Programs That Explain Their Effects

Dominic Orchard

University of Kent

"Haskell is the world's finest imperative programming language."

Simon Peyton Jones

but.....

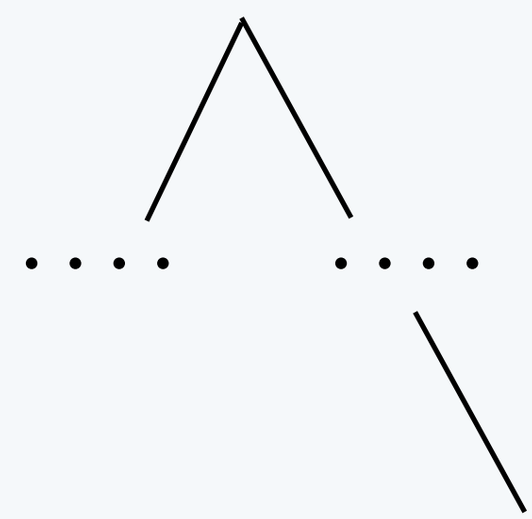
```haskell
myProgram :: State Int String
myProgram = do
  x <- get      -- get :: State Int Int
  let a = somethingPure x
  put (x+1)     -- put :: Int -> State Int ()
  return (a ++ show x)
```

```
> runState myProgram 0
("hello1", 1)
```

```haskell
runState :: State s a -> s -> (a, s)
```

```haskell
myProgram :: State Int String
myProgram = do
  x <- get
  let a = somethingPure x
  put (x+1)
  return (a ++ show x)
```

```haskell
somethingPure :: Int -> String
```

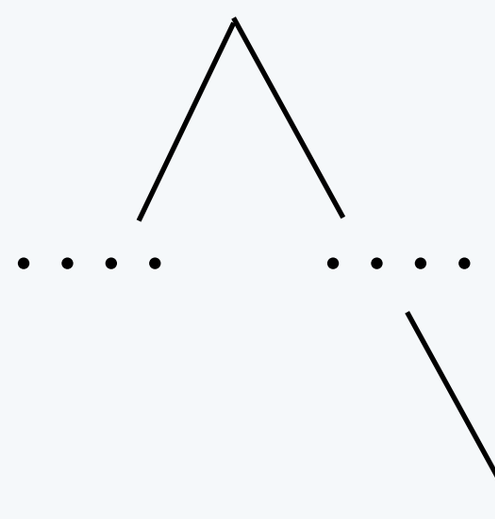


```haskell
f :: ... -> a
```

version 1

```haskell
myProgram :: State Int String
myProgram = do
  x <- get
  let a = somethingPure x
  put (x+1)
  return (a ++ show x)
```

```haskell
somethingPure :: Int -> String
```

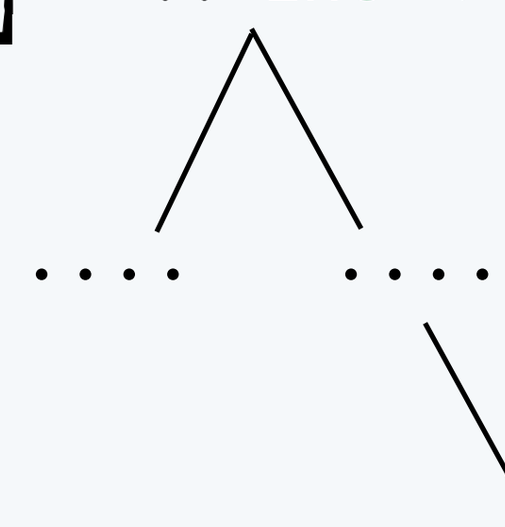


```haskell
f :: ... -> State Int a
```

version 1

```haskell
myProgram :: State Int String
myProgram = do
  x <- get
  let a = somethingPure x
  put (x+1)
  return (a ++ show x)
```

