Prof. Martin Hofmann, PhD
Dr. Steffen Jost

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

9. Musterlösung zur Vorlesung

Programmierung und Modellierung

Hinweise:

- Einmalige Raumänderung für die Übungen am Freitag, den 3.7.2015:
  10:00h→E216, 12:00h→E216, 14:00h→A014 im gleichen Gebäude.
- Donnerstag, 9.7., 17:00, B201: Freiwillige Fragestunde der ProMo-Tutoren!

A9-1 *Hello World* Kreieren Sie mit GHC (nicht GHCI) eine ausführbare Datei, welche nach dem Start eine Begrüßung ausgibt und den Benutzer dazu auffordert, in je eine Zeile nacheinander zuerst ein Lieblingstier und dann eine Lieblingseigenschaft einzugeben. Danach soll das Programm noch einen einzelnen String ausgeben, welcher zuerst die Lieblingseigenschaft und dann das Lieblingstier wiedergibt:

```
> ghc helloTier.hs
...
> ./helloTier
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:
Schildkröte
faule
Psst, willst Du faule Schildkröte kaufen?
```

Hinweis: Das Beispiel zeigt eine Linux-Konsole. Je nach Betriebssystem kann der Aufruf einer ausführbaren Datei abweichen. Die Benutzereingabe waren "Schildkröte" und "faule" in der drittletzten und vorletzten Zeile, der Rest wurde durch das Programm ausgegeben.

LÖSUNGSVORSCHLAG:

```
main = do
  putStrLn "Hi! Gib bitte zuerst Dein Lieblingstier und dann"
  putStrLn "in die nächste Zeile Deine Lieblingseigenschagt ein: "
  tier       <- getLine
  eigenschaft <- getLine
  let ausgabe = "Psst, willst Du " ++ eigenschaft ++ tier ++ " kaufen?"
  putStrLn ausgabe
```

Es ist hier unerheblich, ob der Eingangstext in einer Zeile oder in zwei verschiedenen ausgegeben wird. Es ist auch nicht von Bedeutung, ob die Ausgabe erst mit einem let-Ausdruck unter einem lokalen Bezeichner abgelegt wird oder direkt als Argument an `putStrLn` übergeben wird.

A9-2 *DO-Notation* Versuchen Sie, diese Aufgabe mit Papier und Bleistift zu lösen. Verwenden Sie GHC oder GHCI erst, wenn Sie nicht mehr weiter wissen. Was gibt das folgende Programm am Bildschirm aus? Wie oft wartet das Programm auf eine Benutzereingabe?

Hinweis: Sie dürfen sich selbst ausdenken, was der Benutzer bei jeder Eingabeaufforderung eingibt – alle Eingaben sollten jedoch verschieden sein. Die Aktion `hSetBuffering` (siehe Folie 09-30) können Sie hier ignorieren; diese sorgt nur dafür, dass das Programm auf allen Betriebssystemen gleich funktioniert.

```
import System.IO
main = do hSetBuffering stdout NoBuffering
          putStr "A: "
          a2 <- getLine
          b1 <- putStr "B: "
          let b2 = getLine
          let c1 = putStr "C: "
          c2 <- getLine
          putStr "D: "
          b2 <- b2
          putStrLn $ "A="++a2++" B="++b2++" C="++c2
```

LÖSUNGSVORSCHLAG:

Der Benutzer wird drei Mal aufgefordert, etwas einzugeben. Wir geben der Reihe nach die Zahlen 1,2 und 3 ein:

```
> main1
A: 1
B: 2
D: 3
A=1 B=3 C=2
```

Der Anfang des Programmes sollte inzwischen klar sein: Es wird ein String ausgegeben und dann eine Zeile eingelesen, d.h. bei Eingabe “1” gilt danach `a2="1"`

Die vierte Zeile gibt erneut einen String aus, jedoch wird das Ergebnis des Funktionsaufrufs `putStr "B: "` an die Variable `b1` gebunden. Die Funktion `putStr` liefert jedoch immer nur `()` zurück, d.h. `b1=()` gilt.

