Haskell

A Wild Ride

Sven M. Hallberg

sm@khjk.org

Überblick

- Vorsichtsmaßnahmen
  Einführung und Sprachgrundlagen
- Grund-Attraktionen
  cooles Zeug
- Wipeout!
  das richtig coole Zeug

Vorsichtsmaßnahmen

Vorsichtsmaßnahmen

- Dies ist *kein* Programmierkurs.
- Ich vernachlässige Details, um zügig zu den interessanten Punkten zu kommen.
- Fragen werden gern gehört!
- Dieser Vortrag ist gedacht als rauschende *aber vergnügliche* Fahrt durch das Thema.

Einordnung der Sprache

Haskell ist:

- rein
- funktional
- statisch getypt
- stark getypt
- verzögert ausgewertet (“faul”)

Compiler oder Interpreter?

Es gibt sowohl als auch, compiliert sowie interpretiert.

- GHC: Compiler, Maschinencode
- Hugs: Interpreter
- NHC: Compiler, Bytecode
- HBI/HBC: Interpreter/Compiler
- Helium: Interpreter, für eine Teilsprache

Programmaufbau

- Ein Programm ist eine Folge von Deklarationen.
- Deklarationen sind absolut.
- “Variablen” sind zur Laufzeit unveränderlich!

```
hello    = "Guten Tag!"
goodbye  = "Auf Wiedersehen!"
```

Functionsdefinitionen

```
foo x   = x + 2
bar x y = foo y + x
```

- Functionsargumente nur durch Leerzeichen getrennt.
- Operatoren in Infix-Notation.
- Keine Typsignaturen?
- Typen werden hergeleitet!

Prozeduren

- Alle Functionen sind “rein”.
- Was ist mit Netz- oder Datei-I/O?
- Lösung: Strenge Trennung durch das *Typsystem*!

```
main = do putStrLn hello
          input <- getLine
          putStrLn (show input ++ "?")
          putStrLn goodbye
```

Typen

- Das Typsystem spielt eine der wichtigsten Rollen in Haskell.
- Jedem Ausdruck wird statisch sein Typ zugeordnet.
- Typsignaturen sind *optional*.
  - Dokumentation für den Programmierer
  - Bessere Fehlermeldungen vom Compiler

```
hello :: String
```

Listen

- Wahrscheinlich die wichtigste Datenstruktur.
- Einfach verkettet, wie in LISP (“nil/cons-Struktur”).

```
intlist   :: [Int]
intlist   =  [1,2,3]

charlist  :: [Char]
charlist  =  ['H','i']

emptylist :: [a]
emptylist =  []
```

Functionstypen

- Funktionen sind Werte erster Ordnung.
- Ein Argument:

```
foo :: Int -> Int
```

- Zwei Argumente:

```
bar :: Int -> Int -> Int
```

- Letzter Typ ist der Rückgabewert.

Listen-Konstruktoren

- Werte werden erzeugt durch Konstruktoren.
- Konstruktoren sind Funktionen.
- `[]` und `(:)` sind die Konstruktoren für `[a]`.

```
fourints = 1 : (2 : (3 : (4 : [])))
```

- Die Syntax für Listenliterale ist lediglich syntaktischer Zucker für wiederholte Anwendung von `(:)`!

```
[1,2,3] == 1:2:3:[]
```

Grund-Attraktionen

Lokale Definitionen

- Eingeleitet durch `where`-Klausel nach einer Deklaration.

```
readability :: String -> Float
readability text =
        if n==0 then 1
                else 1 / fromIntegral n
    where
    n = length text
```

- Es gibt auch `let` zur Verwendung innerhalb von Ausdrücken.

```
foo 5 == (let x=5 in x)
```

Konstruktor-Pattern Matching

- Werte werden durch Konstruktoranwendung erzeugt.
- Konstanten = nulläre Konstruktoren, z.B. `[]`, `1`, `2`,...
- Der Konstruktor und seine Argumente *sind* der Wert.
- Inspektion durch pattern matching auf den/die Konstruktor(en).

```
null []     = True
null (x:xs) = False   -- ctor arguments bound
```

Selbstdefinierte Datentypen

Beispiel: Steuerung eines Magnetkartenlesers

```
data Cmd =  Read   Track
         |  Write  Track

data Track = Track1 | Track2 | Track3
```

- Typen und Konstruktoren werden großgeschrieben.
- Merke: Konstruktoren können beliebig viele Argumente beliebiger (festgelegter!) Typen nehmen.

