fnord.hs

Sitzung 2: Algebraische Datentypen

20. August 2017

Produkttypen

- **data** Point = Point **Double Double**

Produkttypen

- **data** Point = Point **Double Double**
- **data** Point = Point { xCoord :: **Double**, yCoord :: **Double** }

Produkttypen

- **data** Point = Point **Double Double**
- **data** Point = Point { xCoord :: **Double**, yCoord :: **Double** }
- **data** Point a = Point { xCoord :: a, yCoord :: a}

Produkttypen

- **data** Point = Point **Double Double**
- **data** Point = Point { xCoord :: **Double**, yCoord :: **Double** }
- **data** Point a = Point { xCoord :: a, yCoord :: a}
- origin :: Point **Double**
  origin = Point 0 0

Produkttypen

- **data** Point = Point **Double Double**
- **data** Point = Point { xCoord :: **Double**, yCoord :: **Double** }
- **data** Point a = Point { xCoord :: a, yCoord :: a}
- origin :: Point **Double**
  origin = Point 0 0
- origin ' = Point{xCoord=0, yCoord=0}

Produkttypen

- **data** Point = Point **Double Double**
- **data** Point = Point { xCoord :: **Double**, yCoord :: **Double** }
- **data** Point a = Point { xCoord :: a, yCoord :: a}
- origin :: Point **Double**
  origin = Point 0 0
- origin ' = Point{xCoord=0, yCoord=0}
- Warum heißen Produkttypen Produkttypen? Was ist der „1-Typ“?

Produkttypen

- **data** Point = Point **Double Double**
- **data** Point = Point { xCoord :: **Double**, yCoord :: **Double** }
- **data** Point a = Point { xCoord :: a, yCoord :: a}
- origin :: Point **Double**
  origin = Point 0 0
- origin ' = Point{xCoord=0, yCoord=0}
- Warum heißen Produkttypen Produkttypen? Was ist der „1-Typ“?
- Das leere Tupel ().

Summentypen

```
data Shape = Circle { radius :: Double }
           | Rectangle { width :: Double,
                         height :: Double }

area :: Shape → Double
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h

area' c@Circle{} = pi * radius c * radius c
area' r@Rectangle{} = width r * height r
```

Warum heißen Summentypen Summentypen? Was ist der „0-Typ“?

Summentypen

```
data Shape = Circle { radius :: Double }
           | Rectangle { width :: Double,
                         height :: Double }

area :: Shape → Double
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h

area' c@Circle{} = pi * radius c * radius c
area' r@Rectangle{} = width r * height r
```

Warum heißen Summentypen Summentypen? Was ist der „0-Typ“?
data Void

Maybe

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

safeHead :: [a] → Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

startsWithOne :: [Int] → Bool
startsWithOne xs =
  case safeHead xs of
    Nothing → False
    Just h → h == 1
```

Either

```
data Either l r = Left l | Right r

radiusOrExtents :: Shape →
                   Either Double (Double,Double)
radiusOrExtents (Circle r) = Left r
radiusOrExtents (Rectangle w h) = Right (w,h)

eitherToShape :: Either Double (Double,Double)
                 → Shape
eitherToShape (Left r) = Circle r
eitherToShape (Right (w,h)) = Rectangle w h
```

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?
- Welcher Typ ist isomorph zu Either Void Int?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?
- Welcher Typ ist isomorph zu Either Void Int?
- Welcher Typ ist isomorph zu Either () Int?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?
- Welcher Typ ist isomorph zu Either Void Int?
- Welcher Typ ist isomorph zu Either () Int?
- Welcher Typ ist isomorph zu ((),Int)?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?
- Welcher Typ ist isomorph zu Either Void Int?
- Welcher Typ ist isomorph zu Either () Int?
- Welcher Typ ist isomorph zu ((),Int)?
- Welcher Typ ist isomorph zu (Void,Int)?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?
- Welcher Typ ist isomorph zu Either Void Int?
- Welcher Typ ist isomorph zu Either () Int?
- Welcher Typ ist isomorph zu ((),Int)?
- Welcher Typ ist isomorph zu (Void,Int)?
- Welcher Typ ist isomorph zu Maybe Void?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?
- Welcher Typ ist isomorph zu Either Void Int?
- Welcher Typ ist isomorph zu Either () Int?
- Welcher Typ ist isomorph zu ((),Int)?
- Welcher Typ ist isomorph zu (Void,Int)?
- Welcher Typ ist isomorph zu Maybe Void?
- Welcher Typ ist isomorph zu [Void]?

Isomorphie

- Eine Funktion f :: a → b heißt Bijektion oder Isomorphismus, wenn es eine Umkehrfunktion g :: b → a gibt mit den Identitäten:
  - f . g === **id**
  - g . f === **id**
- Zwei Typen a und b heißen isomorph, wenn es einen Isomorphismus zwischen ihnen gibt.
- Sind Shape und Either Double (Double,Double) isomorph?
- Sind Int und Maybe Int isomorph?
- Welcher Typ ist isomorph zu Either Void Int?
- Welcher Typ ist isomorph zu Either () Int?
- Welcher Typ ist isomorph zu ((),Int)?
- Welcher Typ ist isomorph zu (Void,Int)?
- Welcher Typ ist isomorph zu Maybe Void?
- Welcher Typ ist isomorph zu [Void]?
- Welcher Typ ist isomorph zu [()]?

Natürliche Zahlen

