Dr. Th. Letschert
Softwarekomponenten

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

Aufgabenblatt 7

Aufgabe 1

1. Eine Berechnung $f(a : A)$ mit Wert $b \in B$ die von einem Kontext abhängt, der Werte $z \in Z$ liefert, kann **imperativ** als "Funktion"

   ```
   def f(a: A): B = {
     ... berechne b in Abhägigkeit von z ...
     return b
   }
   ```

   modelliert werden, die "heimlich irgendwo" die Z-Werte liest.

   Macht man sich in guter funktionaler Art ehrlich, dann wird aus f

   ```
   def f(a: A, z: Z): B = ...
   ```

   also eine Funktion $f : (A \times Z) \Rightarrow B$. In monadischer – flatMap-kompatibler – Form ist das

   ```
   def f(a: A): ZReader[B] = ...
   ```

   Nun kapselt aber ein *ZReader[B]* eine Funktion $Z \Rightarrow B$. Wie kommt man von

   $f : A \times Z \Rightarrow B$

   zu

   $f : A \Rightarrow (Z \Rightarrow B)$

   Demonstrieren Sie den Übergang von

   $A \times Z \Rightarrow B$ zu $A \Rightarrow (Z \Rightarrow B)$

   an einer **imperativen** "Funktion":

   ```
   def isLegalUser(name: String): Boolean = {
     val user = ... suche name in userDB ...
     user.isDefined
   }
   ```

   die, von allem Bösen befreit, in der funktionalen Welt wiedergeboren wird als:

   ```
   def isLegalUser(name: String): UserDBReader[Boolean] = ...
   ```

2. Die einem Reader gekapselte Funktion wird sehr oft (z.B. in Cats) "run" genannt. Warum?

Aufgabe 2

Angenommen wir haben eine *Reader*-Lösung für die Auswertung von Termen:

```scala
class Reader[Z, A](val run: Z => A) {
  def map[B](g: A => B) =
    Reader(run andThen g)
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] =
    Reader(z => g(run(z)).run(z) )
}
object Reader {
  def pure[Z, A](a: A) = Reader( (z:Z) => a )
}

type EnvReader[T] = Reader[Env, T]

def eval(term: Term): EnvReader[Int] = term match {
  case Literal(v) => Reader.pure(v)
  case Const(n) => Reader(env => env(n))
  case Add(t1, t2) =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l+r
  case Sub(t1, t2) => ...
  ...
}
```

1. In Scala können Kontextabhängigkeiten mit *Kontext-Funktionen*[1] ausgedrückt werden. Wie würde die Termauswertung bei der Verwendung von Kontext-Funktionen aussehen:

   ```scala
   type EnvReader[T] = Env ?=> T

   def eval(term: Term): EnvReader[Int] = ...
   ```

   Hinweis: Es könnte sich als hilfreich, oder zumindest die Schönheit verbessernd erweisen, auch diese Version des *Readers* "monadisch" zu machen:

   ```scala
   extension [A] (r: EnvReader[A]) {

     def map[B](g: A => B) =
       (env: Env) ?=> ...

     def flatMap[B](g: A => EnvReader[B]): EnvReader[B] =
       (env: Env) ?=> ...
   }
   ```

2. Die Extension kann natürlich auch noch etwas generischer gemacht werden:

   ```scala
   type Reader[Z, T] = Z ?=> T

   extension [Z, A] (r: Reader[Z, A]) {
     def map[B](g: A => B) = ...
     def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = ...
   }

   def eval(term: Term): Reader[Env, Int] = ...
   ```

3. In der Scala-Dokumention wird im Abschnitt, der sich mit der Implementierung von Typklassen beschäftigt, auch auf die Reader-Monade eingegangen[2] und im Weiteren erläutert, wie ein Reader mit *Typ-*

[1] https://dotty.epfl.ch/docs/reference/contextual/context-functions.html
[2] https://dotty.epfl.ch/docs/reference/contextual/type-classes.html

Abstraktionen, dort *Type Lambdas*[3] genannt, kombiniert werden können. Ist das, Ihrer Meinung nach, nur Schnick-Schnack oder eine sinnvolle Erweiterung der Ausdrucksmöglichkeiten.

Lässt sich diese Idee auf das Beispiel der Ausdrucksauswertung übertragen?

Aufgabe 3

1. In Aufgabenblatt 2 ging es um den Alpha-Beta-Algorithmus in funktionaler Form. In

```
def alphaBeta (
     config: C,
     player: Player,
     alpha: Int,
     beta: Int): Int = {
     ...
     // Funktionale Version des Alpha-Beta-Algorithmus'
 }
```

   hängt die Berechnung von *alphaBeta* von *alpha* und *beta* ab. Der Algorithmus lässt sich darum als *Reader* (von *(alpha, beta)*–Paaren) formulieren. Etwa wie folgt:

```
type AB = (Int, Int)
type ABReader[T] = Reader[AB, T]

def alphaBeta (config: C, player: Player): ABReader[Int] = ...
```

   Hinweis: Der Kern des Algorithmus' ist der Abbruch der Berechnung für die Nachfolger eines Knotens, sobald $a >= beta$ bzw. $b <= alpha$. Übernehmen dies **nicht** in Ihre mit dem *Reader* geschönte Version.

2. Warum kann der Abbruch der Berechnung nicht in die Reader-Version integriert werden – jedenfalls nicht ohne "Gemurkse", das alles an monadischer Verschönerung wieder zerstört.?

[3] https://dotty.epfl.ch/docs/reference/new-types/type-lambdas.html