Trampolines to Free Monad

Stackless Scala with Free Monads

Why this talk?

Blogs explain `Using Free Monad`

Hardly seen focusing on the aspect of being stack safe.

We will start with Trampolines to Free Monads, with a few hands on!

whoami?

Consultant - Simple Machines

In love with FP..

toTrampoline()

Tail call elimination in Scala is limited to self recursive methods.

It just fails to do TCE for mutually recursive calls.

Functions composed of many function calls prone to stack overflows

toTrampoline()

A general state monad.
Lets index lists using state monad.

```
case class State[S, +A](run: S => (A, S)) {
  def map[B](f: A => B): State[S, B] = flatMap(a => State.unit(f(a)))

  def flatMap[B](f: A => State[S, B]): State[S, B] = State(s => {
    val (a, s1) = run(s)
    f(a).run(s1)
  })

  def map2[B, C](sb: State[S, B])(f: (A, B) => C): State[S, C] =
    flatMap(a => sb.map(b => f(a, b)))
}

// an indexing function using state monad
def fun[A](list: List[A]) =
  list.foldLeft(State.unit[Int, List[(Int, A)]](List[(Int, A)]()))((acc, a) =>
  for {
    xs <- acc
    int <- get[Int]
    _ <- set[Int](int + 1)
  } yield (int, a) :: xs).run(0)._1.reverse
```

But when fun is called, it crashes with a StackOverflowError
in State.flatMap if the number of elements
in the list exceeds the size of the virtual machine’s
call stack

toTrampoline()

The reason is state action itself is a function composed of
smaller number of functions proportional to the length of the lists.

Each step calls the next step in a way compiler can’t optimize

Tail Call elimination in Scala

We are familiar with Tail recursions.

A typical tail recursion

```scala
def foldl[A, B](as: List[A], b: B)(f: (B, A) => B): B =
  as match {
    case Nil      => b
    case x :: xs => foldl(xs, f(b, x))(f)
  }
```

Compiler optimizes this tail recursion to a simple jump in the compiled code.

```scala
def foldl [A ,B ]( as : List [A] , b : B)
                 (f: (B , A) => B ): B = {
  var z = b
  var az = as
  while (true) {
    az match {
      case Nil => return z
      case x :: xs => {
        z = f (z , x)
        az = xs
      }
    }
  }
  z
}
```

This kind of optimization has two advantages: a jump
is much faster than a method invocation, and it requires no
space on the stack.

While self recursive calls is easy, replacing tail calls in general with jumps is fairly difficult in many cases. For example the mutual recursion cannot be optimized.

```
def even [ A ]( ns : List [ A ]): Boolean =
  ns match {
    case Nil => true
    case x :: xs => odd ( xs )
  }
def odd [A ]( ns : List [A ]): Boolean =
  ns match {
    case Nil => false
    case x :: xs => even ( xs )
  }
```

```
These functions will overflow the stack if the argument list is larger than the stack size.
```

Trampolines

```scala
sealed trait Trampoline [A] {
  final def runT : A =
    this match {
      case More (k) => k (). runT
      case Done (v) => v
    }
}
case class More [A]( k: () => Trampoline [A])
  extends Trampoline [A]

case class Done [A]( result : A)
  extends Trampoline [A]
```

This solves the mutual recursion problem we saw earlier.
All we have to do is mechanically replace any return type T with Trampoline[T].

Trampolining State Monad

```
def flatMap [B ]( f: A => State [S ,B]) =
  State [S ,B ]( s => More (() => {
    val (a , s1 ) = runS (s ). runT
    More (() => f(a ) runS s1 )
  }))

// This is not efficient, because the call to runT is
not in tail position.
```

A trampoline Monad

The unit is `Done`

All that it needs is Bind.

```
def flatMap [B ]( f: A => Trampoline [B ]) =
  More [B ](() => f( runT ))
```

And so we could use:

```
def flatMap [B ]( f: A => State [S ,B ]) =
  State [S ,B ]( s => More (() => runS ( s) flatMap {
    case (a , s1 ) => More (() => f(a) runS s1 )
  }))
```

```
// AND THAT DOESN'T WORK EITHER ???
```

A solution

```
case class FlatMap [A ,B ](
  sub : Trampoline [A] ,
  k: A => Trampoline [B ]) extends Trampoline [B]
```

```
A trampoline of this form can be thought of as a call to a
subroutine sub whose result is returned to the continuation
k.
```

```
// Finding the next step
final def resume :
 Either [() = > Trampoline [A], A] =
  this match {
    case Done (v) => Right (v )
    case More (k) => Left (k)
    case FlatMap (a ,f ) = > a match {
     case Done (v) => f( v ). resume
     case More (k) => Left (() =>
       FlatMap (k () , f ))
     case FlatMap (b ,g ) = > ( FlatMap (b ,
       (x: Any ) => FlatMap ( g(x) , f) ): Trampoline [A ]). resume }
  }
```

```
// Running all the steps
final def runT : A = resume match {
  case Right (a) => a
  case Left (k) => k (). runT
}
```

Note: We avoided the left associated nesting of FlatMap(FlatMap(b, g), f) by doing FlatMap (b , x => FlatMap (g(x), f))

