Dr. Steffen Jost
Leah Neukirchen

Ludwig-Maximilians-Universität München
Institut für Informatik
12. Juni 2018

Lösungsvorschlag zur 08. Übung zur Vorlesung

Programmierung und Modellierung

A8-1 ***Typsignaturen*** Lösen Sie diese Aufgabe mit Papier und Bleistift! Geben Sie zu jeder der folgenden Deklaration eine möglichst allgemeine Signatur an. Begründen Sie Ihre Antwort informell!

Überlegen Sie sich dazu, welche Argumente jeweils auftreten und wie diese verwendet werden, z.B. welche Funktion auf welches Argument angewendet wird. Falls Sie eine der verwendeten Funktion nicht kennen, schlagen Sie diese in den Vorlesungsfolien oder der Dokumentation der Standardbibliothek nach. Verwenden Sie GHCI höchstens im Nachhinein, um Ihre Antwort zu kontrollieren. Numerische Typklassen müssen Sie zur Vereinfachung nicht angeben, verwenden Sie einfach einen konkreten Typ wie `Int` oder `Double`.

a) `gaa xy z vw = if (read z == xy) then minBound else vw`

Lösungsbeispiel: Wir sehen als erstes, dass es sich um eine Funktion mit drei Argumenten handelt, deren Typ wir bestimmen müssen, sowie den Ergebnistyp der Funktion. Nennen wir diesen Typ vorläufig einmal $a \to b \to c \to d$, also `xy` $:: a$, `z` $:: b$ und `vw` $:: c$.

Argument `xy` wird in einem Vergleich verwendet, also muss a in der Typklasse `Eq` sein.

`z` ist ein Argument für die Funktion `read` und damit ist $b =$ `String`. Da das Ergebnis aber mit `xy` verglichen wird, muss es den gleichen Typ haben, und wir wissen also, dass a auch in der Typklasse `Read` sein muss, denn es wurde ja ein Wert dieses Typs geparsed.

`vw` wird nur als Ergebnis verwendet, daraus können wir $c = d$ schließen. Da die beiden Zweige eines Konditionals den gleichen Typ haben müssen und im `then`-Zweig der Wert `minBound` zurückgegeben wird, muss dieser Typ auch noch in der Klasses `Bounded` sein.

Wir haben also `gaa :: (Eq a, Read a, Bounded c) => a -> String -> c -> c`

Hinweis: Namen von Typvariablen und Reihenfolge der Class Constraints ist unbedeutend. `gaa :: (Read y2, Bounded zz, Eq y2) => y2 -> [Char] -> zz -> zz` wäre z.B. eine äquivalente Typsignatur.

b)
```
guu _ (h1:h2:t) = [h2]
guu x _         = show x
```

LÖSUNGSVORSCHLAG:

```
guu :: Show a => a -> String -> String
```

Die zweite Zeile legt wegen `show :: Show a => a -> String` Rückgabewert auf `String`fest und schränkt den Typ des ersten Argumentes auf die Typklasse `Show` ein. Damit `[h2]` vom Typ `String` ist, muss `h2` vom Typ `Char` sein. `h2` ist aber das zweite Element der Liste, welche wir als zweites Argument erhalten haben, also ist diese eine Liste von Zeichen.

c)
```
foo a b []    e f = show a
foo a b (c:d) e f | a==c, read b  = e ++ e
                  | a/=c          = show d
                  | otherwise     = show e
```

LÖSUNGSVORSCHLAG:

```
foo :: (Eq x, Show x) => x -> String -> [x] -> String -> y -> String
```

Da auf das erste Argument `a` die Funktion `show` angewendet wird, muss es irgendein Typ sein, der in der Typklasse `Show` liegt. Da wir `show a` zurückgegeben, muss der Ergebnistyp der Funktion `foo` schon mal `String` sein.

Das zweite Argument muss wegen der Anwendung von `read` ein String sein. (Das Ergebnis dieses `read`-Aufrufs ist für uns hie unerheblich – wir wissen aber, dass es von dem Typ `Bool` sein muss.)

