Stack Safety for Free

Phil Freeman

August 8, 2015

Abstract

Free monads are a useful tool for abstraction, separating specification from interpretation. However, a naïve 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*, a strict Haskell-like language compiling to Javascript, and demonstrates certain applications - a safe implementation of coroutines, a safe list monad transformer, 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. 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, naïve 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. In this paper, we will develop a safe implementation of free monad and the equivalent *monad transformer* in [PureScript], a strict Haskell-like language which compiles to Javascript.

A naïve implementation of the free monad in PureScript might look like this:

```
newtype Free f a = Free (Either a (f (Free f a)))

resume :: forall f a. Free f a -> Either a (f (Free f a))
resume (Free a) = a
```

```
liftFree :: forall f a. (Functor f) => f a -> Free f a
liftFree = Free <<< Right <<< map return

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

The `Free f` type constructor is made into a `Monad` for any `Functor f`. However, this implementation quickly leads to the problem of *stack overflow*, 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. 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 and its monad transformer, by imposing 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.

Computing with Free Monads

Our free monad type constructor is parameterized by a `Functor` which describes the terms we want to use, and the `Monad` instance provides a way to combine those terms. A computation of type `Free f a` is either complete, returning value of type `a`, or is a *suspension*. The operations available at a suspension are described by the functor `f`.

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 functorCounterF :: Functor CounterF where
  map f (Increment a) = Increment (f a)
  map f (Read k) = Read (f <<< k)
  map 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.

When `Free` is applied to the functor `CounterF`, the type variable `a` will be instantiated to `Free f a`. `a` represents the continuation of the computation after the current step.

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

```
type Counter = Free CounterF

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

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

reset :: Counter Unit
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 PureScript's `Eff` monad of *extensible effects*, using a scoped reference variable (via the `ST` effect) to keep track of the counter state:

```
runCounter :: forall eff h a.
  STRef h Int ->
  Counter a ->
  Eff (st :: ST h | eff) a
runCounter ref = runFree go
  where
  go (Increment a) = do
    modifySTRef ref (1 +)
    return a
  go (Read k) = map k (readSTRef ref)
  go (Reset a) = do
    writeSTRef ref 0
    return a
```

Other implementations might use the asynchronous effect monad `Aff` to update a counter on a remote server, or add log messages to the implementation above, using the `Eff` monad to combine console and mutation effects. This is the power of working with free monads - we have completely separated the interpretation 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 (m (Either a (f (FreeT f m a))))

resumeT :: forall f m a. FreeT f m a -> m (Either a (f (FreeT f m a)))
resumeT (FreeT a) = a

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

runFreeT :: forall f m a.
  (Monad m) =>
  (f (FreeT f m a) -> m (FreeT f m a)) ->
  FreeT f m a -> m a
runFreeT phi = either return ((>>= (runFreeT phi)) <<< phi) <=< resumeT
```

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 functorFreeT :: (Functor f, Functor m) =>
    Functor (FreeT f m) where
  map f = FreeT <<< map (bimap f (map (map f))) <<< resumeT

instance applyFreeT :: (Functor f, Monad m) =>
    Apply (FreeT f m) where
  apply = ap

instance bindFreeT :: (Functor f, Monad m) =>
    Bind (FreeT f m) where
  bind (FreeT a) f = FreeT (a >>= go)
    where
    go (Left a) = resumeT (f a)
    go (Right fs) = return (Right (map (>>= f) fs))

instance applicativeFreeT :: (Functor f, Monad m) =>
    Applicative (FreeT f m) where
  pure = FreeT <<< pure <<< Left

instance monadFreeT :: (Functor f, Monad m) =>
    Monad (FreeT f m)

instance monadTransFreeT :: (Functor f) =>
    MonadTrans (FreeT f) where
  lift = FreeT <<< map Left
```

