Effects for Less

Main types of Effect Systems:

1) **Monad Transformers + Type Classes**
   > Generally believed to be pretty fast
   > Can be complicated and require a lot of boilerplate
   Examples: mtl, fused-effects

2) **Free-like Monads**
   > Performance is a known limitation
   > Highly flexible, can be simpler to use
   Examples: freer-simple, polysemy

We will consider:

- Ordinary monad transformers
- MTL library
- Free monads

Free Monads as an Effect System

```
program :: State Int Int
program = get >>= \n -> if n <= 0
                        then return n
                        else put (n - 1) >> program
```

```
data Eff f a where
    Return :: a   -> Eff a
    Then   :: f a -> (a -> Eff b) -> Eff b
```

```
data State s a where
    Get :: State s s
    Put :: s -> State ()
```

```
program :: Eff (State Int) Int
program = Get Then \n -> if n <= 0
                         then (Return n)
                         else (Put (n - 1) (\_ -> program))
```

```
runState :: s -> Eff (State s) a -> (s, a)
runState s (Return x)        = (s, x)
runState s (Get   `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

Free Monads as an Effect System

Pros:
- Beautifully simple
- Extremely flexible

Cons:
- Have to concretise the entire program as a tree, rather than something more abstract.
- Obscures the program’s structure to the optimiser.

Optimizer Process: Ordinary Monad Transformers

```
newtype State s a = State { runState :: s -> (s, a) }

instance Monad (State s) where
    return x = State $ \s -> (s, x)
    m >>= f  = State $ \s -> case runState m s of
                                   (s’, a) -> runState (f a) s’
get:: State s s
get = State $ \s -> (s, s)
put :: s -> State s ()
put s = State $ \_ -> (s, ())


program :: State Int Int
program = get >>= \n -> if   n <= 0
                        then return n
                        else put (n - 1) >> program
```

Optimizer Process: Ordinary Monad Transformers

```
program :: State Int Int
program = get >>= \n -> if   n <= 0
                        then return n
                        else put (n - 1) >> program


program :: State Int Int
program = State $ \s1 -> case runState get s1 of
                            (s2, n) -> if   n <= 0
                                       then runState (return n) s2
                                       else case runState (put (n - 1)) s2 of
                                              (s3, _) -> runState program s3
```

Optimizer Process: Ordinary Monad Transformers

```
program :: State Int Int
program = State $ \s1 -> case runState get s1 of
                            (s2, n) -> if   n <= 0
                                       then runState (return n) s2
                                       else case runState (put (n - 1)) s2 of
                                              (s3, _) -> runState program s3


program :: Int -> (Int, Int)
program s1 = case get s1 of
               (s2, n) -> if n <= 0
                          then return n s2
                          else case put (n - 1) s2 of
                                 (s3, _) -> program s3
```

Optimizer Process: Ordinary Monad Transformers

```
program :: Int -> (Int, Int)
program s1 = case get s1 of
               (s2, n) -> if n <= 0
                          then return n s2
                          else case put (n - 1) s2 of
                                 (s3, _) -> program s3
```

```
program :: Int -> (Int, Int)
program s1 = case get s1 of
               (s2, n) -> if n <= 0
                          then (s2, n)
                          else case (n - 1, ()) of
                                 (s3, _) -> program s3
```

Optimizer Process: Ordinary Monad Transformers

```
program :: Int -> (Int, Int)
program s1 = case get s1 of
               (s2, n) -> if   n <= 0
                          then (s2, n)
                          else  case (n - 1, ()) of
                                  (s3, _) -> program s3
```

```
program :: Int -> (Int, Int)
program s1 = case get s1 of
               (s2, n) -> if   n <= 0
                          then (s2, n)
                          else program (n - 1)
```

Optimizer Process: Ordinary Monad Transformers

```
program :: Int -> (Int, Int)
program s1 = case get s1 of
               (s2, n) -> if   n <= 0
                          then (s2, n)
                          else program (n - 1)
```

```
program :: Int -> (Int, Int)
program s1 = case (s1, s1) of
               (s2, n) -> if   n <= 0
                          then (s2, n)
                          else program (n - 1)
```

```
program :: Int -> (Int, Int)
program s1 = if   s1 <= 0
             then (s1, s1)
             else program (s1 - 1)
```

```
program :: Int -> (Int, Int)
program n = if n <= 0
            then (n, n)
            else program (n - 1)
```

Limitations of Monad Transformers

```haskell
program :: State Int Int
program = get >>= \n -> if n <= 0
                        then return n
                        else put (n - 1) >> program
```

Limitations of Monad Transformers

The list fusion problem

```
foldr (+) 0 [1 .. 5]

foldr (+) 0 (1 : 2 : 3 : 4 : 5 : [])
```

```
1 : 2 : 3 : 4 : 5 : []

1 + 2 + 3 + 4 + 5 + 0
```

The free monad approach

```
program :: Eff (State Int) Int
program = Get Then \n -> if n <= 0
                         then (Return n)
                         else (Put (n - 1) (\_ -> program))

runState :: s -> Eff (State s) a -> (s, a)
runState s (Return x)        = (s, x)
runState s (Get   `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

Limitations of Monad Transformers

The list fusion problem

```
foldr (+) 0 [1 .. 5]

foldr (+) 0 (1 : 2 : 3 : 4 : 5 : [])
```

```
1 : 2 : 3 : 4 : 5 : []

1 + 2 + 3 + 4 + 5 + 0
```

The free monad approach

```
program :: Eff (State Int) Int
program = Get Then \n -> if n <= 0
                          then (Return n)
                          else (Put (n - 1) (\_ -> program))

runState :: s -> Eff (State s) a -> (s, a)
runState s (Return x)        = (s, x)
runState s (Get   `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

Optimizer Process: The MTL Approach

```
program :: MonadState Int m => m Int
program = get >>= \n -> if n <= 0
                        then (return n)
                        else (put (n - 1) >> program)
```

How Type Classes Are Compiled

```
exclaim :: Show a => a -> String
exclaim x = show x ++ “!”

exclaim :: (a -> String) -> a -> String
exclaim show_a x = show_a x ++ “!”