somethingPurish :: Int -> State Int String

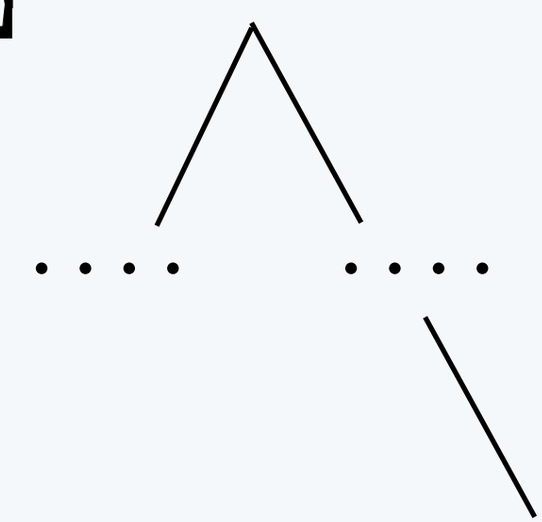


f :: ... -> State Int a

version 1

```haskell
myProgram :: State Int String
myProgram = do
   x <- get
   a <- somethingPurish x
   put (x+1)
   return (a ++ show x)
```

```haskell
somethingPurish
   :: Int -> State Int String
```



```haskell
   f :: ... -> State Int a
```

version 2

```haskell
myProgram :: State Int String
myProgram = do
  x <- get
  a <- somethingPurish x
  put (x+1)
  return (a ++ show x)
```

Effects

shadowed by

```
> runState myProgram 0
("hello1", 1)
```

```haskell
modify :: (s -> s) -> State s ()
```

version 2

Impure

```
State Int String

IO String
```

Pure

```
String
```

But how impure is it?

Impure

Pure

```
State Int String
```

```
String
```

Update Write Read Pure

Regular monads...

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

... replace....

```
import Prelude hiding (Monad(..))
```

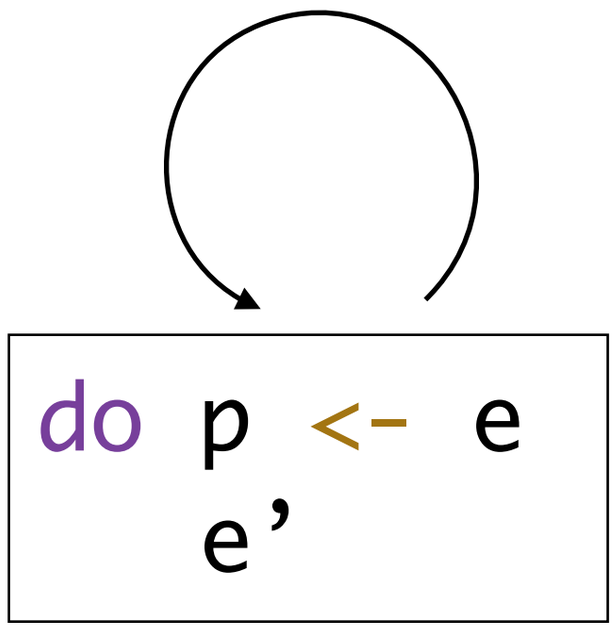
... with parameterised monads (Atkey 2006,2009)

```
class PMonad (p :: k -> k -> * -> *) where
  return :: a -> p x x a
  (>>=) :: p x y a -> (a -> p y z b) -> p x z b
```

What about do? I want do!

