Concurrent Orchestration in Haskell

John Launchbury
Trevor Elliott

galois

Code Puzzle

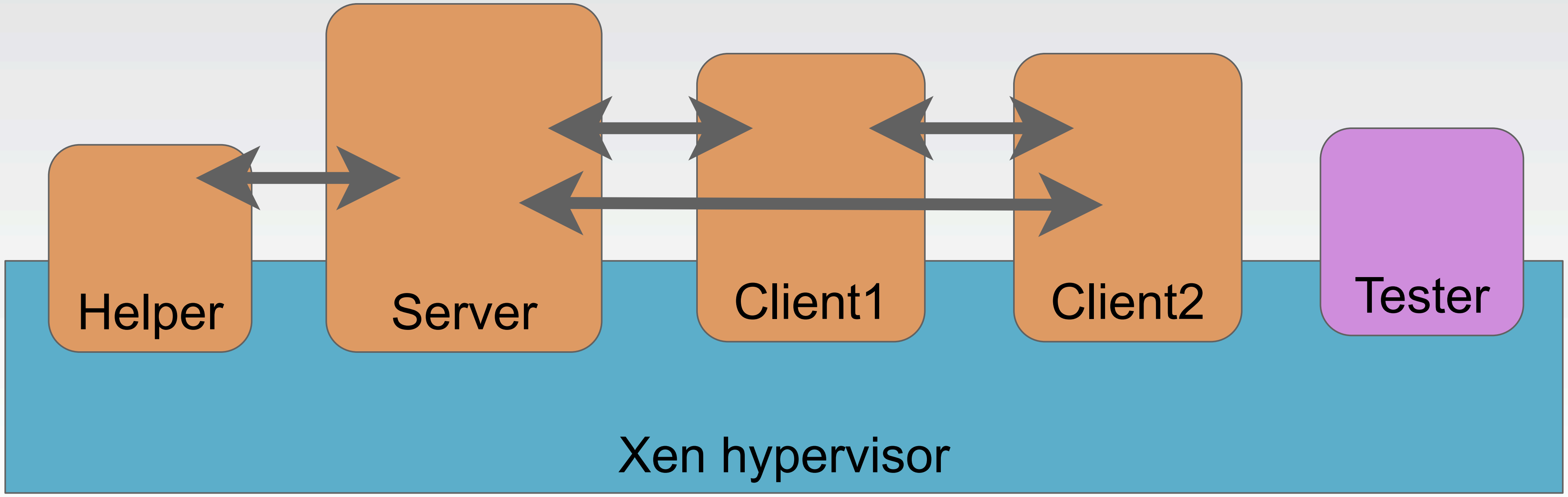
```
foo :: (a -> s -> s) -> s -> Orc a -> Orc s
foo f s p =  do a <- newMVarM s
                x <- p
                v <- takeMVarM a
                let w = f x v
                putMVarM a w
                return w
```

This code implements
a well-known idiom —
as we go on,
try to figure out what it is...

Outline

- Concurrent scripting
- Laws
- Thread management

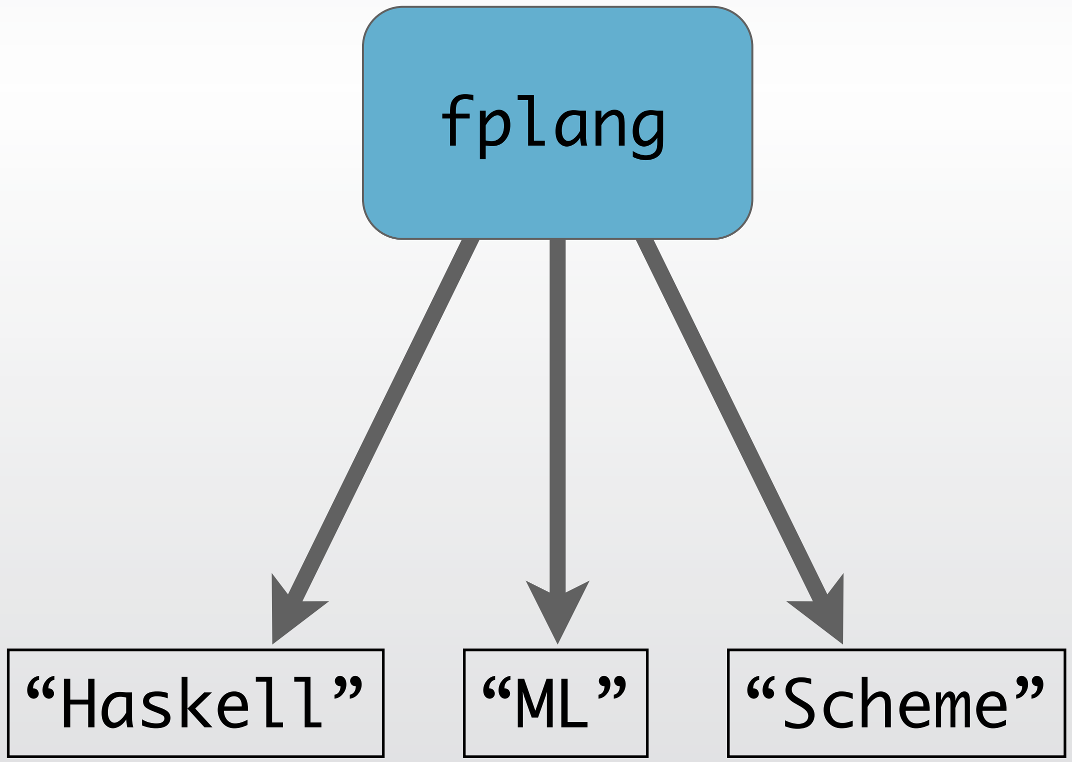
Testing Xen Virtual Machines



- Tester talks with each of the VMs concurrently
- Many possible behaviors are “correct” / “incorrect”
- Timeouts, VMs dying, etc.
- Subtle concurrency bugs in test framework

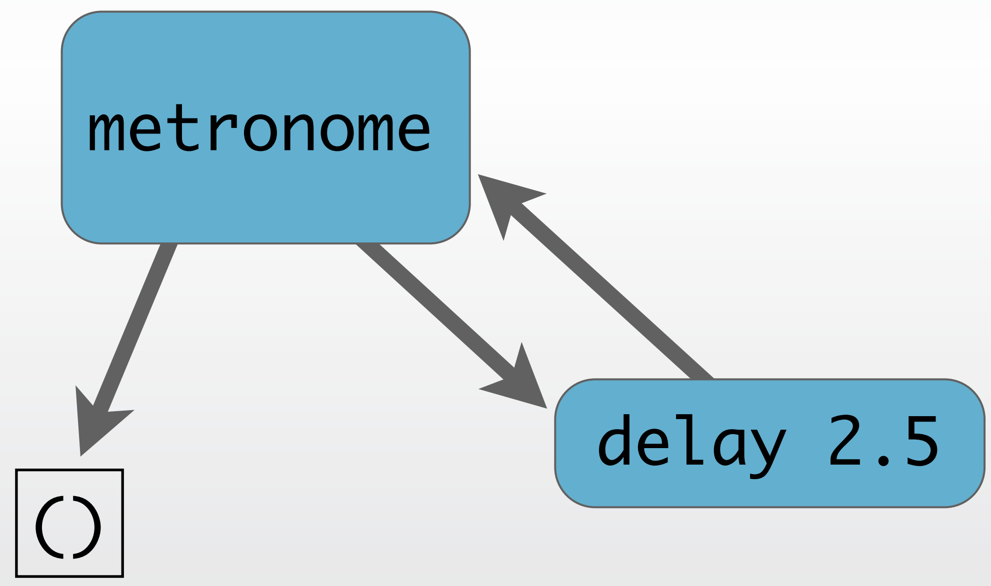
Orc Example

```
fplang :: Orc String
fplang = return "Haskell" <|> return "ML" <|> return "Scheme"
```



