Compiler: Blatt 1

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 27.04.2017 09:44

Dieses Blatt ist ein Tutorial an dem Sie sich selbst entlang hangeln sollen. Wenden Sie sich bei Problemen/Fragen bitte an mich.

Erste Schritte mit Haskell

Option 1: Programmieren auf den Laborrechnern Auf den Laborrechnern ist die Haskell-Platform bereits unter Windows installiert. Sie können z.B. mit Notepad++ programmieren. Zum Übersetzen und Ausführen müssen Sie ein `cmd`-Fenster öffnen.

Auf manchen Rechnern fehlt noch die `msys2`-Library. Installieren Sie diese in einem `cmd`-Fenster mit dem Kommando:

```
stack setup
```

Stack ist in der Lage einen aktuellen GHC und alle benötigten Bibliotheken für Sie herunter zu laden. Nachdem Sie nicht soviel Quota auf den Laborrechnern haben, ist ein aktueller GHC bereits installiert. Um diesen zu nutzen und keinen anderen herunter laden zu lassen, geben Sie folgendes ein:

```
stack config set system-ghc -global true
```

Anschließend sind Sie mit Haskell arbeitsfähig.

Option 2: Installieren Sie Haskell auf Ihrem privaten Rechner Installieren Sie Haskell wie unter https://www.haskell.org/downloads beschrieben

Und los geht es mit einem Template

Stack bringt eine Reihe von Projekt-Templates mit, damit Sie nicht von Null beginnen müssen.

Mit

```
stack templates
```

können Sie die verfügbaren Templates anzeigen lassen.

Initialisieren Sie mit dem Kommando

```
stack new FirstHaskellProjekt new-template
```

ein neues Projekt. Das Projekt wird ausgehend von Ihrem aktuellen Verzeichnis im `cmd`-Fenster im Unterverzeichnis `FirstHaskellProjekt` erzeugt. Sie können natürlich auch einen beliebigen anderen Namen verwenden. Das `new-template` erzeugt ein simples neues Projekt mit einer typischen Gliederung.

Nach dem Wechsel in das neue Verzeichnis, können Sie das Projekt mit dem folgenden Kommando bauen lassen:

```
stack build
```

In `.stack-work` werden die erzeugten Artefakte abgelegt.

Um die, nicht wirklich vorhandene, Haddock-Doku zu bauen, geben Sie folgendes ein:

```
stack haddock --no-haddock-deps
```

Als eine der letzten Ausgaben sehen Sie die Location der Datei `index.html` unterhalb von `.stack-work`.

Zum automatischen Öffnen im Browser können Sie `--open` anhängen, also

```
stack haddock --no-haddock-deps --open
```

Um das erzeugte Binary `FirstHaskellProjekt-exe` auszuführen, geben Sie folgendes ein

```
stack exec FirstHaskellProjekt-exe
```

Sie sehen dann die, nicht sehr spannende Ausgabe

```
someFunc
```

Das Projekt ist außerdem schon so konfiguriert, dass (noch nicht vorhandene) Tests ausgeführt werden können. Geben Sie dazu folgendes ein:

```
stack test
```

Als Ausgabe werden Sie folgendes sehen:

```
Test suite not yet implemented.
```

Ein Cabal-Projekt verstehen

Das Template ist ein Cabal-Projekt, das mit dem `stack`-Tool gebaut werden kann.

Die Konfiguration von `stack` befindet sich in der Datei `stack.yaml`. Darin steht im Wesentlichen, dass das LTS Haskell in der Version 8.5 verwendet werden soll. Infos was darin enthalten ist, finden Sie unter https://www.stackage.org/lts-8.5.

Das Cabal-Projekt wird über die Datei `FirstHaskellProjekt.cabal` konfiguriert. In der Datei sehen Sie verschiedene Bereiche. Der erste Bereich sind allgemeine Infos, wie z.B. Name, Autor, Copyright, …

In den weiteren Bereichen werden die verschiedenen Artefakte konfiguriert. Ein solcher Bereich beginnt immer mit einer Zeile in der ersten Spalte und alles was dazu gehört, ist darunter eingerückt.

Es wird also eine Library erstellt. Diese heisst immer wie das gesamte Package, also `FirstHaskellProjekt`. Der Quellcode befindet sich im Verzeichnis `src`, als Abhängigkeit wird nur das `base`-Paket benötigt und als einziges Modul wird `Lib` von der Library exportiert.

