Woche 4 – 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:

- OTP sehr schön (später im 6. Semester in InfoSec mehr dazu).
- Prime Numbers auch sehr schön gelöst (am effizientesten mit List Compr.).
- Words habe ich teilweise sehr schöne & kurze Ergebnisse gesehen.
- Palindromes teilweise sehr mühsam gelöst. So geht es am schnellsten:

```haskell
palindromes :: [String] -> [String]
palindromes xs = [v ++ w | v <- xs, w <- xs, v ++ w == reverse (v ++ w)]
```

Anmerkungen zu den Theorie Abgaben aus dem letzten Jahr:

- Gemischte Ergebnisse bei 1a): Guckt euch alle noch einmal die Musterlösung an, sollte dann aber selbsterklärend sein. (Auch in diesem Jahr ein guter Hinweis 😉)
- Einige kleinere Probleme bei 1b), die aber sehr wichtig sind:
    1. Wichtig: Jede Umformung benötigt eine Begründung, egal wie trivial
    2. Viele illegale Umformungen. Hier eine Erklärung:

```haskell
fibLouis :: Int -> Int
fibLouis 0 = 0                                  -- fibLouis.1
fibLouis 1 = 1                                  -- fibLouis.2
fibLouis n = fibLouis (n - 1) + fibLouis (n - 2)    -- fibLouis.3
```

Noch einmal zur Erinnerung die Definition von fibLouis. Oft habe ich nun diesen Umformungsschritt gesehen (nächstes Bild). Hier auch direkt die richtige Lösung:

(weiter auf der nächsten Seite →)

```
        (fibLouis (n+1), fibLouis (n+1) + fibLouis (n))
=       (fibLouis (n+1), fibLouis (n+2))                              (by fibLouis.3)

Was ist das Problem?
Es gibt bei fibLouis.3 auf der RHS von fibLouis (n+2) nicht die
Expression "fibLouis (n+1) + fibLouis (n)"
sondern "fibLouis ((n+2)-1) + fibLouis ((n+2)-2)".
DAS IST NICHT DAS GLEICHE, besonders wenn wir unsere
Proofs automatisiert testen wollen. Also muss ein arithmetischer
Zwischenschritt eingefügt werden. Richtig wäre:

        (fibLouis (n+1), fibLouis (n+1) + fibLouis (n))
=       (fibLouis (n+1), fibLouis ((n+2)-1) + fibLouis ((n+2)-2))     (by arith.)
=       (fibLouis (n+1), fibLouis (n+2))                              (by fibLouis.3)
```

Eine weitere Problematik ist folgende gewesen:

```
Ihr definiert: P := aux n = (fibLouis n, fibLouis (n+1))
Ihr wollte beweisen: forall n:Nat.P

Wenn euer Base Case (gleiches Problem auch im Step Case) so aussieht,
habt ihr ein Problem:

    aux 0  =  (0,1)                         (by aux.1)
           =  (fibLouis 0, 1)               (by fibLouis.1)
           =  (fibLouis 0, fibLouis 1)      (by fibLouis.2)

Es fehlt ein letzter Schritt! Ihr wollt ja eigentlich zeigen, dass
aux n = (fibLouis n, fibLouis (n+1)) gilt, hier für n=0, also
aux 0 = (fibLouis 0, fibLouis (0+1)). Was aber im Beweis steht, ist:
aux 0 = (fibLouis 0, fibLouis 1), nicht das gleiche - auch wenn es trivial wirkt.
Also noch den letzten Schritt:

           =  (fibLouis 0, fibLouis 1)
           =  (fibLouis 0, fibLouis (0+1))   (by arith.)
```

Das waren aber beides typische Anfängerfehler, und ansonsten waren die Beweise sehr schön, also macht euch keine grossen Sorgen – aber achtet auf die benannten Probleme!

