Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
8. Juli 2014

11. Musterlösung zur Vorlesung

Programmierung und Modellierung

Probeklausur

Auf den folgenden Seiten finden Sie eine unveränderte Klausur aus einem vergangenem Semester. Die Bearbeitungszeit betrug 120 Minuten und ist damit gleich zur Bearbeitungszeit der Erstklausur zur "Programmierung und Modellierung" in diesem Semester.

Diese Probeklausur ist natürlich nur von beispielhafter Natur. Eine Klausur kann immer nur eine Auswahl der behandelten Themen abfragen. Diese Klausur enthält z.B. keine/kaum Fragen zu Induktion, Monaden oder Semantik, welche in der aktuellen Programmierung und Modellierung sehr wohl behandelt wurden und entsprechend abgefragt werden können. Dafür würden dann aber andere Aufgaben wegfallen – der Gesamtumfang soll weitmöglichst gleich bleiben!

Auch zu diesem Übungsblatt wird eine Musterlösung herausgegeben werden.

Organisatorische Hinweise

- Eine Klausuranmeldung per UniworX ist zur Teilnahme zwingend erforderlich. Die Anmeldefirst ist abgelaufen. Eine Abmeldung ist bis 14.7. möglich. Eine komplette Abmeldung von der Veranstaltung ist nicht notwendig, da wir ohnehin nur die zur Klausur angemeldeten Teilnehmer an das Prüfungsamt melden. Allerdings werden auch angemeldete Klausurteilnehmer gemeldet, welche unentschuldigt nicht erscheinen.
- Jeder Student muss einen gültigen Lichtbildausweis und Studentenausweis mitbringen.
- Es sind keine Hilfsmittel zugelassen. Am Platz darf sich nur Schreibzeug und eventuell ein paar Nahrungsmittel befinden.
- Verwenden Sie keinen Bleistift und keine Stifte in rot oder grün! Verwendung Sie nur dokumentenechte Stifte!
- Papier wird von uns gestellt und darf nicht mitgebracht werden.
- Taschen und Jacken müssen vorne an der Tafel abgelegt werden. Sollte jemand ein Telefon, mp3-Player, oder Ähnliches am Platz haben, ist das ein Täuschungsversuch, der dem Prüfungsausschuss gemeldet wird.
- Sollte ein Telefon klingeln, ist das eine Störung des Prüfungsablaufs und hat den Ausschluss von der weiteren Teilnahme zur Folge.
- Gehen Sie rechtzeitig vor Beginn in den zugewiesenen Raum! Die Raumeinteilung wird erst 2–3 Tage vor Klausurbeginn auf der Vorlesungshomepage bekanntgegeben. Klausurort ist die Theresienstr. 39-41.

Nachklausur Eine Nachklausur ist derzeit in den letzten Wochen vor Beginn des kommenden Wintersemesters geplant; dies kann sich aber aufgrund der verfügbaren Räume und erwarteten Teilnehmerzahlen noch ändern.

Lösung Aufgabe 1 (Verschiedenes): **(6 Punkte)**

Richtig angekreuzt:	je	1 Punkte
Falsch angekreuzt:	je	−1 Punkte
Nicht angekreuzt:	je	0 Punkte
Doppelt angekreuzt:	je	0 Punkte

Veränderungen an der Einrückung (z.B. durch Leerzeichen) können dazu führen, das ein Haskell Programm nicht mehr kompiliert.	[X] ja	[ ] nein
Referentielle Transparenz bedeutet, dass eine Variable nach der Initialisierung nie verändert wird.	[X] ja	[ ] nein
Der Typ `a -> b -> c` ist identisch zu	[X] `a->(b->c)`	[ ] `(a->b)->c`
Die Definitionen `foo x = 2*x` und `foo = \x-> (*) 2 x` verhalten sich gleich.	[X] ja	[ ] nein
Die Typsubstitution $[\texttt{Int}/\alpha, \texttt{Bool} \to \texttt{Int}/\beta]$ unifiziert die beiden Typen $\texttt{Int} \to \delta \to \alpha$ und $\alpha \to \beta$	[ ] ja	[X] nein
Haskell Programme terminieren aufgrund der verzögerten Auswertungsstrategie immer.	[ ] ja	[X] nein

(Aufgabenstellungen von Aufgabe 5 verrät die Lösung bereits!)

