B

Haskell ausführen und dokumentieren

Dieser Anhang beschreibt, wie Haskell-Programme, "*Scripte*" genannt, ausgeführt und bei der Abgabe von Lösungsvorschlägen für Übungsblätter dokumentiert werden können. Das *Schreiben* von Haskell-Programmen wird im Hauptteil dieses Textes erklärt; eine systematische Beschreibung der Sprache Haskell – jedenfalls der für diesen Kurs relevanten Teilemenge – findet sich in Anhang C.

B.1 Haskell ausführen

Für die Bearbeitung der Programmieraufgaben soll das Glasgower Haskell-System benutzt werden. Das kommt mit einem Übersetzer `ghc`, der eigenständige Programme erzeugt. Wir werden meistens den Interpreter `ghci` ("*Glasgow Haskell compiler interactive*") benutzen.

Genau benutzen wir die Version 7.0.3. Für die Bearbeitung der Übungsaufgaben können zwar grundsätzlich auch andere Versionen des `ghci` oder andere Haskell-Systeme benutzt werden, jedoch geschähe dies auf eigene Gefahr: *Die abgegebenen Aufgaben werden nur anhand ihres Verhaltens unter der Referenzversion des `ghci` bewertet!*

Die folgenden Erläuterungen beziehen sich auf die Benutzung in Linux von der Kommandozeile.[1]

Beim Aufruf des `ghci` von der Kommandozeile gibt es folgende Möglichkeiten:

`ghci` [⟨*Options*⟩] [⟨*Script*⟩] ruft die interaktive Variante des Übersetzers auf und lädt ggf. die Datei ⟨*Script*⟩; dabei können die Endungen "`.hs`" oder "`.lhs`", weggelassen werden, die normalerweise für Haskell-Programme bzw. "literarische" Haskell-Programme verwendet werden. Wir empfehlen, die Option `-Wall` anzugeben, die alle Warnungen ausgibt.

`ghci --help` erklärt die möglichen Optionen des Aufrufs.

[1] Hinweise auf die Benutzung unter Windows bzw. MacOS wären mir willkommen – und noch mehr zukünftigen Studierenden!

Wird der Haskell-Compiler ohne Script interaktiv aufgerufen, erscheint – nach einigen Angaben über geladene und gelinkte Module – die Ausgabe "*Prelude*>".

Wurde ein eigenes Script angegeben, erscheint statt dessen die Ausgabe "*M*>", wenn dieses Script das Modul *M* enthält. Statt dessen können auch Fehler gemeldet werden, die bei der Übersetzung dieses Scriptes aufgetreten sind, oder bei der Übersetzung von Modulen, die von diesem Script importiert werden.

Im Interpreter kann man Ausdrücke auswerten und Informationen über das geladene Script und die Bibliothek erfragen.

⟨*Ausdruck*⟩ wertet den HASKELL-Ausdruck aus. Bei Fehlern wird gemeckert, sonst wird das Ergebnis der Auswertung ausgegeben.

`it` ist eine implizit definierte Variable, die den Wert des zuletzt ausgewerteten Ausrucks enthält.

`let` ⟨*Definition*⟩ führt Hilfsdefinitionen von Werten und Funktionen ein – das kann nützlich sein zum Testen. (Datentypen können hier nicht eingeführt werden.)

`:l[oad]` ⟨*Dateiname*⟩ lädt die angegebene HASKELL-Datei.

`:r[eload]` lädt die zuletzt geladene HASKELL-Datei erneut (z.B. nach Veränderungen im Editor)

`:i[nfo]` ⟨*Name*⟩ zeigt Informationen zu dem Typkonstruktor bzw. der Klasse ⟨*Name*⟩ an. Der Modul, in dem der Name definiert wird, muss geladen sein. Der Typkonstruktor für Listen ist "`[]`", die für n-Tupel ist "`(,`$^{n-1}$`)`". Der Typkonstruktor für Funktiontypen ist $(\rightarrow)$.

`:t[ype]` ⟨*Ausdruck*⟩ gibt den Typ des Ausdrucks an. Insbesondere können so die Typen aller sichtbaren "Variablennamen" (von Konstruktoren und Funktionen) angezeigt werden.

