Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
16. Juni 2015

13. 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 Klausur zur "Programmierung und Modellierung" im aktuellen Semester.

Diese Probeklausur ist natürlich nur von beispielhafter Natur. Insbesondere kann eine Klausur immer nur eine Auswahl der behandelten Themen abfragen. Weiterhin weichen auch die behandelten Themen etwas ab: in diesem Jahr wurden die Typregeln für 2-Tuple nicht behandelt; dafür haben wir in diesem Jahr mehr zum Thema Denotationelle Semantik gemacht; anstelle der Zustandsmonade hatten wir in diesem Jahr die Par-Monade betrachtet; usw.

Auch zu diesem Übungsblatt wird eine Musterlösung per UniworX herausgegeben werden.

Organisatorische Hinweise

- Eine Klausuranmeldung per UniworX ist zur Teilnahme zwingend erforderlich. Die Anmeldefirst ist abgelaufen. Unentschuldigt fehlende Klausurteilnehmer werden ans Prüfungsamt gemeldet.
- 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 am Montag auf der Vorlesungshomepage bekanntgegeben.

Nachklausur Eine Nachklausur ist für Anfang Oktober geplant. Der Termin wird unverzüglich auf der ProMo-Homepage bekanntgegeben, so bald die Raumbuchung erfolgte. Die Anmeldung zur Nachklausur erfolgt ca. August per UniworX. Es wird nur eine Nachklausur angeboten werden; wer sich zur Nachklausur anmeldet, ohne vorher an der Erstklausur teilgenommen zu haben, kann die Prüfung erst wieder im Sommersemester 2016 versuchen.

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

a) Rechnen Sie aus, zu welchem Wert die folgenden Haskell-Ausdrücke vollständig auswerten. Geben Sie nur das Endergebnis an, etwaige Nebenrechnungen bitte deutlich abtrennen:

(i) `(4 == 5.0:[]`

__ `[False]` bekannt als A1-5a _____

(ii) `[[z,y]| x<-[1..4], y<-[1,3..8], let z = x-1, x==y]`

__ `[[0,1],[2,3]]` _____

(iii) `(\x y-> y) "yes" (\z -> "no") "which?"`

__ "no" ähnlich zu A1-5f _____

b) Werten Sie folgenden Ausdruck schrittweise vollständig aus, unterstreichen Sie dabei den reduzierten Redex. Um die volle Punktzahl zu erreichen,

Sie dabei dürfen eine beliebige Auswertestrategie verwenden. für volle Punktzahl müssen Sie aber Call-By-Value oder Call-By-Name verwenden.

Beispiel: $(\backslash x \text{ -> } 43 + x)$ $(\underline{\texttt{negate } 1}) \rightsquigarrow \underline{(\backslash x \text{ -> } 43 + x)\ (-1)} \rightsquigarrow \underline{43 + (-1)} \rightsquigarrow 42$

Definitionen:

```
const x y = x
negate x  = -x
```

Ausdruck zum Auswerten:
`const const (negate 1) (negate 2) 3` $\rightsquigarrow$

LÖSUNG: Call-By-Name:

$\underline{\texttt{const const (negate } 1)}\ (\texttt{negate } 2)\ 3 \rightsquigarrow \underline{\texttt{const (negate } 2)\ 3} \rightsquigarrow \underline{\texttt{negate } 2} \rightsquigarrow -2$

Call-By-Value:

$\texttt{const const } (\underline{\texttt{negate } 1})\ (\texttt{negate } 2)\ 3 \rightsquigarrow \texttt{const const } (-1)\ (\underline{\texttt{negate } 2})\ 3$
$\rightsquigarrow \underline{\texttt{const const } (-1)}\ (-2)\ 3 \rightsquigarrow \underline{\texttt{const } (-2)\ 3} \rightsquigarrow -2$

c) Welche Auswertestrategie haben Sie in Aufgabenteil b verwendet?

__ (siehe oben) _____

Lösung Aufgabe 2 (Induktion): **(5 Punkte)**

Wir betrachten folgende Funktionsdefinitionen:

```
length :: [Char] -> Int
length    []  = 0                              -- (LZ)
length (x:xs) = 1 + length xs                  -- (LS)

insert :: Char -> [Char] -> [Char]
insert x    [] = [x]                           -- (IN)
insert x (y:ys) | x <= y    = x : y : ys       -- (IK)
                | otherwise = y : insert x ys  -- (IG)
```

Beweisen Sie mit Induktion über die Länge der Liste, dass für alle Zeichen c und alle Strings s die folgende Gleichung gilt:

`length (insert c s)` = `1 + length s`

Hinweise: Formen Sie beide Seiten der geforderten Gleichung schrittweise in den exakt gleichen Term um. Begründen Sie jeden Umformungsschritt durch Angabe des Kürzels der verwendeten Gleichung, also `(LZ)`, `(LS)`, `(IN)`, `(IK)` oder `(IG)`, und `(IH)` bei Verwendung der Induktionshypothese! Beachten Sie die Pattern-Guards bei Verwendung von `(IK)` oder `(IG)` und begründen Sie kurz, warum der jeweilig Fall eintritt.