Das dritte Argument wird mit Listen-Pattern gemacht, es ist also irgendein Listentyp, sagen wir einfach mal `[x]`. Damit ist die Variable `c` vom Typ `x` und die Variable `d` vom Typ `[x]`. Aufgrund des Vergleiches des ersten Argumentes mit der Variable `c` in Guards wissen wir, das beide den gleichen Typ besitzen, welche zusätzlich noch in der Typklasse `Eq` liegen muss. (Zusätzlich zu der Klasse `Show` wie wir bereits für das erste Argument festgestellt hatten.)

Auf das vierte Argument wird die Infix-Funktion `(++)` angewendet. Diese hat den Typ `[a]->[a]->[a]`. Da wir bereits wissen, dass der Ergebnistyp `String`, also `[Char]` ist, so muss auch das vierte Argument vom Typ `String` sein.

Das fünfte Argument wird nirgendwo verwendet. Es kann daher irgendeinen beliebigen Typ haben, der nicht auf eine Typklasse eingeschränkt werden muss.

d)
```
bar a b c d | b >= c    = bar b a d c
           | otherwise = a==d
```

LÖSUNGSVORSCHLAG:

```
bar :: (Ord a, Ord b) => a -> b -> b -> a -> Bool
```

Im ersten Zweig wird die Ordnung des zweiten und dritten Argumentes geprüft. Damit müssen diese beide Argumente dem gleichen Typ angehören, welcher der Typklasse `Ord` angehören muss.

Der rekursive Aufruf verrät uns leider gar nichts über den Rückgabetyp der Funktion. Allerdings sehen wir für den rekursiven Aufruf das jeweils das erste und zweite, und das dritte und vierte Argument vertauscht werden. GHC schließt daraus schon, dass alle Typen gleich sein müssen, doch dass stimmt nicht: Der Aufruf kann ja auch polymorph sein - es reicht wenn das erste und vierte Argument einem anderen Typ angehören, der den gleichen Einschränkung unterliegt, in diesem Falle also auch der Typklasse `Ord` angehört.

Der zweite Zweig verrät uns den Rückgabetyp: `Bool`, da dies der Ergebnistyp der verwendeten Funktion `(==)` ist. Die explizite Einschränkung der Argumenttypen auf die Typklasse `Eq` können wir uns hier sparen, da `Ord` eine Unterklasse der Typklasse `Eq` ist - was wir ordnen können, können wir auch vergleichen.

Hinweis: Der rekursive Aufruf in `bar` benötigt eine andere Instantiierung des polymorphen Typs (die Argumente tauschen Ihre Positionen, d.h. Typen a und b vertauschen sich bei jedem rekursiven Aufruf). Die nennt man *polymorphe Rekursion*.

Es ist bewiesen, dass Typinferenz für polymorph rekursive Funktionen unentscheidbar ist. GHC kann den oben angegebenen Typ daher nicht inferieren. Allerdings kann GHC eine Typsignatur für eine polymorph rekursive Funktion sehr wohl überprüfen! D.h. wenn wir die Typsignatur explizit im Quellcode angeben, dann prüft und akzeptiert GHC diese auch.

e)
```
gnarf [] _ = mempty
gnarf x  y = y `mappend` x `gnarf` y
```

LÖSUNGSVORSCHLAG:

```
gnarf :: Monoid p => [a] -> [a] -> p
```

Betrachten wir die erste Definition: `mempty` erzwingt, dass der Rückgabewert irgendein Monoid ist. Der Pattern-Match mit `x` erzwingt, dass das erste Argument eine List sein muss.

Nun zur zweiten definierenden Gleichung: Da Funktionsanwendungen implizit links-geklammert sind, haben wir folgenden Rumpf im rekursiven Fall: `(y `mappend` x) `gnarf` y`. Da wir bereits wissen, dass `x` eine Liste ist, instantiiert `mappend` hier zu `(++)`, hat also nichts mit dem Monoid aus dem Rückgabewert zu tun! Damit müssen `x` und `y` den gleichen Typ haben, eine generische Liste.