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

```
type CounterT = FreeT CounterF

incrementT :: forall m. (Monad m) => CounterT m Unit
incrementT = liftFreeT (Increment unit)

readT :: forall m. (Monad m) => CounterT m Int
readT = liftFreeT (Read id)

resetT :: forall m. (Monad m) => CounterT m Unit
resetT = liftFreeT (Reset unit)
```

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

```
readAndResetT :: forall eff.
  CounterT (Eff (console :: CONSOLE | eff)) Int
```

```
readAndResetT = do
  current <- readT
  lift $ log $ "The current value is " ++ show current
  resetT
  return current
```

Deferring Monadic Binds

The naïve implementation of `Free` given above works for small computations such as `readAndReset` example. 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. It is not necessarily possible to even *build* a large computation using `Free`, let alone evaluate it, 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. `Free` can then be interpreted using a tail recursive function, collapsing the structure of deferred monadic binds, giving a free monad implementation which supports deep recursion.

However, there is a restriction: `runFree` cannot be implemented safely for an arbitrary target monad `m`. Only monads which are stack-safe due to some implementation detail (for example, by trampolining) can be used as the target monad `m`.

Additionally, in [Bjarnason], when discussing the extension to a monad transformer, it is 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 is not clear how to extend the `Gosub` technique to the free monad *transformer* if we want to be able to transform an arbitrary monad.

The approach of 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 `Eff`.

A variant of this technique is used to implement free monads in PureScript, in the `purescript-free` library.

```
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, the `Gosub` data constructor has been added to the original definition of `Free`. `Gosub` captures the arguments to a monadic bind, existentially hiding the return type `b` of the intermediate computation.

To understand how `purescript-free` implements `runFree` for this modified structure, we need to understand the class of *tail-recursive monads*.

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.

Large inputs are not the only concern. Monadic combinators such as `forever`, which repeats a monadic action indefinitely, become useless, since they involve an arbitrarily large stack of monadic binds.

A tail-recursive function can make progress in two ways: it can return value immediately, or it can call itself (in tail position) recursively. This motivates the definition of the `tailRec` function, which expresses a generic tail-recursive function:

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

Instead of using explicit tail recursion, we can pass a helper function to `tailRec` which returns 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 approach of using a *trampoline*:

```
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
```

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

```
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))
```

Now we can be sure that our function runs using a constant amount of stack, as long as we know `tailRec` itself does not grow the stack. We no longer need to rely on the compiler's tail-call elimination optimization to take effect.

However, the 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
```

Here, both the helper function, and the return value have been wrapped using the monad `m`.

`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, and might require a slightly more subtle proof.

The `forever` combinator can be given a *safe* implementation for monads in the `MonadRec` class:

```
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.

There are also valid `MonadRec` instances for some standard monad transformers: `ExceptT`, `ReaderT`, `StateT`, `WriterT`, and `RWST`. For example, here is an instance for the state monad transformer:

```
instance monadRecStateT :: (MonadRec m) => MonadRec (StateT s m) where
  tailRecM f a = StateT \s -> tailRecM f' (Tuple a s)
    where
    f' (Tuple a s) = do
      Tuple m s1 <- runStateT (f a) s
      return case m of
        Left a -> Left (Tuple a s1)
        Right b -> Right (Tuple b s1)
```

Note that this instance defers to the `tailRecM` function for the base monad, and that `tailRecM` is used to express all recursion. Thus, the correctness of this instance is given by the correctness of the implied `MonadRec` instance for the base monad `m`.

Instances of `MonadRec` for monad transformers correspond to various variants on standard tail recursion:

- Tail recursion for `StateT` corresponds to adding an *accumulator parameter*.
- Tail recursion for `WriterT` can be seen as a generalization of *tail-recursion modulo cons*, since we build an additional monoidal result before the recursive call.
- Tail recursion for `ExceptT` corresponds to tail recursion in which we can terminate recursion early, returning some error.

`MonadRec` 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

`MonadRec` can be used to implement a safe version of the `runFree` function, using the extended free monad structure which defers monadic binds.

[Bjarnason] explains how to implement the `resume` function as a tail-recursive function. `resume` performs the first step of a free monad computation, unpacking deferred monadic binds where necessary. In `purescript-free`, its type signature is:

```
resume :: forall f a.
  (Functor f) =>
  Free f a ->
  Either (f (Free f a)) a
```

Given a safe implementation of `resume`, a stack-safe implementation of `runFree` becomes simple, using `MonadRec`:

```
runFree :: forall f m a.
  (Functor f, MonadRec m) =>
  (f (Free f a) -> m (Free f a)) ->
  Free f a ->
  m a
runFree phi = tailRecM \m ->
  case resume m of
    Left fs -> map Left (phi fs)
    Right a -> return (Right a)
```

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

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.

We can apply the `Gosub` technique to our naïve implementation of `FreeT`:

```
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. As with `Free`, the remaining problem is how to *run* a computation in some (tail-recursive) target monad `m`.

Bjarnason's `resume` function generalizes to the `FreeT` case, using `tailRecM` to express the (monadic) tail recursion:

```
resume :: forall f m a.
  (Functor f, MonadRec m) =>
  FreeT f m a ->
  m (Either a (f (FreeT f m a)))
resume = tailRecM go
  where
  go :: FreeT f m a -> m (Either (FreeT f m a) (Either a (f (FreeT f m a))))
  go (FreeT f) = map Right (f unit)
  go (Gosub e) = runExists (\(GosubF m f) ->
    case m unit of
      FreeT m -> do
        e <- m unit
        case e of
```

```
        Left a -> return (Left (f a))
        Right fc -> return (Right (Right (map (\h -> h >>= f) fc)))
    Gosub e1 -> runExists (\(GosubF m1 f1) ->
                    return (Left (bind (m1 unit) (\z -> f1 z >>= f)))) e1) e
```

Similarly, our `runFree` function generalizes using `MonadRec` to a safe implementation of `runFreeT`, allowing us to interpret `FreeT f m` whenever `m` itself is a tail-recursive monad:

```
runFreeT :: forall f m a.
  (Functor f, MonadRec m) =>
  (f (FreeT f m a) -> m (FreeT f m a)) ->
  FreeT f m a ->
  m a
runFreeT interp = tailRecM (go <=< resume)
  where
  go :: Either a (f (FreeT f m a)) -> m (Either (FreeT f m a) a)
  go (Left a) = return (Right a)
  go (Right fc) = do
    c <- interp fc
    return (Left c)
```

We have built a safe free monad transformer, with the restriction that we can only *interpret* the computations we build if the underlying monad is a tail-recursive monad.

Stack Safety for Free

We are free to choose any functor `f`, and we are able to build a stack-safe free monad transformer over `f`. In particular, we can consider the free monad transformer when `f` is the `Identity` functor.

```
newtype Identity a = Identity a

runIdentity :: forall a. Identity a -> a
runIdentity (Identity a) = a

type SafeT = FreeT Identity

runSafeT :: forall m a. (MonadRec m) => SafeT m a -> m a
runSafeT = runFreeT (return <<< runIdentity)
```

`SafeT m` is a stack-safe monad for any monad `m`. The `Gosub` technique allows us to build large `SafeT m` computations, and `runSafeT` allows us to interpret them, whenever `m` is a tail-recursive monad.

Since `SafeT` is a monad transformer, we can interpret any computation in `m` inside `SafeT m`.

