Workshop
Einführung in die Sprache Haskell

Nils Rexin und Marcellus Siegburg
nils.rexin@itemis.de msiegbur@imn.htwk-leipzig.de

itemis AG
Niederlassung Leipzig

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

20. Juni 2014

Übersicht

HTWK Leipzig

Allgemeines zu Haskell

- referentielle Transparenz (keine Nebenwirkung)
- Bedarfsauswertung (Lazy Evaluation)
- Funktionen höherer Ordnung
- Entwurfsmusterunterstützung
- kurzer Weg zwischen Spezifikation und Implementation
- ...

HTWK Leipzig

Datentyp

eigener Datentyp Boolean

```
data Boolean = Wahr | Falsch
  deriving Show
```

- **Boolean** ist der Typ
- **Wahr, Falsch** sind Konstruktoren
- | ist das Trennzeichen zwischen Konstruktoren
- **deriving Show** $\approx$ Vererbung .toString() in Java
- Einrückung beachten (Leerzeichen, Tabulatur)

HTWK Leipzig

Pattern Matching

eigener Datentyp Boolean

```
data Boolean = Wahr | Falsch
  deriving Show
```

UND - Funktion

```
und x y =
  case x of
  Falsch → Falsch
  Wahr → case y of
           Wahr → Wahr
           Falsch → Falsch
```

ODER - Funktion

```
oder x y = undefined
```

HTWK Leipzig

Pattern Matching

verschiedene Möglichkeiten

UND - Funktion

```
und x y =
  case x of
  Falsch → Falsch
  Wahr → case y of
           Wahr → Wahr
           Falsch → Falsch
```

. . . mit if-then-else

```
und x y =
  if x == Falsch
  then Falsch
  else if y == Wahr
       then Wahr
       else Falsch
```

. . . ohne spezielle Einrückung

```
und x y =
  case x of {Falsch → Falsch; Wahr → undefined}
```

HTWK Leipzig

Pattern Matching

verschiedene Möglichkeiten

UND - Funktion

```
und x y =
  case x of
  Falsch → Falsch
  Wahr → case y of
           Wahr → Wahr
           Falsch → Falsch
```

. . . mit if-then-else

```
und x y =
  if x == Falsch
  then Falsch
  else if y == Wahr
       then Wahr
       else Falsch
```

. . . ohne spezielle Einrückung

```
und x y =
  case x of {Falsch → Falsch; Wahr → undefined}
```

HTWK Leipzig

Pattern Matching

verschiedene Möglichkeiten

UND - Funktion

```
und x y =
  case x of
  Falsch → Falsch
  Wahr → case y of
           Wahr → Wahr
           Falsch → Falsch
```

... mit if-then-else

```
und x y =
  if x == Falsch
  then Falsch
  else if y == Wahr
       then Wahr
       else Falsch
```

... ohne spezielle Einrückung

```
und x y =
  case x of {Falsch → Falsch; Wahr → undefined}
```

HTWK Leipzig

Pattern Matching

mit un-/konkreten Konstruktoren

UND - Funktion

```
und Wahr Wahr = Wahr
und _    _    = Falsch
```

ODER - Funktion

```
oder Falsch Falsch = Falsch
oder _      _      = Wahr
```

HTWK Leipzig

Pattern Matching

mit konkreten Konstruktoren

Datentyp mit zwei ploymorphen Elementen; positionelle Notation

```
data Paar a = Paar a a

swap (Paar x y) = Paar y x
```

Datentyp mit zwei ploymorphen Elementen; benannte Notation

```
data Paar a = Paar{x :: a, y :: a}

swap p = Paar (y p) (x p)
```

HTWK Leipzig

Pattern Matching

mit konkreten Konstruktoren

Datentyp mit zwei ploymorphen Elementen; positionelle Notation

```
data Paar a = Paar a a

swap (Paar x y) = Paar y x
```

Datentyp mit zwei ploymorphen Elementen; benannte Notation

```
data Paar a = Paar{x :: a, y :: a}

swap p = Paar (y p) (x p)
```

HTWK Leipzig

Pattern Matching

mit konkreten Konstruktoren und case-of-Ausdruck

case-of-Ausdruck mit beliebigen Datentyp

```
data D = Foo {a :: Int, b :: Int }
       | Bar {c :: Int}
  deriving Show

f x = case x of
      Foo {a = x, b = y} → x + y
      Bar {} → c x
```

Allgemein ist zubehalten

Zu jedem Konstruktor eines Datentyps gibt es einen Zweig im case-of-Ausdruck.

HTWK Leipzig

Rekursiver Datentyp

Datentyp für polymorphe Listen definieren

```
data Liste a = Nil | Cons a (Liste a)
  deriving Show
```

Funktionen auf `Liste a`

```
key (Cons x xs) = x
next (Cons x xs) = xs
```

HTWK Leipzig

Rekursiver Datentyp

Datentyp Liste mit benannten Komponenten

```
data Liste a = Nil | Cons {key :: a, next :: Liste a}
  deriving Show
```

Anwendung

Aufgabe

- Datenstruktur die eine Linie, ein Rechteck und ein Kreis zusammenfasst
- Jedes Element soll eine Position enthalten (ganzzahlig) `(Paar Int)`
- Funktion, die den Flächeninhalt der oben genannten Objekte berechnet

Typen

- Typdefinitionen bei Funktionen explizit möglich

Typdefinition

```
nicht :: Boolean → Boolean
nicht a = case a of
  Wahr → Falsch
  Falsch → Wahr
```