Lösung Aufgabe 2 (Abstiegsfunktion): **(6 Punkte)**

Wir wollen mithilfe einer geeigneten Abstiegsfunktion zeigen, dass die folgende rekursive Funktion `foo :: (Int,Int) -> Int`, gegeben in Haskell Notation, immer terminiert:

```
foo (x,y)
 | x > 10 && y >  0 = foo (x-2, y  )
 | x > 0  && y > 10 = foo (x+1, y-1)
 | otherwise        = x + y
```

a) Welchen Wertebereich hat eine Abstiegsfunktion, d.h. in welchen Zahlenbereich bildet eine Abstiegsfunktion die Argumente der untersuchten Funktion ab?

LÖSUNG: Natürliche Zahlen. Die Abstiegsfunktion muss für jeden rekursiven Aufruf echt kleiner werden. Wenn der Zahlenbereich ganzzahlig (abzählbar) ist und ein kleinstes Element hat, garantiert dies Termination (Ansonsten: $[3, 2, 1, 0, -1, -2, -3, \dots]$, $[7, 6.9, 6.89, 6.889, 6.8889, \dots]$).

Die Lösung steht übrigens auch schon in der Aufgabenstellungen von Aufgabenteil c drin!

b) Zeigen Sie, dass die Funktion $m'(x, y) = \max(x, 0)$ keine geeignete Abstiegsfunktion für den Terminationsbeweis von `foo` ist.

LÖSUNG: Wir wählen irgendwelche Argumente, so dass wir im zweiten Fall sind, da dort x nicht kleiner wird, z.B. für $x = 5$ und $y = 12$ haben wir $m'(5, 12) = 5$, aber es findet dann ein rekursiver Aufruf mit den Argumenten $x = 6$ und $y = 11$ statt. Dafür haben wir $m'(6, 11) = 6 \not< 5$. Wir haben also ein Funktionsargument, für den der Wert von m' für einen rekursiven Aufruf nicht echt kleiner (also größer) wird. Somit ist m' keine geeignete Abstiegsfunktion.

Das Argument "weil y in m' nicht vorkommt" nutzt nichts, denn es gibt durchaus auch Programme, bei denen die Rekursion schon allein durch eines der Funktionsargumente beschränkt ist.

c) Finden Sie eine geeignete Abstiegsfunktion $m : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ und beweisen Sie, dass `f` immer terminiert.

LÖSUNG: Wir wählen als Abstiegsfunktion $m(x, y) = \max(x + 2y, 0)$ mit $m : \texttt{Int} \times \texttt{Int} \to \mathbb{N}$. Da ja nur nach $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ gefragt war, hätte man $\max(\cdot, 0)$ auch weglassen können.

Der erste rekursive Aufruf erfolgt für $x > 10$ und $y > 0$. Damit gilt in diesem Fall $\max(x+2y, 0) = x + 2y$. Wir rechnen:

$$m(x, y) = x + 2y > x + 2y - 2 = (x - 2) + 2y = \max(x - 2 + 2y) = m(x - 2, y)$$

Die Gleichung $(x - 2) + 2y = \max(x - 2 + 2y)$ gilt, da wegen $x > 10$ auch $x - 2 > 10 - 2 > 0$ gilt und aus $y > 0$ auch $2y > 0$ folgt.

