Woche 6 – Übersicht, Tricks & Aufgaben

(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

Anmerkungen zu den Code-Expert Abgaben:

- Die Abgaben, die ich bekommen habe, sahen insgesamt sehr gut aus.
- Der Vollständigkeit halber hier nun noch meine Implementation der BFS-Funktion (es geht eleganter 😉):

```haskell
data Tree t = Leaf | Node t (Tree t) (Tree t)

bfs :: Tree t -> [t]
bfs (Leaf) = []
bfs (Node t l r) = aux [(Node t l r)] []
  where
    aux [] fin = fin
    aux ((Node t l r):xs) fin = aux (xs ++ [l,r]) (fin ++ [t])
    aux ((Leaf):xs) fin = aux xs fin
```

Weiteres:

- Wichtig: Ich habe auf meiner letzten Zusammenfassung den Fehler gemacht, Type-Variablen (wenn man “mehr über sie herausfindet”) so zu schreiben: „a :: b -> c“ etc. bitte verwendet an dieser Stelle ein „=“ und den Doppel-Doppelpunkt nur wenn die ihr die Types von konkreten Funktionen bestimmt, bspw. „f :: b -> c“.

Weiter auf der nächsten Seite →

Haskell-Type Inference und Higher Order Programming Beispiel 1:

Da ich sehe, dass ihr euch insgesamt immer noch ein wenig mit dem Konzept Higher Order Programming im Kontext Haskell-Type Inference schwer tut, versuche ich hier nun noch einmal in eigenen Worten das Wichtigste zu erklären:

Betrachten wir dazu das folgende konkrete Beispiel:

$$(\backslash x \rightarrow x\ (<)) \qquad \text{geg.:}\ \ (<) :: Ord\ c \Rightarrow c \rightarrow c \rightarrow Bool$$

Betrachten wir erst einmal die Grobe Struktur der gegebenen Funktion: $(\backslash x \rightarrow x\ (<))$ nimmt als „Input“ (was „links vom Pfeil steht“) ein x und gibt als „Output“ (was „rechts vom Pfeil steht“) die Expression $x\ (<)$ zurück. Das heisst, der Typ von $(\backslash x \rightarrow x\ (<))$ muss irgendwie so aussehen:

$$(\backslash x \rightarrow x\ (<)) :: Type_{Input} \rightarrow Type_{Output}$$

Jetzt müssen wir bestimmen, was genau $Type_{Input}, Type_{Output}$ sind. Dafür gucken wir uns an, was in der Funktion passiert (also die „rechte Seite vom Pfeil“):

$$x(<)$$

Es liegt also eine Function-Application vor: x bekommt das Argument $(<)$. Wir wissen von der Function-Application, dass sie diese Form hat: $t_1 t_2$ wobei t_1 eine Funktion ist, die t_2 als ihr Argument in der Function-Application nimmt (Bspw. ist bei „(2+) 4“ die Funktion „(2+)“ die das Argument „4“ nimmt). Also **muss** x eine Funktion sein, die irgendeinen Input nimmt und irgendeinen Output zurückgibt. Also:

$$\boldsymbol{x :: a \rightarrow b}$$

Wir wissen aber sogar schon ein wenig mehr! Wir wissen, was für einen „Input“ x nehmen kann: In der Function-Application „auf der rechten Seite vom Pfeil“ wird der Funktion x das Argument $(<)$ übergeben, also nimm x Argumente wie $(<)$ als Input. Also ist x eine Funktion, die „Inputs die so sind wie „$(<)$“ als Argument nimmt und irgendwas zurückgibt“. Weil wir uns schon ein bisschen besser mit Haskell auskennen, können den roten Satz jetzt aber schon relativ gut formal ausdrücken:

x ist eine Funktion, die Argumente vom Typen von der Funktion $(<)$ nimmt und irgendwelche Rückgabewerte zurückgibt. So – und was ist der Typ von der Funktion $(<)$?

Den kennen wir schon: $(<) :: Ord\ c \Rightarrow c \rightarrow c \rightarrow Bool$

Also nimmt x Argumente vom Type $Ord\ c => c \rightarrow c \rightarrow Bool$ und gibt Rückgabewerte von irgendeinem – uns nicht näher bekannten – Wert zurück. Gucken wir uns noch einmal kurz an, wie wir eingangs den Typen von x definiert haben: $\boldsymbol{x :: a \rightarrow b}$. Nun wissen wir also, was a ist, da wir herausgefunden haben, welchen Typ die Argumente von x haben:

$$\boldsymbol{a \quad = \quad Ord\ c \Rightarrow c \rightarrow c \rightarrow Bool}$$

