Funktionale Programmierung

Blatt 1

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 22.10.2018 10:50

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

Gemeinsames Repository

Es gibt ein gemeinsames Repository für die Veranstaltung. Damit ich weiß wer von Ihnen am Praktikum teilnimmt um einen Schein zu bekommen, gibt es zwei Möglichkeiten darauf zuzugreifen:

1. Wenn Sie **keinen** Schein benötigen, greifen Sie einfach direkt auf das Repository zu.
2. Wenn Sie einen Schein bekommen wollen, treten Sie über den GitHub-Classroom-Link `https://classroom.github.com/g/9ilKdM6_` dem **einzigen** Team `ws18` bei. Erzeugen Sie **auf keinen Fall** ein anderes Team. Wählen Sie dabei Ihre E-Mail-Adresse aus, damit ich Ihnen Ihren GitHub-Account richtig zuordnen kann. Wenn Ihre E-Mail-Adresse nicht auftaucht, heisst das Sie sind im ZPA nicht für diese Veranstaltung eingetragen und können keine Leistungen ablegen.

Haskell

Wir programmieren in Haskell und nutzen die Haskell Platform auf den Laborrechnern unter Windows. Wenn Sie auf Ihrem eigenen Rechner arbeiten wollen, müssen nur Stack per Hand installieren. Über Stack installieren Sie dann den Glasgow Haskell Compiler und den dazugehörigen Interpreter und können mit einem beliebigen Editor Haskell-Dateien bearbeiten. Es gibt noch einige mehr oder weniger ausgereifte IDEs. Die meisten Haskeller nutzen Texteditoren, wie vim oder Emacs mit Haskell-Plugins.

Einen relativ aktuellen Überblick über verschiedene Ansätze gibt dieses Chart.

Ich selbst war bis vor Kurzem sehr zufrieden mit dem Atom und IDE-Haskell. Einziges Manko ist, dass ein Tool, dass dazu benötigt wird, nicht mit dem aktuellen Glasgow Haskell Compiler arbeiten. Sie können aber einfach mit Stack auf LTS-9.21 gehen, dann funktioniert alles einwandfrei.

Um mit der aktuellen GHC-Version arbeiten zu können, bin ich kürzlich (temporär?) auf VS Code mit der Haskero-Erweiterung umgestiegen.

Auf GitHub finden Sie ein Repository Laborrechner-Config in dem ich unter anderem Infos zur Nutzung von Haskell auf Windowsrechnern, wie wir Sie im Labor haben, sammele. **Über Bug-Reports und Tipps freue ich mich**.

Wenn Sie beispielsweise das cmd-Fenster über einen Doppelklick auf die Datei haskell-cmd.bat öffnen, ist der im Wintersemester 2017/18 benötigte Pfad richtig gesetzt und `stack` etc. wird gefunden.

Ihre Haskell-Installation können Sie mit der Datei testhaskell.bat testen.

Quellcode sollte immer **schön** formatiert sein, aber Quellcode selbst formatieren macht wenig Spaß. Für Haskell gibt es verschiedene Formatierer, die Sie auch in Ihren jeweiligen Editor einbinden können. **Nutzen Sie einen davon**.

Ich selbst habe bis vor Kurzem hindent genutzt. Die Formatierung mit dem relativ neuen brittany gefällt mir aber etwas besser. Einen Vergleich finden Sie zum Beispiel hier. Wenn Sie keine eigenen Vorlieben haben, nutzen Sie bitte brittany.

Aufgabe 1 — Erste Schritte mit Haskell

Der GHC bringt einen sog. REPL (**R**ead **E**valuate **P**rint **L**oop) mit dem Namen GHCi mit. Sie starten ihn auf der Kommandozeile mit dem Befehl `stack repl`[1] hinter dem Prompt (hier als `$` dargestellt):

```
$ stack repl
...
Prelude>
```

Hinter dem `Prelude>`-Prompt können Sie einen Haskell-Ausdruck eingeben und diesen unmittelbar evaluieren lassen. Probieren Sie mindestens die folgenden Ausdrücke aus:

```
2 + 3
sqrt 15
print 12
show 12
```

[1] Wenn Sie die Haskell-Plattform installiert haben, können Sie stattdessen auch das Kommando `ghci` eingeben.

