Von Monaden zu Applikative Funktoren:
Darf es auch ein bisschen weniger sein?

Funktionale Programmierung ist toll, solange Daten mit reinen Funktionen transformiert werden können. Manchmal benötigt man aber z.B. Ein– und Ausgaben oder andere Seiteneffekte. Solche Effekte können in Haskell als Monaden modelliert werden. Dadurch ist man jedoch gezwungen den Code stark umzustrukturieren und insbesondere durch die "do–Notation" bekommt er einen deutlich imperativeren Anstrich.
Es stellt sich jedoch heraus, dass Monaden häufig eine zu mächtige Abstraktion sind. D.h. man könnte dasselbe Programm auch in einer allgemeineren Art und Weise schreiben, die deutlich funktionaleren Code erlaubt. Dies ist genau die Lücke, die von *Applikativen Funktoren* [1] ausgefüllt wird.
Applikative Funktoren, sind Funktoren mit zusätzlichen Eigenschaften, wie auch Monaden, aber weniger ausdrucksstark als diese. Jede Monade ist auch ein applikativer Funktor, aber nicht umgekehrt.
Als motivierendes Beispiel betrachten wir folgendes, kleines Program:

```
import Control.Monad
import Control.Applicative

addSquare x y = square $ x + y
  where square x = x*x

exp = addSquare 2 5
```

Möchten wir nun z.B. eine nicht–deterministische Variante davon erstellen, so könnte es in der Listen–Monade wie folgt aussehen:

```
expM = do x <- [1,2]
          y <- [3,4]
          return $ addSquare x y
```

Im Gegensatz zu dem Originalprogramm, welches einfach eine Funktion auf Argumente anwendet, mussten wir diese nun explizit benennen. Tatsächlich haben wir die Listen–Monade jedoch nur benutzt, um verschiedene Werte zu extrahieren und unsere Funktion jeweils darauf anzuwenden. Dies kann auch als applikativer Funktor ausgedrückt werden und mit dieser Abstraktion sieht das Programm dann wie folgt aus:

```
expA = addSquare <$> [1,2] <*> [3,4]
```

In diesem Fall drückt der Code also wieder aus, dass `addSquare` einfach auf (nicht–deterministiche) Argumente angewandt wird[1].
Neben den rein syntaktischen Unterschieden bieten applikative Funktoren noch einige weitere Vorteile gegenüber Monaden:

- *Allgemeineres Interface*: Monaden haben zusätzliche Eigenschaften (`(>>=)`), die nicht von applikativen Funktoren unterstützt werden. Es gibt also einige Datentypen, die applikative Funktoren darstellen, jedoch keine Monaden sind, z.B. `ZipList`. Dadurch sind Programme, die lediglich das Interface für applikative Funktoren verwenden allgemeiner, als ihre monadische Variante.

[1] Ein erfahrener Haskellianer würde in diesem Fall natürlich **liftM2** verwenden. Damit hat er dann aber eigentlich nur den zugrundeliegenden applikativen Funktor benutzt und **liftA2** hätte es auch getan.

- *Komposition*: Zur Komposition von verschiedenen Monaden werden Monadentransformer benötigt. Dies liegt daran, dass die Komposition zweier Monaden im Allgemeinen keine Monade darstellt. Im Gegensatz dazu sind applikative Funktoren unter Komposition abgeschlossen und können generisch komponiert werden.

  In diesem Tutorial werden dazu einige Beispiele vorgestellt und insbesondere gezeigt, dass sich die Semantik häufig unterscheidet von den jeweiligen Monadentransformern. So wird z.B. in folgendem Beispiel

  ```
  bsp = pure (+) <*> pure 1 <*> nope <* tell "Huhu"
    where nope = ... a failing computation, i.e. Nothing
  ```

  der Effekt `tell "Huhu"` ausgeführt, wenn `Writer` und `Maybe` als applikative Funktoren komponiert werden, nicht jedoch für die monadische Komposition `MaybeT Writer`.

- *Statische Optimierungen*: Applikative Funktoren erlauben einige Optimierungen, die bei Monaden nicht möglich sind. Dies liegt daran, dass die Effekte von applikativen Funktoren, im Gegensatz zu Monaden, nicht von den Ergebnissen vorheriger Berechnungen abhängen können und somit statisch bekannt sind.

  Ein gutes Beispiel dafür ist applikatives Parsen [2]. Insbesondere zum Parsen von kontextfreien Grammatiken ist das applikative Interface völlig ausreichend und es können einige Optimierungen vorgenommen sowie aussagekräftigere Fehlermeldungen generiert werden.

- *Traversable*: Applikative Funktoren arbeiten gut mit dem `Traversable` Interface zusammen [1]. Auch hier wird nur die Möglichkeit der Hintereinanderausführung von Effekten benötigt, da diese jeweils beim verarbeiten eines Elements der traversierten Datenstruktur auftreten und daher nicht von den Ergebnissen vorheriger Berechnungen abhängen können.

  Damit lässt sich eine Liste von Listen z.B. wie folgt transponieren:

  ```
  getZipList $ traverse ZipList [[1,2,3], [4,5,6]]
  ```

Diese Tutorial gibt eine grundlegende Einführung[2] in das Konzept von applikativen Funktoren und behandelt (je nach verfügbarer Zeit und Interesse) einige der obigen Themen. Alle dabei gezeigneten Beispiele benutzen nur die Standardbibliotheken des GHC und benötigen keine weiteren Libraries.

Literatur

[1] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[2] S. Doaitse Swierstra. Combinator parsing: A short tutorial. Technical report, Institut of Information and Computing Sciences, Utrecht University, 2009.

[2] Allgemeine Kenntnisse von Haskell sowie Monaden werden vorrausgesetzt.