Models of Computation (Sheet #3)

Marius Gavrilescu

1.
```
trait Change {
  def undo()
}

trait Command[T] {
  def execute(target: T): Option[Change]
}

trait Account {
  def balance: Int
  def deposit(amount: Int)
  def withdraw(amount: Int)
}

class DepositCommand(val amt : Int) extends Command[Account] {
  def execute(target : Account) : Option[Change] = {
    target.deposit(amt)
    return Some(new Change { def undo = target.withdraw(amt) })
  }
}

class WithdrawCommand(val amt : Int) extends Command[Account] {
  def execute(target : Account) : Option[Change] = {
    target.withdraw(amt)
    return Some(new Change { def undo = target.deposit(amt) })
  }
}
```

```
object Command {
  def makeAndThen[T](first : Command[T], second : Command[T]) : Command[T] = {
    return new Command[T] {
      def execute(target : T) : Option[Change] = {
        val result = first.execute(target)
        if(result.isEmpty) {
          None
        } else {
          val result2 = second.execute(target)
          result2 match {
            case None => result.get.undo(); None
            case Some(ch) => Some(new Change {
              def undo = { ch.undo(); result.get.undo() }
            })
          }
        }
      }
    }
  }

  def makeDoNothing[T]() : Command[T] = {
    return new Command[T] {
      def execute(target : T) : Option[Change] = {
        Some(new Change { def undo = {} })
      }
    }
  }

  def makeTransaction[T](commands : Array[Command[T]]) : Command[T] = {
    if(commands.isEmpty) return makeDoNothing()
    var cmd = commands(0)
    for(i <- 1 until commands.size) {
      cmd = makeAndThen(cmd, commands(i))
    }
    return cmd
  }
}
```

2. The editor amalgamates AmalgInsert changes by checking if the position of the other change is equal to the position of this change plus the length of this change. The user probably expects a movement command to break amalgamation. An easy way to do this is to keep a movement timestamp (similarly to problem 9 in the practical) in EdBuffer, increment it whenever moveCommand is called, save it when AmalgInsertion changes are created, and make the amalgamate method in AmalgChange refuse to amalgamate with a change that has a different movement timestamp.

3.
```
import scala.collection.mutable.PriorityQueue

class InsertCommand[T](val x : T) extends Command[PriorityQueue[T]] {
  def execute(target : PriorityQueue[T]) : Option[Change] = {
    target.enqueue(x)
    Some (new Change() {
      def undo() = {
        val newQueue = target.filterNot((y) => x == y)
        target.clear()
        for(elt <- newQueue)
          target.enqueue(elt)
      }})
  }
}

// This actually returns the maximum element if using the
// default comparators, because PriorityQueue is a max-heap
class DelMinCommand[T] extends Command[PriorityQueue[T]] {
  def execute(target : PriorityQueue[T]) : Option[Change] = {
    if (target.isEmpty)
      None
    else {
      val elt = target.dequeue()
      Some(new Change() {
        def undo() = target.enqueue(elt)
      })
    }
  }
}

type Condition[T] = T => Boolean
```

```
class WhileCommand[T](val test : Condition[T], val cmd : Command[T])
    extends Command[T] {
  def execute(target : T) : Option[Change] = {
    var changes : List[Change] = Nil
    var failed = false
    while(test(target) && !failed) {
      cmd.execute(target) match {
        case None => failed = true
        case Some(change) => changes = change :: changes
      }
    }
    val change = new Change() {
      def undo() = {
        for (ch <- changes) ch.undo()
      }
    }
    if(failed) {
      change.undo()
      None
    } else {
      Some(change)
    }
  }
}

def threshold[T <% Ordered[T]](limit : T) : Condition[PriorityQueue[T]] = {
  (x : PriorityQueue[T]) => x.head.compareTo(limit) < 0
}
```

4.
```
def isVowel(ch : Char) : Boolean = {
  ch match {
    case 'A' => true
    case 'E' => true
    case 'I' => true
    case 'O' => true
    case 'U' => true
    case _   => false
  }
}
```

```
class FilterIterator[T] (private val pred : T => Boolean,
    private val it : Iterator[T]) extends Iterator[T] {
  var buffer : Option[T] = None

  private def maybeAdvance() = {
    while(it.hasNext && buffer.isEmpty) {
      val candidate = it.next
      if(pred(candidate)) buffer = Some(candidate)
    }
  }

  def hasNext : Boolean = {
    maybeAdvance()
    !buffer.isEmpty
  }

  def next : T = {
    maybeAdvance()
    val ret = buffer.get
    buffer = None
    ret
  }
}
```

5.
```
scala> val mySet = HashSet[Any]()
mySet: scala.collection.mutable.HashSet[Any] = Set()

scala> mySet += new Point(0, 0)
res2: mySet.type = Set(Point@0)

[add toString to class Point]

scala> val mySet = HashSet[Any]()
mySet: scala.collection.mutable.HashSet[Any] = Set()

scala> mySet += new Point(5, 7)
res8: mySet.type = Set(5,7)

scala> mySet += new Point(5, 10)
res9: mySet.type = Set(5,10, 5,7)

scala> mySet += 70
res10: mySet.type = Set(5,10, 5,7, 70)

[ Make Point#equals return true for any integer ]

scala> val mySet = HashSet[Any]()
mySet: scala.collection.mutable.HashSet[Any] = Set()

scala> mySet += 70
res11: mySet.type = Set(70)

scala> mySet += 100
res12: mySet.type = Set(70, 100)

scala> mySet += 120
res13: mySet.type = Set(70, 100, 120)

scala> mySet += new Point(7, 1)
res14: mySet.type = Set(70, 100, 7,1, 120)

scala> mySet.contains(200)
res16: Boolean = false

scala> mySet.contains(7000001)
res17: Boolean = true
```

```
class Point(val x : Int, val y : Int) extends Ordered[Point] {
    override def equals(other : Any) : Boolean = {
        other match {
            case o : Point => o.x == x && o.y == y
            case x : Integer => true
            case _ => false
        }
    }

    def compare(that : Point) : Int = {
      if(x < that.x) return -1
      if(x > that.x) return 1
      return y - that.y
    }

    override def hashCode() : Int = x * 1000000 + y

    override def toString() : String = "" + x + "," + y
}
```

6.
```
import scala.collection.mutable.HashMap
import scala.collection.mutable.HashSet
import scala.collection.mutable.MultiMap
import scala.collection.mutable.Set
import scala.ref.WeakReference

class Node(val value: Integer, val next: Node) {
  override def equals(other : Any) : Boolean = {
    if(other.isInstanceOf[Node]) {
      val o : Node = other.asInstanceOf[Node]
      value == o.value && (next eq o.next)
    } else {
      false
    }
  }

  override def hashCode() : Int = {
    var hc = value
    if(next != null) {
      hc = hc * 101 + next.hashCode()
    }
    hc
  }
}

object Node {
  val mem : MultiMap[Integer, WeakReference[Node]] =
      new HashMap[Integer, Set[WeakReference[Node]]]
      with MultiMap[Integer, WeakReference[Node]];
  def extend(i: Integer, next: Node) : Node = {
    val nodes = mem.getOrElseUpdate(i, new HashSet[WeakReference[Node]]())
    val foundNode = nodes.find((wr) => wr.get.exists((n) => n.next eq next))
    foundNode match {
      case None =>
        val newNode = new Node(i, next)
        nodes.add(new WeakReference(newNode))
        newNode
      case Some(wr) => wr.get.get
    }
  }
}
```