Exercise 08

Task 1: Continuations

Discuss advantages and application scenarios for continuations. Also discuss disadvantages.

Answer:

Some ideas:

Advantages:

- captures the rest of program - allows a lot of flexibility, e.g. continuation can be restarted later if there is an error, or it can be replaced with another continuation (even during runtime)
- avoid issues with mutability
- allows for transforming a program so that it only contains tail calls (and then tail call optimization can be applied everywhere)

Application scenarios:

- Web programming
- Web servers etc. - here continuations can be particularly useful for recovering after e.g. a connection loss/server crash, to pick up a computation at the point where it was interrupted
- can be used for implementing new control constructs (exceptions, backtracking, pattern matching,...)

Disadvantages:

- sometimes harder to read and understand a program

Task 2: CPS

CPS transform the following Scala programs:

1. A *recursive* function `sum(n: Int): Int` that sums all numbers from `1` to `n`.
   (transformed function should have the signature `sum(n: Int, k: Int => Nothing): Nothing`)

Answer:

- non-CPS version:

```
def sum(n: Int): Int = {
  if (n == 1)
    1
  else
```

```
    n + sum(n - 1)
  }
```

CPS version (return type: Int)

```
def sum(n: Int, k: (Int) => Int): Int = {
  if (n == 0)
    k(0)
  else
    sum(n - 1, { res => k(n + res)})
}
```

To see why this works, it is instructive to step manually through a concrete call, e.g. with `n = 3`. As initial continuation, we use the identity function here. However, note that the calling continuation could be an arbitrary continuation passed on from a caller that expects the result of calling `sum`.

```
sum(3, res => res)             //k0 = (res => res)
=> sum(2, res => k0(3 + res)) //k1 = (res => k0(3 + res))
=> sum(1, res => k1(2 + res)) //k2 = (res => k1(2 + res))
=> k2(1)
=> k1(2 + 1)
=> k0(3 + 3)
=> 6
```

This function can be modified to return `Nothing` as follows

CPS version (return type: Nothing)

```
def sum(n: Int, k: Int => Nothing): Nothing = {
  if (n == 0)
    k(0)
  else
    sum(n - 1, { res => k(n + res)})
}

case class EndOfWorld(value:Any) extends Throwable
```

and we call this function as

```
sum(3, res => throw EndOfWorld(res))
```

1. A sort algorithm of your choosing (e.g., insertion sort, bubble sort, merge sort, tree sort). We recommend to avoid stateful computations as done in quick sort.

Answer:

Insertion sort, recursive non-CPS version:

```
def insert(elem: Int, list: List[Int]): List[Int] = {
  list match {
```

```
      case x :: xs =>
        if (elem < x)
          elem :: list
        else
          x::insert(elem, xs)
      case Nil => List(elem)
    }
  }

  def insertionSort(list : List[Int]): List[Int] = {
    list match {
      case x :: xs => insert(x, insertionSort(xs))
      case Nil => Nil
    }
  }
```

CPS version:

```
  def insert(elem: Int, list: List[Int], k: List[Int] => Nothing): Nothing = {
    list match {
      case x :: xs =>
        if (elem < x)
          k(elem :: list)
        else
          insert(elem, xs, {res => k(x :: res)})
      case Nil => k(List(elem))
    }
  }

  def insertionSort(list : List[Int], k: List[Int] => Nothing): Nothing = {
    list match {
      case x :: xs => insertionSort(xs,
        { res => insert(x, res, k) })
      case Nil => k(Nil)
    }
  }

  case class EndOfWorld(value:Any) extends Throwable
```

