Programmierung und Modellierung mit Haskell

Benutzerdefinierte Datentypen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

30. April 2015

Tupel / Kartesisches Produkt Wdh.

Kartesisches Produkt: $A \times B = \big\{(a,b) \;\big|\; a \in A \text{ und } b \in B\big\}$

Beispiel:

$$\{1,7\} \times \{\diamondsuit,\heartsuit,\clubsuit\} = \big\{(1,\diamondsuit),(1,\heartsuit),(1,\clubsuit),(7,\diamondsuit),(7,\heartsuit),(7,\clubsuit)\big\}$$

Für endliche Mengen gilt: $|A \times B| = |A| \cdot |B|$

In Haskell können wir Produkte einfach nutzen:

```
> :t (3,True)
(Integer, Bool)

> :t ("Hello",'a',(True,True))
(String, Char, (Bool, Bool))
```

Sowohl den Typ eines Produktes als auch dessen Werte schreiben wir mit runden Klammern.

Typabkürzungen Wdh

Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer Typen, eignen sich aber nicht so gut für die Daten Modellierung!

Typabkürzungen WDH

Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?

```
type Circle =  (Double, Double, Double)
```

PROBLEM: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

MERKE: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer Typen, eignen sich aber nicht so gut für die Daten Modellierung!

TYPABKÜRZUNGEN WDH

Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  ```
  type Circle =  (Double, Double, Double)
  ```
- Ein Punkt in einem dreidimensionalen Raum?
  ```
  type Point3D = (Double, Double, Double)
  ```

PROBLEM: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

MERKE: Typabkürzungen mit type erhöhen die Lesbarkeit langer Typen, eignen sich aber nicht so gut für die Daten Modellierung!

Typabkürzungen

Wdh

Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
```
type Circle =  (Double, Double, Double)
```
- Ein Punkt in einem dreidimensionalen Raum?
```
type Point3D = (Double, Double, Double)
```
- Eckdaten einer Spielfigur in einem Spiel?
```
type Monster = (Double, Double, Double)
```

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer Typen, eignen sich aber nicht so gut für die Daten Modellierung!

Typabkürzungen

WDH

Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  ```
  type Circle =  (Double, Double, Double)
  ```
- Ein Punkt in einem dreidimensionalen Raum?
  ```
  type Point3D = (Double, Double, Double)
  ```
- Eckdaten einer Spielfigur in einem Spiel?
  ```
  type Monster = (Double, Double, Double)
  ```

PROBLEM: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

MERKE: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer Typen, eignen sich aber nicht so gut für die Daten Modellierung!

Typabkürzungen Wdh

Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  ```
  type Circle =  (Double, Double, Double)
  ```
- Ein Punkt in einem dreidimensionalen Raum?
  ```
  type Point3D = (Double, Double, Double)
  ```
- Eckdaten einer Spielfigur in einem Spiel?
  ```
  type Monster = (Double, Double, Double)
  ```

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer Typen, eignen sich aber nicht so gut für die Daten Modellierung!

Typabkürzungen

Was bedeutet der Wert `(17.3, 17.3, 80.0)`?

- Kreis, repräsentiert mit Mittelpunkt-Koordinaten und Radius?
  `type Circle =  (Double, Double, Double)`
- Ein Punkt in einem dreidimensionalen Raum?
  `type Point3D = (Double, Double, Double)`
- Eckdaten einer Spielfigur in einem Spiel?
  `type Monster = (Double, Double, Double)`

Problem: Keine Typsicherheit! Don't do this!
Typakürzung sind transparent, d.h. Werte des Typs `Point3D` können (versehentlich) auch dort verwendet werden, wo `Circle` oder `Monster` erwartet wird!

Merke: Typabkürzungen mit `type` erhöhen die Lesbarkeit langer Typen, eignen sich aber nicht so gut für die Daten Modellierung!

Benutzerdefinierte Datentypen

Haskell erlaubt benutzerdefinierte Datentypen. Der Programmierer erstellt einen neuen Typ und legt dessen Werte fest.

