Stackless Scala With Free Monads

Rúnar Óli Bjarnason

runarorama@gmail.com

Abstract

Tail call elimination (TCE) in the Scala compiler is limited to self-recursive methods, but tail calls are otherwise not eliminated. This makes functions composed of many smaller functions prone to stack overflows. Having a general TCE mechanism would be a great benefit in Scala, particularly for functional programming. Trampolining is a popular technique [6], which can be used for TCE in languages that don't support it natively. This paper gives an introduction to trampolines in Scala and expands on this solution to gain elimination of any method call whatsoever, even calls that are not in tail position at all. This obviates altogether the use of the call stack in Scala programs.

1. Introduction

Since the call stack is a limited resource of the virtual machine, most programmers who have some experience with Scala will have come up against the problem of a seemingly reasonable function running out of stack and crashing the program with a StackOverflowError.

As a practical example, consider traversing a list while maintaining some state. We will use the `State` data type, which represents a transition function in a simple state machine.

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

The `runS` function takes some input state of type `S` and outputs a value of type `A` together with a new state. The `map` and `flatMap` methods allow us to thread the state through a `for`-comprehension, in order to write imperative-looking programs using `State` combinators such as these:

```
def getState[S]: State[S,S] =
  State(s => (s,s))

def setState[S](s: S): State[S,Unit] =
  State(_ => ((),s))

def pureState[S, A](a: A): State[S, A] =
  State(s => (a,s))
```

Note that `pureState` and `flatMap` together make `State` a *monad* [4].

As a simple demonstration, let us write a function that uses `State` to number all the elements in a list. Not because this is a compelling use case for `State`, but because it's simple and it demonstrates the stack overflow.

```
def zipIndex[A](as: List[A]): List[(Int,A)] =
  as.foldLeft(
    pureState[Int, List[(Int,A)]](List())
  )((acc,a) => for {
    xs <- acc
    n  <- getState
    _  <- setState(n + 1)
  } yield (n,a)::xs).runS(0)._1.reverse
```

We use a left fold to emphasize that the traversal of the list is tail-recursive. The fold builds up a state action that starts with an empty list and adds successive elements to the front, reversing the original list. The state is an integer that we increment at each step, and the whole composite state action is run starting from zero before returning the reverse of the result.

But when `zipIndex` 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. The reason is that the state action itself is a function composed of a number of smaller functions proportional to the length of the list. Even though we think of it as a sequence of discrete steps, each step calls the next step in a way that the compiler can't optimize.

It would seem that this seriously limits the utility of functional programming in Scala. This paper will explore the space of solutions to this problem:

Submitted to The Third Scala Workshop, London, Apr 17th 2012.

- In section 3, we will discuss the well-known technique of *trampolining*. In a trampolined program, instead of each step calling the next, functions yield the next step to a single control loop known as the trampoline. This allows us to programmatically exchange stack for heap.
- We will then expand on this technique by adding an *operational monad* (section 4), which allows us to turn any call whatsoever into a tail call that can be subsequently eliminated. That will be the main contribution of this paper.
- There is a subtle detail in the implementation of this monad such that if it is not implemented correctly it will continue to overflow the stack in some cases. In section 4.3 we will look at what those cases are and how to render them harmless.
- Trampolined programs can be interleaved, providing a model of cooperative coroutines. We will see this in action in section 5.
- In section 6, we generalize trampolines to a *free monad*, an extremely versatile recursive data structure. We look at some functions that operate on all such structures (6.1) and find that the work we have already done for trampolines gives us the same benefits for all free monads.

2. Background: Tail-call elimination in Scala

The Scala compiler is able to optimize a specific kind of tail call known as a self-recursive call. For example, the following definition of a left fold over a list is optimized by the compiler to consume a constant amount of stack space:

```
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)
  }
```

When the compiler finds that a method calls itself in the tail position, and if that method cannot be overridden (e.g. because of being declared `private` or `final`), the recursive method invocation is replaced with a simple jump in the compiled code. This is equivalent to rewriting the tail-recursion as a loop:

```
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.

But while optimizing self-recursive calls is easy, replacing tail calls in general with jumps is more difficult. Currently, the Java virtual machine (JVM) allows only local jumps, so there is no way to directly implement a tail call to another method as a jump. For example, this mutual recursion cannot be optimized by the compiler, even though the calls are in tail position:

```
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 call stack if the argument list is larger than the stack size.