Der zweite rekursive Aufruf erfolgt, falls $x > 0$ und $y > 10$ gilt. Wir rechnen:

$$m(x, y) = \max(x + 2y, 0) = x + 2y > x + 2y - 1 = x + 1 + 2y - 2 = x + 1 + 2(y - 1) = m(x + 1, y - 1)$$

Achtung: Die oft genannte Funktion $\max(x + y, 0)$ funktioniert übrigens nicht: $\text{foo}(10, 11)$ führt zum rekursiven Aufruf $\text{foo}(11, 10)$, aber $\max(10 + 11, 0) \not> \max(11 + 10, 0)$.

Lösung Aufgabe 3 (Listen): **(6 Punkte)**

Implementieren Sie folgende Funktionen rekursiv, und ohne die Verwendung von List-Comprehension oder Funktionen höherer Ordnung wie z.B. `map`, `filter`, `foldl`, `foldr`, `zip`, `all` oder `any`.

a) Implementieren Sie die Funktion `alle :: (a -> Bool) -> [a] -> Bool`, welche ein Prädikat und eine Liste als Argumente bekommt und prüft, ob alle Elemente der Liste das Prädikat erfüllen.

Beispiel:

```
> alle even [2,4,8]
True
> alle even [2,4,8,9,10]
False
```

LÖSUNG: Siehe Musterlösungen zu A5-2. Hier noch eine weitere Alternative:

```
alle p   []              = True
alle p (h:t) | p h       = alle p t
             | otherwise = False
```

b) Implementieren Sie die Funktion `index :: [a] -> [(Int, a)]`, welche eine Liste als Argument bekommt und eine Liste von Index-Element-Paaren produziert.

Beispiel:

```
> index ['a','b','c','d']
[(0,'a'),(1,'b'),(2,'c'),(3,'d')]
```

LÖSUNG:

```
index :: [a] -> [(Int,a)]
index xs = index' 0 xs
  where
    index' _ []    = []
    index' i (h:t) = (i,h):index' (i+1) t
```

c) Implementieren Sie erneut die Funktion `alle :: (a -> Bool) -> [a] -> Bool` aus der ersten Teilaufgabe, dieses mal aber ohne Rekursion, dafür dürfen Sie die Funktion foldr verwenden.

Zur Erinnerung:

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

LÖSUNG:

```
alle :: (a -> Bool) -> [a] -> Bool
alle p = foldr (\e acc -> p e && acc) True
    -- Alternativ auch akzeptiert:
alle p xs = foldr (&&) True $ map p xs
```

Lösung Aufgabe 4 (Datenstrukturen): (5 Punkte)

Gegeben ist folgende Datentypdeklaration zur Repräsentation von Typen:

```
data TTyp = TVar Char | TInt | TListe TTyp | TTupel [TTyp]
  deriving (Eq, Show)
```

Schreiben Sie eine Funktion `rename :: (Char -> Char) -> TTyp -> TTyp` welche alle `Char`-Zeichen in einem Wert des Typs `TTyp` gemäß einer gegebenen Funktion umbenennt.

Beispiel:
Den Typ einer Liste von Paaren aus ganzen Zahlen und eines unbekannten Typs `a`, welchen wir in Haskell mit `[(Int,a)]` bezeichnen würden, könnten wir als Wert von `TTyp` durch den Ausdruck `TListe (TTupel [TInt,TVar 'a'])` darstellen.

```
> let t = TListe (TTupel [TVar 'a', TInt, TVar 'c'])
t :: TTyp
> let s = \c -> if c=='a' then 'b' else c
s :: Char -> Char
> rename s t
TListe (TTupel [TVar 'b', TInt, TVar 'c'])
```

LÖSUNG:

```
rename :: (Char -> Char) -> TTyp -> TTyp
rename r (TVar v)     = TVar $ r v
rename _ TInt         = TInt
rename r (TListe tt)  = TListe $ rename r tt
rename r (TTupel tts) = TTupel $ map (rename r) tts
```

Bemerkung:
In dieser Aufgabe ging es eigentlich nur um Bäume. Der Bekanntheit wegen wurde hier ein zur damaligen Aufgabe "Unifikation II" sehr ähnlicher Baum-Datentyp wiederverwendet.