Da `gnarf` i.A. nicht terminiert, wird der Rückgabewert durch die zweite Zeile nicht weiter eingeschränkt, d.h. es bleibt bei einer neuen Typvariable der Typklasse `Monoid` (aus der ersten Zeile gefolgert). (Vergleiche auch `error :: String -> a`, welches einen Wert eines beliebigen Typs verspricht, aber nie liefert.)

A8-2 ***Freie Variablen und Substitution***

a) Berechnen Sie jeweils die Menge der freien Variablen folgender Terme:

i) `(p q) r` $\{p, q, r\}$

ii) `(\a -> b a) (\c -> (d c) e)` $\{b, d, e\}$

iii) `(\x -> x) (\y -> y) (\z -> z)` Leere Menge; keine freien Variablen

iv) `(\u -> v) (\v -> u) (\w -> w)` $\{v, u\}$

v) `(\f -> (\g -> (\h -> (h f) i))) (g j)`
$\{g, i, j\}$, Variable g kommt auch noch gebunden vor

b) Berechnen Sie jeweils folgende Termsubstitutionen:

i) `(\a -> b a)[x/a, y/b]` Ergebnis: `\a -> y a`

ii) `((\x -> (\y -> x (y z))) x)[a/x,b/y,(\c -> d)/z]`
Ergebnis: `((\x -> (\y -> x (y (\c -> d)))) a)`

iii) `p (q r)[q/r,p/t,s/q]` Ergebnis: `p (s s)`

iv) `(\u -> (\v -> u w))[(u v)/w]`
Ergebnis: `(\a -> (\b -> a (u v)))`

Tipp: Kontrollieren Sie Ihre Antworten mithilfe Ihrer Lösung zur H8-1.

A8-3 ***Typherleitung I***

a) Erstellen Sie eine Typherleitung in Baum-Notation für folgendes Typurteil:

$$\{\,\} \vdash (\backslash x \texttt{ -> } (\backslash y \texttt{ -> } (\backslash z \texttt{ -> } x\ z\ y))) :: (\alpha \to \beta \to \delta) \to \beta \to \alpha \to \delta$$

Zusatzfrage: Welchen Namen hat diese Funktion in der Standardbibliothek von Haskell?

LÖSUNGSVORSCHLAG:

Zur Vermeidung von Flüchtigkeitsfehlern empfiehlt es sich, hier erst einmal alle impliziten Klammern explizit einzufügen, siehe auch Folie 1.50.

$$\dfrac{\dfrac{\dfrac{\dfrac{x{::}\alpha \to \beta \to \delta \in \Delta}{\Delta \vdash x :: \alpha \to (\beta \to \delta)}\,\textsc{Var} \qquad \dfrac{z{::}\alpha \in \Delta}{\Delta \vdash z :: \alpha}\,\textsc{Var}}{\Delta \vdash x\ z :: \beta \to \delta}\,\textsc{App} \qquad \dfrac{y{::}\beta \in \{x{::}\alpha \to \beta \to \delta, y{::}\beta, z{::}\alpha\}}{\{x{::}\alpha \to \beta \to \delta, y{::}\beta, z{::}\alpha\} \vdash y :: \beta}\,\textsc{Var}}{\{x{::}\alpha \to \beta \to \delta, y{::}\beta, z{::}\alpha\} \vdash (x\ z)\ y :: \delta}\,\textsc{App}}{\{x{::}\alpha \to \beta \to \delta, y{::}\beta\} \vdash \backslash z \texttt{ -> } (x\ z)\ y :: \alpha \to \delta}\,\textsc{Abs}}{\{x{::}\alpha \to \beta \to \delta\} \vdash \backslash y \texttt{ -> } (\backslash z \texttt{ -> } (x\ z)\ y) :: \beta \to \alpha \to \delta}\,\textsc{Abs}}{\{\,\} \vdash \backslash x \texttt{ -> } (\backslash y \texttt{ -> } (\backslash z \texttt{ -> } (x\ z)\ y)) :: (\alpha \to (\beta \to \delta)) \to \beta \to \alpha \to \delta}\,\textsc{Abs}$$

wobei wir aus Platzgründen die Abkürzung $\Delta := \{x{::}\alpha \to \beta \to \delta, y{::}\beta, z{::}\alpha\}$ für einen Typkontext definieren und im oberen linken Teilbaum verwenden.

