

Haskell Live

[01] Eine Einführung in Hugs

Bong Min Kim

bmktuwien@gmail.com

Christoph Spörk

christoph.spoerk@gmail.com

Florian Hassanen

florian.hassanen@gmail.com

Bernhard Urban

lewurm@gmail.com

14. Oktober 2011

Hinweise

Diese Datei kann als sogenanntes “Literate Haskell Skript” von `hugs` geladen werden, als auch per `lhs2TeX`¹ und \LaTeX in ein Dokument umgewandelt werden.

Kurzeinführung in `hugs`

`hugs`² ist ein Interpreter für die funktionale Programmiersprache Haskell. Abhängig vom Betriebssystem wird der Interpreter entsprechend gestartet, unter GNU/Linux beispielsweise mit dem Befehl `hugs`. Tabelle 1 zeigt eine Übersicht der wichtigsten Befehle in `hugs`.

¹<http://people.cs.uu.nl/andres/lhs2tex>

²Haskell User’s Gofer System

Befehl	Kurzbefehl	Beschreibung
<code>:edit name.hs</code>	<code>:e name.hs</code>	öffnet die Datei <code>name.hs</code> in dem Editor, der in <code>\$EDITOR</code> (Unix) bzw. in WinHugs in den Optionen definiert ist
<code>:load name.hs</code>	<code>:l name.hs</code>	lädt das Skript <code>name.hs</code> . Man kann fortan die einzelnen Funktionen aus dem Script im <code>hugs</code> auszuführen.
<code>:edit</code>	<code>:e</code>	öffnet den Editor mit der zuletzt geöffneten Datei
<code>:reload</code>	<code>:r</code>	erneuert Laden des zuletzt geladenen Skripts
<code>:type Expr</code>	<code>:t Expr</code>	Typ von <code>Expr</code> anzeigen
<code>:info Name</code>		Informationen zu <code>Name</code> anzeigen. <code>Name</code> kann z.B. ein Datentyp, Klasse oder Typ sein
<code>:cd dir</code>		Verzeichnis wechseln
<code>:quit</code>	<code>:q</code>	<code>hugs</code> beenden
<code>Expr</code>		Wertet <code>Expr</code> aus (wobei für diese Auswertung das momentan geladene Skript herangezogen wird; siehe <code>:load</code>)

Table 1: Einige Befehle in `hugs`

Primitiver Haskell Code

```
eins :: Integer
eins = 1
```

```
addiere :: Integer → Integer → Integer
addiere x y = x + y
```

```
addiereFuenf :: Integer → Integer
addiereFuenf x = addiere 5 x
```

```
istGleichEins :: Integer → Bool
istGleichEins 1 = True -- Reihenfolge beachten. Spezielle Patterns zuerst!
istGleichEins x = False
```

Listen notieren

Listen können einfach notiert werden: Zum Beispiel erzeugt der Ausdruck `[1,2,3,4]` eine Liste von gleicher Darstellung. In Tabelle 2 sind einfache Beispiele angeführt.

Ausdruck	Ergebnis	Beschreibung
<code>[1,2,3,4,5]</code>	<code>[1,2,3,4,5]</code>	Erzeugt eine Liste mit den Elemente 1 bis 5
<code>[1..5]</code>	<code>[1,2,3,4,5]</code>	Erzeugt eine Liste mit den Elemente 1 bis 5
<code>[1,4..14]</code>	<code>[1,4,7,10,13]</code>	Siehe nächstes Beispiel
<code>[a,b..x]</code>	<code>[a, b = a + d, a + 2d, a + 3d, . . . , x - d < a + nd ≤ x]</code>	Es wird ein Offset d (Differenz von a und b) ermittelt. Zu der Basis a , wird bis zum Wert x , jede Summe der Basis plus einem Vielfachen des Offsets, der Liste hinzugefügt
<code>[]</code>	<code>[]</code>	Leere Liste aka. "nil"
<code>1:(2:(3:(4:[])))</code>	<code>[1,2,3,4]</code>	<code>(:)</code> aka. "cons"
<code>1:2:3:4:[]</code>	<code>[1,2,3,4]</code>	"cons" ist rechts-assoziativ
<code>"asdf"</code>	<code>"asdf"</code>	Liste von <code>Char</code> .
<code>'a':'s':'d':'f':[]</code>	<code>"asdf"</code>	Beachte, dass der Typ <code>String</code> dem Typen <code>[Char]</code> entspricht.

Table 2: Einfache Beispiele für Listen

Listen verarbeiten

```

my_head :: [Integer] → Integer
my_head [] = -1
my_head (x : xs) = x -- Reihenfolge beachten! (Pattern Matching)
my_head (x : []) = x + 1

```

```

laf1 :: [Integer] → [Integer] -- list.addiere.fuenf
laf1 [] = []
laf1 (x : xs) = (addiereFuenf x) : (laf1 xs)

```

```

laf2 :: [Integer] → [Integer]
laf2 l = [addiereFuenf x | x ← l, x > 10] -- list comprehension

```

```

laf3 :: [Integer] → [Integer]
laf3 l = map (addiereFuenf) l -- map magic

```

Integer VS. Int

```

grosserInteger :: Integer
grosserInteger = 1000 * 1000 * 1000 * 1000 * 1000 * 1000

```

```
grosserInt :: Int
grosserInt = 1000 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000
```

```
gibNimmInteger :: Integer → Integer
gibNimmInteger i = i
```

```
gibNimmInt :: Int → Int
gibNimmInt i = i
```

Besonders nervig, wenn man einen `Int` hat aber einen `Integer` bräuchte (oder vice versa). In so einem Fall könnte man die betroffene Funktion nochmals manuell implementieren:

```
myLength :: [Integer] → Integer
myLength [] = 0
myLength (x : xs) = 1 + (myLength xs)
```

Oder man verwendet `fromIntegral`:

```
passendIntInteger :: Int → Integer
-- fromIntegral in Prelude
passendIntInteger i = gibNimmInteger (fromIntegral (gibNimmInt i))
```

```
passendIntegerInt :: Integer → Int
-- funktioniert in beide Richtungen
-- mit $ kann man Klammern sparen!
passendIntegerInt i = gibNimmInt $ fromIntegral $ gibNimmInteger i
```

Rudimentäres Debugging

Use `trace` from `Debug.Trace`:

```
import Debug.Trace

lafDebug :: [Integer] → [Integer]
lafDebug [] = trace "Liste zu Ende" []
lafDebug (x : xs) = trace debugMessage (neuesX : (lafDebug xs))
  where
    neuesX = addiereFuenf x
    debugMessage = "Berechne: addiereFuenf " ++ (show x) ++ " = " ++ (show neuesX)
```

Dokumentation

- Prelude: <http://www.google.at/search?q=haskell+prelude+documentation>

- Interaktive Einführung in Haskell: <http://tryhaskell.org>