Natürlich konnte man auch ganz ordentlich eine `Functor` Instanz definieren, und `rename = fmap` definieren, was ja ebenfalls behandelt wurde.

Lösung Aufgabe 5 (I/O): **(6 Punkte)**

Schreiben Sie eine Funktion `main :: IO ()`, welche zuerst eine Begrüßung ausgibt, und danach zeilenweise Eingaben mittels `getLine :: IO String` vom Benutzer entgegen nimmt. Ihr Programm soll nicht terminieren. Nach jeder Eingabezeile des Benutzers gibt Ihr Programm die gesamte Zahl aller bisher eingegebenen Punkte-Zeichen `'.'` aus. Verwenden Sie zur Ausgabe der Zahl die Funktionen `putStrLn :: String -> IO ()` und `show :: Show a => a -> String`.

Beispiel:
(Die mit - beginnenden Zeilen wurden vom Benutzer eingegeben, inklusive dem Minus-Symbol.)

```
Sag mir Deine Meinung:
-Listen sind toll. Rekursion ist toll. Haskell ist toll.
Ok, 3 Punkte verstanden. Sag mir mehr:
-Die Klausur ist zu leicht.
Ok, 4 Punkte verstanden. Sag mir mehr:
-Was kann man da sonst noch sagen?!
Ok, 4 Punkte verstanden. Sag mir mehr:
```

LÖSUNG: Dies ist eine vereinfachte Version von A9-3. Die Knackpunkte sind hier:

a) Zählen der Punkte in einem String

b) Verwendung der DO-Notation

c) Idee einer monadischen Hilfsfunktion mit einem Argument um den Zählern zu behalten

```
main :: IO ()
  main = do
      putStrLn "Sag mir Deine Meinung:"
      countLine 0
  where
    countLine :: Int -> IO ()
    countLine i = do
      l <- getLine
      let p = i + (countDot l)
      putStrLn $ "Ok, " ++ (show p) ++ " Punkte verstanden. Sag mir mehr:"
      countLine p

    countDot :: String -> Int
    countDot = length . filter ('.'==)
        -- Es gibt zahlreiche Alternativen
    countDot1 :: String -> Int
    countDot1 s = sum [ 1 | '.' <- s ]
        -- oder auch ganz zu Fuss:
    countDot2 :: String -> Int
    countDot2     []  = 0
    countDot2 ('.':t) = 1 + countDot2 t
    countDot2 ( _ :t) =     countDot2 t
        -- mit if-then-else
    countDot3 :: String -> Int
    countDot3    [] = 0
    countDot3 (c:t) = (countDot3 t) + (if c=='.' then 1 else 0)
```

Lösung Aufgabe 6 (Typisierung): **(4 Punkte)**

Hinweis: Bitte jeweils nur das Ergebnis hinschreiben. Nebenrechnungen bitte deutlich abtrennen.

a) Was ist der allgemeinste Typ des Ausdrucks `[True] : []`

`[[Bool]]`

b) Was ist der allgemeinste Typ der Funktion `bar` mit

```
bar  x y u v
  | y > x     = "True"
  | u         = "False"
  | otherwise = show v
```

`Ord a, Show b => a -> a -> Bool -> b -> String`

c) Geben Sie einen (geschlossenen) Haskell Ausdruck an, welcher den Typ `(a -> b) -> [a] -> [b]` haben kann. "Geschlossen" bedeutet, das der Ausdruck keine freien Variablen enthält, d.h. es ist ein ganz gewöhnlicher Programmausdruck, denn man ohne weitere Definitionen in GHCI eintippen kann.

`\f x -> []` ist eine Lösung, aber auch `map` oder `\f xs -> [ f x | x <-xs]` wären richtig.