Der Programmausdruck beschreibt die Funktion `flip`.

b) Erstellen Sie eine Typherleitung in linearer Notation für folgendes Typurteil:

$$\{x{::}\texttt{Bool}, y{::}\texttt{Double} \to \texttt{Int}\} \vdash (\backslash z \texttt{ -> } z\ x\ 4\ (y\ 7)) :: (\texttt{Bool} \to \texttt{Int} \to \texttt{Int} \to \beta) \to \beta$$

Hinweis: Beachten Sie dabei die übliche Klammerkonventionen für Funktionstypen und Funktionsanwendung!

LÖSUNGSVORSCHLAG:

Aus Platzgründen definieren wir

$$\Gamma := \{x{::}\texttt{Bool}, y{::}\texttt{Double} \to \texttt{Int}, z{::}\texttt{Bool} \to (\texttt{Int} \to (\texttt{Int} \to \beta))\}$$

in der folgenden Typherleitung im linearen Stil – natürlich können wir diese Abkürzung erst definieren, wenn wir später während der Konstruktion der Typherleitung wissen was wir wirklich brauchen, d.h. bei einer Herleitung mit Papier & Bleistift müssten die Abkürzung unten auf dem Blatt niederschreiben (oder anfangs oben etwas Platz lassen). Wir beginnen damit, in dem wir Programmausdruck und Typausdruck vollständig klammern – das ist nicht notwendig, macht aber klarer was zu tun ist.

$\{x{::}\texttt{Bool}, y{::}\texttt{Double} \to \texttt{Int}\} \vdash \backslash z \texttt{ -> } ((z\ x)\ 4)\ (y\ 7)$

(1)	$:: (\texttt{Bool} \to (\texttt{Int} \to (\texttt{Int} \to \beta))) \to \beta$	Abs(2)
(2)	$\Gamma \vdash ((z\ x)\ 4)\ (y\ 7) :: \beta$	App(3, 4)
(3)	$\Gamma \vdash (z\ x)\ 4 :: \texttt{Int} \to \beta$	App(5, 6)
(4)	$\Gamma \vdash y\ 7 :: \texttt{Int}$	App(9, 10)
(5)	$\Gamma \vdash z\ x :: \texttt{Int} \to (\texttt{Int} \to \beta)$	App(7, 8)
(6)	$\Gamma \vdash 4 :: \texttt{Int}$	Const
(7)	$\Gamma \vdash z :: \texttt{Bool} \to (\texttt{Int} \to (\texttt{Int} \to \beta))$	Var
(8)	$\Gamma \vdash x :: \texttt{Bool}$	Var
(9)	$\Gamma \vdash y :: \texttt{Double} \to \texttt{Int}$	Var
(10)	$\Gamma \vdash 7 :: \texttt{Double}$	Cons

c) Beweisen Sie durch eine saubere Typherleitung in der Notation Ihrer Wahl, dass die folgende Haskell Funktion den behaupteten Typ hat:

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Hinweis: Da das in der Vorlesung behandelte Typsystem nicht mit benannten Funktionen umgehen kann, müssen wir zuerst den Funktionsrumpf in eine äquivalenten Term übersetzen, welcher eine anonyme Funktionsdefinition mit Lambda benutzt.

LÖSUNGSVORSCHLAG:

Wir müssen den Funktionsrumpf in einen geschlossenen Programmausdruck verwandeln. Die beiden Argumente der Funktion `twice` müssen also mit Lambdas gebunden werden. Dies können wir auch in Haskell so hinschreiben:

```
twice :: (a -> a) -> a -> a
twice = \f -> \x -> f (f x)
```

Damit können wir nun ganz normal unsere Typherleitung basteln. Wir wählen hier die Baum-Notation, da diese sich nach unserer Ansicht vielleicht etwas schwieriger hinschreiben läßt, dafür aber leichter zu lesen ist, so fern der Baum auf eine Seite passt.

