Stack Safety for Free

Phil Freeman

paf31/codemesh2016

Hello!

- I'm Phil, I write Haskell and PureScript
- paf31 on Twitter/GitHub

Motivation

Motivation

Consider this Haskell function:

```haskell
replicateM_ :: Monad m => Int -> m a -> m ()
replicateM_ 0 _ = return ()
replicateM_ n x = x >> replicateM_ (n - 1) x
```

Motivation

We can test this using GHC:

```
main = print $ replicateM_ 100000000 (Just ())
```

This gets compiled to a tight loop:

```
$ ./test +RTS -s
Just ()
          52,104 bytes allocated in the heap

  MUT     time    0.007s  (  0.008s elapsed)
  GC      time    0.000s  (  0.001s elapsed)

  %GC     time       2.0%  (10.5% elapsed)
```

Motivation

But how would we write this function in a strict language?

(PureScript, Scala, etc.)

Motivation

In PureScript:

```
replicateM_ :: ∀ m a. Monad m => Int -> m a -> m Unit
replicateM_ 0 _ = pure unit
replicateM_ n x = x *> replicateM_ (n - 1) x
```

This fails quickly with

```
RangeError: Maximum call stack size exceeded
```

Tail Recursion

Tail Recursion

We need to avoid allocating a stack frame for each iteration:

```
replicateM_ :: ∀ m a. Monad m => Int -> m a -> m Unit
replicateM_ n x = loop (pure unit) n where
  loop :: m Unit -> Int -> m Unit
  loop acc 0 = acc
  loop acc n = loop (x *> acc) (n - 1)
```

Tail Recursion

This works for some monads:

```
> replicateM_ 1000000 (Just 42)
Just unit
```

but continues to fail for others:

```
> replicateM_ 1000000 (log "testing")
RangeError: Maximum call stack size exceeded
```

Tail Recursion

Recap:

A tail recursive function can either

- return a value
- or loop, modifying some function arguments

at each step

Tail Recursion

Well, let's reify those constraints as a data structure:

```
data Step a b
  = Done b
  | Loop a
```

Tail Recursion

Now we can write a general-purpose tail-recursive function of one argument:

```
tailRec :: ∀ a b. (a -> Step a b) -> a -> b
```

This can be used to write variants with multiple arguments:

```
tailRec2 :: ∀ a b c
          . (a -> b -> Step (a, b) c)
         -> a -> b -> c
```

Tail Recursion

This is enough to reimplement `replicateM_`:

```
replicateM_ :: ∀ m a. Monad m => Int -> m a -> m Unit
replicateM_ n x = tailRec2 loop (pure unit) n where
  loop :: m Unit -> Int -> Step (m Unit, Int) (m Unit)
  loop acc 0 = Done acc
  loop acc n = Loop (x *> acc, n - 1)
```

Of course, this doesn't solve the problem, yet

Tail-Recursive Monads

Tail-Recursive Monads

The trick:

Generalize `tailRec` to *monadic tail recursion* using a new type class

```
class Monad m => MonadRec m where
  tailRecM :: (a -> m (Step a b)) -> a -> m b
```

What should the laws be?

Tail-Recursive Monads

`tailRecM` should be equivalent to the default definition:

```
tailRecM f a =
  step <- f a
  case step of
    Done b -> pure b
    Loop a1 -> tailRecM f a1
```

However, we can provide a more efficient implemenation!

Tail-Recursive Monads

Example: `ExceptT`

```
newtype ExceptT e m a = ExceptT (m (Either e a))

instance MonadRec m => MonadRec (ExceptT e m) where
  tailRecM f = ExceptT <<< tailRecM \a ->
    case f a of ExceptT m ->
      m >>= \m' ->
        pure case m' of
          Left e -> Done (Left e)
          Right (Loop a1) -> Loop a1
          Right (Done b) -> Done (Right b)
```

Tail-Recursive Monads

More Examples

- `Identity`
- `StateT s`
- `WriterT w`
- `ReaderT r`
- `Eff eff`
- `Aff eff`

Tail-Recursive Monads

We can fix `replicateM_` by requiring `MonadRec`:

```
replicateM_ :: ∀ m a. MonadRec m => Int -> m a -> m Unit
replicateM_ n x = tailRecM loop n where
  loop :: Int -> m (Step Int Unit)
  loop 0 = pure (Done unit)
  loop n = x $> Loop (n - 1)
```

This is stack-safe for any law-abiding `MonadRec` instance!

