
SIDE
EFFECTS
HISTORY.COM

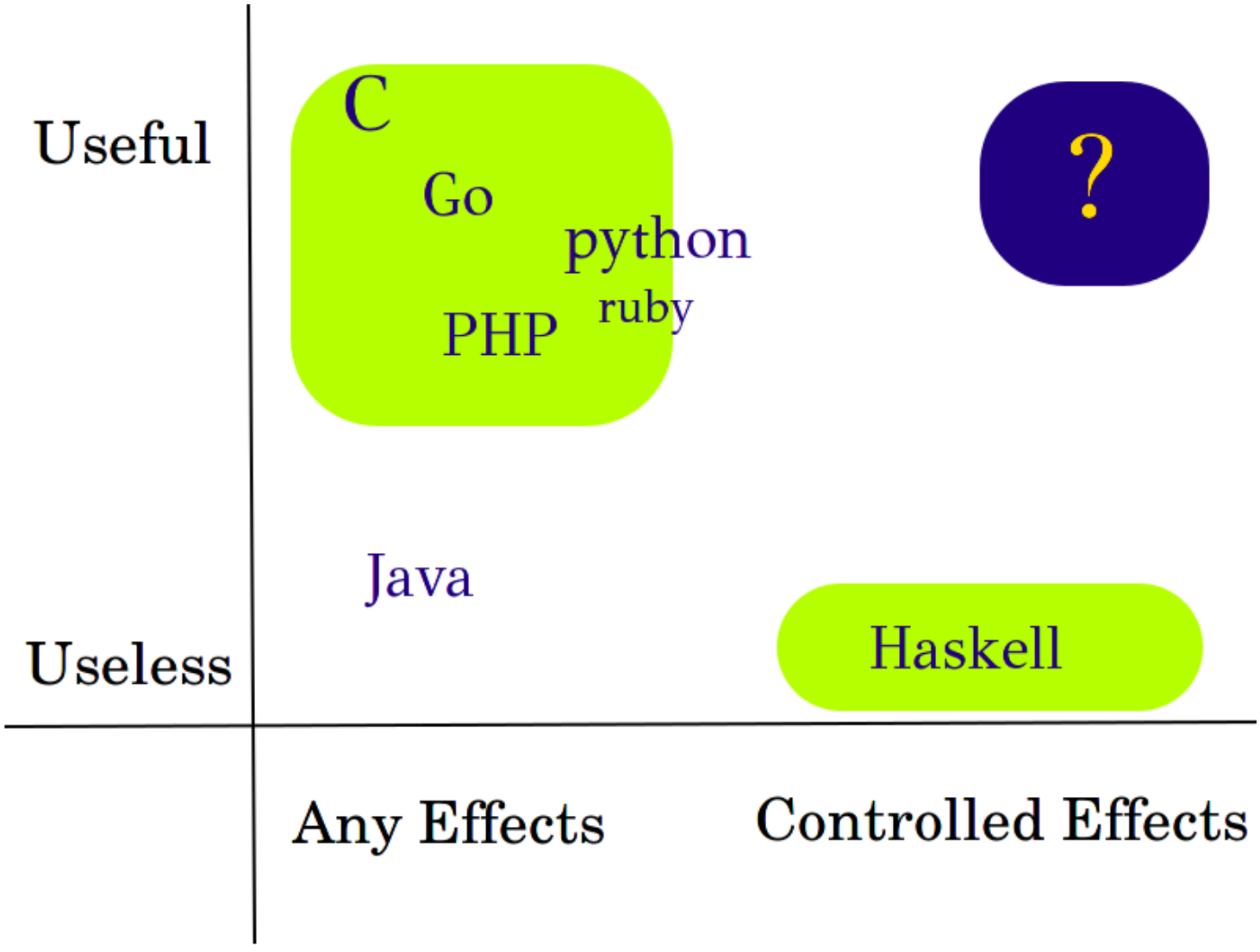
Computer

Effects

Exceptions

Concurrency/Parallelism

Modern Languages



Purity/Controlled effects matter!

Kernel

- Privileges

Security applications

- Side-channel resistant applications

Big Data

- Computing in parallel

Effects as Types

My result is of type T:

```
Exception(IO(T))
```

Effects as Types

Computing the result involves input/output side effects...

`IO(T)`

... which come with exceptions I want to catch:

`Exception(IO(T))`

Effects as Types

Type of getCurrentTime:

```
IO(Time)
```

Effects as Types

Type of getCurrentTime:

```
IO(Time)
```

We can only evaluate time through side effects!

Effects as Types

Type of sum:

```
[Num] -> Num
```

Effects as Types

Type of sum:

```
[Num] -> Num
```

We can evaluate the result without side effects!

Effects as Types

Type of $(+)$:

```
Num -> Num -> Num
```

Effects as Types

Type of $(1 + 1)$:

`Num`

Effects as Types

Implementation

...

Extensible effects

- Monad transformers

Arrow transformers

Kan extensions

...

With Haskell type classes...

... we get monads for free!