Lösung Aufgabe 7 (Typherleitungen): **(6 Punkte)**

Beweisen Sie folgende Typurteile unter Verwendung der zur Erinnerung auf Seite 10 angegeben Typregeln in einer der beiden in der Vorlesung behandelten Notationen (Herleitungsbaum oder lineare Schreibweise).

a) $\{\mathtt{x}{::}\alpha\} \vdash \backslash\mathtt{y} \to \mathtt{y}\ \ \mathtt{x} :: (\alpha \to \beta) \to \beta$

LÖSUNG: Als Herleitungsbaum:

$$\dfrac{\dfrac{\dfrac{}{\{\mathtt{x}{::}\alpha, \mathtt{y}{::}\alpha \to \beta\} \vdash \mathtt{y} :: \alpha \to \beta}\,(\textsc{Var}) \quad \dfrac{}{\{\mathtt{x}{::}\alpha, \mathtt{y}{::}\alpha \to \beta\} \vdash \mathtt{x} :: \alpha}\,(\textsc{Var})}{\{\mathtt{x}{::}\alpha, \mathtt{y}{::}\alpha \to \beta\} \vdash \mathtt{y}\ \ \mathtt{x} :: \beta}\,(\textsc{App})}{\{\mathtt{x}{::}\alpha\} \vdash \backslash\mathtt{y} \to \mathtt{y}\ \ \mathtt{x} :: (\alpha \to \beta) \to \beta}\,(\textsc{Abs})$$

Sehr viele Teilnehmer erreichten in dieser Aufgabe die volle Punktzahl. Dies ist doch überraschend, da in den Klausuren zur Veranstaltung "Logik und Diskrete Strukturen" Aufgaben zu Beweisbäumen im Sequenzenkalkül erfahrungsgemäß eher schlecht ankommen.

Die Aufgabe wird auf dem nächsten Blatt fortgesetzt.

Fortsetzung von Aufgabe 7:

b) $\{y{::}\texttt{Int}\} \vdash \texttt{let}\ x\ =\ \texttt{True}\ \texttt{in}\ (\texttt{if}\ x\ \texttt{then}\ 5\ \texttt{else}\ y) :: \texttt{Int}$

LÖSUNG: Zur Abwechslung in linearer Schreibweise:

$$\{y{::}\texttt{Int}\} \vdash \texttt{let}\ x\ =\ \texttt{True}\ \texttt{in}\ (\texttt{if}\ x\ \texttt{then}\ 5\ \texttt{else}\ y) :: \texttt{Int} \qquad \textsc{Let}(2,3) \quad (1)$$

$$\{y{::}\texttt{Int}\} \vdash \texttt{True} :: \texttt{Bool} \qquad \textsc{Bool} \quad (2)$$

$$\{y{::}\texttt{Int}, x{::}\texttt{Bool}\} \vdash \texttt{if}\ x\ \texttt{then}\ 5\ \texttt{else}\ y :: \texttt{Int} \qquad \textsc{Cond}(4,5,6) \quad (3)$$

$$\{y{::}\texttt{Int}, x{::}\texttt{Bool}\} \vdash x :: \texttt{Bool} \qquad \textsc{Var} \quad (4)$$

$$\{y{::}\texttt{Int}, x{::}\texttt{Bool}\} \vdash 5 :: \texttt{Int} \qquad \textsc{Int} \quad (5)$$

$$\{y{::}\texttt{Int}, x{::}\texttt{Bool}\} \vdash y :: \texttt{Int} \qquad \textsc{Var} \quad (6)$$

Lösung Aufgabe 8 (Typregel): **(4 Punkte)**

Überlegen Sie sich analog zu den in der Vorlesung behandelten Typregeln (siehe Seite 10) sinnvolle Typregeln zu Listen-Einführung und Listen-Elimination. Die Konklusion der Typregeln sind jeweils vollständig vorgegeben. Ergänzen Sie lediglich die fehlende(n) Prämisse(n)!

a) Listen werden in Haskell mit dem Cons-Operator `(:)` gebildet, welcher ein Element an den Anfang einer existierenden Liste stellt.

