Dr. Th. Letschert
Softwarekomponenten

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

Aufgabenblatt 1

Aufgabe 1

1. Einige Anwendungen verwenden verschiedene historisch gewachsene Datenquellen in Form von Datei-Kollektionen. Zu den Daten gehören Metadaten, wie beispielsweise ein Datum, Erzeuger oder Dokumentenart. Leider gibt es weder einheitliche Benennungen noch ein einheitliches Format für die Metadaten. Die einzige Gemeinsamkeit ist, dass sie in die Namen der Dateien nach irgendeiner Konvention codiert wurden.

   Anwendungen mit Zugriff auf unterschiedliche Datenquellen benötigen eine einheitliche Zugriffsschnittstelle. Beispielsweise könnte eine Funktion getFileType das eventuell vorhandene Metadatum *FileType* zu einer Datei liefern, egal in welcher Form und nach welcher Konvention es in den Namen der Datei codiert wurde:

   ```
   enum FileType {
     case CSV
     case XLS
     case TXT
   }

   case class Person(name: String, orgnization: String)

   ...

   val ft: Option[FileType] = getFileType("SomeFileName")
   val fc: Option[Person] = getCreator("SomeFileName")
   ```

   Wie können Typklassen hier helfen?

2. Uups, jetzt gibt es auch noch Datenquellen bei denen die Metainformationen in einer speziellen Datei zu finden sind. Kann auch das (einfach) in den vorhandenen Mechanismus integriert werden?

Aufgabe 2

1. Ist der Typ der Listen von Werten eines beliebigen Typs A mit der leeren Liste als neutralem Element und der Listenverkettung ein Monoid unabhängig davon, ob A ein Monoid ist?

2. Wenn $(M, z, \circ)$ ein Monoid ist, dann auch $(\mathit{Option}[M], z_{O[M]}, \circ_{O[M]})$ mit einem geeignetem neutralen Element $z_{O[M]}$ und einer geeigneten Operation $\circ_{O[M]}$ ein Monoid.

   Definieren Sie eine Typklassen-Instanz für $(\mathit{Option}[M])$ bei gegebenem Monoid M und testen Sie Ihre Definition etwa wie folgt:

   ```
   trait Monoid[A] {
     val z: A
     extension (x: A) {
       def op(y: A): A
   ```

```
  }
}
object Monoid {
  def apply[A: Monoid] = summon[Monoid[A]]
}

given StringMonoid ...

given IntMonoid ...

given optinMonoid[A: Monoid] ...


def sum[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].z)( (acc, item) => acc.op(item))

val loi: List[Option[Int]] = List(Some(1), Some(2), None, Some(3))
val los: List[Option[String]] = List(Some("1"), Some("2"), None, Some("3"))

val sloi = sum(loi) // Some(6)
val slos = sum(los) // Some("123")
```

Aufgabe 3[1]

Das Rechnen im internationalen **SI**-Einheitensystem[2] kann durch den Compiler während der Typprüfung auf fehlerhafte Kombinationen von Messgrößen unterschiedlicher Einheiten geprüft werden. Ein erster Ansatz könnte sein:

```
type Dim = Singleton & Int

// Alle Einheiten sind als Produkte der Basis-Einheiten darstellbar.
// Der Einfachheit halber beschränken wir uns auf drei Basiseinheiten:
// Zeit, Länge und Gewicht
case class SIQ[TimeD <: Dim, LengthD <: Dim, WeightD <: Dim](value: Double)

object UnitOps {
  extension (x: Double) def s : SIQ[1, 0, 0] = SIQ[1, 0, 0](x)
  extension (x: Double) def m : SIQ[0, 1, 0] = SIQ[0, 1, 0](x)
  extension (x: Double) def kg : SIQ[0, 0, 1] = SIQ[0, 0, 1](x)
}
import UnitOps._
import scala.language.postfixOps

// wenn die Dimensionen übereinstimmen, dann kann addiert werden.
// Der Compiler (!) prüft dies.
def add[D1 <: Dim,
        D2 <: Dim,
        D3 <: Dim](q1: SIQ[D1, D2, D3], q2: SIQ[D1, D2, D3]): SIQ[D1, D2, D3] =
  SIQ[D1, D2, D3](q1.value + q2.value)

val v1 = 1 m
val v2 = 1 m
val v3 = 2 kg
val v12 = add(v1, v2)
val v13 = add(v1, v3) // Typfehler !!
```

[1] Miniprojekt (für Scala-Enthusiasten)
[2] https://de.wikipedia.org/wiki/Internationales_Einheitensystem

Eine Verbesserung der Benutzbarkeit, etwa durch durch geeignete Konversionen, ist da sicher noch angebracht.