Woche 3 – 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:

- Insgesamt sehr schön gelöst. Mir sind keine besonderen Probleme aufgefallen.
- Teilweise wurde die Coins-Aufgabe sehr aufwändig gelöst. Hier ist mein Lösungsvorschlag, der den gegebenen Hinweis einfach direkt (mit den Haskell-Tricks, die wir bis jetzt kennengelernt haben) implementiert:

```haskell
cntChange :: Int -> Int
cntChange 0 = 1
cntChange n = aux n [5,10,20,50,100,200,500]
  where
    aux n [] = 0
    aux n (x:xs)
        | n < 0     = 0
        | n == 0    = 1
        | otherwise = (aux (n-x) (x:xs)) + (aux n xs)
```

Anmerkungen zu den Moodle Abgaben:

- Bis auf kleine Fehler wurde die Serie 2 sehr gut gelöst. Um die kleinen Fehler zu beheben, werde ich hier nun noch einmal kurz ein wichtiges Konzept genauer behandeln:
  - Implikationen sind rechts-assoziativ, also $A \to B \to C \equiv A \to (B \to C)$
  - Damit eine Implikation $False$ wird, muss ihre LHS $True$ und ihre RHS $False$ sein, also wird bspw. $A \to B \to C \to D \equiv A \to (B \to (C \to D))$ genau dann $False$, wenn A $True$ wird und $B \to (C \to D)$ $False$ wird. Dies ist der Fall, wenn B $True$ und $C \to D$ $False$ wird. Dies wiederum geschieht gdw. C $True$ und D $False$ ist. Also: $A = B = C = 1$ und $D = 0$ damit die Formel von oben $False$ wird. (Achtung, hinsichtlich der Formalität ist das hier jetzt nicht perfekt, es geht mir um die Intuition, damit ihr versteht, wie Formeln, die mehrere „verschachtelte“ Implikationen enthalten, bestimmte Wahrheitswerte annehmen können).

Natural Deduction – Letzte Aufgabe (Midterm Prep.):

① Beweise $\forall x.\, Q(x) \vee P(x) \to (\exists y.\, Q(y)) \vee P(x)$ mit Nat. Ded.!

Zuerst schreiben wir um:

$$\forall x.\, Q(x) \vee P(x) \to (\exists y.\, Q(y)) \vee P(x)$$

$$\equiv \forall x.\big(Q(x) \vee P(x) \to (\exists y.\, Q(y)) \vee P(x)\big)$$

$$\equiv \forall x.\big((Q(x) \vee P(x)) \to ((\exists y.\, Q(y)) \vee P(x))\big)$$

Ausserdem definieren wir: $\Gamma := Q(x) \vee P(x)$

Nun können wir den Beweis nach bekanntem Muster führen:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash Q(x) \vee P(x)}\,Ax \quad \dfrac{\dfrac{\dfrac{}{\Gamma, Q(x) \vdash Q(x)}\,Ax}{\Gamma, Q(x) \vdash \exists y.\, Q(y)}\,\exists\text{-I}}{\Gamma, Q(x) \vdash (\exists y.\, Q(y)) \vee P(x)}\,\vee\text{-IL} \quad \dfrac{\dfrac{}{\Gamma, P(x) \vdash P(x)}\,Ax}{\Gamma, Q(x) \vdash (\exists y.\, Q(y)) \vee P(x)}\,\vee\text{-IR}}{\Gamma \vdash (\exists y.\, Q(y)) \vee P(x)}\,\vee\text{-E}}{\vdash (Q(x) \vee P(x)) \to ((\exists y.\, Q(y)) \vee P(x))}\,\to\text{-I}}{\vdash \forall x.\big((Q(x) \vee P(x)) \to ((\exists y.\, Q(y)) \vee P(x))\big)}\,\forall\text{-I}^{*}$$

*x ist nichts frei in der (leeren) Menge unserer Annahmen.

Der interessante Schritt ist hier vermutlich die ∨-E – wie kommt man darauf? Wir betrachten einmal die RHS in der "Conclusion" (≈ "unterm Strich"): $(\exists y.\, Q(y)) \vee P(x)$. Wir können sie mit ∨-IL bzw. ∨-IR "aufspalten", allerdings reicht unsere Annahmenmenge $Q(x) \vee P(x)$ nicht aus, um $Q(x)$ oder $P(x)$ einzeln zu beweisen. Um hier zu entscheiden, brauchen wir also eine Regel, die unsere Annahmenmenge (in zwei verschiedenen Fällen) vergrössert, am besten Eignet sich also ∨-E.

