

Algebraische Datentypen

Funktionale Programmierung

Prof. Dr. Oliver Braun

Letzte Änderung: 09.10.2018 06:54

- in Haskell gibt es Werte (auch Funktionen sind in Haskell Werte)
- alle Werte haben einen Typ, z.B.

```
True :: Bool
```

```
1 :: Num p => p
```

```
(++) :: [a] -> [a] -> [a]
```

- d.h. es gibt die Ebene der Werte und dazu eine Ebene von Typen
- auch Typen können zusammengesetzt werden, auch mit Typen kann man “rechnen”

- Typen klassifizieren Werte
- aber auch Typen (Typkonstruktoren) fallen in verschiedene Klassen
- beispielsweise sind **Bool**, **Int** und **(Int, Float)** “fertige” Typen (Typkonstruktoren)
- aber z.B. **[]** oder **Maybe** sind noch nicht “fertig”
- um Regeln zu definieren, wie Typen zusammgebaut werden, brauchen wir eine Klassifizierung von Typen:
 - **Kinds**

- alle “fertigen” Typen haben den Kind $*$, z.B.

```
Prelude> :k Int
```

```
Int :: *
```

```
Prelude> :k [Maybe Bool]
```

```
[Maybe Bool] :: *
```

- der Kind von Typkonstruktoren, die noch Argumente benötigen, wird analog zu Typen von Funktionen notiert, z.B.

```
[] :: * → * -- Achtung: nicht die leere Liste
```

```
Maybe :: * → *
```

```
Either :: * → * → *
```

- damit haben wir Regeln wie Typen zusammengesetzt werden können

Beispiele

```
Prelude> :k Int
```

```
Int :: *
```

```
Prelude> :k Maybe
```

```
Maybe :: * → *
```

```
Prelude> :k Maybe Int
```

```
Maybe Int :: *
```

```
Prelude> :k Maybe Maybe
```

```
<interactive>:1:7: error:
```

- Expecting one more argument to 'Maybe'
Expected a **type**, but 'Maybe' has kind '*** → ***'
- In the first argument **of** 'Maybe', namely 'Maybe'
In the **type** 'Maybe Maybe'

```
Prelude> :k Maybe (Maybe Int)
```

```
Maybe (Maybe Int) :: *
```

Und wo ist die Algebra?

- Summentypen

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

- Produkttypen

```
data Point = Point Int Int
```

```
Prelude> :t True
True :: Bool
Prelude> data Point = Point Int Int
Prelude> :t Point
Point :: Int → Int → Point
Prelude> :t Point 3
Point 3 :: Int → Point
Prelude> :t Point 3 5
Point 3 5 :: Point
Prelude> map (uncurry Point) [ (x,y) | x ← [1..2]
                               , y ← [1..2]]
[Point 1 1,Point 1 2,Point 2 1,Point 2 2]
Prelude> map (`Point` 12) [1..3]
[Point 1 12,Point 2 12,Point 3 12]
```

Getter!? Really? Setter? Not Really!

```
data Point = Point Int Int
```

```
getX (Point x _) = x
```

```
getY (Point _ y) = y
```

```
-- Achtung: Da wird nichts gesetzt ⇒ NEUER Punkt!!!
```

```
setX (Point xOld y) x = Point x y
```

```
setY (Point x yOld) y = Point x y
```

- kann das der Compiler nicht automatisch generieren?


```
data Point = Point
  { x :: Int
  , y :: Int
  }
```

- der Compiler generiert daraus die Funktionen

```
x :: Point → Int
y :: Point → Int
```

- Zugriff

```
Prelude> p = Point 5 3
Prelude> x p
5
```

- Record Update (neuer Point!)

```
Prelude> p = Point 5 3
Prelude> q = p { y = 7 }
Prelude> p
Point {x = 5, y = 3}
Prelude> q
Point {x = 5, y = 7}
```

- Pattern Matching nach wie vor unverändert möglich

type vs. data

- **type** erzeugt Typsynonym, **data** erzeugt neuen (komplexen) Datentyp
- **type** bietet kein Plus an Typsicherheit, z.B.

```
type Grade = Int
```

- damit kann aber jede Funktion die auf einen **Int** angewendet werden, auch auf einen **Grade** angewendet werden
- im Typsystem kein Unterschied
- **data** hat Overhead, z.B.

```
data Grade = Grade Int
```

- nur speziell für **Grade** definierte Funktionen können auf einen **Grade** angewendet werden

Vorteile von beiden — **newtype** (1/2)

- **newtype** erschafft einen neuen Typ, wie **data**, hat aber zur Laufzeit keinen Overhead, wie **type**, z.B.

```
newtype Grade = Grade Int
```

- Nachteil: **newtype** kann nur **einen** Datenkonstruktor mit **einem** Parameter haben
 - d.h. kein Summentyp und auch kein Produkttyp
- Vorteil: der Typchecker überprüft den Typ mit **Grade**, aber zur Laufzeit bleibt nur noch der **Int** übrig

Vorteile von beiden — `newtype` (2/2)

- weiterer Vorteil gegenüber `type`:
 - mit `newtype` können Instanzen von Typklassen implementiert bzw. abgeleitet werden
- Beispiel

```
instance Show Grade where  
  show (Grade 1) = "sehr gut"  
  show (Grade 2) = "gut"  
  ...  
  show _ = "ungültige Note"
```

- aber immer noch so “billig” wie ein `Int`

Polymorphe Funktionen selbst gemacht (1/5)

- wollen Studenten und Mitarbeiter modellieren

```
newtype Students = Students Int
```

```
newtype Staff = Staff Int
```

- wollen einheitlichen Zugriff auf die Anzahl
- nicht möglich

```
newtype Students = Students { headcount :: Int }
```

```
newtype Staff = Staff { headcount :: Int }
```

```
Multiple declarations of 'headcount'
```

- möglich wäre

```
newtype Students =  
  Students { headcountStudents :: Int }  
newtype Staff = Staff { headcountStaff :: Int }
```

- aber schöner wäre es eine Funktion `headcount` zu haben, die auf `Students` und `Staff` angewendet werden kann

- Lösung: eigene Typklasse

```
newtype Students = Students Int
```

```
newtype Staff = Staff Int
```

```
class Headcount a where
```

```
  headcount :: a → Int
```

```
instance Headcount Students where
```

```
  headcount (Students i) = i
```

```
instance Headcount Staff where
```

```
  headcount (Staff i) = i
```


- noch cooler: Instanzen eigener Typklasse herleiten lassen
- nachdem beides ja `Int` sind (`newtype`) reicht die Instanz für `Int`

```
class Headcount a where
```

```
  headcount :: a → Int
```

```
instance Headcount Int where
```

```
  headcount i = i
```

- oben im File (vor `module ...`) Spracherweiterung einschalten:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
```

- und dann reicht

```
newtype Students = Students Int  
  deriving (Headcount)
```

```
newtype Staff = Staff Int  
  deriving (Headcount)
```