Prof. Dr. Th. Letschert
Funktionale Programmierung

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

Aufgabenblatt 4

Aufgabe 1

Endrekursive Funktionen können leicht in Schleifen umgewandelt werden. Was sind endrekursive Funktionen und wie werden sie in Schleifen umgewandelt? Erläutern Sie am Beispiel der ggt-Funktion, die sie händisch in eine Schleife umwandeln:

```
import scala.math.{max, min}
import scala.annotation.tailrec

@tailrec
def gcd(x: Int, y: Int): Int =
   if (x == y) x else gcd (max(x,y) - min(x, y), min(x, y))
```

Aufgabe 2

Linear rekursive Funktionen haben eine "hängende" Operation nach dem rekursiven Aufruf. Sehr viele Funktionen sind linear rekursiv. Die Fakultätsfunktion ist linear rekursiv, ebenso viele Funktionen auf Listen. Linear rekursive Funktionen haben die Form

$$f(x) \;=\; \textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } \psi(f(h(x)), x)$$

Dabei ist ψ die "hängende Operation". Linear rekursive Funktionen können oft mit Hilfe eines zusätzlichen "akkumulierenden Parameters" in endrekusive Form gebracht werden.

1. Bringen Sie die Fakultätsfunktion mit einem akkumulierenden Parameter in endrekursive Form.
2. Die Summe einer Sequenz l sei definiert als:

   $$sum(l) \;=\; \textbf{if } l =<> \textbf{ then } 0 \textbf{ else } head(l) + sum(tail(l))$$

   bringen Sie diese Funktion mit einem akkumulierenden Parameter in endrekursive Form.
3. Eine linear rekursive Funktion $append$ die zwei Sequenzen zusammenfügt sei definiert als:

   $append(Nil, l_2) = l_2$
   $append(h :: t, l_2) = h :: append(t, l_2)$

   Diese Funktion kann (ohne die Hilfe weiterer Operationen wie Umkehren einer Liste, Anhängen eines Elements an eine Liste, etc.) nicht (!) mit einem akkumulierenden Parameter in endrekursive Form gebracht werden. – Warum nicht?

   Selbstverständlich kann $append$ mit einer (einzigen) Schleife irgendwie realisiert werden. – Wirklich? Auch dann wenn keine weiteren Operationen genutzt werden dürfen?

Aufgabe 3

Wandeln Sie folgende Funktion zum Aufsummieren der Elemente einer Liste in den *Continuation Passing Style* (CPS) um:

```
def sum(lst: List[Int]) : Int = lst match {
   case Nil       => 0
   case head :: tail => head + sum(tail)
}
```

Gestalten Sie Ihre Lösung `sumC1` so, dass durch

```
def sumC1(lst: List[Int], k: Int => Unit) : Unit = lst match {
   ????
}

sumC1(List(1,2,3,4,5,6,7,8,9,10), (v:Int)=>println(v) )
```

55 ausgegeben wird.

Modifizieren Sie Ihre Lösung zu `sumC2` derart, dass durch

```
def sumC2(lst: List[Int], k: Int => Int) : Int = lst match {
    ????
}

println( sumC2(List(1,2,3,4,5,6,7,8,9,10), (v:Int) => v) )
```

ebenfalls *55* ausgegeben wird.

Aufgabe 4

Betrachten Sie die Funktion

```
def sum(x: Long, y: Long) : Long =
   if (x==0) y
   else sum(x-1, y) + 1
```

Testen Sie bei welchen (positiven !) Argumenten für `x` ein *java.lang.StackOverflowError* auftreten wird.

Wie steht es wohl mit der CPS-Version:

```
def sumC(x: Long, y: Long, k: Long => Unit) : Unit =
   if (x==0) k(y)
   else sumC(x-1, y, (v:Long) => k(v + 1))
```

... Hmm, das ist doch endrekursiv, der Compiler muss daraus eine Schleife machen. Macht er?

Probieren wir es mal mit der Annotation :

```
@tailrec
def sumC(x: Long, y: Long, k: Long => Unit) : Unit =
   if (x==0) k(y)
   else sumC(x-1, y, (v:Long) => k(v + 1))
```

... Hilft nicht. Vielleicht funktioniert die ganze *Tail Call Elimination* ja auch nicht. Versuchen wir es mit einer Umwandlung per Hand: 5

```
def sumCI(x: Long, y: Long, k: Long => Unit) : Unit = {
   var k_ = k
   var x_ = x
   while (x_ != 0) {
     val kkk = k_ // fix k_
     val (xt, kt) = (x_ - 1, { (v:Long) => kkk(v + 1) })
     x_ = xt
     k_ = kt
   }
```

```
    k_ (y)
}
```

Naja, hilft auch nicht! Für kleine Werte funktioniert das schon, aber bei großen Werten für x bekommen wir wieder unseren *Stackoverflow*.

Noch ein Versuch: Statt die Fortsetzungsfunktionen händisch ineinander zu schachteln, sammeln wir sie auf und dann lassen sie uns dann von Scala verknüpfen:

```
def sumCIL(x: Long, y: Long, k: Long => Unit) : Unit = {

   var k_l : List[Long => Long] = List()
   var x_ = x
   while (x_ != 0) {
     val (x_t, k_t) = (x_ - 1, {(v:Long) => (v + 1)} :: k_l )
     x_ = x_t
     k_l = k_t
   }

   // compose all functions
   val f = k_l . foldLeft(k)( (acc, op) => op andThen acc)

   f(y)
}
```

Hilft das?