- Haskell kann den Typ inferieren (ableiten)

HTWK Leipzig

Rekursive Funktionen

Aufgaben

- Können bei rekursiven Datenstrukturen verwendet werden

Auswertung einer Boolean-Liste

```
undListe :: Liste Boolean → Boolean
undListe liste = case liste of
  Nil → Wahr
  Cons a as → und a (undListe as)
```

Listenverknüpfung

```
verbinden :: Liste a → Liste a → Liste a
verbinden a b = case a of
  Nil → b
  Cons a' as → Cons a' (verbinden as b)
```

ITWK Leipzig

Rekursive Funktionen

- Funktion, die eine Liste umkehrt

Typ der Umpkehrfunktion

```
umkehren :: Liste a → Liste a
```

- Ein Paar von zwei Listen gleicher Länge, gleichen Typs in einer `Paar`-Liste zusammenführen
- Aus einer `Paar` Liste ein Paar mit zwei neuen Listen erzeugen

HTWK Leipzig

Map

Anonyme Funktionen

- Funktion wird nicht benannt
- Verwendung eines Lambda-Ausdrucks

Beispiel

```
addAnonym :: Boolean → Boolean → Boolean
addAnonym = (λ a b → case a of
  Wahr → b
  Falsch → Falsch)

addAnonym2 = (λ a → (λ b → case a of
  Wahr → b
  Falsch → Falsch ) )
```

Map

Funktionen höherer Ordnung

- Funktion als Rückgabewert
- Funktion als Parameter

Beispiel

```
operationPaar :: (a → a → b) → Paar a → b
operationPaar f (Paar a b) = f a b

undPaar :: Paar Boolean → Boolean
undPaar p = operationPaar (λ a b → und a b) p
```

Map

Definition

- Idee: Auf alle Elemente einer rekursiven Struktur die selbe Funktion anwenden
- Definition von Map:

Map für Listen

```
mapListe :: (a → b) → Liste a → Liste b
mapListe f l = case l of
  Nil → Nil
  Cons a as → Cons (f a) (mapListe f as)
```

Map

Aufgaben

- `und` auf alle `Boolean`-Paare in einer Liste anwenden
- Aus einer Liste mit geometrischen Objekten alle Quadrate durch `Wahr` in einer neu erzeugten `Boolean`-Liste hervorheben

HTWK Leipzig

Motivation

Rekursive Funktionen auf Listen

```
summe liste = case liste of
  Nil         → 0
  Cons x xs   → x + summe xs

laenge liste = case liste of
  Nil         → 0
  Cons _ xs   → 1 + laenge xs

undListe liste = case liste of
  Nil         → Wahr
  Cons x xs   → und x (undListe xs)
```

→ gemeinsames Muster?

Fold auf Listen (1)

Muster (Pseudocode)

```
f liste = case liste of
  Nil        → ?exp1?
  Cons x xs  → ?exp2? x (f xs)
```

Abstraktion des gemeinsamen Musters

```
foldListe :: b → (a → b → b) → Liste a → b
foldListe exp1 exp2 liste = case liste of
  Nil        → exp1
  Cons x xs  → exp2 x (foldListe exp1 exp2 xs)
```

HTWK Leipzig

Fold auf Listen (2)

Rekursive Funktionen auf Listen

```
summe liste = case liste of
  Nil       → 0
  Cons x xs → x + summe xs

laenge liste = case liste of
  Nil       → 0
  Cons _ xs → 1 + laenge xs

undListe liste = case liste of
  Nil       → Wahr
  Cons x xs → und x (undListe xs)
```

Definition mit Fold

```
summe' liste = foldListe 0 (λx result → x + result) liste

laenge' liste = foldListe 0 (λx result → 1 + result) liste

undListe' liste = foldListe Wahr (λx result → und x result) liste
```

Fold auf Listen (3)

Übung: Ersetze `undefined`

```
verbinden :: Liste a → Liste a → Liste a
verbinden a b = foldListe undefined undefined a

umkehren :: Liste a → Liste a → Liste a
umkehren liste = foldListe undefined undefined liste
```

Fold im Allgemeinen

- für jeden Konstruktor K eines Types bekommt `fold` ein Parameter p
- p hat die gleiche Stelligkeit wie K
- Parametertypen von K entsprechen den Parametertypen von p (fast, s. Rekursion)

Vergleiche `Liste a` mit `foldListe`

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

foldListe :: b → (a → b → b) → Liste a → b
```

Fold auf Bäumen

Übung: Ersetze ? und undefined

```
data Baum a = Blatt | Zweig (Baum a) a (Baum a)

foldBaum :: ? → ? → Baum a → b
foldBaum blatt zweig baum = undefined

summeSchluessel :: Baum Integer → Integer
summeSchluessel baum = foldBaum undefined undefined baum

hoehe :: Baum a → Integer
hoehe baum = foldBaum undefined undefined baum

inorderSchluessel :: Baum a → Liste a
inorderSchluessel baum = foldBaum undefined undefined baum
```

HTWK Leipzig

Zusammenfassung

Aufgabe

- Datentypen mit Pattern Matching (case-of)
- rekursive Datentypen (Liste, Baum)
- Polymorphie
- Typ-Definition
- Funktionen höherer Ordnung (mapListe)
- Lambda-Ausdruck
- Besuchermuster (fold) auf Listen, Bäumen

Quelltext

Link: `http://www.imn.htwk-leipzig.de/~nrexin/hal9`

HTWK Leipzig

Vielen Dank für Ihre Aufmerksamkeit

HTWK Leipzig