Although a future JVM may implement explicit support for tail calls in the bytecode, this is not without hurdles and may not be as useful as it sounds. For instance, the execution model of the JVM requires the state of each thread of execution to be stored on the thread's stack. Furthermore, exception-handling is implemented by passing an exception up the call stack, and the JVM exposes the stack to the programmer for inspection. In fact, its security model is implemented by looking at permissions granted to each stack frame individually. This, coupled with subclassing, dynamic dispatch, and just-in-time compilation conspires to make tail call optimization in the Scala compiler itself difficult to implement.

Fortunately we can sidestep all of those issues. There is a way that we can mechanically trade stack for heap by using a simple data structure.

3. Trampolines: Trading stack for heap

We begin with a very basic `Trampoline` data type. This is identical in spirit to but differs in implementation from `scala.util.control.TailCalls.TailRec`.

```
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]
```

A `Trampoline` represents a computation that can be stepped through, and each step can have one of two forms. A step of the form `Done(v)` has a value `v` to return and there are no more steps in that case. A step of the form `More(k)` has more work to do, where `k` is a closure that does some work and returns the next step. The `runT` method is a simple tail-recursive method that executes all the steps. It is made `final` so that Scala can eliminate the tail call.

This solves the mutual recursion problem we saw earlier. All we have to do is mechanically replace any return type `T` with `Trampoline[T]`. Here are `odd` and `even`, modified this way:

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

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

Instead of recursing directly, the functions now return the next step as a `Trampoline` which can be executed tail-recursively by calling its `runT` method. This no longer overflows the stack, no matter how large the argument lists are.

4. Making every call a tail call

Let's see if we can apply the `Trampoline` solution to the problem of traversing a list with `State` from before. We need to change the representation of `State` actions to return a trampoline that we can run tail-recursively:

```
case class State[S,+A](
  runS: S => Trampoline[(A,S)])
```

How do we now implement the `flatMap` method for composing `State` actions? We could try this:

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

But that turns out to be insufficient. The `zipIndex` example from section 1 will still overflow the stack for large lists, this time for even smaller lists. The problem is that the call to `runT` is not in the tail position, so it can't be optimized or wrapped in a `Trampoline`.

4.1 A Trampoline monad?

We will attempt to solve this by making `Trampoline` monadic. It already has a monadic unit[1], which is the `Done` constructor. All it needs is monadic bind, which is `flatMap`. Let's add a `flatMap` method directly to `Trampoline`, so we can do this in `State.flatMap`:

[1] A unit for a monad M is a function of type $A => M[A]$, for all A. It is a unit in the sense that passing it to $flatMap$ is an identity.

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

That's a definite improvement. It shifts the problem into the `flatMap` method for `Trampoline`, which we might be tempted to implement like this:

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

But that is not what we want. The call to `runT` is not in a tail position there either. It seems that no matter what we try it's simply not possible to implement a `flatMap` method for `Trampoline` that doesn't require any additional stack.

4.2 Building the monad right in

The way around this limitation is to add a constructor to the `Trampoline` data type, changing `flatMap` from a method call to a constructor call:

```
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`.

The `Trampoline` trait's `runT` method must now take this new constructor into account. To simplify, let's separate the concern of advancing to the next step from the concern of running all the steps:

```
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
  }
}

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

The `resume` method proceeds by pattern matching on the `Trampoline`, returning either the result (on the `Right`) or the next step as a `Function0` (on the `Left`).

The way we handle the `FlatMap(a,f)` case here is subtle but important. We match on the subroutine call `a`, and if it's `Done`, we simply run the continuation. If it's wrapped in a `More` constructor, we advance by one step and `FlatMap` over that. If the subroutine call itself contains a subroutine call, we have a left-associated nesting of `FlatMaps` in an expression like this:

```
FlatMap(FlatMap(b, g), f)
```

It's critical to resolve this case in such a way that remains productive without introducing new stack frames. The trick is to *re-associate the expression to the right*:

```
FlatMap(b, x => FlatMap(g(x), f))
```

When we do that, the next iteration will pattern match on `b`, and so we are able to make a productive tail call to `resume` again.

Since the call to resume is on `FlatMap` here, we must cast explicitly to `Trampoline` for the compiler to be able to figure out that this is in fact a tail-recursive self-call[2]. Don't worry, we will get rid of this cast in section 4.3.

Also note that when we look inside the nested `FlatMap` constructors, there is some type information that has been lost. In a pattern like `FlatMap(FlatMap(b, g), f)` the type of `b` cannot be known, so we must assume `Any` when we construct the right-associated nesting. This is perfectly safe, since we can assume the left-associated nesting was well typed when it was constructed.