To understand why this works, it is instructive to manually execute a call to `insertionSort` with a concrete value, e.g. `2::1::Nil` and `(res => throw EndOfWorld(res))` as an initial continuation:

```
insertionSort(2::1::Nil, res => throw EndOfWorld(res))    //k0 =
(res => throw EndOfWorld(res))
=> insertionSort(1::Nil, res => insert(2, res, k0))    //k1 =
(res => insert(2, res, k0))
```

```
=> insertionSort(Nil, res => insert(1, res, k1))        //k2 =
(res => insert(1, res, k1))
=> k2(Nil)
=> insert(1, Nil, k1)
=> k1(1::Nil)
=> insert(2, 1::Nil, k0)
=> insert(2, Nil, res => k0(1::res))    //k4 = res => k0(1::res)
=> k4(2::Nil)
=> k0(1::2::Nil)
=> throw EndOfWorld(1::2::Nil)
```

mergeSort, recursive non-CPS version:

```
def merge(left: List[Int], right: List[Int]): List[Int] =
  (left, right) match {
  case (x :: xs, y :: ys) =>
    if (x < y)
      x::merge(xs, right)
    else
      y::merge(left, ys)
  case (Nil, _) => right
  case (_, Nil) => left
  }

def mergeSort(list: List[Int]): List[Int] = {
  if (list.size < 2)
    list
  else {
    val (left, right) = list.splitAt(list.size / 2)
    merge(mergeSort(left), mergeSort(right))
  }
}
```

CPS-version:

```
def merge(left: List[Int], right: List[Int], k: List[Int] => Nothing): Nothing =
  (left, right) match {
  case (x :: xs, y :: ys) =>
    if (x < y)
      merge(xs, right,
        { res => k(x :: res) })
    else
      merge(left, ys,
        { res => k(y :: res) })
  case (Nil, _) => k(right)
  case (_, Nil) => k(left)
}
```

```scala
def mergeSort(list: List[Int], k: List[Int] => Nothing): Nothing = {
  if (list.size < 2)
    k(list)
  else {
    val (left, right) = list.splitAt(list.size / 2)
    mergeSort(left,
      { resLeft => mergeSort(right,
        { resRight => merge(resLeft, resRight, k) })})
  }
}
```

1. A function that computes the size (that is, the number of case-class nodes) of an FAE program.

non-CPS version:

```scala
sealed abstract class FAE
case class Num(n: Int) extends FAE
case class Add(lhs: FAE, rhs: FAE) extends FAE
case class Sub(lhs: FAE, rhs: FAE) extends FAE
case class Id(name: Symbol) extends FAE
case class Fun(param: Symbol, bodyExpr: FAE) extends FAE
case class App(funExpr: FAE, argExpr: FAE) extends FAE

def countFAE(expr: FAE): Int =
  expr match {
    case Num(_) => 1
    case Add(lhs, rhs) => countFAE(lhs) + countFAE(rhs) + 1
    case Sub(lhs, rhs) => countFAE(lhs) + countFAE(rhs) + 1
    case Id(_) => 1
    case Fun(_, bodyExpr) => countFAE(bodyExpr) + 1
    case App(funExpr, argExpr) => countFAE(funExpr) + countFAE(argExpr) + 1
}
```

CPS version:

```scala
sealed abstract class FAE
case class Num(n: Int) extends FAE
case class Add(lhs: FAE, rhs: FAE) extends FAE
case class Sub(lhs: FAE, rhs: FAE) extends FAE
case class Id(name: Symbol) extends FAE
case class Fun(param: Symbol, bodyExpr: FAE) extends FAE
case class App(funExpr: FAE, argExpr: FAE) extends FAE


def countFAE(expr: FAE, k: Int => Nothing): Nothing =
  expr match {
    case Num(_) => k(1)
```

```
    case Add(lhs, rhs) => countFAE(lhs,
      { resLeft => countFAE(rhs,
        { resRight => k(resLeft + resRight + 1) })})
    case Sub(lhs, rhs) => countFAE(lhs,
      { resLeft => countFAE(rhs,
        { resRight => k(resLeft + resRight + 1) })})
    case Id(_) => k(1)
    case Fun(_, bodyExpr) => countFAE(bodyExpr,
      { res => k(res + 1) })
    case App(funExpr, argExpr) => countFAE(funExpr,
      { resFun => countFAE(argExpr,
              { resArg => k(resFun + resArg + 1) })})
}
```