Als nächstes wird ein Executable konfiguriert. Der Name steht direkt dahinter. So können in einem Cabal-Projekt auch mehrere, verschiedene Executables erzeugt werden. Der Quellcode befindet sich im Verzeichnis `app`, die `main`-Funktion befindet sich in der Datei `Main.hs`, es werden einige Flags für den GHC gesetzt, Abhängigkeiten sind zu dem `base`-Package und zur vorher definierten Library definiert.

Der dritte Bereich ist die Test-Suite und der vierte das Repository in dem die Projektsourcen sein könnten.

Viel davon können Sie beliebig definieren. Die Struktur mit `src`, `app` und `test` ist üblich, aber natürlich nicht zwingend.

Die Datei `Setup.hs` wird von Cabal benötigt und muss von uns nicht weiter betrachtet werden.

Den Code verstehen

Haskell-Code wird in sog. Module geschrieben. Im Verzeichnis `src` finden Sie die Datei `Lib.hs` mit folgendem Inhalt

```
1 module Lib
2     ( someFunc
3     ) where
4
5 someFunc :: IO ()
6 someFunc = putStrLn "someFunc"
```

Die Zeilen 1-3 definieren die Signatur des Moduls. Das Modul heißt `Lib` (passend zum Dateinamen `Lib.hs`) und exportiert die Funktion `someFunc`. Über die sog. Exportliste wird die Sichtbarkeit gesteuert. Alle Funktionen, die vorhanden wären und nicht in der Exportliste aufgeführt sind, sind *private* in dem Modul.

Mit dem Kommando

```
stack repl
```

können Sie den gesamten Code des Projektes in den GHCi laden. Dort können Sie z.B. `someFunc` direkt eintippen.

Fügen Sie eine Funktion `square`, die einen `Integer` quadriert, zum Modul `Lib` hinzu. Damit Sie `square` auch außerhalb des Moduls nutzen können, fügen Sie es zur Exportliste hinzu:

```
module Lib
    ( someFunc
    , square
    )
```

Nach einem Reload im GHCi mit dem Kommando

```
:r
```

können Sie die Funktion `square` nutzen.

Um Ihre Funktion `square` im Executable zu nutzen, müssen Sie sie importieren. Nachdem in der Datei `app/Main.hs` bereits

```
import Lib
```

steht, werden alle Funktionen, die `Lib` exportiert, importiert. Sie können auch, was nebenbei noch viel besserer Stil ist, die Importliste auf das einschränken was Sie wirklich benötigen. Im Moment also

```
import Lib (someFunc)
```

Für `square` erweitern Sie Zeile dann wie folgt

```
import Lib (someFunc, square)
```

Erweitern Sie die `main`-Funktion anschließend wie folgt und probieren Sie sie aus.

```
1 main = do
2   putStrLn "Geben Sie bitte eine Zahl ein (0 == Ende): "
3   number <- readLn
4   putStrLn ("square(" ++ show number ++ ") = " ++ show (square number))
5   if number == 0
6     then putStrLn "Ciao"
7     else main
```

Wenn Sie den Code analysieren, werden Sie feststellen, dass der Ablauf auch ohne eine Schleife wiederholt werden kann.

Vielleicht gefällt Ihnen nicht, dass das Programm nach der Aufforderung eine Zahl einzugeben, immer in die nächste Zeile springt. Ändern Sie doch einfach mal `putStrLn` in der zweiten Zeile in `putStr` und starten Sie das Programm. Geben Sie eine Zahl ein, auch wenn da nichts steht. Sie sehen dann so eine Ausgabe:

```
12
Geben Sie bitte eine Zahl ein (0 == Ende): square(12) = 144
122
Geben Sie bitte eine Zahl ein (0 == Ende): square(122) = 14884
0
Geben Sie bitte eine Zahl ein (0 == Ende): square(0) = 0
Ciao
```

Das ist ein Phänomen des Betriebssystems bzw. der von Haskell genutzten C-Bibliothek. Der Puffer wird erst mit einem `newline` auf den Bildschirm geschrieben. Die Lösung ist natürlich die selbe wie in C, aber mit etwas anderer Syntax.

Fügen Sie folgenden Import zu Ihrer App hinzu (Importe stehen wie in Java immer oben):

```
import System.IO (BufferMode (NoBuffering), hSetBuffering, stdout)
```

und fügen Sie als erste Zeile des `do`-Blockes der `main`-Funktion folgendes ein:

```
  hSetBuffering stdout NoBuffering
```

Damit haben Sie das Puffern auf `stdout` ausgeschaltet und das Programm sollte sich jetzt wie gewünscht verhalten:

```
Geben Sie bitte eine Zahl ein (0 == Ende): 12
square(12) = 144
Geben Sie bitte eine Zahl ein (0 == Ende): 122
square(122) = 14884
Geben Sie bitte eine Zahl ein (0 == Ende): 0
square(0) = 0
Ciao
```

Ändern Sie Ihre `main`-Funktion so ab, dass beim Ende das Quadrat von `0` nicht mehr ausgegeben wird.

