Funktionales Programmieren

Teil 4

Carl Philipp Reh

Universität Siegen

7. September 2021

Monad-Do

```
(>>=) :: m a -> (a -> m b) -> m b
```

Die do-Notation erlaubt uns, Ketten aus >>= untereinander zu schreiben. Zum Beispiel betrachten wir

```
divBy :: Int -> Int -> Maybe Int
toNat :: Int -> Maybe Nat
```

Mit >>= könnte man schreiben

```
i :: Maybe Nat
i = stringToInt "5" >>= (\x -> divBy 10 x)
    >>= toNat
```

Die äquivalente do-Notation sieht folgendermaßen aus:

```
i = do
    x <- stringToInt "5"
    y <- divBy 10 x
    toNat y
```

Foldable

Wir haben bereits gesehen, wie man `foldr` auf Listen definiert. Andere Datenstrukturen, wie zum Beispiel Bäume, können auch gefaltet werden, wofür Haskell eine Typklasse bereitstellt:

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
```

Das einfachste Beispiel ist wieder `Maybe`:

```
instance Foldable Maybe where
  foldr f s m = case m of
    Nothing -> s
    Just y -> f y s
```

`Maybe` kann man sich also als einen Container vorstellen, der 0 oder 1 Element enthält.

Foldable

Haskell stellt viele Funktionen auf `Foldable` bereit. Ein Beispiel hierfür ist

```
toList :: Foldable t => t a -> [a]
```

Es gilt zum Beispiel

```
toList Nothing = []
toList (Just 1) = [1]
toList [1,2] = [1,2]
```

Man kann hieran erkennen, dass eine Implementierung von `foldr` den Elementen eine „Reihenfolge“ gibt. Ein weiteres Beispiel ist

```
sum :: Foldable t -> t Int -> Int
```

Zum Beispiel gilt

```
sum Nothing = 0
sum (Just 1) = 1
```

Monoid

Bei `Foldable` kann man alternativ zu `foldr` auch

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

definieren. Dazu müssen wir uns zunächst die Typklasse `Monoid` anschauen. Diese ist eine Erweiterung von `Semigroup`:

```
class Semigroup m where
  (<>) :: m -> m -> m
class Semigroup m => Monoid m where
  mempty :: m
```

Hierbei ist `mempty` das neutrale Element und <> die Verknüpfung. Wie bei Monoiden in der Mathematik soll <> assoziativ sein.

Monoid

Ein `Monoid`, das `Int` multipliziert, könnte man zum Beispiel folgendermaßen implementieren:

```
data Times = Mul Int
```

Wir definieren hier zunächst einen neuen Typ `Times`, der dieselben Werte wie `Int` enthält. Dieser dient nur dazu, die Monoid-Operationen festzulegen. Die Implementierung ist dann

```
instance Semigroup Times where
  (<>) x y = case x of
               Mul x' -> case y of
                 Mul y' -> Mul (x' * y')
instance Monoid Times where
  mempty = Mul 1
```

Es gilt zum Beispiel `(Mul 2) <> (Mul 5) = Mul 10`.

Foldable

Zurück zu `foldMap`:

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

Die Idee bei `foldMap f l` ist, dass man jedem Element `x :: a` aus `l` über `f` ein Monoid-Element zuordnet. Man startet mit `mempty` und verknüpft alle Monoid-Elemente mit `<>`.
Für `Maybe` kann man `foldMap` folgendermaßen implementieren

```
instance Foldable Maybe where
  foldMap f m = case m of
    Nothing -> mempty
    Just y -> f y
```

Es gilt zum Beispiel

```
foldMap Mul Nothing = Mul 1
foldMap Mul (Just 2) = Mul 2
foldMap Mul [1,2,3] = Mul 6
```

Weitere Typklassen

Es gibt in Haskell sehr viel mehr nützliche Typklassen, die wir uns aber nicht alle anschauen können. Eine nicht vollständige Liste ist

- `Eq` zum Vergleichen.
- `Ord` zum Sortieren.
- `Num` für numerische Operationen wie `+` und `*`.
- `Fractional` für numerische Operationen wie `/`.
- `Traversable` zum „Herausziehen“ von `Applicative`.

Seiteneffekte

Alle bisher betrachteten Funktionen enthalten keine „Seiteneffekte“. Eine Funktion hat einen Seiteneffekt, wenn das Ausführen von ihr irgendeinen Einfluss auf den Rest des Programms hat. Man könnte sich zum Beispiel vorstellen, dass eine Funktion in C intern eine globale Variable schreibt, die dann woanders im Programm gelesen wird.

```
int x = 0;
int f()
{
    ++x;
    return 0;
}
int g()
{
    return x;
}
```

Monad-IO

Auch wenn das vorhin betrachtete Programm „schlechter Code“ ist, gibt es auch „legitime“ Seiteneffekte wie das Schreiben in eine Datei oder das Lesen aus der Standardeingabe.
In Haskell werden solche Seiteneffekte über die `IO`-Monade implementiert. Zum Beispiel gibt es zum Lesen einer Zeile aus der Standardeingabe

```
getLine :: IO String
```

Man kann sich `getLine` als einen Wert vorstellen, der intern beschreibt, wie man durch Ausführen von Seiteneffekten einen Wert vom Typ `String` erhält.
Ein weiteres Beispiel ist

```
putStrLn :: String -> IO ()
```

womit man einen String ausgibt. Das Ergebnis ist vom Typ `()`, was das leere Tupel ist. Das heißt, dass Ausdrucken „nichts“ liefert.

Monad-IO

Es ist nicht möglich, aus einem Wert vom Typ `IO String` einen String zu erhalten. Weil `IO` aber eine Monade ist, kann man `>>=` aufrufen, um an diesen String zu gelangen. Wegen

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

muss man aber einen weiteren Wert vom Typ `IO` produzieren. Das heißt, man kann die `IO`-Monade nie „verlassen“.
Als Beispiel betrachten wir

```
r :: IO ()
r = getLine >>=
    (\x -> putStrLn ("The␣input␣is␣" ++ x))
```

Haskell bietet die `main`-Funktion, welche den `IO`-Wert wirklich „ausführt“, wenn das Programm gestartet wird:

```
main :: IO ()
main = r
```