Übung 9

1. Aufgabe

a) `zipWith (&&) :: [Bool] -> [Bool] -> [Bool]`

`zipWith (&&) xs ys` verknüpft die beiden Argumentlisten elementweise, und zwar so, daß die Elemente der Ergebnisliste die Konjunktion der entsprechenden Elemente der Argumentliste enthält, denn:

`zipWith :: (a –> b -> c) -> [a] -> [b] -> [c]`

`zipWith f xs ys` verknüpft mittels `f` die beiden Argumentlisten elementweise zu einer Liste vom Typ der Funktionswerte

`(&&) :: Bool -> Bool -> Bool; (&&) ist die logische Konjunktion`

b) `map (==2) :: Num a => [a] –> [Bool]`

`map (==2) xs` bildet die Liste `xs` elementweise auf eine Liste von Booleschen Werten ab: diese geben an, ob das entsprechende Element von `xs == 2` ist, denn:

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

`map f xs` bildet die Liste `xs` elementweise mittels `f` auf eine Liste `ys` ab, die aus den Werten von `f` für die Elemente von `xs` besteht

`(==2) :: Num a => a -> Bool`

`(==2)` bildet Zahlen `x` ab auf den Booleschen Wert `(x == 2)` (2 gehört zu allen Zahlentypen)

c) `filter ( \x -> (x*x==9) ) :: Num a => [a] -> [a]`

diese Funktion liefert zu einer Liste `xs` von Zahlen die Liste derjenigen Elemente von `xs`, deren Quadrat `== 9` ist, denn:

`filter :: (a -> Bool) -> [a] -> [a]`

`filter p xs` liefert die Liste derjenigen Elemente von `xs,` für die `p` gilt.

`\x -> (x*x==9) :: Num a => a -> Bool`

`\x -> (x*x==9)` bildet Zahlen `x` auf `True` ab, wenn `x*x == 9` gilt (wenn also `x==3` oder `x==(-3)`), sonst auf False (die Typbeschränkung ergibt sich aus der Definition von „`*`“)

d) `((\f g -> g.f) (\x -> x*x) (\x -> x+1)) 2 :: Num a => a`

der Wert des Ausdrucks ist `5 = 2*2 + 1 = (\x -> x*x +1) 2,` denn:

`\f g -> g.f :: (a -> b) -> (b -> c) -> a -> c`

die Funktion ist also die Hintereinanderausführung ‘in umgekehrter Reihenfolge’

`\x -> x*x :: Num a => a -> a;` die Funktion bildet Zahlen auf ihr Quadrat ab

`\x -> x+1 :: Num a => a -> a;` die Funktion bildet Zahlen auf ihren Nachfolger ab (die Typbeschränkung ergibt sich aus der Definition von „`*`“ und „`+`“ sowie der Zugehörigkeit von 1 zu allen Zahltypen), also:

`(\f g -> g.f) (\x -> x*x) (\x -> x+1) :: Num a => a -> a`

$$\texttt{(\textbackslash f g -> g.f)}\ \underbrace{\texttt{(\textbackslash x -> x*x)}}_{\texttt{f}}\ \underbrace{\texttt{(\textbackslash x -> x+1)}}_{\texttt{g}} = \underbrace{\texttt{(\textbackslash x -> x+1)}}_{\texttt{g}}.\underbrace{\texttt{(\textbackslash x -> x*x)}}_{\texttt{f}}$$

– das ist die Funktion, die erst quadriert und dann den Nachfolger bildet, also `\x -> x*x + 1`

2. Aufgabe

a) Zu zeigen:

```
filter p (filter q xs) = filter (p &&& q) xs
   where
   p &&& q = \x -> (p x && q x)                        (&&&)
```

Wir benötigen die folgenden Definitionen:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []                                       (f0)
filter p (x:xs)
   | p x       = x : filter p xs                       (f1)
   | otherwise = filter p xs                           (f2)

(&&) :: Bool -> Bool -> Bool
True && x = x                                          (&&1)
False && _ = False                                     (&&2)
```

Wir beweisen die Behauptung durch Induktion über die Struktur von `xs`.

Induktionsanfang: Ist `xs = []`, so gilt die Behauptung, denn :

```
filter p (filter q xs) = filter p (filter q []) = filter p [] = []    (f0, 2 mal)
filter (p &&& q) xs = filter (p &&& q) [] = []      (f0)
```

Andernfalls gibt es `y` und `ys` mit `xs = (y:ys)`. Angenommen, die Behauptung ist für `ys` (genauer: für alle Listen mit der Länge von `ys`) bereits bewiesen (*Induktionsvoraussetzung*).

Für den *Induktionsschritt* unterscheiden wir dann drei Fälle:

Fall 1: `q y && p y`, also auch `(p &&& q) y` nach (&&&) und Kommutativität von `(&&)`. Es folgt:

```
filter p (filter q xs) = filter p (filter q (y:ys))
                       = filter p (y : filter q ys)   (f1), q y
                       = y : filter p (filter q ys)   (f1), p y
filter (p &&& q) xs    = filter (p &&& q) (y:ys)
                       = y : filter (p &&& q) ys      (f1), (p &&& q) y
```

Mit der Induktionsvoraussetzung folgt dann die Behauptung für diesen Fall.

Fall 2: `q y && not p y`, also `False && q y`, d.h.: `(p &&& q) y == False` nach (&&&), (&&2) und Kommutativität von `(&&)`. Es folgt:

```
filter p (filter q xs) = filter p (filter q (y:ys))
                       = filter p (y : filter q ys)   (f1), q y
                       = filter p (filter q ys)       (f2), not p y
filter (p &&& q) xs    = filter (p &&& q) (y:ys)
                       = filter (p &&& q) ys          (f2), not (p &&& q) y
```

Mit der Induktionsvoraussetzung folgt die Behauptung dann auch für diesen Fall.

Fall 3: `not q y`, also `(p &&& q) y == False` nach (&&&) und (&&2). Es folgt:

```
filter p (filter q xs) = filter p (filter q (y:ys))
                       = filter p (filter q ys)       (f2), not q y
filter (p &&& q) xs    = filter (p &&& q) (y:ys)
                       = filter (p &&& q) ys          (f2), not (p &&& q) y
```

Mit der Induktionsvoraussetzung folgt die Behauptung dann auch für diesen Fall (p spielt in diesem Fall offensichtlich keine Rolle).

Damit ist die Behauptung bewiesen.

b) Zu zeigen:

```
filter p . map f = map f . filter (p . f)
```

Neben der Definition von filter aus a) benötigen wir noch die folgenden Definitionen:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []                                (m0)
map f (x:xs) = f x : map f xs                (m1)

(.) :: (b –> c) –> (a -> b) -> a -> b
f . g x = f (g x)                            (h)
```

Wir beweisen die Behauptung wieder mit Induktion über die Struktur von Listen xs, die zum Typ von f passen.

Induktionsanfang: Ist `xs = []`, dann folgt:

```
(filter p . map f) xs        = (filter p . map f) []
                             = filter p (map f [])            (h)
                             = filter p []                    (m0)
                             = []                             (f0)
(map f . filter (p.f)) xs    = map f (filter (p.f) [])        (h)
                             = map f []                       (f0)
                             = []                             (m0)
```

Also gilt die Behauptung in diesem Fall.

Sei nun `xs = (y:ys)` und sei die Behauptung für `ys` (genauer: für alle Listen mit der Länge von `ys`) bereits bewiesen (*Induktionsvoraussetzung*). Dann folgt (*Induktionsschritt*):

```
(filter p . map f) xs        = (filter p . map f) (y:ys)
                             = filter p (map f (y:ys))        (h)
                             = filter p (f y : map f ys)      (m1)
                               ⎧f y : filter p (map f ys)     (f1), p (f y)
                             = ⎨
                               ⎩filter p (map f ys)           (f2), not p (f y)

map f . filter (p.f) xs      = (map f . filter (p.f)) (y:ys)
                             = map f (filter (p.f) (y:ys))       (h)
                               ⎧map f (y: filter (p.f) ys))      (f1), (p.f) y
                             = ⎨
                               ⎩map f (filter (p.f) ys)          (f2), not (p.f) y

                               ⎧f y : map f (filter (p.f) ys)    (m1), (p.f) y
                             = ⎨
                               ⎩map f (filter (p.f) ys)          (m1), not (p.f) y
```

Da nach (h) `p (f y) = (p.f) y` gilt, folgt die Behauptung nun unmittelbar aus der Induktionsvoraussetzung.

Außerdem benötigen wir die folgende Hilfsbehauptung, die wir durch Induktion über die Struktur von xs beweisen:

(**) `filter p (filter q xs) = filter q (filter p xs)`

Für `xs = []` ist dies offensichtlich richtig:

```
filter p (filter q xs)   = filter p (filter q [])
                         = filter p [] = []          (f0, 2 mal)
                         = filter q []               (f0, rückwärts)
                         = filter q (filter p [])    (f0, rückwärts)
                         = filter q (filter p xs)
```

Sei nun `xs = (y:ys)` und sei (**) für `ys` (genauer: für alle Listen mit der Länge von `ys`) bereits bewiesen (Induktionsvoraussetzun). Dann können wir 4 Fälle unterscheiden:

Fall 1: `q y && p y`. Es folgt:

```
filter p (filter q xs) = filter p (filter q (y:ys))
                       = filter p (y : filter q ys)   (f1), q y
                       = y : filter p (filter q ys)   (f1), p y
filter q (filter q xs) = filter q (filter p (y:ys))
                       = filter q (y : filter p ys)   (f1), p y
                       = y : filter q (filter p ys)   (f1), q y
```

Mit der Induktionsvoraussetzung folgt dann die Behauptung für diesen Fall.

Fall 2: `q y && not p y`. Es folgt:

```
filter p (filter q xs) = filter p (filter q (y:ys))
                       = filter p (y : filter q ys)   (f1), q y
                       = filter p (filter q ys)       (f2), not p y
filter q (filter q xs) = filter q (filter p (y:ys))
                       = filter q (filter p ys)       (f2), not p y
```

Mit der Induktionsvoraussetzung folgt die Behauptung auch in diesem Fall.

Fall 3: `not q y && p y.` Es folgt:

```
filter p (filter q xs) = filter p (filter q (y:ys))
                       = filter p (filter q ys)       (f2), not q y
filter q (filter q xs) = filter q (filter p (y:ys))
                       = filter q (y : filter p ys)   (f1), p y
                       = filter q (filter p ys)       (f2), not q y
```

Mit der Induktionsvoraussetzung folgt die Behauptung auch in diesem Fall.

Fall 4: `not q y && not p y.`

```
filter p (filter q xs) = filter p (filter q (y:ys))
                       = filter p (filter q ys)       (f2), not q y
filter q (filter q xs) = filter q (filter p (y:ys))
                       = filter q (filter p ys)       (f2), not p y
```

Also folgt die Behauptung auch im 4. Fall mit der Induktionsbvoraussetzung. Damit ist die Hilfsbehauptung bewiesen.