Trampolines

Making recursion stack safe

The Problem

- Call stack is limited
- Tail call elimination (TCE) in Scala is limited to self recursive functions
  => recursive method invocation is replaced by a jump
- TCE has two advantages:
  => a jump is much faster that a method invocation
  => a jump requires no space on the stack
- Currently the JVM allows only for local jumps
  => no way to directly implement a tail call to another method as a jump

Example

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

def odd[A](as: List[A]): Boolean = as match {
    case Nil      => false
    case _ :: xs => even(xs)
  }

val isEven = even(List.fill(10000)(0))
     => java.lang.StackOverflowError
```

Trampolines for the rescue

- Trading stack for heap
- A `Trampoline` represents a computation that can be stepped through
- Instead of doing a method invocation a `case class` is created

Trampolines

```scala
sealed trait Trampoline[+A]

case class Done[+A](a: A) extends Trampoline[A]
case class More[+A](k: () => Trampoline[A]) extends Trampoline[A]
```

Trampolines

```scala
sealed trait Trampoline[+A] {
  @annotation.tailrec
  final def run: A = this match {
    case Done(a) => a
    case More(k) => k().run
  }
}

case class Done[+A](a: A) extends Trampoline[A]
case class More[+A](k: () => Trampoline[A]) extends Trampoline[A]
```

Simple Solution

```
def even[A](as: List[A]): Trampoline[Boolean] = as match {
  case Nil     => Done(true)
  case _ :: xs => More( () => odd(xs) )
}

def odd[A](as: List[A]): Trampoline[Boolean] = as match {
  case Nil     => Done(false)
  case _ :: xs => More( () => even(xs) )
}

val isEven = even(List.fill(10000)(0)).run
    => true
```

Solution (Vanilla Scala)

```scala
import scala.util.control.TailCalls._

def even[A](as: List[A]): TailRec[Boolean] = as match {
  case Nil     => done(true)
  case _ :: xs => tailcall(odd(xs))
}

def odd[A](as: List[A]): TailRec[Boolean] = as match {
  case Nil     => done(false)
  case _ :: xs => tailcall(even(xs))
}

val isEven = even(List.fill(10000)(0)).result
```

Solution (Cats)

```scala
import cats.Eval

def even[A](as: List[A]): Eval[Boolean] = as match {
  case Nil     => Eval.now(true)
  case _ :: xs => Eval.defer(odd(xs))
}

def odd[A](as: List[A]): Eval[Boolean] = as match {
  case Nil     => Eval.now(false)
  case _ :: xs => Eval.defer(even(xs))
}

val isEven = even(List.fill(10000)(0)).value
```

Solution (Scalaz)

```
import scalaz.Free.Trampoline
import scalaz.Trampoline._

def even[A](as: List[A]): Trampoline[Boolean] = as match {
  case Nil     => done(true)
  case _ :: xs => suspend(odd(xs))
}

def odd[A](as: List[A]): Trampoline[Boolean] = as match {
  case Nil     => done(false)
  case _ :: xs => suspend(even(xs))
}

val isEven = even(List.fill(10000)(0)).run
```

Another example

```scala
def foldRight[A, B](as: List[A], acc: B)(f: (A, B) => B): B = {
  as match {
    case Nil    => acc
    case h :: t => f(h, foldRight(t, acc)(f))
  }
}
```

becomes

```scala
import cats.Eval

def foldRight[A, B](as: List[A], acc: B)(f: (A, B) => B): B = {
  def go(as: List[A], acc: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B] = as match {
    case h :: t => Eval.defer(f(h, go(t, acc)(f)))
    case Nil    => acc
  }

  go(as, Eval.now(acc)) { (a, b) => b.map( f(a, _) ) }.value
}
```

A Trampoline monad

- To be really useful `Trampoline` needs a monadic API
- For use within other monads
- Monadic API
  - `unit :: A => F[A]`
    (which is `Done(…)`)
  - `bind :: A => F[B] => F[B]`
    Need to implement `flatMap`

A Trampoline monad

