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)
```

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

   ....    ....

               f :: ... -> T
```

version 1

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

```
somethingPure :: Int -> String

    ....    ....

f :: ... -> State Int T
```

version 1

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

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

  ....     ....

     f :: ... -> State Int T
```

version 1

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

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

....    ....

    f :: ... -> State Int T
```

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	Pure
`State Int String`	`String`
`IO String`	
But how impure is it?	

Impure	Pure

`State Int String` | `String`

Update Write Read Pure

Programs That Explain Their Effects

(in a fine-grained way!)

using types

and parameterised monads

and graded monads

Regular monads...

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

... replace....

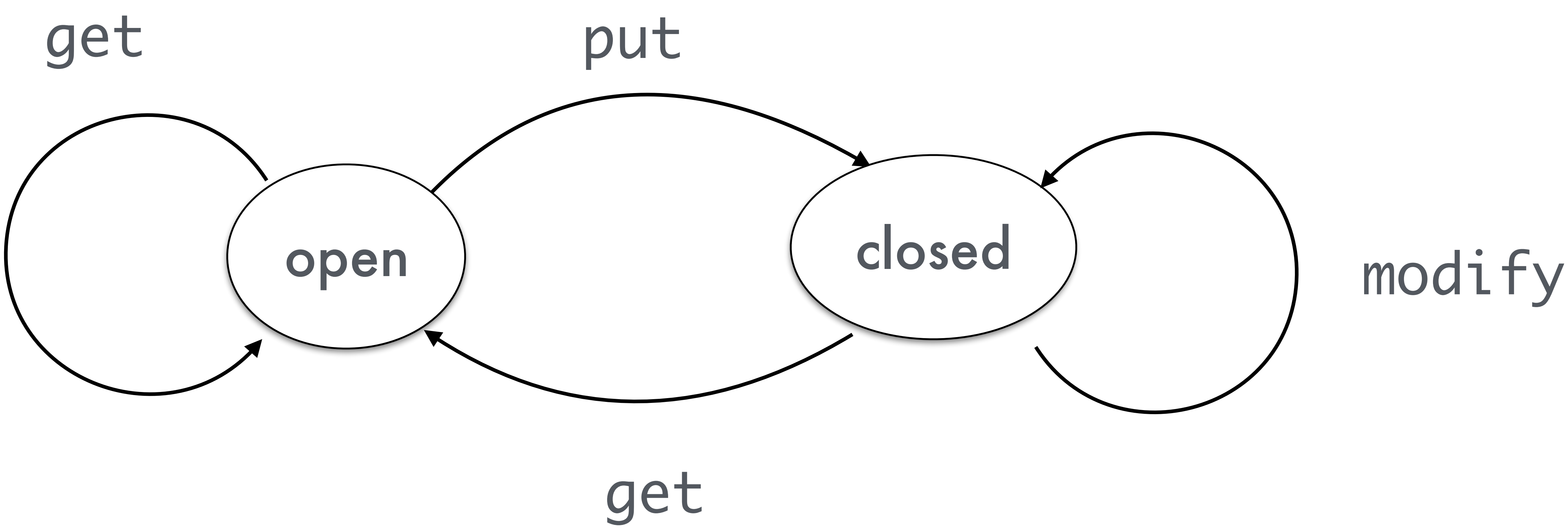
```
import Prelude hiding (Monad(..))
```

... 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
```

Simple protocols

AtomicState



What about do? I want do!

Usually…

Type checking

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

desugar

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

Uses Monad class

What about do? I want do!

Use `{-# LANGUAGE RebindableSyntax #-}`

Type checking

```
do p <- e
   e’
```

desugar

```
e >>= (\p -> e’)
```

Uses Monad class

What about do? I want do!

Use `{-# LANGUAGE RebindableSyntax #-}`

Type checking

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

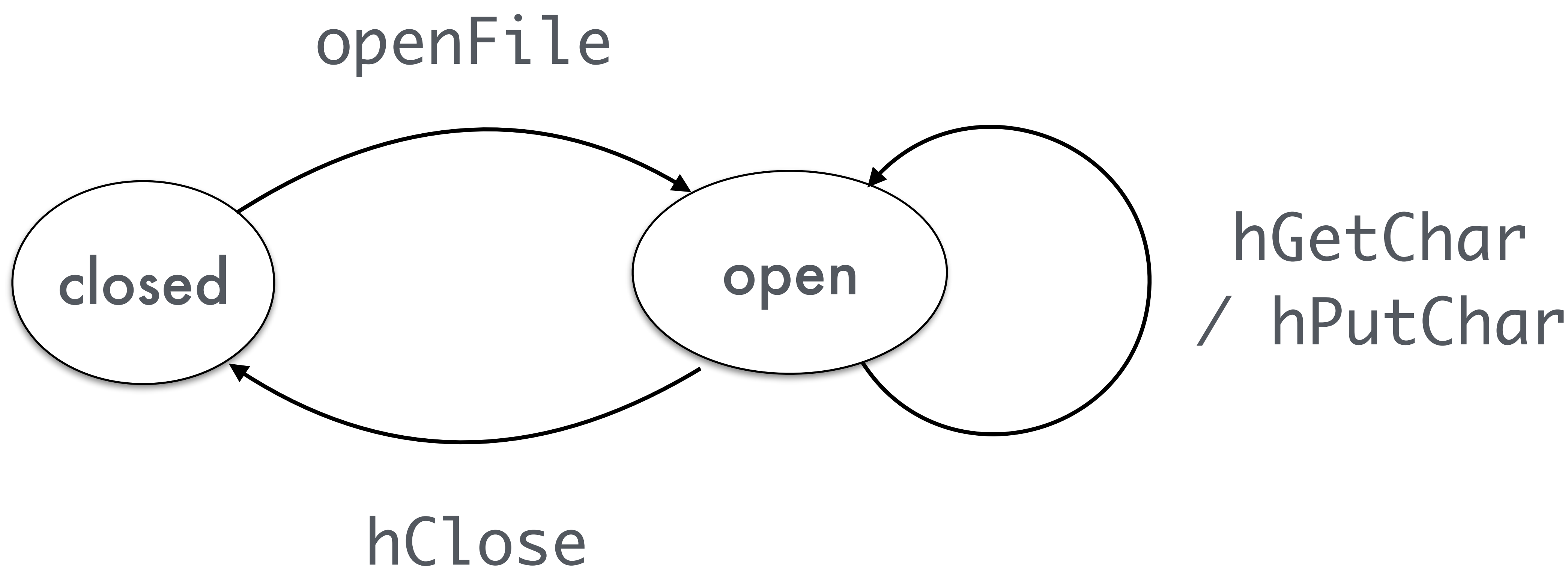
desugar

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

Use any `>>=` in scope

Simple protocols

FileHandle



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
```