```
data MeinTyp = MeinKonstruktor Int Bool
```

Datentypdeklaration beginnen mit Schlüsselwort `data`, dann ein frischer Name für den neuen Typ, beginnend mit Großbuchstabe.

Dann folgt der **Konstruktor**, mit einer Auflistung der Typen seiner Argumente. Konstruktoren kann man als Funktionen betrachten, die Werte des neuen Typs konstruieren:

```
> :t MeinKonstruktor
MeinKonstruktor :: Int -> Bool -> MeinTyp
```

- Konstruktoren beginnen immer mit einem Großbuchstaben.
  Infix-Konstruktoren beginnend mit `:` sind auch erlaubt.
- Bezeichner von Konstruktoren und Typen dürfen gleich sein;
  Konstruktoren sind Ausdrücke;
  Ausdrücke und Typen leben in verschiedenen Welten.

Konstruktoren in Mustervergleichen

```
data MeinTyp = MeinKonstruktor Int Bool
```

Haskell kann solchen benutzerdefinierter Datentypen zunächst wenig anfangen; nicht einmal Bildschirmausgabe ist möglich.

Mit Pattern-Matching können die Argumente eines Konstruktors wieder ausgepackt werden, zum Beispiel um eine Funktion zur Umwandlung in einen String zu schreiben:

```
myShow :: MeinTyp -> String
myShow (MeinKonstruktor i True)  = "T(" ++ (show i)++")"
myShow (MeinKonstruktor i False) = "F(" ++ (show i)++")"
```

MERKE:

- Konstruktoren mit Argumenten zum Matching immer in runde Klammern fassen. Da `(:)` ein Konstruktor ist, schreiben wir Listen-Patterns auch `(kopf:rumpf)` und *nicht* `[k:r]`
- Reihenfolge der Argumente beachten!

Konstruktoren in Mustervergleichen

```
data MeinTyp = MeinKonstruktor Int Bool
  deriving (Show, Eq, Ord)
```

Haskell kann solchen benutzerdefinierten Datentypen zunächst
w…
M…
w…
Umwandlung in einen String zu schreiben.

Tipp: GHC kann solche Funktionen automatisch erzeugen. Das `deriving (Show)` hinter der Deklaration erzeugt z.B. Funktion `show :: MeinTyp -> String` zur Bildschirmausgabe. Die **Typklassen** `Show`, `Eq`, … behandeln wir im nächsten Kapitel.

```
myShow :: MeinTyp -> String
myShow (MeinKonstruktor i True)  = "T(" ++ (show i)++")"
myShow (MeinKonstruktor i False) = "F(" ++ (show i)++")"
```

Merke:

- Konstruktoren mit Argumenten zum Matching immer in runde Klammern fassen. Da `(:)` ein Konstruktor ist, schreiben wir Listen-Patterns auch `(kopf:rumpf)` und *nicht* `[k:r]`
- Reihenfolge der Argumente beachten!

Konstruktoren in Mustervergleichen

```
data MeinTyp = MeinKonstruktor Int Bool
  deriving (Show, Eq, Ord)
```

Haskell kann solchen benutzerdefinierter Datentypen zunächst w… M… w… Umwandlung in einen String zu schreiben.

Tipp: GHC kann solche Funktionen automatisch erzeugen. Das `deriving (Show)` hinter der Deklaration erzeugt z.B. Funktion `show :: MeinTyp -> String` zur Bildschirmausgabe. Die **Typklassen** `Show`, `Eq`, … behandeln wir im nächsten Kapitel.

```
myShow :: MeinTyp -> String
myShow (MeinKonstruktor i True)  = "T(" ++ (show i)++")"
myShow (MeinKonstruktor i False) = "F(" ++ (show i)++")"
```

MERKE:

Bei Fehlermeldungen wie:

```
No instance for (Show MeinTyp) arising from ...
```

einfach Typdeklaration um `deriving (Show)` erweitern.

- Reihenfolge der Argumente beachten!

Beispiel: Verwechslung verhindern

Drei Datentypen, welche alle lediglich ein Tripel von `Double` sind, jedoch unterschiedlich zu nutzen sind:

```
data Circle  = Circle  Double Double Double
data Point3D = Point3D Double Double Double
data Monster = Monster Double Double Double

myCircle :: Circle
myCircle = Circle 17.3 17.3 80.0

hydralisk :: Monster
hydralisk = Monster 17.3 17.3 80.0

area :: Circle -> Double
area (Circle _ _ r) = pi * r^2

> area myCircle
20106.192982974677

> area hydralisk
<interactive>:13:6:
    Couldn't match expected type `Circle' with actual type `Monster'
    In the first argument of `area', namely `hydralisk'
    In the expression: area hydralisk
```

Aufzählungen

Ein Datentyp darf auch mehrere Konstruktoren besitzen:

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

Das | liest man als "oder": Ein Wert des Typs `Bool` wurde entweder mit dem Konstruktor `True` oder dem Konstruktor `False` konstruiert. Die Reihenfolge der Konstruktoren innerhalb der Deklaration ist meist egal. Ausnahme: `deriving (Ord,Enum)`

Spezialfall: Konstruktoren ohne Argumente
Falls alle Konstruktoren einess Datentyps keine Argumente haben, wie im Beispiel `Bool`, dann bezeichnet man diesen Datentyp auch als **Aufzählung** (engl. **Ennumeration**).
Ein Wert eines Aufzählungs-Typen ist immer eine von endlich vielen möglichen Konstanten.

Beispiel: Aufzählung

Aufzählungen werden wie gewohnt mit Pattern-Matching verarbeitet:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

next :: Day -> Day
next Mon = Tue
next Tue = Wed
next Wed = Thu
next Thu = Fri
next Fri = Sat
next Sat = Sun
next Sun = Mon
```

- Datentyp `Day` hat 7 Konstuktoren
- Funktion `next` zählt einfach einen Wochentag weiter

Beispiel: Aufzählung

Aufzählungen werden wie gewohnt mit Pattern-Matching verarbeitet:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Enum)

next :: Day -> Day
next Sun = Mon
next d   = succ d

-- Unter Anderen durch 'deriving' generiert:
succ, pred :: Enum a => a -> a
toEnum     :: Enum a => Int -> a
```

- Datentyp `Day` hat 7 Konstuktoren
- Funktion `next` zählt einfach einen Wochentag weiter

Kombination: Alternativen & Argumente

```
data Frucht = Apfel (Int,Int)           -- 1 Argument
            | Birne  Int Int            -- 2 Argumente
            | Banane Int Int Double     -- 3 Argumente

meineFrüchte :: [Frucht]
meineFrüchte = [Apfel (2,90), Apfel (3,300),
                Birne 60 1, Banane 80 7 0.3]

hatApfel :: [Frucht] -> Bool
hatApfel           []    = False
hatApfel ((Apfel _):_)  = True
hatApfel (  _      :t)  = hatApfel t

> hatApfel meineFrüchte
True
> hatApfel (drop 2 meineFrüchte)
False
```

Kombination: Alternativen & Argumente

```
data Frucht = Apfel (Int,Int)           -- 1 Argument
            | Birne  Int Int            -- 2 Argumente
            | Banane Int Int Double     -- 3 Argumente
```

Es empfiehlt sich meist, Alternativen in einer eigenen Funktion zu behandeln:

```
gesamtPreis :: [Frucht] -> Int
gesamtPreis []    = 0
gesamtPreis (h:t) = preis h + gesamtPreis t

preis :: Frucht -> Int
preis (Apfel (z,p))  = z * p
preis (Birne  p z )  = z * p
preis (Banane p z _) = z * p

> gesamtPreis meineFrüchte
1700
```

Kombination: Alternativen & Argumente

```
data Frucht = Apfel (Int,Int)          -- 1 Argument
            | Birne  Int Int           -- 2 Argumente
            | Banane Int Int Double    -- 3 Argumente
```

Es empfiehlt sich meist, Alternativen in einer eigenen Funktion zu beha

Schlechte Modellierung!

```
gesamtPreis :: [Frucht] -> Int
gesamtPreis []    = 0
gesamtPreis (h:t) = preis h + gesamtPreis t

preis :: Frucht -> Int
preis (Apfel (z,p))  = z * p
preis (Birne  p z )  = z * p
preis (Banane p z _) = z * p

> gesamtPreis meineFrüchte
1700
```

Duplikation vermeiden

Schlecht:

```
data Frucht = Apfel (Int,Int)
            | Birne  Int Int
            | Banane Int Int Double
```

Wenn ein Argument mit der gleichen Bedeutung in allen Konstruktoren vorkommt, sollte man den Datentyp meist unterteilen, um Redundanzen im Code zu vermeiden.

Besser:

```
data Frucht = Frucht Sorte Int Int
data Sorte  = Apfel | Birne | Banane Double

preis  :: Frucht -> Int
preis  (Frucht _ p _) = p
anzahl :: Frucht -> Int
anzahl (Frucht _ _ z) = z
```

Duplikation vermeiden

Schlecht:

```
data Frucht = Apfel (Int,Int)
            | Birne  Int Int
            | Banane Int Int Double
```

Wenn ein Argument mit der gleichen Bedeutung in allen Konstruktoren vorkommt, sollte man den Datentyp meist unterteilen, um Redundanzen im Code zu vermeiden.

Besser:

```
data Frucht = Frucht Sorte Int Int
data Sorte  = Apfel | Birne | Banane Double

hatApfel ((Frucht Apfel _ _):_) = True
hatApfel (_:t)                  = hatApfel t
hatApfel  []                    = False
```

Dank Verschachtelter Patterns entsteht kein Nachteil.

Records

Eine weitere Alternative bietet die **Record**-Syntax:

```
data Frucht = Apfel  { preis::Int, anzahl::Int } -- 2 Arg.
            | Birne  { preis::Int, anzahl::Int } -- 2 Arg.
            | Banane { preis::Int, anzahl::Int
                     , krümmung::Double        } -- 3 Arg.
```

Jedes Argument eines Konstruktors bekommt einen Namen, diese werden dann auch als **Felder** bezeichnet.

- Namen sind sinnvoll, wenn man viele Argumente hat
- Reihenfolge der Felder in geschweiften Klammern ist egal
- **Projektion** für jedes Feld werden automatisch definiert; hier:

```
preis    :: Frucht -> Int
anzahl   :: Frucht -> Int
krümmung :: Frucht -> Double
```

  Warnung: partiell

- Auch Pattern-Matching erlaubt dann Record-Syntax; Patterns mit `{ }` müssen nicht alle Felder matchen

Records

Eine weitere Alternative bietet die **Record**-Syntax:

```
data Frucht = Apfel  { preis::Int, anzahl::Int } -- 2 Arg.
            | Birne  { preis::Int, anzahl::Int } -- 2 Arg.
            | Banane { preis::Int, anzahl::Int
                     , krümmung::Double        } -- 3 Arg.
```

Datentypen können leicht nachträglich zu Records gemacht werden, da die Record-Syntax überall optional ist.
Werden mal keine geschweiften Klammern und Feldnamen verwendet, gilt die Reihenfolge in der Definition wie üblich.

Obige Datentypdeklaration in Record-Syntax erlaubt also auch alles, welches folgende gewöhnliche Deklaration ermöglichen würde:

```
data Frucht = Apfel  Int Int
            | Birne  Int Int
            | Banane Int Int Double
```

Rekursive Datentypen

Typdeklarationen dürfen auch (wechselseitig) rekursiv sein.

Beispiel: Listen

Konstruktor `ListKnoten` hat ein rekursives Argument `IntList`

```
data IntList = LeereListe | ListKnoten Int IntList

myList :: IntList
myList = ListKnoten 1 (ListKnoten 2 (LeereListe))

mySum :: IntList -> Int
mySum LeereListe = 0
mySum (ListKnoten h t) = h + mySum t

> mySum myList
3
```

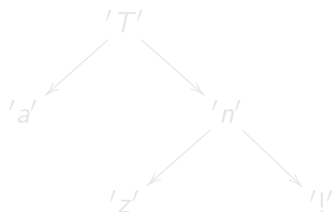
Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum
```

```
> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum
```

```
myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
           (Knoten (Blatt 'z') 'n' (Blatt '!'))
```



Terminologie

- **Wurzel**-knoten `'T'`
- **Blätter** `'a'`,`'z'`,`'!'`
- **linker Teilbaum** von **Knoten** `'n'` ist das Blatt `'z'`

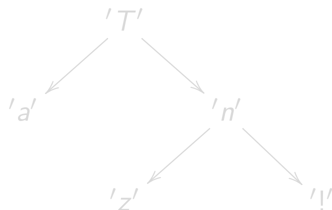
Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum

> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum

myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))
```



Terminologie

- Wurzel-knoten `'T'`
- Blätter `'a'`,`'z'`,`'!'`
- linker Teilbaum von Knoten `'n'` ist das Blatt `'z'`

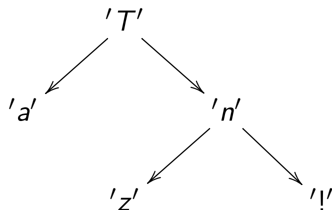
Rekursive Datentypen

Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum

> :type Knoten
Knoten :: Baum -> Char -> Baum -> Baum

myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
         (Knoten (Blatt 'z') 'n' (Blatt '!'))
```



Terminologie

- **Wurzel**-knoten `'T'`
- **Blätter** `'a'`,`'z'`,`'!'`
- **linker Teilbaum** von **Knoten** `'n'` ist das Blatt `'z'`

Rekursive Datentypen

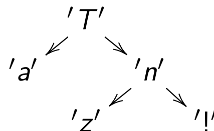
Beispiel: Binärbäume

```
data Baum = Blatt Char | Knoten Baum Char Baum

myBaum :: Baum
myBaum = Knoten (Blatt 'a') 'T'
          (Knoten (Blatt 'z') 'n' (Blatt '!'))

dfCollect :: Baum -> String
dfCollect (Blatt c) = [c]
dfCollect (Knoten links c rechts)
          = c : dfCollect links ++ dfCollect rechts

> dfCollect myBaum
"Tanz!"
```



Wechelseitig Rekursiv

Beispiel:

```
data Datei = Datei String | Verzeichnis Dir
data Dir   = Local String [Datei] | Remote String

verzeichnis  :: Dir
verzeichnis = Local "root"
                 [ Datei "info.txt"
                 , Verzeichnis (Remote "my.url/work")
                 , Verzeichnis (Local "tmp" [])
                 , Datei "help.txt"
                 ]
```

- Reihenfolge der Definition ist egal
- Ganz normale Verwendung

Bemerkung Mann könnte hier auch nur einen Datentyp mit 3 Alternativen definieren, doch dann könnte man keine Funktionen schreiben, welche nur gezielt Verzeichnisse bearbeiten.

Typparameter – Typdeklarationen mit “Loch”

Datentypen können **Typvariablen** als Parameter verwenden:

Beispiel: Listen

```
data List a = Leer | Element a (List a)

iList :: List Int
iList = Element 1 (Element 2 (Leer))

iSum :: List Int -> Int
iSum Leer = 0
iSum (Element h t) = h + iSum t

type IntList = List Int -- Typspezialisierung
```

- Typvariablen werden immer klein geschrieben
- `List` bezeichnen wir als **Typkonstruktor**. Nur durch Anwendung auf einen Typ wird ein Typ daraus: `List Int`.

Typparameter – Typdeklarationen mit "Loch"

Datentypen können **Typvariablen** als Parameter verwenden

Beispiel: Listen

```
data List a = Leer | Element a (List a)

myLength :: (List a) -> Int
myLength Leer = 0
myLength (Element _ t) = 1 + myLength t

> myLength (Element 'a' (Element 'b' Leer))
2
```

Im Gegensatz zu `iSum :: List Int -> Int` kann die Funktion `myLength:: List a -> Int` mit Listen umgehen, die einen beliebigen Typ in sich tragen.

Solche Funktionen, bei denen Typvariablen in der Signatur auftauchen, nennt man auch **polymorph**.

Ausblick: Polymorphe Funktionen

Beispiele polymorpher Funktionen aus der Standardbibliothek:

```
id :: a -> a
id x = x

fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y

replicate :: Int -> a -> [a]

drop :: Int -> [a]-> [a]

zip :: [a] -> [b] -> [(a,b)]
```

Maybe

Maybe ist ein wichtiger polymorpher Datentyp der Standardbibliothek

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

Damit können wir auf eine sichere Weise ausdrücken, dass eine Berechnungen fehlschlagen kann.

