Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
6. Mai 2014

3. Musterlösung zur Vorlesung

Programmierung und Modellierung

A3-1 ***Wiederholung: Listen, Rekursion*** *Schnell und jeder für sich, egal ob am Computer oder mit Papier und Bleistift:* Implementieren Sie die Funktion `concat :: [[a]] -> [a]` aus der Standardbibliothek, welche die Elemente einer Liste von Listen miteinander verkettet. Verwenden Sie dazu Pattern-Matching auf Listen und Rekursion; und zur Vermeidung von Namenskonflikten einen frischen Funktionsnamen, z.B. `myConcat`.

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

> concat ["Hi ", "there", "!"]
"Hi there!"

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

LÖSUNGSVORSCHLAG:

Um Namenskonflikte zu vermeiden, nehmen wir einen frischen Namen:

```
myConcat :: [[a]] -> [a]
myConcat (  []) = []
myConcat (l:ls) = l ++ myConcat ls
```

Unsere Eingabe ist eine Liste, also unterscheiden wir zuerst per Pattern-Matching ob die Eingabeliste leer ist, oder nicht. Ist die Eingabeliste nicht leer, dann müssen wir das erste Element laut Aufgabenstellung verketten, also `(++)` anwenden; doch was ist das zweite Argument für diese Verkettung? Das zweite Element ist die Verkettung der restlichen Elemente, welche wir rekursiv durch die Anwendung von `myConcat` erhalten. Wir verketten also das erste Element der Eingabeliste mit der Verkettung der restlichen Elemente der Eingabeliste.

Die runden Klammern im zweiten Match sind notwendig, im ersten aber nicht. Denn `(:)` ist ein Infix-Konstruktor mit zwei Argumenten, während `[]` ein Konstruktor ohne Argumente ist.

Eine endrekursive Version ist hier nicht möglich (bzw. unsinnig, da `(++)` nicht endrekursiv formuliert werden kann, wenn die Reihenfolge der Elemente beibehalten werden soll.

A3-2 ***Abstiegsfunktion***

Zeigen Sie jeweils mithilfe einer geeigneten Abstiegsfunktion, dass jede der folgenden Funktionsdefinitionen für alle ganzen Zahlen terminiert!

a)
```
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))
```

LÖSUNGSVORSCHLAG:

Wir beweisen die Terminierung von `bar` für beliebige Argumente:

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.

i) 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 x ist ungerade annehmen. Ob y gerade oder ungerade ist, hilft hier nicht weiter. Aus der anderen Vorbedingung folgt aber die erste und letzte Gleichheit dieser Kette: $\max(x + y, 0) = x + y > x + y - 1 = \max((x - 1) + y, 0)$.

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

b)
```
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 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 $\max(x,0) + 8 * \max(y,0) = 0 + 8 * y > -1 + 8 * y = 7 + 8 * (y-1) = \max(7,0) + 8 * \max(y-1,0)$.

A3-3 *Induktion* Betrachten Sie die folgende rekursive Funktion `foo :: (Integer,Integer) -> Integer`

