Dr. Steffen Jost
Leah Neukirchen

Ludwig-Maximilians-Universität München
Institut für Informatik
26. April 2018

Lösungsvorschlag zur 03. Übung zur Vorlesung

Programmierung und Modellierung

Hinweis: Die Vorlesung am Montag, den 30. April 2018, entfällt planmäßig. Am Dienstag, den 1. Mai 2018, entfallen aufgrund des Feiertags alle Übungen ersatzlos.

A3-1 ***Endrekursive Fakultät*** Definieren Sie Fakultätsfunktion von Folie 3.3 erneut in einer endrekursiven Version durch Einführung eines Akkumulators!

```
fakultät 0 = 1
fakultät n = n * fakultät (n-1)
```

LÖSUNGSVORSCHLAG:
Wir führen einen Akkumulator für das Ergebnis ein:

```
fakultät' :: Integer -> Integer
fakultät' x = facc 1 x
  where facc :: Integer -> Integer -> Integer
        facc acc 0 = acc
        facc acc y = facc (acc*y) (y-1)
```

Die Definition von `facc` ist hier der essentielle Teil der Lösung. Ob man diese in einer `where`-Klausel versteckt und einen Aufruf zur Initialisierung des Akkumulators hinzufügt, ist dagegen nicht so wichtig.

A3-2 ***Quicksort*** Das Quicksort-Verfahren ist ein rekursiver Sortier-Algorithmus:

Schritt 1: Ist die Liste leer, so ist diese fertig sortiert. Ansonsten wähle man irgendein Element der sortierenden Liste, z.B. das erste Element.

Schritt 2: Teile die Restliste in zwei Teillisten auf: eine Teilliste enthalte alle Elemente, welche kleiner oder gleich sind als das gewählte Element; die andere alle größeren Elemente.

Schritt 3: Sortiere beide Teillisten durch rekursive Verwendung des Quicksort-Verfahrens.

Schritt 4: Füge die beiden sortierten Teillisten mit `(++) :: [a]->[a]->[a]` wieder zusammen, wobei das anfangs ausgewählte Element in die Mitte dazwischen gesteckt wird.

Implementieren Sie diesen Algorithmus in Haskell, indem Sie folgende Funktionen definieren:

```
quicksort :: [Int] -> [Int]
splitBy   :: Int -> [Int] -> ([Int],[Int])
```

Die Funktion `quicksort` sortiert eine Liste. Die Funktion `splitBy` teilt eine Liste von Zahlen wie angegeben in zwei Teillisten auf. *Beispiele:*

```
> splitBy 6 [3,9,4,2,6,8,5,1,7,2,2]
([3,4,2,6,5,1,2,2],[9,8,7])

> quicksort [3,9,4,2,6,8,5,1,7,2,2]
[1,2,2,2,3,4,5,6,7,8,9]
```

a) Implementieren Sie die Funktion `splitBy` und testen Sie diese anschließend.

Wem nicht einfällt, wie man hier beginnen könnte, kann einfach zwei List-Comprehensions verwenden. Das funktioniert, ist aber nicht besonders effizient, da die Eingabeliste dabei zwei Mal durchlaufen werden muss.

Schöner ist es, wenn die Eingabeliste nur einmal durchlaufen wird, z.B. durch eine Rekursion über die Eingabeliste mit anschließendem Anfügen des aktuellen Elementes in die richtige Ergebnisliste. Da die Reihenfolge der Elemente in den beiden Ergebnislisten nicht spezifiziert wurde, läßt sich dann auch schnell eine endrekursive Version schreiben.

