COMP3141

Liam O'Conn
CSE, UNSW (and Data6

Term 2 2019

Effects

Effects

Effects are observable phenomena from the execution of a program.

Effects

Effects

Effects are observable phenomena from the execution of a program.

Exa

```
in
... // read and write
*p = *p + 1;
```

Effects

Effects

Effects are observable phenomena from the execution of a program.

Exa


```
in
... // read and write
*p = *p + 1;
```

Effects

Effects

Effects are observable phenomena from the execution of a program.

Exa

```
in
... // read and write
*p = *p + 1;
```


E

```
// exception effect
throw new Exception();
```

Effects

Effects

Effects are observable phenomena from the execution of a program.

Exa

```
in
... // read and write
*p = *p + 1;
```


Example (Non-termination)

```
// infinite loop
while (1) {};
```

E

```
// exception effect
throw new Exception();
```

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function.
Internal effects

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function.
Internal effects

Example (Exte

Console, file and network I/O; termination and non-termin
etc.

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function.
Internal effects

Example (Exte

Console, file and network I/O; termination and non-termin
etc.

Are memory effects *external* or *internal*?

Internal vs. External Effects

External Observability

An *external* effect is an effect that is observable outside the function.
Internal effects

Example (Exte

Console, file and network I/O; termination and non-termin
etc.

Are memory effects *external* or *internal*?

Answer: Depends on the scope of the memory being accessed. Global variable accesses are *external*.

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function*

`a -> b` is *fully* … `a` to the codomain type …

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function*

`a -> b` is *fully* `a` to the

codomain type

Consequences:

- Two invocations with the same arguments result in the sa

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function*

`a -> b` is *fully* a to the

codomain type

Consequences:

- Two invocations with the same arguments result in the sa
- No observable trace is left beyond the result of the function

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function*

a -> b is *fully* ... a to the codomain type

Consequences:

- Two invocations with the same arguments result in the sa
- No observable trace is left beyond the result of the function
- No implicit notion of time or order of execution.

Purity

A function with no external effects is called a *pure* function.

Pure functions

A *pure function*

`a -> b` is *fully* … `a` to the codomain type …

Consequences:

- Two invocations with the same arguments result in the sa…
- No observable trace is left beyond the result of the function
- No implicit notion of time or order of execution.

Question: Are Haskell functions pure?

Haskell Functions

Haskell functions are technically not pure.

Haskell Functions

Haskell functions are technically not pure.

- They can loop infinitely.

Haskell Functions

Haskell functions are technically not pure.

- They can loop infinitely.
- They can thr

Haskell Functions

Haskell functions are technically not pure.

- They can loop infinitely.
- They can thr
- They can fo

Haskell Functions

Haskell functions are technically not pure.

- They can loop infinitely.
- They can thr
- They can fo

Caveat

Purity only applies to a particular level of abstraction. Even igno assembly instructions produced by GHC aren't really pure.

Haskell Functions

Haskell functions are technically not pure.

- They can loop infinitely.
- They can thr
- They can fo

Caveat

Purity only applies to a particular level of abstraction. Even igno assembly instructions produced by GHC aren't really pure.

Despite the impurity of Haskell functions, we can often reason as though they are pure. Hence we call Haskell a purely functional language.

The Danger of Implicit Side Effects

- They introduce (often subtle) requirements on the evaluation order.

The Danger of Implicit Side Effects

- They introduce (often subtle) requirements on the evaluation order.
- They are not v

The Danger of Implicit Side Effects

- They introduce (often subtle) requirements on the evaluation order.
- They are not v
- They introd

increasing

The Danger of Implicit Side Effects

- They introduce (often subtle) requirements on the evaluation order.
- They are not v
- They introd
  increasing
- They interfere badly with strong typing, for example mu
  reference types in ML.

