Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
L. Noschinski, L. Hupel, Dr. J. Blanchette, M. Eberl

WS 2014/15
07.11.2014
Abgabe: 14.11.2014

Einführung in die Informatik 2
5. Übung

Aufgabe G5.1 Mysteriöse Funktion

Beschreiben Sie in eigenen Worten das Verhalten des folgenden Ausdrucks:

```
head (tail (map gg (zip [xx, xx] [yy, yy])))
```

mit `gg`, `xx` und `yy` beliebig. Die Funktion `zip :: [a] -> [b] -> [(a, b)]` berechnet die Liste von Paaren der Elemente zweier paralleler Eingabelisten. Für `zip` gilt folgende Gleichung:

$$\texttt{zip}\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] = [(x_1,\ y_1), \ldots, (x_n,\ y_n)]$$

Aufgabe G5.2 Iteration von Funktionsaufrufen

1. Schreiben Sie eine Funktion `iter :: Int -> (a -> a) -> a -> a`. Diese soll eine Zahl n, eine Funktion f und einen Wert x als Eingabe bekommen und f n-mal auf x anwenden. `iter` n f x berechnet dann $f^n(x)$. Negative n sollen wie $n = 0$ behandelt werden.

   Beispiel: `iter 3 sq 2 == 256`, wobei `sq x = x * x`

2. Nutzen Sie `iter`, um die folgenden Funktionen ohne Rekursion zu implementieren:

   a) Die Exponentialfunktion `pow :: Int -> Int -> Int` mit `pow` n $k = n^k$ (für alle $k \geq 0$).

   b) Die Funktion `drop :: Int -> [a] -> [a]` aus der Standardbibliothek nimmt eine Zahl k und eine Liste $[x_1, \ldots, x_n]$ und gibt die Liste $[x_{k+1}, \ldots, x_n]$ zurück.

   Implementieren Sie eine eigene Version dieser Funktion. Sie dürfen annehmen, dass $k \leq n$ gilt.

   c) Die Funktion `replicate :: Int -> a -> [a]` aus der Standardbibliothek nimmt eine Zahl $n \geq 0$ und einen Wert x und gibt die Liste $[\underbrace{x, \ldots, x}_{n\text{-mal}}]$ zurück.

   Implementieren Sie eine eigene Version dieser Funktion.

Aufgabe G5.3 Partitionierung

Die Funktion `partition :: (a -> Bool) -> [a] -> ([a], [a])` aus der Bibliothek `Data.List` teilt eine Liste in zwei Teillisten. Die eine Teilliste enthält die Elemente der Eingabeliste, die das gegebene Prädikat erfüllen. Die zweite Teilliste enthält die übrigen Elemente. Beide Teillisten sollen die Elemente in der gleichen Reihenfolge wie in der Eingabeliste enthalten.

Die folgenden Beispiele sind alle `True` (`xs` ist eine beliebige Liste):

```
partition alwaysTrue xs == (xs, [])
partition alwaysFalse xs == ([], xs)
partition even [4, 4, 1, 2] == ([4, 4, 2], [1])
```

wobei `alwaysTrue` und `alwaysFalse` – wie die Namen schon sagen – Funktionen sind, die stets `True` bzw. `False` liefern.

1. Implementieren Sie `partition` rekursiv.
2. Implementieren Sie `partition` mithilfe von `filter`.
3. Prüfen Sie mit QuickCheck, ob Ihre Funktionen mit `Data.List.partition` übereinstimmen.
4. Wie könnte ein sinnvoller QuickCheck-Test aussehen, der Aussagen über das Verhalten von Partition auf einer Liste xs ++ ys beschreibt?
5. Erörtern Sie die Vor- und Nachteile der zwei Implementierungen.

Aufgabe G5.4 Bewiesen. Oder nicht?