Let’s correlate this to IO

// A simple IO

```
// A simple IO
trait IO {
  def run: Unit
}
```

NOT A GREAT IO

```
sealed trait IO[A] { self =>
  def run: A
  def map[B](f: A => B): IO[B] = new IO[B] { def run = f(self.run) }
  def flatMap[B](f: A => IO[B]): IO[B] = {
    new IO[B] { def run = f(self.run).run }
  }
}

implicit object IO extends Monad[IO] {
  def unit[A](a: => A): IO[A] = new IO[A] { def run = a }
  def flatMap[A, B](ma: IO[A])(f: (A) => IO[B]): IO[B] = ma flatMap f
  def apply[A](a: => A): IO[A] = unit(a)
}

def ReadLine = IO { StdIn.readLine }
def PrintLine(msg: String) = IO { println(msg) }

def fahrenheitToCelsius(f: Double): Double = (f - 32) * 5.0 / 9.0
```

NOT A GREAT IO

```scala
def converter: IO[Unit] = for {
  _ <- PrintLine("Enter a temperature in degree celsius")
  d <- ReadLine.map(_.toDouble)
  _ <- PrintLine(fahrenheitToCelsius(d).toString)
} yield ()

// More examples ??..
```

IO Without Stack Overflow

A BETTER IO

```scala
trait IO[A] {
  def flatMap[B](f: A => IO[B]): IO[B] = FlatMap(this, f)
  def map[B](f: A => B): IO[B] = flatMap(f andThen (Return(_)))
}


case class Return[A](a: A) extends IO[A]
case class Suspend[A](resume: () => A) extends IO[A]
case class FlatMap[A, B](bs: IO[A], f: A => IO[B]) extends IO[B]


@tailrec
def run[A](io: IO[A]): A = io match {
  case Return(a)  => a
  case Suspend(r) => r()
  case FlatMap(x, f) => x match {
    // we didn't do run(f(run(x)) to make things tail recursive
    case Return(a)     => run[A](f(a))
    case Suspend(r)    => run(f(r()))
    case FlatMap(y, g) => run(y flatMap (a => g(a) flatMap f))
  }
}


// Do some fun things with this type!
```

Function0 as Suspend?

When interpreter sees `FlatMap(Suspend(s),k)` it executes s() and wait..!!

Our IO was

```
type IO[A] = Free[Function0, A]
```

Free Monad

```
sealed trait Free[F[_], A] {
  def flatMap[B](f: A => Free[F, B]): Free[F, B] =
    FlatMap(this, f)
  def map[B](f: A => B): Free[F, B] =
    flatMap(f andThen (Return(_)))
}

case class Return[F[_], A](a: A) extends Free[F, A]
case class Suspend[F[_], A](s: F[A]) extends Free[F, A]
case class FlatMap[F[_], A, B](s: Free[F, A], f: A => Free[F, B]) extends Free[F, B]


It’s straight forward to create a monad instance for Free[F, ?] !
// Hands on!
```

Interpreter

```
def runFree[F[_], G[_], A](a: Free[F, A])(t: F ~> G)(implicit G: Monad[G]): G[A] = a match {
  case Return(v)   => G.unit(v)
  case Suspend(fa) => t(fa)
  case FlatMap(sub, cont) => sub match {
    case Return(v)                => runFree(cont(v))(t)
    case Suspend(fa)              => G.flatMap(t(fa))(a => runFree(cont(a))(t))
    case FlatMap(subsub, contcont) => runFree(subsub.flatMap(g = contcont(g)  flatMap (cont)))(t)
  }
}


A specialized interpreter if F is Function0, and see the difference!

@annotation.tailrec
def runTrampoline[A](a: Free[Function0, A]): A = a match {
  case Return(r)      => r
  case Suspend(thunk) => thunk()
  case FlatMap(sub, cont) => sub match {
    case Return(r)                => runTrampoline(cont(r))
    case Suspend(thunk)           => runTrampoline(cont(thunk()))
    case FlatMap(subsub, contcont) => runTrampoline(subsub.flatMap(g => contcont(g) flatMap (cont)))
  }
}

// Example project, show toThunk() and toReader()
```

Use!

```scala
/**
*/
trait ScalazConsole[A] {
  def toReader: Reader[String, A]
}

object ScalazConsole {

  case object ReadLine extends ScalazConsole[Option[String]] {
    override def toReader: Reader[String, Option[String]] = Reader(_.some)
  }

  case class PrintLine(string: String) extends ScalazConsole[Unit] {
    override def toReader: Reader[String, Unit] = Reader(_ => ())
  }

  def readLine[A]: Free[ScalazConsole, Option[String]] =
    Suspend(ReadLine)

  def printLine[A](s: String): Free[ScalazConsole, Unit] =
    Suspend(PrintLine(s))

  val f: Free[ScalazConsole, Option[String]] = for {
    _ <- printLine("I interact with only console")
    s <- readLine
  } yield s

  val translator = new (ScalazConsole ~> Reader[String, ?]) {
    def apply[A](a: ScalazConsole[A]): Reader[String, A] = a.toReader
  }

 implicit def readerMonad: Monad[Reader[String, ?]] = new Monad[Reader[String, ?]] {
    override def flatMap[A, B](ma: Reader[String, A])(f: (A) => Reader[String, B]): Reader[String, B] = ma.flatMap(f)
    override def unit[A](a: => A): Reader[String, A] = Reader(_ => a)
  }

  def reader: Reader[String, Option[String]] = FreeMonad.runFree[ScalazConsole, Reader[String, ?], Option[String]](f)(translator)

  // reader.run("Bob") and you test your flow f nicely without any side effect. This is a rather simple
  // way of saying that the tranlsated F should be pure like a Reader Monad that removes the side effect.
  // The actual side effect may be handled out side. Something like, scalaz.IO(readLine()).map( reader.run) etc.
}
```

:+1!