Usually…

Type checking



desugar

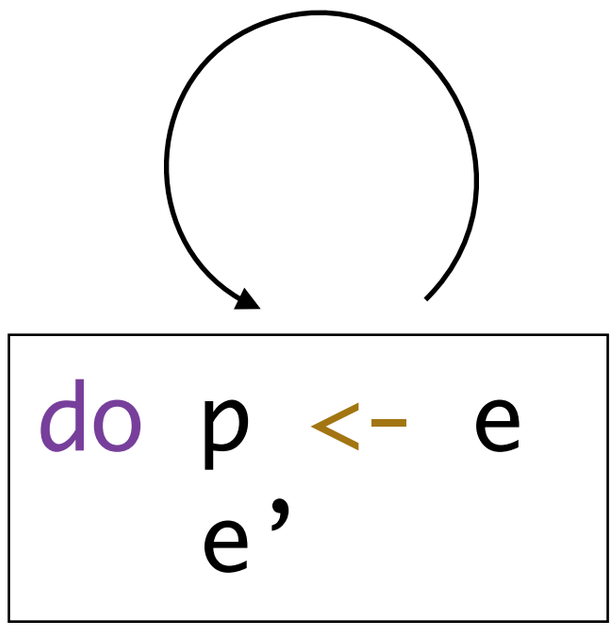
```
e >>= (\p -> e')
```

Uses Monad class

What about do? I want do!

```
Use {-# LANGUAGE RebindableSyntax #-}
```

Type checking

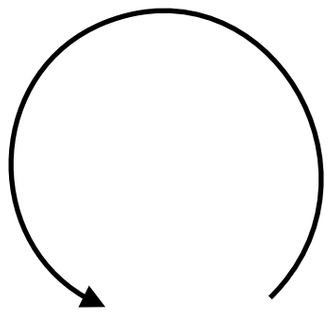


Uses Monad class

What about do? I want do!

```
Use {-# LANGUAGE RebindableSyntax #-}
```

Type checking



```
do p <- e
   e'
```

desugar

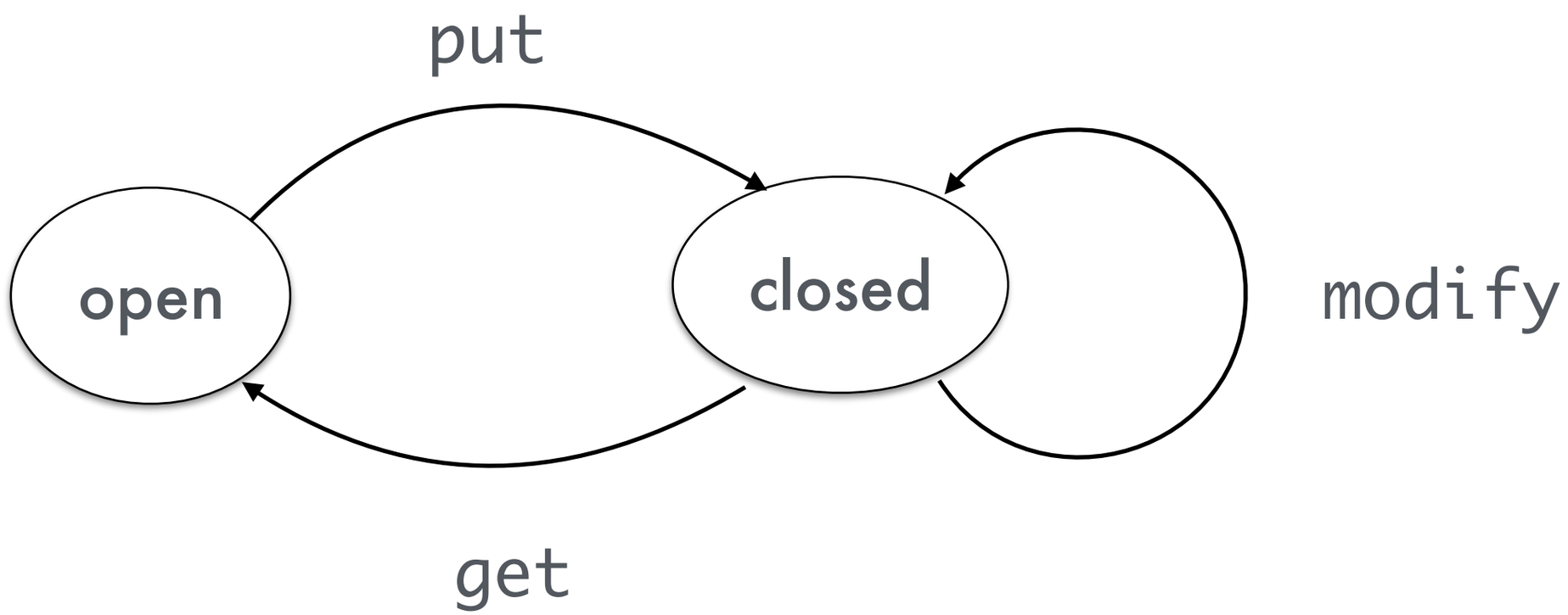
```