Orc Example

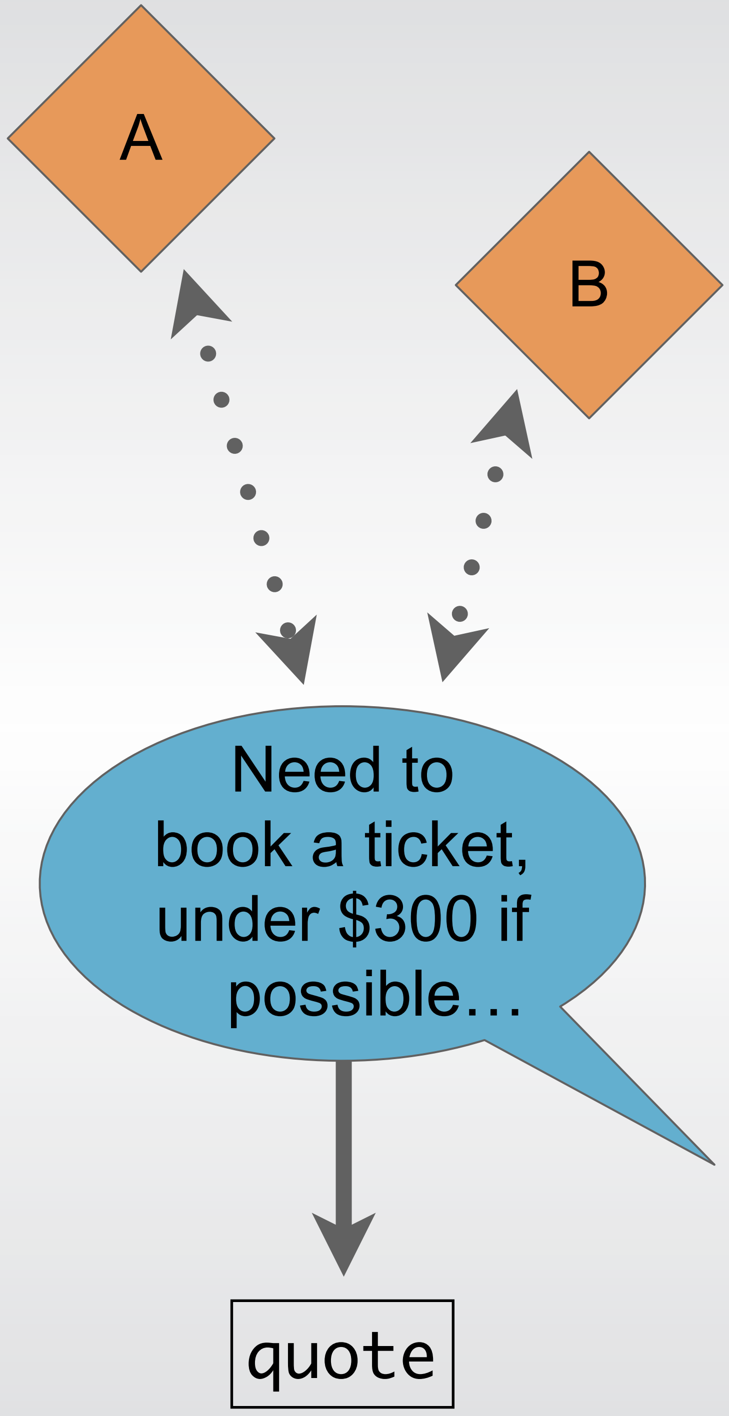
```
metronome :: Orc ()
metronome = return () <|> (delay 2.5 >> metronome)
```



galois

Orc Example

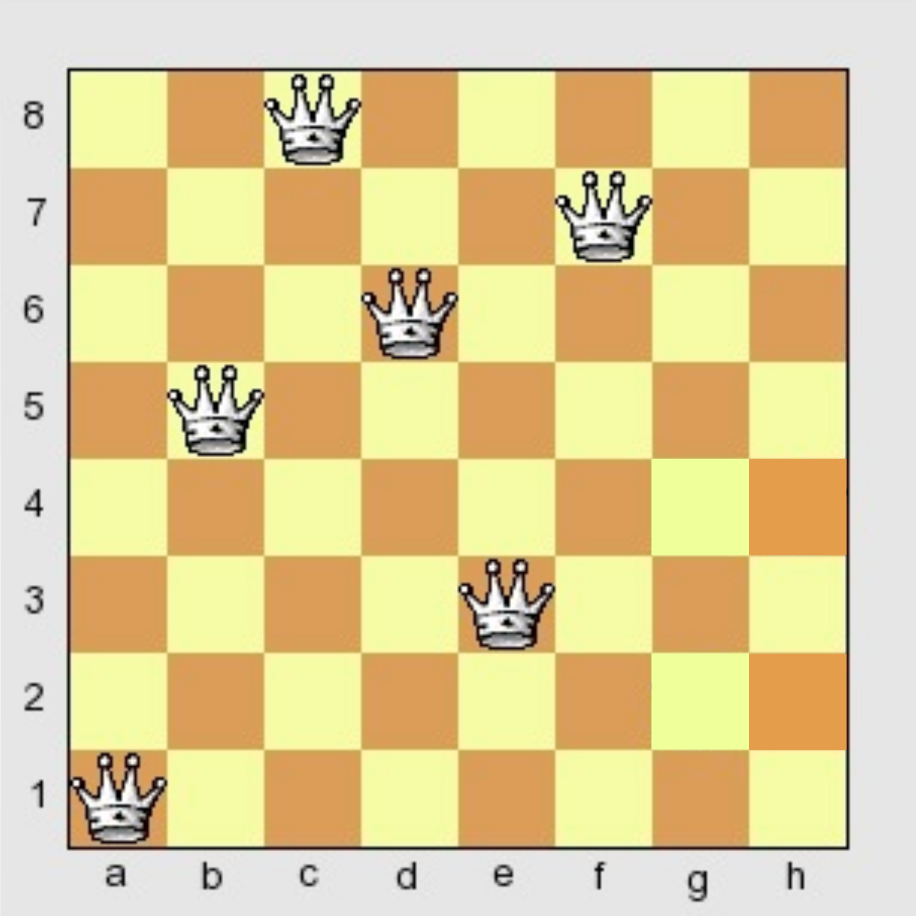
```haskell
quotes :: Query -> Query -> Orc Quote
quotes srcA srcB = do
  quoteA <- eagerly $ getQuote srcA
  quoteB <- eagerly $ getQuote srcB
  cut (  (return least <*> quoteA <*> quoteB)
     <|> (quoteA >>= threshold)
     <|> (quoteB >>= threshold)
     <|> (delay 25 >> (quoteA <|> quoteB))
     <|> (delay 30 >> return noQuote))

least x y = if price x < price y then x else y
threshold x = guard (price x < 300) >> return x
```



