Why is a Monad Like a Writing Desk?

EdgeCase SOFTWARE ARTISANS @carinmeier

Monad

Monad

Monad

Monad

Monad

Monad

Monad

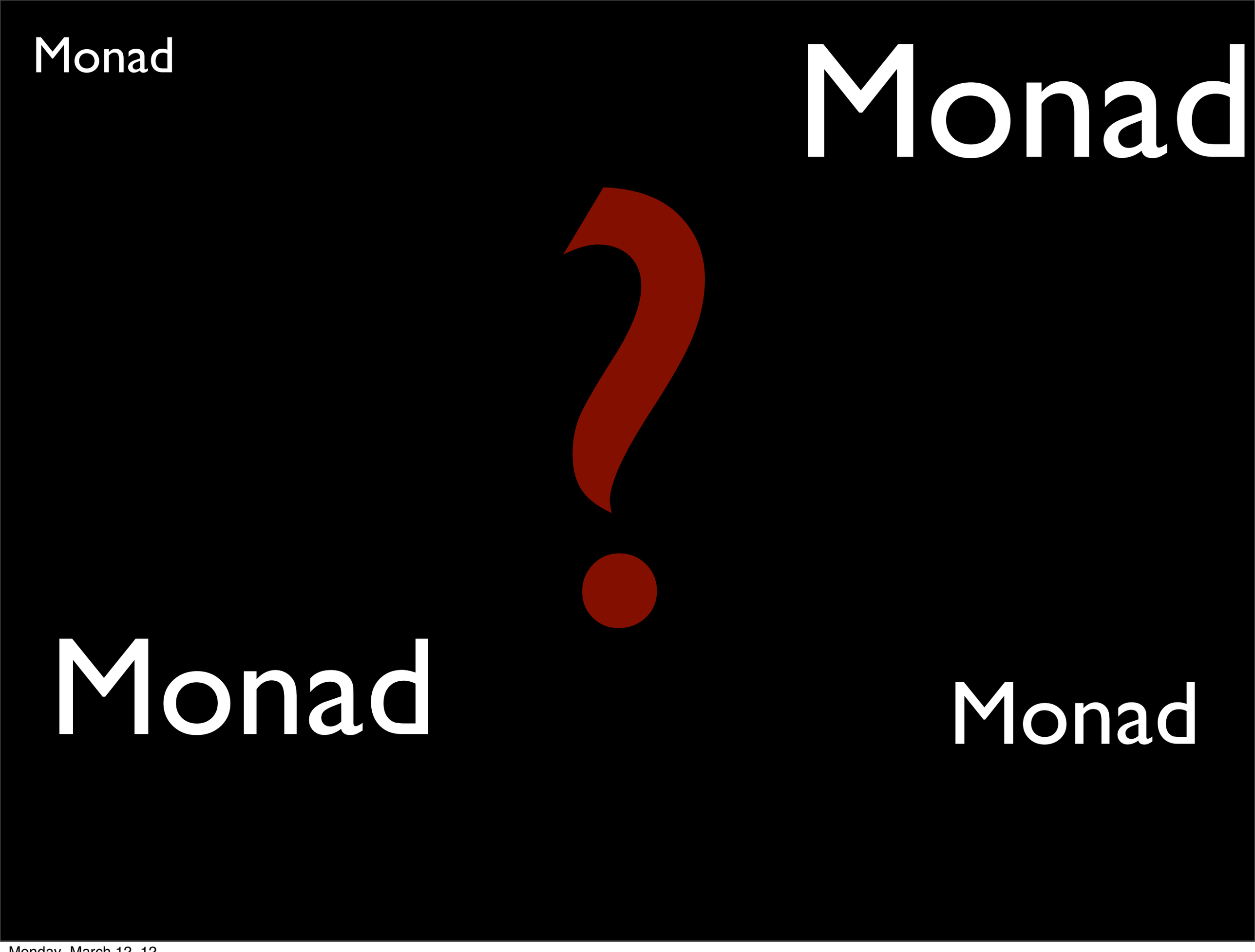
Monad

Monad

Monad

Monad

Monad



Monad

Monad

Monday, March 12, 12

Comprehending Monads

Philip Wadler
University of Glasgow

Monads for functional programming

Philip Wadler, University of Glasgow⋆

Department of Computing Science, University of Glasgow, G12 8QQ, Scotland
(wadler@dcs.glasgow.ac.uk)

“Shall I be pure or impure?”