Wenn nicht: Dann gibt es noch einen letzten Versuch: Wir probieren wir es mal damit, die Funktionen in einer Liste zu sammeln und – satt sie zu einer Funktion zu verknüpfen – Schritt für Schritt abzuarbeiten.

```
def sumCILL(x: Long, y: Long, k: Long => Unit): Unit = {
   var k_L = List[Long => Long]()

   // ... k_L aufbauen ....

   // ... Funktionen in k_L ausfuehren ...
   // und result berechnen

   k(result)
}
```

Wie weit kommen wir damit?

Aufgabe 5

Eine "doppelt verzweigende" rekursive Funktion hat die Form

$$f(x) \;=\; \textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } \psi(f(h_1(x)), f(h_2(x)))$$

Eine solche Funktion kann in den *Continuation Passing Style* gebracht werden:

$$f(x,k) \;=\; \textbf{if } p(x) \textbf{ then } k(g(x)) \textbf{ else } f(h_1(x), v_1 \rightarrow f(h_2(x), v_2 \rightarrow k(\psi(v1, v2))))$$

Beispielsweise können wir eine Auswertungsfunktion für arithmetische Ausdrücke derart umformen.

Aus:

```
object Direct_Sem {

  type Result = Int

  def eval(e: Exp): Result = e match {
    case Const(v) => v
    case Add(e1, e2) => eval(e1) + eval(e2)
    case Sub(e1, e2) => eval(e1) - eval(e2)
  }

}
```

wird dann

```
object CPS_Sem {

  type Result = Int
  type Cont = Int => Result

  def eval(e: Exp): Cont => Result = e match {
    case Const(v) =>
      k => k(v)
    case Add(e1, e2) =>
      k => eval(e1)(v1 => eval(e2)(v2 => k(v1+v2)))
    case Sub(e1, e2) =>
      k => eval(e1)(v1 => eval(e2)(v2 => k(v1-v2)))
  }

}
```

mit

```
Direct_Sem.eval(Add(Const(40), Const(2)))              // = 42
CPS_Sem.eval(Add(Const(40), Const(2)) )((v:Int) => v) // = 42
```

Das ist zunächst einmal nichts weiter als eine Umformulierung. Aus einer Funktion

$eval : Exp \to Result$

wird eine Funktion

$eval : Exp \to (Int \to Result) \to Result$

Das Ganze kann verallgemeinert werden. Statt

$eval : Exp \to Result$

nehmen wir eine Funktion f mit:

$f : A \to B$

und *cps*-ifizieren sie:

```
object CPS_Gen {
  // transform function into CPS
  def cpsIfy[A,B, R](f: A => B): A => (B => R) => R =
    (a:A) => lift(f(a))

  // lift a value to a CPS-value
  def lift[A, R](a: A): (A => R) => R = (f: A => R) => f(a)
}
```

Funktioniert das mit *eval*: Wird *Direct_Sem.eval* in *CPS_Sem.eval* transformiert? Testen Sie *cpsIfy*!

Aufgabe 6

Der *Continuation*-Parameter kann allgemein genutzt werden um den "(Verwendungs-) Kontext" eines Wertes zu modellieren. Im einfachsten Fall wird der Kontext auf den Wert angewendet. Das haben wir oben betrachtet:

$lift(a) = c => c(a)$

Bei Bedarf kann der Kontext aber auch kreativer eingesetzt werden.

Ein Ausdruck *e* in einem *Try-Catch*-Konstrukt

`try` *e* `catch` *clauses*

hat beispielsweise einen Kontext, der aus zwei Fortsetzungen besteht: Im Erfolgsfall den normalen, im Fehlerfall die *Catch*-Klauseln.

Betrachten wir konkret einen Backtracking-Algorithms, der ganz klassisch mit Exceptions implementiert wird:

```
object Backtracking {

  class BacktrackException extends Exception

  trait State {
    type Step                    // Type of steps to successor state
    def isSolution: Boolean      // is this state a solution
    def isAcceptable(step: Step): Boolean // is step an acceptable step to go
    def nextSteps: List[Step]    // all possible steps from this state
    def go(step: Step): State    // successor state when step is taken
  }

  def solve(s: State) : State = if (s.isSolution) {
    s
  } else {
    for (step <- s.nextSteps) {
      try {
        if (s.isAcceptable(step)) {
          return solve(s.go(step))
        }
      } catch {
        case e:BacktrackException =>
      }
    }
    throw new BacktrackException()
  }

}
```

Mit diesem kann beispielsweise das N-Damen-Problem gelöst werden:

```
case class Board(
    n: Int,
    private val filledUpTo: Int,
    private val placement: List[Int]
  )
  extends Backtracking.State {

  type Step = Int //in range (0 .. n-1)

  def this(n:Int) = this(n, -1, List())
```

```
  override def isSolution: Boolean = filledUpTo == n-1

  override def isAcceptable(x: Step): Boolean =
    (0 to filledUpTo) forall ( j =>
     { val i = filledUpTo+1
       val d = i - j
       ((placement(j) != x) && (placement(j) != x - d) && (placement(j) != x + d))
     })

  override def nextSteps = (0 until n).toList

  override def go(step: Step): Board =
     Board(n, filledUpTo+1, placement ++ List(step))

  override def toString(): String = placement.toString()
}

object Board {
  def apply(n: Int) = new Board(n)
}
```

Eine Anwendung wäre:

```
object Backtrack_App {
  def main(args: Array[String]): Unit = {
    println(
      "classical solution " + Backtracking.solve(Board(8))
    )
  }
}
```

Definieren Sie nun *solveCPS*, eine Variante von *solve* im *Continuation Passing Style*, die keine Exceptions, sondern *Continuations* verwendet, um Rücksetzpunkte zu verwalten.

Ein Anwendungsbeispiel wäre etwa:

```
object Backtrack_App {
  def main(args: Array[String]): Unit = {
    Backtracking.solveCPS(
       Board(8),
       {b => println("CPS solution " + b)},
       { println("no Solution") } )
  }
}
```