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

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

```
main :: Eff (console :: CONSOLE) Unit
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 `SafeT`:

```
main :: Eff (console :: CONSOLE) Unit
main = runSafeT $ go 100000
  where
  go n | n <= 0 = return unit
  go n = do
   lift (print n)
   go (n - 2)
   go (n - 1)
```

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, `emit`, which emits a single output value:

```
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 only produce values:

```
type Producer o = FreeT (Emit o)

emit :: forall o m. (Monad m) => o -> FreeT (Emit o) m Unit
emit o = liftFreeT (Emit o unit)
```

By using `lift`, we can create coroutines which perform actions in some base monad at each suspension:

```
producer :: forall eff.
  Producer String (Eff (console :: CONSOLE | eff)) 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, transform values, and combinations of these. This is described in [Blaževic], 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 [Blaževic] into PureScript. In addition to `Emit`, 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)

await :: forall i m. (Monad m) => Consumer i m 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 eff.
  (Show a) =>
  Consumer a (Eff (console :: CONSOLE | eff)) Unit
consumer = forever do
  s <- await
  lift (print s)
```

The use of the safe `FreeT` implementation, and `MonadRec` make these coroutines stack-safe.

The `purescript-coroutines` library defines a stack-safe combinator `fuseWith` using `MonadRec`. `fuseWith` can be used to connect compatible coroutines in various ways, by defining how their operations interact when they suspend:

```
fuseWith :: forall f g h m a.
  (Functor f, Functor g, Functor h, MonadRec m) =>
  (forall a b c. (a -> b -> c) -> f a -> g b -> h c) ->
  FreeT f m a -> FreeT g m a -> FreeT h m a
```

For example, `fuseWith` can be used to define an operator `$$`, to connect producers and consumers:

```
($$) :: forall o m a.
  (MonadRec m) =>
  Producer o m a ->
  Consumer o m a ->
  SafeT m a
($$) = fuseWith \f (Emit o a) (Await k) -> Identity (f a (k o))
```

We can connect our producer and consumer pair, and then run them together in parallel using a constant amount of stack:

```
main = runSafeT (producer $$ consumer)
```

Running this example will generate an infinite stream of `"Hello World"` messages 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: List Monad Transformer

The *list monad transformer* `ListT` can be used to represent non-deterministic computations with underlying side-effects described by some base monad. Effects in the base monad may or may not be required, depending on the number of elements required from the final list of successes. For example, a search algorithm might require data from a server, but we would like to minimize the number of round-trips required, if we only require one successful result. Such a problem would be a natural fit for a list monad transformer over an asynchronous monad such as PureScript's `Aff` monad.

Unfortunately, it is not obvious how to construct a stack-safe list monad transformer. However, we can lean on our new free monad transformer, as we'll see.

To see the connection to the free monad transformer, consider the following naïve implementation of `ListT`:

```
data ListF a t = Nil | Cons a t

newtype ListT m a = ListT (m (ListF a (ListT m a)))
```

`ListT` looks like a variant of a cons list, expressed using explicit recursion over a signature functor `ListF`. However, effects from the base monad `m` are allowed every time a new cons cell is constructed.

Adding an additional parameter at the end of the list of cons cells makes the connection to `FreeT` clearer:

```
data ListF a r t = Nil r | Cons a t

newtype ListT m a r = ListT (m (ListF a r (ListT m a r)))
```

The type `ListT m a r` is isomorphic to `FreeT (Emit a) m r` via various simple substitutions. In other words, we can model `ListT` using a free monad transformer by restricting the result type of a producer:

```
newtype ListT m a = ListT (Producer a m Unit)
```

We can run a computation in our hypothetical `ListT` monad by using a writer monad transformer to accumulate multiple results:

```
runListT :: forall m a. (MonadRec m) => ListT m a -> m (List a)
runListT (ListT producer) =
  execWriterT $
    hoistFreeT lift producer $$ consumer
  where
  consumer :: Consumer a (WriterT (List a)) Unit
  consumer = forever do
    a <- await
    lift (tell (singleton a))
```

Here, `hoistFreeT` allows us to change the base monad under a free monad transformer using a natural transformation:

```
hoistFreeT :: forall f m n a.
  (Functor f, Functor n) =>
  (forall a. m a -> n a) ->
  FreeT f m a ->
  FreeT f n a
```

Note that the call to `runFreeT` is acceptable since `WriterT w m` is an instance of `MonadRec` whenever `m` is.

