Karlsruhe Institute of Technology

Haskell für Mathematiker

Joachim Breitner

AG Seminar Topology

12. Mai 2016, Karlsruhe

LEHRSTUHL PROGRAMMIERPARADIGMEN

Übersicht

- Crash-Kurs Haskell-Syntax
- Rein funktionale Programmierung
- Typen
- Monaden und Kategorientheorie
- Lazyness
- Die Topologie von Haskell
- Beweise in Haskell?
- Wie gehts weiter?

Funktionen

Funktion definieren:

```
collatz 1 = 0
collatz n | even n = collatz (n `div` 2)
          | odd n  = collatz (3 * n + 1)
```

Funktionen

Funktion definieren:

```
collatz 1 = 0
collatz n | even n = collatz (n `div` 2)
          | odd n  = collatz (3 * n + 1)
```

Funktion verwenden:

```
*Main> map collatz [1..25]
[0,1,7,2,5,8,16,3,19,6,14,9,9,17,17,4,12,20,20,7,7,15,15,10,23]
```

Datentypen

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

leftMost (Leaf a) = a
leftMost (Node t1 _) = leftMost t1

mirror (Leaf a) = a
mirror (Node t1 t2) = Node (mirror t2) (mirror t1)

flatten :: (a -> a -> a) -> Tree a -> a
flatten (<>) = go
  where go (Leaf a) = a
        go (Node t1 t2) = go t1 <> go t2
```

Typsystem

- Primitive Basis-Typen:
  Int, Integer, Double
- Der Funktionstyp:
  (->)
- Algebraische Datentypen:
  Bool, [a], (a,b), Maybe a, Either a b
- Typsynonyme:
  type RatTree = Tree (Int,Int)
- Der eingebaute IO-Typ:
  IO a

Typsystem

- Primitive Basis-Typen:
  Int, Integer, Double
- Der Funktionstyp:
  (->)
- Algebraische Datentypen:
  Bool, [a], (a,b), Maybe a, Either a b
- Typsynonyme:
  type RatTree = Tree (Int,Int)
- Der eingebaute IO-Typ:
  IO a

Wenn es kompiliert, dann funktioniert es.

Typsystem

- Primitive Basis-Typen:
  Int, Integer, Double
- Der Funktionstyp:
  (->)
- Algebraische Datentypen:
  Bool, [a], (a,b), Maybe a, Either a b
- Typsynonyme:
  type RatTree = Tree (Int,Int)
- Der eingebaute IO-Typ:
  IO a

Wenn es kompiliert, dann funktioniert es.

Weitere Vorteile:

- Typ-geführtes Programmieren.
- Einfaches Refactoring.
- Typen sind Dokumentation.

Rein Funktionale Programmierung

- Funktionen sind wirklich Funktionen!
  - Gleiche Eingabe heißt gleiche Ausgabe
  - Kein globaler Zustand, keine Seiteneffekte
  - $\Rightarrow$ einfacher zu verstehen, zu verändern, richtig zu machen

Rein Funktionale Programmierung

- Funktionen sind wirklich Funktionen!
  - Gleiche Eingabe heißt gleiche Ausgabe
  - Kein globaler Zustand, keine Seiteneffekte
  - $\Rightarrow$ einfacher zu verstehen, zu verändern, richtig zu machen
- Funktionen sind Daten erster Klasse:
  - Funktionen dürfen an Funktionen übergeben werden (siehe flatten)
  - Funktionen können in Datenstrukturen gespeichert werden
  - $\Rightarrow$ sehr ausdrucksstark

Typklassen

Typklassen erlauben Algorithmen abstrakt zu fomulieren:

```
class Semigroup a where
    (&) :: a -> a -> a --- Law: (x & y) & z ≡ x & (y & z)
```

```
flattenS :: Semigroup a => Tree a -> a
flattenS (Leaf a) = a
flattenS (Node t1 t2) = flattenS t1 & flattenS t2
```

Typklassen

Typklassen erlauben Algorithmen abstrakt zu fomulieren:

```
class Semigroup a where
    (&) :: a -> a -> a --- Law: (x & y) & z ≡ x & (y & z)


flattenS :: Semigroup a => Tree a -> a
flattenS (Leaf a) = a
flattenS (Node t1 t2) = flattenS t1 & flattenS t2
```

Konkrete Typen können mit einer Instanz versehen werden:

```
instance Semigroup Integer where (&) = (+)
instance Semigroup [a] where (&) = (++)
instance Semigroup a => Semigroup (Maybe a) where
    Nothing & Nothing = Nothing
    Just x    & Nothing = Just x
    Nothing & Just x    = Just x
    Just x1  & Just x2  = Just (x1 & x2)
```

Monaden in Haskell

A monad is just a monoid in the category of endofunctors, what's the issue?

Monaden in Haskell

A monad is just a monoid in the category of endofunctors,
what's the issue?

Eine Monade in Haskell ist eine Typklasse:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

Monad-Laws:

```
return a >>= f    ≡ f a
m >>= return      ≡ m
(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
```

Einige Monaden

- Ein potentieller Wert:
  instance Monad Maybe
- Viele mögliche Werte:
  instance Monad []
- Ein Wert, der von einem Parameter abhängt:
  instance Monad ((->) b)
- Berechnungen, die etwas protokollieren:
  instance Monoid m => Monad (Writer m)
  newtype Writer m a = Write (m,a)
- Zustandsbehaftete Berechnungen:
  instance Monad (State m)
  newtype State s a = State (s -> (s,a))
- Ein String-Parser:
  instance Monad Parser
  newtype Parser x = Parser (String -> Maybe (String, x))
- Ein Wert, der mit der echten Welt interagiert: **instance** Monad IO

Kategorientheoretische Monaden

Mögliche alternative Definition:

```
class Monad m where
    map :: (a -> b) -> m a -> m b
    pure :: a -> m a
    join  :: m (m a) -> m a
```

Laws:

$$\begin{aligned}
\texttt{map g . pure} &\equiv \texttt{pure . g} \\
\texttt{map g . join} &\equiv \texttt{join . map (map g)} \\
\texttt{join . map join} &\equiv \texttt{join . join} \\
\texttt{join . return} &\equiv \texttt{join . map return} \equiv \texttt{id}
\end{aligned}$$

Kategorientheoretische Monaden

Mögliche alternative Definition:

```
class Monad m where
    map :: (a -> b) -> m a -> m b
    pure :: a -> m a
    join  :: m (m a) -> m a
```

Laws:

$$\texttt{map g . pure} \equiv \texttt{pure . g}$$
$$\texttt{map g . join} \equiv \texttt{join . map (map g)}$$
$$\texttt{join . map join} \equiv \texttt{join . join}$$
$$\texttt{join . return} \equiv \texttt{join . map return} \equiv \texttt{id}$$

Äquivalenz:

```
map f = (\a -> a >>= (return . f))
pure  = return
join a = a >>= id
```

```
return = pure
a >>= f = join (map f a)
```

Kategorientheoretische Monaden

Mögliche alternative Definition:

```
class Monad m where
  map :: (a -> b) -> m a -> m b
  pure :: a -> m a
  join  :: m (m a) -> m a
```

Laws:

$$\texttt{map g . pure} \equiv \texttt{pure . g}$$
$$\texttt{map g . join} \equiv \texttt{join . map (map g)}$$
$$\texttt{join . map join} \equiv \texttt{join . join}$$
$$\texttt{join . return} \equiv \texttt{join . map return} \equiv \texttt{id}$$

Äquivalenz:

```
map f = (\a -> a >>= (return . f))
pure  = return
join a = a >>= id
```

```
return = pure
a >>= f = join (map f a)
```

Kategorientheorie:

- m bildet Typen (Objekte) auf Typen ab
  map Funktionen (Morphismen) auf Funktionen
  $\Rightarrow$ wir haben einen Funktor **Hask** $\rightarrow$ **Hask**.
- pure entspricht ε, join dem μ.

do-Notation

Schöne Syntax für monadische Werte:
z.B. IO:

```
main = do
    input <- readLine
    let w = words input
    putStrLn $ "You entered "++ show (length w) ++ "words."

main = readLine >>= (\input ->
        let w = words input
        in putStrLn $ "You entered "++ show (length w) ++ "words.")
```

do-Notation

Schöne Syntax für monadische Werte:
z.B. Parser:

```
rangeParser :: Parser [Word16]
rangeParser = concat <$>
    oneRangeParser ‘sepBy1‘ many1 (char ’ ’ <|> char ’,’)

oneRangeParser :: Parser [Word16]
oneRangeParser = do
    n <- read ‘fmap‘ many1 digit <?> "Number"
    skipMany (char ’ ’)
    choice [  do char ’-’
                 n’ <- read ‘fmap‘ many1 digit <?> "Number"
                 unless (n’ > n) $ fail $ printf "%d is not larger than %d" n’ n
                 return [n..n’]
             ,  return [n]  ]
```

Lazyness

In Haskell herrscht *Bedarfsauswertung*:

```
f x y = if x < 0 then x * x else x * y
```

```
f (-2) (3 `div` 0) = if -2 < 0 then (-2) * (-2) else (-2) * (3 `div` 0)
                   = if True   then (-2) * (-2) else (-2) * (3 `div` 0)
                   = (-2) * (-2)
                   = 4
```

Ermöglicht

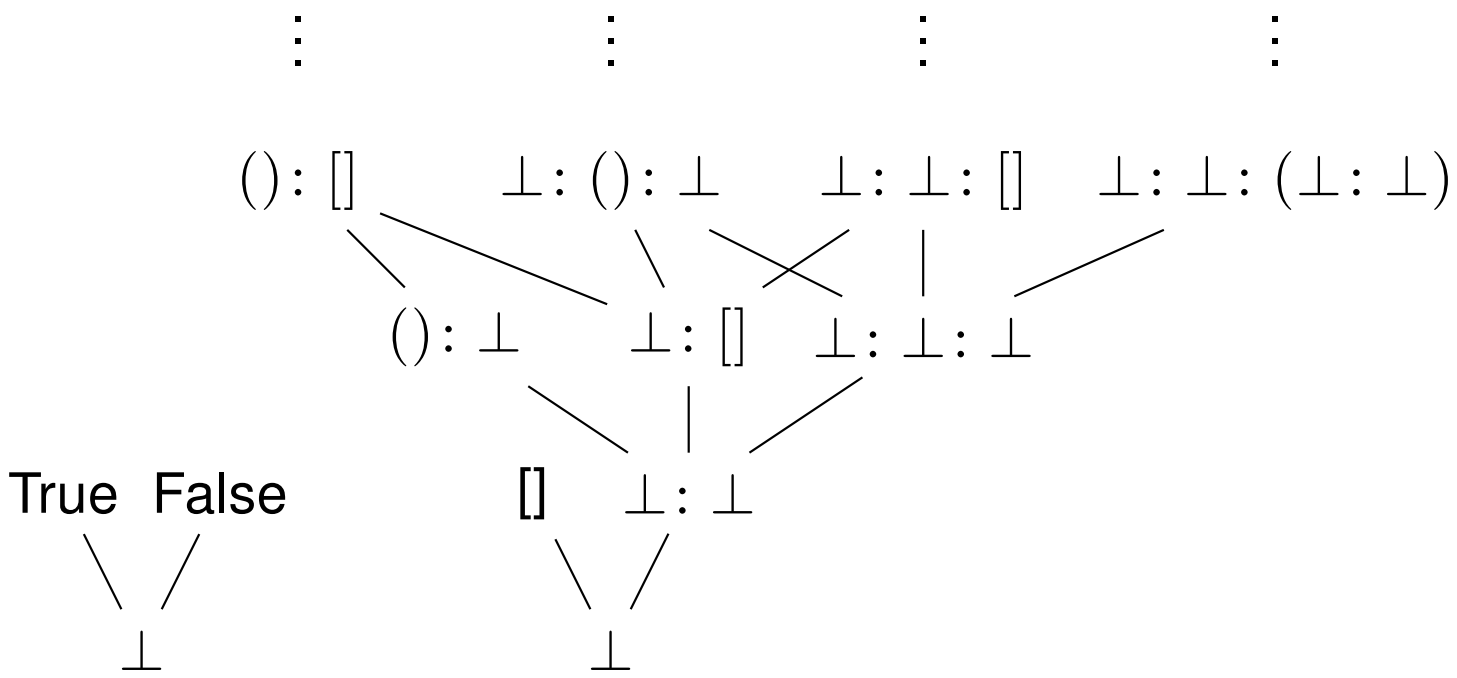
- mehr und einfachere Abstraktion
- unendliche Datenstrukturen

  ```
  enumerate :: [a] -> [(Integer, a)]
  enumerate = zip [1..]
  ```

- lustige und obskure Beweistricks
  (Watkinss Beweis von Conways Lost Theorem)

Die Topologie von Haskell

Alle Typen sind partiell geordnet (mit kleinstem Element und Limiten von gerichteten Ketten, pcpo), z.B. Bool und [()]:



Definierbare Funktionen sind *monoton* und *stetig*.
Dazu kann man die Scott-Topolgie nehmen: Abgeschlossen sind die nach unten abgeschlossenen Mengen, die ihre Suprema enthalten.

Beweisen mit Haskell

Mehrere Ansätze:

- Entscheidungsprozeduren in Haskell implementieren
  Dank Typsystem und Reinheit weniger Fehleranfällig,
  aber immernoch unverifiziert.
- Obskure Tricks mit Lazyness
- Howard-Curry-Isomorphismus

$$\text{Aussagen} \iff \text{Typen}$$
$$\text{Beweise} \iff \text{Programme (Terme)}$$

```
curry   :: ((a,b) -> c) -> (a -> b -> c)
curry   = \f x y -> f (x,y)
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry = \f p = case p of (x,y) -> f x y
```

Besser geeignet für ernsthafte Beweise: Coq, Agda, (Isabelle)

Bibliotheken

- Hackage (`http://hackage.haskell.org` mit fast 1000 Paketen
- Installierbar mit `cabal install` oder `stack install`
- Für viele praktischen Probleme gute Bibliotheken
  - Parser
  - Datenformate lesen und schreiben
  - Datenbanken
  - Grafik
  - GUI
  - Web-Zeug
  - uvm.
- Einiges zu Algebra (`algebra`, `constructive-algebra`), wenig zu Topologie
- Keine (ernstzunehmende) Anbindung zu Gap, Sage, Mathemathica

Fazit

Programmieren mit Haskell

- geht schneller
- produziert weniger Bugs
- ist für echte Anwendungen geeignet
- und macht mehr Spaß

Und wie geht es weiter?

- `http://haskell.org`
- Tutorials und Bücher, z.B. *Learn You a Haskell for Great Good!*
- `#haskell` im IRC
- `haskell-cafe@haskell.org` Mailing-Liste
- `[haskell]` auf `stackoverflow.com`