Abschliessend kann noch erwähnt werden, dass sich die zu beweisende Aussage ganz einfach in wenigen Zeilen direkt beweisen lässt (ohne Induktion zu verwenden), sobald man das Hilfslemma $aux\ n = (fibLouis\ n, fibLouis\ (n+1))$ per Induktion bewiesen hat (hier geht es nicht ohne Induktion).

(weiter auf der nächsten Seite →)

Induktion über Listen in Haskell:

In der letzten Woche haben wir Induktionsbeweise über natürliche Zahlen eingeführt. In dieser Woche wollen wir uns mit Induktionsbeweisen über Listen in Haskell beschäftigen. Von der grundlegenden Idee sind diese sehr ähnlich, allerdings gibt es einige Tricks und Fallen die man kennen und beachten muss. Diese sind:

- Man muss die richtige Induktionsvariable auswählen
- Man muss häufig erst ein Hilfslemma aufstellen
- Man muss häufig „generalisieren“

Das gucken wir uns gleich anhand eines Beispiels genauer an. Zunächst die allgemeine Syntax für Induktionsbeweise, dieses Mal im CYP-Style:

```
Lemma: <was ihr beweisen wollt>
Proof by induction on List xs
Case []
    Show: <Base Case>
    Proof
        ...
    QED

Case z:zs
    Fix z, zs
    Assume
        IH: <eure Induktionshypothese>
    Then Show: <was ihr im I.S. zeigen wollt>
    Proof
        ...
    QED
QED
```

Ganz wichtig, eure Prädikate, die an die „grünen“ Stellen kommen, dürfen kein“=“ enthalten, sondern: „.=.“ (kp warum ehrlich gesagt).

Hier ein Beispiel mit der korrekten Syntax aus der Übungsstunde (zu finden auch auf CodeExpert, Demoaufgaben CYP):

(weiter auf der nächsten Seite →)

```
Lemma: length (xs ++ ys) .=. length xs + length ys
Proof by induction on List xs
Case [] -- Base case
    Show: length ([] ++ ys) .=. length [] + length ys
    Proof
                                length ([] ++ ys)
        (by def ++)          .=. length ys

                                length [] + length ys
        (by def length)      .=. 0 + length ys
        (by arith)           .=. length ys
    QED

Case z:zs -- Induction step
    Fix z, zs
    Assume
        IH: length (zs ++ ys) .=. length zs + length ys
    Then Show: length ((z : zs) ++ ys) .=. length (z : zs) + length ys
    Proof
                                length ((z:zs) ++ ys)
        (by def ++)          .=. length (z : (zs ++ ys))
        (by def length)      .=. 1 + length (zs ++ ys)
        (by IH)              .=. 1 + (length zs + length ys)

                             length (z:zs) + length ys
        (by def length)      .=. (1 + length zs) + length ys
        (by arith)           .=. 1 + (length zs + length ys)
    QED
QED
```

Nun wollen wir uns ein Beispiel ansehen, anhand dessen wir erkennen, wieso wir die oben genannten Tricks benötigen.

Dafür definieren wir zunächst folgende Haskell-Funktionen:

```haskell
rev :: [a] -> [a]
rev []     = []                         -- rev.1
rev (x:xs) = rev xs ++ [x]              -- rev.2

qrev :: [a] -> [a] -> [a]
qrev [] ys     = ys                     -- qrev.1
qrev (x:xs) ys = qrev xs (x:ys)         -- qrev.2
```

(weiter auf der nächsten Seite →)

Wir wollen nun folgendes beweisen: $\forall xs :: [a] \,.\, \text{rev xs} = \text{qrev xs []}$
Wir verwenden dafür zunächst unseren allgemeinen Ansatz:

```
Lemma: rev xs .=. qrev xs []
Proof by induction on List xs
Case []
    Show: rev [] .=. qrev [] []
    Proof
                                        rev []
                (by def rev)      .=. []
                (by def qrev)     .=. qrev [] []
    QED
Case y:ys
    Fix y, ys
    Assume
        IH: rev ys .=. qrev ys []
    Then Show: rev (y:ys) .=. qrev (y:ys) []
    Proof
                                        rev (y:ys)
                (by def rev)      .=. rev ys ++ [y]
                (by IH)           .=. qrev ys [] ++ [y]
                                  ? ? ?
-- Vielleicht andersherum ?:
                                        qrev (y:ys) []
                (by def qrev)     .=. qrev ys [y]
                                  ? ? ?
```

