FMFP Exercise Session 4

LEE C 114, Wed 8-10

12. March 2025

Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.
Bei Fehlern: robin.ebersberger@inf.ethz.ch
Inspiriert von Maximilian Schlegels Notizen aus 2023.

1 Exercise Sheet - Common Mistakes

1.1 Induction

Insgesamt sehr gut gelöst worden. Unten findet ihr nochmal das Pattern von den Slides. Die häufigsten Fehler waren:

- Fehlende Klammern: fibLouis m+1 $=$ (fibLouis $m) + 1 \neq$ fibLouis $(m + 1)$
- Fehlende side-condition bei der ND Rule zu Induktion
- Substitution: Grossbuchstaben stehen i.A. für beliebige Formeln, z.B. $P \equiv \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
  Jedenfalls im FP-Teil der Vorlesung schreiben wir nicht $P(n)$, sondern $P[n \mapsto ...]$. Das ist wieder Substitution (der freien Variable n), wie wir das in den ersten Wochen gesehen haben
- IH unklar aufgeschrieben: Im Induktionsschritt zeigt ihr: $\forall m \in \mathbb{N}.(P[n \mapsto m] \to P[n \mapsto m + 1])$
  Dh. ihr zeigt für ein beliebiges $m \in \mathbb{N}$, dass $P[n \mapsto m + 1]$ hält, angenommen $P[n \mapsto m]$ hält.
  Etwas wie „Assume $P[n \mapsto m] \equiv ...$ holds for all $m \in \mathbb{N}$. Show $P[n \mapsto m + 1]$" klingt sehr nach $(\forall m \in \mathbb{N}.P[n \mapsto m]) \to P[n \mapsto m + 1]$, also angenommen $P[n \mapsto m]$ hält für beliebiges $m \in \mathbb{N}$, dann wird auch $P[n \mapsto m + 1]$ für irgendein $m + 1$ halten. Also nehmt ihr an, dass $\forall n.P$ hält, um $\forall n.P$ zu beweisen…
- Kombinieren von Regeln und arithmetischen Umformungen

Auf den letzten Punkt möchte ich kurz genauer eingehen. Ihr könnt in den Schritten eures Beweises die gegebenen Regeln und natürlich arithmetische Umformungen (wie $x + 0 = x$) verwenden. Allerdings solltet ihr nicht zu viele arithmetische Umformungen in einem Schritt machen. Was „zu viele" hier bedeutet, liegt ein wenig im Auge des Betrachters.

Was aber nicht passieren sollte, ist das Kombinieren von arithmetischen Umformungen und gegebenen Regeln. Hier nochmal aus Serie 3:

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

Einige haben nun das folgende versucht:

$$\begin{aligned} & \text{fibLouis } m + \text{fibLouis } (m+1) \\ = \; & \text{fibLouis } (m+2) && \text{(by fibLouis.3)} \end{aligned}$$

Allerdings lässt sich die fibLouis.3 Regel hier nicht anwenden. Zuerst muss noch mindestens eine arithm. Umformung gemacht werden:

$$\begin{aligned}
& \text{fibLouis } m + \text{fibLouis } (m+1) \\
= \; & \text{fibLouis } (m+1) + \text{fibLouis } m && \text{(by arith.)} \\
= \; & \text{fibLouis } ((m+2)-1) + \text{fibLouis } ((m+2)-2) && \text{(by arith.)} \\
= \; & \text{fibLouis } (m+2) && \text{(by fibLouis.3)}
\end{aligned}$$

2 Induction over Lists - Generalization

2.1 Beispiel

Betrachten wir die folgenden Regeln:

rev [] = []	(rev.1)
rev (x:xs) = rev xs ++ [x]	(rev.2)
qrev [] ys = ys	(qrev.1)
qrev (x:xs) ys = qrev xs (x:ys)	(qrev.2)
(++) [] ys = ys	(app.1)
(++) (x:xs) ys = x : (xs ++ ys)	(app.2)
$\forall$xs $\in [a]$. xs ++ [] = xs	(app_Nil)
$\forall$xs, ys, zs $\in [a]$.(xs ++ys) ++ zs = xs ++(ys ++zs)	(app_assoc)

Unser Ziel ist es, folgendes zu beweisen: $\forall$xs $\in [a]$. rev xs = qrev xs []
Wir fangen wie gewohnt an:

Let $P \equiv$ rev xs = qrev xs []
Prove $\forall$xs $\in [a].P$ by induction on xs.
Base Case: Show P[xs $\mapsto$ []] ...
Step Case: Let y :: a, ys :: $[a]$ be arbitrary but fixed. Show P[xs $\mapsto$ y : ys] assuming IH: P[xs $\mapsto$ ys]

$$\begin{aligned}
& \text{rev (y:ys)} \\
= \; & \text{rev ys ++ [y]} && \text{(by rev.1)} \\
= \; & \text{qrev ys [] ++ [y]} && \text{(by IH)} \\
= \; & ?
\end{aligned}$$

Hier kommen wir nicht weiter. Probieren wir mal die andere Richtung:

$$\begin{aligned}
& \text{qrev (y:ys) []} \\
= \; & \text{qrev ys (y:[])} && \text{(by qrev.2)} \\
= \; & \text{qrev ys [y]} && \text{(by (:))} \\
= \; & ?
\end{aligned}$$

Auch hier geht es nicht weiter... Was nun?

2.2 Beispiel Generalization

Statt $\forall$xs $\in [a]$. rev xs = qrev xs [] beweisen wir erstmal eine allgemeinere Aussage, nämlich

Lemma 1: $\forall$xs $\in [a]\forall$zs $\in [a]$. rev xs ++ zs = qrev xs zs

Let $P' \equiv \forall$zs $\in [a]$. rev xs ++ zs = qrev xs zs
Show $\forall$xs $\in [a].P'$ by induction on xs (generalizing zs).
Let zs :: $[a]$ be arbitrary but fixed.
Base Case: Show P'[xs $\mapsto$ []] ...
Step Case: Let Let y :: a, ys :: $[a]$ be arbitrary but fixed. Show P'[xs $\mapsto$ y : ys] assuming IH: P'[xs $\mapsto$ ys]
...

Prove $\forall$xs $\in [a]$. rev xs = qrev xs []
Let xs :: $[a]$ be arbitrary but fixed. Then

rev xs
= rev xs ++ [] (by app_Nil)
= qrev xs [] (by Lemma 1)

Hier fallen direkt drei Sachen auf:

1. Es ist eine neue Variable zs dazugekommen, die in P' an einen $\forall$-Quantifier gebunden ist.
2. In unserer Induktion müssen wir die Aussage also für alle zs $\in [a]$ beweisen. Deshalb wählen wir direkt am Anfang ein zs $\in [a]$ arbitrary but fixed.
3. Der Beweis von $\forall$xs $\in [a]$. rev xs = qrev xs [] am Ende ist mithilfe des Lemmas sehr kurz. Das führt dazu, dass er gerne vergessen wird. Da das ganze Ziel der Aufgabe aber der Beweis dieser Aussage war, sollte das nicht passieren.

2.3 Tipp für Generalization

Der beste Ansatz bei Induktionsbeweisen ist, den Beweis (insbesondere den Step-Case) zuerst einmal im Kopf durchzudenken, um zu schauen, ob sich ein Problem ergeben könnte. Im Beispiel oben hätte man das Problem vielleicht direkt am Anfang sehen können, ohne überhaupt etwas aufzuschreiben.

Dann geht es um Verständnis: Warum scheitert der Beweis? Was müsste ich anpassen, damit er funktioniert? Brauche ich eventuell ein Hilfslemma?

Der **höchst inoffizielle Tipp zu Generalization** ist:
Schaut euch die zu beweisende Aussage ganz genau an, insbesondere euer P. Gibt es darin eine Funktion, die zwar allgemeine Inputs (z.B. beliebige Listen, Trees, Integer) annimmt, aber im P an einer Stelle immer eine Konstante (z.B. [], Leaf, 0) bekommt? Dann müsst ihr oft Generalisieren.
Der Prozess ist: Ersetze die Konstante durch eine freie Variable. Betrachte die andere Seite des „=“ in P und überprüfe, ob sich etwas verändern muss, damit die Gleichheit weiter hält. Dann folge dem Schema!