$$\dfrac{\dfrac{\dfrac{f \in \{f{::}\alpha \to \alpha, x{::}\alpha\}}{\{f{::}\alpha \to \alpha, x{::}\alpha\} \vdash f :: \alpha \to \alpha}\,\textsc{Var} \qquad \dfrac{f \in \{f{::}\alpha \to \alpha, x{::}\alpha\}}{\{f{::}\alpha \to \alpha, x{::}\alpha\} \vdash f :: \alpha \to \alpha}\,\textsc{Var} \qquad \dfrac{x \in \{f{::}\alpha \to \alpha, x{::}\alpha\}}{\{f{::}\alpha \to \alpha, x{::}\alpha\} \vdash x :: \alpha}\,\textsc{Var}}{\{f{::}\alpha \to \alpha, x{::}\alpha\} \vdash f\ x :: \alpha}\,\textsc{App}}{\{f{::}\alpha \to \alpha, x{::}\alpha\} \vdash f\ (f\ x) :: \alpha}\,\textsc{App}$$

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

Ende der Lösungsvorschläge für die Präsenzaufgaben.
Lösungsvorschläge für die Hausaufgaben folgen nach Ende der Abgabezeit.

H8-1 ***Substitution*** (2 Punkte; Datei `H8-1.hs` als Lösung abgeben)
Gegeben ist folgende Datentypdeklaration zur Repräsentation von Termen:

```
data Term = Var Char | Const Int | App Term Term | Abs Char Term
```

a) Schreiben Sie eine Funktion `freeVars :: Term -> [Char]` welche die freien Variablen eines Terms effizient berechnet. *Hinweis:* Die Aufgabe ist sehr einfach, wenn Sie den Unterschied zwischen freien und gebundenen Variablen verstanden haben, siehe dazu auch Folien 8.10

b) Implementieren Sie die Substitution von Folie 8.9 effizient als Funktion `subst :: (Char, Term) -> Term -> Term`. `subst ('x', t1) t2` berechnet den Term, den man erhält wenn man in `t2` alle freien Vorkommen von `Var 'x'` durch `t1` ersetzt.

Verwenden Sie zur Vereinfachung folgende partielle Funktion zur Erzeugung frischer Variablen:

```
genFreshV :: [Char] -> Char
genFreshV vs = head $ filter (\c -> not $ c `elem` vs) ['a'..'z']
```

H8-2 ***Typherleitung II*** (2 Punkte; Abgabe: `H8-2.txt` oder `H8-2.pdf`)
Erstellen Sie eine Typherleitung für folgendes Typurteil:

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

Sie dürfen sich aussuchen, ob Sie die Herleitung in Baum-Notation oder in linearer Notation verfassen.

H8-3 ***Unifikation*** (2 Punkte; Abgabe: `H8-3.txt` oder `H8-3.pdf`)

a) Berechnen Sie den allgemeinsten Unifikator für folgende Menge von Typgleichungen:

$$\{\texttt{Int} \to \alpha = \beta \to \texttt{Bool} \to \beta, \gamma \to \texttt{Int} = \alpha\}$$

b) Gegeben ist folgende Menge von Typgleichungen:

$$\alpha \to \beta = \delta \to \gamma, \beta \to \delta = \theta \to \eta, \eta = \alpha$$

Geben Sie eine nicht-leere Substitution an, welche alle Variablen durch konkrete Typen ersetzt, aber welche kein Unifikator für die Gleichungsmenge ist.

c) Gegeben ist folgende Menge von Typgleichungen:

$$\eta \to \beta = \delta \to \theta, \alpha = \eta, \alpha \to \beta = \delta \to \gamma$$

Geben Sie einen Unifikator für die Gleichungsmenge an, welche nicht der allgemeinste Unifikator ist.

Abgabe: Lösungen zu den Hausaufgaben können bis Samstag, den 23.6.18, mit UniWorX nur als `.zip` abgegeben werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bis zu 3 Studierende können gemeinsam als Gruppe abgeben. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ss-2018/promo/`).