In der fünften und sechsten Zeile werden Ausdrücke an lokale Bezeichner `b2` und `c1` gebunden. Das `let` ist eine rein funktionale Abkürzung, d.h. es ist außerhalb der IO-Monade. Weder `getLine` noch `putStr` werden an dieser Stelle dadurch ausgeführt!

Zeile 7 führt die zweite Eingabeaufforderung aus, und bindet das Ergebnis an den lokalen Bezeichner `c2`. Bei Eingabe “2” gilt danach also `c2="2"`.

Zeile 9 führt nun den an `b2` gebundenen Ausdruck innerhalb der Monade aus, d.h. es kommt zu einer Eingabeaufforderung. Das Ergebnis der Eingabe wird an den lokalen Bezeichner `b2` gebunden. Die vorangegangene Definition von `b2` wird dadurch überschattet. Der Rückpfeil der DO-Notation ist im Gegensatz zu `let` nicht rekursiv! Es gilt also bei Eingabe “3” jetzt `b2="3"`.

A9-3 ***Der richtige Kontext*** Berechnen Sie mit Papier & Bleistift für jedes der folgenden Typisierungsurteile einen möglichst *allgemeinen* und *minimalen* Kontext Γ, so dass das entsprechende Typurteil wahr wird. Typklassen ignorieren wir in dieser Aufgabe.

Beispiel: Das Typurteil $\Gamma \vdash x + 1.0 :: \texttt{Double}$ ist z.B. für den Kontext $\Gamma = \{x{::}\texttt{Double}\}$ wahr (welcher auch minimal ist, d.h. keine unnötigen Variablen enthält), nicht aber jedoch für den Kontext $\Gamma = \{y{::}\texttt{Double}, x{::}\texttt{Int}\}$.

a) $\Gamma_a \vdash$ `if x then "Akzeptiert!" else f y` $::$ `String`

LÖSUNGSVORSCHLAG: $\Gamma_a = \{x{::}\texttt{Bool},\ f{::}\alpha \to \texttt{String}, y{::}\alpha\}$

b) $\Gamma_b \vdash$ `\y -> \z -> x z (y z)` $:: (\alpha \to \beta) \to \alpha \to \gamma$

LÖSUNGSVORSCHLAG: $\Gamma_b = \{x{::}(\alpha \to \beta \to \gamma)\}$

c) $\Gamma_c \vdash$ `\x -> if x then \y -> y x else \z -> 1.0` $::\texttt{Bool} \to (\texttt{Bool} \to \texttt{Double}) \to \texttt{Double}$

LÖSUNGSVORSCHLAG: $\Gamma_c = \{\}$ – alle vorkommenden Variablen waren gebunden

GHCI's Typinferenz kann diese Aufgaben auch lösen. *Nachdem* Sie die Aufgabe von Hand berechnet haben können Sie Ihr Ergebnis mit GHCI überprüfen. Was müssen Sie dazu jedoch mit dem Programmausdruck vorher noch tun?

LÖSUNGSVORSCHLAG:

GHCI akzeptiert nur geschlossene Programmausdrücke, d.h. es dürfen keine freien Variablen vorkommen. Wir können aber einfach alle ungebunden Variablen am Anfang des Programmausdrucks mit einen Lambda binden (die Reihenfolge ist dabei egal).

Beispiel: Im Programmausdruck für Teilaufgabe a) haben drei ungebundene Variablen: `x f y`. Wir binden diese mit Lambda und fragen GHCI nach dem Typ des Ausdrucks:

```
> :type \x f y -> (if x then "Akzeptiert!" else f y)
\x f y -> (if x then "Akzeptiert!" else f y) :: Bool -> (t -> [Char]) -> t -> [Char]
```

Der Ausdruck ist also eine Funktion, welche zuerst ein Argument des Typs `Bool` verlangt. Da wir als erste Variable `x` gebunden hatten, ist dies der Typ für `x`. Der Typ für `f` entspricht dem zweiten Argument, also $\alpha \to$ `String`. Der Typ für `y` ist das dritte Argument, welches wieder die gleiche Typvariable ist, welche schon im Typ für `f` verwendet wurde, also α (GHCI nutzt Kleinbuchstaben für Typvariablen, aber wir benutzen griechische Kleinbuchstaben für Typvariablen zu besseren Unterscheidung).

