Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Dr. Jasmin Blanchette, Dmitriy Traytel

WS 2012/13
13.11.2012
Abgabe: 21.11.2012, 15:30

Einführung in die Informatik 2
5. Übung

Wegen Wartungsarbeiten an der Lehrstuhlinfrastruktur ist der Übungserver am Dienstag, den 20.11.2012 planmäßig von 6:00 bis 14:00 nicht erreichbar. Der Abgabetermin wird daher ausnahmsweise auf *Mittwoch, den 21.11.2012 um 15:30* verschoben.

Aufgabe G5.1 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 4 (+ 2) 1 == 9`.

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 G5.2 Partitionen und zweistellige `map`-Funktion

1. 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 (\x -> True) xs == (xs, [])
   partition (\x -> False) xs == ([], xs)
   partition even [4, 4, 1, 2] == ([4, 4, 2], [1])
   ```

   a) Implementieren Sie `partition` rekursiv.

b) Implementieren Sie `partition` mithilfe von `foldr`.

c) Implementieren Sie `partition` mithilfe von `filter`.

d) Prüfen Sie mit QuickCheck, ob Ihre Funktionen mit `Data.List.partition` übereinstimmen.

e) Wie könnte ein sinnvoller QuickCheck-Test aussehen, der Aussagen über das Verhalten von Partition auf einer Liste xs ++ ys beschreibt?

f) Erörtern Sie die Vor- und Nachteile der drei Implementierungen.

2. Die Funktion `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` aus `Prelude` wendet eine zweistellige Funktion auf die Elemente aus zwei parallelen Listen an und sammelt die Ergebnisse in einer Liste. Die Funktion ist charakterisiert durch die Gleichung

$$\texttt{zipWith}\ f\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] = [f\ x_1\ y_1, \ldots, f\ x_n\ y_n]$$

Beispiele:

```
zipWith (*) [10, 22] [5, 3] == [50, 66]
zipWith (++) ["Hey", "Was", "Tja"] ["!", "??", "."] ==
    ["Hey!", "Was??", "Tja."]
```

Implementieren Sie eine eigene Version von `zipWith`. Sie dürfen annehmen, dass die zwei Eingabelisten gleich lang sind.

Aufgabe G5.3 Mysteriöse Funktionen

Beschreiben Sie in eigenen Worten das Verhalten folgender λ-Ausdrücke.

```
\xs -> foldr (++) [] (map (\x -> [x]) xs)
\gg xx yy -> head (tail (zipWith gg [xx, xx] [yy, yy]))
```

(Die Funktion `zipWith` ist in G5.2 spezifiziert.)

Aufgabe G5.4 Bewiesen. Oder nicht?

1. Gegeben seien die folgenden Definitionen:

```
zeros :: [Int]
zeros = 0 : zeros                          (zeros_def)

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

Aus diesen Gleichungen folgt

```
length zeroes
  = length (0 : zeros)                    by zeros_def
  = 1 + length zeros                      by length_Cons
```

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

2. Gegeben sei die Funktion

```
h :: Integer -> Integer -> Integer
h m n | m == n = 0                    (h_eq)
      | m < n = h m (n - 1)           (h_lt)
      | m >= n = h n m + 1            (h_geq)
```

Da $0 \geq 0$ gilt, wenden wir die Gleichung `h_geq` auf `h 0 0` an und „beweisen“ wieder einen Widerspruch:

```
h 0 0 = h 0 0 + 1                     by h_geq
```

a) Erklären Sie den Fehler.

b) Werten Sie `h 0 0` aus.

c) Welche Aussage können Sie über die Terminierung von `h` treffen?

Wichtig: Bei bisherigen Hausaufgaben gab es immer wieder eingereichte Lösungen die zwar inkorrekt waren aber dennoch alle unsere Tests bestanden haben. Das ist nichts ungewöhnliches, da die Tests die Korrektheit nicht garantieren. Allerdings sind wir daran interessiert unsere Tests zu verbessern, indem eventuelle Lücken geschlossen werden.