Der GHCi hat weiterhin Kommandos mit denen Sie z.B. zusätzliche Informationen erhalten können. Probieren Sie mindestens folgendes aus:

```
:h
:t "Hallo"
:i String
:browse Data.Char
```

Aufgabe 2 — Ein erstes Programm

Ein Haskell-Programm besteht aus Modulen. Ein Modul entspricht dabei einer Datei. Wir beginnen mit einem ganz einfachen Programm mit dem wir das Quadrat einer ganzen Zahl berechnen können.

Öffnen Sie dazu eine Datei `Foo.hs` mit einem beliebigen Texteditor, z.B. dem Atom oder VS Code, in einem beliebigen Verzeichnis, in dem Sie auch per Kommandozeile sind. Definieren Sie eine Funktion mit dem Namen `square` die das Quadrat einer ganzen Zahl berechnet. Schreiben Sie dazu als einzige Zeile

```
square n = n^2
```

in das Haskell-Modul.

Um die Funktion auszuprobieren, können Sie Ihr Modul in den GHCi laden:

```
$ stack repl
...
Prelude> :l Foo
```

Der Prompt wechselt damit zu `Main>`, da Sie nun das Modul `Main` geladen haben, auch wenn wir dem Modul in der Datei `Foo.hs` keinen expliziten Namen gegeben haben.

Wenn Sie in Ihrem Modul etwas ändern und die Datei abspeichern, können Sie sie mit `:r` im GHCi reloaden.

Berechnen Sie nun ein paar Quadrate.

Lassen Sie sich mit `:t square` den Typ anzeigen:

```
Main> :t square
square :: Num a => a -> a
```

Das bedeutet, dass Sie `square` auf jeden Typ `a` anwenden können, der in der Typklasse `Num` ist. Sehen Sie sich an, wer alles in der Typklasse `Num` ist und welche Funktione in der Klasse definiert sind, durch

```
:i Num
```

D.h. Sie haben durch die einfache Definition oben gleich eine polymorphe Funktion definiert, die Sie auf verschiedene Typen anwenden können. Und **Achtung**: Es gibt in Haskell kein implizites Umwandeln (*casten*) eines Wertes in einen anderen (wie z.B. `int -> double` in Java).

Das geht noch weiter. Sehen Sie sich doch mal den Typ der Zahl `1` an:

```
Main> :t 1
1 :: Num p => p
```

D.h. auch die 1 ist polymorph. Nochmal: Es wird nichts umgewandelt, sondern die `1` ist als Literal **überladen**. Je nach Kontext steht das Literal `1` dann aber für einen konkreten Typ.

Betrachten wir als nächstes beispielsweise die vordefinierte Funktion `sqrt`. Lassen Sie sich den Typ anzeigen. Sie sehen wieder eine polymorphe Funktion, aber diesesmal mit einer anderen Typklasse. Sehen Sie sich auch dieses Mal an, was die Typklasse enthält und welche Typen dazu gehören.

Es gehört zum guten Stil— auch wenn Haskell die Typen im Normalfall selbst berechnen kann —für alle Top-Level-Definitionen, wie in unserem Fall `square`, den Typ anzugeben.

D.h. wir ergänzen unser Modul `Foo`, so dass es folgendermaßen aussieht:

```
square :: Num a => a -> a
square n = n^2
```

Damit ändert sich gar nichts, wir haben nur unseren Code *lesbarer* gemacht. Wir können aber so eine explizite Typangabe auch nutzen um eine Funktion auf eine Menge von Typen einzuschränken.

Ersetzen Sie die **Typsignatur** zunächst durch:

```
square :: Integer -> Integer
```

und dann durch

```
square :: Fractional a => a -> a
```

und probieren Sie die Funktion im GHCi aus. Wenn Sie im zweiten Fall (`Fractional`) z.B. `square 1` eingeben, bekommen Sie als Ergebnis `1.0`, ein Literal das für einen `Float` oder einen `Double` stehen kann.