Type classes

Eq for Equality

```
class Eq a where
  (==) :: a -> a -> Bool
```

Type classes

Booleans can be equal, for instance:

```
instance Eq Bool where
    True == True = True
   False == False = True
       _ == _ = False
```

Monads from category theory

Two operations we want:

```
 unit T = M(T)
join M(M(T)) = M(T)
```

Monads in Haskell

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

The Maybe Monad

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x
```

The Identity Monad

```
data Identity a = Identity a

instance Monad Identity where
  return x = Identity x
  m >>= _ = m
```

unit and join in Haskell monads

unit is return

```
return T = M(T)
```

unit and join in Haskell monads

```
id x = x

join :: m (m a) -> m a
join m = m >>= id
```

Something crazy

```
data Maybe a = Just a | Nothing

newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }
```

Something crazy

```
data Maybe a = Just a | Nothing

newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }

instance Monad (MaybeT Identity) where
  return x = MaybeT (Identity x)
  m >>= f = MaybeT (runMaybeT m >>= maybe')
   where maybe' (Just x) = runMaybeT (f x)
         maybe' Nothing = Identity Nothing
```

Monad transformers

returning Nothing is break!

```
instance Monad (MaybeT Identity) where
  return x = MaybeT (Identity x)
  m >>= f = MaybeT (runMaybeT m >>= maybe')
   where maybe' (Just x) = runMaybeT (f x)
         maybe' Nothing = Identity Nothing
```

Monad transformers

```
instance Monad (MaybeT Identity) where
  return x = MaybeT (Identity x)
  m >>= f = MaybeT (runMaybeT m >>= maybe')
   where maybe' (Just x) = runMaybeT (f x)
         maybe' Nothing = Identity Nothing
```

put MaybeT on any monad transformer stack!

```
instance Monad m => Monad (MaybeT m) where
  return x = MaybeT (return x)
  m >>= f = MaybeT (runMaybeT m >>= maybe')
   where maybe' (Just x) = runMaybeT (f x)
         maybe' Nothing = return Nothing
```

Monad transformers

break any monadic computation!

```
instance Monad m => Monad (MaybeT m) where
  return x = MaybeT (return x)
  m >>= f = MaybeT (runMaybeT m >>= maybe')
   where maybe' (Just x) = runMaybeT (f x)
         maybe' Nothing = return Nothing
    fail _ = MaybeT (return Nothing)
```

Idiomatic use

```
main :: IO ()
main = do
  runMaybeT verify
  putStrLn "Bye!"

verify :: MaybeT IO ()
verify = forever $ do
  line <- lift getLine
  when (line == "richard") (fail "")
```

Idiomatic use

```
f =
  m1 >>= \a ->
  m2 >>= \b ->
  m3 >>= \c ->
  return (a, b, c)
```

```
f = do
  a <- m1
  b <- m2
  c <- m3
  return (a, b, c)
```

```
f = liftM3 (,,) m1 m2 m3
```

```
f = (,,) <$> m1 <*> m2 <*> m3
```

How to lift

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

How to lift

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a

instance MonadTrans MaybeT where
  lift m = MaybeT (liftM Just m)
  where liftM f m = m >>= return (f m)
```

How to lift

```
lift2 m = lift (lift m)
```

How to lift

```
lift2 ::
  ( MonadTrans t
  , MonadTrans u
  , Monad (t m)
  , Monad m
  ) => m a -> u (t m) a
lift2 m = lift (lift m)
```

Futures

`Future(T)`

Futures

"The eventual result of an asynchronous operation."

`Future(T)`

Futures

```
type Future a = MaybeT STM a
```

Futures

```
runFuture :: Future a -> IO (Maybe a)
runFuture f = atomically (runMaybeT f)
```

Futures

```
input :: TMVar Int -> Future Event
input transactional = do
  status <- lift (takeTMVar transactional)
  when (status < 0) (fail "")
  return (eventFromStatus status)
```

Futures

```
eventFromStatus :: Int -> Event
```

Futures

```
main :: IO ()
main = do
  transactional <- newTMVar 0
  forkIO (inputDevice0 transactional)
  forkIO (inputDevice1 transactional)
  -- ...
  loop

loop :: IO ()
loop transactional = do
  event <- runFuture (input transactional)
  unless (isNothing event) (loop transactional)
```

Promises

```
type Reason = String
type Promise a = EitherT Reason STM a
```

Promises

```
data Either a b = Left a | Right b

runPromise :: Promise a -> IO (Either Reason a)
runPromise p = atomically (runEitherT p)
```

Promises

```
loop :: IO ()
loop transactional = do
  event <- runFuture (event transactional)
  case event of
    Left  reason -> putStrLn reason
    Right _      -> loop transactional
```

Promises as Functors

```
liftPromise :: (a -> b) -> Promise a -> Promise b
liftPromise f p = p >>= apply
  where apply x = return (f x)
```

Monads are Functors

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f m = m >>= apply
  where apply x = return (f x)
```

Functors in Haskell

```
class Functor a where
  fmap :: (a -> b) -> f a -> f b
```

Functors in Haskell

```
class Functor a where
  fmap :: (a -> b) -> f a -> f b

instance Functor Promise where
  fmap = liftM
```

Functors in Haskell

```
promise :: Promise [Int]

sum :: [Int] -> Int
```

Functors in Haskell

```
promise :: Promise [Int]

sum :: [Int] -> Int

liftM sum promise  :: Promise Int
fmap sum promise   :: Promise Int
sum <$> promise    :: Promise Int
sum . promise      :: Promise Int
```

Monads as specialized functors

```
(F, unit, join)
```

Purity/Controlled effects matter!

Kernel

- Privileges

Security applications

- Side-channel resistant applications

Big Data

- Computing in parallel

All applications have effects, take control!

Sources

- http://github.com/promises-aplus
- http://okmij.org/ftp/Haskell/extensible
- http://hackage.haskell.org/package/transformers
- http://www.haskell.org/arrows
- http://hackage.haskell.org/package/kan-extensions
- http://www.haskell.org/haskellwiki/Monad_tutorials_timeline