galois

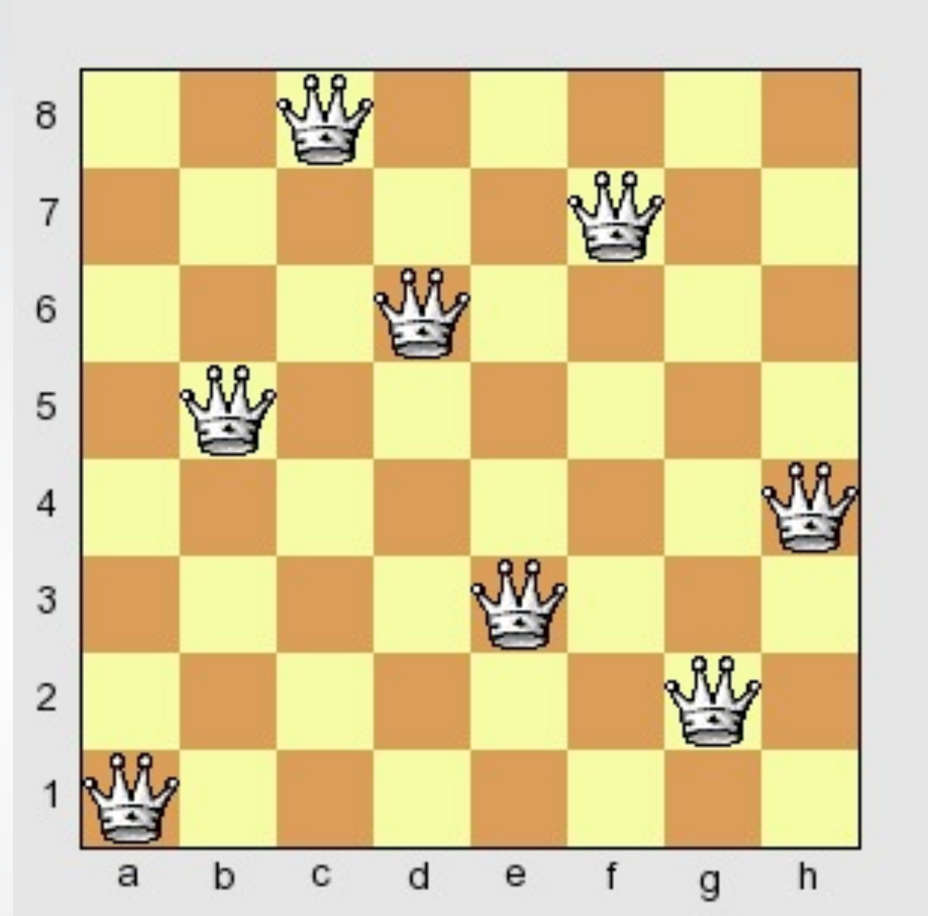
Orc Example

```
queens =    fmap show (extend [])
       <|> return ("Computing 8-queens...")

extend :: [Int] -> Orc [Int]
extend xs = if length xs == 8
            then return xs
            else do
              j <- listOrc [1..8]
              guard $ not (conflict xs j)
              extend (j:xs)

conflict :: [Int] -> Int
conflict = ...

listOrc :: [a] -> Orc a
listOrc = foldr (<|>) stop . map return
```



Orc Example

```
*Main> printOrc (queens)
Ans = "Computing 8-queens..."
Ans = "[5,7,1,3,8,6,4,2]"
Ans = "[5,2,4,7,3,8,6,1]"
Ans = "[6,4,2,8,5,7,1,3]"
Ans = "[5,3,8,4,7,1,6,2]"
Ans = "[4,2,7,3,6,8,5,1]"
:

*Main> printOrc (queens)
Ans = "Computing 8-queens..."
Ans = "[4,2,7,3,6,8,5,1]"
Ans = "[6,4,7,1,8,2,5,3]"
Ans = "[3,6,8,1,4,7,5,2]"
Ans = "[3,6,4,2,8,5,7,1]"
Ans = "[2,7,3,6,8,5,1,4]"
:
```



galois

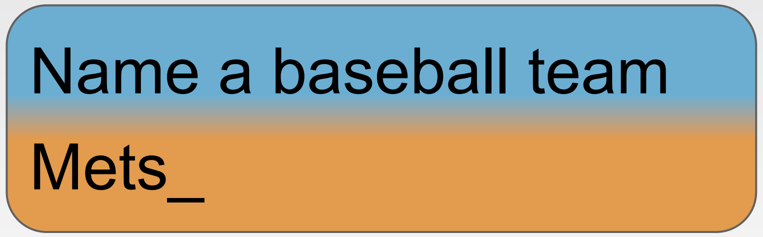
Orc Example

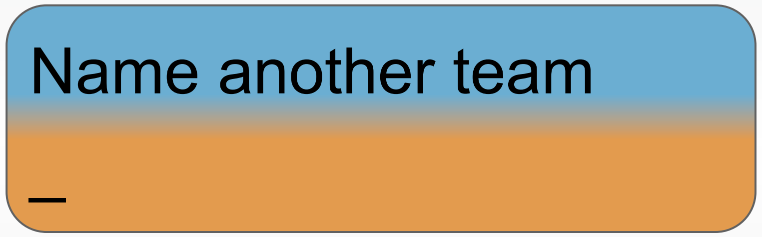
```
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
            `after` (12, return "Yankees")
      <|> prompt "Name another team"
            `notBefore` 10
      <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like "++team++"?")
           `after` (20, guard (team/="Mets") >> return "maybe")

  return (team, agree)
```

galois

Orc Example

```
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
            `after` (12, return "Yankees")
      <|> prompt "Name another team"
            `notBefore` 10
      <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like "++team++"?")
           `after` (20, guard (team/="Mets") >> return "maybe")

  return (team, agree)
```





galois

Orc Example

```
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
            `after` (12, return "Yankees")
      <|> prompt "Name another team"
            `notBefore` 10
      <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like "++team++"?")
           `after` (20, guard (team/="Mets") >> return "maybe")

  return (team, agree)
```

Name a baseball team
Mets_

Name another team
_

Do you like Mariners?
_

galois

Orc Example

```
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
            `after` (12, return "Yankees")
      <|> prompt "Name another team"
            `notBefore` 10
      <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like "++team++"?")
           `after` (20, guard (team/="Mets") >> return "maybe")

  return (team, agree)
```

Name a baseball team

Mets_

Name another team

_

Do you like Mets?

_

Do you like Mariners?

_

galois

Orc Example

```
baseball :: Orc (String,String)
baseball = do

  team <- prompt "Name a baseball team"
             `after` (12, return "Yankees")
      <|> prompt "Name another team"
             `notBefore` 10
      <|> (delay 8 >> return "Mariners")

  agree <- prompt ("Do you like "++team++"?")
            `after` (20, guard (team/="Mets") >> return "maybe")

  return (team, agree)
```

Name a baseball team

Mets_

Name another team

_

Do you like

_

Do you like Mets?