Selbstdefinierte Datentypen

Nehme folgendes Steuerprotokoll für das Lesegerät an:

- Kommandos bestehen aus drei Bytes, zu senden über Serielle.
- Erstes Byte: `'a'` = “read”, `'b'` = “write”
- Zweites Byte: immer `'a'`
- Drittes Byte: `'a'`, `'b'`, `'c'` für Track 1,2,3 resp.

Z.B.: `"aaa"` für “read track 1”.

Selbstdefinierte Datentypen

$\Rightarrow$ Triviale Haskell-Function um `Cmds` auf Steuersequenzen abzubilden:

```
ctlstr :: Cmd -> String
ctlstr (Read   Track1) = "aaa"
ctlstr (Read   Track2) = "aab"
ctlstr (Read   Track3) = "aac"
ctlstr (Write  Track1) = "baa"
ctlstr (Write  Track2) = "bab"
ctlstr (Write  Track2) = "bac"
```

Selbstdefinierte Datentypes

- Gegeben: I/O-Routine `sendstr` zum Senden einer Steuersequenz zum Gerät:

```
sendcmd cmd = sendstr (ctlstr cmd)
```

⇒ Interaktive Gerätesteuerung vom Interpreter:

```
Main> sendcmd (Read Track1)
...stuff happens...
```

- Merke: `Cmds` können herumgereicht und in Datenstrukturen abgelegt werden.

List Comprehensions

- Listen und Listenoperationen kommen alltäglich vor.
- List comprehensions sind Syntaxzucker zum gleichzeitigen Ausdrücken der
  - Sammlung und Auswahl von (Eingabe-)Elementen, sowie der
  - Erzeugung der entsprechenden Ergebniselemente.
- Angelehnt an mathematische Mengenschreibweise:

$$\{x^2 | x \in \mathbb{N}, x > 5\}$$

List Comprehensions

Beispiel: Haskell-Implementation von Quicksort:

```
qs []     = []
qs (x:xs) = qs [y | y<-xs, y<x]
            ++ [x] ++
            qs [y | y<-xs, y>=x]
```

Wipeout!

The Hackers Must Have Slack.

- Verzögerte Auswertung erlaubt die Konstruktion unendlicher Datenstrukturen.
- Unendliche Listen insbesondere
- Durch rekursive Definitionen

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Wer Benutzt Denn RC4?

Beispiel: Memoisierung einer unendlichen Datenstruktur.

“Arcfour” funktioniert ungefähr so:

- Aus dem Schlüssel, erzeuge eine S-Box.
- Iteriere:
  - Entnimm ein bestimmtes Element der S-Box $\rightarrow$ in den Keystream.
  - Transformiere die S-Box auf eine bestimmte Weise.
- XOR des resultierenden Keystreams mit dem Plaintext.

Wer Benutzt Denn RC4?

Man stelle sich vor, das in C zu implementieren.

- Datenstruktur für die S-Box
- Routine zur Initialisierung aus dem Schlüssel
- Erzeugung des Keystreams in Blöcken.

In Haskell:

- keine Blöcke
- keine Initialisierungsroutine!

Wer Benutzt Denn RC4?

Pseudocode:

```
type Key  = String
data SBox = ...

mksbox    :: Key -> SBox
keystream :: SBox -> [Word8]

rc4 :: Key -> [Word8] -> [Word8]
rc4 k xs = zipWith xor xs
           (keystream (mksbox k))
```

- Wo liegt der Witz?

Partialle Applikation

Angenommen, man will einen haufen Dateien mit dem selben Schlüssel verschlüsseln.

Übermäßig verbose:

```
key = "deadbeef"

file1_encrypted = rc4 key file1
file2_encrypted = rc4 key file2
```

Partialle Applikation

Angenommen, man will einen haufen Dateien mit dem selben Schlüssel verschlüsseln.