The Danger of Implicit Side Effects

- They introduce (often subtle) requirements on the evaluation order.
- They are not v
- They introd
  increasing
- They interfere badly with strong typing, for example mu
  reference types in ML.

We can't, in general, reason equationally about effectful prog

Can we program with pure functions?

Yes! We've been doing it for the past 6 weeks.

Can we program with pure functions?

Yes! We've been doing it for the past 6 weeks.

Typically, a computation involving some state of type `s` and returning a result of type `a` can be expresse

Can we program with pure functions?

Yes! We've been doing it for the past 6 weeks.

Typically, a computation involving some state of type `s` and returning a result of type `a` can be expresse

Rather than change the state, we return a new copy of the state.

Can we program with pure functions?

Yes! We've been doing it for the past 6 weeks.

Typically, a computation involving some state of type `s` and returning a result of type `a` can be expresse

Rather than change the state, we return a new copy of the state.

Efficiency?

All that copying might seem expensive, but by using tree data structures, we can usually reduce the cost to an $\mathcal{O}(\log n)$ overhead.

State Passing

Example (Labelling Nodes)

```
data Tree a = Branch a (Tree a) (Tree a) | Leaf
```

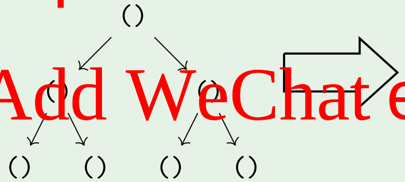
State Passing

Example (Labelling Nodes)


```haskell
data Tree a = Branch a (Tree a) (Tree a) | Leaf
```

Given a tree, label ea

```haskell
label :: Tree () -> T
```

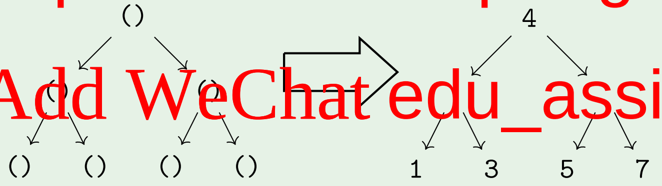
State Passing

Example (Labelling Nodes)


```
data Tree a = Branch a (Tree a) (Tree a) | Leaf
```

Given a tree, label ea

```
label :: Tree () -> T
```

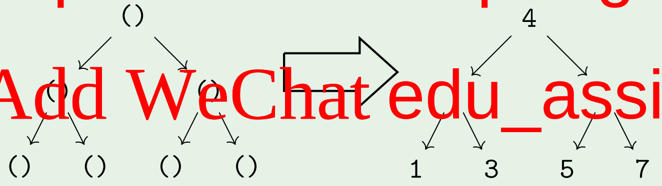
State Passing

Example (Labelling Nodes)

```
data Tree a = Branch a (Tree a) (Tree a) | Leaf
```

Given a tree, label ea

```
label :: Tree () -> T
```

Let's use a data type to simplify this!

State

`newtype State s a` = A procedure that, manipulating some state of type `s`, returns `a`

State

newtype State s a = A procedure that, manipulating some state of type s, returns a

State Operations

```
get :: State s s
put :: s -> State s ()
pure :: a -> State s a
evalState :: State s a -> s -> a
```

State

`newtype State s a` = A procedure that, manipulating some state of type `s`, returns `a`

State Operations

```
get :: State s s
put :: s -> State s ()
pure :: a -> State s a
evalState :: State s a -> s -> a
```

Sequential Composition

do blocks:

```
(>>) :: St
```

State

`newtype State s a` = A procedure that, manipulating some state of type s, returns a

State Operations

```
get :: State s s
put :: s -> State s ()
pure :: a -> State s a
evalState :: State s a -> s -> a
```

Sequential Composition

do blocks:


```
(>>) :: St
```


Bind

The 2nd step can depen

```
do x <- get
   pure (x+1)
```

desugars $\Longrightarrow$

```
get >>= \x -> pure (x + 1)
```

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

State

`newtype State s a` = A procedure that, manipulating some state of type `s`, returns a

State Operations

```haskell
get :: State s s
put :: s -> State s ()
pure :: a -> State s a
evalState :: State s a -> s -> a
```

Sequential Composition

do blocks:


```haskell
(>>) :: St
```


Example

Implement `modify`:

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

And re-do the tree labelling.