LÖSUNGSVORSCHLAG:

Hier sind drei vollständig unabhängige Lösungsmöglichkeiten:

```
-- Wir gehen zweimal über die Eingabeliste mit List-Comprehension:
splitBy0 :: Int -> [Int] -> ([Int],[Int])
splitBy0 n xs = ( [x | x<-xs, x<=n] , [ x | x<-xs, x>n] )

-- Wir gehen einmal rekursiv über die Eingabeliste:
splitBy1 :: Int -> [Int] -> ([Int],[Int])
splitBy1 _ []    = ([],[])
splitBy1 p (h:t) | h <= p    = (h:smaller,   bigger)
                 | otherwise = (  smaller, h:bigger)
  where
    (smaller,bigger) = splitBy1 p t

-- Wir gehen einmal endrekursiv über die Eingabeliste:
splitBy2 :: Int -> [Int] -> ([Int],[Int])
splitBy2 pivot = splitBy_Aux ([],[])
  where
    splitBy_Aux :: ([Int],[Int]) -> [Int] -> ([Int],[Int])
    splitBy_Aux acc     []    = acc
    splitBy_Aux (sl,bg) (h:t)
      | h <= pivot = splitBy_Aux (h:sl,   bg) t
      | otherwise  = splitBy_Aux (  sl, h:bg) t
```

b) Implementieren Sie nun die Funktion `quicksort` gemäß des angegebenen Algorithmus.

LÖSUNGSVORSCHLAG:

```
qsort :: [Int] -> [Int]
qsort   []  = []
qsort (h:t) = (qsort smaller) ++ [h] ++ (qsort bigger)
  where
    (smaller,bigger) = splitBy2 h t
```

Anstatt `[h] ++ (qsort bigg)` kann man natürlich auch `h : (qsort bigg)` schreiben!

A3-3 ***Abstiegsfunktion*** Zeigen Sie jeweils mithilfe einer geeigneten Abstiegsfunktion, dass jede der die folgende Funktionsdefinition für alle ganzen Zahlen terminiert!

a) `foo x = if x > 1 then 2*x + foo(x-2) else x `div` 2`

LÖSUNGSVORSCHLAG: Hier reicht die abgeschnittene Identität $\max(x, 0)$ als Abstiegsfunktion aus, da der rekursive Aufruf nur für $x > 1$ (also $x \geq 2$, da wir hier den Typ `int` betrachten) ausgeführt wird und in diesem Falle immer $\max(x, 0) > \max(x - 2, 0)$ gilt. Insbesondere bemerken wir, das für $\max(x, 0) = 0$ kein rekursiver Aufruf stattfindet.

b) Gegeben ist folgende Funktionsdefinition:

```
bar :: Integer -> Integer -> Integer
bar x y | x+y < 1 = 1
        | odd  y  = (x+1) + (bar (x-1) y)
        | even y  = (y+1) * (bar x (y-1))
```

i) Warum ist $f(x, y) = \max(x, 0)$ hier keine geeignete Abstiegsfunktion?

LÖSUNGSVORSCHLAG:

Wir zeigen dies durch Angabe eines Gegenbeispiels, d.h. wir müssen Funktionsargumente wählen, so dass ein rekursiver Aufruf statt findet, aber der Wert von f sich dabei nicht verringert. Es gibt meist viele Gegenbeispiele — eins reicht aus.

Für `bar 1 2` erfolgt ein rekursiver Aufruf mit `bar 1 1` im letzten Fall, da weder `1 + 2 < 1` noch `odd 2` zutrifft, aber `even 2` zu `True` auswertet.

Nun rechnen wir einfach nach: $f(1, 2) = \max(1, 0) = 1 \not> 1 = \max(1, 0) = f(1, 1)$. Der Wert der Abstiegsfunktion wurde für die Argumente des rekursiven Aufrufs also nicht echt kleiner, wie es notwendig wäre.

ii) Zeigen Sie mithilfe einer geeigneten Abstiegsfunktion, dass `bar` immer für alle beliebigen ganzen Zahlen terminiert!

LÖSUNGSVORSCHLAG:

Wir beweisen die Terminierung von `bar` für beliebige Argumente. *Hinweis:* Diese Lösung ist besonders ausführlich verfasst zur Verdeutlichung der Vorgehensweise.

Auf) Wir sollen die Terminierung für alle ganzzahligen Argumente zeigen, d.h. wir haben $A' = A$ und $A = \mathbb{Z} \times \mathbb{Z}$. Wir suchen also eine Abstiegsfunktion $\mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$. Es gibt zwei rekursive Aufrufe, welche beide die Vorbedingung $x + y \geq 1$ haben. Die rekursiven Aufrufe erfolgen also offensichtlich für Argument in A' – mit dieser Problematik muss man sich auch nur dann beschäftigten, wenn man die Terminierung für eine Teilmenge zeigen möchte, also z.B. für eine Funktion, welche für negative Argumente divergiert aber für alle positiven Argumente terminiert.