Also können wir jetzt genauer definieren, welchen Type x hat:

$$\boldsymbol{x :: a \rightarrow b = Ord\ c \Rightarrow (c \rightarrow c \rightarrow Bool) \rightarrow b}$$

Gucken wir uns nun abschließend noch einmal $(\backslash x \rightarrow x\ (<))$ an: Wir haben eingangs festgestellt, dass: $(\backslash x \rightarrow x\ (<)) :: Type_{Input} \rightarrow Type_{Output}$. Wir wissen, dass x der Input und $x(<)$ der Output sind. Den Type vom Input kennen wir jetzt schon: $\boldsymbol{x :: Ord\ c \Rightarrow (c \rightarrow c \rightarrow Bool) \rightarrow b}$. Jetzt müssen wir nur noch den Type vom Output, also von $x\ (<)$ bestimmen: Dazu gucken wir uns noch einmal x an: Die Funktion x nimmt Argumente vom Typ $\boldsymbol{Ord\ c \Rightarrow (c \rightarrow c \rightarrow Bool)}$ und gibt

Argumente vom Type $\boldsymbol{b}$ zurück. Hier, auf der „rechten Seite vom Pfeil“ bei unserer Funktion $(\backslash \boldsymbol{x} \rightarrow \boldsymbol{x}\ (<))$ übergeben wir an $\boldsymbol{x}$ die Funktion $(<)$ – die den Typen $\boldsymbol{Ord\ c \Rightarrow (c \rightarrow c \rightarrow Bool)}$ hat!

Da ein Argument gegeben wurde, gibt es eine Rückgabe vom Type $\boldsymbol{b}$, also hat die Funktion Application $x\ (<)$ den Type $\boldsymbol{b}$.

Also gilt für unsere Funktion:

$$(\backslash \boldsymbol{x} \rightarrow \boldsymbol{x}\ (<))\ ::\ Type_{Input} \rightarrow Type_{Output}$$

$$= Type\_von\_x \rightarrow Type\_von\_x(<)$$

$$= \boldsymbol{Ord\ c \Rightarrow ((c \rightarrow c \rightarrow Bool) \rightarrow b) \rightarrow b}$$

Also, schön in einer Zeile: $(\backslash \boldsymbol{x} \rightarrow \boldsymbol{x}\ (<)) :: \boldsymbol{Ord\ c \Rightarrow ((c \rightarrow c \rightarrow Bool) \rightarrow b) \rightarrow b}$

Haskell-Type Inference und Higher Order Programming Beispiel 2:

Hier noch ein weiteres Beispiel, das auf WhatsApp/Discord Fragen aufgeworfen hat:

Was ist der Unterschied zwischen:

$$(\backslash x\ y\ z \rightarrow (x\ y)\ z) \quad \text{und} \quad (\backslash x\ y\ z \rightarrow x\ (y\ z))$$

Dazu müssen wir uns zunächst noch einmal das altbekannte Mantra ins Gedächtnis rufen: „Function Application associates to the left“. Also, wenn a eine Funktion ist, dann: $a\ b\ c = (a\ b)\ c$. Beispielsweise: $add\ 2\ 4 = (add\ 2)\ 4$.

Warum ist das wichtig? Weil Haskell Funktionen immer nur ein Argument nehmen können. Bislang haben wir immer so getan, als wäre $add\ x\ y = x + y$ eine Funktion, die zwei Argumente x, y nimmt und dann deren Summe zurückgibt. Tatsächlich ist $add\ x\ y = x + y$ aber eine Funktion, die ein Argument x nimmt und dann eine Funktion $(x +)$ zurückgibt, die wiederum ein Argument y nimmt und einen Wert (hier konkret: Die Summe) zurückgibt.

Wenn wir so etwas: „$func\ x\ y$“ sehen, dann wissen wir, dass $func$ eine Funktion ist, die nach unserem „alten Verständnis“ zwei Argumente nimmt und irgendeinen Wert zurückgibt. Genauer wissen wir nun, dass $func$ ein Argument x nimmt und eine Funktion $(func\ x)$ zurückgibt. Also könnten wir $func\ x\ y$ auch gleich so schreiben: $(func\ x)\ y$.

Das heisst genauer für uns: Wenn wir so etwas sehen: $(f\ x)\ y$ dann können wir es uns im Kopf übersetzen zu $f\ x\ y$ – und sehen dann, dass f eine Funktion ist und x, y Argumente, die wir „Stück für Stück“ f übergeben.

