COMP3141

Software System Design and Implementation

Effects and State

Liam O'Connor
CSE, UNSW (and Data61)

Term 2 2019

Effects

Effects

Effects are observable phenomena from the execution of a program.

Example (Memory effects)

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

Example (IO)

```
// console IO
c = getchar();
printf("%d",32);
```

Example (Non-termination)

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

Example (Control flow)

```
// 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 are not observable from outside.

Example (External effects)

Console, file and network I/O; termination and non-termination; non-local control flow; 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* is the mathematical notion of a function. That is, a function of type `a -> b` is *fully* specified by a mapping from all elements of the domain type `a` to the codomain type `b`.

Consequences:

- Two invocations with the same arguments result in the same value.
- 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.

- They can loop infinitely.
- They can throw exceptions (partial functions).
- They can force evaluation of unevaluated expressions.

Caveat

Purity only applies to a particular level of abstraction. Even ignoring the above, 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.
- They are not visible from the type signature of the function.
- They introduce non-local dependencies which is bad for software design, increasing *coupling*.
- They interfere badly with strong typing, for example mutable arrays in Java, or reference types in ML.

We can't, in general, reason equationally about effectful programs!

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 expressed as a function:

```
s -> (s, a)
```

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.

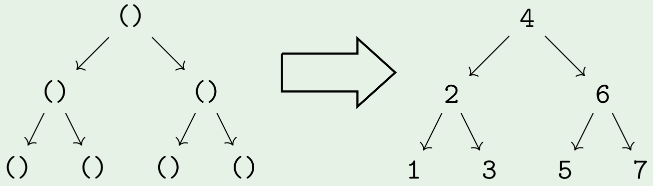
State Passing

Example (Labelling Nodes)

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

Given a tree, label each node with an ascending number in infix order:

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



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 Operations

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

Sequential Composition

Do one state action after another with `do` blocks:

```haskell
do put 42          desugars    put 42 >> put True
   pure True       ==>
```

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

Example

Implement `modify`:

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

And re-do the tree labelling.

Bind

The 2nd step can depend on the first with bind:

```haskell
do x <- get        desugars    get >>= \x -> pure (x + 1)
   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!

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

Example

Let's implement each of the `State` operations for this newtype.

Caution

In the Haskell standard library `mtl`, the `State` type is actually implemented slightly differently, but the implementation essentially works the same way.

Effects

Sometimes we need side effects.

- We need to perform I/O, to communicate with the user or hardware.
- We might need effects for maximum efficiency.
  (but usually internal effects are sufficient)

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`.

World interpretation

`IO a` is an abstract type. But we can think of it as a function:

```
RealWorld -> (RealWorld, a)
```

(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
```

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

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

The only function that gets an a from an `IO a` is >>=:

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

But it returns an `IO` procedure as well.

Conclusion

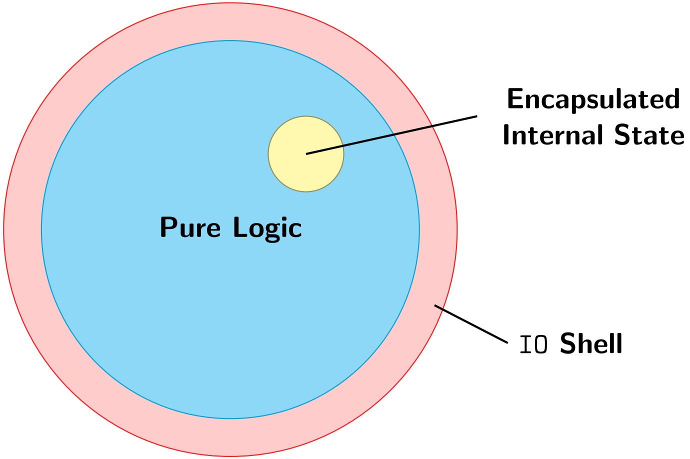
The moment you use an `IO` procedure in a function, `IO` shows up in the types, and you can't get rid of it!

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`:

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



Examples

Example (Triangles)

Given an input number n, print a triangle of $*$ characters of base width n.

Example (Maze Game)

Design a game that reads in a $n \times n$ maze from a file. The player starts at position $(0,0)$ and must reach position $(n-1, n-1)$ to win. The game accepts keyboard input to move the player around the maze.

Benefits of an IO Type

- Absence of effects makes type system more informative:
  - A type signatures captures entire interface of the function.
  - All dependencies are explicit in the form of data dependencies.
  - All dependencies are typed.
- It is easier to reason about pure code and it is easier to test:
  - Testing is local, doesn't require complex set-up and tear-down.
  - 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 (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Example (Effectful Average)

Average a list of numbers using `IORef`s.

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

The extra `s` parameter is called a state thread, that ensures that mutable variables 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 -> PropertyM IO ()
run       :: IO a -> PropertyM IO a
```

Do notation and similar can be used for `PropertyM IO` procedures just as with `State s` and `IO` procedures.

Example (Testing `average`)

Let's test that our `IO` average function works like the non-effectful one.

Example (Testing `gfactor`)

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

Homework

1. New exercise out, due the week after next week.
2. Last week's quiz is due on Friday.
3. This week's quiz is due the Friday after the following Friday.