Functional Programming with Scala

Homework 4

Instructor: Dr. Ionut Cardei

Chapter 6: practice with the Rand and State types.
Chapter 7: work with the parallelism library and contribute to the design.

Read the textbook chapter and the **lecture notes posted online** for material and examples not covered in the textbook.

Make sure you follow exactly all requirements listed in this document to get full credit.

Academic Honesty

For every assignment, including this, you have to work alone. Your submission must be your original work, that was not shared with anybody else. It is not enough to rename identifiers and move code around. Your solution for non-trivial answers (e.g. one liners) must be substantially different from any other.
Note that it is easy for the grader to find code or answers taken from online sources or to find solutions that are too similar.
Also, do not copy text (like examples, definitions) from notes/textbook or other sources, unless explicitly required to do so.
When in doubt, please contact the instructor at icardei@fau.edu.

Coding Requirements

For this assignment we can use Intellij IDEA to write the code. Emacs also accepted.

Don't forget to read the **Important Notes** at the end of this file.

This homework has 5 problems.
Total score: 100 points + extra credit.

Homework Formatting and Delivery

1. Start a new Word document called h4.docx. At the top write a title and your name.
2. Write the text answers for the problems in the given order (e.g. 1, 2, 3,...). Include the Scala code from your source files where applicable. Do not reorder solutions in the document.
3. Although syntax highlighting is not required, it is very appreciated by the grader, as it makes your code more readable, hence more likely to prevent grading mistakes. Paste directly from IDEA or check out http://hilite.me.
4. Make sure the code builds and runs correctly.

5. Convert the Word file to PDF.
6. Upload the PDF file with the answers and the Scala files on Canvas.

Follow strictly all requirements to get 100% credit. E.g. if a problem says: “use function foldRight” then your solution must use function foldRight to get full credit.

For this homework you will need the two source files discussed in class and that come with the solutions to textbook exercises: State.scala and Par.scala.

Problem 1.

Write the solution in a file called p1.scala, in an object called p1. Use the functions from the RNG object.

a) Write a state action (called *rollDice)* for the Rand[Int] type that produces a pair (x, y) of random integers with 0 <= x, y < 7.

b) Write a state action called *unitCirclePoint* for the Rand[Int] that produces a (x, y) pair so that the 2-D point (x, y) is on the unit circle, i.e. $x^2+y^2=1$.

c) Write a function called *myPie* that estimates (and returns) the value of π like this: first generate a large number *n* of 2D (x, y) points, with $x, y \in [0,1)$ and then measure the ratio of the number of points generated inside the unit disk vs. *n*. With a bit more math you get an approximation of π. Use function *sequence*.

d) Write a function *main* in object p1 that illustrates how the above functions can be used.

Problem 2.

Write the solution in a file called p2.scala, in an object called p2. Use the *State* type and the functions from the RNG object. The *Stack* and *Counter* examples from the lecture notes could be useful.

a) Write a state action combinator **inside class *State*** called *cond* with the following signature:

```
def cond[B](f: A => Boolean, btrue: =>B, bfalse: => B): State[S, B]
```

If *sa* is of type State[S,A], expression *sa.cond*(*f*)(*btrue*)(*bfalse*) returns a new state action that passes the value generated by *sa* to function *f* and if *f*(*a*) is true, returns expression *btrue*, else it returns *bfalse*.

Here is an example how to use *cond*:

```
val intSa: State[RNG, Int] = State(RNG.nonNegativeInt)   // s.a. for int >= 0

// new s.a. generating string "even" if intSa returns even number, else "odd":
val evenOdd: State[RNG, String] = intSa.cond(_ % 2 == 0, "even", "odd")

val rng: RNG.Simple = RNG.Simple(2L)
```

```
val rndEvenOddStr: (String, RNG) = evenOdd.run(rng)
println(rndEvenOddStr)  // prints: (odd,Simple(50429807845))
```

Write a version of function *cond* that uses the *map* function and another version that does not use *map*.

b) Write a polymorphic (in type B) state action method called *probCond* that takes at least these parameters:
- a State[RNG, Double] state action called *ps* that generates a random number in interval [0, 1).
- a Double argument *prob* in interval [0, 1).
- non-strict argument called *btrue* of type B
- non-strict argument called *bfalse* of type B

The *probCond* method returns value *btrue* with probability *prob* and value *bfalse* with probability 1 - *prob*. Your implementation must use method *cond* from class State (from part **a)**) and must sample random Doubles using state action *ps*.
Hint: if a number *r* is randomly uniformly distributed in [0,1) and *p* is in [0,1] then Prob($r < p$) = *p*. Use action state *ps* to obtain *r*.

c) Write a state action called *unfairCoinTosses* with arguments *n* and *prob* that produces a string of length *n* where each character in the string is 'H' with probability *prob* and 'T' with probability 1-prob. Use functions *sequence* and *probCond*.

d) Consider this class definition for a university course:

```
case class Course(name: String, roster: List[String])
```

name is the course name (e.g. "COT6789") and *roster* is a list with the names of the students currently enrolled.
Treat *Course* as a state type and write in object p2 these state actions (methods returning State[Course, ...]) :

```
def enroll(student: String): State[Course, Unit]

def enrolled(student: String): State[Course, Boolean]

def count: State[Course, Int]
```

enroll adds the student name give to the course roster if it was not already there.

enrolled returns true if the student name given is on the roster or false.

count returns the number of students on the roster.

Your implementations must use these State methods: *set*, *get*, and *modify*.
Hint: take inspiration from the Counter and Stack examples from the Chapter 6 lecture notes.

e) Write a *main* method in object p2 that demonstrates how to use the functions written for parts a-c). Do not use the same code from the example given in part a).
Write code with *for* comprehension(s) demonstrating the use of the Course class and **all** its state actions from part d).