Am Beispiel von oben: $P \equiv$ rev xs = qrev xs []
Wir sehen direkt, dass qrev :: $[a] \to [a] \to [a]$ ist, aber in P als zweites Argument immer eine Konstante (nämlich []) bekommt. Mit der Heuristik von oben ersetzen wir [] durch eine neue Variable zs:
$P' \equiv$ rev xs = qrev xs zs
Jetzt überprüfen wir, ob sich an der anderen Seite des „=“ etwas ändern muss, damit „das = wieder Sinn ergibt“. Die Antwort ist Ja, wir erhalten:
$P'' \equiv$ rev xs ++ zs = qrev xs zs
Wie kommt man darauf? Hier braucht man ein Verständnis für die Funktionsweise von qrev. Das erhält man entweder durch „Anstarren“, oder man probiert einige Inputs aus, z.B. qrev [1,2,3] [] oder auch qrev [1,2,3] [4,5,6]

3 Haskell

3.1 Folding

Hier nur nochmal aus den Vorlesungs & Exercise Session Slides:

`foldr`: right-associative fold

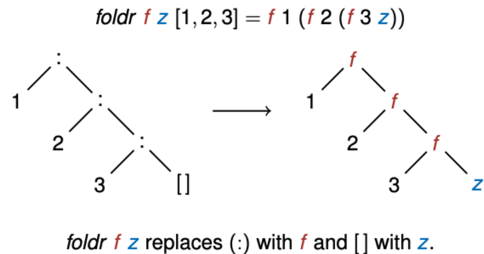
$$foldr\ (\oplus)\ e\ [l_1, l_2, \dots, l_n] = l_1 \oplus (l_2 \oplus \cdots \oplus (l_n \oplus e))$$

```
foldr           :: (a -> b -> b) -> b -> [a] -> b

foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

`foldl`: left-associative fold

$$foldl\ (\oplus)\ e\ [l_1, l_2, \dots, l_n] = ((e \oplus l_1) \oplus l_2) \oplus \cdots \oplus l_n$$

```
foldl           :: (b -> a -> b) -> b -> [a] -> b

foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```



foldr und foldl sind beides higher-order functions. Das heisst einfach, dass sie als mind. 1 Argument eine weitere Funktion nehmen.
Ich stelle mir das ganze so vor: Die Funktion „faltet/schiebt“ die Liste von links (foldl) bzw. rechts (foldr) zusammen, wobei zusätzlich das Anfangselement (oben e) dazu kommt.

Gerade foldr ist extrem mächtig, darauf kommen wir nächste Woche zu sprechen. Als kleiner Fun-Fact schonmal: foldr kann (manchmal) mit infinite lists umgehen, foldl (und auch foldl’) können das nicht.

3.2 Partial Application II

Wir erinnern uns von letzter Woche:

ghci> tenBy = (/) 10
ghci> tenBy 5
2.0

Hier ist (/) :: $(a \to (a \to a))$ und ((/) 10) :: $a \to a$
Ausserdem erinnern wir uns daran, dass alle Haskell Funktionen eigentlich nur 1 Argument nehmen und dann potentiell eine andere Funktion zurückgeben, die weiter angewendet wird.

Nun ist folgendes dazu gekommen:

ghci> byFive = (/ 5)
ghci> byFive 10
2.0

Das funktioniert mit fast allen Infix-Operatoren (wie +, *, /, ...). Die Idee ist einfach, dass wir sozusagen eine Lücke in dem (/ 5) haben, die ein Argument erwartet.
Einigen von euch ist direkt aufgefallen, dass das nicht echte Partial Application sein kann, da
(/) :: $(a \to (a \to a))$ und wir hier den Wert des „zweiten“ Arguments mit 5 belegen, obwohl Haskell Funktionen doch eigentlich nur ein Argument nehmen. Interessierte verweise ich auf den Haskell Wiki Eintrag zu Sectioning. TLDR: Es ist wieder einmal „syntactic sugar“.

3.3 Curry und Uncurry

Der einzige Weg, wie wir in Haskell wirklich mehr als 1 Argument auf einmal übergeben können, sind Tuples:

sum :: Num a ⇒ $(a, a) \to a$
sum (x,y) = x+y

Wir unterscheiden zwischen curried und uncurried functions. Hier ist sum eine uncurried function, da sie mit Tupeln arbeitet. Curried functions sind z.B. die Funktionen, die ihr schon die ganze Zeit verwendet.

Es gibt einen einfachen Weg, zwischen curried und uncurried zu wechseln:

curry :: $((a, b) \to c) \to a \to b \to c$
uncurry :: $(a \to b \to c) \to (a, b) \to c$

ghci> :t (curry sum)
(curry sum) :: Num c $\Rightarrow c \to c \to c$

3.4 Type Inference

Type Inference ist einfach der Prozess, mit dem der Typ eines Ausdrucks (systematisch) bestimmt wird. Wir schauen uns nächste Woche einen Algorithmus an, der immer funktioniert.
Ich empfehle aber, es zuerst mit „Anstarren und Nachdenken" zu probieren, hier ein Beispiel von Max, wie das aussehen könnte.

```
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]
```