```
data Nat = Zero | Succ Nat

add :: Nat → Nat → Nat
add Zero n = n
add (Succ a) b = Succ (add a b)

zero = Zero

-- Vergleiche
data [a] = [] | a : [a]

(++) :: [a] → [a] → [a]
[] ++ xs = xs
(x:xs) ++ ys = x : (xs++ys)

empty = []
```

Monoide

Ein Monoid ist ein Typ a mit einer assoziativen Verknüpfung mappend :: a → a → a und einem neutralen Element mempty :: a.

```
class Monoid a where
  mempty :: a
  mappend :: a → a → a

instance Monoid Nat where
  mempty = Zero
  mappend = add

instance Monoid [a] where
  mempty = []
  mappend = (++)

(<>) = mappend
```

Monoide

```
mconcat :: Monoid a => [a] -> a
mconcat [] = mempty
mconcat (x:xs) = x `mappend` mconcat xs
```

```
Prelude> mconcat [Succ Zero, Succ Zero,
                  Succ (Succ Zero)]
Succ (Succ (Succ (Succ Zero)))
Prelude> mconcat [] :: Nat
Zero
Prelude> mconcat [[1,2,3],[4,5],[6,7]]
[1,2,3,4,5,6,7]
Prelude> mconcat [] :: [Int]
[]
Prelude> Succ mempty <> mempty
Succ Zero
```

Funktoren

```
mapMaybe :: (a → b) → Maybe a → Maybe b
mapMaybe _ Nothing = Nothing
mapMaybe f (Just a) = Just (f a)

mapRight :: (a → b) → Either e a → Either e b
mapRight _ (Left e) = Left e
mapRight f (Right a) = Right (f a)

-- Vergleiche
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Funktoren

```
class Functor f where
  fmap :: (a → b) → f a → f b

instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap = mapMaybe

instance Functor (Either e) where
  fmap = mapRight

(<$>) = fmap

(<$) :: Functor f ⇒ a → f b → f a
a <$ f = const a <$> f
```

Funktoren

```
Prelude> fmap (*2) [1,2,3,4]
[2,4,6,8]
Prelude> fmap (*2) (Just 5)
Just 10
Prelude> fmap (*2) Nothing
Nothing
Prelude> True <$ [1,2,3,4]
[True,True,True,True]
Prelude> 1 <$ Right "keks"
Right 1
```

Applikative Funktoren

```
Prelude> :t fmap (+)
fmap (+) :: (Num a, Functor f) ⇒ f a → f (a → a)
Prelude> :t fmap (+) [1,2,3]
fmap (++) [1,2,3] :: Num a ⇒ [a → a]
Prelude> fmap (+) [1,2,3] [4,5,6]

<interactive>:4:1: error:
 – Couldn't match expected type ‘[Integer] → t’
                  with actual type ‘[Integer → Integer]’
[...]
```

Wie wenden wir (++) jetzt auf zwei Argumente an?

Applikative Funktoren

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b

instance Applicative [] where
  pure a = [a]
  fs <*> xs = [f x | f ← fs, x ← xs]

instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  (Just f) <*> (Just x) = Just (f x)
```

Applikative Funktoren

```
Prelude> (+) <$> [1,2,3] <*> [10,20,30]
[11,21,31,12,22,32,13,23,33]
Prelude> (,,) <$> [1,2] <*> [True,False] <*> "ab"
[(1,True,'a'),(1,True,'b'),(1,False,'a'),(1,False,'b')
 (2,True,'a'),(2,True,'b'),(2,False,'a'),(2,False,'b')
Prelude> (,,,) <$> Just 1 <*> Just 2
               <*> Just 3 <*> Just 4
Just (1,2,3,4)
Prelude> (,,,) <$> Just 1 <*> Just 2
               <*> Just 3 <*> Nothing
Nothing
```