We can also implement other functions like `mapM` and `foldM`.

Tail-Recursive Monads

Taxonomy of Recursion Schemes

- `StateT`: Additional accumulator
- `WriterT`: Tail-call modulo "cons"
- `ExceptT`: Tail-call with abort

Applications

1. Free Monads

Free Monads

The free monad:

```
data Free f a = Pure a | Impure (f (Free f a))
```

has a similar problem in strict languages.

Free Monads

```
runFree :: ∀ m f a
         . Monad m
        => (f (Free f a) -> m (Free f a))
        -> Free f a
        -> m a
runFree f (Pure a) = pure a
runFree f (Impure xs) = do
  next <- f xs
  runFree f next
```

We cannot interpret deep computations without risking blowing the stack.

Free Monads

Solution:

Instead we use

```
runFree :: ∀ m f a
         . MonadRec m
        => (f (Free f a) -> m (Free f a))
        -> Free f a
        -> m a
```

`runFree` can be written using `tailRecM` directly and uses a constant amount of stack.

2. Free Monad Transformers

Free Monad Transformers

The free monad transformer:

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

interleaves effects from the base monad `m`.

Free Monad Transformers

The technique extends to the free monad transformer:

```
runFreeT :: ∀ m f a
          . MonadRec m
         => (f (FreeT f m a) -> m (FreeT f m a))
         -> FreeT f m a
         -> m a
```

3. Stack Safety for Free

Stack Safety for Free

We can write a `MonadRec` instance for `FreeT f m` whenever `m` itself has `MonadRec`.

In particular, we can write an instance for

```
type SafeT = FreeT Identity
```

Stack Safety for Free

`SafeT` is a monad transformer:

```
lift :: ∀ m a. m a -> SafeT m a
```

but we can also lower values:

```
lower :: ∀ f a. MonadRec m => SafeT m a -> m a
lower = runFreeT (pure <<< runIdentity)
```

This gives a way to evaluate arbitrarily-nested monadic binds safely for any `MonadRec`!

Stack Safety for Free

If we are feeling lazy, we can just use the original implementation!

```
replicateM_ :: ∀ m a. MonadRec m => Int -> m a -> m Unit
replicateM_ = (lower <<<) <<< go where
  go 0 _ = pure unit
  go n x = lift x *> replicateM_ (n - 1) x
```

`SafeT` also has induced instances for several `mtl` classes.

4. Coroutines

Coroutines

The free monad transformer gives a nice model for coroutines.

For example:

```
data Emit o a = Emit o a

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

`Producer _ (Aff _)` is useful for modelling *asynchronous generators*

Coroutines

Consumers:

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

type Consumer i = FreeT (Consumer i)
```

`Consumer _ (Aff _)` is useful for modelling *asynchronous enumerators*

E.g. chunked handling of HTTP responses

Coroutines

```
type Fuse f g h = ∀ a b c
                . (a -> b -> c)
               -> f a
               -> g b
               -> h c

producerConsumer :: Fuse (Emit o) (Await o) Identity

fuse :: ∀ f g h m a
      . (Functor f, Functor g, Functor h, MonadRec m)
     => Fuse f g h
     -> FreeT f m a
     -> FreeT g m a
     -> FreeT h m a
```

Coroutines

Examples:

- Websockets
- File I/O
- AJAX

Conclusion

Conclusion

- `MonadRec` can make a variety of tasks safe in a strict language like PureScript
- We trade off some instances for a safe implementation
- `MonadRec` has been implemented in PureScript, Scalaz, cats and fantasy-land.

Thanks!