Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

16. Februar 2018

Aufgaben zur Klausur **Grundlagen der funktionalen Programmierung** im WS 2017/18 (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 8 Seiten.

Aufgabe 1:

Ganze Zahlen werden in einer Maschine als Bitstrings repräsentiert. Diese Bitrepräsentation kann man auch explizit berechnen, indem man eine Funktion `toBitstring` entwickelt, die eine Liste von Wahrheitswerten als Resultat besitzt.

$$
\begin{aligned}
&\mathit{toBitstring} :: \mathit{Int} \to [\mathit{Bool}] \\
&\mathit{toBitstring}\ 0 = [\,] \\
&\mathit{toBitstring}\ i = (i\ `\mathit{mod}`\ 2\ \neq\ 0) : \mathit{toBitstring}\ (i\ `\mathit{div}`\ 2)
\end{aligned}
$$

Zum Beispiel liefert `toBitstring 13` das Resultat `[True,False,True,True]`.

Entwickeln sie eine Funktion `fromBitstring`, die die Umkehrfunktion zu `toBitstring` ist. Es muss also das folgende Gesetz gelten:

$$\mathit{fromBitstring} \circ \mathit{toBitstring} = \mathit{id}$$

In `fromBitstring` kann die folgende Hilfsfunktion genutzt werden:

$$\mathit{fromEnum} :: \mathit{Enum}\ a \Rightarrow a \to \mathit{Int}$$

Die Funktion `fromBitstring` als rekursive Funktion:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Diese Funktion besitzt eine häufig wiederkehrende Struktur, so dass sie mit einer `fold`-Funktion implementiert werden kann. Redefinieren sie `fromBitstring` so, dass sie mit einem `fold` arbeitet:

. . . . . .

. . . . . .

. . . . . .

Aufgabe 2:

Die `map`-Funktion dient zur elementweisen Verarbeitung von Listen. Sie besitzt folgenden Typ

$$map :: (a \to b) \to [a] \to [b]$$

Implementieren Sie `map` mit Hilfe einer rekursiven Funktion

. . . . . .

. . . . . .

. . . . . .

Implementieren Sie `map` mit Hilfe von `foldr`

. . . . . .

. . . . . .

. . . . . .

Aufgabe 3:

Gegeben sei die folgende Datenstruktur für einen binären Baum. Dieses ist eine Erweiterung der Baumstruktur aus der Vorlesung. Der `Fork`-Konstruktor besitzt eine 3. Komponente vom Typ `Int`. In dieser wird gespeichert, wie viele Elemente der Baum enthält.

Invariante: Wie in der Vorlesung auch, soll hier mit Bäumen gearbeitet werden, die `Nil` nie als Kind eines `Fork`-Knotens enthalten.

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

Weiter seien die folgenden Funktionen vorgegeben:

$$
\begin{aligned}
&\mathit{size} :: \mathit{Tree}\ a \to \mathit{Int}\\
&\mathit{size}\ \mathit{Nil} \quad\ = 0\\
&\mathit{size}\ (\mathit{Leaf}\ x) = 1\\
&\mathit{size}\ (\mathit{Fork}\ n\ \mathit{t1}\ \mathit{t2}) = n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{conc} :: \mathit{Tree}\ a \to \mathit{Tree}\ a \to \mathit{Tree}\ a\\
&\mathit{conc}\ \mathit{Nil}\ \mathit{t2} = \mathit{t2}\\
&\mathit{conc}\ \mathit{t1}\ \mathit{Nil} = \mathit{t1}\\
&\mathit{conc}\ \mathit{t1}\ \mathit{t2} = \mathit{Fork}\ (\mathit{size}\ \mathit{t1} + \mathit{size}\ \mathit{t2})\ \mathit{t1}\ \mathit{t2}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fromList} :: [a] \to \mathit{Tree}\ a\\
&\mathit{fromList}\ [\,] = \mathit{Nil}\\
&\mathit{fromList}\ [x] = \mathit{Leaf}\ x\\
&\mathit{fromList}\ \mathit{xs} = \mathit{fromList}\ \mathit{ys}\ `\mathit{conc}`\ \mathit{fromList}\ \mathit{zs}\\
&\quad \textbf{where}\\
&\qquad (\mathit{ys}, \mathit{zs}) = \mathit{splitAt}\ (\mathit{length}\ \mathit{xs}\ `\mathit{div}`\ 2)\ \mathit{xs}
\end{aligned}
$$

`size` berechnet die Anzahl Elemente im Baum, `conc` konkateniert 2 Bäume so, dass die Invariante eingehalten wird und dass im `Fork`-Knoten die Anzahl Elemente im Baum gespeichert wird, `fromList` konvertiert eine Liste in einen Baum.

Entwickeln Sie als erstes die Funktion `toList`, die die inverse Funktion zu `fromList` sein soll, es muss also das Gesetz gelten

$$toList \circ fromList = id$$

`toList`:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie die Funktionen `cons` und `snoc`, die ein Element vorne bzw. hinten im Baum einfügen. Achten Sie darauf, dass beide Funktionen in konstanter Zeit laufen.

$$cons :: a \to Tree\ a \to Tree\ a$$

. . . . . .

. . . . . .

$$snoc :: Tree\ a \to a \to Tree\ a$$

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `safeHead`, die das erste Element eines Baumes berechnet. `safeHead` ist eine totale Funktion.

$$safeHead :: Tree\ a \to Maybe\ a$$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Für `safeHead` und `cons` muss das folgende Gesetz gelten

$$safeHead\ (cons\ x\ t) = Just\ x$$

Der indizierte Zugriff auf Elemente einer Liste ist sehr ineffizient. Die Zeit hängt linar ab von der Größe des Index. Man erkennt dieses an der folgenden Beispiel-Implementierung (`drop(i-1)` in `atList`).

$$\begin{aligned}
&atList :: [a] \to Int \to Maybe\ a \\
&atList\ xs\ i \\
&\quad \mid 0 \leqslant i\ \&\&\ i < length\ xs = Just\ (head\ (drop\ (i-1)\ xs)) \\
&\quad \mid otherwise = Nothing
\end{aligned}$$

Für indizierte Zugriffe gilt die Konvention, dass das erste Element einer Liste oder eines Baumes den Index 0 besitzt.

Auf Bäumen kann man ebenfalls einen indizierten Zugriff implementieren (`atTree`), der dann aber noch ineffizienter wird, als der Zugiff auf die Listen.

$$\begin{aligned}
&atTree :: Tree\ a \to Int \to Maybe\ a \\
&atTree\ t\ i = atList\ (toList\ t)\ i
\end{aligned}$$

Die Speicherung der Anzahl der Elemente in den `Fork`-Knoten ermöglicht es, die `size`-Funktion in konstanter Zeit zu implementieren. Zusätzlich kann diese Speicherung auch Gewinn bringend für einen effizienteren indizierten Zugriff genutzt werden. Möchte man an einem `Fork`-Knoten auf das i-te Element zugreifen, so kann man an Hand der Anzahl der Elemente im linken Teilbaum entscheiden, ob der Index in den linken oder den rechten Teilbaum zeigt.

Implementieren Sie die Funktion `at`, so dass diese immer das gleiche Resultat liefert wie `atTree`, nur viel effizienter arbeitet.

$$at :: Tree\ a \to Int \to Maybe\ a$$

Entwickeln Sie eine `map`-Funktion für `Tree`, die analog zur `map`-Funktion für Listen arbeitet.

Wie bei Listen kann man auch auf Bäumen eine `filter`-Funktion implementieren:

$$\mathit{filter} :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Tree}\ a$$