Dr. Th. Letschert
Softwarekomponenten

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

Aufgabenblatt 9

Aufgabe 1

1. Eine Berechnung

   $f : A => B$

   wird im Stil des *Continuation / Callback Passings* zu

   $f^C : (A \times (B => Unit)) => Unit$

   Mit der Continuation-Monade $C[\cdot]$ ist das dann

   $f^C : A => C[B]$

   wobei

   $C[B] \sim (B => Unit) => Unit$

   Wie kommt man von

   $(A \times (B => Unit)) => Unit$

   zu

   $A => (B => Unit) => Unit$?

2. Im CPS mit einer einer Continuation-Monade kann die Fakultät wie folgt berechnet werden:

```
trait Functor[F[_]] {
  extension[A] (x: F[A]) {
    def map[B](f: A => B): F[B]
  }
}

trait Monad[F[_]] extends Functor[F] {
  def pure[A](x: A): F[A]
  extension[A] (x: F[A]) {
    def flatMap[B](f: A => F[B]): F[B]
  }
}
object Monad {
  def apply[F[_]:Monad] = summon[Monad[F]]
}

type Cont[R, A] = (A => R) => R
type ContToUnit[A] = Cont[Unit, A]

given contMonad: Monad[ContToUnit] with {
  def pure[A](a: A): ContToUnit[A] = ???

  extension [A](x: ContToUnit[A]) {
    def flatMap[B](f: A => ContToUnit[B]): ContToUnit[B] = ???
```

```
    def map[B](f: A => B): ContToUnit[B] = ???
  }
}

def computeFac[F[_]: Monad](x: Int): F[Int] =
  if (x == 0) Monad[F].pure(1)
  else for( y <- computeFac(x-1) ) yield x * y

val fak5: ContToUnit[Int] = computeFac(5)

fak5(res => println(res))
```

Ergänzen Sie die fehlenden Codestücke.

3. Eine Berechnung der Fibonacci-Zahlen ist schnell definiert:

```
def fib(n: Int): Int =
 if (n <= 1) n
 else fib(n-1) + fib(n-2)
```

Formulieren Sie diese Funktion um in den *CPS*:

```
def fibCPS(n: Int, k: Int => Unit): Unit = ???
```

Definieren Sie jetzt eine monadische Berechnung der Fibonacci-Zahlen

```
def fibM[F[_] : Monad](n: Int): F[Int] = ...
```

und führen sie in Gegenwart einer Continuation-Monade aus:

```
given contMonad : Monad[ContToUnit] with { ... }

val fib10: ContToUnit[Int] = fibM(10)
fib10(res => println(s"fib(10) = $res"))
```

Aufgabe 2

Linear-rekursive Funktionen sind nach einer Transformation in den *CPS* end-rekursiv. Das lässt sich an einigen wichtigen linear-rekursiven Listenfunktionen demonstrieren:

```
@tailrec
def appendCPS[A, B >: A](lst1: List[A], lst2: List[B],
                         k: List[B] => Unit): Unit = lst1 match {
  case Nil => k(lst2)
  case head :: tail => appendCPS(tail, lst2, l => k(head::l))
}

@tailrec
def reverseCPS[A](lst: List[A],
              k: List[A] => Unit): Unit = lst match {
  case Nil => k(Nil)
  case head :: tail =>
    reverseCPS(tail, l => appendCPS(l, List(head), k))
}

def pure[A](a: A, k: A => Unit): Unit = k(a)
```

```
val lst_1: List[Int] = List(1, 2, 3)
val lst_2: List[Int] = List(4, 5, 6)

@main
def printLst: Unit =
 pure(lst_1, x =>
   appendCPS(x, lst_2, y =>
    reverseCPS(y, z =>
     println(z) // List(6, 5, 4, 3, 2, 1)
    )
   )
 )
```

So richtig übersichtlich und schön ist das natürlich nicht.

Vielleicht verbessert sich die Übersichtlichkeit und Schönheit, wenn sie monadisch gemacht wird:

```
def append[F[_]: Monad, A, B >: A](lst1: List[A], lst2: List[B]): F[List[B]] = ...

def reverse[F[_]: Monad, A](lst: List[A]): F[List[A]] = ...

val lst_1: List[Int] = List(1, 2, 3)
val lst_2: List[Int] = List(4, 5, 6)

type Cont[R, A] = (A => R) => R
type ContToUnit[A] = Cont[Unit, A]

given contMonad : Monad[ContToUnit] with ...

val l1plusl2Reversed = ...

l1plusl2Reversed(l => println(l)) // List(6, 5, 4, 3, 2, 1)
```