Problem 3.

Write the solution in a file called p3.scala, in an object called p3. Use the *Par* type discussed in class and linked above.
In this problem we will write a parallel version of an algorithm and we will benchmark it using the *timeIt* function posted on the homework's Canvas page.

a) Write a sequential tail recursive function with the following signature:

```
def partition[A](s: IndexedSeq[A], pivot: A)(less: (A, A) => Boolean) :
                                    (IndexedSeq[A], IndexedSeq[A], IndexedSeq[A])
```

that splits sequence *s* (e.g. a Vector[Int]) in three sequences returned in a tuple: one with elements ***less*** than *pivot*, one with elements equal to *pivot*, and a third – with elements greater than *pivot*.
Your function should traverse the sequence only once, so using *filter* or *foldLeft/Right* is wrong.

Example:

```
   partition(Vector(5, 0, 3, 4, 1, 5, 1, 3, 9, 2, 7, 4, 3, 6), 4)(_ < _)
```

returns tuple `(Vector(0, 3, 1, 1, 3, 2, 3),Vector(4, 4),Vector(5, 5, 9, 7, 6))`

b) Functions that parallelize algorithms that are too 'fine grained' can actually be slower than a sequential version due to the overhead of sequence manipulation and thread management. For instance, the parallel *sum* version from the book splits the sequence in two if its length is greater than 1. For simple tasks (e.g. summation, computing maximum) breaking down vectors of a small size (e.g. 2) and proceeding in parallel is more costly than doing the calculation sequentially, for instance using the *sum* and *max* methods from IndexSeq.

Write a parallel and pure version of *partition* called *parPartition* that:

- returns type Par[(IndexedSeq[A], IndexedSeq[A], IndexedSeq[A])]
- when executed with *run* it computes the same tuple as function *partition*
- delegates the work to the sequential function *partition* if the size of the sequence is less than an *Int* parameter called *threshold*

Hint: follow the approach used for the parallel function *sequenceBalanced* from the textbook.
Make sure to check your function returns the same correct values as *partition*.

c) Write a function called *benchmark* with arguments *threshold* and *n* that:

- builds a random Vector[Int] called *vec* of size *n* using the *RNG.Simple* type from Chapter 6
- uses function *timeIt* to compute the average execution time for the sequential *partition* function on *vec*, *pivot* being *vec(0), and less* being `_ < _` , with 10 iterations
- uses function *timeIt* to compute the average execution time for the *parPartition* function **on the same Vector** *vec* with the given threshold, *pivot* being *vec(0), and less* being `_ < _` , with 10 iterations

- returns the speedup, i.e. ratio of the sequential time vs. the parallel time on the same input vector.

Use a thread pool with at least 1000 threads.

Write a function called *mkChartData* that runs function benchmark for a constant vector length of $n=2^{22}$ and varies the *threshold* parameter in this list: [1, n / 2, n / 4, n / 8, n / 16, n / 32, n / 64], and that returns a list with the speedup numbers obtained.

Create a line chart (e.g. in Excel) that shows on the horizontal axis the *threshold* value and the speedup on the vertical axis.

Include in the h4.pdf file the chart figure after the source code from p3.scala.

Then, do this:

c1) Indicate how many cores are on your computer and how much RAM it has.
c2) Analyze the results from the charts
c3) Comment on the results.

Problem 4.

Write the solution in a file called p4.scala, in an object called p4. Use the *Par* type discussed in class and linked above.

a) Write a function *parZipWith* with the signature below that returns a Par[C] that executes *f(a, b)*, in a separate logical thread, on parameters *a* from lists *sa* and on parameters *b* from *sb*.

```
def parZipWith[A,B,C](sa: List[A], sb: List[B])(f: (A, B) => C): Par[List[C]]
```

Try make the function tail recursive.
Hint: use function *Par.sequence*.

b) Write function *parOrElse* with the signature below that computes thunks *opt* and *alternative* in parallel and returns the evaluated value of *opt* if not *None* or else it returns the evaluated value of *alternative*. The value returned must be the same as for expression *opt.orElse*(*alternative*) the difference being that both *opt* and *alternative* are computed in parallel.

```
def parOrElse[A, B >: A](opt: =>Option[A], alternative: => Option[B]):
                                                    Par[Option[B]]
```

c) Write a main method in class p3 that demonstrates how to use the functions from parts a) and b).

d) Prove the following law using the substitution principle:

asyncF(f)(x) == unit(f(x))

e) Prove the following law using the substitution principle:

map(map(y)(g))(f) == map(y)(f compose g)

(it looks harder than it is!)

EXTRA CREDIT : 10 points
Problem 5

Write the solution in a file called p5.scala, in an object called p5. Use the *Par* type discussed in class and linked above.
a) Use the *parPartition* function from Problem 3 to implement a parallel (and pure) version of the Quicksort algorithm.

b) Benchmark it compared to the sequential version of the same algorithm using the *timeIt* function.
c) Include a chart showing the dependence of the sorting time and parallel speedup on the random vector's length, using *threshold*=1000. Pick a range of vector sizes that reaches into
d) Comment on the results.

Grading:

Problems 1-4 are worth 25 points each.

The points for programming problems are split into:

- code correctness: 85%
- programming style: 15%.

IMPORTANT NOTES:

- A submission that does not follow the instructions 100% (i.e. perfectly) will not get full credit.
- Submit the **h4.pdf** PDF file and the required .scala source files.
- Only submissions uploaded before the deadline will be graded.
- You have unlimited attempts to upload this assignment, but only the last one will be graded.