Beispiel

```
data Frucht = Apfel  { preis::Int, anzahl::Int}
            | Birne  { preis::Int, anzahl::Int}
            | Banane { preis::Int, anzahl::Int
                     , krümmung::Double       }

getKrümmung :: Frucht -> Maybe Double
getKrümmung Banane {krümmung=k} = Just k
getKrümmung _                   = Nothing
```

Maybe

Beispiele

```
data Maybe a = Nothing | Just a

isJust         :: Maybe a -> Bool
isJust Nothing = False
isJust _       = True

fromMaybe :: a -> Maybe a -> a
fromMaybe standardWert Nothing  = standardWert
fromMaybe      _       (Just x) = x

catMaybes :: [Maybe a] -> [a]
catMaybes ls = [x | Just x <- ls]
```

Either

Polymorphe Datentypen können auch mehrere Typparameter haben. `Either` ist ein wichtiges Beispiel:

```
data Either a b  =  Left a | Right b
```

Zum Beispiel kann `Either a String` anstatt `Maybe a` für die Rückgabe von Fehlermeldungen verwendet werden, ohne die Berechnung komplett abzubrechen.

`Either` ist der Typ für disjunkte Vereinigungen:

$$A_1 \dot{\cup} A_2 = \{(i, a) \mid a \in A_i\}$$

Beispiel:

$$\{\diamondsuit, \heartsuit\} \dot{\cup} \{\clubsuit, \spadesuit, \heartsuit\} = \{(1, \diamondsuit), (1, \heartsuit), (2, \clubsuit), (2, \spadesuit), (2, \heartsuit)\}$$

Für endliche Mengen gilt: $|A_1 \dot{\cup} A_2| = |A_1| + |A_2|$

Man spricht daher auch von "Summentypen"

Either

Beispiele

```
data Either a b  =  Left a | Right b

isRight :: Either a b -> Bool
isRight (Left  _) = False
isRight (Right _) = True

lefts   :: [Either a b] -> [a]
lefts x = [a | Left a <- x]

partitionEithers :: [Either a b] -> ([a],[b])
partitionEithers [] = ([],[])
partitionEithers (h : t) =
  let (ls,rs) = partitionEithers t
  in case h of  Left l  -> (l:ls,   rs)
                Right r -> (  ls, r:rs)
```

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort data
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter

  optionale Alternativen

  frischer Konstruktor – muss mit Großbuchstaben beginnen

  beliebige Typen oder Typparameter als Argumente

  optionale deriving Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale deriving-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale deriving-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
   = Konstruktor1 arg_11 ... arg_1i
   | Konstruktor2 arg_21 ... arg_2j
   | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale deriving-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale deriving-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort data
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale deriving-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
```

- Schlüsselwort `data`
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale deriving-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale `deriving`-Klausel mit Liste von Typklassen

Datentypdeklaration

Syntax der Datentypdeklaration:

```
data Typname par_1 ... par_m
  = Konstruktor1 arg_11 ... arg_1i
  | Konstruktor2 arg_21 ... arg_2j
  | Konstruktor3 arg_31 ... arg_3k
  deriving (class_1, ..., class_l)
```

- Schlüsselwort `data`
- frischer Typname – beginnt immer mit Großbuchstabe
- optionale Typparameter
- optionale Alternativen — lies | als "oder"
- frischer Konstruktor – muss mit Großbuchstaben beginnen
- beliebige Typen oder Typparameter als Argumente
- optionale `deriving`-Klausel mit Liste von Typklassen

Programmierung und Modellierung mit Haskell

Benutzerdefinierte Datentypen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

7. Mai 2015

Zusammenfassung: Datentypen

- Ein Typ (oder **Datentyp**) ist eine Menge von Werten
- Unter einer **Datenstruktur** versteht man einen Datentyp plus alle darauf verfügbaren Operationen
- Moderne Programmiersprachen ermöglichen, dass der Benutzer neue Typen definieren kann
- Datentypen können andere Typen als Typparameter haben
- Datentypen können (wechselseitig) rekursiv definiert werden
- **Konstruktoren** können als Funktionen betrachtetet werden
- **Records** erlauben Benennung der Konstruktorargumente
- Typdeklarationen:
  - `data` Deklaration wirklich neuer Typen
  - `type` Typabkürzungen nur für Lesbarkeit für Menschen

Informelle Beschreibung

Informell könnte man sagen: Konstruktoren packen Päckchen mit ihren Argumenten und kleben ein Etikett mit Ihren Namen drauf. Pattern Matching packt Päckchen aus, wenn das Etikett stimmt.

Beispiel:

```
data S   = SL1 Double | SL2 Int Int | SL3     deriving Show
data T a = TL1 a | TL2 a (T a)                deriving Show

umpacker :: S -> T Double
umpacker (SL1 d  ) = TL1 d
umpacker (SL3    ) = TL1 0.0
umpacker (SL2 x y) = TL2 xd (TL1 yd)
  where xd = fromIntegral x
        yd = fromIntegral y
```

Funktion `umpacker` erwartet Päckchen der "Sorte" `S`. Es wird das Etikett inspiziert. Wenn das Etikett `SL1` ist, wird der Inhalt ausgepackt und in ein Päckchen mit Etikett `TL1` umverpackt. Dieses Päckchen ist dann von der Sorte `T Double`.

Vergleich der Notation

Vergleich der Notation zwischen dem eingebauten Listentyp `[a]` und einem äquivalenten, selbst-definierten Typ `List a`, den wir verwenden könnten, wenn es keine eingebauten Listen gäbe:

```
data [a]     = []   | (:)     a [a]
data List a = Leer | Element a (List a)
```

```
f1 :: [a] ->  ...        f2 :: List a  ->   ...
f1 []     =   ...        f2 Leer          =  ...
f1 (h:t)  =   ...        f3 (Element h t) =  ...
```

- `List a` ist ein rekursiver Datentyp, da es einen Konstruktor gibt, der ein Argument dieses Typs fordert.
- `List a` ist ein polymorpher Datentyp, da es einen Typparameter gibt. Dieser erlaubt es uns, Listen über beliebige andere Typen als Inhalt zu bilden.

Klammerung im Pattern Matching

Im Pattern-Matching Konstruktoren mit Argumenten immer klammern; Reihenfolge der Argumente ist immer zu beachten!

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b

foo :: Maybe Int -> Either Bool String -> String
foo  Just x   Left y   = "NOT OK"  --  Syntaxfehler
foo (Just x) (Left y)  = "OK"
foo Nothing  (Right _) = "OK"
```

Konstruktoren ohne Argumente müssen im Pattern-Matching nicht geklammert werden.

Klammerung bei Mustervergleich mit Listen

Schreibweise `[1,2,3]` ist erlaubte Kurzform für `1:2:3:[]`

`(:)` ist ein Infix-Konstruktor;

Konstruktoren mit Argumenten sind im Pattern zu klammern!

```
bar :: [a] -> String
bar []           = "Leere Liste"
bar [x]          = "Genau 1 Element"
bar [x,y]        = "Genau 2 Elemente"
bar [x,y,z]      = "Genau 3 Elemente"
bar (x:xs)       = "Mindestens 1 Element"
bar (x:(y:zs)) = "Mindestens 2 Elemente"
bar (x:y:zs)     = "Mindestens 2 Elemente" --eine () reicht
bar (x:y:z:ls) = "Mindestens 3 Elemente"
```

Typen der Variablen im Beispiel:

```
x,y,z     ::  a    -- ein Element
xs,zs,ls :: [a]    -- eine Liste
```

Listenbezeichner oft mit englischem Plural-s: ein `x`, mehrere `xs`

Patterns vor Wächter

Tipp: Pattern Matching bevorzugen vor Wächtern:

```
data Maybe a = Nothing | Just a

fooOK :: Maybe a -> Int
fooOK Nothing  = 0
fooOK (Just x) = 1

fooBAD :: Eq a => Maybe a -> Int
fooBAD x | x == Nothing = 0
         | otherwise    = 1
```

Anwendung von `(==)` erzwingt Einschränkung auf Typklasse `EQ`: für Gleichheit muss der komplette Wert betrachtet werden!

Bei Pattern-Matching wird dagegen nur der Konstruktor verglichen, aber nicht notwendigerweise der gesamte Wert!

Anhänge

Die folgenden Folien behandeln nützliche Besondertheiten von GHC, welche einem die Programmierung erleichtern können. Diese Folien sind nicht prüfungsrelevant und wurden in der Vorlesung auch nicht behandelt.

Hinweis: die Beherrschung der Record-Syntax im Rahmen von Folie 04-12ff. wird schon erwartet, d.h. Record-Syntax zumindest lesen können und eventuell auch Projektionen verwenden können.

Wenn Sie möchten, dürfen Sie die vorgestellten Techniken in Ihren Hausübungen verwenden.

