Stack Safety for Free

Phil Freeman

August 3, 2015

Abstract

Free monads are a useful tool for abstraction, separating specification from interpretation. However, a naive free monad implementation can lead to stack overflow depending on the evaluation model of the host language. This paper develops a stack-safe free monad transformer in *PureScript*, an eager language compiling to Javascript, and demonstrates certain applications - a safe implementation of coroutines, and a generic mechanism for building stack-safe control operators.

Introduction

Techniques from pure functional programming languages such as Haskell have been making their way into mainstream programming, slowly but surely, in the form of projects like ("Scalaz Project GitHub Repository") in Scala. Abstractions such as monoids, functors, applicative functors, monads, arrows, etc. afford a level of expressiveness which can give great productivity gains, and improved guarantees of program correctness.

However, naive implementations of these abstractions can lead to poor performance, depending on the evaluation order of the host language. In particular, deeply recursive code can lead to *stack overflow*.

One example of a desirable abstraction is the *free monad* for a functor `f`. Free monads allow us to separate the specification of a monad (by specifying the base functor `f`), from its implementation. However, a naive translation of the standard Haskell definition

```
newtype Free f a = Free { unFree :: Either a (f (Free f a)) }

runFree :: (forall a. f a -> m a) -> Free f a -> m a
runFree phi = either return ((>>= (runFree phi)) . phi) . unFree
```

to languages like Scala or C#, can lead to stack overflows, both during construction of a computation of type `Free f a`, and during interpretation.

The free monad can be generalized to a monad transformer, where a monad `m` is used to track effects at each step of the computation:

```
newtype FreeT f m a = FreeT { unFreeT :: m (Either a (f (Free f a))) }
```

Current attempts to generalize stack-safe implementations of the free monad to the free monad transformer `FreeT` have met with difficulty. In this paper, we'll construct a stack-safe implementation of the free monad transformer, with a restriction on the class of monads which can be transformed.

We will work in the PureScript programming language, a pure functional language inspired by Haskell which compiles to Javascript. PureScript features an expressive type system, with support for type classes and higher-kinded types, but unlike Haskell, evaluation in PureScript is *eager*, so PureScript provides a good environment in which to demonstrate these ideas. The same techniques should be applicable to other languages such as Scala, however.

Free Monads in Haskell

Free monads are a useful tool in Haskell (and Scala, and PureScript, and other languages) when you want to separate the specification of a `Monad` from its interpretation. We use a `Functor` to describe the terms we want to use, and construct a `Monad` for free, which can be used to combine those terms.

The definition of the free monad in Haskell is given as a recursive data type:

```
newtype Free f a = Free { unFree :: Either a (f (Free f a)) }
```

A computation of type `Free f a` is either complete, returning value of type `a`, or a step, where the operations available to the programmer at each step are described by the functor `f`.

`Free` is easily made into a `Monad` whenever `f` is a `Functor`:

```
instance (Functor f) => Functor (Free f) where
  fmap f = Free . either f (fmap (fmap f)) . unFree

instance (Functor f) => Monad (Free f) where
  return = Free . Left
  m >>= f = either f (Free . Right . fmap (>>= f)) (unFree m)
```

If `Free f` represents syntax trees for a language with operations described by `f`, then the monadic bind function implements substitution at the leaves of the tree, substituting new computations depending on the result of the first computation.

For example, we might choose the following functor as our base functor:

```
data CounterF a
  = Increment a
  | Read (Int -> a)
  | Reset a

instance Functor CounterF where
  fmap f (Increment a) = Increment (f a)
  fmap f (Read k) = Read (f <<< k)
  fmap f (Reset a) = Reset (f a)
```

This functor describes three possible operations on a simulated counter: `Increment`, which increments the counter by one, `Read`, which provides the current value of the counter, and `Reset` which resets the counter to zero.

The values of type `a` in the definition of `CounterF` represent "what to do next". When `Free` is applied to `CounterF`, the type `a` will be instantiated to `Free f a`, the next step of the computation.

We can define constructors for our three operations, and a synonym for our free monad:

```
type Counter = Free CounterF

liftFree :: (Functor f) => f a -> Free f a
liftFree fa = Free . Right . fmap return

increment :: Counter ()
increment = liftFree (Increment unit)

read :: Counter Int
read = liftFree (Read id)

reset :: Counter ()
reset = liftFree (Reset unit)
```

Given these constructors, and the `Monad` instance above, we can construct computations in our new `Counter` monad:

```
readAndReset :: Counter Int
readAndReset = do
  current <- read
  reset
  return current
```

Running a computation in the `Counter` monad requires that we give an interpretation for the operations described by the functor `CounterF`. We must choose a monad `m` in which to interpret our computation, and then provide a natural transformation from `CounterF` to `m`.