e >>= (\p -> e')
```

Use any `>>=` in scope

Code time!

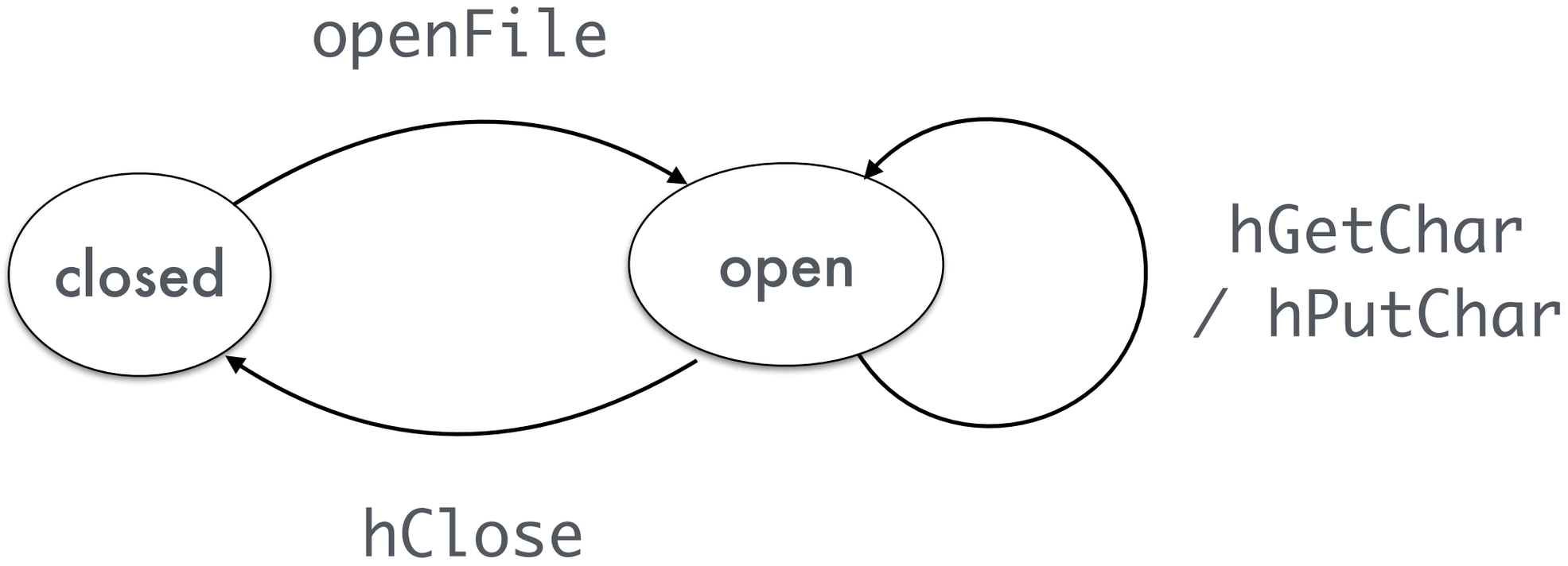
Simple protocols

AtomicState



Simple protocols

FileHandle



Parameterised monads

Protocols

- ➡ Atomic state
- ➡ Safe file handlers

Extensible, fine-grained state

Graded monads

Resource counting

More at: https://github.com/dorchard/effect-monad

Many more examples of both!

Thank you!

https://github.com/dorchard/haskellXbytes2017

@dorchard