Bind

The 2nd step can depen

```haskell
do x <- get
   pure (x+1)
```

desugars $\Longrightarrow$

```haskell
get >>= \x -> pure (x + 1)
```

```haskell
(>>=) :: State s a -> (a -> State s b) -> State s b
```

State Implementation

The State type is essentially implemented as the same state passing we did before!

```
newtype State s a = State (s -> (s,a))
```

Example

Let's implement e

Caution

In the Haskell standard library mtl, the State t ntly differently, but the implementation essentially works the same way.

Effects

Sometimes we need side effects.

- We need to pe
- We might ne
  (but usuall i

Effects

Sometimes we need side effects.

- We need to pe
- We might ne
  (but usuall i

Haskell's approach

Pure by default. Effectful when necessary.

The IO Type

A procedure that performs some side effects, returning a result of type a is written as IO a

The IO Type

A procedure that performs some side effects, returning a result of type `a` is written as `IO a`.

World interpretation

`IO a` is an abstra

(that's how it's implemented in GHC)

The IO Type

A procedure that performs some side effects, returning a result of type a is written as IO a.

World interpretation

IO a is an abstra

(that's how it's implemented in GHC)


```
(>>=) :: IO a -> (a -> IO b) -> IO b
pure  :: a -> IO a

getChar :: IO Char
readLine :: IO String
putStrLn :: String -> IO ()
```

Infectious IO

We can convert pure values to impure procedures with `pure`:

```
pure :: a -> IO a
```

Infectious IO

We can convert pure values to impure procedures with `pure`:

```
pure :: a -> IO a
```

But we can't convert impure procedures to pure values:

```
???? :: IO a -> a
```

Infectious IO

We can convert pure values to impure procedures with `pure`:

```
pure :: a -> IO a
```


But we can't convert impure procedures to pure values:

```
???? :: IO a -> a
```

The only function t


```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

But it returns an `IO` procedure as well.

Infectious IO

We can convert pure values to impure procedures with `pure`:


```
pure :: a -> IO a
```

But we can't convert impure procedures to pure values:

```
???? :: IO a -> a
```

The only function t


```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

But it returns an `IO` procedure as well.

Conclusion

The moment you use an `IO` procedure in a function, `IO` shows up in the types, and you can't get rid of it!

If a function makes use of `IO` effects directly or indirectly, it will have `IO` in its type!

Haskell Design Strategy

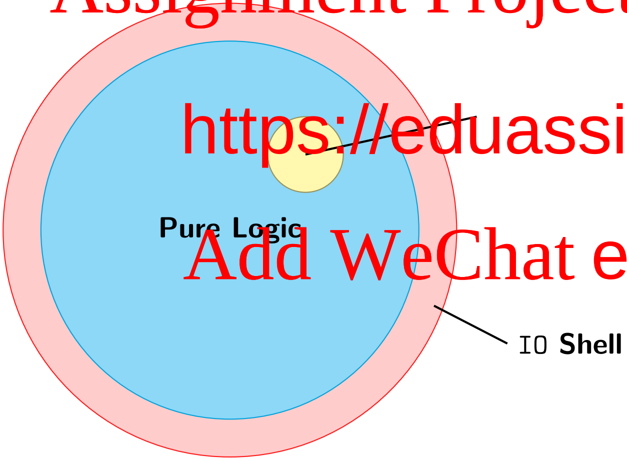
We ultimately "run" `IO` procedures by calling them from `main`:

```
main :: IO ()
```

Haskell Design Strategy

We ultimately "run" `IO` procedures by calling them from `main`:

```
main :: IO ()
```

Examples

Example (Triangles)

Given an input nu n.