Um unser Miniprogramm außerhalb des GHCi nutzen zu können, benötigen wir, wie in anderen Sprachen, eine `main`-Funktion. Ergänzen Sie Ihr Modul `Foo` um folgende `main`-Funktion:

```haskell
main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  putStr "Geben Sie bitte eine Zahl ein (0 == Ende): "
  number <- readLn
  putStrLn ("square(" ++ show number ++ ") = " ++ show (square number))
  if number == 0
    then putStrLn "Ciao"
    else main
```

Fügen Sie außerdem **vor** allen Funktionsdefinitionen folgendes Import-Statement hinzu:

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

Diese Buffering-Sachen benötigen wir nur, da das Betriebssystem normalerweise die Ausgaben puffert und erst mit einem Newline wirklich aus gibt. Sie können die Zeile in der `main`-Funktion ja mal auskommentieren und ausprobieren was dann passiert. Um diese speziellen Funktionen zu nutzen, müssen wir Sie erst importieren.

Sie können dieses Programm jetzt auf verschiedene Arten ausführen:

Im GHCi:

```
*Main> main
```

oder

```
*Main> :main
```

Mit der zweiten Version könnten Sie dahinter im GHCi Kommandozeilenargumente angeben.

Oder als Script direkt auf der Kommandozeile:

```
$ stack runhaskell Foo
```

oder

```
$ stack runhaskell Foo.hs
```

Oder Sie compilieren es und führen es dann aus:

```
$ stack ghc Foo
```

oder

```
$ stack ghc Foo.hs
```

und dann

```
$ ./Foo
```

Probieren Sie das Programm aus.

Ändern Sie die Typsignatur von `square` wieder zu

```
square :: Num a => a -> a
```

Funktioniert es immer noch für `Int`, `Integer`, `Float` und `Double`?

Es scheint nur noch mit ganzen Zahlen zu gehen:

```
Geben Sie bitte eine Zahl ein (0 == Ende): 12
square(12) = 144
Geben Sie bitte eine Zahl ein (0 == Ende): 1.2
Foo.hs: user error (Prelude.readIO: no parse)
```

Aber woran kann das liegen? `square` kann es nicht sein. Sehen Sie sich mal im GHCi an was `show` und `readLn` für Typen haben:

```
Prelude> :t show
show :: Show a => a -> String
Prelude> :t readLn
readLn :: Read a => IO a
```

`show` nimmt einen Wert von einem beliebigen Typ `a` der in der Typklasse `Show` ist. Damit ist `show` so etwas wie `toString()` in Java. Sollte also kein Problem machen.

`readLn` ist da schon etwas spezieller. Die Funktion macht I/O und gibt danach einen Wert von einem beliebigen Typ der in der Typklasse `Read` ist zurück. Erstmal doch nichts Ungewöhnliches. Oder etwa doch?

Die Funktion `readLn` könnte es in Java gar nicht geben, denn es ist eine polymorphe Funktion deren Instanzen sich nur durch den Rückgabetyp unterscheiden. Und das ist in Java schlichtweg nicht erlaubt, weil die Laufzeitumgebung nicht die richtige Funktion auswählen kann.

Aber wie macht Haskell das? Haskell sieht an späterer Stelle wird auf das Ergebnis von `readLn`, nämlich `number`, die Funktion `square` angewendet. Dazu muss eine `readLn`-Funktion gebunden werden, die eine Zahl (`Num a => a`) zurück gibt. Aber das ist für ein konkretes Programm immer noch zu polymorph. Haskell muss eine konkrete Funktion

auswählen. Und wenn es nur von einer Zahl weiß, nimmt es die Instanz für `Integer`. Also läuft unser Programm nur für ganze Zahlen.

Was aber wenn wir einen `Double` eingeben wollen?

Das bekommen wir hin in dem wir dem Haskell-Compiler durch eine *Typ-Annotation* einen Tipp geben, was wir haben wollen. Wir schreiben hinter die Zeile mit dem `readLn` einfach den konkreten Typ von `readLn`:

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

Jetzt funktioniert es für `Double`. Wir können aber immer noch einfach nur eine `1` eingeben. In dem Fall hat die `1` aber den Typ `Double` :-)

Aufgabe 3 — Ein erstes Projekt

Wir wollen jetzt gleich im nächsten Schritt ein richtiges Haskell-Projekt mit allem Drum und Dran erstellen. Da können wir dann Übersetzen, Ausführen, Doku generieren, Tests ausführen, usw.

