Praktische Informatik 3: Funktionale Programmierung

Vorlesung 5 / 21. November 2023

Algebraische Datentypen

Thomas Barkowsky

Wintersemester 2023/24

Universität Bremen

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- Testen und Qualitätssicherung
- I/O, Aktionen und Zustände
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

heute in dieser Vorlesung...

- Algebraische Datentypen
  - Definition neuer Typen mit `type` und `data`
  - Aufzählungstypen
  - Parametrisierte Datentypen
  - Datentypen zur Fehlerbehandlung
  - Rekursive Datentypen (Listen, Bäume & Co.)

Hinweis:

- Aufgabenblatt heute früh aktualisiert (.pdf und .zip)!

Definition neuer Datentypen

Definition neuer Datentypen: `type`

- Neue Typen auf der Basis existierender Typen
- Beispiel, aus *standard prelude*:

```
type String = [Char]
```

- Typbezeichner beginnen mit Großbuchstaben
- Typdefinitionen können geschachtelt werden
- Beispiel:
  - Typ Position als Integer-Paar:

```
type Pos = (Int,Int)
```

  - Typ Transformation als Funktion über Positionen:

```
type Trans = Pos -> Pos
```

`type` mit Parametern

- Paare gleichen Typs:
```
type Pair a = (a, a)
```
- Lookup-Tabelle mit Schlüssel-Wert-Paaren:

```
type Assoc k v = [(k, v)]
```

```
find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k==k']
```

- Was geht nicht mit `type`? – Rekursion!
  - Beispiel:
```
type Tree = (Int, [Tree])
```

Was tut `type` ?

- `type` erzeugt ein **Synonym** für existierende Typen
  - kein wirklich neuer Typ, nur neuer Name für Vorhandenes
  - bessere Lesbarkeit des Codes
  - komplexe Datentypen aus Standardtypen
  - werden bei Auswertung immer auf originale Typen reduziert

- Mächtiger als `type`: `data`
  - Generierung wirklich neuer Typen
  - **Algebraische Datentypen**
  - und damit auch: **rekursive** Datentypen

Algebraische Datentypen

Aufzählungen

- Generierung durch `data`
- Der einfachste Aufzählungstyp (*standard prelude*):

```
data Bool = False | True
```

- Typbezeichner und Werte beginnen mit Großbuchstaben
- `Bool` hat genau zwei mögliche Werte
  - Werte sind disjunkt
- `True` und `False` sind die **Konstruktoren** des Typs `Bool`
- Konstruktoren müssen **einmalig** sein
  - d.h. dürfen nur in einem Typ vorkommen

Verwendung neuer Datentypen

- Genauso wie alle vordefinierten Typen:
  - Argumente von Funktionen
  - Rückgabewerte von Funktionen
  - Speicherung in Datenstrukturen
  - Verwendung in *Pattern Matching*

- Beispiel:

```
data CardDir = North | South | East | West
```

Verwendung von CardDir

```
data CardDir = North | South | East | West
```

```
move :: CardDir -> Pos -> Pos
move North (x,y) = (x,y+1)
move South (x,y) = (x,y-1)
move East  (x,y) = ...
```

```
moves :: [CardDir] -> Pos -> Pos
moves []  p     = p
moves (m:ms) p = moves ms (move m p)
```

```
rev :: CardDir -> CardDir
rev North = South
rev South = North
rev East  = ...
```

Datentypen mit Parametern

- Mit `data` definierte Datentypen können parametrisiert werden
- Beispiel: ein Datum besteht aus Tag, Monat, Jahreszahl:
- ```
  data Date = Datum Year Month Day
  ```
  mit

  ```
  type Year  = Int
  data Month = Jan | Feb | Mar | ...
  type Day   = Int
  ```

- Was ist bei `Date` der Konstruktor? – `Datum`, eine dreistellige Funktion:

  ```
  > :t Datum
  Datum :: Year -> Month -> Day -> Date
  ```
  ```

Verwendung von `Date`

- z.B.