Ändern Sie als nächstes die `square`-Funktion so, dass sie nicht nur `Integer`, sondern auch andere Zahlen akzeptiert und starten Sie Ihre Anwendung erneut.

Probieren Sie das Quadrat von `1.2` zu berechnen:

```
Geben Sie bitte eine Zahl ein (0 == Ende): 1.2
FirstHaskellProjekt-exe: user error (Prelude.readIO: no parse)
```

Warum passiert das? Wenn Sie `square` im GHCi (`stack repl`) ausprobieren, werden Sie sehen, dass es nicht an `square` liegt. Das Problem macht an dieser Stelle der Typinferenzmechanismus. In der `main` steht in einer Zeile

```
if number == 0
```

Das Literal kann zwar für eine beliebige Zahl stehen, aber wenn ein Programm dann gelinkt wird, muss ein Typ fest gewählt werden. Solange der GHC nicht mehr über die `0` weiß, nimmt er an es ist ein `Integer`. Damit ist aber `number` auch ein `Integer` und `readLn` zum Parsen eines (fast) beliebigen Wertes, will dann aus dem eingegebenen String `1.2` einen `Integer` machen.

Es gibt zwei Möglichkeiten das Problem zu lösen:

1. (die weniger elegante): Sie schreiben statt `number == 0`:

   ```
   if number == 0.0
   ```

2. (die elegantere): Sie schränken `readLn` durch Angabe eines exakten Typs ein:

   ```
   number <- readLn :: IO Double
   ```

In beiden Fällen akzeptiert nun das Programm auschließlich Zeichenketten, die in einen `Double` umgewandelt können, aber zum Glück geht das in Haskell auch ohne Punkt (sie sehen das z.B. an der Ausgabe zur Zahl 2):

```
Geben Sie bitte eine Zahl ein (0 == Ende): 1.2
square(1.2) = 1.44
Geben Sie bitte eine Zahl ein (0 == Ende): 2
square(2.0) = 4.0
Geben Sie bitte eine Zahl ein (0 == Ende): 0
Ciao
```

Haddock

Was für Java JavaDoc ist, ist für Haskell Haddock. Damit nehmen wir uns als erstes das Modul `Lib` vor. Wir dokumentieren es z.B. so

```
-- | A Lib module.
module Lib (square) where

-- | Calculate the square of a number
square :: Num a
       => a -- ^ the number
       -> a -- ^ the square
square n = n^2
```

Erzeugen wir anschließend mit Stack die Doku (siehe weiter vorne), können wir unsere Kommentare in den HTML-Seiten sehen.

Testen

Haskell Code wird üblicherweise mit HUnit und/oder QuickCheck getestet. HSpec ist ein Testing-Framework mit einer angenehmen Syntax das beide Ansätze enthält.

Als erstes ersetzen wir im Verzeichnis `test` den Inhalt der Datei `Spec.hs` durch die einzige Zeile

```
{-# OPTIONS_GHC -F -pgmF hspec-discover #-}
```

als Inhalt. Dies weist den GHC an im aktuellen Verzeichnis nach Dateien zu suchen, die mit `Spec.hs` enden und als Modul eine Funktion `spec :: Spec` exportieren.

Um die Tests mit `stack` ausführen zu können, erweitern wir die `build-depends` im Abschnitt `test-suite` um die Packages `hspec` und `QuickCheck`:

```
test-suite FirstHaskellProjekt-test
  ...
  build-depends:       base
                     , Blatt01
                     , hspec
                     , QuickCheck
  ...
```

Nachdem es nun durch `hspec` einige Packages gibt, die vorher noch nicht in unserem Projekt enthalten waren, wird bei

```
$ stack test
```

einiges neu installiert. Das macht Stack alles automatisch.

Jetzt wollen wir aber natürlich auch etwas testen. Wir schreiben eine Spezifikation für unser Lib-Modul. Dazu schreiben wir in die Datei `LibSpec.hs` im Verzeichnis `test`:

```
1  {-# LANGUAGE ScopedTypeVariables #-}
2  module LibSpec (spec) where
3
4  import           Lib             (square)
5  import           Test.Hspec
6  import           Test.QuickCheck
7
8  spec :: Spec
9  spec =
10     describe "square" $ do
11         it "calculates the square of 5.3" $
12             square 5.3 `shouldBe` 28.09
13         it "calculates the square of an arbitrary integer" $
14             property $ \(n :: Integer)  -> square n == n * n
15         it "calculates the square of an arbitrary double" $
16             property $ \(n :: Double)  -> square n == n * n
```

Die `spec`-Funktion ist eine Spezifikation. Als solche soll sie möglichst gut, auch für Nicht-Programmierer, lesbar sein.