LÖSUNG: *Diese Aufgabe kennen wir in leicht anderer Formulierung als A6-3a.*
Der Beweis wird mit Induktion über die Länge der Liste `s` geführt. *Wichtig:* Die andere Variable `c` verbleibt als Variable von einem beliebigen Wert, denn wir wollen die Gleichung ja auch für beliebige Werte von `c` beweisen!

Induktionsanfang für eine Liste der Länge 0: Es gilt also `s = []`.

`length (insert c []) =(IN)= length [c] =(LS)= 1 + length []`

Damit haben wir die linke Seite der geforderten Gleichung direkt in die rechte Seite überführt. Den Schritt `1 + length [] =(LZ)= 1 + 0 = 1` benötigen wir also gar nicht mehr.

Induktionsschritt für Liste der Länge $n > 0$: Da die Liste `s` in diesem Fall nicht leer ist, können wir dem Kopf und Rumpf von `s` eigene, frische Namen geben: Es sei `s = (h:t)`. Damit ist `t` eine Liste mit einer Länge kleiner als n. Somit dürfen wir als Induktionshypothese `(IH)` die Gleichung `length (insert d t)` = `1 + length t` für ein beliebiges Zeichen d verwenden.

Wir müssen die Gleichung `length (insert c (h:t)) = 1 + length (h:t)` beweisen. Zu Beginn möchten wir gerne `(IK)` oder `(IG)` anwenden. Da wir keine Information haben, welcher Fall eintritt, müssen wir einfach beide Fälle betrachten. Dies tun wir einer Fallunterscheidung:

Fall 1: $c <= h$
`length (insert c (h:t)) =(IK)= length (c:h:t) =(LS)= 1 + length (h:t)`
Die Induktionshypothese benötigen wir in diesem einfachen Fall also gar nicht.

Fall 2: $c > h$
`length (insert c (h:t)) =(IG)= length (h:(insert c t)) =(LS)=`
`1 + length (insert c t) =(IH)= 1 + 1 + length t =(LS)= 1 + length (h:t)`
Wer Schwierigkeiten mit dem letzten Umformungsschritt hat, sollte diesen einfach von rechts-nach-links lesen.

Lösung Aufgabe 3 (Abstiegsfunktion): **(5 Punkte)**

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

```
foo (x, y, z)
  | x > 10, z > 0  = 2 * foo (x-1, y+3, z+2)
  | x >  0, z > 20 = 4 * foo (x+3, y-1, z `div` 2)
  | otherwise = y
```

a) Zeigen Sie, dass die Funktion $m'(x, y, z) = \max(2x + z, 0)$ keine geeignete Abstiegsfunktion für den Terminationsbeweis von `foo` ist.

LÖSUNG: Wir wählen als Argument $(50, 50, 50)$. Damit sind wir im ersten Fall der Funktionsgleichung. Wenn m' eine Abstiegsfunktion wäre, dann müsste $m'(50, 50, 50) > m'(49, 50, 52)$ gelten.

Dies gilt aber nicht, wie wir durch nachrechnen beweisen:

$$m'(50, 50, 50) = \max(100 + 50, 0) = 150$$
$$m'(49, 50, 52) = \max(98 + 52, 0) = 150$$

Wir haben also ein Funktionsargument, für den der Wert von m' für einen rekursiven Aufruf nicht echt kleiner wird. Somit ist m' keine geeignete Abstiegsfunktion.

Das Argument "weil y in m' nicht vorkommt" nutzt gar nichts; wie wir im zweiten Aufgabenteil sehen, muss nicht jedes Argument in der Abstiegsfunktion berücksichtigt werden.

b) Finden Sie eine Abstiegsfunktion m und beweisen, dass diese eine tatsächlich eine geeignete Abstiegsfunktion ist, also dass `foo` immer terminiert.
Hinweise: Auf Auf) und Def) verzichten wir hier. Weiterhin dürfen Sie folgende Abschätzung verwenden: Wenn $z > 20$ dann gilt auch $(z \text{ `div` } 2) < z - 10$

LÖSUNG: Wir wählen als Abstiegsfunktion $m(x, y, z) = \max(3x + z, 0)$ mit $m : \texttt{Int} \times \texttt{Int} \times \texttt{Int} \to \mathbb{N}$.