```
day :: Date -> Day
day (Datum y m d) = d

month :: Date -> Month
month (Datum y m d) = m
```

`day` **und** `month` **sind** **Selektoren** des Datentyps `Date`

- Die Definition von Date macht diesen Typ unterscheidbar, z.B. von:

```
data VonBis = Tage Int Month Int
```

- Aber: die Typdefinition verhindert nicht die Definition unsinniger Daten:

```
> :t Datum 42 Jan (-0815)
Datum 42 Jan (-0815) :: Date
```

Warum “Algebraische” Datentypen?

- Algebraische Datentypen werden durch
  - **kartesische Produkte** (Konstruktoren mit Parametern)
    und
  - **disjunktive Verknüpfungen** (Summen) gebildet (Aufzählung von Konstruktoren)
- Algebraische Datentypen besitzen
  - **Konstruktoren** (implizit durch `data` Definition gegeben)
  - **Selektoren** (z.B. `day`, `month` für `Date`)
  - **Diskriminatoren**, Beispiel:

```
isValidDate :: Date -> Bool
isValidDate (Datum y m d) = y>=0 && d>=1 & ...
```

Algebraische Datentypen verallgemeinert

- Kartesische Produkte **UND** Disjunktion von Werten

  und

- **Rekursive** Algebraische Datenstrukturen

Beispiel: Geometrische Objekte

- Unterschiedliche Formen, unterschiedliche Parameter:

```
data Shape = Circle Float | Rect Float Float
```

- Unterschiedliche Konstruktor-Funktionen:

```
> :t Circle
Circle :: Float -> Shape
> :t Rect
Rect :: Float -> Float -> Shape
```

Beispiel: Geometrische Objekte

- Verwendung von `Shape`, z.B.:
- Fallunterscheidung in Funktionen, z.B.:
- mit

```
square :: Float -> Shape
square a = Rect a a
```

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x  * y
```

```
c1 = (Circle 1.5)
r1 = (Rect 1.5 3.0)
```

```
> area c1
7.0685835
> area r1
4.5
> area (square 2.5)
6.25
```

`data` mit Parametern

- `data` Definitionen können parametrisiert werden (vgl. `type`)
- Beispiel (*standard prelude*):

```
data Maybe a = Nothing | Just a
```

  - Werte von `Maybe` sind entweder `Nothing` oder `Just x` vom Typ `a`
- Wofür ist das gut? – Fehlerbehandlung!
  - `Nothing` = Fehler vs. `Just x` = erfolgreiche Berechnung
- Beispiel: wir wissen, dass

```
> div 3 0
*** Exception: divide by zero
```

- ... und dass

```
> head []
*** Exception: Prelude.head: empty list
```

Fehlerbehandlung mit `Maybe`

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv a b = Just (div a b)
```

```
safehead :: [a] -> Maybe a
safehead []     = Nothing
safehead (a:as) = Just a
```

- `Just` … oder `Nothing` – und was jetzt?

`Maybe`: vordefinierte Funktionen

- Achtung:

```
import Data.Maybe
```

```
fromJust :: Maybe a -> a
```

  - `fromJust` ist partiell!

```
> fromJust (Just 42)
42
```

- ähnlich:

```
maybeToList :: Maybe a -> [a]
```

  - aber:

```
> maybeToList Nothing
[]
```

`Maybe`: vordefinierte Funktionen

- Wie unser `safehead`:

```
listToMaybe :: [a] -> Maybe a
```

```
fromMaybe :: a -> Maybe a -> a
```

```
> fromMaybe "das war nix" (Just "supergut")
"supergut"
> fromMaybe "das war nix" Nothing
"das war nix"
```

- Mehr in `Data.Maybe` !

Rekursive Algebraische Datentypen

Rekursive Algebraische Datentypen

- Der durch `data` definierte Typ kann auf der rechten Seite der Definition benutzt werden!
- Auf diese Weise werden (potentiell) unendlich große Strukturen definiert
- Funktionale Behandlung durch Rekursion
- Beispiel: Listen selbstgemacht:

```
data List a = Nil | Cons a (List a)
```

  - eine `List` des Typs `a` besteht entweder aus `Nil` (die leere Liste) oder aus einem `Cons` bestehend aus einem Element des Typs `a` und einer Liste des Typs `a`

Verwendung von `List`

- Die leere Liste (“`[]`”):
- Einelementige Liste (“`[3]`”):
- Zwei Elemente (“`[2,3]`”):
- …
- Verwendung in Funktionen durch Rekursion, z.B.:

```
Nil
```

```
Cons 3 Nil
```

```
Cons 2 (Cons 3 Nil)
```

```
len :: List a -> Int
len Nil          = 0
len (Cons _ xs) = 1 + len xs
```

- Anmerkung: Listen in Haskell sehr ähnlich definiert: (spezielle Syntax!)