Dabei zählen wir auf Ihre Unterstützung, die wir gerne mit zusätzlichen Hausaufgabenpunkten entlohnen werden. Falls Sie also eine inkorrekte Implementierung haben, die unsere Tests besteht, können Sie zusätzliche Tests in Ihren hochgeladenen Lösungsvorschlag einschließen. Benutzen Sie bitte die Tags `{-QC-}` ... `{-CQ-}` um die neuen Tests zu kennzeichnen.

Wir werden die sinnvollen Tests in unsere Testsuite einpflegen. Wenn Ihre Tests Fehler unter den eingereichten Lösungen offenbaren, die wir nicht gefunden haben, bekommen Sie einen zusätzlichen Punkt gutgeschrieben (es gilt First-Come-First-Served).

Aufgabe H5.1 Fixpunktiteration (7 Punkte)

1. Schreiben Sie eine Funktion `fixpoint :: (a -> a -> Bool) -> (a -> a) -> a -> a`, die als Parameter einen Gleichheitstest *eq*, eine Funktion f und einen Wert x bekommt und f solange wiederholt auf x anwendet, bis sich der Wert nicht mehr verändert (bezüglich des Gleichheitstests *eq*).

   Falls dieser Prozess terminiert, gibt es also eine nicht-negative Zahl n, so dass zum einen `fixpoint` *eq* f $x = f^n(x) =$ `iter` n f x und zum anderen $f^n(x)$ `` `eq` `` $f^{n+1}(x)$ gilt.

Beispiele:

```
fixpoint (==) (delete 'a') "Halli-Hallo" == "Hlli-Hllo"
fixpoint (==) (delete 'l') "Halli-Hallo" == "Hai-Hao"
fixpoint (\xs ys -> length xs == length ys)
  (\xs -> reverse (delete 'l' xs)) "Halli-Hallo" == "Hai-Hao"
```

Wenn im letzten Beispiel `==` statt `\xs ys -> length xs == length ys` verwendet wird, dann terminiert die Berechnung nicht.

2. Welche Beträge echt kleiner als 100 Cents kann man mit ausschließlich 5-Cent- und (hypothetischen) 7-Cent-Münzen bezahlen (ohne Wechselgeld)?

   Schreiben Sie eine Funktion `cents :: [Integer] -> [Integer]`, so dass die Evaluierung von `fixpoint (==) cents [5,7]` diese Frage beantwortet.

3. In Aufgabe G2.5 haben wir n-schrittige Erreichbarkeit in gerichteten Graphen berechnet. Nun sind wir an der allgemeinen Erreichbarkeit interessiert, d.h. über beliebig viele Kanten (mindestens eine)—das ist die sogenannte *transitive Hülle*.

   Schreiben Sie eine Funktion `trancl :: [(Integer, Integer)] -> [(Integer, Integer)]`, die für eine Liste von Kanten xs die duplikatfreie Liste der Paare (u, v) berechnet, sodass v von u aus über Kanten (mindestens eine) in G erreichbar ist. Das heißt, (u, v) ist genau dann in `trancl` xs enthalten, wenn es eine Folge von Kanten

   $$(u, w_1), (w_1, w_2), \ldots, (w_{n-1}, w_n), (w_n, v)$$

   gibt, so dass jede dieser Kanten in xs liegt.

   Die einzigen Funktionen höherer Ordnung, die Sie hierfür benutzen dürfen, sind `map`, `filter` und `fixpoint`.