Für Haskell gibt es ein spezielles Build-Tool Cabal, so wie `make` für Unix, `sbt` für Scala, `maven` für Java, … Über Stack wird Cabal gleich mit eingebunden. Gesteuert wird ein Haskell-Projekt dann durch eine Datei die die Endung `.cabal` trägt. Da die `.cabal`-Datei einige Redundanzen enthält, wird diese automatisch aus einer einfacheren `package.yaml` generiert.

Wir wollen unser Projekt `MyFirstProject` nennen, es in einem GitHub-Repository speichern und mit Travis CI bauen und testen.

Erzeugen Sie als nächstes ein neues Stack-Projekt aus einem Template. Wechseln Sie dazu auf der Kommandozeile in ein Verzeichnis in dem dann ein Unterverzeichnis für das neue Projekt erstellt wird. Geben Sie dann das Kommando:

```
$ stack new MyFirstProject
```

ein. Der Inhalt des Verzeichnisses ist anschließend:

```
MyFirstProject
   ChangeLog.md
   LICENSE
   MyFirstProject.cabal
   README.md
   Setup.hs
   app
       Main.hs
```

```
package.yaml
src
    Lib.hs
stack.yaml
test
    Spec.hs
```

Sehen Sie sich die verschiedenen Dateien an. Das Projekt ist so aufgebaut, dass es eine Applikation im Verzeichnis `app` gibt und diese eine Library im Verzeichnis `src` nutzt. Außerdem ist schon ein Verzeichnis `test` für die Tests angelegt.

Die Datei `package.yaml` enthält die Konfiguration des Projektes. Darin können Sie Ihren Namen etc. anpassen und im Anschluß auch weitere Anpassungen des Projektes machen. Die Datei `Setup.hs` wird von Cabal genutzt. Lassen Sie diese einfach unverändert. Die Datei `MyFirstProject.cabal` wird automatisch aus der Datei `package.yaml` generiert. Änderungen, die Sie direkt in `MyFirstProject.cabal` machen, werden beim nächsten Aufruf von `stack` einfach überschrieben.

In der Datei `stack.yaml` können Sie Stack konfigurieren. Das ist aber für uns ausreichend vorkonfiguriert.

Wir können das Projekt nun schon bauen lassen, obwohl wir noch gar kein Stück eigenen Programmcode geschrieben haben.

```
$ stack build
```

Um die Applikation dann auszuführen, geben wir den in `Blatt01.cabal` spezifzierten Namen an:

```
$ stack exec MyFirstProject-exe
```

Um die Tests auszuführen, geben wir folgendes ein:

```
$ stack test
```

Bevor wir nun selbst etwas an dem Projekt verändern, wollen wir den initialen Stand auf GitHub pushen und von Travis bauen und testen lassen. Ein leeres GitHub-Repository bekommen Sie über den Link https://classroom.github.com/a/x2-m2DS5.

Öffnen Sie dann die Webpage des erzeugten GitHub-Repositories. In der Mitte der Seite finden Sie den Punkt `…or create a new repository on the command line`. Die erste Zeile (`echo ...`) streichen wir und dritte Zeile ersetzen wir durch:

```
git add .
```

Statt `first commit` nehmen wir vielleicht besser sowas wie `initial stack project`. Nach dem `git push` sollten Sie den Code nun auf der GitHub-Seite sehen.

Wichtig: Ihr Projekt muss **top-level** im Git-Repository liegen. Das heisst, der **Inhalt** von `MyFirstProject` und **nicht** das Verzeichnis `MyFirstProject`!

Neben GitHub nutzen wir noch Travis CI als Continuous Integration System. Um es in dem Repository zu aktivieren, müssen wir einfach eine Datei `.travis.yml` erstellen. Für ein Stack-Projekt wird es unter der URL https://docs.haskellstack.org/en/stable/travis_ci/ ausführlich beschrieben. Fügen Sie den folgenden Inhalt in eine Datei mit dem Namen `.travis.yml` und pushen Sie diese auf GitHub (Achtung: Code aus dem PDF zu kopieren funktioniert nicht vernünftig. Sie finden die komplette Datei, wie sie am Ende dieses Blattes benötigt wird, in folgendem Gist):

```
# Use new container infrastructure to enable caching
sudo: false

# Do not choose a language; we provide our own build tools.
language: generic

# Caching so the next build will be fast too.
cache:
  directories:
  - $HOME/.stack

before_install:
# Download and unpack the stack executable
- mkdir -p ~/.local/bin
- export PATH=$HOME/.local/bin:$PATH
- travis_retry curl -L https://get.haskellstack.org/stable/linux-x86_64.tar.gz \
    | tar xz --wildcards --strip-components=1 -C ~/.local/bin '*/stack'

# Ensure necessary system libraries are present
addons:
  apt:
    packages:
      - libgmp-dev

install:
# Build dependencies
- stack --no-terminal --install-ghc test --only-dependencies

script:
# Build the package, its tests and run the tests
- stack --no-terminal test
```

