Musterlösung zur 14. Übung

Aufgabe 1. Diese Monade heißt deswegen „trivial“, da sie den Typ a einfach nur kapselt, ohne ein bestimmtes Berechnungsmuster zu repräsentieren. Trotzdem können wir hier einfach die Gesetze noch einmal rekapitulieren. Für unsere Instanzdeklaration

```
instance Monad W where
```

definieren wir die geforderten Funktionen return und $(\gg\!=)$:

- return soll einfach nur ein a „einpacken“, also setzen wir

```
return x      = W x
```

- Für den Fischschwanz betrachten wir zunächst noch einmal den Typ $(\gg\!=)$ :: W a →(a → W b) →W b. Wir bekommen also ein W a und eine Funktion, die aus a ein W b macht. Wir definieren den Fischschwanz einfach so, dass wir f auf das a hinter dem W anwenden:

```
(W x) >>= f   = f x
```

Damit haben wir die geforderten Funktionen definiert, nun überprüfen wir die Gültigkeit der Gesetze:

(i) return x $\gg\!=$ f = f x
Sei also x :: a, dann ist

$$\begin{aligned} \texttt{return x} \gg\!= \texttt{f} &= \texttt{(W x)} \gg\!= \texttt{f} && \text{Definition return} \\ &= \texttt{f x} && \text{Definition } (\gg\!=) \end{aligned}$$

(ii) y $\gg\!=$ return = y
Sei also y :: W a, dann ist y = W x für ein geeignetes x :: a, wir haben

$$\begin{aligned} \texttt{y} \gg\!= \texttt{return} &= \texttt{(W x)} \gg\!= \texttt{return} \\ &= \texttt{return x} && \text{Definition } (\gg\!=) \\ &= \texttt{W x} && \text{Definition return} \\ &= \texttt{y}. \end{aligned}$$

(iii) y $\gg\!=$ f $\gg\!=$ g = y $\gg\!=$ (\z → f z $\gg\!=$ g)
Sei wieder y = W x :: W a, wir haben dann

$$\begin{aligned} \texttt{y} \gg\!= \texttt{f} \gg\!= \texttt{g} &= \texttt{W x} \gg\!= \texttt{f} \gg\!= \texttt{g} \\ &= \texttt{f x} \gg\!= \texttt{g} && \text{Definition } (\gg\!=) \\ &= \texttt{(z→ f z} \gg\!= \texttt{g) x} && \text{Lambdanotation} \\ &= \texttt{(W x)} \gg\!= \texttt{(z→ f z} \gg\!= \texttt{g)} && \text{Definition } (\gg\!=) \\ &= \texttt{y} \gg\!= \texttt{(z→ f z} \gg\!= \texttt{g)}. \end{aligned}$$

Aufgabe 2. So, nun zu den Listen. Die selbst definierten Listen werden hier deswegen benutzt, da Haskells Standardlisten schon eine Monadeninstanz sind. Die Idee ist, dass return ein Element in eine einelementige Liste packt und der Fischschwanz die Funktion auf jedes Element der Liste anwendet und die Listen danach konkateniert (für Haskells Standardlisten ist das [bis auf die Reihenfolge der Argumente] die Funktion concatMap). Zur Definition brauchen wir für unsere Listen eine Konkatenierungsfunktion, die wir kurz definieren:

```
concat' :: List a → List a → List
concat' xs Nil          = xs
concat' xs (Cons y ys) = concat (add xs y) ys
```

Die Hilfsfunktion add fügt ein Element hinten an eine Liste an, also

```
  where
    add Nil          y = y
    add (Cons x xs) y = Cons x (add xs y)
```

Damit können wir nur — unserer obigen Idee folgend — definieren:

```
instance Monad List where
  return x          = Cons x Nil
  Nil          >>= f = Nil
  (Cons x xs) >>= f = concat' (f x) (xs >>= f)
```

Das nachrechnen der Monadenaxiome ist hier etwas fummelig, daher war es ja auch nicht Aufgabe ;-).

Aufgabe 3. Wir betrachen den do-Block

```
f = do
      x
      c ← y
      let foo x = x in
        foo c
```

und lösen ihn Schritt für Schritt auf: Zunächst die erste Zeile

```
f = x >>= \_
      → do
           c ← y
           let foo x = x in
             foo c
```

Die zweite Zeile wird zu einem let

```
f = x >>= \_
      → let ok c =
                do
                  (let foo x = x in
                    foo c)
              ok _ = fail "Aargh!"
        in y >>= ok
```

Das let können wir aus dem do einfach herausziehen:

```
f = x >>= \_
      → let ok c =
                (let foo x = x in
```

```
                do foo c)
          ok _ = fail "Aargh!"
      in y >>= ok
```

Bei nur einem Ausdruck geht es ohne do, wir haben

```
f = x >>= \_
    → let ok c =
            (let foo x = x in
                foo c)
          ok _ = fail "Aargh!"
      in y >>= ok
```

Jetzt können wir noch (war aber nicht gefordert) sehen, das das Pattern matchen muss, also ok gerade foo ist und foo die Identität, also

```
f = x >>= \_ → y >>= id
```

Aufgabe 4. Das Programm liefert alle Permutationen einer Liste. Um dies einzusehen formulieren wir es zunächst in Fischschwanznotation um:

```
p [] = [[]]
p as = do
         a  ← as
         as' ← p (delete a as)
         return (a:as')
```

Als Fischschwanz erhalten wir (beachte, dass keine Muster vorne stehen, wir also die ← direkt auswerten können)

```
p [] = [[]]
p as =
  as >>= \a →
  p (delete a as) >>= \as' →
    return (a:as')
```

In Aufgabe 2 hatten wir uns überlegt, dass der Fischschwanz gerade concatMap ist, wir können also as >>= \a → als „für jedes Element der Liste . . . “ lesen. Jetzt ist schon einsichtiger, dass oben alle Permutationen erzeugt werden: Wir wählen zunächst ein Element der Liste, dann eine Restpermutation und fügen unser Element vorne an. Das machen wir für alle Elemente und alle Restpermutationen, wir erhalten also alle Permutationen.