_

Do you like Mariners?

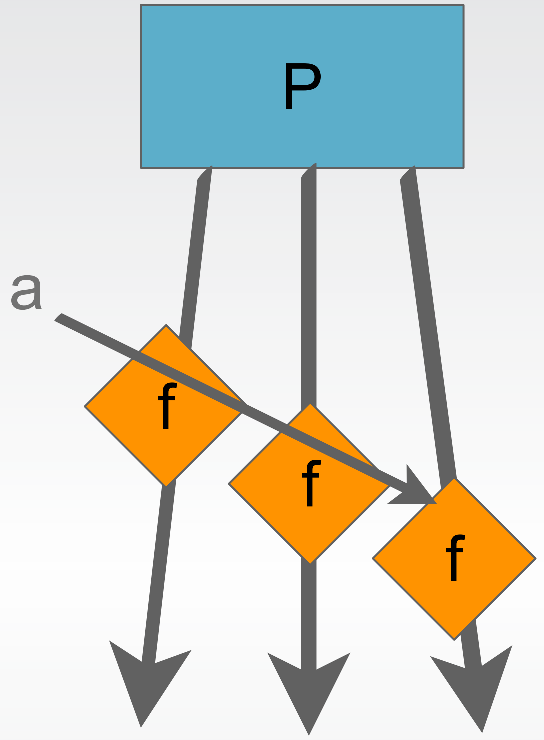
_

Code Puzzle

```
foo  :: (a -> s -> s) -> s -> Orc a -> Orc s
foo f s p =  do a <- newMVarM s
                x <- p
                v <- takeMVarM a
                let w = f x v
                putMVarM a w
                return w
```

Orc Code

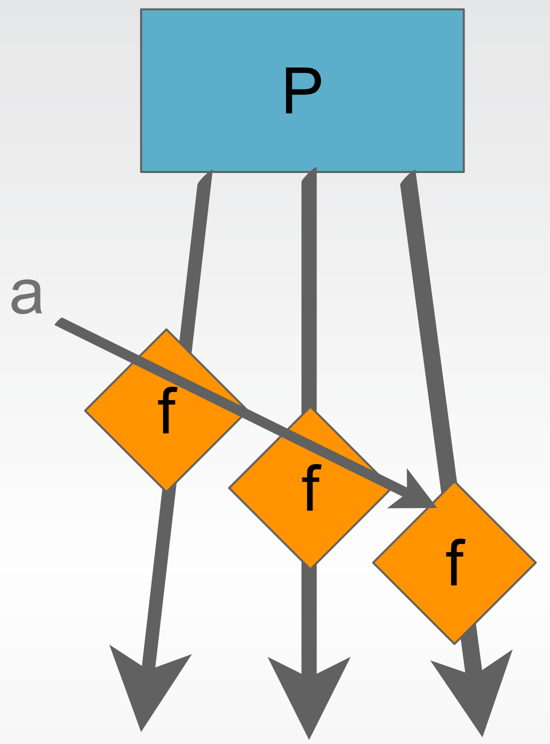
```
scan :: (a -> s -> s) -> s -> Orc a -> Orc s
scan f s p = do a <- newMVarM s
                x <- p
                v <- takeMVarM a
                let w = f x v
                putMVarM a w
                return w

% printOrc (scan (+) 0 $ listOrc [1,2,3,4,5])
```



galois

Orc Code

```
scan :: (a -> s -> s) -> s -> Orc a -> Orc s
scan f s p = do a <- newMVarM s
                x <- p
                v <- takeMVarM a
                let w = f x v
                putMVarM a w
                return w

% printOrc (scan (+) 0 $ listOrc [1,2,3,4,5])
Ans = 1
Ans = 3
Ans = 6
Ans = 11
Ans = 15
%
```



Layered Implementation

Orc Scripts

Orc Monad	multiple results
HIO Monad	thread control
IO Monad	external effects

Transition Semantics

- Layered implementation — layered semantics
  - Properties at one level depend on properties at the level below
- What properties should Orc terms satisfy?
  - Hence, what properties should be built into HIO?
- Unresolved question: what laws should the basic operations of the IO monad satisfy?

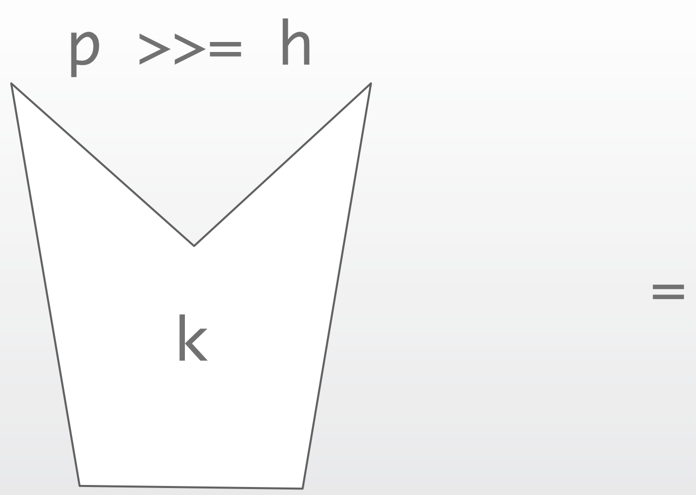
Key Definitions

```
type Orc a = (a -> HIO ()) -> HIO ()

return x = \k -> k x
p >>= h  = \k -> p (\x -> h x k)
p <|> q  = \k -> fork (p k) >> q k

stop     = \k -> return ()

runOrc p = p (\x -> return ())
```

galois

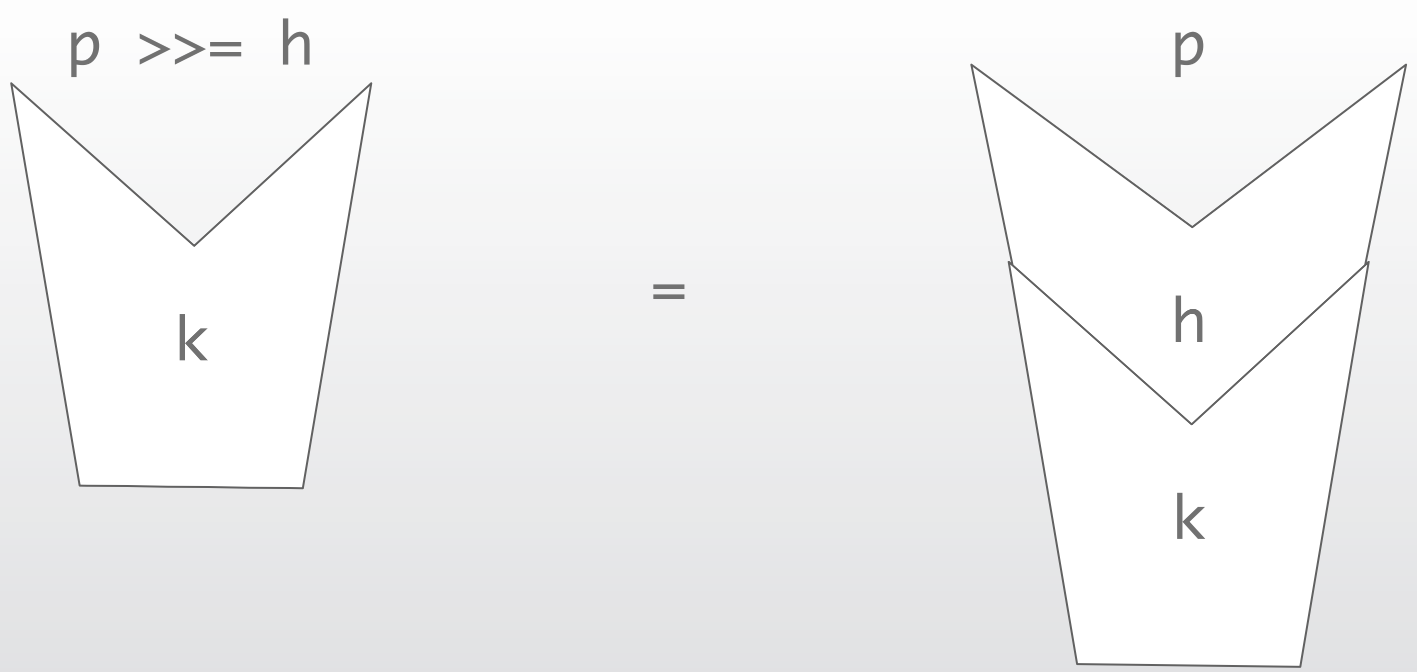
Bind

```
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x
p >>= h  = \k -> p (\x -> h x k)
p <|> q  = \k -> fork (p k) >> q k
```