One possible implementation might use a `State` monad to keep track of the counter state:

```
runCounter :: Counter a -> State Int a
runCounter = runFree interpret
  where
  interpret :: CounterF a -> State Int a
  interpret (Increment a) = modify (1 +) >> return a
  interpret (Read k) = fmap k get
  interpret (Reset a) = put 0 >> return a
```

Other implementations might use the `IO` monad to update a counter on a remote server, or add log messages to the implementation above, using the `StateT Int IO` monad transformer stack. This is the power of working with free monads - we have completely separated the meaning of our computations from the syntax that describes them.

Free Monad Transformers

The free monad construction given above can be generalized to a free monad transformer, `FreeT`:

```
newtype FreeT f m a = FreeT { unFreeT :: m (Either a (f (FreeT f m a))) }
```

The free monad transformer allows us to interleave effects from the base monad `m` at each step of the computation.

The `Functor` and `Monad` instances for `FreeT` look similar to the instances for `Free`. In addition, we now also have an instance for `MonadTrans`, the type class of monad transformers:

```
instance (Functor f, Functor m) => Functor (FreeT f m) where
  fmap f = FreeT . fmap (either f (fmap (fmap f))) . unFreeT

instance (Functor f, Monad m) => Monad (Co f m) where
  return a = FreeT (return (Left a))
  m >>= f = unFreeT m >>= either f (FreeT . return . Right . fmap (>>= f))

instance MonadTrans (FreeT f) where
  lift = FreeT . fmap Left
```

The `Counter` operations given above can be lifted to work in the free monad transformer:

```
type CounterT = FreeT CounterF

liftFreeT :: (Functor f, Monad m) => f a -> FreeT f m a
liftFreeT = FreeT . return . Right . fmap return

increment :: (Monad m) => CounterT m ()
increment = liftFreeT (Increment unit)

read :: (Monad m) => CounterT m Int
read = liftFreeT (Read id)

reset :: (Monad m) => CounterT m ()
reset = liftFreeT (Reset unit)
```

We can now modify our original computation to include console logging, for example:

```
readAndReset :: CounterT IO Int
readAndReset = do
  current <- read
  lift $ putStrLn $ "Current value is " ++ show current"
  reset
  return current
```

Free Monads in Scala and PureScript

Implementing free monads in languages like Scala and PureScript is tricky. A translation of the Haskell implementation is possible in a language supporting higher-kinded types (and ideally type classes). A naive translation works well for small computations such as the one above. However, the `runFree` function is not tail recursive, and so interpreting large computations often results in *stack overflow*. Techniques such as monadic recursion become unusable. In an eager language, it is not necessarily possible to even build a large computation in a free monad, since each monadic bind has to traverse the tree to its leaves.

Fortunately, a solution to this problem has been known to the Scala community for some time. Bjarnason describes how to defer monadic binds in the free monad, by capturing binds as a data structure. `runFree` can then be implemented as a tail recursive function, interpreting this structure of deferred monadic binds, giving a free monad implementation which supports deep recursion.

However, there is an important restriction: `runFree` cannot be implemented safely for an arbitrary target monad `m`. In the Scalaz implementation, at the time of writing, the only stack-safe implementations correspond to the `Identity` monad and the `State` monad, or monads which are themselves stack-safe due to some implementation detail (for example, by trampolining).

This technique is also used to implement free monads in PureScript, in the `purescript-free` library, where the data constructor capturing the bind is named `Gosub`:

```
newtype GosubF f a b = GosubF (Unit -> Free f b) (b -> Free f a)

data Free f a
  = Free (Either a (f (Free f a)))
  | Gosub (Exists (GosubF f a))
```

Here, we add the `Gosub` constructor which directly captures the arguments to a monadic bind, existentially hiding the return type `b` of the intermediate computation.

By translating the implementation in Bjarnason, `purescript-free` builds a stack-safe free monad implementation for PureScript, which has been used to construct several useful libraries.

However, in Bjarnason, when discussing the extension to a monad transformer, it is correctly observed that:

> 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.

That is, it's not clear how to extend the `Gosub` trick to the free monad *transformer* if we want to be able to transform an arbitrary monad.

Additionally, the approach of putting using another monad transformer to transform `Free` is strictly less expressive than using the free monad transformer, since we would be unable to transform monads which did not have an equivalent transformer, such as `IO`.

