Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

8. Februar 2019

Aufgaben zur Klausur **Grundlagen der funktionalen Programmierung** im WS 2018/19 (BInf)

Zeit: 90 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Nutzen Sie die Rückseiten der Klausur zur Entwicklung der Lösungen und übertragen die fertigen Lösungen in das Aufgabenblatt.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 7 Seiten.

Aufgabe 1:

Bei der Listenverarbeitung werden häufig die Funktionen `take` und `drop` zum Selektieren und Löschen von Anfangsstücken einer Liste genutzt.

Die Gegenstücke zu `take` und `drop` sind Funktionen, die am Ende einer Liste selektieren oder löschen, hier als `takeLast` und `dropLast` bezeichnet. Sie sollen analog zu `take` und `drop` arbeiten. Die Funktionen haben folgende Typen:

$takeLast :: Int \to [a] \to [a]$
$dropLast :: Int \to [a] \to [a]$

1. Implementieren Sie `takeLast` nur unter Verwendung von `take`, `drop` und `length`

   .....

   .....

   .....

2. Implementieren Sie `dropLast` nur unter Verwendung von `take`, `drop` und `length`

   .....

   .....

   .....

3. Implementieren Sie `takeLast` nur unter Verwendung von `take`, `drop` und `reverse`

   .....

   .....

   .....

4. Implementieren Sie `dropLast` nur unter Verwendung von `take`, `drop` und `reverse`

   .....

   .....

   .....

5. Welche der beiden Implementierungen von `takeLast` ist Laufzeit-effizienter? Begründen Sie Ihre Antwort.

. . . . . .

. . . . . .

. . . . . .

6. Zum elementweisen Verknüpfen 2-er Listen gibt es die Funktion `zipWith`

$$
\begin{aligned}
&zipWith :: (a \to b \to c) \to [a] \to [b] \to [c] \\
&zipWith\ op\ [\,]\ \_ = [\,] \\
&zipWith\ op\ \_\ [\,] = [\,] \\
&zipWith\ op\ (x : xs)\ (y : ys) = op\ x\ y : zipWith\ op\ xs\ ys
\end{aligned}
$$

Es sei die Funktion `seltsam` wie folgt definiert

$$seltsam\ n\ xs = zipWith\ const\ xs\ (drop\ n\ xs)$$

mit

$$
\begin{aligned}
&const :: a \to b \to c \\
&const\ x\ y = x
\end{aligned}
$$

Welchen Typ besitzt `seltsam`?

. . . . . .

. . . . . .

Welches Resultat liefert der Aufruf `seltsam 3 [1..10]`

. . . . . .

. . . . . .

7. Welche der hier entwickelten Implementierungen für das Löschen am Ende einer Liste ist die effizienteste? Begründen Sie Ihre Antwort.

. . . . . .

. . . . . .

Aufgabe 2:

Gegeben sei der folgende Datentyp für binäre Bäume mit Information an den inneren Knoten. In diesem Datentyp kommt an 2 Stellen Rekursion vor. Dieses hat entscheidenden Einfluss auf die Struktur der Algorithmen.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a \\
&\quad = \mathit{Nil} \\
&\quad \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)
\end{aligned}
$$

Weiter seien die folgenden Konstanten und Funktionen vordefiniert. `insert` wird weiter unten entwickelt.

$$
\begin{aligned}
&\mathit{empty} :: \mathit{Tree}\ a \\
&\mathit{empty} = \mathit{Nil} \\
&\mathit{leaf} :: a \to \mathit{Tree}\ a \\
&\mathit{leaf}\ x = \mathit{Bin}\ \mathit{empty}\ x\ \mathit{empty} \\
&\mathit{null} :: \mathit{Tree}\ a \to \mathit{Bool} \\
&\mathit{null}\ \mathit{Nil} = \mathit{True} \\
&\mathit{null}\ \_ = \mathit{False} \\
&\mathit{toList} :: \mathit{Tree}\ a \to [a] \\
&\mathit{toList}\ \mathit{Nil} = [\,] \\
&\mathit{toList}\ (\mathit{Bin}\ l\ x\ r) = \mathit{toList}\ l +\!\!+ [x] +\!\!+ \mathit{toList}\ r \\
&\mathit{fromList} :: [a] \to \mathit{Tree}\ a \\
&\mathit{fromList}\ xs = \mathit{foldr}\ \mathit{insert}\ \mathit{empty}\ xs
\end{aligned}
$$

Entwickeln Sie für diesen Datentyp eine `map`-Funktion, die wie die `map`-Funktion auf Listen arbeitet (einschließlich Typ).

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie ein Funktion `all`, die testet, ob alle in einem Baum gespeicherten Werte ein bestimmtes Prädikat erfüllen.

$$all :: (a \to Bool) \to Tree\ a \to Bool$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Diese binären Bäume können für die effiziente Suche in einer Menge von Werten eingesetzt werden, wenn man beim Einfügen in einen Baum immer darauf achtet, dass Elemente, die kleiner sind als der Wert in einem `Bin`-Knoten links eingefügt werden, und die größer sind, rechts. Ist der einzufügende Wert gleich dem im Knoten wird der Baum nicht verändert.

Entwickeln Sie die Funktion `insert`, die die oben beschriebene Eigenschaft besitzt.

$$insert :: Ord\ a \Rightarrow a \to Tree\ a \to Tree\ a$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Wenn ein Baum immer mit `insert`-Operationen aufgebaut wird, kann das Suchen sehr effizient geschehen. Man muss nur den zu suchenden Wert mit dem in einem `Bin`-Knoten vergleichen und entweder links oder rechts weiter suchen oder man hat den Wert gefunden. Entwickeln Sie die Suchfunktion `member`

$$member :: Ord\ a \Rightarrow a \rightarrow Tree\ a \rightarrow Bool$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Zum Testen der Konsistenz einer Baumstruktur ist eine Invariante von Nutzen. Mit dieser wird für ALLE Knoten überprüft, ob ALLE Werte in den linken Teilbäumen kleiner und ALLE in den rechten Teilbäumen größer sind als das Element im Knoten selbst. Tipp: ALLE hat nicht nur vom Namen gewisse Ähnlichkeit mit `all`.

$$inv :: Ord\ a \Rightarrow Tree\ a \rightarrow Bool$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Zu dieser Baumstruktur kann, wie bei dem in der Vorlesung besprochenen Baum, eine `fold`-Funktion mit folgendem Typ entwickelt werden.

$$fold :: (b \to a \to b \to b) \to b \to Tree\ a \to b$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Reimplementieren Sie `all` mit dieser `fold`-Funktion.

. . . . . .

. . . . . .

. . . . . .

Reimplementieren Sie die `toList`-Funktion mit `fold`.

. . . . . .

. . . . . .

. . . . . .