Save your Stack
Lambda Conf 2015

Vincent Marquez
@runT1ME
github/vmarquez

Recursion is awesome, but can be dangerous on the JVM.

```
def generateDates(sd: DateTime, ed: DateTime): List[DateTime] = {
    if (sd isBefore ed)
      sd :: generateDates(sd.plusDays(1), ed) //cons is the last call
    else
      List(sd)
  }
```

What is a stack overflow

A stack overflow is an undesirable condition in which a particular computer program tries to use more memory space than the call stack has available. In programming, the call stack is a buffer that stores requests that need to be handled. —Google

Scalac can prevent SOs when recursing if the recursive call is the last method called in an execution path

Use Tail Recursion with an accumulator

```
def genDatesSafe(sd: DateTime, ed: DateTime): List[DateTime] = {
   def rec(s: DateTime, l: List[DateTime]): List[DateTime] =
      if (s isBefore endDate)
        rec(s.plusDays(1), s :: l) //last call is rec, itself
      else
        list
    rec(sd, List())
  }
```

Tree Structures can be hard to deal with even with an accumulator

```
sealed trait FS
  case class File(s: String) extends FS
  case class Directory(s: String, l: List[FS]) extends FS {
    override def toString(): String =
      s + " children size = " + l.size //why is this here? funny story…
  }
```

Let’s generate one for testing. first try failed :(

```
def generateFakeFiles(h: Int, w: Int): FS =  {
    def rec(h: Int): FS = h match {
      case 0 => Directory(h.toString, (0 to w).map(i => File(i.toString)).toList)
      case 1 => Directory(h.toString, (0 to w).map(_ => rec(h-1)).toList)
      case _ => Directory(h.toString, List(rec(h-1)))
    }
    rec(h)
}
```

Trampoline Monad to the rescue!

```
def generateDeepFakeFilesTrampolined(h: Int, w: Int): FS = {
  def rec(h: Int): Trampoline[FS] = h match {
    case 0 => Trampoline.done(Directory(h.toString, (0 to w).map(i => File("filefile")).toList))
    case 1 => (0 to w)
                .map(_ => rec(h -1))
                .toList
//sequence goes from F[G[A]] to G[F[A]],
// so in this case a List[Trampoline[FS]] to Trampoline[List[FS]]
                .sequence
               .map(l => Directory(h.toString, l)) //map on the Trampoline to get access
    case _ => rec(h-1).map(n => Directory(h.toString, List(n)))
  }
  rec(h).run
}
```

What is a trampoline? (approximation)

```
trait VTrampoline[A] {
  def flatMap[B](f: A => VTrampoline[B]): VTrampoline[B] = {
    More(() => this, f)
  }
}
case class More[A,B](a: () => VTrampoline[A],
  f: A => VTrampoline[B]) extends VTrampoline[B]

case class NoMore[A](a: A) extends VTrampoline[A]
```

Trampoline example again

```
def generateDeepFakeFilesTrampolined(h: Int, w: Int): FS = {
  def rec(h: Int): Trampoline[FS] = h match {
    case 0 => Trampoline.done(Directory(h.toString, (0 to w).map(i => File("filefile")).toList))
    case 1 => (0 to w)
                .map(_ => rec(h -1))
                .toList
//sequence goes from F[G[A]] to G[F[A]],
// so in this case a List[Trampoline[FS]] to Trampoline[List[FS]]
                .sequence
               .map(l => Directory(h.toString, l)) //map on the Trampoline to get access
    case _ => rec(h-1).map(n => Directory(h.toString, List(n)))
  }
  rec(h).run
}
```

When does a trampoline fail? Whenever the bind is nested… :(

```
(0 to 10000)
.map(ii => StateT[Free.Trampoline,Int,Int](i => Trampoline.done((i,ii))) )
.foldLeft( StateT[Free.Trampoline, Int,Int](i => Trampoline.done((i,0))) )
   ( (s,a) => s.flatMap(i => a.map(ii => (ii+i) ))) //this dies
```

Why does it die?

```
def flatMap[S, B](f: A => StateT[F, S, B])(implicit F: Bind[F]): StateT[F, S, B] =
  IndexedStateT(s => F.bind(apply(s)) {
        case (s1, a) => f(a)(s1)
    })
```

Bind happens AFTER function creation!

Are we hosed? No…
John De Goes had an idea for a better transformer

The F monad's bind happens first now!

```
case class JDState[F[_], S, A](sf: F[S => F[(S,A)]]) {

    def flatMap[B](f: A => JDState[F, S, B])(implicit M: Monad[F]): JDState[F, S, B] =
      JDState[F, S, B](((s1: S) => {
        sf.flatMap(sfa => sfa(s1).flatMap(t =>
          f(t._2).sf.flatMap(z => z(s1))
        ))
      }).point[F])
  }
```

Using it

```
val c = (0 to 10000)
.map(ii => JDState[Free.Trampoline,Int,Int](Trampoline.done((i:Int) =>
Trampoline.done((i,ii)))) )
.foldLeft( JDState[Free.Trampoline, Int,Int](Trampoline.done(i =>
    Trampoline.done((i,0))))))((s,a) => s.flatMap(i => a))


c.sf.map(i => i(0)).join.run
```

Another Option?, we can lift to a strait up Free Monad

```
(0 to 10000)
.map(ii => State[Int,Int](i => (i,ii)).liftF )
.foldLeft( State[Int,Int](i => (i,0)).liftF )( (s,a) =>
  s.flatMap(i => a.map(ii => (ii+i) )))
.foldRun(0)( (a,b) => b(a)) //magic is here!
```

Other ideas?