Induktion über Natürliche Zahlen – Recap und Induktion mit Haskell-Funktionen:

Ich hatte es in der ersten Übungsstunde ja schon einmal kurz angeschnitten, daher jetzt hier noch einmal:

Prinzip der vollst. Induktion:

$(A(n_0) \wedge [\forall n \in \mathbb{N}, n \geq n_0 : A(n) \Rightarrow A(n+1)]) \Rightarrow \forall n \in \mathbb{N}, n \geq n_0 : A(n)$

- Behauptung: $\forall n \in \mathbb{N}, n \geq n_0 : A(n)$
- Induktionsanfang / Base Case (IA/BC): Zeige, dass A für kleinstes n_0 gilt
  "A(n) gilt für EIN **konkretes** n"
- Induktionshypothese / Induktionsvorraussetzung (IH / IV) $A(m)$ gilt für bel. fixes n.
  "A(m) gilt für EIN **allgemeines** m"
- Induktionsschritt (IS): Beweis von $A(m) \Rightarrow A(m+1)$
  "Wenn A(m) für ein m gilt, dann gilt immer auch A(m+1), da wir m allgemein gehalten haben"

$\Rightarrow$ *Induktionsbehauptung (IB):* $\forall n \in \mathbb{N}, n \geq n_0 : A(n)$

Prinzip der Starken Induktion:

$\left(A(n_0) \wedge \left[\forall n \in \mathbb{N}, n > n_0 : \left(\forall k \in \mathbb{N}, n_0 \leq k < n : A(k)\right) \Rightarrow A(n)\right]\right) \Rightarrow \forall n \in \mathbb{N}, n > n_0 : A(n)$

- Behauptung: $\forall n \in \mathbb{N}, n \geq n_0 : A(n)$
- Induktionsanfang / Base Case (IA/BC): Zeige, dass A für kleinstes $n_0 \in \mathbb{N}$ (oder eine Menge an kleinsten $n_0, \dots, n_x \in \mathbb{N}$) gilt
  "A(n) gilt für EIN **konkretes** n_0 (bzw. EINE **konkrete Menge** $M := \{n_0, \dots, n_x\}$)"
- Induktionshypths. / Induktionsvorstz. (IH / IV): Für bel. fix $m \geq n_0 : \forall n_0 \leq k < m : A(k)$
  "A(k) gilt für alle k < m für EIN **allgemeines** m"
- Induktionsschritt (IS): Beweis von $A(m) \Rightarrow A(m+1)$
  "Wenn A für alle $k \in \{n_0, \dots, m\}$ gilt, dann gilt immer auch A(m+1), da wir n allgemein gehalten haben"

$\Rightarrow$ *Induktionsbehauptung (IB):* $\forall n \in \mathbb{N} : A(n)$

Allgemeine Regeln zur Induktion in FMFP:

- **Struktur:**

```
Proof by Induction:

Let P := ...

We show \forall n:Nat.P by induction.

Base Case: Show P[n->0].
Proof...

Step Case: Let m \in N be arbitrary (where m is not free in P).
           Assume P[n->m], then show P[n->m+1]
Proof...

QED
```

- Ganz wichtig: Bei **jedem** Schritt die Begründung für diesen Schritt in Klammern angeben (üblicherweise am Ende der Zeile, auch wenn es noch so trivial wirken mag – es geht um das Einhalten von Formlt.). P[n->m] kann auch als P[n/m] (CYP-Syntax) geschrieben werden.

Beispiel 1 – Induktionsbeweise mit Haskell-Funktionen:

Wir beginnen mit einem vergleichsweise einfachen (und hoffentlich noch aus Algorithmen & Datenstrukturen bekannten) Beispiel zur Gauß-Summenformel:

Wir nehmen an, uns wird diese Haskell-Funktion gegeben:

```haskell
gsum :: Int -> Int
gsum 0 = 0                    -- gsum.1
gsum n = n + gsum (n-1)       -- gsum.2
```

Die Person, die uns diese Funktion gegeben hat, behauptet, dass sie die Gauß'sche Summenformel berechnet, also $\sum_{i=0}^{n} i = \frac{n*(n+1)}{2}$ für ein gegebenes n. Nun wollen wir d as formal korrekt beweisen, also: $\forall n\text{: Nat} \,.\, \text{gsum n} = \text{n} \cdot (\text{n} + 1)/2$:

Proof by Induction:

Let P := gsum n = n·(n+1)/2

We show ∀n: Nat. P by induction.