Tail Recursive Monads

Our solution is to reduce the candidates for the target monad `m` from an arbitrary monad, to the class of so-called tail-recursive monads. To motivate this abstraction, let's consider tail call elimination for pure functions.

The PureScript compiler performs tail-call elimination for self-recursive functions, so that a function like `pow` below, which computes integer powers by recursion, gets compiled into an efficient `while` loop in the generated Javascript.

```
pow :: Int -> Int -> Int
pow n p = go (Tuple 1 p)
  where
  go (Tuple acc 0) = acc
  go (Tuple acc p) = go (Tuple (acc * n) (p - 1))
```

However, we do not get the same benefit when using monadic recursion. Suppose we wanted to use the `Writer` monad to collect the result in the `Product` monoid:

```
powWriter :: Int -> Int -> Writer Product Unit
powWriter n = go
  where
  go 0 = return unit
  go m = do
    tell n
    go (m - 1)
```

This time, we see a stack overflow at runtime for large inputs to the `powWriter` function, since the function is no longer tail-recursive: the tail call is now inside the call to the `Writer` monad's bind function.

We can refactor the original `pow` function to isolate the recursive function call:

```
tailRec :: forall a b. (a -> Either a b) -> a -> b

pow :: Int -> Int -> Int
pow n p = tailRec go (Tuple 1 p)
  where
  go :: Tuple Int Int -> Either (Tuple Int Int) Number
  go (Tuple acc 0) = Right acc
  go (Tuple acc p) = Left (Tuple (acc * n) (p - 1))
```

Here, the `tailRec` function expresses a generic tail-recursive function, where in the body of the loop, instead of calling the `go` function recursively, we return a value using the `Left` constructor. To break from the loop, we use the `Right` constructor.

`tailRec` itself is implemented using a tail-recursive helper function, which makes this approach very similar to the trampoline approach:

```
tailRec :: forall a b. (a -> Either a b) -> a -> b
tailRec f a = go (f a)
  where
  go (Left a) = go (f a)
  go (Right b) = b
```

However, type of `tailRec` can be generalized to several monads using the following type class, which is defined in the `purescript-tailrec` library:

```
class (Monad m) <= MonadRec m where
  tailRecM :: forall a b. (a -> m (Either a b)) -> a -> m b
```

`tailRecM` can actually be implemented for *any* monad `m`, by modifying the `tailRec` function slightly as follows:

```
tailRecM :: forall a b. (a -> m (Either a b)) -> a -> m b
tailRecM f a = f a >>= go
  where
  go (Left a) = f a >>= go
  go (Right b) = return b
```

However, this would not necessarily be a valid implementation of the `MonadRec` class, because `MonadRec` comes with an additional law:

> A valid implementation of `MonadRec` must guarantee that the stack usage of `tailRecM f` is at most a constant multiple of the stack usage of `f` itself.

This unusual law is not necessarily provable for a given monad using the usual substitution techniques of equational reasoning, but might require some slightly more subtle reasoning.

We can write some helper functions for instances of the `MonadRec` class, such as `forever`, which iterates a monadic action forever, as a variant of the function with the same name from Haskell's standard library:

```
forever :: forall m a b. (MonadRec m) => m a -> m b
```

`MonadRec` becomes useful, because it has a surprisingly large number of valid instances: `tailRec` itself gives a valid implementation for the `Identity` monad, and there are valid instances for PureScript's `Eff` and `Aff` monads, which are synchronous and asynchronous effect monads similar in some respects to Haskell's `IO`.

There are also valid `MonadRec` instances for some standard monad transformers: `ExceptT`, `StateT`, `WriterT`, `RWST`, which gives a useful generalization of tail recursion to monadic contexts. We can rewrite `powWriter` as the following safe variant, for example:

```
powWriter :: Int -> Int -> Writer Product Unit
powWriter n = tailRecM go
  where
  go :: Int -> Writer Product (Either Int Unit)
  go 0 = return (Right unit)
  go m = do
    tell n
    return (Left (m - 1))
```

Interpreting Free Monads Safely

Tail recursive monads provide a safe alternative to the `runFree` function, which previously was restricted to a handful of monads.

Instead of interpreting a free monad in an arbitrary monad `m`, we modify `runFree` to target a monad with a valid `MonadRec` instance:

```
runFreeM :: forall f m a. (Functor f, MonadRec m) =>
                          (f (Free f a) -> m (Free f a)) ->
                          Free f a ->
                          m a
```

Here, the `MonadRec` instance is used to define a tail-recursive function which unrolls the data structure of monadic binds.

