Solve 164 in Scala

Andrea

Let's try to solve this in Scala: https://www.4clojure.com/problem/164

We create a directory structure for sbt:

```
mkdir -p /tmp/problem-164-in-scala/problem164scala/src/test/scala /tmp/problem-164-in-scala/
```

First we define the software we are going to use to have a reproducible build:

```
{ }:
let
pkgs = import <nixpkgs> {};
stdenv = pkgs.stdenv;
sbt = pkgs.sbt;
scala = pkgs.scala;

in stdenv.mkDerivation {
name = "test_derivation";

buildInputs = [ sbt scala ];
}
```

We create a sandbox with this software installed:

```
nix-shell .
```

Then we import the test frameworks to test our solution:

```
name := "problem164scala"

// unit testing
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.1"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.1" % "test"
```

We create an sbt project for the Emacs Ensime mode:

```
sbt clean ensimeConfig
```

Then let's translates the tests given by https://www.4clojure.com/problem/164:

```
package problem164scala

import org.scalatest._
```

```
class GivenTests extends FunSuite with Matchers {

def makeDFA(states: Set[String], alphabet: Set[String], start: String, accepts: Set[String],
val states1 = states.map((x:String) => State(x))
val alphabet1 = alphabet.map((x:String) => Word(x))
val start1 = State(start)
val accepts1 = accepts.map((x:String) => State(x))
val transitions1 = transitions.map((flt:(String,String,String)) => Transition(State(flt._1),
DFA(states1,alphabet1,start1,accepts1,transitions1)
}

test("produce a set {'a' 'ab' 'abc'} for the given DFA") {
val states = Set("q0","q1","q2","q3")
val alphabet = Set("a","b","c")
val start = "q0"
val accepts = Set("q1","q2","q3")
val transitions = Set(("q0","a","q1"),("q1","b","q2"),("q2","c","q3"))
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
Solution.solve(dfa) should be (Set("a","ab","abc"))
}

test("produce a set {'hi' 'hey' 'hello'} for the given DFA") {
val states = Set("q0","q1","q2","q3","q4","q5","q6","q7")
val alphabet = Set("e","h","i","l","o","y")
val start = "q0"
val accepts = Set("q2","q4","q7")
val transitions = Set(("q0","h","q1"),("q1","i","q2"),("q1","e","q3"),("q3","l","q5"),("q3",
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
Solution.solve(dfa) should be (Set("hi","hey","hello"))
}


test("produce set created by list comprehension for the given DFA") {
val states = Set("q0","q1","q2","q3","q4")
val alphabet = Set("v","w","x","y","z")
val start = "q0"
val accepts = Set("q4")
val transitions = Set(("q0","v","q1"),("q0","w","q1"),("q0","x","q1"),("q0","y","q1"),("q0",
         ("q1","v","q2"),("q1","w","q2"),("q1","x","q2"),("q1","y","q2"),("q1","z","q2"),
         ("q2","v","q3"),("q2","w","q3"),("q2","x","q3"),("q2","y","q3"),("q2","z","q3"),
         ("q3","v","q4"),("q3","w","q4"),("q3","x","q4"),("q3","y","q4"),("q3","z","q4"))
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
val s = "vwxyz"
val result =
(for
(v <- 0 until 5;
w <- 0 until 5;
x <- 0 until 5;
y <- 0 until 5)
yield
(s(v)::s(w)::s(x)::s(y)::Nil).mkString
)
Solution.solve(dfa) should be (result.toSet)
}



// FIXME traduce tests from clojure https://www.4clojure.com/problem/164
}
```