: exclaim True
> exclaim show_Bool True
```

How Type Classes Are Compiled: Dictionary Passing

```
class Show a where
     show :: a -> String
     showPrec :: Int -> a -> ShowS
     showList :: [a] -> ShowS



data Show a = ShowDict
     { show      :: a -> String,
       showPrec :: Int -> a -> ShowS,
       showList :: [a] -> ShowS }



exclaim :: Show a -> a -> String
exclaim dict x = show dict x ++ “!”
```

Pros:

- Elegantly simple
- Cheap to compile

How Type Classes Are Compiled: Dictionary Passing

What's the run-time cost?

```
program :: MonadState Int m => m Int
program = get >>= \n -> if n <= 0
                        then return n
                        else put (n - 1) >> program
```

```
program :: MonadState Int m => m Int
program stateDict@(MonadStateDict monadDict _ _) =
    (>>=) monadDict
        (get stateDict)
        (\n -> if   n <= 0
               then return monadDict n
               else (>>) monadDict
                        (put stateDict (n - 1))
                        (program stateDict)
```

```
class (Monad m) => MonadState s m
```

Known Calls

- Exposes strictness information which in turn exposes more optimisations
- Arguments can be unboxed - an unboxed type is represented by the value itself, so no pointers or heap allocation is involved.
- Can be rewritten by rewrite rules (good thing)
- Can be inlined (if sufficiently small)

Unknown Calls

- Assumes (pessimistically) to be lazy in all arguments
- Arguments can’t be unboxed. GHC can’t do anything special to those functions because it doesn’t know what they are.
- Are fundamentally opaque to any rules
- Are never inlined

GHC is an “ahead-of-time” compiler

Type Class Overloading → Unknown Calls

Type class overloading results in dictionary passing and then pulling functions out of these dictionary records.

=

Passing functions to functions directly.

Overloading is NOT free and has a performance cost!

Unknown calls does not mean we're doomed

```
sumIndices :: Eq a => a -> [a] -> [Int]
sumIndices v xs = sum
                . map fst
                . filter ((== v) . snd)
                . zip [1 ..] xs
```

(==) doesn't exist "in between" any of these functions!

Bind (>>=)

(>>=) exists “in between” functions!

```
foo :: k -> Map k Int -> Either String Int
foo key vals = do
     nums <- case Map.lookup key vals of
               Nothing  -> Left “not found”
               Just val -> Right [1 .. val]
     Right $ sum nums
```

```
foo :: MonadError String m => k -> Map k Int -> m Int
foo key vals = do
     nums <- case Map.lookup key vals of
               Nothing  -> throwError “not found”
               Just val -> return [1 .. val]
     return $ sum nums
```

Conclusion: MTL is inconsistently performant

This is due to specialization

The idea behind specialization is:

1) GHC looks for calls to overloaded functions at known, concrete types.
2) A function must satisfy one of the following criteria in order to be specialised.
    - It is defined in the current module.
    - It is declared with an `INLINEABLE` pragma.
    - It is a class method and its source code is stored in the interface file (.hi).

```haskell
program :: MonadState Int m => m Int
program = do
   n <- get
   if n <= 0
     then return n
     else put (n - 1) >> program
```

When a function is called in the **same module** it was defined in...

```haskell
program :: State Int Int
program = get >>= \n ->
               if n <= 0
               then return n
               else put (n - 1) >> program
```

When a function is called in a **different module** it was defined in...

```haskell
program :: MonadState Int m => m Int
program stateDict@(MonadStateDict monadDict _ _) =
     (>>=) monadDict
            (get stateDict)
            (\n -> if   n <= 0
                   then return monadDict n
                   else (>>) monadDict
                               (put stateDict (n - 1))
                               (program stateDict)
```

Can we avoid this performance regression? Is specialisation really necessary? (no)

We make two propositions:

1. Effect systems are about dynamic dispatch.

```
program :: State Int Int
program = get >>= \n -> if n <= 0
                        then return n
                        else put (n - 1) >> program
```

An effect system dynamically provides an interpretation of these two operations (that may be far away from where the function is originally defined).

Can we avoid this performance regression? Is specialisation really necessary? (no)

We make two propositions:

2. Unknown calls are not the core problem, the problem is (>>=).

   Passing (>>=) via dictionary passing creates problems:

   - It gets called a lot!
   - It acts like “glue” between operations - it needs to be inlined in order to expose any further optimisations.
   - Unknown calls to (>>=) increases “closure allocation”.

“If (>>=) is an unknown call, the caller must allocate a closure for the continuation”

```
f x y = foo x >>= \z -> bar (+ y z)
```

```
f x y = case foo x of
          Left e -> Left e
          Right z -> bar (+ y z)
```

USING MTL

```
program :: MonadState Int m => m Int
program = do
      n <- get
      if n <= 0
        then return n
        else put (n - 1) >> program
```

USING FREE MONADS

```
program :: Eff (State Int) Int
program =
      Get Then
        (\n -> if n <= 0
               then (Return n)
               else (Put (n - 1)
                          Then (\_ -> program))
```

Escape Plan

1) We need to have a monad with an inlineable (>>=)
2) (>>=) must not allocate any closures
3) It must be able to handle all algebraic effects
4) Effect dispatch can be dynamic, but it must be fast

~~Monad Transformers~~

~~Free Monads~~

Delimited Continuations