LÖSUNGSVORSCHLAG:
Auch wenn es nicht explizit gefragt war, so ist es eine gute Übung, die vollständige Typherleitung einmal hinzuschreiben:

a)

$$\dfrac{\dfrac{}{\Gamma_a \vdash \texttt{x} :: \texttt{bool}}\,(\textsc{Var}) \quad \dfrac{}{\Gamma_a \vdash \texttt{"Akzeptiert!"} :: \texttt{string}}\,(\textsc{Const}) \quad \dfrac{\dfrac{}{\Gamma_a \vdash \texttt{f} :: \alpha \to \texttt{string}}\,(\textsc{Var}) \quad \dfrac{}{\Gamma_a \vdash \texttt{y} :: \alpha}\,(\textsc{Var})}{\Gamma_a \vdash \texttt{f y} :: \texttt{string}}\,(\textsc{Cond})}{\Gamma_a \vdash \texttt{if x then "Akzeptiert!" else f y} :: \texttt{string}}$$

b)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_b, \texttt{y} : \alpha \to \beta, \texttt{z} : \alpha \vdash \texttt{x} :: \alpha \to \beta \to \gamma}\,(\textsc{Var}) \quad \dfrac{}{\Gamma_b, \texttt{y} : \alpha \to \beta, \texttt{z} : \alpha \vdash \texttt{z} :: \alpha}\,(\textsc{Var})}{\Gamma_b, \texttt{y} : \alpha \to \beta, \texttt{z} : \alpha \vdash \texttt{x z} :: \beta \to \gamma}\,(\textsc{App}) \quad \dfrac{\dfrac{}{\Gamma_b, \texttt{y} : \alpha \to \beta, \texttt{z} : \alpha \vdash \texttt{y} :: \alpha \to \beta}\,(\textsc{Var}) \quad \dfrac{}{\Gamma_b, \texttt{y} : \alpha \to \beta, \texttt{z} : \alpha \vdash \texttt{z} :: \alpha}\,(\textsc{Var})}{\Gamma_b, \texttt{y} : \alpha \to \beta, \texttt{z} : \alpha \vdash \texttt{y z} :: \beta}\,(\textsc{App})}{\Gamma_b, \texttt{y} : \alpha \to \beta, \texttt{z} : \alpha \vdash \texttt{x z (y z)} :: \gamma}\,(\textsc{App})}{\Gamma_b, \texttt{y} : \alpha \to \beta \vdash \texttt{\textbackslash z -> x z (y z)} :: \alpha \to \gamma}\,(\textsc{Abs})}{\Gamma_b \vdash \texttt{\textbackslash y -> \textbackslash z -> x z (y z)} :: (\alpha \to \beta) \to \alpha \to \gamma}\,(\textsc{Abs})$$

c)

$$\dfrac{\dfrac{}{\Gamma_c, \texttt{x} : \texttt{Bool} \vdash \texttt{x} :: \texttt{Bool}}\,(\textsc{Var}) \quad \dfrac{\dfrac{\dfrac{}{\Gamma_c, \texttt{x} : \texttt{Bool}, \texttt{y} : \texttt{Bool} \to \texttt{Double} \vdash \texttt{y} :: \texttt{Bool} \to \texttt{Double}}\,(\textsc{Var}) \quad \dfrac{}{\Gamma_c, \texttt{x} : \texttt{Bool}, \texttt{y} : \texttt{Bool} \to \texttt{Double} \vdash \texttt{x} :: \texttt{Bool}}\,(\textsc{Var})}{\Gamma_c, \texttt{x} : \texttt{Bool}, \texttt{y} : \texttt{Bool} \to \texttt{Double} \vdash \texttt{y x} :: \texttt{Double}}\,(\textsc{App})}{\Gamma_c, \texttt{x} : \texttt{Bool} \vdash \texttt{\textbackslash y -> y x} :: (\texttt{Bool} \to \texttt{Double}) \to \texttt{Double}}\,(\textsc{Abs}) \quad \dfrac{\dfrac{}{\Gamma_c, \texttt{x} : \texttt{Bool}, \texttt{z} : \texttt{Bool} \to \texttt{Double} \vdash \texttt{1.0} :: \texttt{Double}}\,(\textsc{Const})}{\Gamma_c, \texttt{x} : \texttt{Bool} \vdash \texttt{\textbackslash z -> 1.0} :: (\texttt{Bool} \to \texttt{Double}) \to \texttt{Double}}\,(\textsc{Abs})}{\Gamma_c, \texttt{x} : \texttt{Bool} \vdash \texttt{if x then \textbackslash y -> y x else \textbackslash z -> 1.0} :: (\texttt{Bool} \to \texttt{Double}) \to \texttt{Double}}\,(\textsc{Cond})$$