And let's satisfy the tests writing some code:

```
package problem164scala

case class State[S](state: S)

case class Word[W](word: W)

case class Transition[S,W](from: State[S], label: Word[W], to: State[S])

case class DFA[S,W](states: Set[State[S]], alphabet: Set[Word[W]], start: State[S], accepts:

object Solution {

def solve(dfa: DFA[String,String]): Set[String] = {

def collectFinished(_starts: => Set[(State[String],String)], _acc: Set[String]) : Set[String
lazy val starts = _starts
def f1(acc: Set[String], kv: (State[String],String)) : Set[String] = {
kv match {
case (k,v) => {
dfa.accepts.contains(k) match {
case true => // we can return the string
acc + v
case false => //we cannot return the string
acc
}
}
}
}
starts.foldLeft(_acc)(f1)
}
```

```
def leaveOnlyTransitionable(_starts: => Set[(State[String],String)]): Set[(State[String],Str
// get all accepted states from which no transition start
val ss = dfa.accepts.filterNot(s => dfa.transitions.exists(t => t.from == s))
// we do not want the pairs with a state in [ss]
_starts.filterNot(kv => ss.contains(kv._1))
}

def applyAvailableTransitions(_starts: => Set[(State[String],String)]): Set[(State[String],S
def f1(acc: Set[(State[String],String)], kv: (State[String],String)): Set[(State[String],Str
// get all transitions starting from k
val kTransitions = dfa.transitions.filter(t => t.from == kv._1)
// apply transition
val pacc = kTransitions.map(t => (t.to,kv._2 + t.label.word))
acc ++ pacc
}
_starts.foldLeft(Set():(Set[(State[String],String)]))(f1)
}

def solve1(_starts: => Set[(State[String],String)], _acc: Set[String]): (Set[(State[String]
lazy val starts = _starts
// we collect all the strings that are finished
val acc = collectFinished(starts,_acc)
// then we clean all the states that are accepted and that
// cannot run a transition from _starts
val starts0 = leaveOnlyTransitionable(starts)
// then we update [starts0] by applying the transitions
val starts1 = applyAvailableTransitions(starts0)
(starts1,acc)
}


def recsolve1(_starts: => Set[(State[String],String)], _acc: Set[String]): Set[String] = {
lazy val (starts,acc) = solve1(_starts,_acc)
if (starts.isEmpty) {
acc
} else {recsolve1(starts,acc)}
}

recsolve1(Set((dfa.start,"")),Set())

}
}
```

Now that we have the core logic of the program, let’s add the tests on laziness:

```
package problem164scala
```

```scala
import org.scalatest._

class GivenTests extends FunSuite with Matchers {

def makeDFA(states: Set[String], alphabet: Set[String], start: String, accepts: Set[String],
val states1 = states.map((x:String) => State(x))
val alphabet1 = alphabet.map((x:String) => Word(x))
val start1 = State(start)
val accepts1 = accepts.map((x:String) => State(x))
val transitions1 = transitions.map((flt:(String,String,String)) => Transition(State(flt._1),
DFA(states1,alphabet1,start1,accepts1,transitions1)
}

test("produce stream {'a' 'ab' 'abc'} for the given DFA") {
val states = Set("q0","q1","q2","q3")
val alphabet = Set("a","b","c")
val start = "q0"
val accepts = Set("q1","q2","q3")
val transitions = Set(("q0","a","q1"),("q1","b","q2"),("q2","c","q3"))
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
Solution.solve(dfa) should be (Stream("a","ab","abc"))
}

test("produce a stream {'hi' 'hey' 'hello'} for the given DFA") {
val states = Set("q0","q1","q2","q3","q4","q5","q6","q7")
val alphabet = Set("e","h","i","l","o","y")
val start = "q0"
val accepts = Set("q2","q4","q7")
val transitions = Set(("q0","h","q1"),("q1","i","q2"),("q1","e","q3"),("q3","l","q5"),("q3",
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
Solution.solve(dfa) should be (Stream("hi","hey","hello"))
}


test("produce stream created by list comprehension for the given DFA") {
val states = Set("q0","q1","q2","q3","q4")
val alphabet = Set("v","w","x","y","z")
val start = "q0"
val accepts = Set("q4")
val transitions = Set(("q0","v","q1"),("q0","w","q1"),("q0","x","q1"),("q0","y","q1"),("q0",
         ("q1","v","q2"),("q1","w","q2"),("q1","x","q2"),("q1","y","q2"),("q1","z","q2"),
         ("q2","v","q3"),("q2","w","q3"),("q2","x","q3"),("q2","y","q3"),("q2","z","q3"),
         ("q3","v","q4"),("q3","w","q4"),("q3","x","q4"),("q3","y","q4"),("q3","z","q4"))
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
val s = "vwxyz"
val result =
(for
(v <- 0 until 5;
w <- 0 until 5;
x <- 0 until 5;
y <- 0 until 5)
yield
(s(v)::s(w)::s(x)::s(y)::Nil).mkString
)
Solution.solve(dfa).sorted should be (result.sorted.toStream)
}

test("produce stream 01 for the given DFA and test by property") {
val states = Set("q0","q1")
val alphabet = Set("0","1")
val start = "q0"
val accepts = Set("q0")
val transitions = Set(("q0","0","q0"),("q0","1","q1"),
         ("q1","0","q1"),("q1","1","q0"))
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
val stream = Solution.solve(dfa)
val res = stream.take(2000)
val pred1 = res.forall(s => s.matches("""0*(?:10*10*)*""".r.toString()))
val pred2 = res == res.distinct
(pred1 && pred2) should be (true)
}


test("produce stream nm for the given DFA and test by property") {
val states = Set("q0","q1")
val alphabet = Set("n","m")
val start = "q0"
val accepts = Set("q1")
val transitions = Set(("q0","n","q0"),("q0","m","q1"))
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
val stream = Solution.solve(dfa)
val res = stream.take(2000)
val pred1 = res.forall(s => s.matches("""n*m""".r.toString()))
val pred2 = res == res.distinct
(pred1 && pred2) should be (true)
}


test("produce stream ilompt for the given DFA and test by property") {
val states = Set("q0","q1","q2","q3","q4","q5","q6","q7","q8","q9")
val alphabet = Set("i","l","o","m","p","t")
val start = "q0"
val accepts = Set("q5","q8")
val transitions = Set(("q0","l","q1"),
         ("q1","i","q2"),("q1","o","q6"),
         ("q2","m","q3"),
         ("q3","i","q4"),
         ("q4","t","q5"),
         ("q6","o","q7"),
         ("q7","p","q8"),
         ("q8","l","q9"),
         ("q9","o","q6"))
val dfa = makeDFA(states,alphabet,start,accepts,transitions)
val stream = Solution.solve(dfa)
val res = stream.take(2000)
val pred1 = res.forall(s => s.matches("""limit|(?:loop)+""".r.toString()))
val pred2 = res == res.distinct
(pred1 && pred2) should be (true)
}
}
```