Def) Der Funktionsrumpf von `bar` ist offensichtlich für alle Argumente definiert: Die Fallunterscheidung mit Wächtern muss alle Fälle abdecken. Da `odd` und `even` komplementär sind, muss einer dieser beiden Fälle immer eintreten. Abgesehen von rekursiven Aufrufen werden lediglich die Funktionen `(+),(-),(<),odd,(*)` aufgerufen, welche bekanntlich für alle ganze Zahlen definiert sind.

Abst) Als Abstiegsfunktion nehmen wir die abgeschnittene Summe von $f(x, y) := \max(x+y, 0)$. Überprüfen wir nun, ob die Abstiegsfunktion für jeden rekursiven Aufruf kleiner wird.

A) Der erste rekursive Aufruf: `bar (x-1) y`. Wir müssen also zeigen, dass der Wert der Abstiegsfunktion für die Argumente $(x - 1)$ und y echt kleiner ist als für x und y. x und y sind beliebige ganze Zahlen, aber nach der vorausgegangenen Fallunterscheidung dürfen wir hier $x + y \geq 1$ und y ist ungerade annehmen. Dass y ungerade ist, hilft uns hier nicht weiter, aber aus der Bedingung $x + y \geq 1$ folgt die erste und letzte Gleichheit dieser Kette: $\max(x + y, 0) = x + y > x + y - 1 = \max((x - 1) + y, 0)$.

B) Der zweite rekursive Aufruf: `bar x (y-1)`. Nach der vorausgegangenen Fallunterscheidung dürfen wir $x + y \geq 1$ und y gerade annehmen. Mithilfe der ersten Annahme folgt $\max(x + y, 0) = x + y > x + y - 1 = \max(x + (y - 1), 0)$.

c)
```
foobar :: Integer -> Integer -> Integer
foobar x y
 | x > 0     = 1 + (foobar (x-1) y)
 | y > 0     = foobar 7 (y-1)
 | otherwise = 0
```

LÖSUNGSVORSCHLAG:

Auf) Auch hier müssen wir nach Aufgabenstellung $A' = \mathbb{Z} \times \mathbb{Z}$ betrachten. Rekursive Aufrufe finden jedoch nur statt, wenn mindestens eines der Argumente positiv ist.

Def) Die Fallunterscheidung ist aufgrund des Wächters `otherwise` offensichtlich vollständig. Es werden keine Funktion verwendet, welche möglicherweise nicht terminieren oder gar Fehlermeldung liefern können (z.B. kein Aufruf von `head`, keine Division mit Null, etc.).

Abst) Als Abstiegsfunktion wählen wir $\max(x, 0) + 8 * \max(y, 0)$.

Für den ersten rekursiven Aufruf haben wir $\max(x, 0) + 8 * \max(y, 0) = x + 8 * \max(y, 0) > x - 1 + 8 * \max(y, 0) = \max(x - 1, 0) + 8 * \max(y, 0) \geq 0$, da wir $x \geq 1$ annehmen dürfen. Für den zweiten rekursiven Aufruf haben wir $y \geq 1$ und $x \leq 0$. Damit gilt $\max(x, 0) + 8 * \max(y, 0) = 0 + 8 * y > -1 + 8 * y = 7 + 8 * (y - 1) = \max(7, 0) + 8 * \max(y - 1, 0)$.

d)
```
barfoo x
  | x < 111 = (barfoo (x+1) * barfoo (x+2) * x) `mod` 111
  | otherwise = 1
```

LÖSUNGSVORSCHLAG: Wir wählen $\max(111 - x, 0)$ als Abstiegsfunktion. Für beide rekursiven Aufrufe haben wir als Vorbedingung $x < 111$. Für den zweiten rekursiven Aufruf brauchen wir hier bei genauer Betrachtung eine Fallunterscheidung: Falls $x < 110$, dann folgt $0 \leq \max(111 - (x + 2), 0) = 109 - x < 111 - x = \max(111 - x, 0)$, ansonsten gilt für $x = 110$ die Betrachtung $\max(111 - (110 + 2), 0) = \max(-1, 0) < 1 = 111 - x = \max(111 - x, 0)$.