Also gucken wir uns doch die beiden Funktionen von oben noch einmal genauer an. Zuerst die linke:

$$(\backslash x\ y\ z \rightarrow (x\ y)\ z)$$

Auf der „rechten Seite vom Pfeil“ sehen wir $(x\ y)\ z$ – und wir haben jetzt gelernt, wie man das übersetzen kann! x ist eine Funktion und y und z sind Argumente die an x, bzw. z an $(x\ y)$ übergeben werden. Also muss x vom Type $x :: a \rightarrow b \rightarrow c$ sein, da es ein Argument y vom (uns nicht genauer bekannten) Typ a nimmt und eine Funktion $(x\ y)$ zurückgibt, die ein Argument z vom (uns nicht genauer bekannten) Typ b nimmt und dann irgendeinen (uns nicht genauer bekannten) Rückgabewert von (uns nicht genauer bekannten) Type c zurückgibt. Also wissen wir jetzt:

$$(\backslash x\ y\ z \rightarrow (x\ y)\ z) :: (a \rightarrow b \rightarrow c) \rightarrow a \rightarrow b \rightarrow c$$

Betrachten wir nun die andere Funktion: $(\backslash x\ y\ z \rightarrow x\ (y\ z))$. Was ist hier anders? Wir haben Klammern um $(y\ z)$. Das heisst, wir haben hier nicht eine Funktion x die „nacheinander" zwei Argumente y, z „konsumiert", sondern eine Funktion x, die nur ein Argument – undzwar $(y\ z)$ – konsumiert. Und aus dem Argument $(y\ z)$ können wir folgern, da es sich dabei um eine Function-Application handelt, dass y eine Funktion sein muss, die Argumente vom Type von z nimmt und irgendwelche Rückgabewerte von einem Type zurückgibt, die x als Input nimmt.

Also: x hat den Type: $x :: a \rightarrow b$ (wobei wir noch nicht genau wissen, was a, b sind). Jetzt gucken wir uns das Argument für x an: $(y\ z)$. Dabei handelt es sich um eine Function-Application, also muss y eine Funktion sein, und irgendeinen Type $y :: c \rightarrow a$ haben (da der Rückgabewert von y als Argument an x übergeben wird – und x Argumente vom Type a nimmt). Da y als Argument ein z bekommt, muss z also vom Type $z :: c$ sein.

Also: $x :: a \rightarrow b, \quad y :: c \rightarrow a, \quad z :: c$

Also: $(\backslash x\ y\ z \rightarrow x\ (y\ z)) = (\backslash x \rightarrow \backslash y \rightarrow \backslash z \rightarrow x\ (y\ z)) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow c \rightarrow b$

(Man kann aufgrund der Rechtsassoziativität von Abstractions $(\backslash x\ y \rightarrow ..)$ immer zu $(\backslash x \rightarrow \backslash y \rightarrow ..)$ umschreiben. Dadurch wird es dann oft einfacher, später die Types ordentlich aufzuschreiben – ohne dabei Fehler bei den Klammern zu machen).

Also zusammengefasst: Einmal haben wir eine Funktion, die als Input eine Funktion x nimmt, die wiederum zwei weitere Argumente y, z nacheinander nimmt und irgendwas zurückgibt – und das andere Mal haben wir eine Funktion, die zwei Funktionen x, y und ein weiteres Argument z als Input nimmt, wobei die Funktion x als Input den Rückgabewert der Application $(y\ z)$ nimmt.

Type-Inference Proofs:

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \textit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash \lambda x.\, t :: \sigma \rightarrow \tau}\ \textit{Abs}$$

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1\ t_2 :: \tau}\ \textit{App} \qquad \frac{\Gamma \vdash t :: Int}{\Gamma \vdash \textbf{iszero}\ t :: Bool}\ \textit{iszero}$$

$$\frac{}{\Gamma \vdash n :: Int}\ \textit{Int} \qquad \frac{}{\Gamma \vdash True :: Bool}\ \textit{True} \qquad \frac{}{\Gamma \vdash False :: Bool}\ \textit{False}$$

$$\frac{\Gamma \vdash t_1 :: Int \qquad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1\ \textbf{op}\ t_2) :: Int}\ \textit{BinOp} \qquad \text{for } \textbf{op} \in \{+, *\}$$

$$\frac{\Gamma \vdash t_0 :: Bool \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 :: \tau}\ \textit{if}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}\ \textit{Tuple} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \textbf{fst}\ t :: \tau_1}\ \textit{fst} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \textbf{snd}\ t :: \tau_2}\ \textit{snd}$$

Hier nun eine Aufgabe aus einer (relativ aktuellen) Klausur (FS20):