We add laziness to the logic of the program by introducing the Stream (also let's use Seq instead of Sets):

```
package problem164scala

case class State[S](state: S)

case class Word[W](word: W)

case class Transition[S,W](from: State[S], label: Word[W], to: State[S])

case class DFA[S,W](states: Set[State[S]], alphabet: Set[Word[W]], start: State[S], accepts:

object Solution {

def solve(_dfa: => DFA[String,String]): Stream[String] = {
lazy val dfa = _dfa
def collectFinished(_starts: Seq[(State[String],String)], _acc: Seq[String]) : Seq[String] =
lazy val starts = _starts
def f1(acc: Seq[String], kv: (State[String],String)) : Seq[String] = {
kv match {
case (k,v) => {
dfa.accepts.contains(k) match {
case true => // we can return the string
acc :+ v
case false => //we cannot return the string
acc
```

```
}
}
}
}
starts.foldLeft(_acc)(f1)
}

def leaveOnlyTransitionable(_starts: Seq[(State[String],String)]): Seq[(State[String],String
// get all accepted states from which no transition start
val ss = dfa.accepts.filterNot(s => dfa.transitions.exists(t => t.from == s))
// we do not want the pairs with a state in [ss]
_starts.filterNot(kv => ss.contains(kv._1))
}

def applyAvailableTransitions(_starts: Seq[(State[String],String)]): Seq[(State[String],Stri
def f1(acc: Seq[(State[String],String)], kv: (State[String],String)): Seq[(State[String],Str
// get all transitions starting from k
val kTransitions = dfa.transitions.filter(t => t.from == kv._1)
// apply transition
val pacc = kTransitions.map(t => (t.to,kv._2 + t.label.word))
acc ++ pacc
}
_starts.foldLeft(List().view:(Seq[(State[String],String)]))(f1)
}

def solve1(_starts: Seq[(State[String],String)], _acc: Seq[String]): (Seq[(State[String],Str
lazy val starts = _starts
// we collect all the strings that are finished
val acc = collectFinished(starts,List().view)
// then we clean all the states that are accepted and that
// cannot run a transition from _starts
val starts0 = leaveOnlyTransitionable(starts)
// then we update [starts0] by applying the transitions
val starts1 = applyAvailableTransitions(starts0)
(starts1,acc)
}


def recsolve1(_starts: Seq[(State[String],String)], _acc: Seq[String]): Stream[String] = {
val (starts,acc) = solve1(_starts,_acc)
if (starts.isEmpty) {
acc.toStream
} else {acc.toStream #::: recsolve1(starts,acc.view)}
}

recsolve1(List((dfa.start,"")), List())
```

```
}
}
```

Finally, let's check that all tests pass:

```
sbt test
```