Base Case:

Show P[n/0]. Proof:

gsum 0 = 0 (by gsum.1)

= 0·(0+1)/2 (by arith.)

Step Case:

Let m: Nat be arbitrary (note that m is not free in P).

Assume

IH: P[n/m] (gsum m = m·(m+1)/2)

Then show

P[n/(m+1)] (gsum (m+1) = (m+1)·((m+1)+1)/2)

(weiter auf der nächsten Seite →)

Proof:

$$
\begin{aligned}
\text{gsum}\ (m+1) &= (m+1) + \text{gsum}\ ((m+1)-1) && \text{(by gsum.2)}\\
&= (m+1) + \text{gsum}\ (m) && \text{(by arith.)}\\
&= (m+1) + (m\cdot(m+1)/2) && \text{(by IH)}\\
&= (2(m+1) + m\cdot(m+1))/2 && \text{(by arith.)}\\
&= (m+1)\cdot(m+2)/2 && \text{(by arith.)}\\
&= (m+1)\cdot((m+1)+1)/2 && \text{(by arith.)}
\end{aligned}
$$

QED

Beispiel 2 – Induktionsbeweise mit Haskell-Funktionen:

Hier ein weiteres einfaches Beispiel, wir wollen beweisen, dass die Funktionen dasselbe berechnen, also $\forall n : \text{Nat} \,.\, \text{flup}\ n = \text{flop}\ n$.

```haskell
flup :: Int -> Int
flup 0 = 0                                  -- flup.1
flup n = (2*n - 1) + flup (n-1)             -- flup.2

flop :: Int -> Int
flop n = n * n                              -- flop.1
```

Und hier der dazugehörige Proof:

Proof by Induction:

Let P := flup n = flop n

We show ∀n:Nat.P by induction.

Base Case:

Show P[n/0]. Proof:

$$
\begin{aligned}
\text{flup}\ 0 &= 0 && \text{(by flup.1)}\\
&= 0\cdot 0 && \text{(by arith.)}\\
&= \text{flop}\ 0 && \text{(by flop.1)}
\end{aligned}
$$

(weiter auf der nächsten Seite →)

Step Case:

Let m:Nat be arbitrary (note that m is not free in P).

Assume

IH: P[n/m] (flup m = flop m)

Then show

P[n/(m+1)] (flup (m+1) = flop (m+1))

Proof:

$$
\begin{aligned}
\text{flup } (m+1) &= (2\cdot(m+1)-1) + \text{flup } ((m+1)-1) && \text{(by flup.2)}\\
&= (2\cdot(m+1)-1) + \text{flup } m && \text{(by arith.)}\\
&= ((2m+2)-1) + \text{flup } m && \text{(by arith.)}\\
&= (2m+1) + \text{flup } m && \text{(by arith.)}\\
&= (2m+1) + \text{flop } m && \text{(by IH)}\\
&= (2m+1) + (m\cdot m) && \text{(by flop.1)}\\
&= (m+1)\cdot(m+1) && \text{(by arith.)}\\
&= \text{flop } (m+1) && \text{(by flop.1)}
\end{aligned}
$$

QED

Haskell Rekursion 2:

Die Funktionen, die wir in der Übungsstunde implementiert haben:

```haskell
myMax :: (Ord t, Num t) => [t] -> t
myMax xs = aux xs 0
    where
        aux [] m = m
        aux (x:xs) m
            | x > m  = aux xs x
            | otherwise = aux xs m
```

```haskell
-- myFilter, given arguments a test and a list should only leave
-- elements in the list that pass the test, e.g.
-- myFilter (>3) [1,2,3,4,5] = [4,5]
myFilter :: (a->Bool) -> [a] -> [a]
myFilter test [] = []
myFilter test (x:xs)
    | test x = x : myFilter test xs
    | otherwise = myFilter test xs
```

```haskell
-- Reverse of a List
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

Haskell – List Comprehensions:

In Haskell gibt es ein sehr mächtiges Werkzeug, um mit Listen zu arbeiten: Sogenannte List Comprehensions. Aus der (bspw. Diskreten) Mathematik wissen wir noch, dass wir Mengen sehr bequem mit folgender Schreibweise angeben können:

$$M := \{2 * x : x \in \mathbb{Z}, 0 < x < 5\} \quad \text{(Beispiel)}$$

Genau das können wir in Haskell auch: List Comprehensions in Haskell haben die Form:

```
[exp | qualifier_1, ... ,qualifier_n]