Wir kommen mit diesem Proof nicht weiter. Egal von welcher Seite wir versuchen, den Step-Case zu beweisen, haben wir keine Chance, den Beweis fortzusetzen, weil uns die Definitionen/Regeln fehlen. Genauer: Wir können $\forall x :: a \,.\, \forall xs :: [a] \,.\, rev\ xs + + [x] = qrev\ xs\ [x]$ nicht beweisen.

Nun zu dem Trick (bzw. wie ich bei so einer Aufgabe vorgehen würde): Ich versuche mir immer erst einmal die Induktion (insbesondere den Step-Case) einmal im Kopf durchzudenken und zu überlegen, an welchem Punkt ich an ein Problem kommen könnte. Hier ist das Problem offensichtlich, dass ich $rev\ xs + + [x] = qrev\ xs\ [x]$ nicht beweisen kann. Also verwende ich folgenden Trick: Ich überlege mir ein „Extra-Lemma“, dass sich beweisen lässt, und beweise es extern: Hier wollen wir $rev\ xs + + [x] = qrev\ xs\ [x]$ zeigen können, das können wir als Extra (oder Hilfs-) Lemma definieren:

$$\forall xs :: [a] \,.\, \forall zs :: [a] \,.\, rev\ xs + + zs = qrev\ xs\ zs$$

Um so ein Hilfs-Lemma aufzustellen, hilft es, sich immer anzugucken, woran der „naive“ Approach scheitert und wie genau die Funktionen, die man in seinem Beweis benutzt, überhaupt funktioniert.

Ich gebe jetzt hier einen ganz inoffiziellen Tipp, der aber so gut wie immer hinkommt: Wenn eine Funktion, die ihr in eurem Prädikat P benutzt für allgemeine Listen definiert ist, z.B. hier $qrev\ xs\ zs$, ihr in eurem Prädikat aber eine der beiden Listen durch die leere Liste (oder bspw. bei natürlichen Zahlen eine Funktion $f\ n\ k = \cdots$ im Prädikat durch $f\ n\ 0$ ersetzt wird) oder ein anderes „spezifisches

Objekt“ ersetzt wird, funktioniert der naive Approach nicht und ihr müsst euch ein Hilfslemma überlegen, bei dem im Prädikat das „spezifische Objekt“ generalisiert wird, also für eine beliebige Liste oder eine beliebige natürliche Zahl gilt. Dabei geht man so vor, dass man in seinem Prädikat das „spezifische Objekt“ durch den allgemeinen Typen ersetzt (also hier in unserem Beispiel $qrev\ xs\ []$ durch $qrev\ xs\ zs$ für ein allg. zs ersetzt und sich überlegt, wie man die „andere Seite“ der Gleichung im Prädikat anpassen muss:

Dafür guckt man sich am besten die Funktionen an und überlegt, was genau diese machen. Konkret hier in diesem Beispiel: $qrev$ nimmt sich (so lange möglich), das je erste Element vom ersten Argument, der ersten Liste xs und hängt es **vorne** an das zweite Argument, die Liste zs an.

Dabei wird natürlich die Liste xs „reversed“ vorne an ys rangehängt, also macht $qrev\ xs\ zs$ eigentlich folgendes: $vorne\_ranhängen(reversed(xs), zs)$ und – naja, das ist jetzt bestimmt klar – rev reversed einfach eine beliebige Liste xs, also können wir „$vorne\_ranhängen(reversed(xs), zs)$“ auch folgendermassen mit rev ausdrücken: $rev\ xs\ +\!+\ zs$, denn aus meiner ersten Übungsstunde wissen wir, dass „xs ++ zs“ einfach xs „vorne an zs ranhängt“ (oder wie der Profi sagt: „diE liStEn xs uNd ys kOnKatTiNieRt“).