Hätten wir als Abstiegsfunktion besser $\max(222 - x, 0)$ gewählt, was ebenfalls möglich ist, so könnten wir uns diese letzte lästige Fallunterscheidung sparen. Das `k` für alle x mit $\max(222 - x, 0) = 0$ terminiert ist auch bei dieser Abstiegsfunktion klar, da daraus ja $x \geq 222$ folgt.

Ende der Lösungsvorschläge für die Präsenzaufgaben.
Lösungsvorschläge für die Hausaufgaben folgen nach Ende der Abgabezeit.

H3-1 ***Abstiegsfunktion II*** (2 Punkte; Abgabe: `H3-1.txt` oder `H3-1.pdf`)
Gegeben ist folgende Funktionsdefinition:

```
magic :: Integer -> Integer -> Integer -> String
magic a b c | even c, b > 0, a < 0 = 'x' : (magic (1+a)  b    (1+c))
            | odd  c, b > 0, a < 0 = 'y' : (magic    a  (b-1) (c+1))
            | otherwise            = show c
```

a) Warum ist die Funktion $f(a, b, c) := \max(a+c, 0)$ hier keine geeignete Abstiegsfunktion? Zeigen Sie dies durch Angabe eines Gegenbeispiels, also Werte für `a`,`b` und `c` so dass ein rekursiver Aufruf stattfindet, für welchen f verbotenerweise nicht kleiner wird.

b) Beweisen mithilfe einer geeigneten Abstiegsfunktion ausführlich, dass die folgend definierte Funktion für alle ganzen Zahlen terminiert.

Hinweise: Diese Funktion enthält für die Termination irrelevanten Code. Versuchen Sie die Abstiegsfunktion so einfach wie möglich zu wählen. Ihre Abstiegsfunktion muss nur *eine* obere Schranke an die rekursiven Aufrufe beschreiben; sie muss nicht exakt sein.

H3-2 ***Rekursion mit Listen*** (2 Punkte; Datei `H3-2.hs` als Lösung abgeben)
Implementieren zu Fuß[1] mit Pattern-Matching und Rekursion...

a) die Funktion `entferne :: Int -> [Int] -> [Int]` welche alle Vorkommen der gegebenen Zahl aus der Liste entfernt, aber ansonsten die Reihenfolge unverändert lässt:

```
> entferne 7 [4,5,6,7,7,8,9,8,7,6,7,4,7]
[4,5,6,8,9,8,6,4]
```

b) eine endrekursive Funktion `umdrehen :: [a] -> [a]` welche die Reihenfolge einer Liste umkehrt.

Abgabe: Lösungen zu den Hausaufgaben können bis Samstag, den 5.5.18, mit UniWorX nur als `.zip` abgegeben werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bis zu 3 Studierende können gemeinsam als Gruppe abgeben. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ss-2018/promo/`).

Es kann zu einem Punktabzug kommen, falls Ihre Abgabe nicht die geforderten Dateinamen genau einhalten, Funktionsnamen nicht richtig geschrieben wurden, andere Archive als `.zip` verwendet werden, oder wenn Syntaxfehler vorliegen.

Falls Sie an einer Stelle nicht weiter wissen, dann kommentieren Sie die entsprechenden Stellen mit einem Hinweisen aus und/oder vervollständigen Sie Ihre Abgabe mit einem Aufrufen der Funktion `error :: String -> a`. *Beispiel:*

```
foo _ [] = [] -- Fall ok!
foo _ [x] = error "H3-9b, foo: behandlung einelementiger Listen unklar" -- Hilfe!
-- foo x [h:t] = foo t ++ [h*x] -- Zeile kompiliert leider nicht. Hilfe!
```

Jeglicher Hinweistext sollte ordnungsgemäß als Kommentar in den Code geschrieben werden, d.h. hinter `--` oder in Kommentarklammern `{- mein hinweistext -}`

[1] Wie im Hinweis auf Blatt 2 angegeben ohne kompliziertere Funktionen aus der Standardbibliothek.