Einführung in Haskell

Bernhard Tittelbach

RealRaum — http://www.realraum.at — reality://innere_stadt.graz.at/sporgasse/16

19. Juli 2008

Haskell

- funktionale Sprache
  - basiert auf Lambda Kalkül und Rekursion
  - statt Statements zum Ablauf
  - Verkettung von Funktionen
  - derzeit der Standard unter funktionalen Sprachen
- Glasgow Haskell Compiler: `ghc`
  - wichtigstes Commando: `ghc --make <hauptdatei.hs>`
  - Interactive Shell: `ghci`
  - Aktuelle Version: 6.8.3
- Standard Library namens Prelude
- `http://www.haskell.org`

Lazy Evaluierung

- Ausdrücke werden erst ausgewertet, wenn sie wirklich benötigt werden
- Im Besten Fall: Berechnunszweige die nie gebraucht werden, werden auch nie berechnet
- Im Schlechtesten Fall: jede Berechnung hat overhead
- Syntax erlaubt erzwungene strikte Evaluierung: `$!`

wichtige Begriffe und Eigenschaften

Pattern Matching Ausführung der Funktionsdefinition mit passendem Muster

List Comprehension automatische Listen generierung aus Statements

Guards Vor-Bedingung für die Ausführung eines Statements

Monaden Statements mit Seiteneffekten (z.b. IO)

Single Assignment

- keine Variablen, nur Funktionen
- Funktionen können nicht plötzlich umdefiniert werden
- simpelste aller Funktionen liefert immer den selben Wert (=Variable)

Curry-ing Statt: Eine Funktion nimmt Argumente und liefert Ergebnis:
Eine Funktion nimmt 1 Argument und gibt eine Funktion zurück
$f : (X \times X) \to Z \qquad curry(f) : X \to (Y \to Z)$

wichtige Begriffe und Eigenschaften

Pattern Matching Ausführung der Funktionsdefinition mit passendem Muster

List Comprehension automatische Listen generierung aus Statements

Guards Vor-Bedingung für die Ausführung eines Statements

Monaden Statements mit Seiteneffekten (z.b. IO)

Single Assignment

- keine Variablen, nur Funktionen
- Funktionen können nicht plötzlich umdefiniert werden
- simpelste aller Funktionen liefert immer den selben Wert (=Variable)

Curry-ing Statt: Eine Funktion nimmt Argumente und liefert Ergebnis:
Eine Funktion nimmt 1 Argument und gibt eine Funktion zurück
$f : (X \times X) \to Z \qquad curry(f) : X \to (Y \to Z)$

wichtige Begriffe und Eigenschaften

Pattern Matching Ausführung der Funktionsdefinition mit passendem Muster

List Comprehension automatische Listen generierung aus Statements

Guards Vor-Bedingung für die Ausführung eines Statements

Monaden Statements mit Seiteneffekten (z.b. IO)

Single Assignment

- keine Variablen, nur Funktionen
- Funktionen können nicht plötzlich umdefiniert werden
- simpelste aller Funktionen liefert immer den selben Wert (=Variable)

Curry-ing Statt: Eine Funktion nimmt Argumente und liefert Ergebnis:
Eine Funktion nimmt 1 Argument und gibt eine Funktion zurück
$f : (X \times X) \to Z \qquad curry(f) : X \to (Y \to Z)$

Pattern Matching und Guards

Example (Pattern Matching)

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * f (n-1)
```

Example (Guards)

```
factorial :: Int -> Int
factorial n
  | n > 0     = n * f (n-1)
  | otherwise = 1
```

- Nur Beispielhaft, Nicht ganz dasselbe ! (neg. Werte)

List Comprehensions

Example (Unendliche Liste)

```
inf_pos = [0..]
s = [ 2*x | x <- inf_pos, x^2 > 3 ]
```

Example (pythagoräisch Dreiecke)

```
pyth = [(x,y,z) | x <- [1..20], y <- [x..20], z <- [y..20],
        x^2 + y^2 == z^2]
```

Example (Fibonacci Zahlen)

```
fibs = 0 : 1 : [ a+b | a <- fibs | b <- tail fibs ]
```

Datentypen

Basistypen Bool,Int,Integer,Float,Double,Char, $\perp$

Listen

- Wichtigster Datentyp
- Listen haben einen eindeutigen Content-Typ

Tuple Sequenz von Werten (z.b.: Paar,Triple,etc)

Strings Listen von Chars

Rational Typ für eine Bruchzahl (bestehend aus 2 Integer)

Maybe vielleicht ein Typ

Typen-System

stark es gibt keine implizite Konvertierung
`2 + "10"` wirft Compiler-Fehler

statisch Typen Checks werden zur compile-time durchgeführt

implizit Wird der Typ nicht angegeben, so verwendet Haskell die minimal passende Typen-Klasse
Typen lassen sich direkt durch Funktions-Signaturen angeben

Typen-Klassen I

- es gibt Klassen von Typen
- jeder Typ (auch selbstdef.) kann zu mehreren Klassen gehören
- Eine TypenKlassen bestimmt welche Operationen (auch selbstdef.) möglich sind.
- TypenKlassen sind teilw. von BasisTypenKlassen abgeleitet.

Typen-Klassen II

wichtigste Klassen:

Num	+,−,∗,`fromInteger`, usw
Integral	Integer Division `div`,`mod`,`divMod`,`toInteger`
Fractional	Ergebnis einer echten Division /, `recip`, `fromRational`
Real	`toRational`
Eq	vergleichbare Typen. ==, /=
Ord	sortierbare Typen: <, <=, >, >=, `max`, `min`
Monad	Abstrakter Datentyp `IO` <`typ`> kennzeichnet Seiteneffekt >>=, >>, `return`, `fail`, (nicht in Klasse: `do`, <−)