Formally infer the most general type of the following expression (FS20):

$$(\lambda x.\, \lambda y.\, \text{iszero}\ ((\text{snd}\ y)(x\,y)))\ (\lambda x.\, \text{fst}\ x)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x:t_1, y:t_5 \vdash y :: (t_8, t_7 \to \text{Int})}\ \text{Var}^5}{x:t_1, y:t_5 \vdash \text{snd}\ y :: t_7 \to \text{Int}}\ \text{snd} \qquad \dfrac{T_2}{x:t_1, y:t_5 \vdash x\,y :: t_7}}{x:t_1, y:t_5 \vdash (\text{snd}\ y)\ (x\,y) :: \text{Int}}\ \text{App}}{x:t_1, y:t_5 \vdash \text{iszero}\ ((\text{snd}\ y)(x\,y)) :: t_6}\ \text{iszero}^4}{x:t_1 \vdash \lambda y.\ \text{iszero}\ ((\text{snd}\ y)(x\,y)) :: t_0}\ \text{Abs}^3}{\vdash (\lambda x.\, \lambda y.\ \text{iszero}\ ((\text{snd}\ y)(x\,y))) :: t_1 \to t_0}\ \text{Abs} \qquad \dfrac{\dfrac{\dfrac{}{x:t_2 \vdash x :: (t_3, t_4)}\ \text{Var}^2}{x:t_2 \vdash \text{fst}\ x :: t_3}\ \text{fst}}{\vdash (\lambda x.\ \text{fst}\ x) :: t_1}\ \text{Abs}^1}{\vdash (\lambda x.\, \lambda y.\ \text{iszero}\ ((\text{snd}\ y)(x\,y)))\ (\lambda x.\ \text{fst}\ x) :: t_0}\ \text{App}$$

T_2:

$$\dfrac{\dfrac{}{x:t_1, y:t_5 \vdash x :: t_9 \to t_7}\ \text{Var}^6 \qquad \dfrac{}{x:t_1, y:t_5 \vdash y :: t_9}\ \text{Var}^7}{x:t_1, y:t_5 \vdash x\,y :: t_7}\ \text{App}$$

Constraints:

$^1\ t_1 = t_2 \to t_3$	$^4\ t_6 = \text{Bool}$	$^7\ t_9 = t_5$
$^2\ t_2 = (t_3, t_4)$	$^5\ t_5 = (t_8, t_7 \to \text{Int})$	
$^3\ t_0 = t_5 \to t_6$	$^6\ t_1 = t_9 \to t_7$	

Um die Constraints zu vereinen, wollen wir nun einen Algorithmus laufen lassen, bis die Menge der Constraints C' die folgende Form hat: $\{x_1 = u_1, \ldots, x_m = u_m\}$, wobei die $x_1, \ldots, x_m$ unterschiedliche type-variables sind und $u_1, \ldots, u_m$ terme sind, in denen keines der $x_1, \ldots, x_m$ vorkommt. Dazu verwenden wir diesen Unification-Algorithmus:

Solving the set of constraints:

Algo:
1. Lösche triviale Constraints, e.g. $t_1 = t_1$
2. Wandle alle $f(s_0, \dots, s_k) = f(t_0, \dots, t_k)$ in $s_0 = t_0, \dots, s_k = t_k$ um, bspw.: $(t_3, t_4) = (t_5, t_6)$ gibt: $t_3 = t_5$ und $t_4 = t_6$
3. Falls ein Constraint der Form $f(s_0, \dots, s_k) = g(t_0, \dots, t_m)$ mit $f \neq g$ oder $k \neq m$ existiert: No Solution.
   Bsp.: $(t_3, \text{Int}) = (t_5, t_6 \to t_7)$
4. Transformiere alle $f(s_0, \dots, s_k) = x$ zu $x = f(s_0, \dots, s_k)$
5. Falls ein Constraint $x = t$ (mit x nicht in t) existiert, und x in unseren constraints vorkommt, substituiere $C \to C[x/t]$.
6. Falls constraint der Form $x = f(s_0, \dots, s_k)$ mit x in $s_0, \dots, s_k$ vorkommt: No solution. Bsp.: $t_1 = (t_1, t_2)$

(Algo-Regel 5): Substituiere $t_6 = \text{Bool}$:

1 $t_1 = t_2 \to t_3$	1 $t_1 = t_2 \to t_3$
2 $t_2 = (t_3, t_4)$	2 $t_2 = (t_3, t_4)$
3 $t_0 = t_5 \to t_6$	3 $t_0 = t_5 \to \text{Bool}$
4 $t_6 = \text{Bool}$	4 $t_5 = (t_8, t_7 \to \text{Int})$
5 $t_5 = (t_8, t_7 \to \text{Int})$	5 $t_1 = t_9 \to t_7$
6 $t_1 = t_9 \to t_7$	6 $t_9 = t_5$
7 $t_9 = t_5$	