- Wadler

Pure

Lazy Evaluation
Equational Reasoning

Impure

State
Error Handling
Input/ Output

Pure

Haskell

Impure

Clojure
Scheme

"Is there a way to combine the indulgences of impurity and the blessings of purity?"

- Wadler

Engenio Moggi

Engenio Moggi

Proposed way of structuring semantics
to describe state, exceptions, etc in programming

Engenio Moggi

Proposed way of structuring semantics
to describe state, exceptions, etc in programming

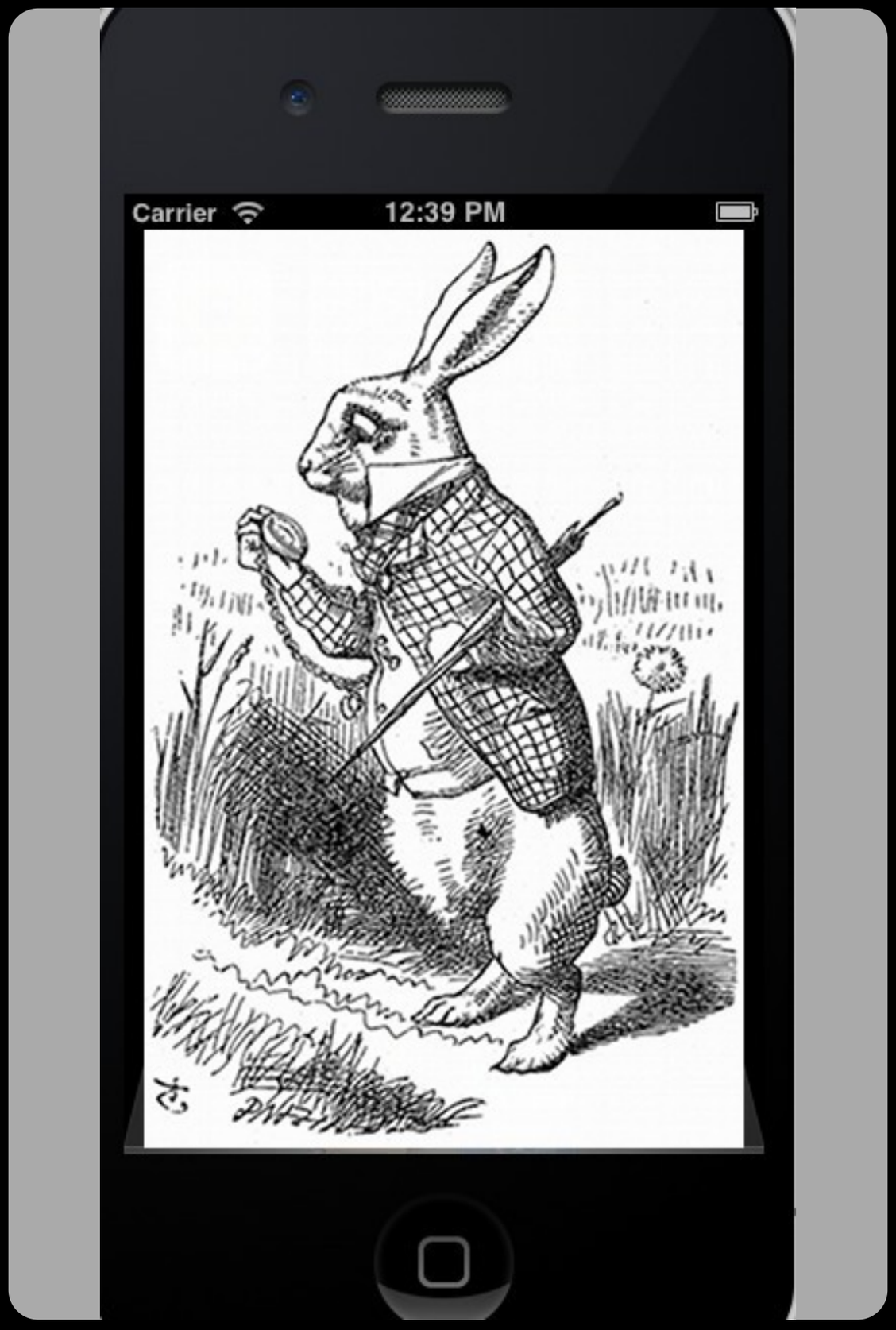
Category Theory: Monads

Monad
(Category Theory)

