COMP3141

Software System Design and Implementation

Lecture 7: Monads, IO

Johannes Åman Pohjola
University of New South Wales
Term 2 2023

Announcements

As of Monday, Assignment 2 is out!
Due 04 Aug 2023

Two Weeks of Monads

- So far we've met `Maybe`, `State`, and `List`.

Two Weeks of Monads

- So far we've met `Maybe`, `State`, and `List`.
- They allowed us to abstract away repetitive code:
    - `Maybe`: constantly case-checking for `Nothing`
    - `State`: manually threading through state

Two Weeks of Monads

- So far we've met `Maybe`, `State`, and `List`.
- They allowed us to abstract away repetitive code:
  - `Maybe`: constantly case-checking for `Nothing`
  - `State`: manually threading through state
- They allowed us to more easily write common code:
  - `List`: Exploring all possible choices
  - `List`: Building solutions by backtracking
  - `List`: List comprehensions

The Monad Type Class

All of these (seemingly different) things are instances of the same abstract concept: a *monad*.

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

The Haskell community uses monads to solve many system design problems, including but not limited to the ones we've seen.

The original purpose

Monadic I/O

In Oct 1992, Simon Peyton Jones and Philip Wadler presented a new model, based on monads, for **performing input and output** in pure functional languages such as Haskell.

The original purpose

Monadic I/O

In Oct 1992, Simon Peyton Jones and Philip Wadler presented a new model, based on monads, for **performing input and output** in pure functional languages such as Haskell.

We still haven't done any I/O!
Now that we know a few examples of monads, we'll be able to understand how to use monads to do input/output, and what problems this solves.

Monads

Recall how two weeks ago we defined our own State type and monad using

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

State Operations

```
get :: State s s
put :: s -> State s ()
return :: a -> State s a
(>>=) :: State s a -> (a -> State s b) -> State s b
evalState :: State s a -> s -> a
```

Monads

Recall how two weeks ago we defined our own State type and monad using

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

State Operations

```
get :: State s s
put :: s -> State s ()
return :: a -> State s a
(>>=) :: State s a -> (a -> State s b) -> State s b
evalState :: State s a -> s -> a
```

We need to perform I/O, to communicate with the user and with the hardware. A State-like monad will allow us to do this.

The IO Type

`IO a` is a procedure that may perform side effects, and returns a result of type `a`.

The IO Type

`IO a` is a procedure that may perform side effects, and returns a result of type `a`.

World interpretation

`IO a` will be an abstract type. But what if we thought of it as a function:

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

We can! This was Jones' and Wadler's original idea. And if we do, we get a monad. (that's close to how it's implemented in GHC)

The IO Type

IO a is a procedure that may perform side effects, and returns a result of type a.

World interpretation

IO a will be an abstract type. But what if we thought of it as a function:

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

We can! This was Jones' and Wadler's original idea. And if we do, we get a monad. (that's close to how it's implemented in GHC)

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

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

Infectious IO

The `RealWorld` type is purely abstract. You can't `get` or `put` it.
We can convert values to procedures with `return`:

```
return :: a -> IO a
```

This is a procedure that returns the given value, and does nothing else.

Infectious IO

The `RealWorld` type is purely abstract. You can't `get` or `put` it.
We can convert values to procedures with `return`:

```
return :: a -> IO a
```

This is a procedure that returns the given value, and does nothing else. But we can't convert procedures to pure values:

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

Infectious IO

The `RealWorld` type is purely abstract. You can't get or put it. We can convert values to procedures with `return`:

```
return :: a -> IO a
```

This is a procedure that returns the given value, and does nothing else. But we can't convert 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.

Infectious IO

The `RealWorld` type is purely abstract. You can't get or put it. We can convert values to procedures with `return`:

```
return :: a -> IO a
```

This is a procedure that returns the given value, and does nothing else. But we can't convert 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!

Equational Reasoning

Demos: Hello World, Referential Transparency, Equational Reasoning

Haskell Design Strategy

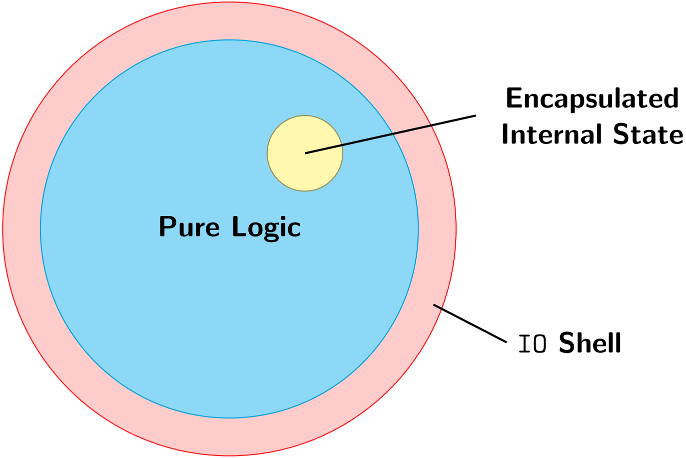
We ultimately “run” `IO` procedures by calling them from `main`:

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

Haskell Design Strategy

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

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



Examples

Example (Triangles)

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

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 undeclared effects (i.e. side effects) makes the type system more informative:
  - Type signatures capture the entire interface of a function.
  - All dependencies are explicit in the form of data dependencies.
  - All dependencies are typed.

Benefits of an IO Type

- Absence of undeclared effects (i.e. side effects) makes the type system more informative:
  - Type signatures capture the entire interface of a function.
  - All dependencies are explicit in the form of data dependencies.
  - All dependencies are typed.
- Equational reasoning works, and code 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.

The Either Monad

```
data Either a b = Left a | Right b
```

The `Either` type represents values with two possibilities: a value of type `Either a b` is either `Left a` or `Right b`.

This type is sometimes used to represent a value which is either correct or an error; by convention, the `Left` constructor is used to hold an error value and the `Right` constructor is used to hold a correct value (mnemonic: "right" also means "correct"). **Demo**

FIN

1. **Thanks!**