(Algo-Regel 5): Substituiere $t_9 = t_5$:

1	$t_1 = t_2 \to t_3$	1	$t_1 = t_2 \to t_3$
2	$t_2 = (t_3, t_4)$	2	$t_2 = (t_3, t_4)$
3	$t_0 = t_5 \to \text{Bool}$	3	$t_0 = t_5 \to \text{Bool}$
4	$t_5 = (t_8, t_7 \to \text{Int})$	4	$t_5 = (t_8, t_7 \to \text{Int})$
5	$t_1 = t_9 \to t_7$	5	$t_1 = t_5 \to t_7$
6	$t_9 = t_5$		

(Algo-Regel 5): Substituiere $t_1 = t_5 \to t_7$:

1	$t_1 = t_2 \to t_3$	1	$t_5 \to t_7 = t_2 \to t_3$
2	$t_2 = (t_3, t_4)$	2	$t_2 = (t_3, t_4)$
3	$t_0 = t_5 \to \text{Bool}$	3	$t_0 = t_5 \to \text{Bool}$
4	$t_5 = (t_8, t_7 \to \text{Int})$	4	$t_5 = (t_8, t_7 \to \text{Int})$
5	$t_1 = t_5 \to t_7$		

(Algo-Regel 2): Schreibe um: $t_5 \to t_7 = t_2 \to t_3 \Leftrightarrow t_5 = t_2,\ t_7 = t_3$

1	$t_5 \to t_3 = t_2 \to t_3$	1	$t_5 = t_2$
2	$t_2 = (t_3, t_4)$	2	$t_7 = t_3$
3	$t_0 = t_5 \to \text{Bool}$	3	$t_2 = (t_3, t_4)$
4	$t_5 = (t_8, t_7 \to \text{Int})$	4	$t_0 = t_5 \to \text{Bool}$
		5	$t_5 = (t_8, t_7 \to \text{Int})$

(Algo-Regel 5): Substituiere: $t_7 = t_3$

1	$t_5 = t_2$	1	$t_5 = t_2$
2	$t_7 = t_3$	2	$t_2 = (t_3, t_4)$
3	$t_2 = (t_3, t_4)$	3	$t_0 = t_5 \to \text{Bool}$
4	$t_0 = t_5 \to \text{Bool}$	4	$t_5 = (t_8, t_3 \to \text{Int})$
5	$t_5 = (t_8, t_7 \to \text{Int})$		

(Algo-Regel 5): Substituiere $t_5 = t_2$

$$\begin{array}{ll} 1\ t_5 = t_2 & 1\ t_2 = (t_3, t_4) \\ 2\ t_2 = (t_3, t_4) & 2\ t_0 = t_2 \to \text{Bool} \\ 3\ t_0 = t_5 \to \text{Bool} & 3\ t_2 = (t_8, t_3 \to \text{Int}) \\ 4\ t_5 = (t_8, t_3 \to \text{Int}) & \end{array}$$

(Algo-Regel 5): Substituiere $t_2 = (t_8, t_3 \to \text{Int})$

$$\begin{array}{ll} 1\ t_2 = (t_3, t_4) & 1\ (t_8, t_3 \to \text{Int}) = (t_3, t_4) \\ 2\ t_0 = t_2 \to \text{Bool} & 2\ t_0 = (t_8, t_3 \to \text{Int}) \to \text{Bool} \\ 3\ t_2 = (t_8, t_3 \to \text{Int}) & \end{array}$$

(Algo-Regel 2): Schreibe um: $(t_8, t_3 \to \text{Int}) = (t_3, t_4) \Leftrightarrow t_8 = t_3,\ t_3 \to \text{Int} = t_4$

$$\begin{array}{ll} 1\ (t_8, t_3 \to \text{Int}) = (t_3, t_4) & 1\ t_8 = t_3 \\ 2\ t_0 = (t_8, t_3 \to \text{Int}) \to \text{Bool} & 2\ t_3 \to \text{Int} = t_4 \\ & 3\ t_0 = (t_8, t_3 \to \text{Int}) \to \text{Bool} \end{array}$$

(Algo-Regel 4): Schreibe um: $t_3 \to \text{Int} = t_4 \Leftrightarrow t_4 = t_3 \to \text{Int}$

$$\begin{array}{ll} 1\ t_8 = t_3 & 1\ t_8 = t_3 \\ 2\ t_3 \to \text{Int} = t_4 & 2\ t_4 = t_3 \to \text{Int} \\ 3\ t_0 = (t_8, t_3 \to \text{Int}) \to \text{Bool} & 3\ t_0 = (t_8, t_3 \to \text{Int}) \to \text{Bool} \end{array}$$