```scala
sealed trait Trampoline[+A] {
  @annotation.tailrec
  final def run: A = this match {
    case Done(a) => a
    case More(k) => k().run
  }

  final def map[B](f: A => B): Trampoline[B] =
    More( () => Done(f(run)) )

  final def flatMap[B](f: A => Trampoline[B]): Trampoline[B] =
    More( () => f(run) )
}

case class Done[+A](a: A) extends Trampoline[A]
case class More[+A](k: () => Trampoline[A]) extends Trampoline[A]
```

NOT STACK SAFE !

A Trampoline monad

```scala
sealed trait Trampoline[+A] {

  def map[B](f: A => B): Trampoline[B] = flatMap(x => More(() => Done(f(x))))
  def flatMap[B](f: A => Trampoline[B]): Trampoline[B] = this match {
    case FlatMap(a, g) => FlatMap(a, (x: Any) => g(x) flatMap f)
    case x             => FlatMap(() => x, f)
  }

  @tailrec 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(() => k() flatMap f)
        case b FlatMap g => b().flatMap((x:Any) => g(x) flatMap f).resume
      }
    }

  @tailrec final def run: A = resume match {
    case Right(a) => a
    case Left(k) => k().run
  }
}

case class Done[+A](a: A) extends Trampoline[A]
case class More[+A](k: () => Trampoline[A]) extends Trampoline[A]
case class FlatMap[A, +B](a: () => Trampoline[A], k: A => Trampoline[B]) extends Trampoline[B]
```

A Generalization

- Trampoline suspends via a `Function0` constructor
- We abstract over that constructor

A Generalization

```
sealed trait Trampoline[+A]

case class Done[+A](a: A) extends Trampoline[A]
case class More[+A](k: () => Trampoline[A]) extends Trampoline[A]
```

becomes

```
sealed trait Trampoline[S[+ _], +A]

case class Done[S[+ _], +A](a: A) extends Trampoline[S, A]
case class More[S[+ _], +A](k: S[Trampoline[S, A]]) extends Trampoline[S, A]
```

A Generalization

```scala
sealed trait Free[S[+ _], +A]

case class Done[S[+ _], +A](a: A) extends Free[S, A]
case class More[S[+ _], +A](k: S[Free[S, A]]) extends Free[S, A]

type Trampoline[+A] = Free[Function0, A]
```

A Generalization

```scala
sealed trait Free[S[+ _], +A]

case class Done[S[+ _], +A](a: A) extends Free[S, A]
case class More[S[+ _], +A](k: S[Free[S, A]]) extends Free[S, A]

type Trampoline[+A] = Free[Function0, A]
```

Free monads everywhere!

Scalaz Free

```scala
sealed abstract class Free[S[_], A] {
  final def map[B](f: A => B): Free[S, B] = flatMap(a => Return(f(a)))
  final def flatMap[B](f: A => Free[S, B]): Free[S, B] = Gosub(this, f)

  @tailrec final def resume(implicit S: Functor[S]): (S[Free[S,A]] \/ A) =
    this match {
      case Return(a) => \/-(a)
      case Suspend(t) => -\/(S.map(t)(Return(_)))
      case b @ Gosub(_, _) => b.a match {
        case Return(a) => b.f(a).resume
        case Suspend(t) => -\/(S.map(t)(b.f))
        case c @ Gosub(_, _) => c.a.flatMap(z => c.f(z).flatMap(b.f)).resume
      }
    }
}

private case class Return[S[_], A](a: A) extends Free[S, A]
private case class Suspend[S[_], A](a: S[A]) extends Free[S, A]
private case class Gosub[S[_], A0, B](a0: Free[S, A0], f0: A0 => Free[S, B]) extends Free[S, B] {
  type A = A0
  def a: Free[S, A] = a0
  def f: A => Free[S, B] = f0
}
```

Free monad

- Many monads are based on `Free`
  - `scalaz.concurrent.Future`
  - `scalaz.concurrent.Task`
  - `scalaz.effect.IO`
  - `scalaz.Trampoline`
  - …

Recap

- General way to make any recursion stack safe
- `Trampoline`s trade stack for heap
- Building a `Trampoline` monad is hard
- Generalization of `Trampoline` leads to `Free`
- Don’t fear the `Free` monad ;-)
- So much to discover in FP !!

Reading

- http://blog.higher-order.com/assets/trampolines.pdf

Thanks!