Der erste Test in den Zeilen 11-12 ist ein HUnit-Test. Ein Ausdruck `square 5.3` wird berechnet und soll zum Ergebnis `28.09` kommen.

Der zweite Test in den Zeilen 13-14 ist etwas ungewöhnlicher. Er spezifiziert eine Eigenschaft (*property*), nämlich, dass für alle ganzen Zahlen `n` gilt: `square n` ist gleich `n * n`. Wir müssen `n` für den Test auf einen konkreten Typ einschränken, in dem Fall `Integer`. Damit wir das elegant in dem Lambda-Ausdruck machen können, nutzen wir eine Erweiterung des GHC mit dem Namen `ScopedTypeVariables`, die wir über das Language-Pragma in Zeile 1 aktivieren.

Wie kann jetzt aber das Property getestet werden? Die dazu notwendige QuickCheck-Bibliothek erzeugt Zufallswerte, standardmäßig 100 Stück, und füttert die Funktion damit. Für jeden Wert wird dann überprüft ob die Gleichheit gilt.

In den Zeilen 15-16 wird das Gleiche für `Double`s überprüft.

Um die Tests nun auszuführen, müssen Sie dem GHC die Chance geben, die neue Datei zu finden. Am Einfachsten gelingt das, wenn Sie das Projekt erst "cleanen" und dann neu bauen:

```
$ stack clean
$ stack test
```

Aufgabe

Erweitern Sie das Modul `Lib` um eine Funktion `ggT`, die den größten gemeinsamen Teiler zweier `Integer` berechnet.

```
ggT :: Integer -> Integer -> Integer
```

Sofern in der Vorlesung noch nicht komplett besprochen, sehen Sie sich dazu die weiteren Folien des Haskell-Einführungskapitels an, insbesondere zu `if-then-else` und Rekursion.

Sie können Ihre Funktion anschließend im GHCi ausprobieren:

```
stack repl
```

oder Sie schreiben Ihre `main`-Funktion um, so dass Sie zwei Zahlen einlesen können und der größte gemeinsame Teiler berechnet und ausgegeben wird.

Was Sie auf jeden Fall noch machen sollen, ist Tests für `ggT` zu schreiben. Öffnen Sie dazu noch einmal die Datei `LibSpec.hs`. Nachdem Sie mehr als eine Funktion *beschreiben* wollen, müssen Sie einen `do`-Block verwenden, d.h. Sie ersetzen die Zeile

```
spec =
```

durch

```
spec = do
```

Das ist analog zu geschweiften Klammern. Wenn Sie nur einen Ausdruck, d.h. hier nur ein `describe` haben, ist das `do` nicht notwendig. Jetzt fügen Sie ein zweites `describe` mit dem `d` exakt unter dem ersten `d` des ersten `describe` in der Zeile unter dem letzten Property an. Darunter können Sie analog zu den Tests für `square` weitere Tests anfügen.

Fügen Sie mindestens einen HUnit-Test an, der den GGT von zwei Zahlen überprüfen soll. Fügen Sie Außerdem ein Property an, dass dieses Mal jeweils zwei Zufallszahlen erzeugt:

```
property $ \a b -> ...
```

Nachdem `ggT` nur für `Integer` definiert ist, müssen wir bei `a` und `b` natürlich keinen Typ angeben. Wenn, wie bei `square`, verschiedene Typen möglich wären, könnten wir den Typ von `a` und `b` wieder entsprechend einschränken:

```
property $ \(a :: Integer) (b :: Integer) -> ...
```

Das Property soll überprüfen ob bei ihrer Funktion `ggT` das selbe heraus kommt wie beim der bereits verfügbaren Funktion `gcd`.

Mit

```
stack test
```

sollten Sie jetzt eine solche (oder ähnliche) Ausgabe sehen:

```
...
Progress: 1/2
Lib
  square
    calculates the square of 5.3
    calculates the square of an arbitrary integer
    calculates the square of an arbitrary double
  ggT
    calculates the gcd of 96 and 126
    calculates the same as Haskells gcd

Finished in 0.0063 seconds
5 examples, 0 failures
```