(Algo-Regel 5): Substituiere: $t_8 = t_3$

$$\begin{array}{ll} 1\ t_8 = t_3 & 1\ t_4 = t_3 \to \text{Int} \\ 2\ t_4 = t_3 \to \text{Int} & 2\ t_0 = (t_3, t_3 \to \text{Int}) \to \text{Bool} \\ 3\ t_0 = (t_8, t_3 \to \text{Int}) \to \text{Bool} & \end{array}$$

Das Constraint-Set hat die gewünschte Form, also: $t_0 = (t_3, t_3 \to \text{Int}) \to \text{Bool}$

Also:

$(\lambda x.\, \lambda y.\, \text{iszero}\ ((\text{snd}\ y)(x\ y)))\ (\lambda x.\, \text{fst}\ x) :: (a, a \to \text{Int}) \to \text{Bool}$

Folds:

Kurz zu Algebraic Data Types (alles weitere könnt ihr entweder in den VL-Slides oder bspw. hier: http://learnyouahaskell.com/making-our-own-types-and-typeclasses#algebraic-data-types selbst nachlesen und durch selbst ausprobieren viel besser lernen, als ich es je erklären könnte):

Wir definieren einen Algebraic Data Type in Haskell folgendermassen:

$$data\ Dtype = C_1\ \ldots \mid C_2\ \ldots \mid\ \ldots\ \mid C_N \ldots$$

Wobei wir das Ganze als „**datatype definition**“ bezeichnen.

Den Namen unseres datatypes (hier „Dtype“) nennt man „**type constructor“**. Die einzelnen C_i nennt man „**data constructor**“ (oder manchmal auch „value constructors“ oder „tag“, je nachdem, wo ihr lest).

Auf jeden data constructor C_i folgen ≥ 0 types, beispielsweise: $data\ Point = P\ \boldsymbol{int\ int.}$

Wenn wir nun eine konkrete Instanz des datatypes Dtype erstellen wollen, verwenden wir einen der data constructors. Hier mal ein konkretes Beispiel:

```haskell
data Point = Pkart Float Float | Ppolar Float Float
    deriving (Show, Eq)

distance :: Point -> Point -> Float
distance (Pkart x1 y1) (Pkart x2 y2)       = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))
distance (Ppolar r1 ang1) (Ppolar r2 ang2) = sqrt(r1*r1 + r2*r2 - 2*r1*r2*cos(ang1-ang2))
distance _ _                               = 0
```

Wir haben den datatype „Point“ definiert: Entweder als Punkt in kartesischen Koordinaten oder als Punkt in Polarkoordinaten. Die data constructors sind also Pkart und Ppolar. Möchte ich nun einen konkreten Punkt erstellen, verwende ich einen der data constructors. Das könnte bspw. so aussehen:

```
*Main> :l uebung6.hs
[1 of 1] Compiling Main             ( uebung6.hs, interpreted )
Ok, one module loaded.
*Main> let a = Pkart 2 4
*Main> let b = Pkart 3 1
*Main> distance a b
3.1622777
*Main> let a = Ppolar 4 0.4
*Main> let b = Ppolar 2 2.1
*Main> distance a  b
4.696968
*Main>
```

Als kleines extra habe ich noch eine Funktion „distance“ geschrieben, die die Distanz zwischen zwei Punkten berechnet, sofern beide im gleichen Koordinatensystem gegeben sind und sonst 0 zurückgibt (das hat keine mathematischen Gründe ich bin einfach zu faul). Wenn wir uns den Type von distance ansehen, bemerken wir, dass dort nicht von Pkart oder Ppolar die Rede ist, sondern von dem allgemeinen type constructor, „Point“. Die Funktion distance nimmt also ein Point Argument und gibt eine Funktion zurück, die ein Argument vom Type Point nimmt und einen Float zurückgibt (die Distanz).

Wenn wir in Funktionen bestimmte Instanzen von datatypes als Parameter behandeln wollen, ist es ganz wichtig, dass wir sie **in Klammern schreiben**! Also bspw. (Ppolar r1 ang1).

Das gleiche gilt auch, falls wir bei einem data constructor nicht bloss einen gewöhnlichen Type (Int, Float, …) übergeben wollen, sondern eine Application eines type constructors, bspw.:

```
data MyList a = Empty | Cons a (MyList a)
```

Hier soll eine Liste definiert werden, die entweder ein „Empty“ ist, oder ein „Cons“, wobei Cons ein data constructor ist, der ein „a“ nimmt und eine Liste aus „a“. Damit wir diese type constructor application („MyList a“) hier anwenden können, sind die Klammern wichtig.