NEWTYPE

Für Datentypen mit *genau einen* Konstruktor der *genau einem* Argument hat, bietet sich newtype an:

```
newtype Circle  = Circle  (Double, Double, Double)
newtype Point3D = Point3D (Double, Double, Double)
newtype Monster = Monster (Double, Double, Double)
```

Anstatt dem Schlüsselwort data kann dann einfach newtype verwendet werden.

- newtype ist ein optimierter Spezialfall: zur Laufzeit keine Unterscheidungen zwischen dem Typ und Newtyp.
- Nicht so strikt wie äquivalente Datentypdeklaration
- Eigene Klassen Instanzen möglich; ebenso automatische Übernahme der Klasseninstanzen des ursprünglichen Typs

mit Erweiterung

Records

Wenn ein Konstruktor viele Argumente hat, kann deren Auflistung schnell unübersichtlich werden.

```
data Person' = Person'  String Int Int Int Int
p0 = Person' "Tyrion" 135 26 7 0

data Person  = Person { name::String, height,age,mates,offspring::Int }
p2 = Person  {height=166, age=35, offspring=3, mates=3, name="Cersei"}
p3 = Person  "Jaimie" 187 35 1 3      -- alte Syntax auch noch erlaubt
```

Record Notation erlaubt es, die **Argumente eines Konstruktors** zu benennen; diese werden dann auch als **Felder** bezeichnet.

- Reihenfolge der Felder innerhalb der geschweiften Klammern ist immer beliebig
- Datentypen können nachträglich zu Records gemacht werden: Werden mal keine geschweiften Klammern verwendet, muss auf Feldnamen verzichtet werden und es gilt die Reihenfolge gemäß Definition wie üblich.

Pattern Matching mit Records

Pattern Matching darf Record Syntax verwenden;
die Reihenfolge der Felder ist beliebig:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

showPerson :: Person -> String
showPerson Person { age=a, name=n } = n ++ ' ':show a

hasChildren :: Person -> Bool
hasChildren Person { offspring=n } | n > 0 = True
hasChildren _ = False

> showPerson p2
"Cersei 35"
```

Das Matching darf auch partiell sein, d.h. es müssen nicht alle Felder gemachted werden.

Record Projektionen

Projektionen werden für jeden Feldnamen *automatisch* definiert:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
> :t name
name :: Person -> String

> name p3
"Jaimie"

> :t age
age :: Person -> Int

> age p3
35
```

Record Pseudoupdate

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
p1 = Person "Tyrion" 135 26 7 0
```

Funktionale “Field-updates” sind ebenfalls möglich. Dabei werden natürlich Kopien erstellt — denn bestehende Werte werden in der funktionalen Welt ja nie verändert!

```
p4 = p1 { name = "Imp" }
p5 = p1 { mates = 2 + mates p1 }

> p5
Person {name = "Tyrion", height = 135, age = 26,
        mates = 9, offspring = 0}
> p1
Person {name = "Tyrion", height = 135, age = 26,
        mates = 7, offspring = 0}
```

Record Refactoring

Records kann man bei Bedarf nachträglich leicht erweitern.

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

p6 = Person { name="Daenerys", height=157, offspring=0 }
```

ist gültiger Code (wenn z.B. `age` und `mates` nachträglich in die Definition aufgenommen wurden). Es wird interpretiert als:

```
p6 = Person "Daenerys" 157 undefined undefined 0
```

GHC gibt in diesem Fall aber entsprechende Warnungen heraus, dass nicht alle Felder initialisiert wurden.

Dies Warnungen sollte man nach einer solchen nachträglich Erweiterung des Record-Typs auch abarbeiten.

Standardwerte für Records definieren

Um die Wartbarkeit des Codes zu erhöhen ist es meist ratsam, einen globalen Record mit Default-Werten anzulegen:

```
data Foo = Foo { bar :: Int, baz :: Int, quux :: Int }

fooDefault = Foo { bar = 1, baz = 2, quux = 3 }

newRecord = fooDefault { bar = 69, baz = 42 }
```

Ansatt einen neuen Record anzulegen, wird lediglich ein Field-Update des Standardwerts durchgeführt

Fügt man später ein weiteres Feld ein, dann muss man lediglich den Default-Wert anpassen.
Nachteil: Der Kompiler gibt dann keine Warnungen mehr heraus, wenn nicht alle Felder initialisiert werden.