$$\Gamma_c \vdash \texttt{\textbackslash x -> if x then \textbackslash y -> y x else \textbackslash z -> 1.0} :: \texttt{Bool} \to (\texttt{Bool} \to \texttt{Double}) \to \texttt{Double}$$

Abbildung 1: Herleitungen zur A9-3 (waren nicht gefragt)

H9-1 ***Hello Again*** (2 Punkte; Datei `H9-1.hs` als Lösung abgeben)

Ändern Sie Ihre Lösung zu Aufgabe A9-1 wie folgt ab:

a) Falls beide Eingaben leer waren, soll als Antwort nur der String `"Spielverderber!"` ausgeben werden, und danach soll das Programm wieder automatisch von vorne beginnen.

b) Falls nur die Eingabe für das Tier leer war, so beginnt das Programm ebenfalls von vorne, aber merkt sich heimlich die eingegebene Lieblingseigenschaft. Wenn danach mal Tier und Eigenschaft komplett eingegeben werden, wird die komplette Liste aller zuvor eingegeben Eigenschaften ausgeben.

Beispiel:

```
> ./helloTier3
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:


Spielverderber!
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:

tolle
Tier eingeben!
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:

schnelle
Tier eingeben!
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:
Kröte
grüne
Psst, willst Du grüne schnelle tolle Kröte kaufen?
```

Hinweis: Für die erste Teilaufgabe könnte Ihnen Folie 09-20 die notwendige Inspiration liefern. Für die zweite Teilaufgabe muss man vielleicht etwas nachdenken. Wir verraten nur so viel: die Lösung benötigt keineswegs irgendwelche monadischen Tricks; es reicht ein gewöhnlicher funktionaler Akkumulator.

LÖSUNGSVORSCHLAG:

Die erste Teilaufgabe könnte man so lösen:

```
main :: IO ()
main = do
  putStrLn "Hi! Gib bitte zuerst Dein Lieblingstier und dann"
  putStrLn "in die nächste Zeile Deine Lieblingseigenschaft ein: "
  tier        <- getLine
  eigenschaft <- getLine
  if null tier && null eigenschaft
    then do
      putStrLn "Spielverderber!"
      main
    else
      putStrLn $ "Psst, willst Du " ++ eigenschaft ++ " " ++ tier ++ " kaufen?"
```

Um auch noch die zweite Teilfaufgabe zu lösen, setzen wir einen gewöhnlichen Akkumulator ein, welcher die bisher eingegeben Eigenschaften aufsammelt. Dazu müssen wir aber den gesamten Code in eine Hilfsfunktion `mainAkkum` auslagern. Das wir anstatt eines verschatelten

if-then-else hier auch noch eine lokale Hilfsfunktion `checkInputs` einsetzen hat damit aber nichts zu tun – letzteres ist eine reine Stilfrage (genauso auch der Einsatz von `>>` anstatt `do`).

```
main :: IO ()
main = mainAkkum []

mainAkkum :: [String] -> IO ()
mainAkkum ps = do
    putStrLn "Hi! Gib bitte zuerst Dein Lieblingstier und dann"
    putStrLn "in die nächste Zeile Deine Lieblingseigenschaft ein: "
    tier       <- getLine
    eigenschaft <- getLine
    checkInputs tier eigenschaft
  where
    checkInputs "" "" = putStrLn "Spielverderber!" >> mainAkkum    ps
    checkInputs "" e  = putStrLn "Tier eingeben!"  >> mainAkkum (e:ps)
    checkInputs t  e  =
      let props = concat $ map (' ':) (e:ps) in -- oder Data.List.intersperse
      putStrLn $ "Psst, willst Du" ++ props ++ " " ++ t ++ " kaufen?"
```

