Klausur : Einführung in die funktionale Programmierung

Musterklausur

Prof. Dr. Frank Puppe
Felix Herrmann
Alexander Gehrke

Musterlösung

Name :	
Matrikelnummer :	
Studiengang :	

Punkteübersicht :

Frage	1	2	3	4	5	6	7	8	9	10	11	Summe
Punkte	5	6	14	4	4	5	4	4	4	4	4	58
Erreicht												

1. Kurzfragen

(a) Sie möchten eine Funktion schreiben, die entweder eine Zahl (Int) oder einen Fehlercode zurückgeben soll. Welchen Rückgabetyp würden Sie verwenden? (1)

> **Solution:** `Either[Error, Int]` (Für `Error` kann ein beliebiger Fehlertyp eingesetzt werden. Für schlechte Fehlertypen wie z.B. Int ggf. halber Punkt Abzug)

(b) Geben Sie die Signatur der für einen Funktor definierten Methode an (wahlweise als Extension oder in normaler Funktionsschreibweise). (1)

```
trait Functor[F[_]]:
```

> **Solution:**
>
> ```
> trait Functor[F[_]]:
>         def map[A,B](fa: F[A])(f: A => B): F[B]
>
> // -- or as extension --
>
> trait Functor[F[_]]:
>         extension [A](fa: F[A])
>                 def map[B](f: A => B): F[B]
> ```

(c) Definieren Sie den Begriff Algebra bezogen auf Typen, wie er in der Vorlesung eingeführt wurde. (3)

> **Solution:** Eine Algebra besteht aus:
>
> - Einer Menge von Typen bzw. Eine Menge von Mengen von Werten
> - Einer Menge von Operationen auf diesen Typen
> - Einer Menge von Axiomen / Gesetzen

2. Rekursionen

Gegeben sei folgende Datenstruktur eines Binärbaumes. Ein Binärbaum ist hier entweder ein Blatt mit einem Wert, oder ein Zweig, der einen linken und rechten Unterbaum besitzt.

```
enum Tree[+A]:
  case Leaf(value: A)
  case Branch(left: Tree[A], right: Tree[A])

  final def doSomething(p: A => Boolean): Option[A] =
    this match
      case Leaf(a) => if p(a) then Some(a) else None
      case Branch(l,r) => l.doSomething(p) match
        case Some(a) => Some(a)
        case None => r.doSomething(p)
```

Betrachten Sie nun die Methode `doSomething` im Enum `Tree`.

(a) Erklären Sie in eigenen Worten, was die Methode tut. (4)

Solution: Die Methode durchsucht rekursiv den Baum nach einem Element, welches die Bedingung `p` erfüllt (in Form einer Tiefensuche).
Hierbei wird jeweils der linke Teilbaum zuerst ausgewertet.

(b) In `doSomething` sind zwei rekursive Aufrufe enthalten. Entscheiden Sie für jeden, ob dieser in Tail-Position steht (d.h. die Bedingungen für eine tailrekursive Funktion erfüllt) und ob die gesamte Methode tail-rekursiv ist. (2)

Solution:

- erster Aufruf (Z.4, auf `l`): Nicht in Tail-Position, da Ergebnis für Pattern Match benutzt wird
- zweiter Aufruf (Z.6, auf `r`): in Tail-Position
- mindestens ein Aufruf nicht in Tail-Position => nicht tail-rekursiv

3. Monadengesetze

Gegeben sei folgende Datenstruktur eines Binärbaumes und eine dazu passende Monad-Instanz:

```
enum Tree[+A]:
  case Leaf(value: A)
  case Branch(left: Tree[A], right: Tree[A])

object Tree:
  given Monad[Tree] with
    extension [A](fa: Tree[A])
      def flatMap[B](f: A => Tree[B]): Tree[B] =
        fa match
          case Leaf(a) => f(a)
          case Branch(l, r) => Branch(flatMap(l)(f), flatMap(r)(f))

    def pure[A](a: A): Tree[A] = Leaf(a)
```

(a) Zeigen Sie, dass die gegebene Monadinstanz das Assoziativitätsgesetz erfüllt: (8)

```
m.flatMap(f).flatMap(g) == m.flatMap(a => f(a).flatMap(g))
```