```
foo (n,m)
  | n == 0 = m+1
  | n >  0 = foo (n-1, foo (n-1,m))
  | otherwise = error "First Argument for foo is not allowed to be negative."
```

Raten Sie zuerst durch Ausprobieren dieser Funktion eine (nicht-rekursive) mathematische Formel, welche den Wert von `foo (n,m)` für beliebige natürliche Zahlen $n, m \in \mathbb{N}$ berechnet.

Beweisen Sie anschließend mit Induktion, dass Sie zuvor richtig geraten haben!

LÖSUNGSVORSCHLAG:

Durch Einsetzen von konkreten Werten erhält man die Vermutung $\texttt{foo}(n, m) = 2^n + m$. Dies beweisen wir mithilfe des Induktionsprinzips für natürliche Zahlen über n.

Induktionsanfang: Sei $n = 0$. Es gilt $\texttt{foo}(0, m) = m + 1 = 1 + m = 2^0 + m$ wie benötigt. Damit ist die Behauptung für den Basisfall bewiesen.

Induktionsschritt: Sei $n > 0$. Nehmen wir an, dass die Induktionshypothese für alle kleineren Werte als n gilt, d.h. wir dürfen die Gleichung $\texttt{foo}(x, y) = 2^x + y$ für alle $x < n$ verwenden.

Betrachten wir nun, was die Funktion `foo` an der Stelle (n, m) mit $n > 0$ macht. Nach der angegebenen Definition haben wir $\texttt{foo}(n, m) = \texttt{foo}(n - 1, \texttt{foo}(n - 1, m))$. Da $n - 1 < n$ gilt, dürfen wir die Induktionsannahme zwei Mal einsetzen:

$$\begin{aligned}\texttt{foo}(n, m) &= \texttt{foo}(n - 1, 2^{n-1} + m) \\ &= 2^{n-1} + (2^{n-1} + m)\end{aligned}$$

Jetzt müssen wir nur noch etwas umformen:

$$\begin{aligned}&= (2^{n-1} + 2^{n-1}) + m \\ &= (2 \cdot 2^{n-1}) + m \\ &= 2^n + m\end{aligned}$$

womit auch dieser Fall abgeschlossen wäre.

Damit haben wir mit Induktion also bewiesen: Für alle $n \in \mathbb{N}$ gilt $\texttt{foo}(n, m) = 2^n + m$. Da im Beweis keine Annahme über m gemacht wurde, gilt diese Formel sogar für alle $n \in \mathbb{N}$ und für alle $m \in \mathbb{N}$.

Generelle Hinweise zur Hausaufgabenabgabe:

Wird zu einer Hausaufgabe die Abgabe einer .hs-Datei gefordert, so muss diese als einzelne Datei pro Aufgabe abgegeben werden (Teilaufgaben in einer gemeinsamen Datei). Die Abgabe kann mit 0 Punkten bewertet werden, falls diese Datei von GHC/GHCi mit einer Fehlermeldung abgelehnt wird! Korrigieren Sie also alle Syntaxfehler vor der Abgabe! Falls Sie dies nicht hinbringen, dann kommentieren Sie die entsprechenden Stellen mit Hinweisen aus und vervollständigen Sie Ihre Abgabe mit Aufrufen der Funktion `error :: String -> a`.
Beispiel:

```
foo _ [] = [] -- Fall ok!
foo _ [x] = error "A0-3b, 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 -}`

Wenn nichts anderes angegeben ist, dürfen Sie Code von Vorlesungsfolien und vorangegangenen Aufgaben wiederverwenden. Die Verwendung von Funktionen der Standardbibliothek ist tabu, abgesehen von den Grundoperationen wie `(:)`, `(++)`, `(>)`, `(<=)`, `div`, `mod`, etc.

H3-1 ***Wdh.: Listen, Rekursion*** **(0 Punkte)** (`.hs`-Datei als Lösung abgeben)
Implementieren Sie flott, lediglich mit Pattern-Matching und Rekursion:

a) die Infix-Funktion `(++) :: [a] -> [a] -> [a]` aus der Standardbibliothek selbst. Diese Funktion verkettet zwei einzelne Listen miteinander, also `[1,2,3] ++ [4,5] == [1,2,3,4,5]`. Zur Vermeidung von Namenskonflikten geben Sie Ihrer Version den Namen `myAppend :: [a] -> [a] -> [a]` in Präfix-Notation, also:

```
> myAppend [1,2,3] [4,5]
[1,2,3,4,5]
```

LÖSUNGSVORSCHLAG:

```
myAppend []    l = l
myAppend (h:t) l = h : myAppend t l
```

b) eine endrekursive Version von `reverse :: [a] -> [a]` welche die Reihenfolge einer Liste umkehrt.

