Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
5. Mai 2015

3. Musterlösung zur Vorlesung

Programmierung und Modellierung

A3-1 ***Termination*** Ein Riesenfass ist mit vielen Weißwürsten und Brezen gefüllt. Außerdem steht neben dem Fass ein Backofen, der beliebig viele Brezen erzeugen kann. Der folgende Vorgang soll nun solange ausgeführt werden, bis er sich nicht mehr wiederholen läßt:

1) Entnehmen Sie zwei beliebige Nahrungsmittel aus dem Fass.

2) Falls beide vom gleichen Typ sind, essen Sie sie auf und füllen eine Breze aus dem Backofen in das Fass.

3) Haben Sie eine Breze und eine Weißwurst genommen, dürfen Sie nur die Breze verputzen. Die Weißwurst muss wieder in das Fass zurück.

Beantworten Sie folgende Frage: Terminiert dieser Vorgang? Falls ja, wie viele Schmankerl verbleiben am Ende im Fass? Falls nein, wie entwickelt sich das Verhältnis von Weißwürsten zu Brezn? Begründen Sie Ihre Antwort!

LÖSUNGSVORSCHLAG: Es bezeichne B die Anzahl der Brezen und W die Anzahl der Weißwürste im Fass. Der Zustand des Fasses läßt sich also durch das Paar (B, W) beschreiben. Es gibt drei mögliche Ausgänge eines Schrittes, nämlich:

- Es werden zwei Weißwürste genommen. Dann wird W zu $W - 2$ geändert, und B zu $B + 1$, also der Zustand (B, W) zu $(B + 1, W - 2)$.
- Es werden zwei Brezen genommen. Dann wird (B, W) zu $(B - 1, W)$ geändert.
- Es werden zwei verschiedene Schmankerl genommen. Auch in diesem Fall wird (B, W) zu $(B - 1, W)$ geändert.

Nach jedem Schritt hat sich also die Summe $B + W$ um eins verringert, daher terminiert der Vorgang, wenn $B + W = 1$ ist, da dann kein Schritt mehr durchführbar ist. $B + W$ wäre also eine geeignete Abstiegsfunktion.

Ist W am Anfang ungerade, dann endet der Vorgang mit einer Weißwurst im Fass, andernfalls (W gerade) mit einer Breze.

A3-2 ***Abstiegsfunktion*** 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))
```

a) 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.

b) 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.

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 $\max(x+y,0) = x+y > x+y-1 = \max(x+(y-1),0)$.

A3-3 ***Datentypen Modellieren***

a) Geben Sie eine Datentypdeklaration für einen Typ `Akteur` an, um die Akteure eines Spiels zu verwalten: Es gibt Spieler, welche durch einen Namen (`String`) und ein Rating (`Double`) beschrieben werden; Computer, welche eine einstellbare Spielstärke (`Int`) besitzen; und Zuschauer, welche lediglich einen Namen (`String`) haben.

LÖSUNGSVORSCHLAG:

```
data Akteur = Spieler String Double | Computer Int | Zuschauer String
```

b) Implementieren Sie eine Funktion `anzeige :: Akteur -> String`, welche einen Akteur in einen String umwandelt, z.B. um diesen auf dem Bildschirm auszugeben. Bei der Darstellung als String soll ein Spieler nur durch seinen Namen repräsentiert werden, z.B. `"Steffen"`, d.h. das Rating des Spielers bleibt geheim, z.B. `"Steffen"`; bei einem Computer wird die Spielstärke mitangegeben, z.B. `"KI(42)"`; ein Zuschauer wird ebenfalls nur durch seinen Namen dargestellt, aber zur Unterscheidung soll dieser in Klammern eingefasst werden, z.B. `"[Martin]"`.

Hinweise: Folgende Bibliotheksfunktion könnten dabei nützlich sein: `show` zur Umwandlung von Zahlen in Strings; `(++)` zum Aneinanderhängen zweier Strings.

LÖSUNGSVORSCHLAG:

```
anzeige :: Akteur -> String
anzeige (Spieler   name _)   = name
anzeige (Computer  strength) = "KI(" ++ show strength ++ ")"
anzeige (Zuschauer name)     = "[" ++ name ++ "]"
```

H3-1 ***Abstiegsfunktion II*** (3 Punkte; Abgabeformat: Text oder PDF)
Gegeben ist folgende Funktionsdefinition:

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

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

LÖSUNGSVORSCHLAG:

Wir müssen die Werte so wählen, dass ein rekursiver Aufruf statt findet, z.B. für `foobar (-10) 2 1` erfolgt ein rekursiver Aufruf mit `foobar (-9) 3 1`, da `y` gerade war, `z > 0` und `x < 0`. Der Wert von f verringert sich jedoch nicht, wie man leicht nachrechnen kann: $f(-10, 2, 1) = \max(-10 + 2, 0) = \max(-8, 0) = 0 \not> \max(-6, 0) = \max(-9 + 3, 0) = f(-9, 3, 1)$

Bemerkenswerterweise terminert der Aufruf `foobar (-10) 2 0` sogar nach nur einem Schritt. Dies ist aber irrelevant, da f nicht dazu geeignet ist, dies zu zeigen.

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.

LÖSUNGSVORSCHLAG:

Auf) Wir sollen die Terminierung für alle ganzzahligen Argumente zeigen, d.h. wir haben $A' = A = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. Alle rekursiven Aufruf finden offensichtlich wieder nur für ganze Zahlen satt.

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 gewählte Abstiegsfunktion für die Argumente kleiner wird. Als Abstiegsfunktion wählen wir $f(x, y, z) := \max(z - x, 0)$.

Für den ersten rekursiven Aufruf haben wir die Vorbedingungen y ist gerade (für uns irrelevant), $z > 0$ und $x < 0$. Wir rechnen: $f(x, y, z) = \max(z - x, 0)$. Wegen $z > 0$ und $x < 0$ folgt $\max(z-x, 0) = z-x > 0$. Damit gilt auch $z-x > z-x-1 \geq 0$.

Weiterhin gilt $z - x - 1 = \max(z - (x + 1), 0) = f(x + 1, 1 + y, z)$. Der Wert der Abstiegsfunktion wurde also echt kleiner.

Für den zweiten rekursiven Aufruf benötigen wir ebenfalls nur $z > 0$ und $x < 0$. Wir rechnen ganz analog: $f(x, y, z) = \max(z - x, 0) = z - x > z - x - 1 = \max((z - 1) - x), 0) = f(x, y + 1, z - 1)$ wobei die inneren Gleicheiten und die Ungleicheit wie zuvor aus $z > 0$ und $x < 0$ folgen. Der Wert der Abstiegsfunktion wurde also wieder 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.

H3-2 ***Benutzerdefinierte Listen*** (2 Punkte; Datei `H3-2.hs` als Lösung abgeben)
Listen verstehen die meisten Teilnehmer sehr gut, so dass es Ihnen hoffentlich nicht mehr allzu viel Mühe bereitet, folgende Definition aus der Standardbibliothek zu verstehen:

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

Die Infix-Funktion `(++)` verkettet zwei Listen miteinander, also

```
> [1..3] ++ [4..5]
[1,2,3,4,5]
> (++) ['K','l'] ('a':'r':['o','!'])
"Klaro!"
```

Fragen Sie Ihren Assisstenten oder Tutor, falls Sie mit dieser polymorphen, rekursiven Infix-Definition noch Schwierigkeiten haben!

Viele Teilnehmer haben jedoch Probleme mit benutzerdefinierten Datentypen, weshalb auf Vorlesungsfolien 04-14 und 04-18 gezeigt wurde, wie man gewöhnliche Listen äquivalent "zu Fuss" deklarieren müsste, falls diese in Haskell nicht eingebaut wären:

```
data List a = Leer | Element a (List a)
  deriving Show
```

Ihre Aufgabe: Schreiben Sie eine Funktion `verketten :: List a -> List a -> List a`, welche zwei Listen mit einander verkettet. Sie müssen also lediglich den oben angegeben Code zur Benutzung des Datentyps auf Folie 04-18 umschreiben. Die Struktur des Codes (Fallunterscheidungen, Rekursion, etc.) bleiben gleich. *Beispiel:*

```
> verketten (Element 1 Leer) (Element 2 (Element 3 Leer))
Element 1 (Element 2 (Element 3 Leer))
```

LÖSUNGSVORSCHLAG:

```
verketten :: List a -> List a -> List a
verketten Leer           ys = ys
verketten (Element x xs) ys = Element x (verketten xs ys)
```

H3-3 ***Benutzerdefinierte Datentypen*** (4 Punkte; Datei `H3-3.hs` als Lösung abgeben)
Gegeben sind folgende Datentypdeklarationen

```
data Vergleich = Schlechter | Gleich | Besser          deriving Show
data Spiel     = Brettspiel String Int | Bausatz Int   deriving Show
```

Ein Spiel ist entweder ein Brettspiel, beschrieben durch einen Namen und eine Anzahl an erlaubten Spielern; oder einen Bausatz mit einer gewissen Anzahl an Bauteilen.

Schreiben Sie eine Funktion `vergleiche :: Spiel -> Spiel -> Vergleich`, welche zwei gegebene Spiele vergleicht.

Die Spiele vergleichen wir nach der Anzahl der möglichen Mitspieler: Je mehr, desto besser. Die Anzahl der Mitspieler eines Bausatzes ergibt sich aus der Anzahl der Teile dividiert durch 50 (abgerundet). Zum Beispiel gelten zwei Bausätze mit 212 und 243 Teilen gelten also ebenfalls als `Gleich`, da beide 4 Spieler erlauben.

Hinweis: Verwenden Sie die Funktion `div` aus der Standardbibliothek zur abgeschnittenen Division zweier ganzer Zahlen. Wie immer dürfen Sie gerne Hilfsfunktionen definieren, falls es Ihnen hilft.

LÖSUNGSVORSCHLAG:

```
vergleiche :: Spiel -> Spiel -> Vergleich
vergleiche x y
  | xPlayers > yPlayers = Besser
  | xPlayers < yPlayers = Schlechter
  | otherwise = Gleich
  where xPlayers = getPlayers x
        yPlayers = getPlayers y

getPlayers :: Spiel -> Int
getPlayers (Brettspiel _ spieler) = spieler
getPlayers (Bausatz teile _)      = teile `div` 50
```

In obiger Lösung wurde also zuerst eine Hilfsfunktion eingesetzt, um ein Spiel auf die Anzahl der Mitspieler abzubilden. Dies reduziert dann praktischerweise die Anzahl der notwendigen Fallunterscheidungen.

Natürlich kommt man auch ohne eine Hilfsfunktion aus, dann muss man aber etwas mehr Schreibarbeit leisten:

```
vergleiche' :: Spiel -> Spiel -> Vergleich
vergleiche' (Bausatz xTeile) (Bausatz yTeile)
  | xPlayers > yPlayers = Besser
  | xPlayers < yPlayers = Schlechter
  where xPlayers = xTeile `div` 50
        yPlayers = yTeile `div` 50
vergleiche' (Brettspiel _ xPlayers) (Bausatz yTeile)
  | xPlayers > yPlayers = Besser
  | xPlayers < yPlayers = Schlechter
  where yPlayers = yTeile `div` 50
vergleiche' (Bausatz xTeile) (Brettspiel _ yPlayers)
  | xPlayers > yPlayers = Besser
  | xPlayers < yPlayers = Schlechter
  where xPlayers = xTeile `div` 50
vergleiche' (Brettspiel _ xPlayers) (Brettspiel _ yPlayers)
  | xPlayers > yPlayers = Besser
  | xPlayers < yPlayers = Schlechter
vergleiche' _ _ = Gleich
```

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 12.05.2015, 11:00 Uhr mit UniworX abgegeben werden. Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen.

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 -}`