H9-2 ***Typregel für Case*** (3 Punkte; Abgabeformat: Text oder PDF)
Erstellen Sie eine neue Typregel für Case-Ausdrücke! Zur Vereinfachung der Aufgabe hat unser Case-Ausdruck immer genau zwei Fälle, und die Patterns sind auch festgelegt auf `Nothing` im ersten Mustervergleich und `Just x` im zweiten. Der Ausdruck hat also immer die Form `case` e_1 `of` {`Nothing ->` e_2; `Just` x `->` e_3} wobei die e_1, e_2, e_3 und x hier Metavariablen sind. Überlegen Sie sich, welche Anforderungen Haskell an diesen Ausdruck stellt und formalisieren Sie diese durch eine Typregel!

LÖSUNGSVORSCHLAG:

1. Prämisse: Da der Mustervergleich mit `Nothing` und `Just x` stattfindet, muss der gematchte Teilausdruck e_1 den Typ `Maybe ?` sein. Anstatt ? setzen wir eine frische Metavariable B dafür ein; eine echte Typvariable β wäre aber falsch, denn der Ausdruck muss ja nicht polymorph sein, sondern darf irgendeinen beliebigen, aber fest gewählten Typ haben.

2. Prämisse: Der Teilausdruck e_2 muss lediglich vom gleichen Typ wie der Gesamtausdruck sein.

3. Prämisse: Auch der Teilausdruck e_3 muss den gleichen Typ wie der Gesamtausdruck haben. Allerdings darf in e_3 nun die Variable x frei vorkommen, da diese durch den Mustervergleich eingeführt (gebunden) wird. Die Typisierung von e_3 hat also eine zusätzliche Annahme im Typkontext, wobei x vom gleichen Typ B ist, wie in der ersten Prämisse.

$$\frac{\Gamma \vdash e_1 \texttt{ of Maybe } B \qquad \Gamma \vdash e_2 \texttt{ of } A \qquad \Gamma, x{::}B \vdash e_3 \texttt{ of } A}{\Gamma \vdash \texttt{ case } e_1 \texttt{ of } \{\texttt{Nothing -> } e_2; \texttt{Just } x \texttt{ -> } e_3\}{::}A} \quad (\textsc{Case})$$

H9-3 ***Typfehler*** (4 Punkte; Abgabeformat: Text oder PDF)
Geben Sie eine Herleitung (Baum- oder lineare Notation) für jedes der folgenden Typurteile an, falls möglich. Anderfalls begründen Sie, warum eine Typherleitung nicht möglich ist.

a) $\{f{::}\gamma \to \alpha \to \beta,\ g{::}\beta \to \gamma,\ x{::}\beta\} \vdash f\ g\ x :: \alpha \to \beta$

LÖSUNGSVORSCHLAG:

Wir fügen zuerst die impliziten Klammern ein, welche hier den Kern des Problems berühren:

(1)	$\{f{::}\gamma \to (\alpha \to \beta), g{::}\beta \to \gamma, x{::}\beta\} \vdash (f\ g)\ x :: \alpha \to \beta$	App(2, 3)
(2)	$\{f{::}\gamma \to (\alpha \to \beta), g{::}\beta \to \gamma, x{::}\beta\} \vdash (f\ g) :: \beta \to (\alpha \to \beta)$	App(4, 5)
(3)	$\{f{::}\gamma \to (\alpha \to \beta), g{::}\beta \to \gamma, x{::}\beta\} \vdash x :: \beta$	Var
(4)	$\{f{::}\gamma \to (\alpha \to \beta), g{::}\beta \to \gamma, x{::}\beta\} \vdash f :: (\beta \to \gamma) \to (\beta \to (\alpha \to \beta))$	$\lightning$
(5)	$\{f{::}\gamma \to (\alpha \to \beta), g{::}\beta \to \gamma, x{::}\beta\} \vdash g :: \beta \to \gamma$	Var

Die Applikation in Zeile 2 erfordert, dass f eine Funktion mit Argument g und Rückgabetyp $\beta \to (\alpha \to \beta)$. Der Kontext gibt dies aber nicht her: entweder passt der Typ von f nicht (hier dargestellt), oder der Typ von g passt nicht.