Bind

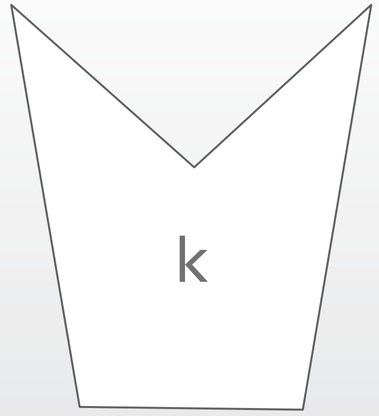
```
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x
p >>= h  = \k -> p (\x -> h x k)
p <|> q  = \k -> fork (p k) >> q k
```



Par

```
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x
p >>= h  = \k -> p (\x -> h x k)
p <|> q  = \k -> fork (p k) >> q k
```

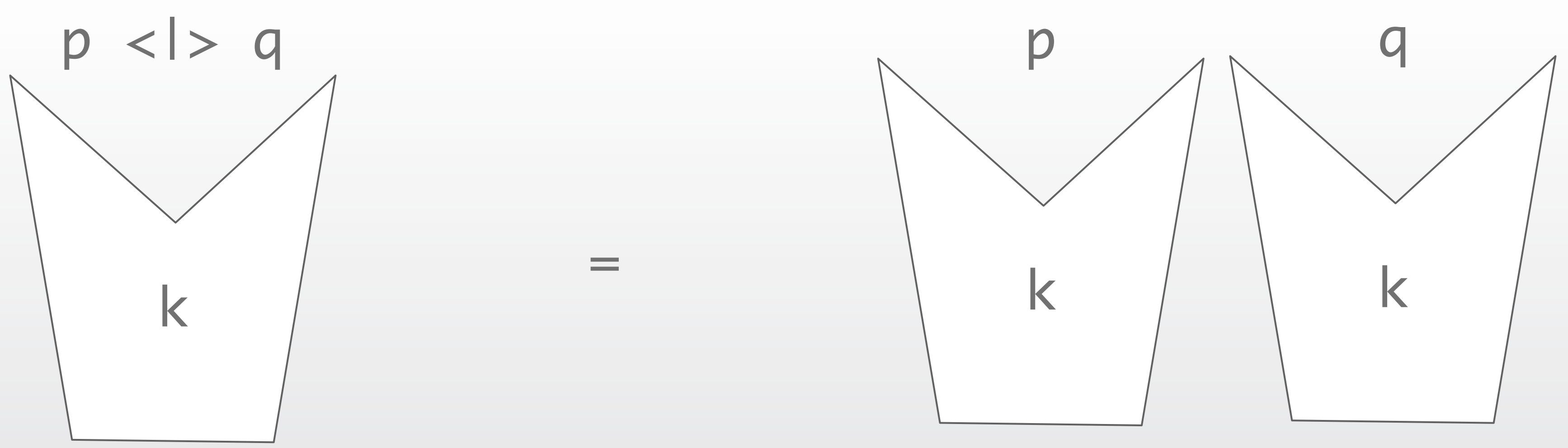
`p <|> q`



=

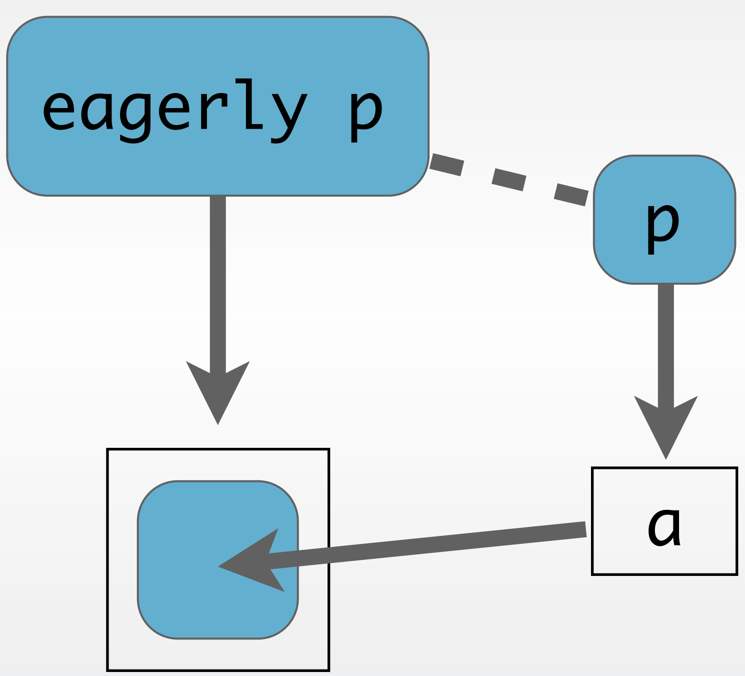
Par

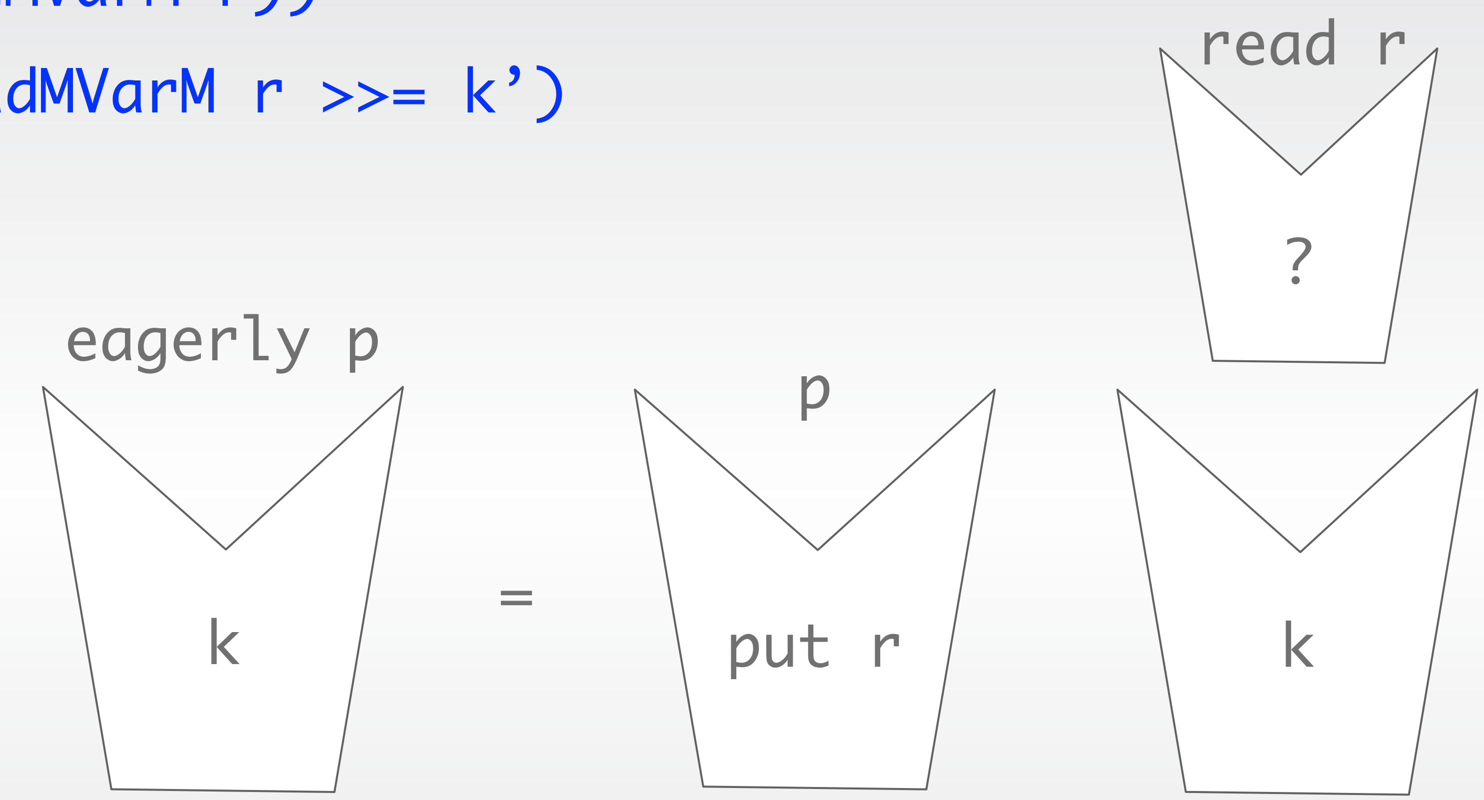
```
type Orc a = (a -> HIO a) -> HIO a

return x = \k -> k x
p >>= h  = \k -> p (\x -> h x k)
p <|> q  = \k -> fork (p k) >> q k
```



Eagerly

```
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
              r <- newEmptyMVarM
              forkM (p (putMVarM r))
              k (\k' -> readMVarM r >>= k')
```





- Give p a continuation that will store its result
- Return the “value” that accesses that result for the then current continuation

Eagerly

```
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
              r <- newEmptyMVarM

              forkM (p `saveOnce` (r  ))
              k (\k’ -> readMVarM r >>= k’)

saveOnce :: Orc a -> (MVar a         ) -> HIO ()
p `saveOnce` (r  ) = do

    p (\x ->                   putMVarM r x          )
```

- Give p a continuation that will store its result (but once only even if duplicated)
- Return the “value” that accesses that result for the then current continuation

galois

Eagerly

```
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
               r <- newEmptyMVarM

               forkM (p `saveOnce` (r  ))
               k (\k’ -> readMVarM r >>= k’)

saveOnce :: Orc a -> (MVar a         ) -> HIO ()
p `saveOnce` (r  ) = do
    ticket <- newMVarM ()
    p (\x -> takeMVarM ticket >> putMVarM r x          )
```

- Give p a continuation that will store its result (but once only even if duplicated)
- Return the “value” that accesses that result for the then current continuation

Eagerly

```
eagerly :: Orc a -> Orc (Orc a)
eagerly p = \k -> do
              r <- newEmptyMVarM
              e <- newLocality
              local e $ forkM (p `saveOnce` (r,e))
