

Programs That Explain Their Effects

Dominic Orchard

University of
Kent

“Haskell is the world’s finest
imperative programming language.”

Simon Peyton Jones

but.....

```
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)
```

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

```
myProgram :: State Int String
```

```
myProgram = do
```

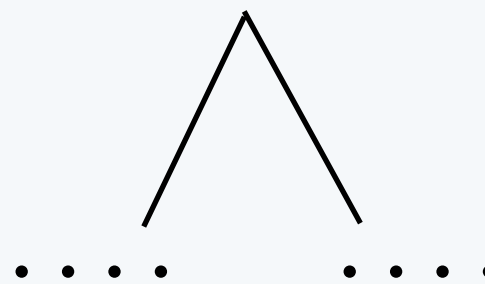
```
  x <- get
```

```
  let a = somethingPure x
```

```
  put (x+1)
```

```
  return (a ++ show x)
```

`somethingPure :: Int -> String`



`f :: ... -> T`

version 1

```
myProgram :: State Int String
```

```
myProgram = do
```

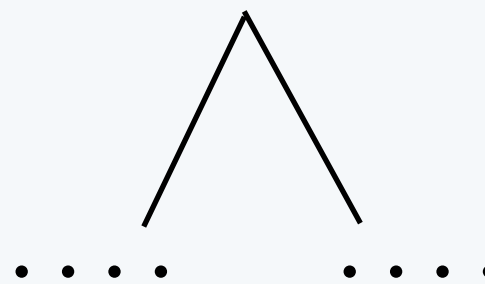
```
  x <- get
```

```
  let a = somethingPure x
```

```
  put (x+1)
```

```
  return (a ++ show x)
```

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



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

version 1 

```
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 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
```

```
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)
```

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


Impure

State Int String

IO String

But how impure is it?

Pure

String

Impure

State Int String

Update

Write

Read

Pure

Pure

String

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....

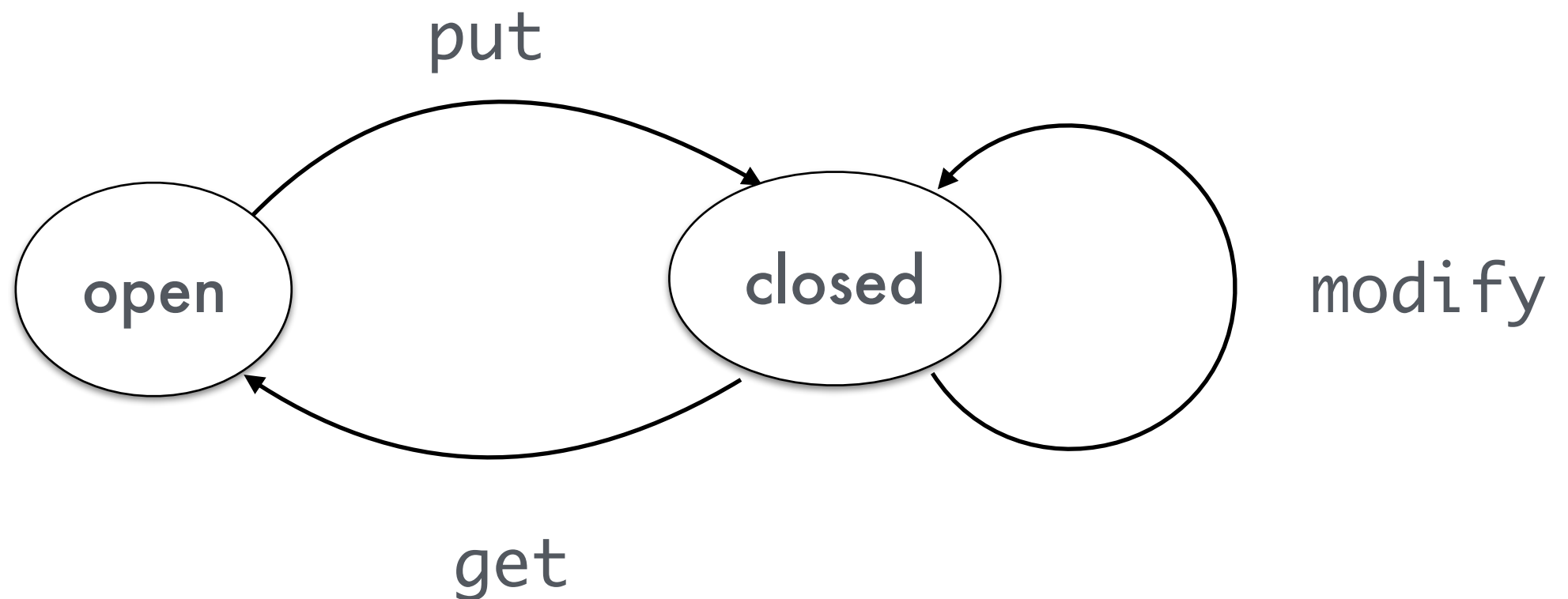
```
import Prelude hiding (Monad(..))
```