Schlank:

```
enc = rc4 "deadbeef"

file1_encrypted = enc file1
file2_encrypted = enc file2
```

Partialle Applikation

Angenommen, man will einen haufen Dateien mit dem selben Schlüssel verschlüsseln.

Schlank:

```
enc = rc4 "deadbeef"

file1_encrypted = enc file1
file2_encrypted = enc file2
```

- Ein Problem: Keystream wird bei jedem Aufruf von `enc` neuberechnet.

Memoisierung

- Der Keystream hängt nur vom Schlüssel ab.
- Das Haskell-System ist nicht schlau genug, das zu erkennen.
- Explizitmachen durch “Reinziehen” einer Closure.

```
rc4 k = \xs -> zipWith xor xs ks
    where
    ks = keystream (mksbox k)
```

- Danach referenzieren alle Aufrufe von `rc4 key` das selbe `ks`.

Typklassen

- Haskell unterstützt Compile-Zeit-Polymorphie.

```
null :: [a] -> Bool
```

- Oft ist das zu allgemein.
- `(+)` ist polymorph, aber nur eingeschränkt auf bestimmte Typen.

```
(+) :: (Num a) => a -> a -> a
```

- Sprich: “`a -> a -> a` unter der Bedingung, daß a eine Zahl ist”.

Typklassen

Die Definition einer Typklasse sieht so aus:

```
class Num a where
    (+)      :: a -> a -> a
    (*)      :: a -> a -> a
    negate   :: a -> a
    ...
```

- Typklassen schreiben eine Schnittstelle vor.
- Immer noch *Compile-Zeit*-polymorph!

Krassere Typen

- `Num` beschreibt im Prinzip die Klasse der Ringe.
- `Fractional` beschreibt im Grunde Körper.

```
class (Num a) => Fractional a where
    (/)    :: a -> a -> a
    recip  :: a -> a
```

- Ziel: Deklaration der Klasse der *Vektorräume*.
- Problem: Vektorräume $\leftrightarrow$ Skalarenkörper

```
    smul   :: a -> v -> v
```

“Multi-parameter type classes”

- Lösung: Erweitere Typklassen (i.e. Mengen von Typen) zu Relationen zwischen Typen!

```
class (Fractional a) => VS v a
    where
    -- vector add and subtract
    (^+^)  :: v -> v -> v
    (^-^)  :: v -> v -> v
    -- scalar multiplication
    (*^)   :: a -> v -> v
```

“Multi-parameter type classes”

- Lösung: Erweitere Typklassen (i.e. Mengen von Typen) zu Relationen zwischen Typen!
- Außerdem: Funktionale Abhängigkeiten in Typrelationen.

```
class (Fractional a) => VS v a |v->a
    where
    -- vector add and subtract
    (^+^)  :: v -> v -> v
    (^-^)  :: v -> v -> v
    -- scalar multiplication
    (*^)   :: a -> v -> v
```

Vektorraum-Beispiel

Um den Typ “Paare von `Float`” zu einem `Float`-Vektorraum zu deklarieren:

```
instance VS (Float,Float) Float where
    (x,y) ^+^ (a,b) = (x+a, y+b)
    (x,y) ^-^ (a,b) = (x-a, y-b)
    k     *^  (a,b) = (k*a, k*b)
```

Bemerke:

- Multi-parameter type classes und “fundeps” sind nicht Haskell 98.
- Beide Erweiterungen werden von allen großen Implementationen unterstützt.

Zusammenfassung

- Haskell ist ein weitläufiges Thema.
- Erweiterungen werden aktiv erforscht.
- Trotzdem ist die Sprache sehr klar.
- Vieles läßt sich sehr natürlich ausdrücken.
  - Programme werden kurz und knapp.
  - Rapid prototyping
- Sicherer und robuster Code

Weiterführende Links

- Alles über Haskell:

  `http://www.haskell.org/`

- Diese Folien und die ursprüngliche Ausarbeitung (Englisch):

  `http://www.khjk.org/~sm/`