              k (\k' -> readMVarM r >>= k')

saveOnce :: Orc a -> (MVar a,Locality) -> HIO ()
p `saveOnce` (r,e) = do
    ticket <- newMVarM ()
    p (\x -> takeMVarM ticket >> putMVarM r x >> close e)
```

- Give p a continuation that will store its result (but once only even if duplicated)
- Return the "value" that accesses that result for the then current continuation
- Thread management can be carried over too

galois

Eagerly

```
sync :: (a->b->c) -> Orc a -> Orc b -> Orc c
sync f p q = do
  po <- eagerly p
  qo <- eagerly q
  return f <*> po <*> qo

notBefore:: Orc a -> Float -> Orc a
p `notBefore` w = sync const p (delay w)
```

- Entering the handle waits for the result
- Synchronization
- cut

Eagerly

```
sync :: (a->b->c) -> Orc a -> Orc b -> Orc c
sync f p q = do
  po <- eagerly p
  qo <- eagerly q
  return f <*> po <*> qo

notBefore:: Orc a -> Float -> Orc a
p `notBefore` w = sync const p (delay w)
```

```
cut:: Orc a -> Orc a
cut p = do
  po <- eagerly p
  po
```

- Entering the handle waits for the result
- Synchronization
- cut

Eagerly

```
sync :: (a->b->c) -> Orc a -> Orc b -> Orc c
sync f p q = do
  po <- eagerly p
  qo <- eagerly q
  return f <*> po <*> qo

notBefore:: Orc a -> Float -> Orc a
p `notBefore` w = sync const p (delay w)
```

- Entering the handle waits for the result
- Synchronization
- cut

```
cut:: Orc a -> Orc a
cut p = do
  po <- eagerly p
  po
```

```
cut:: Orc a -> Orc a
cut = join . eagerly
```

Orc Laws

Left-Return: `(return x >>= k) = k x`

Right-Return: `(p >>= return) = p`

Bind-Associativity: `((p >>= k) >>= h) = (p >>= (k >=> h))`

Stop-Identity: `p <|> stop = p`

Par-Commutativity: `p <|> q = q <|> p`

Par-Associativity: `p <|> (q <|> r) = (p <|> q) <|> r`

Left-Zero: `(stop >>= k) = stop`

Par-Bind: `((p <|> q) >>= k) = ((p >>= k) <|> (q >>= k))`

Non-Laws

Bind-Par?: `p >>= (\x -> h x <|> k x) = (p >>= h) <|> (p >>= k)`

Right-Zero?: `p >> stop = stop`

galois

Non-Laws

Bind-Par?: `p >>= (\x -> h x <|> k x) = (p >>= h) <|> (p >>= k)`

Right-Zero?: `p >> stop = stop`

```
p `until` done = cut (silent p <|> done)
silent p = p >> stop
```