```
data [a] = [] | a : [a]
```

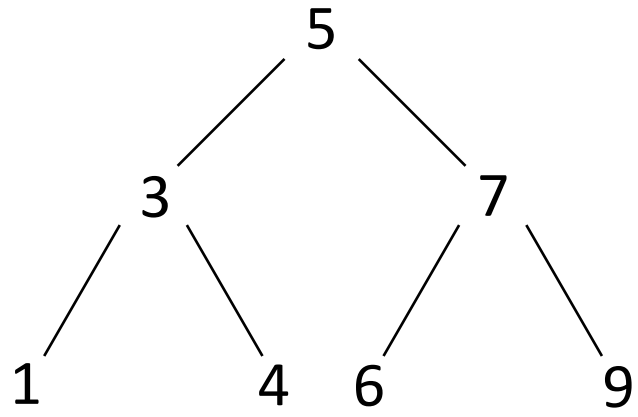
Interessanter als Listen: Bäume

- Binärbaum:

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

  - ein Baum für Werte des Typs `a` ist entweder ein Blatt des Typs `a` oder aber ein Knoten, der aus einem Wert und zwei Subbäumen besteht
  - ein `Int` - Baum:

```
t :: Tree Int
t = Node
    (Node (Leaf 1) 3 (Leaf 4))
    5
    (Node (Leaf 6) 7 (Leaf 9))
```



Funktionen für Bäume

```
occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf y)      = x==y
occurs x (Node l y r) = x==y || occurs x l || occurs x r
```

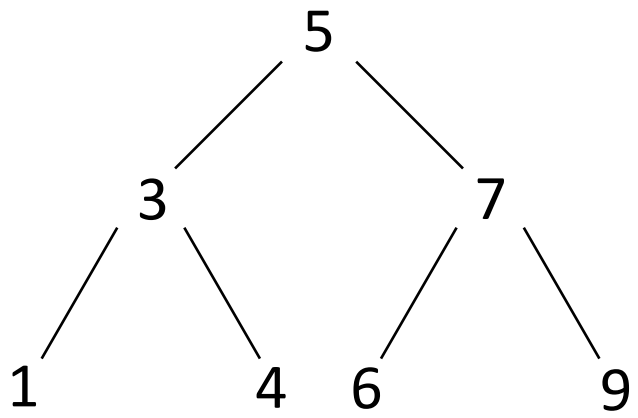
```
> occurs 1 t
True
```

```
> occurs 8 t
False
```

```
flatten :: Tree a -> [a]
flatten (Leaf x) = [x]
flatten (Node l x r) = flatten l ++ [x] ++ flatten r
```

```
> flatten t
[1,3,4,5,6,7,9]
```

Geordnete Bäume



- `t` ist ein **geordneter** Baum (so ein Zufall!):
  - Werte im linken Teilbaum immer kleiner, im rechten Teilbaum immer größer als der aktuelle Knoten
  - damit effizienter:

```
occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf y) = x==y
occurs x (Node l y r) | x==y = True
                      | x<y = occurs x l
                      | otherwise = occurs x r
```

Design von Bäumen

- Diverse Optionen, je nach Bedarf:
  - `t` trägt Daten an Blättern **und** an Knoten
  - Daten nur an Knoten:

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

  - Daten nur an Blättern:

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

  - verschiedende Typen an Knoten und Blättern:

```
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
```

  - Blätter implizit durch Knoten mit leerer Subbaum-Liste (kein Binärbaum):

```
data Tree a = Node a [Tree a]
```

Zusammenfassung

- `type` erzeugt Synonyme existierender Datentypen
- Algebraische Datentypen mit `data`
  - Disjunktion (Summen) von Werten: Aufzählungen
  - Kartesische Produkte: Konstruktoren mit Argumenten
  - komplexe Datentypen aus Produkten **und** Summen
- Fehlerbehandlung mit `Maybe`
- Rekursive Algebraische Datentypen
  - rekursive Verwendung des definierten Typs
  - unendlich große Strukturen (Listen, Bäume & Co.)

nächstes Mal…

- Algebraische Datentypen
  - `newtype`
  - *Records*
- Typklassen und Instanzen
  - `class`, `instance`
  - `data` … `deriving`
- Rekursive und zyklische Datenstrukturen
  - Traversion von Labyrinthen
  - variadische Bäume