It is possible to write `Functor`, `Applicative` and `Monad` instances for the new `ListT`, although the details are omitted here. It is also possible to write functions which allow us to express non-deterministic computations:

```
nat :: forall m. (Monad m) => ListT m Int

oneOf :: forall m a. (Monad m) => List a -> ListT m a

empty :: forall m a. (Monad m) => ListT m a
```

- `nat` represents a computation which succeeds non-deterministically with some natural number as the result. It is implemented as a producer which emits the naturals in order.
- `oneOf` represents a non-deterministic computation with one of a list of possible successful results. It is implemented as a producer which emits each of the elements in the input list in order.
- `empty` represents a computation which fails. It is implemented as a producer which halts immediately without emitting any values.

We can also implement a variant of the standard `zip` function for `ListT` by fusing two producers using `fuseWith`.

Implementing the `filter` function is possible if we add an additional operator to the base functor for our producers:

```
data EmitMaybe o a = Emit o a | Stall a

newtype ListT m a = ListT (FreeT (EmitMaybe a) m Unit)
```

`filter` replaces each `Emit` with `Stall` whenever the coroutine suspends and the emitted value fails to match the input predicate.

We can now write long-running computations using our `ListT` monad without worrying about stack overflow. For example, this program enumerates an infinite collection of Pythagorean triples and prints them to the console as they are found:

```
triples :: ListT (Eff (console :: CONSOLE | eff)) (Array Int)
triples = do
  x <- nat
  y <- oneOf (1 .. x)
  z <- oneOf (1 .. y)
  if (x * x == y * y + z * z)
    then do lift $ print [x, y, z]
            return [x, y, z]
    else none
```

The caller can incorporate this search into some larger program, and prune the final search tree, running only those side-effects which are necessary to produce some finite result.

Application: Lifting Control Operators

The fact that `SafeT 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 mean functions like `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 unit
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 `SafeT`.

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

```
safeReplicateM_ :: forall m a. (MonadRec m) => Int -> m a -> m Unit
safeReplicateM_ n m = runSafeT (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 n m -> MMorph m n -> o n -> o m
```

Here, the `MMorph f g` type represents a monad morphism from `f` to `g`. `mapO` is given a pair of monads, `m` and `n`, and a pair of monad morphisms, one from `m` to `n`, and one from `n` to `m`. `mapO` is responsible for using these monad morphisms to adapt an implementation of a control operator on `m` to an implementation of that operator on `n`.

In practice, the `SafeT m` monad will be used in place of `n`, where `m` is some tail recursive monad. However, the generality of the type prevents the developer from implementing `mapO` incorrectly.

We can define a function `safely` for any choice of `Operator`:

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

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.

Further Work

Completely-Iterative Monads

In [Capretta], the type of `tailRecM` appears in the definition of a *completely-iterative monad*. Completely-iterative monads seem to solve a related problem: where tail-recursive monads enable stack-safe monadic recursion in a strict language like PureScript, where the evaluation model leads to stack overflow, completely-iterative monads provide the ability to use iteration in total languages where non-termination is considered an effect.

Our `SafeT` monad transformer looks suspiciously similar to the *free completely-iterative monad*, defined as:

```
newtype IterT m a = IterT (m (Either a (IterT m a)))
```

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

This connection might be worth investigating further.

References

- [Bjarnason] *Stackless Scala With Free Monads* by Rúnar Óli Bjarnason
- [Blaževic] *Coroutine Pipelines* by Mario Blaževic, in *The Monad Reader*, Issue 19.
- [Capretta] *Partiality Is an Effect* by Venanzio Capretta, Thorsten Altenkirch, and Tarmo Uustalu, in *Dependently Typed Programming*, Dagstuhl, 2004.
- [PureScript] *PureScript Programming Language* http://purescript.org/
- [Scala] *Scala Programming Language* http://www.scala-lang.org/.