Beispiel der Verwendung von `(:)` in Haskell:

```
> 1 : [2,3]
[1,2,3]
```

Ihre Typregel:

$$\frac{\Gamma \vdash e_1 :: C \qquad \Gamma \vdash e_2 :: [C]}{\Gamma \;\vdash\; e_1 : e_2 \;::\; [C]} \quad \text{(List-Intro)}$$

LÖSUNG:

Instantiieren wir zur Überprüfung mal die Werte aus dem Beispiel, d.h. wir setzen $e_1 = 1$ und $e_2 = [2,3]$. Selbst wenn man die Lösung noch nicht kennt, könnte man durch dieses Einsetzen in die vorgegebene Konklusion Γ und C praktisch schon ablesen – und man sieht dann auch gleich, dass e_1 keinen Listentyp hat, wie fälschlicherweise oft angenommen wurde!

$$\frac{\{\,\} \vdash 1 :: \texttt{Int} \qquad \{\,\} \vdash [2,3] :: [\texttt{Int}]}{\{\,\} \vdash 1 : [2,3] :: [\texttt{Int}]} \quad \text{(List-Intro (Instanz))}$$

b) Listen werden in Haskell durch Pattern-Matching auseinandergenommen, z.B. durch Case-Ausdrücke wie etwa:

```
> case [1,2,3] of [] ->  0  ; (h:t) -> h
1
> case [1,2,3] of [] -> [0] ; (h:t) -> t
[2,3]
```

Hinweis:
Für Interpreter (und Typregel) ist es praktischer, den Case-Ausdruck mit Hilfe eines Semikolons in eine einzelne Zeile zu schreiben. Ohne Semikolon müsste man wie gewohnt Einrückung nach der Layout-Regel verwenden, zum Beispiel so:

```
case [1,2,3] of
  []    -> 0
  (h:t) -> h
```

Ihre Typregel:

$$\frac{\Gamma \vdash e_1 :: [A] \qquad \Gamma \vdash e_2 :: C \qquad \Gamma,\, h{::}A,\, t{::}[A] \vdash e_3 :: C}{\Gamma \;\vdash\; \texttt{case}\ e_1\ \texttt{of []} \to e_2\,;\,(h : t) \to e_3 \;::\; C} \quad \text{(List-Elim)}$$

Typregeln:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)} \quad \text{(Var)}$$

Alternative Schreibweise für Var:

$$\frac{x{::}A \in \Gamma}{\Gamma \vdash x :: A} \quad \text{(Var)}$$

$$\frac{c \text{ ist eine Integer Konstante}}{\Gamma \vdash c :: \texttt{Int}} \quad \text{(Int)}$$

$$\frac{c \in \{\texttt{True}, \texttt{False}\}}{\Gamma \vdash c :: \texttt{Bool}} \quad \text{(Bool)}$$

$$\frac{\Gamma \vdash e_1 :: A \to B \qquad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1\ e_2 :: B} \quad \text{(App)}$$

$$\frac{\Gamma, x{::}A \vdash e :: B}{\Gamma \vdash \backslash x \to e :: A \to B} \quad \text{(Abs)}$$

$$\frac{\Gamma \vdash e_1 :: A \qquad \Gamma \vdash e_2 :: B}{\Gamma \vdash (e_1, e_2) :: (A, B)} \quad \text{(Pair-Intro)}$$

$$\frac{\Gamma \vdash e_1 :: (B, C) \qquad \Gamma, x{::}B, y{::}C \vdash e_2 :: A}{\Gamma \vdash \texttt{let}\ (x, y)\ \texttt{=}\ e_1\ \texttt{in}\ e_2 :: A} \quad \text{(Pair-Elim)}$$

$$\frac{\Gamma \vdash e_1{::}A \qquad \Gamma,\, x{::}A \vdash e_2 :: C}{\Gamma\ \vdash\ \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 :: C} \quad \text{(Let)}$$

$$\frac{\Gamma \vdash e_1 :: \texttt{Bool} \qquad \Gamma \vdash e_2 :: A \qquad \Gamma \vdash e_3 :: A}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: A} \quad \text{(Cond)}$$