This re-association is taking advantage of the monad laws. Trampoline is a monad, and monads are by definition associative. Therefore the right-associated continuations are always exactly equivalent to the left-associated ones.

4.3 An easy thing to get wrong

There is one more corner case to consider here. It's now possible for `resume` to overflow the stack if the left-leaning tower of `FlatMaps` is taller than the call stack. Then the call `f(v)` will make the call `g(x)`, which will make another inner call, etc. We avoid this by disallowing the construction of deeply nested left-associated binds in the first place. We make the `FlatMap` constructor `private`, exposing instead the `flatMap` method on `Trampoline`, which we rewrite to always construct right-associated binds:

```
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)
  }
```

To close the gap, we must also prevent the `resume` method from constructing such a tower, by replacing calls to the `FlatMap` constructor with calls to the `flatMap` method:

```
case FlatMap(a,f) => a match {
  case Done(v) => f(v).resume
  case More(k) => Left(() => k() flatMap f)
  case FlatMap(b,g) =>
    b.flatMap((x:Any) => g(x) flatMap f).resume
}
```

[2] Since the $runT$ method is declared $final$, there is no theoretical reason that the recursive call could be dispatched on a different class. It's possible that a future version of Scala will automatically infer this typecast.

Finally, in order to use our `Trampoline` monad with Scala's `for`-comprehensions we also need to implement `map`, which is of course just defined in terms of `flatMap`:

```
def map[B](f: A => B): Trampoline[B] =
  flatMap(a => Done(f(a)))
```

4.4 Stackless Scala

The `zipIndex` method from before can now run without a StackOverflowError, for any size of input list, by using the trampolined `State` monad.

`Trampoline` as presented here is a general solution to stack frame elimination in Scala. We can now rewrite any program to use no stack space whatsoever. Consider a program of this form:

```
val x = f()
val y = g(x)
h(y)
```

It can very easily be rewritten this way:

```
for {
  x <- f()
  y <- g(x)
  z <- h(y)
} yield z
```

Given the following implicit definition:

```
implicit def step[A](a: => A): Trampoline[A] =
  More(() => Done(a))
```

The only kind of call where the `step` transformation is inappropriate is (not coincidentally) in a self-recursive call. These are easy to detect, and in those cases we could call the `More` constructor explicitly, as in this recursive function to find the n^{th} Fibonacci number:

```
def fib(n: Int): Trampoline[Int] =
  if (n <= 1) Done(n) else for {
    x <- More(() => fib(n-1))
    y <- More(() => fib(n-2))
  } yield x + y
```

Since this transformation is completely mechanical, we can imagine that one could write a compiler plugin or otherwise augment the Scala compiler to transform all programs this way.

5. Cooperative multitasking

We've seen how it's possible to compose `Trampoline` computations sequentially using `flatMap`. But it's also possible to compose them in parallel by interleaving computations:

```
def zip[B](b: Trampoline[B]): Trampoline[(A,B)] =
  (this.resume, b.resume) match {
    case (Right(a), Right(b)) =>
      Done((a, b))
    case (Left(a), Left(b)) =>
      More(() => a() zip b())
    case (Left(a), Right(b)) =>
      More(() => a() zip Done(b))
    case (Right(a), Left(b)) =>
      More(() => Done(a) zip b())
  }
```

To see this in action, we can introduce side-effects to print to the console:

```
val hello: Trampoline[Unit] = for {
  _ <- print("Hello, ")
  _ <- println("World!")
} yield ()
```

And we can see what happens if we interleave this computation with itself:

```
scala> (hello zip hello).runT
Hello, Hello, World!
World!
```

While this is parallelism with a single thread, it's easy to imagine distributing work among many threads. For a set of trampolines under execution, any trampoline that is not `Done` could be resumed by any available thread.

It turns out that this generalizes to a model of full symmetric coroutines. We will discuss that generalization in the next section.

6. Free Monads: A Generalization of Trampoline

We can think of `Trampoline` as a coroutine that may be suspended in a `Function0` and later resumed. But this is not the only type constructor that we could use for the suspension. If we abstract over the type constructor, we get the following data type:

```
sealed trait Free[S[+_],+A] {
  private case class FlatMap[S[+_],A,+B](
    a: Free[S,A],
    f: A => Free[S,B]) extends Free[S,B]
}

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]
```

Now, `Trampoline` can be defined simply as:

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

As evidenced by the `Done` and `FlatMap` data constructors above, `Free[S,A]` is a monad for *any covariant functor* `S`. Seen categorically, it is precisely the free monad generated by that functor [4].

