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`
[2] Haskell User's Gofer System

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

Table 1: Einige Befehle in `hugs`

Primitiver Haskell Code

$eins :: Integer$
$eins = 1$

$addiere :: Integer \to Integer \to Integer$
$addiere\ x\ y = x + y$

$addiereFuenf :: Integer \to Integer$
$addiereFuenf\ x = addiere\ 5\ x$

$istGleichEins :: Integer \to Bool$
$istGleichEins\ 1 = True$ -- 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
`[1,2,3,4,5]`	`[1,2,3,4,5]`	Erzeugt eine Liste mit den Elemente 1 bis 5
`[1..5]`	`[1,2,3,4,5]`	Erzeugt eine Liste mit den Elemente 1 bis 5
`[1,4..14]`	`[1,4,7,10,13]`	Siehe nächstes Beispiel
`[`a`,`b`..`x`]`	`[` a, $b = a + d$, $a + 2d$, $a + 3d$, $\ldots$, $x - d < a + nd \leqslant x$ `]`	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
`[]`	`[]`	Leere Liste aka. "nil"
`1:(2:(3:(4:[])))`	`[1,2,3,4]`	`(:)` aka. "cons"
`1:2:3:4:[]`	`[1,2,3,4]`	"cons" ist rechts-assoziativ
`"asdf"`	`"asdf"`	Liste von `Char`.
`'a':'s':'d':'f':[]`	`"asdf"`	Beachte, dass der Typ `String` dem Typen `[Char]` entspricht.

Table 2: Einfache Beispiele für Listen

Listen verarbeiten

$my\_head :: [Integer] \to Integer$
$my\_head\ [\,] = -1$
$my\_head\ (x : xs) = x$ -- Reihenfolge beachten! (Pattern Matching)
$my\_head\ (x : [\,]) = x + 1$

$laf1 :: [Integer] \to [Integer]$ -- **l**ist_**a**ddiere_**f**uenf
$laf1\ [\,] = [\,]$
$laf1\ (x : xs) = (addiereFuenf\ x) : (laf1\ xs)$

$laf2 :: [Integer] \to [Integer]$
$laf2\ l = [addiereFuenf\ x \mid x \leftarrow l, x > 10]$ -- list comprehension

$laf3 :: [Integer] \to [Integer]$
$laf3\ l = map\ (addiereFuenf)\ l$ -- map magic

Integer VS. Int

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

```
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`