Wir bemerken, dass das leicht andere Typurteil

$$\{f{::}\gamma \to (\alpha \to \beta), g{::}\beta \to \gamma, x{::}\beta\} \vdash f\ (g\ x) :: \alpha \to \beta$$

dagegen schon eine Herleitung besitzt!

b) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta\} \vdash \backslash y \texttt{ -> } \backslash x \texttt{ -> if } f\ x \texttt{ then } z \texttt{ else } y\ z :: (\beta \to \beta) \to \alpha \to \beta$

LÖSUNGSVORSCHLAG:

(6) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta\} \vdash$
$\backslash y \texttt{ -> } \backslash x \texttt{ -> if } f\ x \texttt{ then } z \texttt{ else } y\ z :: (\beta \to \beta) \to \alpha \to \beta$ ABS(7)

(7) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta\} \vdash$
$\backslash x \texttt{ -> if } f\ x \texttt{ then } z \texttt{ else } y\ z :: \alpha \to \beta$ ABS(8)

(8) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash$
$\texttt{if } f\ x \texttt{ then } z \texttt{ else } y\ z :: \beta$ COND(9, 10, 11)

(9) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash f\ x :: \texttt{Bool}$ APP(12, 13)

(10) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash z :: \beta$ VAR

(11) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash y\ z :: \beta$ APP(14, 15)

(12) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash f :: \alpha \to \texttt{Bool}$ VAR

(13) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash x :: \alpha$ VAR

(14) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash y :: \beta \to \beta$ VAR

(15) $\{f{::}\alpha \to \texttt{Bool}, z{::}\beta, y{::}\beta \to \beta, x{::}\alpha\} \vdash z :: \beta$ VAR

c) $\{\,\} \vdash \backslash x \texttt{ -> } (\backslash f \texttt{ -> } f\ x\ x) :: \alpha \to (\alpha \to \alpha) \to \alpha$

LÖSUNGSVORSCHLAG:

(16)	$\{\,\} \vdash \backslash x \texttt{ -> } (\backslash f \texttt{ -> } (f\ x)\ x) :: \alpha \to (\alpha \to \alpha) \to \alpha$	Abs(17)
(17)	$\{x{::}\alpha\} \vdash \backslash f \texttt{ -> } (f\ x)\ x :: (\alpha \to \alpha) \to \alpha$	Abs(18)
(18)	$\{x{::}\alpha, f{::}\alpha \to \alpha\} \vdash (f\ x)\ x :: \alpha$	App(19, 20)
(19)	$\{x{::}\alpha, f{::}\alpha \to \alpha\} \vdash f\ x :: \alpha \to \alpha$	App(21, 22)
(20)	$\{x{::}\alpha, f{::}\alpha \to \alpha\} \vdash x :: \alpha$	Var
(21)	$\{x{::}\alpha, f{::}\alpha \to \alpha\} \vdash f :: \alpha \to (\alpha \to \alpha)$	$\lightning$
(22)	$\{x{::}\alpha, f{::}\alpha \to \alpha\} \vdash x :: \alpha$	Var

Die Typherleitung schlägt fehl, da f eine Funktion mit Stelligkeit 1 ist, aber auf zwei Argumente angewendet wird. Prinzipiell kann man bei der Applikation in Zeile 18 für ein beliebiges B den Typ $B \to \alpha$ für 19 wählen, aber dann schlägt schon 20 fehl.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 30.06.2015, 11:00 Uhr mit UniworX abgegeben werden. Abschreiben ist verboten!