When we say that `S` must be a functor, we mean more precisely that there must exist an instance of `Functor[S]`[3]:

```
trait Functor[F[_]] {
  def map[A,B](m: F[A])(f: A => B): F[B]
}
```

For `Function0` this is straightforward:

```
implicit val f0Functor =
  new Functor[Function0] {
    def map[A,B](a: () => A)(f: A => B) =
      () => f(a())
  }
```

[3] This trait is taken from version 7 of the *Scalaz* library [2]

6.1 Functions defined on all free monads

To make concrete the claim that all free monads can benefit from the work we already did, we can generalize the methods previously defined for `Trampoline`. For example, here is the general form of `resume`:

```
final def resume(implicit S: Functor[S]):
Either[S[Free[S, A]], A] =
  this match {
    case Done(a) => Right(a)
    case More(k) => Left(k)
    case a FlatMap f => a match {
      case Done(a) => f(a).resume
      case More(k) => Left(S.map(k)(_ flatMap f))
      case b FlatMap g => b.flatMap((x: Any) =>
        g(x) flatMap f).resume
    }
  }
```

Note that this definition is essentially the same as the one for `Trampoline`. The only differences are the type signature, the additional implicit `Functor` argument, and the fact that we have replaced explicit construction of `Function0` with calls to `map` for our functor. This is also true for the generalized implementations of `zip`, `map`, and `flatMap` [10]. Here is zip:

```
def zip[B](b: Free[S,B])(
  implicit S: Functor[S]): Free[S, (A,B)] =
  (resume, b.resume) match {
    case (Left(a), Left(b)) =>
      More(S.map(a)(x =>
      More(S.map(b)(y => x zip y))))
    case (Left(a), Right(b)) =>
      More(S.map(a)(x => x zip Done(b)))
    case (Right(a), Left(b)) =>
      More(S.map(b)(y => Done(a) zip y))
    case (Right(a), Right(b)) =>
      Done((a, b))
  }
```

6.2 Common data types as free monads

Informally, we can view `Free[S,A]` as the type of any computation that may branch by some functor `S` and terminate with some data of type `A` at the leaves. To gain an intuition for this, consider the ordinary binary decision tree. It is a free monad whose functor "splits" the computation in two at every branch:

```
type Pair[+A] = (A,A)
type BinTree[+A] = Free[Pair, A]
```

The `Done` case for `BinTree[A]` is a leaf that holds a value of type `A`, while the `More` case is a branch holding two values of type `BinTree[A]`. Our free monad (by the `FlatMap` case) lets us take every leaf in a tree, apply a tree-producing function to it, and graft the resulting tree in place of that leaf. And because this is an instance of `Free`, the work we already did on `Trampoline` lets us do so in constant time and stack space.

To get a tree with any number of possible branches at each node instead of just two, we would branch by the `List` functor:

```
type Tree[+A] = Free[List, A]
```

Indeed, `List` itself can be expressed as an application of `Free`[4]:

```
type List[A] =
  Free[({type λ[+α] = (A, α)})#λ, Unit]
```

In this view, a `List[A]` is a coroutine that produces a value of type `A` each time it resumes, or `Unit` if it's the empty list. The action of the free monad here is not the action of the "list monad" as such (whose monadic bind would substitute a new list for every element in the list), but a monad that lets us append one list to another[5]. Again, since it is an application of `Free`, we can perform that operation in constant time and space.

The List type given here is invariant in its type argument, but could be made covariant. This is left as an exercise.

6.3 A free State monad

While the examples of free monads presented here are very simple, the suspension functor for a free monad could be of arbitrary complexity. It could produce outputs and expect inputs in any conceivable combination. We could, for example, model `State` as a little language, where we can get and set the state:

```
sealed trait StateF[S,+A]
case class Get[S,A](f: S => A)
  extends StateF[S,A]
case class Put[S,A](s: S, a: A)
  extends StateF[S,A]
```

In the `Get` constructor, `f` is a function that expects the current value of the state. In the `Put` constructor, `s` is the new state, and `a` is the rest of the computation (that may conceivably do something with that state).