... 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

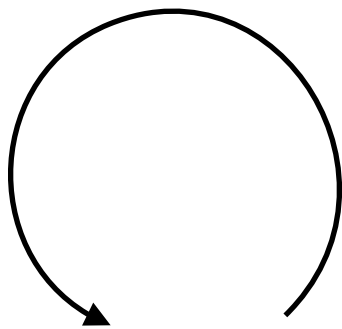
AtomicState



What about do? I want do!

Usually...

Type checking



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

desugar

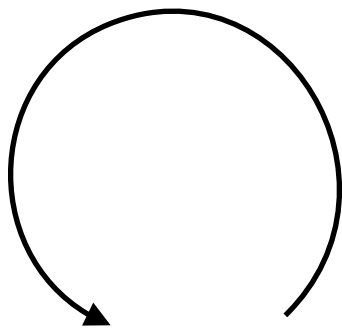
```
e >>= (\p -> e')
```

Uses **Monad** class

What about do? I want do!

Use `{-# LANGUAGE RebindableSyntax #-}`

Type checking



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

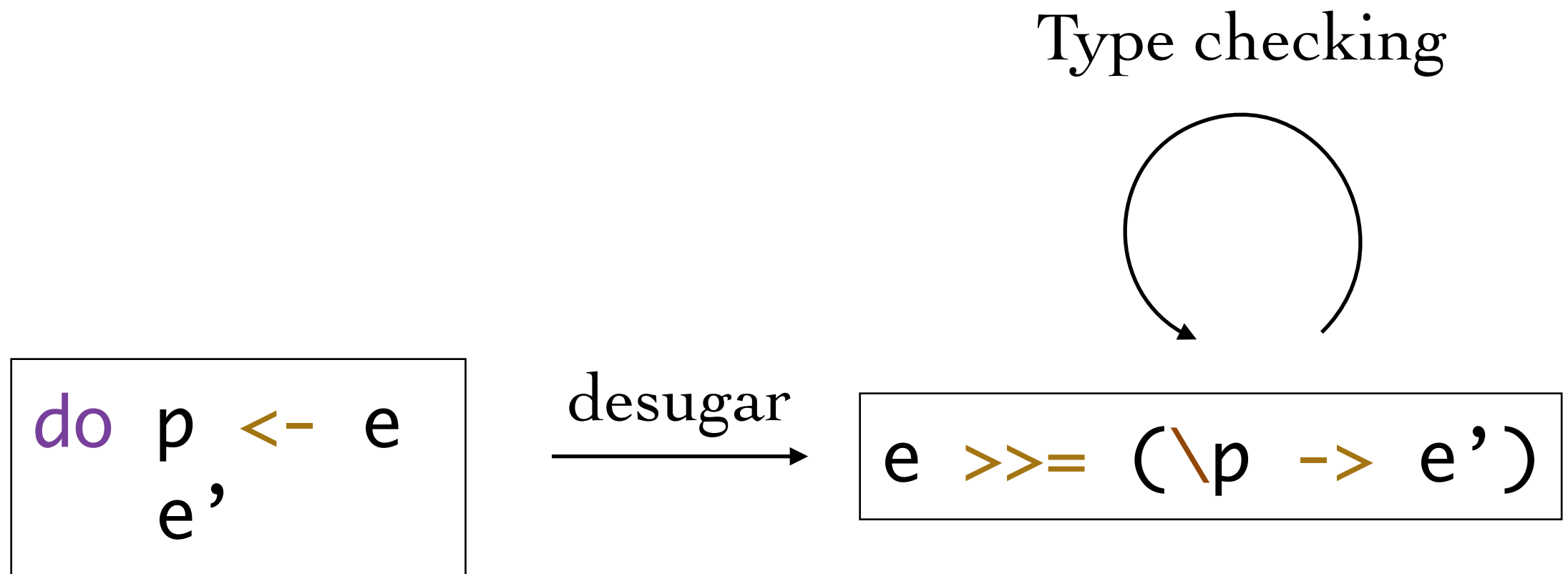
desugar

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

Uses **Monad** class

What about do? I want do!

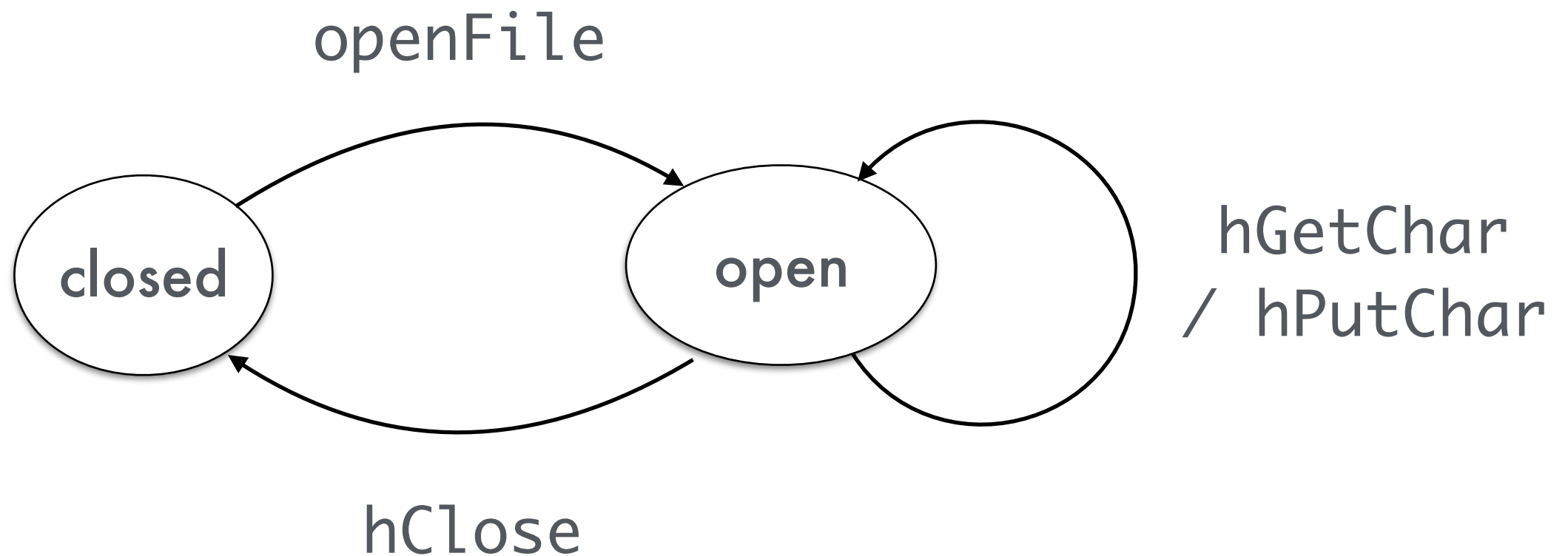
Use `{-# LANGUAGE RebindableSyntax #-}`



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
```

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/effectful-explanations-talk>

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



@dorchard