Also wissen wir, was $qrev\ xs\ zs$ macht und wie wir das mit rev ausdrücken können, nämlich:

$$qrev\ xs\ ys = rev\ xs\ +\!+\ zs$$

So und wir wollten das für ein allgemeines ys beweisen (deshalb ja auch der ganze Aufriss), also schreiben wir:

$$P := \forall ys :: [a]\ .\ qrev\ xs\ zs = rev\ xs\ +\!+\ zs$$

Und dieses Prädikat beweisen wir nun nach gewohntem Schema. Wichtig: Jetzt haben wir ein „forall“ in unserem Prädikat. Wie drücken wir das in CYP aus? Mit dem Wörtchen „generalizing“, also so: "Proof by induction on List xs generalizing zs“.

Der fertige Proof sieht dann so aus:

```
Lemma aux: rev xs ++ zs .=. qrev xs zs
Proof by induction on List xs generalizing zs
Case []
  For fixed zs
  Show: rev [] ++ zs .=. qrev [] zs

  Proof
                                    rev [] ++ zs
    (by def rev)                .=. [] ++ zs
    (by def ++)                 .=. zs
    (by def qrev)               .=. qrev [] zs
  QED

Case y:ys
  Fix y, ys
  Assume
    IH: forall zs: rev ys ++ zs .=. qrev ys zs
  Then for fixed zs
  Show: rev (y:ys) ++ zs .=. qrev (y:ys) zs

  Proof
                                    rev (y:ys) ++ zs
    (by def rev)                .=. (rev ys ++ [y]) ++ zs
    (by app_assoc)              .=. rev ys ++ ([y] ++ zs)
    (by def ++)                 .=. rev ys ++ (y : ([] ++ zs))
    (by def ++)                 .=. rev ys ++ (y:zs)
    (by IH)                     .=. qrev ys (y:zs)
    (by def qrev)               .=. qrev (y:ys) zs
  QED
QED
```

Hier noch ein weiteres, vergleichbar einfaches Beispiel zur Übung:

```
Lemma: foldr (:) [] xs .=. xs
Proof by induction on List xs
Case []
  Show: foldr (:) [] [] .=. []

  Proof
                                          foldr (:) [] []
    (by def foldr)                  .=. []
  QED

Case y:ys
  Fix y, ys
  Assume
    IH: foldr (:) [] ys .=. ys
  Then Show: foldr (:) [] (y:ys) .=. y:ys

  Proof
                                          foldr (:) [] (y:ys)
    (by def foldr)                  .=. y : (foldr (:) [] ys)
    (by IH)                         .=. y : ys
  QED
QED
```

Ein paar foldr-Übungen:

```haskell
and' xs = foldr (&&) True xs

or' xs = foldr (||) False xs

sum' xs = foldr (+) 0 xs

product' xs = foldr (*) 1 xs

concat' xs = foldr (++) [] xs

all' p xs = foldr ((&&).p) True xs

any' p xs = foldr ((||).p) False xs

scanl' = undefined -- Nächste Stunde ;)
```

Higher-Order-Programming in Haskell:

Vorneweg ein neues Mantra (nach „Implication associates to the right”):

"**Function Application associates to the left.”**

Bis jetzt haben wir immer so getan, als würden unsere Funktionen mehrere Argumente nehmen können, bspw. $sum\ a\ b = a + b$ – eine Funktion, die scheinbar die beiden Argumente a, b nimmt und deren Summe zurückgibt. Tatsächlich gibt es das in Haskell aber nicht: Jede Funktion kann höchstens ein Argument entgegennehmen. Aber wie funktioniert dann $sum\ a\ b = a + b$?

$$sum\ a\ b = (sum\ a)\ b \quad - \text{gem. Assoc. von Func. Application}$$

Das bedeutet folgendes: Wenn wir bspw. $sum\ 2\ 4$ schreiben, wird erst die Funktion $(sum\ 2)$ erstellt, die den Parameter 2 entgegennimmt und eine Funktion zurückgibt, die ein Argument nimmt und dieses mit 2 addiert, also eigentlich ist $sum\ 2 = (2 + \_)$.