Examples

Example (Triangles)

Given an input nu n.

Example (Maz

Design a game that reads in a $n \times n$ maze from a fil
$(0, 0)$ and must reach position $(n-1, n-1)$ to win ut
to move the player around the maze.

Benefits of an IO Type

- Absence of effects makes type system more informative:
  - A type si
  - All de
  - All de

Benefits of an IO Type

- Absence of effects makes type system more informative:
  - A type si
  - All de
  - All de

- It is easier to reason about pure code and it is easier to test:
  - Testing is local, doesn't require complex set-up and tea
  - Reasoning is local, doesn't require state invariants.
  - Type checking leads to strong guarantees.

Mutable Variables

We can have honest-to-goodness mutability in Haskell, if we really need it, using `IORef`.

```haskell
data IORef a
newIORef :: a -> IO (
readIORef :: IO
writeIORef :: IORef a -> a -> IO ()
```

Mutable Variables

We can have honest-to-goodness mutability in Haskell, if we really need it, using IORef.

```
data IORef a
newIORef :: a -> IO (
readIORef :: IO
writeIORef :: IORef a -> a -> IO ()
```


Example (Effectful Average)

Average a list of numbers using IORefs.

Mutable Variables, Locally

Something like averaging a list of numbers doesn't require external effects, even if we use mutation internally.

Mutable Variables, Locally

Something like averaging a list of numbers doesn't require external effects, even if we use mutation internally.


```
data STRef s a
newSTRef :: a -> ST (
readSTRef :: ST
writeSTRef :: S
runST :: (forall s. ST s a) -> a
```

Mutable Variables, Locally

Something like averaging a list of numbers doesn't require external effects, even if we use mutation internally.


```
data STRef s a
newSTRef :: a -> ST (
readSTRef :: ST
writeSTRef :: S
runST :: (forall s. ST s a) -> a
```


The extra s parameter is called a state thread, that ensures don't leak outside of the ST computation.

Mutable Variables, Locally

Something like averaging a list of numbers doesn't require external effects, even if we use mutation internally.


```
data STRef s a
newSTRef :: a -> ST (
readSTRef :: ST
writeSTRef :: S
runST :: (forall s. ST s a) -> a
```


The extra s parameter is called a state thread, that ensures don't leak outside of the ST computation.

Note

The ST type is not assessable in this course, but it is useful sometimes in Haskell programming.

QuickChecking Effects

QuickCheck lets us test `IO` (and `ST`) using this special property monad interface:


```
monadicIO :: PropertyM IO () -> Property
pre       :: Bool -> PropertyM IO ()
assert    :: Bool -> Prop
run       :: IO a -> PropertyM IO a
```


Do notation and s[illegible]st as with State s and IO procedures.

QuickChecking Effects

QuickCheck lets us test IO (and ST) using this special property monad interface:


```
monadicIO :: PropertyM IO () -> Property
pre       :: Bool -> PropertyM IO ()
assert    :: Bool -> Prop
run       :: IO a -> PropertyM IO a
```


Do notation and s... st as with State s and IO procedures.

Example (Testing average)

Let's test that our IO average function works like the n

QuickChecking Effects

QuickCheck lets us test `IO` (and `ST`) using this special property monad interface:


```
monadicIO :: PropertyM IO () -> Property
pre       :: Bool -> PropertyM IO ()
assert    :: Bool -> Prop
run       :: IO a -> PropertyM IO a
```


Do notation and s... st as with State s and `IO` procedures.

Example (Testing `average`)

Let's test that our `IO` average function works like the n

Example (Testing `gfactor`)

Let's test that the GNU `factor` program works correctly!

Homework

1. New exerci
2. Last week's
3. This week's quiz is due the Friday after the following Frida