Wobei qualifier folgendes sein können:
    - "generatoren" der Form x <- xs, mit x :: t und xs :: [t]
    - "boolean guards"

Meistens sehen List comprehensions so aus:
[expression_mit_x | x <- liste, boolean_guard]

Bspw.: [2*x | x <- [1..20], x % 2 == 1]    -- Also: Das Doppelte aller ungeraden Zahlen von 1 bis 20
```

(Kleiner Fehler hier: Wir müssten in Haskell natürlich den „mod“ Operator verwenden, statt „%“)

Das Coole: Da links vom „|“ eine Expression stehen soll, können wir dort alles hinschreiben, was in Haskell nun mal eine Expression ist, bspw. auch „if … then … else …“ oder „let … in …“ etc.

Da man das am besten mit Beispielen versteht, hier nun Einige:

(weiter auf der nächsten Seite →)

```haskell
-- Return a list of every even integer from 1 to 100, but squared:
f1 = [x*x | x <- [1..100], even x]

-- Return a list of all integers from 1 to 100 that are divisible by 3 but not by 6
f2 = [x | x <- [1..100], mod x 3 == 0, mod x 6 /= 0]

-- Return a list consisting of "PRIME" for every index of a prime number and "NOT" for every other
-- index for all integers from 1 to 10
f3 = [(if (prime x) then "PRIME" else "NOT") | x <- [1..10]]
    where
        prime 1 = False
        prime 2 = True
        prime n = [x | x <- [1..(ceiling(sqrt(fromIntegral n)))], n `mod` x == 0] == [1]

-- (Source: learnyouahsakell.com/starting-out)
-- Let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,6]]
-- Return a list of lists, consisting only of the even elements of the lists in xxs:
f4 = [[x | x <- xs, even x] | xs <- xxs]

-- Return a list of tuples of all possible pairs of letters from "abc" and "xyz", e.g.
-- [('a','x'), ('a','y'), ('a','z'), ... , ('c','z')]
f5 = [(l,r) | l <- "abc", r <- "xyz"]
```

Auch lässt sich mithilfe von List-Comprehensions und einfachen List-Operationen der Quicksort-Algorithmus deutlich kürzer implementieren:

```haskell
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort [l | l <- xs, l <= x]) ++ [x] ++ (quicksort [r | r <- xs, r > x])
```

Haskell Lists – List Functions Pt. 3:

Wir haben bereits einige fundamentale Haskell Prelude Funktionen für Listen kennengelernt, bspw. head, tail, sum, max, etc.. Es gibt jedoch noch weitere sehr nützliche Funktionen, von denen wir uns hier nun drei genauer angucken wollen.

List-Funktion 1: „map“:

```haskell
-- So ist map definiert:
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : (map f xs)

-- Also: Map wendet eine Funktion auf jedes einzelne Element der Liste an. Bspw. so:
map (*2) [1..10]                                  -- [2,4,6,8,10,12,14,16,18,20]
map reverse ["abc", "def", "ghi"]                 -- ["cba", "fed", "ihg"]
map (\x -> 2*x + 1) [1..5]                        -- [3,5,7,9,11]
map (\x -> if even x then "EVEN" else "ODD") [1..10] -- ["ODD","EVEN","ODD",..."EVEN"]
```

List-Funktion 2: „filter“:

```haskell
-- So ist filter definiert:
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs

-- Also: Filter verhält sich ziemlich genau so, wie man es vom Namen erwartet und behält nur Elemente
--       in der Liste, die einen Test bestehen. Beispiele:

filter (>3) [1..10]                     -- [4,5,6,7,8,9,10]
filter (even) [1..10]                   -- [2,4,6,8,10]
filter (\x -> x*x < x) [0.0,0.1..2.0]   -- [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
```

List-Funktion 3: „foldr“:

```haskell
-- So ist foldr definiert:
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

-- Also: foldr "akkumuliert" die Werte einer Liste auf eine bestimmte (durch die Funktion def.) Art.:
-- Genauer: foldr f z (a:b:c:[]) = f a (f b (f c (f z []))) wobei f immer zwei Argumente nimmt.
-- Beispielsweise könnte foldr die Werte einer Liste aufsummieren.

foldr (+) 0 [1..4]                                      -- 10
foldr (*) 1 [1..4]                                      -- 1 * (2 * (3 * (4 * (1)))) = 24
foldr (\x acc -> x + acc) 0 [1..4]                      -- 10 (same as "foldr (+) 0 [1..4]")
foldr (\x acc -> if even x then x : acc else acc) [] [1..10]  -- [2,4,6,8,10]
```