Also wird die Funktion $sum\ 2$ zurückgegeben, die ein Argument nimmt und dieses mit 2 addiert, und da wir als „zweites“ Argument 4 gegeben haben, steht da also: $(sum\ 2)\ \ 4$, also übergeben wir der Funktion "$sum\ 2$" nun die 4 und wir wissen: $(sum\ 2)\ \ 4\ \ =\ \ 6$.

Falls das nun noch unintuitiv ist, kann man es sich auch folgendermassen denken: $(sum\ 2)$ ist wie eine ganz neue Funktion, nennen/definieren wir sie $addwithTwo\ x = 2 + x$. Wenn wir nun $addwithTwo\ 4$ schreiben, sehen wir direkt, was das Ergebnis ist (6). Und das passiert auch, wenn wir eine Funktion schreiben, die (logisch betrachtet) mehrere Argumente nimmt: Die Funktion „schluckt“ das erste Argument und gibt dann eine neue Funktion zurück. Diese neue Funktion hat keinen Namen und wir schreiben sie bspw. bei $add\ a\ b$ einfach als $(add\ 2)\ b\ \ = 2 + b$, aber theoretisch könnten wir ihr auch einen Namen geben (wie vorhin mit addwithTwo).

Gucken wir uns mal den type von sum an:

$$(Num\ a) \Rightarrow a \rightarrow a \rightarrow a$$

Nun gilt für diesen Pfeil bei types die gleiche Assoziativität wie bei Implikationen, also steht da:

$$(Num\ a) \Rightarrow a \rightarrow (a \rightarrow a)$$

Und das passt zu unserer neuen Erklärung: sum ist also eine Funktion, die ein Argument vom Typen a (wobei a ein Numerischer Typ ist) entgegennimmt und eine Funktion zurückgibt (Erinnere dich: Funktionen in Haskell dürfen auch Funktionen zurückgeben), die ein weiteres Argument vom Typen a entgegennimmt und ein Return-Value vom Typen a zurückgibt.

Wozu das Ganze nun? Wir können Funktionen absichtlich mit weniger Parametern aufrufen, um keine „fertigen“ Ergebnisse zu erhalten, sondern Funktionen zurückzuerhalten, wie vorhin bei addWithTwo. Zeit für ein Beispiel (aus learnyouahaskell.com):

```haskell
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z

-- Nun erhalten wir für multThree 9 eine Funktion, die "zwei Argumente nimmt" (tut sie ja
-- nicht wirklich, wie wir mittlerweile wissen) und das Produkt der beiden Argumente, multipliziert mit 9
-- zurückgibt

multTwoArgumentsWithNine x y = multThree 9 x y

-- Oder ganz einfach:

multTwoArgumentsWithNine = multThree 9

-- Wir könnten auch eine Funktion erstellen, die ein Argument nimmt, und das Produkt dieses Arguments
-- mit 18 zurückgibt, also:

multOneArgumentWithEighteen = multThree 9 2

-- Oder wieder:

multOneArgumentWithEighteen = multTwoArgumentsWithNine 2
```

Der einzige Weg, wie wir in Haskell wirklich zwei oder mehrere Argumente „auf einmal“ übergeben können, ist, indem wir sie als Tupel übergeben:

$$sum\ (a, b) = a + b$$

Um damit besser arbeiten zu können, gibt es in der Prelude die Funktion curry und uncurry:

```haskell
-- func1 nimmt eigentlich Tuple, aber arbeitet mit einer curried Function
-- uncurry verwandelt die curried function in eine uncrried function:
-- uncurry :: (a -> b -> c) -> ((a,b) -> c)
-- also können wir die curried function mit uncurry auch mit tuples benutzen
func1 :: Num c => (c, c) -> c
func1 (x,y) = uncurry (\x y -> x + y) (x,y)

-- Hier ist es genau umgekehrt:
func2 :: Num c => c -> c -> c
func2 x y = curry (\(x,y) -> x + y) x y
```