Nach dem Pushen sollten Sie hinter dem Commit auf der Seite der Commits auf GitHub einen Indikator für das CI sehen. Ein oranger Punkt heißt es läuft gerade ein Build. Ein grüner Haken und ein rotes Kreuz sprechen für sich. Ein Klick darauf bringt mehr Infos bzw. führt Sie auf die Travis-Seite auf der Sie z.B. das Build-Log anschauen können.

Nun wollen wir unser Projekt aber etwas mit Leben füllen. Dazu erweitern wir die Datei `Lib.hs` im Verzeichnis `src` um die `square`-Funktion von oben. Diese Funktion wollen wir von außen nutzbar machen, dazu müssen wir sie zur Exportliste hinzufügen. Die Funktion `someFunc` entfernen wir.

```
1 module Lib
2     ( square
3     ) where
4
5 square :: Num a => a -> a
6 square n = n^2
```

Wir wollen jetzt aber wieder ein ausführbares Programm erzeugen. Dazu ersetzen wir dir vorhandene `Main.hs` durch folgendem Inhalt:

```
module Main where

import           Lib       (square)
import           System.IO (BufferMode (NoBuffering), hSetBuffering, stdout)

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  putStr "Geben Sie bitte eine Zahl ein (0 == Ende): "
  number <- readLn :: IO Double
  putStrLn ("square(" ++ show number ++ ") = " ++ show (square number))
  if number == 0
    then putStrLn "Ciao"
    else main
```

Jetzt können wir mit Stack bauen und ausführen.

```
$ stack build
$ stack exec MyFirstProject-exe
```

Mit einem

```
$ stack repl
```

wird jetzt automatisch das gesamte Projekt in den GHCi geladen. D.h. wir können dort direkt auf `square` zugreifen.

Aufgabe 4 — Dokumentieren und Testen

Zur Software-Qualität gehört dokumentierter und getesteter Code.

Als Allererster stellen Sie sicher, dass in der `package.yaml` alle Infos (name, github, …) passen.

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
```

Mit dem Kommando

```
$ stack haddock
```

können wir nun eine API-Dokumentation in HTML erzeugen. Anschließend können wir die Dokumentation unterhalb von `.stack-work/dist/` finden und im Browser öffnen. Bei der Ausgabe von `stack haddock` wird auch angezeigt wie hoch die Abdeckung der Dokumentation ist.

In der `.travis.yml` können wir die letzte Zeile

```
- stack --no-terminal test
```

ersetzen durch

```
- stack --no-terminal test --haddock --no-haddock-deps
```

Damit wird beim Test auch Haddock erzeugt, aber nur für dieses Projekt und nicht für die *dependencies*.

Um die Dokumentation zu sehen, können wir vom Travis aus beispielsweise auf eine, zum Repository gehörende, GitHub Page deployen. Dazu fügen wir unter der letzten Zeile in der `.travis.yml`-Datei an:

```
- ln -s `stack path --local-doc-root` docs

deploy:
  provider: pages # speziell für GitHub Pages
  skip_cleanup: true # der Branch soll **nicht ** vorher gelöscht werden
  github_token: $GITHUB_TOKEN # das Token damit der Travis das darf,
                              # soll nicht in den Logs stehen, daher Variable
  on:
    branch: master # auch wenn alle Branches gebaut werden, deployed werden
                   # soll nur vom master
  local_dir: docs # in den Branch soll nur die Haddock-Dokumentation
```

Um nun als Travis wirklich in Ihr Repository pushen zu dürfen, erzeugen Sie ein GitHub-Token unter https://github.com/settings/tokens. Wählen Sie dabei unter `Select scopes` den Oberpunkt `repo` aus. Gehen Sie anschließend auf die Travis CI Plattform um den GitHub-Token dort zu hinterlegen. Ersetzen Sie in der GitHub-URL Ihres Repositorys einfach `github` durch `travis-ci`, z.B.