   Beispiele:

```
sort (trancl [(1,2),(2,3),(3,4)]) ==
     [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
sort (trancl [(1,2),(2,3),(2,5),(4,5),(5,4)]) ==
     [(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),
      (4,4),(4,5),(5,4),(5,5)]
```

Aufgabe H5.2 Deterministische Automaten (9 Punkte)

Ein *deterministischer Automat* dient dazu, eine *Sprache*, d.h. eine Menge von Zeichenketten, zu beschreiben. Ein solcher Automat besteht aus einem *Startzustand* `start`, einer *Zustandsübergangsfunktion* `delta` und einer Funktion `final` um zu bestimmen, welcher Zustand ein *Endzustand* ist. Die Zustandsübergangsfunktion beschreibt die Änderung des Zustands beim Lesen eines Buchstaben. Wir verwenden folgende Typen um einen Automaten als Tripel `(start, delta, final)` darzustellen:

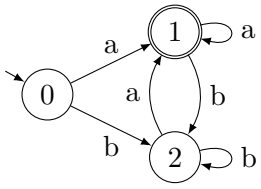

Abbildung 1: Automat `a = (0, delta, \x -> x == 1)` mit

```
delta 0 'a' = 1    delta 0 'b' = 2
delta 1 'a' = 1    delta 1 'b' = 2
delta 2 'a' = 1    delta 2 'b' = 2
```

```
type State = Integer
type DA = (State, State -> Char -> State, State -> Bool)
```

Anschaulicher kann man deterministische Automaten graphisch darstellen (Abb. 1). Die Zustände werden als Kreise dargestellt. Die Zustandsübergangsfunktion entspricht den beschrifteten Kanten. Der Startzustand wird durch einen eingehenden Pfeil und Endzustände durch eine doppelte Umrandung gekennzeichnet.

Hinweis: Da Haskell Funktionen nicht ausgeben kann, verwenden wir für die Darstellung der Gegenbeispiele in den QuickCheck-Tests den Typ

```
type ListDA = (State, [((State, Char), State)], [State])
```

wobei Zustandsübergangsfunktion und Endzustände durch Listen statt Funktionen dargestellt werden. Der Beispielautomat aus Abb. 1 sähe in dieser Darstellung wie folgt aus:

```
listA = (0, [((0,'a'),1), ((1,'a'),1), ((2,'a'),1),
             ((0,'b'),2), ((1,'b'),2), ((2,'b'),2)], [1])
```

Für Zustandsübergänge, die nicht in der Liste vorkommen, gibt die erzeugte Übergangsfunktion 0 zurück. Zur Umwandlung in den Typ `DA` nutzen Sie die Hilfsfunktion `toDA :: ListDA -> DA` aus der Vorlage.

1. Wenn ein Automat A ein Zeichen c liest, wechselt er von seinem aktuellen Zustand s in einen neuen Zustand `delta` c s, wobei `delta` seine Zustandsübergangsfunktion ist.

   Ein Automat liest eine Zeichenkette buchstabenweise von links nach rechts und wechselt wie beschrieben mit jedem gelesenen Buchstaben seinen Zustand.

   Schreiben Sie eine Funktion `advance :: DA -> State -> String -> State`, die einen Automaten A, einen Zustand s und eine Zeichenkette xs als Eingabe bekommt und den Zustand von A nach Lesen der Zeichenkette xs ausgibt. Am Anfang befinde sich der Automat im Zustand s.

   ```
   advance a 0 "" == 0
   advance a 0 "a" == 1
   advance a 0 "aa" == 1
   advance a 0 "aab" == 2
   ```

```
advance a 0 "aababababababa" == 1
```

2. Schreiben Sie drei QuickCheck-Tests

   ```
   prop_advance_empty :: DA -> State -> Bool
   prop_advance_single :: DA -> State -> Char -> Bool
   prop_advance_concat :: DA -> State -> String -> String -> Bool
   ```

   die das Verhalten von `advance` vollständig beschreiben. Betrachten Sie das Verhalten auf leeren Zeichenketten, Zeichenketten der Länge 1 und Konkatenationen von Zeichenketten.