Non-Laws

Bind-Par?: `p >>= (\x -> h x <|> k x) = (p >>= h) <|> (p >>= k)`

Right-Zero?: `p >> stop = stop`

```
p `until` done = cut (silent p <|> done)
silent p = p >> stop

hassle = (metronome >> email("Simon","Hey!"))
           `until`
              (delay 60 >> return ())
```

Eagerly Laws

Eagerly-Par:

```
eagerly p >>= (\x -> k x <|> h) = (eagerly p >>= k) <|> h
```

Eagerly-Swap:

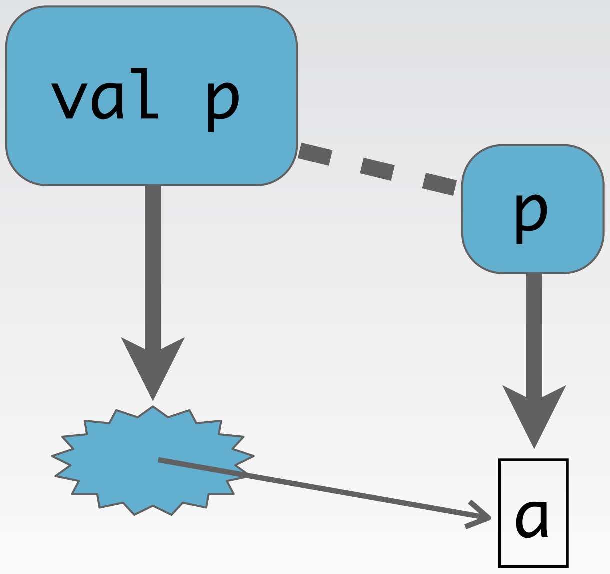
```
do y <- eagerly p    =   do x <- eagerly q
   x <- eagerly q           y <- eagerly p
   return (x,y)             return (x,y)
```

Eagerly-IO:

```
eagerly (ioOrc m) >> p = (ioOrc m >> stop) <|> p
```

Val

```
val       :: Orc a -> Orc a
val p       = \k -> do
                r <- newEmptyMVarM
                e <- newLocality
                local e $ forkM (p `saveOnce` (r,e))
                k (unsafePerformIO $ readMVarM r)

saveOnce :: Orc a -> (MVar a,Locality) -> HIO ()
p `saveOnce` (r,e) = do
    ticket <- newMVarM ()
    p (\x -> takeMVarM ticket >> putMVarM r x >> close e)
```



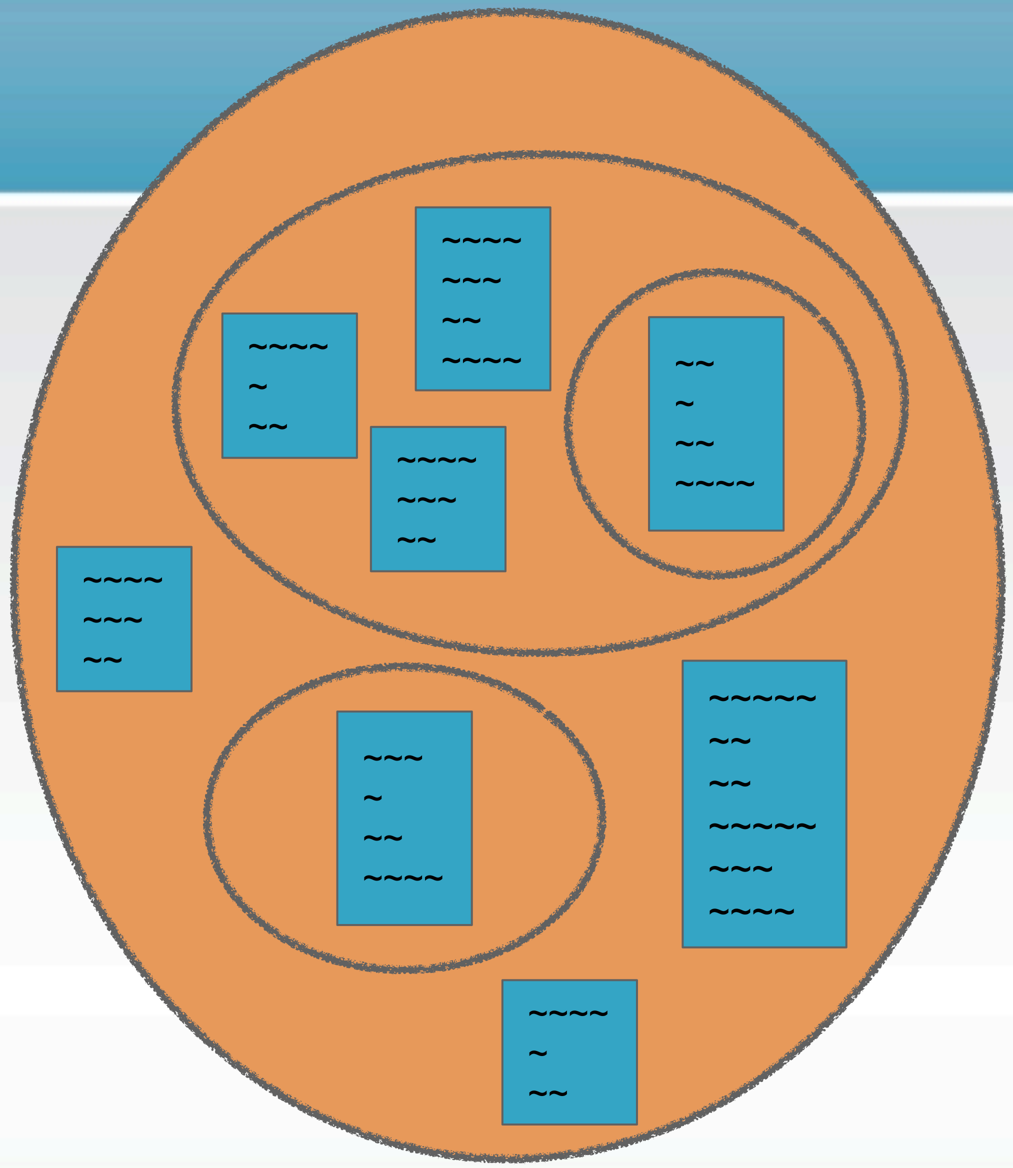
- The implementation of val (the alternative that uses lazy thunks) is almost identical

Example

```haskell
quotesVal :: Query -> Query -> Orc Quote
quotesVal srcA srcB = do
  quoteA <- val $ getQuote srcA
  quoteB <- val $ getQuote srcB
  cut (  publish (least quoteA quoteB)
     <|> (threshold quoteA)
     <|> (threshold quoteB)
     <|> (delay 25 >> (publish quoteA <|> publish quoteB))
     <|> (delay 30 >> return noQuote))

publish :: NFData a => a -> Orc a
publish x = deepseq x $ return x
```