```
https://travis-ci.com/ob-fun-ws18/blatt-1-aufgabe-3-obcode
```

Loggen Sie sich mit Ihrem GitHub-Account ein. Rechts unter `More Options` finden Sie den Punkt `Settings`. Tragen Sie im Abschnitt `Environment Variables` eine neue Variable ein mit dem Namen `GITHUB_TOKEN` und dem Wert Ihres GitHub-Tokens. `Display value in build log` muss dabei ausgeschaltet bleiben. Wenn Sie jetzt pushen, sollte der Travis am Ende des Builds die erzeugte Dokumentation in den `gh-pages`-Branch ihres Repositorys pushen. Damit ist sie unter der entsprechenden URL erreichbar, z.B.

```
https://ob-fun-ws18.github.io/blatt-1-aufgabe-3-obcode/
```

Fügen Sie Ihre URL auf der GitHub-Page Ihres Repositorys oben zur Beschreibung hinzu. Klicken Sie dazu auf `Edit` rechts von dem Text `blatt-1-aufgabe-3-... created by GitHub Classroom` und tragen Sie die URL unter Website ein.

Achtung: Auch wenn das Repository `private` ist, sind die GitHub-Pages immer öffentlich zugänglich.

Testen

Haskell Code wird üblicherweise mit HUnit und/oder QuickCheck getestet. HSpec ist ein Testing-Framework mit einer angenehmen Syntax das beide Ansätze enthält. Solche Spezifikationen als Tests sind mittlerweile in vielen Sprachen üblich.

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 die mit `Spec.hs` enden und als Modul eine Funktion `spec :: Spec` exportieren.

Um die Tests mit `stack` ausführen zu können, erweitern wir die `dependencies` im Abschnitt `tests` um die Packages `hspec` und `QuickCheck`:

```
tests:
MyFirstProject-test:
    main:                Spec.hs
    source-dirs:         test
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - MyFirstProject
    - hspec
    - QuickCheck
```

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

```
$ stack test
```

nun einiges neu installiert. Das macht Stack alles automatisch.

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

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
module LibSpec (spec) where

import           Lib             (square)
import           Test.Hspec
import           Test.QuickCheck

spec :: Spec
spec =
    describe "square" $ do
        it "calculates the square of 5.3" $
            square 5.3 `shouldBe` 28.09
        it "calculates the square of an arbitrary integer" $
            property $ \(n :: Integer)  -> square n == n * n
        it "calculates the square of an arbitrary double" $
            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 wird 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 `Doubles` überprüft.

Um die Tests nun auszuführen, geben Sie ein

```
$ stack test
```

Wenn Sie jetzt committen und pushen, sollte der Travis diese Tests auch (erfolgreich) ausführen.

Benchmarks

Haskell bietet mit der Criterion-Bibliothek eine einfache Möglichkeit Benchmarks durchzuführen an. Wir erweitern unser Projekt dazu wie folgt. Zunächst erzeugen wir ein Unterverzeichnis `benchmark` und darin eine Datei `Bench.hs` mit folgendem Inhalt:

```
module Main (main) where

import           Criterion.Main (bench, bgroup, defaultMain, nf)
import           Lib

main :: IO ()
main = defaultMain
    [ bgroup "Lib" [ bench "1"     $ nf square (1     :: Int)
                   , bench "200"   $ nf square (200   :: Integer)
                   , bench "200.0" $ nf square (200.0 :: Double)
                   , bench "20.8"  $ nf square (20.8  :: Float)
                   ]
    ]
```

Damit werden 4 Benchmarks in einer Gruppe mit dem Namen `Lib` ausgeführt. Jeder Benchmark (`bench`) hat einen Titel und einen Ausdruck den er auswertet. Nachdem Haskell lazy ist, muss die Auswertung in die Normalform mit der Funktion `nf` erzwungen werden. Da die Funktion `square` polymorph ist müssen wir jeweils angeben, welchen Typ der Parameter und damit die Funktion hat.

Um die Benchmarks von Stack ausführen zu lassen, fügen wir folgenden Teil an die Datei `package.yaml` an:

```
benchmarks:
MyFirstProject-benchmark:
    main: Bench.hs
    source-dirs: benchmark
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - MyFirstProject
    - criterion
```

Anschließend können wir die Benchmarks mit dem Kommando

```
$ stack bench
```

ausführen.

Dies gibt uns Informationen über die Zeit, Mittelwert und Standardabweichung auf der Kommandozeile.

Wir können aber auch einen HTML-Report mit Graphen generieren lassen durch:

```
$ stack bench --benchmark-arguments="-o benchmarks.html"
```

Anschließend können wir die Datei `benchmarks.html` im Browser betrachten.

Wir können die Benchmarks natürlich auch auf dem Travis ausführen. Statt in den Logs die Ausgabe von `stack bench` anzusehen, lassen wir den Travis auch die report.html auf GitHub pushen.

Der Einfachheit halber ergänzen wir das `script`-Target in der `.travis.yml` einfach so:

```
script:
# Build the package, its tests and run the tests
- stack --no-terminal test --haddock --no-haddock-deps
- ln -s `stack path --local-doc-root` docs
- stack bench --benchmark-arguments="-o docs/benchmarks.html"
```

Damit liegt der Report direkt top-level auf der GitHub-Page, ist aber nicht verlinkt. Sie können sich natürlich auch selbst eine `index.html` basteln oder den Report einfach aus der `README.md` verlinken.

Z.B. nutzen Sie folgende `docs/index.html`:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>My First Project</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>
    <ul>
        <li><a href="doc/index.html">Haddock</a></li>
        <li><a href="report.html">Benchmarks</a></li>
    </ul>
</body>

</html>
```

und ändern in der `.travis.yml`:

```
script:
# Build the package, its tests and run the tests
- stack --no-terminal test --haddock --no-haddock-deps
- mv `stack path --local-doc-root` docs/doc
- stack bench --benchmark-arguments="-o docs/report.html"
```

Code-Qualität

Haskell bietet eine Menge an Möglichkeiten die Code-Qualität zu überprüfen.

Test Coverage

Für Haskell gibt es mit HPC ein Tool mit dem die Test Coverage gemessen werden kann. Mit `stack` kann es ganz einfach genutzt werden. Statt

```
$ stack test
```

verwenden Sie

```
$ stack test --coverage
```

bzw. in der `.travis.yml` hängen Sie an die entsprechende Zeile einfach das `--coverage` an:

```
- stack --no-terminal test --haddock --no-haddock-deps --coverage
```

Nachdem der Coverage-Report in `stack path --local-hpc-root` abgelegt wird, ergänzen Sie außerdem in der `.travis.yml` das `script`-Target einfach um

```
- mv `stack path --local-hpc-root` docs/hpc
```

und in der `docs/index.html` ergänzen Sie um:

```
<li><a href="hpc/index.html">Code Coverage</a></li>
```

Und schon können Sie nach dem nächsten Travis-Build Ihre Code-Coverage auf der GitHub-Page finden.

HLint

HLint inspiziert Haskell Code und gibt Verbesserungsempfehlungen. HLint gibt es als Kommandozeilentool das Sie durch

```
$ stack install hlint
```

installieren können. HLint lässt sich auch gleich in den Editor einbinden, so dass Sie direkt beim Speichern gleich Verbesserungsvorschläge bekommen.

Auf der Kommandozeile können Sie das installierte `hlint` in Ihrem Projekt so nutzen:

```
$ hlint .
```

Mit

```
$ hlint . --report
```

können Sie sich auch einen HTML-Report mit den Hinweisen generieren lassen.

In der `.travis.yml` ergänzen Sie das `script`-Target um

```
- travis_retry curl -sSL \
    https://raw.github.com/ndmitchell/hlint/master/misc/travis.sh \
    | sh -s .
```

Damit wird der Build rot, sobald `hlint` ein Problem findet.

Neben den vorgestellten Tools zur Code-Qualität gibt es z.B. noch die Möglichkeit die Dokumentations-Coverage zu überprüfen oder den Code, der in Kommentaren steht zu überprüfen.

Die letzten beiden Aufgaben habe ich, modulo Anpassungen und Übersetzung, von http://taylor.fausak.me/2014/03/04/haskeleton-a-haskell-project-skeleton/.