist effizientere Definition eines Datentyps, wenn dieser nur einen Konstruktor und ein Argument hat
- Beispiele von abstrakten Datentypen
  - bekannt: `Queue`, `Double`, `Char`, `Integer`, ...
  - weitere Beispiele: Mengen (Data.Set), Stapel (Data.Stack), Wörterbücher (Data.Map), ...