Abschliessend lässt sich noch erwähnen, dass wir auch Polymorphic Datatypes definieren können: Wenn wir beispielsweise wollen, dass nicht sofort klar ist, was die Typen bei den data constructors sind, können wir dem type constructor dafür Argumente geben, hier ein Beispiel:

```
data PolyPoint a = Pkartpoly a a | Ppolarpoly a a


*Main> let a = Pkartpoly "Maximilian" "Schlegel"
```

Jetzt haben wir einen Punkt, der Beispielsweise als Koordinaten auch Strings erlaubt. (Mir fällt jetzt aber keine sinnvolle Verwenden für dieses konkrete Beispiel ein lol).

So kommen wir nun zu Folds:

Wir haben bereits die Funktion „foldr“ kennengelernt, die eine Liste nimmt und sie auf einen Wert „kollabiert“. Dieser Wert kann selbst auch eine Liste oder eine Funktion oder sonst irgendetwas sein – aber wir wissen, dass foldr einmal durch die Liste geht und dabei die Listenelemente alle auf irgendeine (durch uns spezifizierte) Weise auf etwas reduziert.

Tatsächlich ist per Definition von „folding“ (eine Datenstruktur auf einen Wert reduzieren) nicht nur foldr eine Funktion, die einen „fold“ durchführen kann, die Funktion hier tut das bspw. auch:

```
len :: [a] -> Int
len []     = 0
len (x:xs) = 1 + len xs
```

Wir haben bislang eigentlich aber nur mit Listen gearbeitet, das heißt, alle Folds, die wir bislang kennengelernt haben, haben Listen auf einen Wert reduziert. Nun wollen wir Folds betrachten, die mit anderen Datentypen, bspw. Trees arbeiten können. Es ist eine häufige Klausuraufgabe, zu einem

bestimmten Datentypen die zugehörigen Fold-Funktion anzugeben. Dafür gibt es glücklicherweise ein konkretes Schema. (Hier von den Slides):

Given a data type

```
data DType = C1 ... | C2 ... | ... | CN ...
```

The fold function for `DType` that folds it to type `b` has the following type:

```
foldDType :: foldC1 ->
             foldC2 ->
             ... ->
             foldCN ->
             DType -> b
```

where each type `foldCi` corresponds to the i-th constructor of `DType`.
It is derived from the type of the constructor by replacing `DType` with `b`.

Um das genauer zu verstehen, betrachten wir das folgende Beispiel:

```haskell
data Prop a = Var a | Not (Prop a) | And (Prop a) (Prop a) | Or (Prop a) (Prop a)

-- Das Schema ist:
--    foldProp :: foldVarType -> foldNotType -> foldAndType -> foldOrType -> (Prop a) -> b
-- Also:

foldProp :: (a -> b) -> (b -> b) -> (b -> b -> b) -> (b -> b -> b) -> (Prop a) -> b
```

Wir haben einfach das bekannte Schema angewandt. Dafür betrachten wir mal bspw. „foldVarType“, für das ich den Type „(a -> b)“ bei foldProp hingeschrieben habe: Der data constructor Var nimmt ein „a“ und gibt ein (Prop a) zurück (konkret: ein Var, aber der allg. Type ist ja trotzdem Prop a). Gemäss dem Schema ersetzen wir alle Vorkomnisse des type constructors („Prop a“) durch den return type der Funktion, also “b” und erhalten damit dann für die Funktion, die Var’s folded den Type (a -> b).

Wenn man das dann alles zusammensetzt bekommt man den Type der oben dasteht. Nun überlegen wir uns noch, wie man foldProp wirklich konkret implementieren kann – aber das sollte mit einer einfachen Überlegung und dem genauen Betrachten des Types von foldProp relative einfach sein:

```haskell
data Prop a = Var a | Not (Prop a) | And (Prop a) (Prop a) | Or (Prop a) (Prop a)

-- Das Schema ist:
--    foldProp :: foldVarType -> foldNotType -> foldAndType -> foldOrType -> (Prop a) -> b
-- Also:

foldProp :: (a -> b) -> (b -> b) -> (b -> b -> b) -> (b -> b -> b) -> (Prop a) -> b
foldProp fVar fNot fAnd fOr prop = go prop      -- prop eig. unnötig, aber fürs Verständnis
    where
        go (Var x)   = fVar x
        go (Not x)   = fNot (go x)
        go (And x y) = fAnd (go x) (go y)
        go (Or x y)  = fOr (go x) (go y)
```

Auch hier bedienen wir uns wieder eines einfachen Schemas (das **immer** funktioniert): Die Funktion foldProp nimmt als Argumente die Funktionen, die zu den einzelnen data constructors korrespondieren, hier also fVar (die ein “Prop a” folded, falls dieses “Prop a” konkret eine “Var a” ist), fNot, etc..