Solution:

```
// Assoziativität für Leaf
Leaf(v).flatMap(f).flatMap(g) == Leaf(v).flatMap(a => f(a).flatMap(g))
f(v).flatMap(g) == (a => f(a).flatMap(g))(v)
f(v).flatMap(g) == f(v).flatMap(g)

// Assoziativität für Branch

Branch(l, r).flatMap(f).flatMap(g) == Branch(l, r).flatMap(a => f(a).flatMap(g))

   Branch(l.flatMap(f).flatMap(g), r.flatMap(f).flatMap(g))
== Branch(l.flatMap(a => f(a).flatMap(g)),r.flatMap(a => f(a).flatMap(g)))


l.flatMap(f).flatMap(g) == l.flatMap(a => f(a).flatMap(g))
// = Assoziativitätsgesetz
somit assoziativ für Branch(l, r), wenn assoziativ für l und r
l, r können nur Branch oder Leaf sein => assoziativ
```

(wie vorher)

```
enum Tree[+A]:
  case Leaf(value: A)
  case Branch(left: Tree[A], right: Tree[A])

object Tree:
  given Monad[Tree] with
    extension [A](fa: Tree[A])
      def flatMap[B](f: A => Tree[B]): Tree[B] =
        fa match
          case Leaf(a) => f(a)
          case Branch(l, r) => Branch(flatMap(l)(f), flatMap(r)(f))

    def pure[A](a: A): Tree[A] = Leaf(a)
```

(b) Zeigen Sie, dass die gegebene Monadinstanz die Identitätsgesetze erfüllt. (6)

`x.flatMap(pure) == x`

`pure(y).flatMap(f) == f(y)`

Solution:

```
// pure rechts
x.flatMap(pure) == x

Leaf(v).flatMap(pure) == Leaf(v)
pure(v) == Leaf(v) // q.e.d.

Branch(l, r).flatMap(pure) == Branch(l, r)
Branch(l.flatMap(pure), r.flatMap(pure)) == Branch(l, r)
x.flatMap(pure) == x

// pure links
pure(y).flatMap(f) == f(y)
Leaf(y).flatMap(f) == f(y)
f(y) == f(y)
```

4. Int Monoids (4)

Ein Monoid wurde im Code wie folgt dargestellt:

Implementieren Sie zwei verschiedene Monoide für Int, die die Monoid-Laws erfüllen:

```
given intMonoid1: Monoid[Int] with
```

```
given intMonoid2: Monoid[Int] with
```

.

Solution:

```
given intMonoid1: Monoid[Int] with
        def zero = 1
        def combine(a: Int, b: Int) = a * b
```

```
given intMonoid2: Monoid[Int] with
        def zero = 0
        def combine(a: Int, b: Int) = a + b
```

5. Parametrizität

Gegeben sei folgende Funktionssignatur:

```
def p2[A,B,C,D](a: A, b: B)(f: (A,B) => C, g: (A,C) => D): D
```

(a) Geben Sie nur anhand der Typen eine gültige Implementation der Funktion an (die Signatur muss nicht nochmal abgeschrieben werden). (2)

Solution:
```
= g(a, f(a,b))
```

(b) Warum reicht hier die Signatur aus, um Aussagen über das Verhalten zu treffen, solange sich die Implementation referentiell transparent verhält (also keine Exception wirft o.ä.)? (2)

Solution: Einige mögliche Lösungen:

- Da die Typen variabel sind, können keine Operationen außer den übergebenen ausgeführt werden
- Polymorphe Funktionen schränken durch Eingrenzen der möglichen Operationen die möglichen Implementationen ein.
- Es gibt keine Möglichkeit, die variablen Typen C und D zu instanziieren. Da ein D zurückgegeben werden muss, muss dieses über die gegebenen Funktionen erzeugt werden.

6. Rekursion : takeWhileTCO (5)

Gegeben ist die folgende nicht tail-rekursive Funktion:

```
def takeWhile[A](l: List[A], pred: A => Boolean): List[A] = l match {
  case Nil   => Nil
  case x::xs => if (pred(x)) x :: takeWhile(xs, pred)
                else Nil
}
```

Sie bekommt eine Liste und ein Prädikat und gibt den längsten Anfang der Liste zurück, der nur Elemente enthält, die das Prädikat erfüllen.

Geben Sie eine Version dieser Funktion an, die tail-rekursiv ist.

Sie dürfen dabei die Methode `.reverse` auf Listen benutzen.

Solution:

```
def takeWhile[A](l: List[A], pred: A => Boolean) = {
  def go(accu: List[A], list: List[A]): List[A] = list match {
    case Nil       => accu.reverse
    case a :: tail => if (pred(a)) go(a :: accu, tail)
                      else accu.reverse
  }

  go(Nil, l)
}
```

7. Recursion zipAdd (4)

Implementieren Sie die Funktion `zipAdd(l1: List[Int], l2: List[Int]): List[Int]`, welche zwei Listen von Ints bekommt und eine Liste zurück liefert, in der die Elemente paarweise addiert wurden. Beim Input `List(1,2,3)` und `List(4,5,6)` wird `List(5, 7, 9)` zurückgeben. Sollte eine Liste kürzer als die andere sein, soll so getan werden, als hätten beide Listen die Länge der kürzeren.