LÖSUNGSVORSCHLAG:

Wir müssen hier das vorläufige Ergebnis in einem zusätzlichen Argument mitführen. Ob wir dieses zusätzliche Argument `acc` als erstes oder zweites Argument definieren, ist dabei egal.

```
reverse :: [a] -> [a]
reverse l = rev_aux [] l
  where rev_aux :: [a] -> [a] -> [a]
        rev_aux acc    [] = acc
        rev_aux acc (h:t) = umkehrer_aux (h:acc) t
```

H3-2 *Abstiegsfunktion II* (4 Punkte) (Abgabeformat: Text oder PDF)
Beweisen mithilfe einer geeigneten Abstiegsfunktion ausführlich, dass die folgend definierte Funktion für alle ganzen Zahlen terminiert:

```
barfoo :: Integer -> Integer
barfoo x
  | x >= 123       = x - 666
  | x <= 1, x >= 0 = 42
  | otherwise      = barfoo (x+1) * barfoo (x*x) * 3
```

LÖSUNGSVORSCHLAG:

Auf) Wir sollen die Terminierung für alle ganzzahligen Argumente zeigen, d.h. wir haben $A' = \mathbb{Z}$. Rekursive Aufrufe finden nur für $x < 123$ und $x \neq 0$ statt.

Def) Die Fallunterscheidung ist wegen dem `otherwise`-Fall trivialerweise vollständig. Es werden keine Funktion verwendet, welche möglicherweise nicht terminieren oder gar Fehlermeldung liefern können.

Abst) Es gibt zwei rekursive Aufrufe, für welche wir getrennt zeigen müssen, dass die Abstiegsfunktion für die Argumente kleiner wird. Wir können die bereits gemachte Beobachtung verwenden, dass rekursive Aufruf nur für $x \in \{x \mid x \in \mathbb{Z} \text{ und } x < 123 \text{ und } x \neq 0 \text{ und } x \neq 1\}$ stattfinden.

Als Abstiegsfunktion wählen wir $f(x) := \max(123-x, 0)$. Aus der Vorbedingung $x < 123$ folgt $\max(123 - x, 0) = 123 - x > 0$.

Der erste rekursive Aufruf hat das Argument $x+1$. Wir rechnen nach: $\max(123-x, 0) = 123-x > (123-x)-1 = 123-(x+1) = \max(123-(x+1), 0)$, d.h. die Abstiegsfunktion wird also echt kleiner wie benötigt.

Für den zweiten rekursiven Aufruf mit Argument $x * x$ benötigen wir noch, dass aus den gegebenen Vorbedingung $0 \neq x$ und $1 \neq x$ auch $x * x > x$ folgt. Damit rechnen wir zunächst: $\max(123 - x, 0) = 123 - x > 123 - (x * x)$. Jetzt gibt es zwei Unterfälle zu betrachten: Im Fall $123-(x*x) = \max(123-(x*x), 0)$ sind wir fertig. Im Fall $123-(x*x) < 0$ haben wir $\max(123-(x*x), 0) = 0$. Da wir am Anfang bereits $\max(123-x, 0) > 0$ festgestellt hatten, wurde also auch in diesem Teilfall die Abstiegsfunktion noch echt kleiner.

Damit haben wir gezeigt, dass die Abstiegsfunktion in allen auftretenden Fällen echt kleiner wird. Da 0 der kleinste Wert der Abstiegsfunktion ist, muss die Rekursion spätestens enden, wenn dieser Wert erreicht wurde.