Um das vertiefend zu üben, hier noch einige weitere Beispiele:

```haskell
module Folds where

data Unit = U

foldUnit :: b -> Unit -> b    -- Die Funktion die eine Unit folded
                              -- hat einfach nur den Type "b" da der data
                              -- constructor "U" sofort ein Unit zurückgibt
                              -- und wir per Schema alle "Unit" durch b
                              -- ersetzen
foldUnit fU = go
  where
    go U = fU


data Boolean = T | F

foldBoolean :: b -> b -> Boolean -> b
foldBoolean fT fF = go
  where
    go T = fT
    go F = fF


data IList = INil | ICons Int IList

foldIList :: b -> (Int -> b -> b) -> IList -> b
foldIList fNil fCons = go
  where
    go (INil) = fNil
    go (ICons s e) = fCons s (go e)


data Tree a = Leaf | Node a (Tree a) (Tree a)

foldTree :: b -> (a -> b -> b -> b) -> Tree a -> b
foldTree fLeaf fNode = go
  where
    go (Leaf) = fLeaf
    go (Node s a b) = fNode s (go a) (go b)


data G a b = A Int | B (G a b) Bool (G a b) | C a b

foldG :: (Int -> c) -> (c -> Bool -> c -> c) -> (a -> b -> c) -> G a b -> c
foldG fA fB fC = go
  where
    go (A n) = fA n
    go (B g b z) = fB (go g) b (go z)
    go (C a b) = fC a b
```

Lazy vs. Eager Evaluation:

Eigentlich muss man nur folgendes genau befolgen:

Nehmen wir an, wir wollen (t1 t2) evaluieren.

Lazy Evaluation:

1. Evaluiere t1
2. Substituiere t2 in t1 – ohne t2 zu evaluieren
3. No Evaluation under Abstraction

Eager Evaluation:

1. Evaluiere t1
2. Evaluiere t2, substituiere dann t2 in t1
3. Evaluation is carried out under Abstraction

Hier ein konkretes Beispiel zum Verständnis. Achtung: Um die Binding Structure nicht zu verletzen muss man manchmal (durch eine lambda-abstraction) gebundene Variablen umbenennen:

Weiter auf der nächsten Seite →

```
Evaluate (λx. x (λy. x y)) (λx. y x) using Lazy & Eager Evaluation:

Lazy:
(λx. x (λy. x y)) (λx. y x)

Wir können t1 nicht weiter evaluieren, also machen wir mit Schritt 2 weiter
und substituieren t2 in t1.

Um die Substitution zu verstehen, gucken wir uns t1 an: t1 nimmt ein x
und gibt x (λ. x y) zurück. Das heisst, wir ersetzen an jeder Stelle im Rück-
gabewert, an der x steht, das x durch t2.

Dabei bennen wir das y in t1 in z um, damit es nicht zu einer Verletzung der
Binding Structure kommt.

= (λx. y x) (λz. (λx. y x) z)

Wieder eine Substitution (gleiches Prinzip wie oben):

= y (λz. (λx. y x) z)

Wir können nicht weiter evaluieren.

Eager:
(λx. x (λy. x y)) (λx. y x)

(Exakt gleicher Schritt, wie der erste bei Lazy: Substitution. Und wir
können t2 hier nicht vorab evaluieren.

= (λx. y x) (λz. (λx. y x) z)

Nun haben wir wieder eine Funktion-Application t1 t2 dastehen. Zunächst
können wir hier aber t2 evaluieren (Diesen Teil: "(λx. y x) z" in t2).

= (λx. y x) (λz. y z)

Und nun substitutieren wir abschliessend in t1:

= y (λz. y z)

Wir können nicht weiter evaluieren.
```

Induction on Trees:

Gleiches Prinzip wie „normale“ Induktionsproofs mit Listen, bloss ist der Base Case hier ein Leaf und der Step Case benötigt oft 2 oder sogar 3 Induktionshypothesen.

Falls der Tree irgendwie so definiert ist:

$$data\ Tree = Leaf \mid Node\ a\ \ (Tree\ a)\ \ (Tree\ a)$$

Arbeitet ihr im Step Case üblicherweise mit dem Case „Node x l r“ und formuliert eure Induktionshypothesen für die Subtrees „l“ und „r“ und zeigt dann, dass die zu beweisende Aussage auch für (Node x l r) gilt, wenn sie per Annahme für die Subtrees l,r gilt.

Falls es hierzu Fragen gibt, schreibt mir einfach. Guckt bei Syntax-Fragen in die Zusammenfassungen der Vorwochen.