Sie muss nicht tailrekursiv sein.

Solution:

```
def zipAdd(l1: List[Int], l2: List[Int]): List[Int] = (l1, l2) match {
  case (Nil, _  ) => Nil
  case (_  , Nil) => Nil
  case (l1h :: l1t, l2h :: l2t) => (l1h + l2h) :: zipAdd(l1t, l2t)
}
```

8. Modellierung einer Geldbörse (4)

Im folgenden Programm gibt es einige Probleme. Nennen Sie 4 Stellen, die im Gegensatz zu in dieser Vorlesung gelernten Prinzipien stehen und nennen Sie jeweils eine mögliche Lösung:

```
final case class Wallet(
        amountMoney:       Int,
        numberOfDocuments: Int,
)
// --- später ---
def transformWallet(w: Wallet): Wallet {
        if (w == null) {
                changeWallet(w)
        } else {
                throw new RuntimeException("no wallet");
        }
}
```

Solution:

- Unterschiedliche Konzepte mit selbem Typ. Führe neue Typen für Geld/Anzahl Dokumente ein.
- Benutze nie null, sondern stattdessen Option.
- Vermeide boolean blindness, z.B. durch Pattern Matching.
- Wird keine Exceptions, nutze stattdessen Either/Option.

9. State Monad

Gegeben sei folgende Case Class für einen State-Monad:

```
case class State[S, +A](run: S => (A, S)):
  def map[B](f: A => B): State[S, B] = ???

  def flatMap[B](f: A => State[S, B]): State[S, B] = ???
```

Die definierten Methoden seien so implementiert, dass sie die Gesetze für Monaden und Applicatives erfüllen. Außerdem seien die folgenden Methoden wie in der Vorlesung implementiert:

```
def get[S]: State[S, S]

def set[S](s: S): State[S, Unit]
```

(a) Geben Sie den Typ von `intState` in folgendem Ausdruck an: (1)

```
val intState = for {
  i <- get[Int]
  q = i * i
  _ <- set(q)
} yield q.toString
```

Solution: `State[Int, String]`

(b) Übersetzen Sie die for-Comprehension im obigen Ausdruck in Aufrufe der auf `State` definierten Methoden `flatMap` und `map`. (3)

Solution:

```
get[Int].flatMap(i => {
  val q = i * i;
  set(q).map(_ => q.toString)
})
```

10. Folds

(a) Erläutern Sie den Unterschied zwischen `foldLeft` und `foldRight` auf Listen. (2)

Solution: Mögliche Antworten:

- `foldLeft` arbeitet mit links-assoziativem Operator, `foldRight` mit rechts-assoziativem.
- foldLeft kombiniert Elemente in umgekehrter Reihenfolge (im Vergleich zu foldRight) mit dem Akkumulator.
- In Baumdarstellung der Operationen erhält man einen links- bzw. rechtslehnenden Baum.

(b) Nennen Sie einen Fall, in dem `foldLeft` und `foldRight` auf der selben Liste das gleiche Ergebnis bei gleicher Eingabeliste liefern. (2)

Solution: Selbes Ergebnis wenn (eine Variante ausreichend):

- benutzter Operator ist assoziativ.
- Liste ist symmetrisch (`x.foldLeft(z)(f) == x.reverse.foldRight(z)((a,b) => f(b,a))` gilt immer)

11. Referential Transparency (4)

Gegeben ist die das folgende Programm, welches die `scala.collection.mutable.Stack` Klasse benutzt. Die Methode `pop` entfernt ein Element vom Stack und gibt dieses Element zurück. Zeigen Sie, dass das Programm nicht referentiell transparent ist.

```
def sum(s: Stack[Int]): Int {
        val a = s.pop()
        val b = s.pop()
        a + b
}
```

Solution: Wenn wir die Funktion mit einem Stack aufrufen, auf dem 3 und 4 oben liegt, dann gibt sie 7 zurück. Im folgenden Programm ist jedoch der Rückgabewert 6, obwohl nur eine Expression in eine Variable ausgelagert wurde.

```
def sum(s: Stack[Int]): Int {
        val ss = s.pop()
        val a = ss
        val b = ss
        a + b
}
```