This is enough to allow us to use monadic recursion with `Free` in PureScript, and then interpret the resulting computation in any monad with a valid `MonadRec` instance.

We have enlarged our space of valid target monads to a collection closed under several standard monad transformers.

Stack-Safe Free Monad Transformers

The class of tail recursive monads also allow us to define a safe free monad transformer in PureScript.

A naive implementation might look like

```
newtype FreeT f m a = FreeT (m (Either a (f (FreeT f m a))))
```

We can apply the same `Gosub` trick from the `Free` monad implementation and apply it to our proposed `FreeT`:

```haskell
data GosubF f m b a = GosubF (Unit -> FreeT f m a) (a -> FreeT f m b)

data FreeT f m a
  = FreeT (Unit -> m (Either a (f (FreeT f m a))))
  | Gosub (Exists (GosubF f m a))
```

We also thunk the computation under the `Free` constructor, which is necessary to avoid stack overflow during construction.

The instances for `Functor` and `Monad` generalize nicely from `Free` to `FreeT`, composing binds by nesting `Gosub` constructors. This allows us to build computations safely using monadic recursion. The difficult problem is how to *run* a computation once it has been built.

Instead of allowing interpretation in any monad, we only support interpretation in a monad with a valid `MonadRec` instance. We can reduce the process of interpreting the computation to a tail recursive function in that monad:

```haskell
runFreeT :: forall f m a. (Functor f, MonadRec m) =>
                          (forall a. f a -> m a) ->
                          FreeT f m a ->
                          m a
```

Stack Safety for Free

Capretta, Altenkirch, and Uustalu defines the *free completely-iterative monad transformer*. In Haskell, it might be defined as:

```haskell
newtype IterT m a = IterT { runIterT :: m (Either (IterT f m a) a) }
```

where the fixed point is assumed to be the greatest fixed point.

This looks a lot like our definition of the free monad transformer for the `Identity` functor, but there, the least fixed point was implied. However, our encoding of the free monad transformer in PureScript allows for infinite values, so we can consider our `FreeT Identity m` as embedding in `IterT m`. In PureScript, we will take `FreeT Identity` as our encoding of the free completely-iterative monad transformer:

```haskell
type IterT = FreeT Identity
```

`IterT m` is stack-safe for any monad `m`, thanks to the `Gosub` trick. Also, `IterT m` can be interpreted in `m` using `runFreeT return`, and this interpretation is stack-safe whenever `m` is a tail-recursive monad. Since `IterT` is a monad transformer, we can interpret any computation in `m` inside `IterT m`.

This means that for any tail-recursive monad `m`, we can work instead in `IterT m`, including deeply nested left and right associated binds, without worrying about stack overflow. When our computation is complete, we can use `runFreeT` to move back to `m`.

For example, this computation quickly terminates with a stack overflow:

```
main = go 100000
  where
  go n | n <= 0 = return unit
  go n = do
   print n
   go (n - 2)
   go (n - 1)
```

but can be made productive, simply by lifting computations into `IterT`:

```
main = runFreeT return $ go 100000
  where
  go n | n <= 0 = return unit
  go n = do
   lift (print n)
   go (n - 2)
   go (n - 1)
```

Note that this would not be possible by using a trampolined free monad, since the `Eff` monad has no equivalent monad transformer.

Application: Coroutines

Free monad transformers can be used to construct models of *coroutines*, by using the base functor to specify the operations which can take place when a coroutine suspends.

For example, we can define a base functor `Emit` which supports a single operation at suspension - emitting a single output value. In PureScript, it would look like this:

```
data Emit o a = Emit o a

instance functorEmit :: Functor (Emit o) where
  map f (Emit o a) = Emit o (f a)
```

We can define a type `Producer` of coroutines which produce values and perform console IO at suspension:

```
type Producer o = FreeT (Emit o) (Eff (console :: CONSOLE))

emit :: o -> Producer o Unit
emit o = liftFreeT (Emit o unit)

producer :: Producer String Unit
producer = forever do
  lift (log "Emitting a value...")
  emit "Hello World"
```

We can vary the underlying `Functor` to construct coroutines which produce values, consume values, fork child coroutines, join coroutines, and combinations of these. This is described in Blažević, where free monad *transformers* are used to build a library of composable coroutines and combinators which support effects in some base monad.