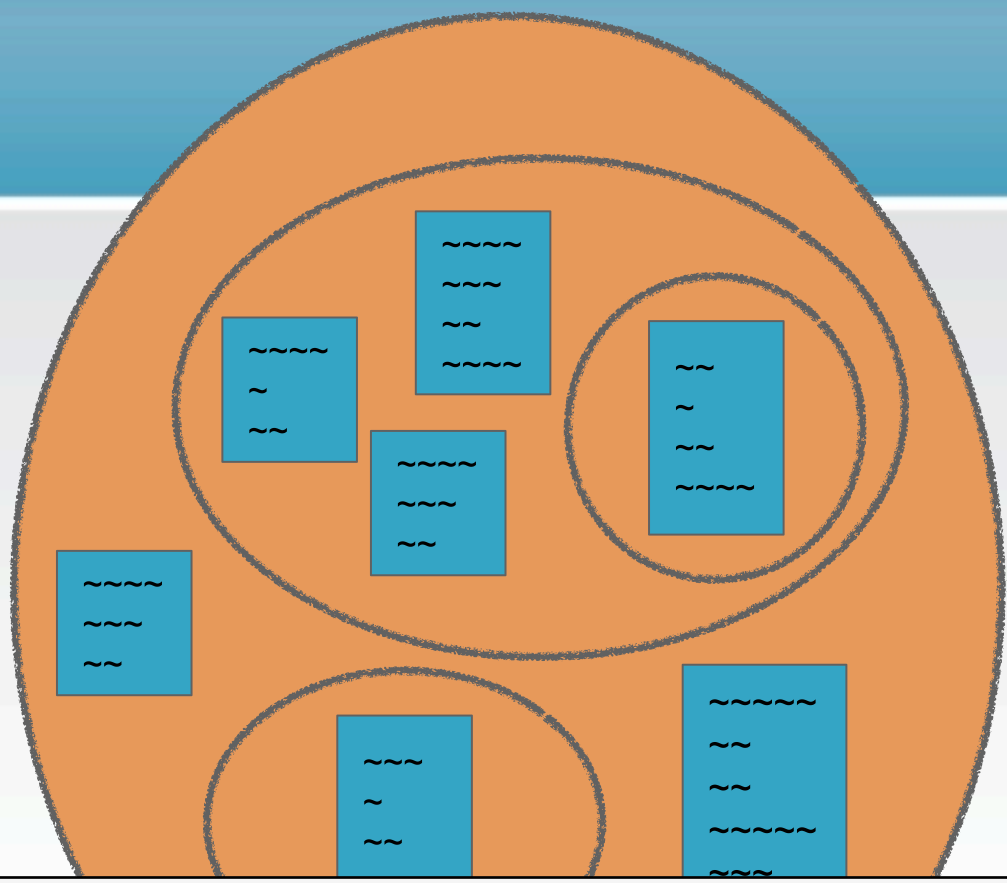
- Good: use the lazy values directly
- Bad: have to be careful about evaluation

HIO Monad



- Don’t want the programmer to have to do explicit thread management
  - Nested groups of threads
- Want richer equational theory than IO
  - e.g. by managing asynchronous exceptions

HIO Monad



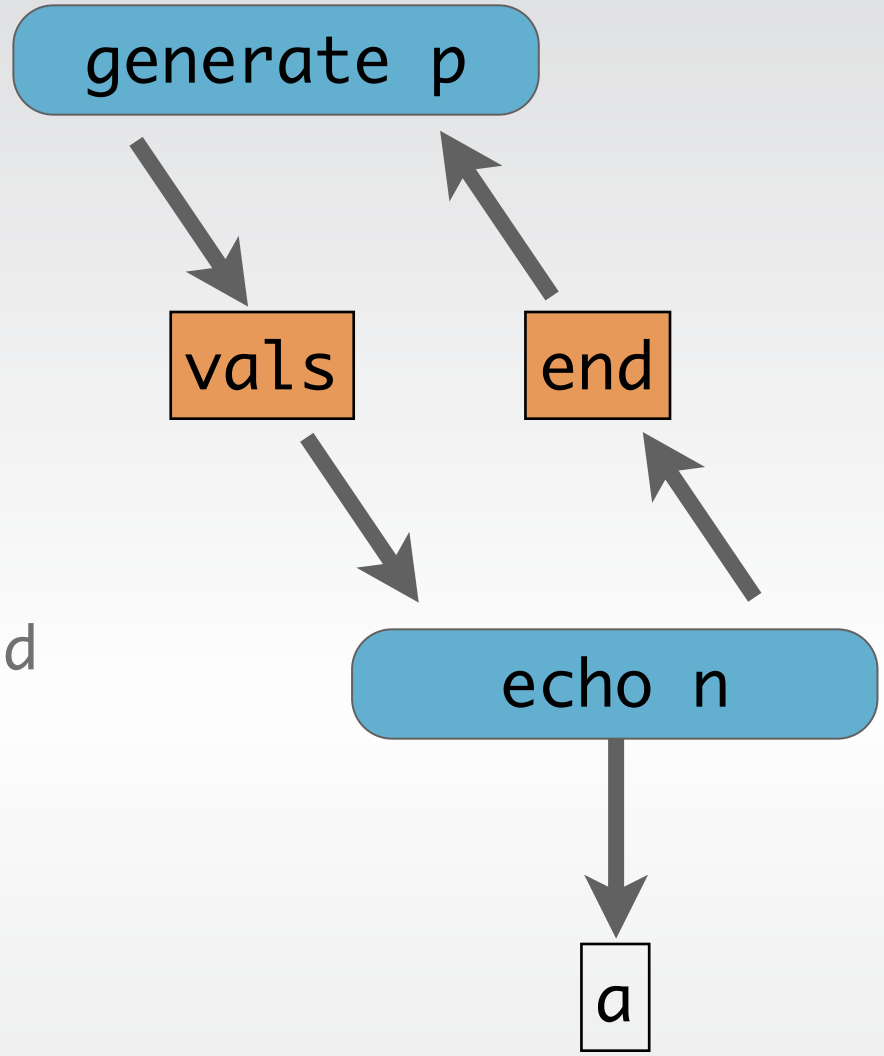
- Don't want the programmer to have to do explicit thread management
  - Nested groups of threads
- Want richer equational theory than IO
  - e.g. by managing asynchronous exceptions

```
newtype HIO a = HIO {inGroup :: Locality -> IO a}
type Group   {- abstract -}
data Entry  = Thread ThreadId
            | Group Group

newGroup  :: IO Group
register  :: Entry -> Group -> IO ()
killGroup :: Group -> IO ()
```

galois

Orc Example

```
first :: Int -> Orc a -> Orc a
first n p = do
  vals <- newEmptyMVarM
  end <- newEmptyMVarM
  echo n vals end
    <|> silent (generate p vals end)

generate p vals end =
  (p >>= putMVarM vals) `until` takeMVarM end

echo n vals end = loop n
  where loop 0 = silent $ putMVarM end ()
        loop n = do x <- takeMVarM vals
                    return x <|> loop (n-1)
```



- Use MVars to communicate
- Use `until` to kill-off work when finished

Final Fun

Standard function:

```
filterM _ []      =  return []
filterM p (x:xs) =  do
    b <- p x
    ys  <- filterM p xs
    return (if b then x:ys else ys)
```

```
baz :: [a] -> Orc [a]
baz xs = filterM pred xs

pred x = return False <|> return True
```

This code implements a well-known function — what is it?