Exam: Introduction to Functional Programming

Sample exam

Prof. Dr. Frank Puppe
Felix Herrmann
Alexander Gehrke

Solutions

Name:	
Matriculation number:	
Course of studies:	

Point overview:

Question:	1	2	3	4	5	6	7	8	9	10	11	Total
Points:	5	6	14	4	4	5	4	4	4	4	4	58
Score:												

1. Short questions

(a) You want to write a function that should return either a number (Int) or an error code. Which return type would you use? (1)

> **Solution:** `Either[Error, Int]` (An arbitrary error type can be used for `Error`. For particularly unsuited error types, e.g. Int, a half point may be deducted)

(b) Give the signature for the method which is defined for a Functor (optionally as extension or in normal function notation). (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) Define the concept of algebras in regard to types, as introduced in the lecture. (3)

> **Solution:** An algebra consists of
>
> - a set of types / a set of sets of values
> - a set of operations on these types
> - a set of axioms / laws

2. Recursions

Given is the following data structure of a binary tree. A binary tree here is either a leaf with a value or a branch, which has a left and right subtree.

```
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)
```

Now look at the method `doSomething` in the `Tree` enum.

(a) Explain in your own words, what the method does. (4)

Solution: The method recursively searches the tree for an element, which fulfills the condition `p` (with a depth-first search).
It always looks at the left subtree first.

(b) There are two recursive calls in `doSomething`. Decide for each of them, if it is in tail position (i.e. the conditions for a tailrecursive function are fulfilled), and if the the whole function is tail recursive. (2)

Solution:

- first call (L.4, on `l`): Not in tail position, as the result is used for the pattern match.
- second call (L.6, on `r`): in tail position
- at least one call not in tail position => not tail recursive

3. Monad laws

Given is the following data structure of a binary tree and a matching monad instance:

```
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) Prove that the given monad instance fulfills the associativity law: (8)

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

Solution:

```
// Associativity for 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)

// Associativity for 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))
// = Associativity law
therefore associative for Branch(l, r), if associative for l and r
l, r can only be Branch or Leaf => associative
```

(as before)

```
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) Prove that the given monad instance fulfills the identity laws. (6)

`x.flatMap(pure) == x`

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

Solution:

```
// pure right
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 left
pure(y).flatMap(f) == f(y)
Leaf(y).flatMap(f) == f(y)
f(y) == f(y)
```

4. Int Monoids (4)

We represented a monoid in code as follows:

Implement two different monoids for Int, which fulfill the monoid laws.

```
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. Parametricity

We are given the following function signature:

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

(a) Give a valid implementation of the function only based on the types (you don't have to write the signature again). (2)

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

(b) Why is the signature sufficient here to make assertions about the function's behaviour, as long as the implementation behaves referentially transparent (i.e. doesn't throw exceptions etc.)? (2)

Solution: Some possible solutions:

- As the types are variable, no operations except the given ones can be executed.
- Polymorphic functions limit the possible implementations by limiting the possible operations available.
- There is no possibility to instanciate the variable types C and D. As a D has to be returned, it has to be created via the passed functions.

6. Recursion: takeWhileTCO (5)

Given is the following, non tail recursive function:

```
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
}
```

It takes a list and a predicate and returns the list's longest prefix which only contains elements fulfilling the predicate.

Give a version of this function, which is tail recursive.

You may use the method `.reverse` on lists.

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. : zipAdd (4)

Implement the function `zipAdd(l1: List[Int], l2: List[Int]): List[Int]`, which takes two lists of ints and returns a list, in which the elemens have been added pairwise. For the inputs `List(1,2,3)` and `List(4,5,6)` the result is `List(5, 7, 9)`. If one list is shorter than the other, the longer list should be treated as if it had only the length of the shorter list.

The function isn't required to be tail recursive.

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. Modelling a wallet (4)

The following program has several problems. Point out 4 things, that are in contradiction to the principles presented in the lecture and give possible solutions for each.

```
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:

- Different concepts use same types. Introduce new types for money/number of documents.
- Never use null, use option instead.
- Forgo boolean blindness, use pattern matching.
- don't throw exceptions, use Either/Option instead.

9. State Monad

Given is the following case class for a 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] = ???
```

Assume that the defined methods are implemented in a way, that fulfills the laws of Monad and Applicative. Also the following methods are implemented as in the lecture

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

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

(a) Give the type of `intState` in the following expression: (1)

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

Solution: `State[Int, String]`

(b) Translate the for comprehension in the above expression into calls of the methods `flatMap` and `map` defined on `State`. (3)

Solution:

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

10. Folds

(a) Explain the difference between `foldLeft` and `foldRight` on lists. (2)

Solution: Possible answers:

- `foldLeft` uses a left associative operator, `foldRight` a right associative one.
- foldLeft combines elements with the accumulator in reverse order (compared with foldRight)
- When representing the operations as a tree, you get either a left leaning or a right leaning tree respectively.

(b) Name one possible condition, under which `foldLeft` and `foldRight` return the same result for the same input list. (2)

Solution: Same result, if (one item is sufficient):

- the used operator is associative.
- the list is symmetrical (`x.foldLeft(z)(f) == x.reverse.foldRight(z)((a,b) => f(b,a))` is always true)

11. Referential Transparency (4)

Given is the following program, which uses the `scala.collection.mutable.Stack` class. The method `pop` removes an element from the Stack and returns that element. Show that this program is not referentially transparent.

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

Solution: If we call the function with a stack, that has 3 and 4 as its top elements, the method returns 7. But the following program returns 6, even though we only extracted an expression into a variable.

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