3. Ein Automat akzeptiert eine Zeichenkette xs, wenn ihn das Lesen von xs vom Start in einen Endzustand bringt. Schreiben Sie eine Funktion `accept :: DA -> String -> Bool`, die testet, ob ein eine Zeichenkette akzeptiert. Beispiele:

   ```
   not (accept a "")
   accept a "a"
   accept a"aa"
   not (accept a"aab")
   accept a "aababababababa"
   ```

4. Schreiben sie eine Funktion `reachableStates :: DA -> State -> [Char] -> [State]`, die einen Automaten, einen Zustand s und ein Alphabet als Eingabe bekommt und eine Liste der Zustände zurück gibt, die von s aus mit den Buchstaben des Alphabetes gebildeten Zeichenketten erreichbar sind. Beispiele:

   ```
   reachableStates a 0 "ab" = [0,1,2]
   reachableStates a 0 "a" = [0,1]
   reachableStates a 0 "b" = [0,2]
   reachableStates a 1 "ab" = [1,2]
   reachableStates a 1 "a" = [1]
   reachableStates a 1 "b" = [1,2]
   ```

Aufgabe H5.3 Quasi-Teilsequenzen (4 Punkte, Wettbewerbsaufgabe)

Die Liste $[x_1, \ldots, x_n]$ ist eine *Teilsequenz* der Liste ys, wenn ys sich in die Form $w_0 \mathbin{+\!\!+} [x_1] \mathbin{+\!\!+} w_1 \mathbin{+\!\!+} [x_2] \mathbin{+\!\!+} w_2 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} w_{n-1} \mathbin{+\!\!+} [x_n] \mathbin{+\!\!+} w_n$ für gewisse $w_0, \ldots, w_n$ bringen lässt. Das heißt, alle Elemente in $[x_1, \ldots, x_n]$ müssen in ys vorkommen und das in der gleichen Reihenfolge, wobei ys zusätzlich andere Elemente enthalten kann (vgl. Aufgabe H3.4). Die Liste $[x_1, \ldots, x_n]$ ist eine *Quasi-Teilsequenz* der Liste ys, wenn $[x_1, \ldots, x_n]$ eine Teilsequenz von ys ist oder wenn $[x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n]$ für irgendein k eine Teilsequenz von ys ist.

Gesucht ist eine Funktion mit der Signatur `quasiSubseq :: Eq a => [a] -> [a] -> Bool`, die `True` zurückgibt, wenn das erste Argument eine Quasi-Teilsequenz des zweiten Arguments ist und ansonsten `False`. Für die Implementierung sind allein die Bibliothekfunktionen aus `Prelude` erlaubt.

Beispiele, die `True` sind:

```
quasiSubseq "12" "a1b2c"
quasiSubseq "012" "a1b2c"
quasiSubseq "102" "a1b2c"
quasiSubseq "120" "a1b2c"
quasiSubseq "abracadabra" "abrakadabra"
quasiSubseq "xxxxxxxxxxxxxxxxxxxYx" "xxxxxxxxxxxxxxxxxxxx"
quasiSubseq "a" ""
quasiSubseq "" ""
not (quasiSubseq "0012" "a1b2c")
not (quasiSubseq "baa" "abbc")
not (quasiSubseq [1 .. 5000] ([5000] ++ [2 .. 4999] ++ [1]))
```

Für den Wettbewerb zählt die Effizienz, vor allem die Skalierbarkeit. Die Lösung des Master of Competition benötigt 0,4 Sekunden für das Beispiel

```
not (quasiSubseq [1 .. 50000] ([50000] ++ [2 .. 49999] ++ [1]))
```

Versuchen Sie, ihn zu schlagen! GHCi zeigt Ihnen die benötigte Zeit zum Auswerten eines Ausdrucks an, wenn Sie vorher einmal den Befehl `:set +s` ausführen. Lösungen, die sich effizienztechnisch ähnlich verhalten, werden bzgl. ihrer Lesbarkeit verglichen, wobei Formatierung und Namensgebung besonders gewichtet werden. Um teilzunehmen müssen Sie Ihre vollständige Lösung innerhalb der Kommentare {`-WETT-`} und {`-TTEW-`} eingeben.

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.