1. Übungsblatt

Typ-basiertes Programmieren und Schließen in Funktionalen Sprachen

Jun.-Prof. Dr. Janis Voigtländer / Dipl.-Math. Daniel Seidel

Wintersemester 2009/10

Aufgabe 1 (Warmup: Syntaxvarianten)
Schreiben Sie eine Funktion *sgn*, die für eine gegebene Zahl deren Signum (also: 1, 0 oder -1) liefert. Experimentieren Sie mit verschiedenen Varianten (explizites **if-then-else**, Pattern-Matching, Guards, lokale Funktionsdefinitionen, ...). ◇

Aufgabe 2 (Funktionen auf Listen)

1. Definieren Sie die folgenden Funktionen auf Listen:[1]
   - *head*: liefert das erste Element der Liste
   - *tail*: liefert die Liste ohne erstes Argument.
   - *take n*: nimmt die ersten n Elemente einer Liste
   - *reverse*: dreht eine Liste um (möglichst effizient!)
2. Nutzen Sie die soeben definierten Funktionen (und evtl. aus der Vorlesung bekannte Funktionen), um weitere Funktionen zu definieren:
   - *last*: liefert das letzte Element einer Liste
   - *getAt n*: gibt das $n$te Element einer Liste zurück
   - *splitAt n*: teilt eine Liste an der $n$ten Stelle
3. Definieren Sie die konstante *unendliche* Liste *fibs*, die alle Fibonacci-Zahlen enthält. (Hinweis: Schauen Sie bei Aufgabe 4, ob Sie etwas Brauchbares finden.) ◇

[1] Nutzen Sie nicht bereits definierte Äquivalente der Prelude-Bibliothek. Um Namenskonflikte zu vermeiden, beginnen Sie Ihre Datei mit „**import** *qualified Prelude*.“

Aufgabe 3 (Typen)

1. Fügen Sie allen bisherigen Funktionsdefinitionen die (allgemeinsten) Typannotationen hinzu.

2. Annotieren Sie die folgenden Werte mit ihren Typen:

   $[\texttt{'a'}, \texttt{'b'}]$
   $(\texttt{'a'}, \texttt{'b'})$
   $(\texttt{'a'}, 3)$
   $[(\mathit{False}, \texttt{'0'}), (\mathit{True}, \texttt{'1'})]$
   $([\mathit{False}, \mathit{True}], [\texttt{'0'}, \texttt{'1'}])$
   $[\mathit{tail}, \mathit{init}, \mathit{reverse}]$ ◇

Aufgabe 4 (Higher-Order-Funktionen)

1. Schreiben Sie folgende Funktionen:

   - *comp*: Funktionskomposition
   - *zipWith*: wie *zip*, nur wird eine Operation übergeben, mit der (statt Tupelbildung) die Listenelemente verknüpft werden
   - *zip* unter Verwendung von *zipWith*

2. Nutzen Sie *foldr*, um eine Version von *reverse* zu schreiben, die

   $$\mathit{reverse}\ [\,] = [\,]$$
   $$\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs +\!\!+ x$$

   entspricht.

3. Nutzen Sie *foldr*, um eine Version von *reverse* zu schreiben, die der effizienten Variante aus Aufgabe 2 entspricht.

4. Schreiben Sie *reverse* mittels *foldl*. (Nachdem Sie herausgefunden haben, was *foldl* macht.)

5. Schreiben Sie *map f* und *filter p* mit Hilfe von *foldr*.

6. Schreiben Sie *zip* mit Hilfe von (einem?, zwei?) *foldr*.

7. Schreiben Sie eine Funktion *sortBy*, die ein Sortierprädikat und eine Liste nimmt und Letztere entsprechend des Prädikats sortiert. ◇

Aufgabe 5 (Typklassen, und was Typen verraten)
Typklassen fassen Typen mit gemeinsamen Eigenschaften zusammenzufassen. Ein Beispiel ist die Klasse *Ord*. Auf jedem Typ, der eine Instanz der Klasse ist, sind $<$, $>$, $\leqslant$, $\geqslant$,*min*, *max* und $\equiv$ (verbatim: `==`) definiert. Somit kann z.B. ein Sortieralgorithmus den polymorphen Typ $Ord\ a \Rightarrow [a] \to [a]$ besitzen.

In Haskell sind unter Anderem die Typklassen *Eq*, *Ord*, *Show*, *Read* und *Num* definiert, wobei *Ord* in *Eq* enthalten ist und *Num* in *Show* und *Eq*.

1. Untersuchen Sie den allgemeinsten Typ der folgenden Funktionen:

   removeDuplicates

   removeDuplicatesBy

   group (gruppiert gleiche Elemente in eine Unterliste)

   $\lambda x \to print\ (\text{“Hallo”} +\!\!+ x)$

   $\lambda x \to print\ (\text{“Hallo”} +\!\!+ show\ x)$

2. Geben Sie an (bzw. schreiben Sie) jeweils eine oder mehrere Funktionen, die den folgenden allgemeinsten Typ haben:

   $Eq\ a \Rightarrow a \to [a] \to [a]$

   $(Num\ a, Eq\ b) \Rightarrow [b] \to [(a, b)]$

   $a \to a$

   $(a, b) \to a$

   $(a, a) \to a$

   $[a] \to a$

   a

3. Versuchen Sie allgemein zu erläutern, was Funktionen vom Typ $a \to a$, $[a] \to a$, $[a] \to Int$ „können“, und was nicht. ◇

Aufgabe 6
Stellen Sie sich einen Nachrichtenkanal vor, über den Texte verschickt werden sollen. Eine Zeichenkette muss Zeichen für Zeichen in 8 Bit lange Binärzahlen verwandelt werden, um verschickt zu werden. Am Ende müssen die Zeichen wieder aus dem Bitstream zusammengesetzt werden. Programmieren Sie eine Kodierungs- und eine Dekodierfunktion. (Hinweis: Im Modul *Char* sind die Funktionen $ord :: Char \to Int$ und $chr :: Int \to Chr$ definiert. Das Modul wird eingebunden mit „**import** *Char*“) ◇