Der erste rekursive Aufruf erfolgt für $x > 10$ und $z > 0$. Damit gilt in diesem Fall einfach $\max(3x + z, 0) = 3x + z$ und auch $\max(3(x - 3) + (z + 2), 0) = 3(x - 3) + (z + 2)$. Wir rechnen:

$$\begin{aligned} m(x, y, z) &= \max(3x + z, 0) = 3x + z > 3x + z - 1 = 3x - 3 + z + 23(x - 1) + (z + 2) \\ &= \max(3(x - 1) + (z + 2), 0) = m(x - 1, y + 3, z + 2) \end{aligned}$$

Der zweite rekursive Aufruf erfolgt, für $x > 0$ und $z > 20$. Damit gilt in diesem Fall erneut $\max(3x + z, 0) = 3x + z$ und insbesondere auch $\max(3(x + 3) + z \text{ `div` } 2, 0) = 3(x + 3) + z \text{ `div` } 2$, da bei der ganzzahligen Division einer positiven Zahl größer als 1 wieder eine positive Zahl als Ergebnis herauskommt.

Wir rechnen:

$$\begin{aligned} m(x, y, z) &= \max(3x + z, 0) = 3x + z > 3x + z - 1 = 3x + 9 + z - 10 > 3x + 9 + (z \text{ `div` } 2) \\ &= 3(x + 3) + (z \text{ `div` } 2) = \max(3(x + 3) + (z \text{ `div` } 2), 0) = m(x + 3, y - 1, z \text{ `div` } 2) \end{aligned}$$

Wird der Wert von y in der Abstiegsfunktion berücksichtigt, kann man im Gegensatz zu unserem Lösungsvorschlag hier die max-Funktion nicht mehr so einfach verschwinden lassen!

Lösung Aufgabe 4 (Maybe): **(5 Punkte)**

Gegeben ist folgende Datentypdeklaration aus der Standardbibliothek:

```
data Maybe a = Nothing | Just a
```

a) Implementieren Sie die Funktion `fromMaybe :: a -> Maybe a -> a` zu Fuss, d.h. ohne Verwendung von Funktionen der Standardbibliothek.

Diese Funktion liefert das erste Argument zurück, falls das zweite Argument `Nothing` war; ansonsten wird der im zweiten Argument "verpackte´´ Wert zurückgegeben.

LÖSUNG:

```
fromMaybe x Nothing  = x
fromMaybe _ (Just y) = y
```

b) Implementieren Sie eine Funktion `andMaybe :: Maybe Bool -> Maybe Bool -> Maybe Bool` zu Fuss, d.h. ohne Verwendung von Funktionen der Standardbibliothek; lediglich die Funktion `&& :: Bool -> Bool -> Bool` dürfen Sie verwenden.

Der Typkonstruktor `Maybe` kann als Monade aufgefasst werden. Die volle Punktzahl erhalten Sie für diese Aufgabe nur dann, wenn Ihre Lösung die DO-Notation sinnvoll einsetzt. Die Verwendung von `return :: a -> m a` ist erlaubt.

LÖSUNG: Ohne DO-Notation:

```
andMaybe :: Maybe Bool -> Maybe Bool -> Maybe Bool
andMaybe (Just a) (Just b) = Just (a && b)
andMaybe  _        _       = Nothing
```

Mit DO-Notation:

```
andMaybe :: Maybe Bool -> Maybe Bool -> Maybe Bool
andMaybe ma mb = do a <- ma
                    b <- mb
                    return $ a && b
```

Lösung Aufgabe 5 (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: Diese Aufgabe war schon aus der Probeklausur bekannt.

```
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 6 (Funktionen höherer Ordnung): (5 Punkte)

a) Implementieren Sie zu Fuss, d.h. ohne Verwendung von Funktionen aus der Standardbibliothek, die Infix-Funktion zur Funktionskomposition: `(.) :: (b -> c) -> (a -> b) -> a -> c`
Beispiel: `((+1).(*2)) 3 == 7`

LÖSUNG: Siehe Folie 7-25:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

b) Schreiben Sie eine Funktion `compose :: [a -> a] -> a -> a` welche eine Liste von Funktion von rechts-nach-links auf einen Startwert anwendet:
Beispiel: `compose [(+1),(*2)] 3 == 7`

Sie dürfen alle Funktionen aus der Standardbibliothek einsetzen. Um die volle Punktzahl zu erhalten, dürfen Sie keine direkte Rekursion verwenden! Verwenden Sie stattdessen Funktionen der Standardbibliothek, wie z.B. `foldr :: (a -> b -> b) -> b -> [a] -> b`

LÖSUNG:

```
compose []     x = x
compose [f]    x = f x
compose (f:fs) x = f (compose fs x)


compose = foldr (.) id
```

Lösung Aufgabe 7 (Monaden): **(5 Punkte)**

a) Schreiben Sie eine Funktion `main :: IO ()`, welche eine Zeile von der Tastatur einliest und anschließend rückwärts ausgibt und beendet, also bei Eingabe von `"evil"` wird `"live"` ausgegeben. Sie dürfen alle Funktionen der Standardbibliothek benutzen. DO-Notation ist erlaubt.

LÖSUNG: Die Lösung ist ähnlich zu Übung A9-1:

```
main = do i <- getLine
          putStrLn $ reverse i
```

Wem `reverse` nicht einfällt, kann das auch selbst hinschreiben. Die vielleicht einfachste Möglichkeit wäre:

```
myreverse    [] = []
myreverse (h:t) = myreverse t ++ [h]
```

Vielleicht nicht besonders elegant oder effizient, aber es genügt den Anforderungen der Aufgabe.

b) Implementieren Sie die Funktion `forM :: Monad m => [a] -> (a -> m b) -> m [b]` zu Fuss, d.h. ohne Verwendung von Funktionen der Standardbibliothek. DO-Notation ist erlaubt.

LÖSUNG: Analog zur H9-2, nur das dieses mal das andere Argument als Liste vorliegt:

```
forM    []  _ = return []
forM (x:xs) f = do b  <- f x
                   bs <- forM xs f
                   return (b:bs)
```

Lösung Aufgabe 8 (Typen): **(7 Punkte)**

a) Geben Sie jeweils den allgemeinsten Typ des gegebenen Haskell-Ausdrucks an, inklusive etwaiger Typklassen Einschränkungen. Bitte nur das Ergebnis hinschreiben. Nebenrechnungen bitte deutlich abtrennen.

(i) `fst ('7',(()))`

__ `Char` _____

(ii) `[False] : []`

__ `[[Bool]]` siehe A6a aus der Probeklausur _____

(iii) `(\f x y -> (f y) ++ show x)`

__ `Show a => (b -> String) -> a -> b -> String` _____

b) Geben Sie denn allgemeinsten Unifikator für folgende Typgleichung an:

$\{\alpha \to \beta \to \texttt{Int} = \beta \to \gamma\}$ __ $[\beta/\alpha,\ (\beta \to \texttt{Bool})/\gamma]$ _____

c) Beweisen Sie folgendes Typurteil unter Verwendung der zur Erinnerung auf Seite 9 angegeben Typregeln in einer der beiden in der Vorlesung behandelten Notationen (Herleitungsbaum oder lineare Schreibweise).

(iv) $\Gamma \ \vdash\ \texttt{let}\ (x,y)\ =\ (f,\texttt{True})\ \texttt{in}\ x\ y :: \texttt{Int}$ wobei $\Gamma = \{\texttt{f::Bool -> Int}\}$

LÖSUNG: Als Herleitungsbaum aufgeschrieben:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f\texttt{::Bool -> Int})}\,(\textsc{Var}) \quad \dfrac{}{\Gamma \vdash \texttt{True::Bool}}\,(\textsc{Bool})}{\Gamma\ \vdash\ (f,\texttt{True}) :: (\texttt{Bool -> Int}, \texttt{Bool})}\,(\textsc{Pair-Intro}) \quad \dfrac{\dfrac{}{\Delta \vdash x\texttt{::Bool -> Int}}\,(\textsc{Var}) \quad \dfrac{}{\Delta \vdash y\texttt{::Bool}}\,(\textsc{Var})}{\Delta \vdash x\ y\texttt{::Int}}\,(\textsc{App})}{\Gamma\ \vdash\ \texttt{let}\ (x,y)\ =\ (f,\texttt{True})\ \texttt{in}\ x\ y :: \texttt{Int}}\,(\textsc{Pair-Elim})$$

wobei $\Delta = \{\texttt{f::Bool -> Int},\ x\texttt{::Bool -> Int},\ y\texttt{::Bool}\}$

Typregeln:

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

Alternative Schreibweise für Var:

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

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

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

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

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

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

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

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

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