We will need evidence that our data type is in fact a functor:

```
implicit def statefFun[S] =
  new Functor[({type λ[+α] = StateF[S,α]})#λ] {
    def map[A,B](m: StateF[S, A])(f: A => B) =
      m match {
        case Get(g) => Get((s:S) => f(g(s)))
        case Put(s, a) => Put(s, f(a))
      }
    }
```

We can then encode a `State`-like monad directly as the free monad generated by our `StateF` functor:

```
type FreeState[S,+A] =
  Free[({type λ[+α] = StateF[S,α]})#λ, A]
```

The `pureState` combinator from section 1 comes "for free" with the `Done` constructor of our free monad[6]:

```
def pureState[S,A](a: A): FreeState[S,A] =
  Done(a)
```

And the other two, for getting and setting the state, are easy to define:

```
def getState[S]: FreeState[S,S] =
  More(Get(s => Done(s)))

def setState[S](s: S): FreeState[S,Unit] =
  More(Put(s, Done(())))
```

To run the state action given an initial state is a simple loop:

```
def evalS[S,A](s: S, t: FreeState[S,A]): A =
  t.resume match {
    case Left(Get(f)) => evalS(s, f(s))
    case Left(Put(n, a)) => evalS(n, a)
    case Right(a) => a
  }
```

We can now write pure functions that keep some state in this monad. For example, here is `zipIndex` from section 1, this time using our `FreeState` monad:

```
def zipIndex[A](as: List[A]): List[(Int,A)] =
  evalS(0, as.foldLeft(
    pureState[Int, List[(Int,A)]](List())) {
      (acc, a) => for {
        xs <- acc
        n <- getState
        _ <- setState(n + 1)
      } yield (n,a)::xs}).reverse
```

The implementation is almost identical, and this runs in constant stack without having to go through `Trampoline`. The conclusion is that it's not always necessary or desirable to trampoline an existing data structure. We can invent new free monads of our own design *à la carte* (see 7.4) and reap the same benefits.

7. Existing work

Nothing presented here is particularly original, although the pieces are taken from various sources and have not been put together in this way before to my knowledge, particularly not in Scala.

7.1 Trampolines

Using trampolined functions to implement tail calls in stack-oriented languages is a well-known technique, implemented in many languages as libraries or in compilers [6].

The standard Scala library [1] (as of this writing, at version 2.9.1) includes a package named `TailCalls` that provides a data structure `TailRec`, which is a limited equivalent of the `Trampoline` in this paper. Notably missing are monadic functions such as `map` and `flatMap`. This paper has

[4] This definition uses a "type lambda", which is defining a type constructor inline and simultaneously using it. An anonymous structural type is declared in parentheses and a single type constructor λ is defined as its member. The λ member is then projected out inline using the $\#$ syntax.

[5] The free structure being generated by the λ functor here is technically the free monoid generated by A.

[6] As of this writing the Scala compiler is unable to infer the type arguments to the $More$ and $Done$ constructors in these definitions. Type annotations are omitted here for clarity.

presented how they might be implemented if added and why they would be implemented that way.

7.2 Operational monads

This paper's implementation of free monads has monadic bind reified on the heap as a data constructor rather than a method call on the stack. This allowed us to manipulate binds as data and perform re-association. We can refer to this technique as using an *operational monad*, based on work by Apfelmus [3]. He discusses monads that represent programs with an explicit reified stack but not free monads or trampolines as presented here.

Our `FreeState` monad is also an operational monad, explicitly reifying the two operations `Get` and `Put`.

7.3 Free monads and coroutines

The idea of generalizing trampolines came from *Coroutine Pipelines* by Blažević in the October 2011 issue of The Monad Reader [5]. Although he makes no explicit reference to free structures, it's clear that his `Coroutine` type is a monad transformer incarnation of `Free`. In the present implementation in Scala, it's necessary to forego the parameterization on an additional monad, in order to preserve tail call elimination. Instead of being written as a monad transformer itself, `Free` could be *transformed by* a monad transformer for the same effect. For example `Coroutine[A,({type λ[α]=State[S,α]})#λ,B]` becomes `StateT[S,({type λ[α]=Free[A,α]})#λ,B]`, given:

```
case class StateT[S,M[_],A](run: S => M[(A,S)])
```

Blažević's discussion goes into much more detail about the kinds of cooperating coroutines that are expressible as free monads, with producers, consumers, and transducers between them.

7.4 The expression problem

The idea of the `FreeState` monad, modeling state transitions using the coproduct of `Get` and `Put` is taken from Wouter Swiestra's 2008 paper *Data types à la carte* [11]. Swiestra uses the free monad to construct ad-hoc recursive data types from the coproduct of arbitrary functors, and Haskell type classes to handle the dispatch on different cases of the coproduct.