Given a stack-safe implementation of the free monad transformer, it becomes simple to translate the coroutines defined in Blažević into PureScript. We can define a functor for awaiting values, and a coroutine type `Consumer`:

```
data Await i a = Await (i -> a)

instance functorAwait :: Functor (Await i) where
  map f (Await k) = Await (f <<< k)

type Consumer i = FreeT (Await i) (Eff (console :: CONSOLE))

await :: forall i. Consumer i i
await o = liftFreeT (Await id)
```

Here is an example of a `Consumer` which repeatedly awaits a new value before logging it to the console:

```
consumer :: forall a. (Show a) => Consumer a Unit
consumer = forever do
  s <- await
  lift (print s)
```

The use of the safe `FreeT` implementation, and `MonadRec` make these coroutines stack-safe. They can be connected and run using a constant amount of stack:

```
main = runFreeT id (producer $$ consumer)
```

`$$` is an operator defined in the `purescript-coroutines` library, which supports a handful of combinators for connecting producers, consumers and transformers, as well as more powerful, generic coroutine machinery taken from Blažević. Running this example will generate an infinite stream of the string `"Hello World"` printed to the console, interleaved with the debug message `"Emitting a value..."`.

In a pure functional language like PureScript, targeting a single-threaded language like Javascript, coroutines built using `FreeT` might provide a natural way to implement cooperative multithreading, or to interact with a runtime like NodeJS for performing tasks like non-blocking file and network IO.

Application: Lifting Control Operators

The fact that `IterT m` is stack-safe for any monad `m` provides a way to turn implementations of *control operators* with poor stack usage, into implementations with good stack usage for free.

By a control operator, we are referring to functions such as `mapM_`, `foldM`, `replicateM` and `iterateM`, which work over an arbitrary monad.

Consider, for example, the following definition of `replicateM_`, which replicates a monadic action some number of times, ignoring its results:

```
replicateM_ :: forall m a. (Monad m) => Int -> m a -> m Unit
replicateM_ 0 _ = return Nil
replicateM_ n m = do
  _ <- m
  replicateM (n - 1) m
```

This function is not stack-safe for large inputs. There is a simple, safe implementation of `replicateM` where the `Monad` constraint is strengthened to `MonadRec`, but for the purposes of demonstration, let's see how we can *derive* a safe `replicateM` instead, using `IterT`.

It is as simple as lifting our monadic action from `m` to `IterT m` before the call to `replicateM`, and lowering it down using `runFreeT return` afterwards:

```
safeReplicateM_ :: forall m a. (MonadRec m) => Int -> m a -> m Unit
safeReplicateM_ n m = runFree return (replicateM_ n (lift m))
```

We can even capture this general technique as follows. The `Operator` type class captures those functions which work on arbitrary monads, i.e. control operators:

```
type MMorph f g = forall a. f a -> g a

class Operator o where
  mapO :: forall m n. MMorph m n -> MMorph n m -> o m -> o n
```

Here, MMorph represents a monad morphism. mapO is given a pair of monad morphisms representing an embedding-retraction pair, and is responsible for changing the monad being operated on accordingly.

In practice, our two monads will be m and IterT m for some tail recursive monad m:

```
safely :: forall o m a. (Operator o, MonadRec m) =>
                        (forall t. (Monad t) => o t) ->
                        o m
safely o = mapO runProcess lift o
```

safely allows us to write a control operator for any Monad, trading the generality of a Monad constraint for the ability to be able to write code which is not necessarily stack-safe, and returns an equivalent combinator which works with any MonadRec, safely.

Given this combinator, we can reimplement our safe version of replicateM_ by defining a wrapper type and an instance of Operator:

```
newtype Replicator m = Replicator (forall a. Int -> m a -> m Unit)

instance replicator :: Operator Replicator where
  mapO to fro (Replicator r) = Replicator \n m -> to (r n (fro m))

runReplicator :: forall m a. Replicator m -> Int -> m a -> m Unit
runReplicator (Replicator r) = r

safeReplicateM_ :: forall m a. (MonadRec m) => Int -> m a -> m Unit
safeReplicateM_ = runReplicator (safely (Replicator replicateM_))
```

We can use the safely combinator to derive safe versions of many other control operators automatically.

Conclusion and Further Reading

In Capretta, Altenkirch, and Uustalu, monads supporting a tailRecM operation are called *completely iterative*, although the conditions for validity of instances is different. There they are used to capture monads supporting the side-effect of *partiality* in a total language. Our instance for Identity would be considered unsafe, for example.

References

Bjarnason, Rúnar Óli. "Stackless Scala With Free Monads."

Blažević, Mario. "Coroutine Pipelines." *The Monad Reader*.

Capretta, Venanzio, Thorsten Altenkirch, and Tarmo Uustalu. "Partiality Is an Effect."

"Scalaz Project GitHub Repository." https://github.com/scalaz/scalaz.