(lots… Katsumata 2014, Orchard&Petricek 2014)

Graded monads

```
class GMonad (g :: k -> * -> *) where
  return :: a -> g Zero a
  (>>=) :: g x -> (a -> g y b) -> g (Plus x y) b
```

Type-level operations

```
Plus :: k -> k -> k
Zero :: k
```

The Essence of Graded Modality

Recall, monoid homomorphism $\quad (X, \oplus, I) \xrightarrow{F} (Y, \bullet, e)$

$$\begin{tikzcd} 1 \arrow[d, "I"'] \arrow[dr, bend left, "e"] & \\ X \arrow[r, "F"'] & Y \end{tikzcd} \qquad \begin{tikzcd} X \times X \arrow[r, "F \times F"] \arrow[d, "\oplus"'] & Y \times Y \arrow[d, "\bullet"] \\ X \arrow[r, "F"'] & Y \end{tikzcd}$$

Graded necessity, with $(\mathcal{X}, \oplus, I)$ is a **lax** monoid homomorphism

$$\begin{tikzcd} 1 \arrow[d, "I"'] \arrow[dr, bend left, "\mathsf{Id}"] \arrow[dr, phantom, "\Swarrow refl"] & \\ \mathcal{X} \arrow[r, "T"'] & \mathcal{C}^{\mathcal{C}} \end{tikzcd} \qquad \begin{tikzcd} \mathcal{X} \times \mathcal{X} \arrow[r, "T \times T"] \arrow[d, "\oplus"'] \arrow[dr, phantom, "\Swarrow trans"] & \mathcal{C}^{\mathcal{C}} \times \mathcal{C}^{\mathcal{C}} \arrow[d, "\circ"] \\ \mathcal{X} \arrow[r, "T"'] & \mathcal{C}^{\mathcal{C}} \end{tikzcd}$$

$$A \xrightarrow{refl} T_I A \qquad\qquad T_x T_y A \xrightarrow{trans} T_{x \oplus y} A$$

The Essence of Graded Modality

Graded necessity, with $(\mathcal{X}, \oplus, I)$ is a **lax** monoid homomorphism

$$\begin{tikzcd} 1 \arrow[d, "I"'] \arrow[dr, bend left, "\mathsf{Id}"] \arrow[d, phantom, shift left=6ex, "\overset{refl}{\swarrow}"] & \\ \mathcal{X} \arrow[r, "T"'] & \mathcal{C}^{\mathcal{C}} \end{tikzcd} \qquad \begin{tikzcd} \mathcal{X} \times \mathcal{X} \arrow[r, "T \times T"] \arrow[d, "\oplus"'] & \mathcal{C}^{\mathcal{C}} \times \mathcal{C}^{\mathcal{C}} \arrow[d, "\circ"] \arrow[dl, Rightarrow, "trans"'] \\ \mathcal{X} \arrow[r, "T"'] & \mathcal{C}^{\mathcal{C}} \end{tikzcd}$$

$$A \xrightarrow{refl} T_I A \qquad T_x T_y A \xrightarrow{trans} T_{x \oplus y} A$$

General graded modality is a **lax** functor (category homomorphism)

$$\begin{tikzcd} 1 \arrow[d, "id_P"'] \arrow[dr, bend left, "\mathsf{Id}"] \arrow[d, phantom, shift left=8ex, "\overset{refl}{\swarrow}"] & \\ \mathbb{C}(P,P) \arrow[r, "T"'] & \mathcal{C}^{\mathcal{C}} \end{tikzcd} \qquad \begin{tikzcd} \mathbb{C}(P,Q) \times \mathbb{C}(Q,R) \arrow[r, "T \times T"] \arrow[d, "\circ"'] & \mathcal{C}^{\mathcal{C}} \times \mathcal{C}^{\mathcal{C}} \arrow[d, "\circ"] \arrow[dl, Rightarrow, "trans"'] \\ \mathbb{C}(P,R) \arrow[r, "T"'] & \mathcal{C}^{\mathcal{C}} \end{tikzcd}$$

$$A \to T_{id_P} A \qquad T_f T_g A \to T_{g \circ f} A$$

Parameterised monads

Protocols

- ➡ Atomic state
- ➡ Safe file handlers

Extensible, fine-grained state

Graded monads

Resource counting

Security level

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

Many more examples of both!

Thank you!

https://github.com/dorchard/effectful-explanations-talk

https://hackage.haskell.org/package/effect-monad

@dorchard