H3-3 *Benutzerdefinierte Datentypen* (4 Punkte) (`.hs`-Datei als Lösung abgeben)

a) Gegeben sind folgende Definitionen:

```
data Brotzeit = Weisswurscht Int | Breze Int Brotzeit
              | Leberkas Double  | Radi Double Brotzeit
  deriving (Show)

bz_steffen = Breze 3 ( Radi 0.5 (Weisswurscht 2))
bz_martin  = Breze 1 ( Radi 0.02 ( Radi 0.01 ( Breze 1 ( Leberkas 1.6 ))))
```

Eine `Brotzeit` ist also entweder eine Anzahl Weisswürschte, oder eine Anzahl Brezen als Beilage einer anderen Brotzeit, oder ein Leberkas mit einem Gewicht in Kilogramm, oder ein Radi mit Gewicht in Kilogramm als Beilage einer anderen Brotzeit. `bz_steffen` beinhaltet zum Beispiel 3 Brezen, ein Radi mit 500g und zwei Weisswürschte.

Schreiben Sie eine Funktion `anzahl :: Brotzeit -> Int` welche die Summe der Objekte einer `Brotzeit` ausrechnet. Ein Radi und ein Leberkas zählen unabhängig vom Gewicht als ein Objekt. *Beispiele:*

```
> anzahl bz_steffen
6
> anzahl bz_martin
5
```

LÖSUNGSVORSCHLAG:

```
anzahl :: Brotzeit -> Int
anzahl (Weisswurscht x) = x
anzahl (Leberkas _)     = 1
anzahl (Breze x b)      = x + anzahl b
anzahl (Radi _  b)      = 1 + anzahl b
```

Eine Brotzeit ist im Grunde genommen wie eine Liste, nur mit zwei Varianten für Knoten und Ende.

b)
```
data Korb = Korb { weisswürschte, brezen :: Int,
                   leberkas :: Double, radi :: (Int,Double) }
    deriving (Show)

leererKorb = Korb { weisswürschte=0, brezen=0, leberkas=0, radi=(0,0) }
tcs_korb = leererKorb { weisswürschte=2, brezen=3, leberkas=1.6, radi=(3,0.53) }
```

Ein `Korb` enthält also eine Anzahl Weisswürschte und eine Anzahl an Brezen, und einen Leberkäse mit einem angegebenen Gesamtgewicht in Kilogramm und eine Anzahl Radi mit einem Gesamtgewicht in Kilogramm.

Schreiben Sie eine Funktion `brotzeit2korb :: Brotzeit -> Korb` welche einen Wert des Typs `Brotzeit` in einen Wert des Typs `Korb` umrechnet, so dass äquivalente Lebensmittel enthalten sind. *Beispiele:*

```
> brotzeit2korb bz_steffen
Korb {weisswürschte = 2, brezen = 3, leberkas = 0.0, radi = (1,0.5)}
> brotzeit2korb bz_martin
Korb {weisswürschte = 0, brezen = 2, leberkas = 1.6, radi = (2,3.0e-2)}
```

LÖSUNGSVORSCHLAG:

```
brotzeit2korb (Weisswurscht x) = leererKorb { weisswürschte = x }
brotzeit2korb (Leberkas  g)    = leererKorb { leberkas = g }
brotzeit2korb (Breze x b)      = restKorb { brezen = x + (brezen restKorb) }
  where restKorb = brotzeit2korb b
brotzeit2korb (Radi  g b)      =
  let restkorb    = brotzeit2korb b
      radi_bisher = radi restkorb
      (radi_anzahl_alt, radi_gewicht_alt) = radi_bisher
      radi_neu    = (radi_anzahl_alt + 1, radi_gewicht_alt + g)
  in restkorb { radi = radi_neu }
```

Zur Veranschaulichung der verschiedenen Möglichkeiten wurde hier einmal `where` und einmal `let` verwendet. Ihr Code ist jedoch vermutlich lesbarer, wenn Sie sich pro Funktionsdefinition auf eine Notation festlegen.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 13.05.2014, 11:00 Uhr mit UniworX abgegeben werden. Die Hausaufgaben müssen von Ihnen alleine gelöst werden; Abschreiben wird als Betrug gewertet und ans Prüfungsamt gemeldet.