Gegeben seien die folgenden Definitionen:

```
zeros :: [Int]
zeros = 0 : zeros

length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```

Aus diesen Gleichungen folgt

```
                      length zeros
(by def zeros)  .=. length (0 : zeros)
(by def length) .=. 1 + length zeros
```

Nimmt man von beiden Seiten `length zeros` weg, bekommt man `0 = 1`, einen Widerspruch. Erklären Sie dies.

Aufgabe H5.1 Repetitiv

Implementieren Sie eine Funktion

```
while :: (a -> Bool) -> (a -> a) -> a -> a
```

die für `while p f x` die Funktion `f` so lange wiederholt auf den Anfangswert x anwendet, bis die Eigenschaft p nicht mehr erfüllt ist.

Beispiel:

```
while (<10) (+1) 0 == 10
while even half 12 == 3
    where half x = x `div` 2
```

Benutzen Sie anschließend Ihre `while`-Funktion, um folgende Funktionen zu implementieren:

1. `removeFactor k n` teilt n so lange durch k, bis k kein Teiler des Ergebnisses mehr ist. (d.h. `n == k^x * removeFactor k n` und `(removeFactor k n) `mod` k /= 0`)
2. `ld n` berechnet $\lfloor \log_2 n \rfloor$ für ein positives n, d.h. die größte nichtnegative ganze Zahl k, sodass $2^k \leq n$ gilt. Der Zustand ist hierbei ein Paar `(n',k)`, wobei welches initial `(n,0)` ist und in dem am Ende der Berechnung das Ergebnis in der zweiten Komponente steht.

Sie müssen dabei nur die Schleifenbedingung p und die Transformationsfunktion f angeben (siehe Template).

Aufgabe H5.2 Erreichbar unter ...

Ein Graph lässt sich in Haskell ganz einfach als Liste von Paaren `[(a, a)]` darstellen. Jedes dieser Paare repräsentiert eine (gerichtete) Kante im Graphen zwischen den angegebenen Knoten.

In dieser Aufgabe sei ein Graph g gegeben, sowie ein Startpunkt s_1 und ein Endpunkt s_2. Gesucht ist eine Funktion `reachable :: Eq a => [(a, a)] -> a -> a -> Bool`, die ermittelt, ob der Punkt s_2 von s_1 aus erreichbar ist. Zwischen s_1 und s_2 dürfen beliebig viele Schritte gemacht werden. Formal ausgedrückt: Existiert eine Sequenz $k_1, ..., k_n$ so dass

1. $n \geq 2$ und
2. $k_1 = s_1$ und
3. $k_n = s_2$ und
4. für alle $1 \leq i < n$: $(k_i, k_{i+1}) \in g$,

dann gilt s_2 als erreichbar von s_1.

Beispiele:

```
reachable [(1, 2), (2, 3), (2, 4)] 1 4 == True
reachable [(1, 2), (2, 3), (2, 4)] 3 4 == False
reachable [(1, 2), (2, 1), (2, 3)] 1 3 == True
reachable [(1, 2), (2, 1), (2, 2)] 1 3 == False

reachable [] 1 1 == False
reachable [(1, 1)] 1 1 == True
```

Beachten Sie insbesondere, dass der übergebene Graph auch Zyklen haben darf. Vergessen Sie daher nicht, eine entsprechende Abbruchbedingung einzubauen. Sie dürfen nicht davon ausgehen, dass die übergebene Liste keine Duplikate enthält.

Hinweis: Hierfür kann die Funktion `\\` aus `Data.List` hilfreich sein.

Aufgabe H5.3 Durchschleifen

Sie haben bereits die Funktion `map` kennengelernt, die eine Funktion auf jedes Element einer Liste anwendet. Manchmal ist es aber auch hilfreich, Wissen über den bereits verarbeiteten Teil

der Liste verwenden zu können.

Die Funktion

```
mapState :: (s -> a -> (b,s)) -> s -> [a] -> ([b], s)
```

nimmt eine Funktion f, einen Startzustand s und eine Liste xs als Parameter. Die Funktion f nimmt einen Zustand und ein Listenelement und gibt das neue Listelement und den nächsten Zustand zurück. Ähnlich wie bei `map` wird die Funktion f auf jedes Listenelement angewendet, wobei der Zustand immer durchgereicht wird.

- Implementieren Sie die Funktion `mapState`

  Beipiele:

  ```
  inc s x = ((s, x), s + 1) -- Hilfsfunktion

  mapState inc 0 [] == ([], 0)
  mapState inc 0 ['a']
      == ([(0, 'a')], 1)
  mapState inc 0 "abc"
      == ([(0, 'a'), (1, 'b'), (2, 'c')], 3)
  ```

- Geben Sie eine Funktion `f :: String -> Char -> (Char, String)` an, so dass der Aufruf `mapState f [] xs` jeweils das erste Vorkommen eines Zeichens in xs in einen Großbuchstaben umwandelt:

  Beispiel:

  ```
  fst (mapState f [] "abrakadabra") == "ABRaKaDabra"
  fst (mapState f [] "ottos mops kotzt") == "OTtoS MoPs KotZt"
  ```

  Verwenden Sie die Funktion `toUpper` aus `Data.Char` um ein Zeichen in Großbuchstaben umzuwandeln.

Aufgabe H5.4 Korrekturaufteilung (Wettbewerbsaufgabe)

Wie Sie wissen, sind die MCs (Sen. und Jun.) sehr beschäftigt. Sie haben sich jetzt dazu entschieden, die Korrektur der Wettbewerbsabgaben unter sich aufzuteilen. Es gibt n Abgaben mit verschiedenen Längen (in Token gemessen). Damit die Aufteilung der zu korrigierenden Abgaben gerecht ist, soll sie so gewählt sein, dass die Summe der Längen der von jedem MC zu korrigierenden Aufgaben in etwa gleich ist.

Schreiben Sie hierfür eine Funktion

```
splitSubmissions ::
    [(a, Integer)] -> ([(a, Integer)], [(a, Integer)])
```

Die Funktion erhält als Parameter `xs` die Liste der Einreichungen. Jede Einreichung ist ein Paar `(s,n)`, wobei `s` die Einreichung selbst ist (die Sie nicht interessieren muss) und `n` die Länge der

Einreichung in Tokens. Die Rückgabe der Funktion sollen zwei Listen `ys` und `zs` von Submissions sein, sodass:

- jede Einreichung aus `xs` entweder in `ys` oder in `zs` ist
- jede Einreichung aus `ys` und `zs` auch in `xs` ist
- keine Einreichung sowohl in `ys` als auch in `zs` ist
- die Differenz der Tokenzahlsummen von `ys` und `zs` möglichst klein ist

Um den letzten Punkt noch zu präzisieren: Für

```
tokenSum ys = sum [n | (_, n) <- ys]
badness (ys,zs) = abs (tokenSum ys - tokenSum zs)
```

soll `badness (splitSubmissions xs)` möglichst klein sein.

Falls Sie nicht am Wettbewerb teilnehmen möchten, reicht es, wenn `badness (ys,zs)` kleiner oder gleich der Tokenzahl der längsten Einreichung in `xs` ist. Dies lässt sich z.B. mit einem einfachen Greedy-Algorithmus erreichen, der die Liste der Einreichungen durchläuft und eine neue Einreichung immer dem MC mit der geringeren Gesamttokenzahl zuweist.

Für den Wettbewerb werden die Lösungen nach folgenden Kriterien in absteigender Priorität bewertet:

1. Gerechtigkeit der Aufteilung, d.h. `badness (splitSubmissions xs)` sollte so klein wie möglich sein
2. Effizienz (benötigte Zeit/benötigter Speicher)
3. Eleganz des Codes

Bei nicht offensichtlichen Algorithmen sind Kommentare für den MC hilfreich.

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken. Wenn Sie nicht am Wettbewerb teilnehmen, sondern die Aufgabe allein im Rahmen der Hausaufgabe abgeben möchten, lassen Sie bitte die `{-WETT-}` ... `{-TTEW-}` Kommentare weg. Bei der Bewertung Ihrer Hausaufgabe entsteht Ihnen hierdurch kein Nachteil.