So und jetzt verstehen wir auch, wieso wir bspw. so etwas schreiben dürfen:

```haskell
func3 :: (Num a) => [a] -> [a]
func3 = map (+2)
```

Eigentlich braucht die Funktion „map“ ja eine Funktion und eine Liste. Aber wir wissen jetzt, dass bei zwei Argumenten eigentlich folgendes passiert:

$$map\ f\ xs = (map\ f)\ xs$$

Das heisst eigentlich wird erst eine Funktion $(map\ f)$ erstellt, die eine Liste nimmt und dann eine modifizierte Liste (jedes Element mit f „bearbeitet“) zurückgibt. Das heisst, wir können auch einfach nur $(map\ f)$ schreiben – und wissen, dass es sich um eine Funktion handelt, die eine Liste nimmt und eine Liste zurückgibt. Wir haben also eine eigene coole Funktion erstellt, indem wir einer anderen Funktion einfach nicht genug Argumente gegeben haben. So können wir in funktionalen Programmiersprachen sehr praktisch bereits gebaute Funktionen „anders verwenden“ indem wir quasi nur „Teile“ von ihnen verwenden.

Beim obigen Beispiel mit $func3$ wissen wir nun, dass $func3$ eine Funktion ist, die eine Liste aus numerischen Werten entgegennimmt und eine Liste mit numerischen Werten zurückgibt, wobei jeder Wert mit 2 addiert wurde.

Typen von Haskell Funktionen bestimmen – Grundlagen:

Mit ghci können wir ganz einfach mit dem command „:t <function>“ den typ einer Funktion bestimme, bspw.: $:t\ (\backslash x\ y \rightarrow x + y)$ gibt uns: (Num a) => a -> a -> a, wie erwartet.

(weiter auf der nächsten Seite →)

Nun wollen wir das gleiche aber auch alleine und von Hand schaffen. Das lernen wir am Besten mit ein paar Beispielen:

```
Weitere Beispiele:

\x -> \y -> x + y

Können wir umschreiben als:
\x y -> x + y

Wir wissen:
(+) :: (Num a) => a -> a -> a

Also folgt:
x :: (Num a) => a
y :: (Num a) => a

Also:
\x -> \y -> x + y :: (Num a) => a -> a -> a
------------------------------------------------------------

filter (<3)

Wir wissen:
filter :: (a -> Bool) -> [a] -> [a]
(<) :: (Ord a) => a -> a -> Bool
3 :: (Num a) => a

(Das würde man aus Zeitgründen normalerweise hier bei so einer einfachen Aufgabe
einfach weglassen, aber soll hier der Vollsätndigkeit halber dennoch getan werden):
Wir benennen die Variablen um, damit wir leichter "matchen" können:

filter :: (a -> Bool) -> [a] -> [a]
(<) :: (Ord b) => b -> b -> Bool
3 :: (Num c) => c

Nun fangen wir mit der "äussersten" Funktion an:

filter nimmnt als Argument eine Funktion vom Typen (a->Bool) und gibt eine Funktion
vom Typen [a] -> [a] zurück. An Filter übergeben wir das Argument (<3), das wir uns
nun noch genauer angucken werden:

(<) nimmt ein Argument vom Typen a (es muss einer geordenten Menge entspringen)
und gibt eine Funktion zurück, die ein Argument vom gleichen Typen a nimmt und ein
Bool zurückgibt. An (<) übergeben wir das Argument 3, vom Typen (Num a) => a, also:

(<3) :: (Ord c, Num c) => c -> Bool

Wenn wir das mit dem Typen des Arguments von Filter "matchen", sehen wir:
a :: (Num c) => c   Bool :: Bool

Also:
filter (<3) :: (Ord c, Num c) => [c] -> [c]
```