`:li[st]` ⟨*Name*⟩ gibt die Definition von ⟨*Name*⟩ in einem der geladenen Module an. Dies funktioniert aber nicht für Module aus der Bibliothek.

`:q[uit]` verlässt `ghci`.

B.2 Haskell dokumentieren

Haskell-Programme *beschreiben* kann man mit Kommentaren; sie sollen das Programm erläutern – nicht verzieren oder mit nutzlosem Text überschwemmen.

In Haskell gibt es zwei Arten von Kommentaren:

- *zeilenweise* Kommentare stehen zwischen "`--`" und dem nächsten Zeilenende.
- mehrzeilige Kommentare werden zwischen "`{-`" und "`-}`" eingeschlossen. Das geht auch geschachtelt.

Ein Beispiel zeigt beide Formen:

```
{- Lehrveranstaltung Funktionales Programmieren (PI3)
   (c) Berthold Hoffmann, Universität Bremen, hof@informatik.uni-bremen.de
   Beispielprogramme für die Vorlesung am 2. November 2011            -}

data List tau = Empty | Cons tau (List tau) -- Listen polymorph

cat :: List tau -> List tau -> List tau     -- Listenverkettung
cat  Empty      ys = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

B.3 Haskell literarisch beschreiben

In Haskell gibt es noch eine alternative, *literarische*, Sicht auf Programme, im Sinne des von Donald E. Knuth propagierten *Literate Programming* [Knu84].[2]: In *Literate Haskell* werden, in Dateien mit der Endung `.lhs` statt `.hs`, nicht die Kommentaranteile markiert, sondern umgekehrt der *Programmtext*. Alles andere ist Literatur. (*Ähm*, Kommentar.)

Hier gibt es wiederum zwei Möglichkeiten. Man kann alle Programmzeilen mit $>$ in der ersten Spalte kennzeichnen:

```
Lehrveranstaltung Funktionales Programmieren (PI3)
(c) Berthold Hoffmann, Universität Bremen, hof@informatik.uni-bremen.de
Beispielprogramme für die Vorlesung am 2. November 2011

> data List tau = Empty | Cons tau (List tau) -- Listen polymorph

> cat :: List tau -> List tau -> List tau     -- Listenverkettung
> cat  Empty      ys = ys
> cat (Cons x xs) ys = Cons x (cat xs ys)
```

(Die mit "`--`" eingeleiteten Zeilenkommentare gehen trotzdem noch.) Oder man schreibt Programmstücke zwischen `\begin{code}` und `\end{code}`:

```
Lehrveranstaltung Funktionales Programmieren (PI3) (c)
Berthold Hoffmann, Universität Bremen, hof@informatik.uni-bremen.de
Beispielprogramme für die Vorlesung am 2. November 2011

\begin{code}
data List tau = Empty | Cons tau (List tau) -- Listen polymorph

cat :: List tau -> List tau -> List tau     -- Listenverkettung
```

[2] Dort heißt es auf Seite 97: *I must confess that there may also be a bit of malice in my choice of a title. During the 1970s I was coerced like everybody else into adopting the ideas of structured programming, because I couldn't bear to be found guilty of writing* unstructured *programs. Now I have a chance to get even. By coining the phrase "literate programming," I am imposing a moral commitment on everyone who hears the term; surely nobody wants to admit writing an* illiterate *program.*

```
cat Empty       ys = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
\end{code}
```

Der Text "`\end{code}`" muss *unbedingt* am Zeilenanfang stehen!

Diese Art von literarischem Haskell ist besonders praktisch zur Benutzung mit LaTeX, weil man dann ein Programms in einem LaTeX-Dokument setzen kann und *die gleiche Datei* auch mit `ghci` Übersetzen kann. Dies soll auch für die Übungsaufgaben benutzt werden.

B.4 Haskell-Aufgaben literarisch dokumentieren

Die LaTeX-Klasse `pi3.cls`[3] unterstützt die Dokumentation von Übungsaufgaben. Der Lösungsvorschlag für eine Aufgabe kann so in eine Datei `Blatt<Nr>.tex` geschrieben werden:

```
\documentclass{pi3}
\tutor{<Tutorenname>}
\uebungsgruppe{<Gruppenname>}
\teilnehmer{<Name_1>\\<Name_1>\\<Name_3>}
\blatt{{<Nr>}}
\begin{document}
\maketitle
\input{Aufgabe01.lhs}
\input{Aufgabe02.lhs}
\input{Aufgabe03.lhs}
\end{document}
```

Jede Datei `Aufgabe<Nr>.lhs` enthält eine literarisches Haskell-Script.

- Mit `\solution{<Titel der Teilaufgabe>}` wird Nummer und Titel für die Lösung einer Teilaufgabe gesetzt.
- Zwischen `\begin{code}` und `\end{code}` (in der ersten Spalte einer Zeile) wird Code für den Haskell-Compiler geschrieben.
- Zwischen `\begin{xcode}` und `\end{xcode}` (in der ersten Spalte einer Zeile) wird Haskell-Code geschrieben, den der Haskell-Compiler *nicht* lesen soll – z.B. eine schon vorher definierte Funktion, deren Definition hier nur noch einmal gezeigt werden soll.
- Zwischen `\begin{comment}` und `\end{comment}` (in der ersten Spalte einer Zeile) steht Kommentar, der nicht im Dokument erscheint. Hierin kann man – zwischen `\begin{code}` und `\end{code}` – Code für den Haskell-Compiler schreiben, der im Text *nicht* auftauchen soll, z. B. weil er für die Dokumentation nicht relevant ist.
- Mit `\hs...!` kann im laufenden Text der Dokumentation ein Schnipsel Haskell-Code formatiert werden.

[3] Herunter zu laden unter der URL `http://www.informatik.uni-bremen.de/agbkb/lehre/pi3/aufgaben_d.htm`.

- Mit `\haskellcode{Datei.hs}` kann eine ganze – unliterarische – Haskell-Datei formatiert eingefügt werden. Dies ist aber nur für kurze Dateien zu empfehlen.

Auch dieser Text benutzt LaTeX auf diese Weise zur Formatierung.

B.5 Haskell poetisch formatieren

Für die Formatierung der Haskell-Programme in den Umgebungen `code` und `xcode` wird das LaTeX-Paket `listings` benutzt, mit dem Programme schön dargestellt werden können. Dies sorgt dafür, das Kommentare anders gefärbt werden, Schlüsselwörter im Text hervorgehoben werden, Bezeichner ("Variablen" in Haskell-Terminologie) geneigt geschrieben werden usw. Insbesondere werden einige Haskell-Symbole nach Tabelle B.1 durch mathematische Symbole ersetzt.

Mit den LaTeX-Umgebungen `code` und `xcode` werden Haskell-Programme so formatiert:

```
data List τ ≜ Empty | Cons τ (List τ) -- Listen polymorph

cat :: List τ → List τ → List τ -- Listenverkettung
cat Empty      ys ≜ ys
cat (Cons x xs) ys ≜ Cons x (cat xs ys)
```

Nimmt man im laufenden Text Bezug auf Haskell-Schnipsel, kann man mit dem Makro `\hs...!!` erreichen, dass die Schnipsel genau so aussehen wie im Code. Beispiel: "*Die Listen* `data List τ ≜ Empty | Cons τ(List τ) ...`".

Für die Schnipsel `\hs...!!` kann man auch ein anderes Zeichen als "!" benutzen; auf jeden Fall müssen sie aber *in einer Zeile stehen*.

(Dies ist Fassung A.A.001 von 2. November 2011.)

Haskell-Code	LaTeX-Formatierung	Bemerkungen
`=`	$\triangleq$	Definitionszeichen
`==`	$=$	Vergleichsprädikate
`/=`	$\neq$	
`<`	$<$	
`>`	$>$	
`<=`	$\leqslant$	
`>=`	$\geqslant$	
`.`	$\circ$	Funktionskomposition
`\`	λ	Lambda-Abstraktion
`&&`	$\wedge$	logische Operationen
`||`	$\vee$	
`not`	$\neg$	
`alpha`	α	Typvariablen
`beta`	β	
`gamma`	γ	
`delta`	δ	
`sigma`	σ	
`tau`	τ	
`` `elem` ``	$\in$	Mengenprädikate (auf Listen)
`` `notElem` ``	$\notin$	
`` `subset` ``	$\subseteq$	
`` `union` ``	$\cup$	Mengenoperationen (auf Listen)
`` `intersect` ``	$\cap$	
`\\`	$\setminus$	

Tabelle B.1. Poetisches Haskell