7.5 Codensity

The act of associating all monadic binds to the right can be understood as an application of *codensity* [4]. This key insight came from a discussion with Ed Kmett on his work with an I/O data structure expressed as a free monad [7].

Janis Voigtländer discusses codensity of free monads in his paper *Asymptotic Improvement of Computations over Free Monads* [12] although he does not explicitly name the concept "codensity". The goal of his paper is not to conserve stack space as such, but to improve performance. Voigtländer explains how descending into left-associated binds in a free monad has a quadratic overhead, and he proceeds to eliminate this overhead by using a codensity monad.

7.6 Iteratees

Safe and modular incremental I/O has long been elusive in purely functional languages such as Haskell. Oleg Kiselyov and John Lato discuss the idea of an *iteratee*, a pure automaton that consumes input, produces effects, and may maintain some internal state [9] [8].

Iteratees are not exactly free monads, but they are the composition of a free monad with another functor, so they can be expressed as an application of `Free`. The `IterV` type in Lato's article can be very roughly expressed as follows:

```
type IterV[I,O] =
  Free[({type λ[α] = I => α})#λ, (I,O)]
```

Because `IterV` tracks the remainder of the input (much like a parser), it is not technically free. But the article also discusses an "internal iterator" they call *enumeratee*, which converts one stream of inputs to a stream of outputs, and this type can more easily be expressed as a free monad:

```
type Enumeratee[I,O,A] = Free[({
  type λ[α] = Either[I => α, (O, α)]})#λ, A]
```

This kind of coroutine is a stream transducer that may either request an input of type `I` or produce an output of type `O` each time it resumes, and will terminate with a value of type `A`.

8. Conclusions and further work

We have seen how we can use trampolines to make recursive calls on the heap instead of the stack in Scala. The technique is straightforward but there are some details that we must implement a certain way, given the abilities and limitations of the Scala language and the JVM, or else the solution will not work.

We saw how trampolines relate to the general idea of free monads over a functor and how all of our concerns about trampolines, as well as all the benefits, readily transfer to any such monad.

There is more research to be done on putting these ideas to work in the Scala compiler—possibly as a plugin—to enable stackless programming without any additional effort from the programmer.

Free monads have many more applications than we have imagined here. With increasing pressure in industry towards pure functional programming comes the need for a programming model that enables composition of modular computations that are efficient in terms of time and memory. Promising work is being done on iteratees and monadic streams, but there is not yet a very clean API for these in Scala. Perhaps a library based on free monads could make all the difference.

References

[1] The Scala Standard Library API documentation, available at `http://www.scala-lang.org/api/current`

[2] The *Scalaz* library source code, available at `http://scalaz.org`

[3] H. Apfelmus, *The Operational Monad Tutorial*, in The Monad Reader **15**, January 2010. pp. 37-56, available at `http://themonadreader.files.wordpress.com/2010/01/issue15.pdf`

[4] S. Awodey, *Category Theory*, Second Edition, Oxford University Press, New York, 2010.

[5] M. Blažević, *Coroutine Pipelines*, in The Monad Reader **19**, October 2011. pp. 29-50, available at `http://themonadreader.files.wordpress.com/2011/10/issue19.pdf`

[6] Steven E. Ganz and Daniel P. Friedman and Mitchell Wand, *Trampolined Style*, in International Conference on Functional Programming, ACM Press, 1999, pp. 18–27.

[7] E. A. Kmett, *Free Monads for Less*, in The Comonad Reader, June 2011, available at `http://comonad.com/reader/2011/free-monads-for-less`

[8] J. W. Lato, *Iteratee: Teaching an Old Fold New Tricks*, in The Monad Reader **16**, May 2010. pp. 19–36, available at `http://themonadreader.files.wordpress.com/2010/05/issue16.pdf`

[9] O. Kiselyov, *Incremental multi-level input processing and collection enumeration*, available at `http://okmij.org/ftp/Streams.html`

[10] Rúnar Ó. Bjarnason, source code for examples in this paper, available at `https://github.com/runarorama/Days2012`

[11] W. Swiestra, *Data types à la carte*, in Journal of Functional Programming, Cambridge University Press, 2008.

[12] J. Voigtländer and Technische Universität Dresden, *Asymptotic Improvement of Computations over Free Monads*, in proceedings, Mathematics of Program Construction, 2008, pp. 388–403.