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), ) //cons is the last call
    else
      List(sd)
  }
```

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
  }
```

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) //we're done
      case 1 => Directory(h.toString, (0 to w).map(_ => rec(h-1)).toList) //can’t put this in tail position
      case _ => Directory(h.toString, List(rec(h-1)))
    }
    rec(h)
  }
```

Trampoline Monad to the rescue!

```
import scalaz.Free._
  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?

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) )))
```

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])
  }
```

Or, 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)) //we just have Frees, and we're going to fold throught he structure threading through state
```