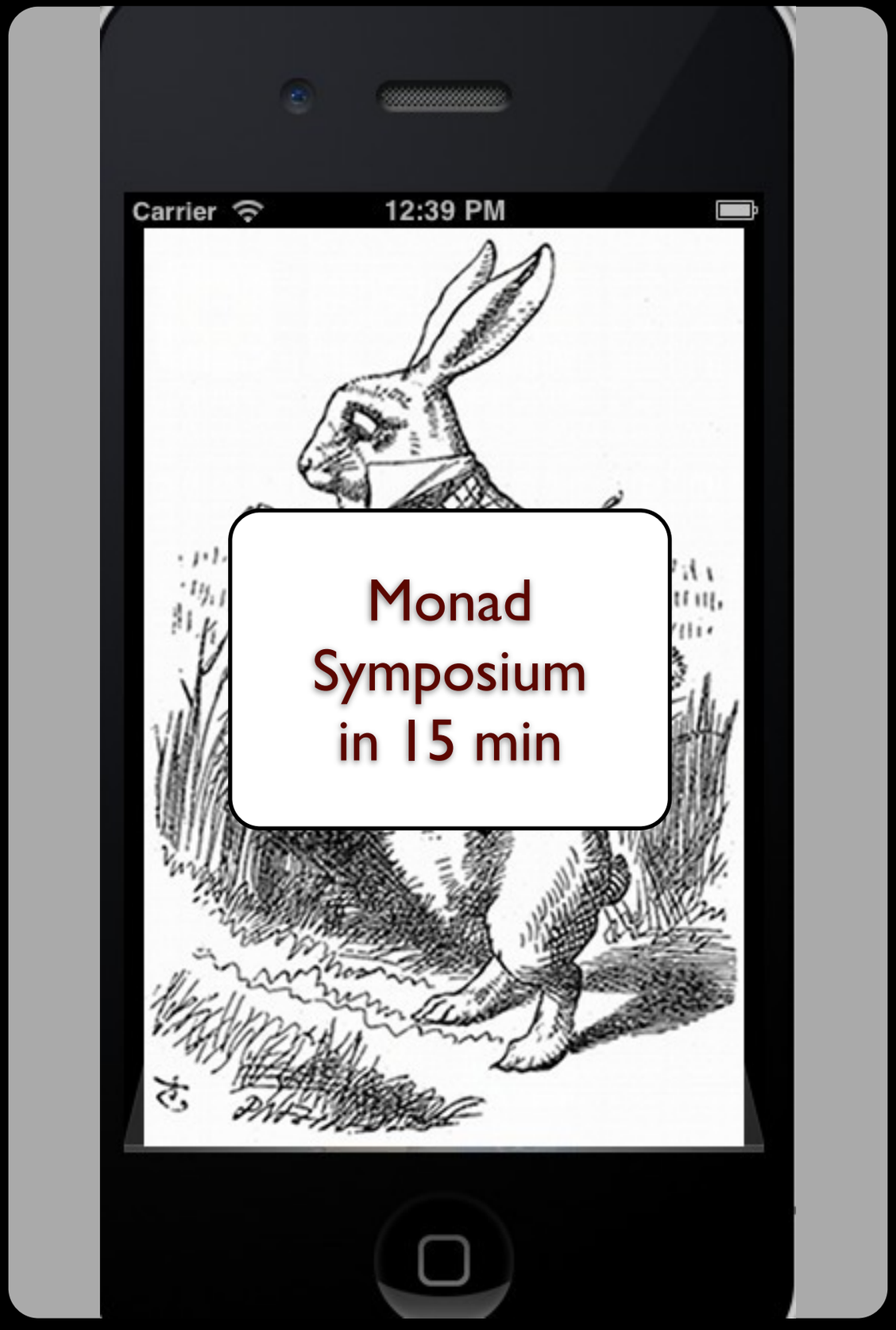
Also known as a Kleisi triple or triple

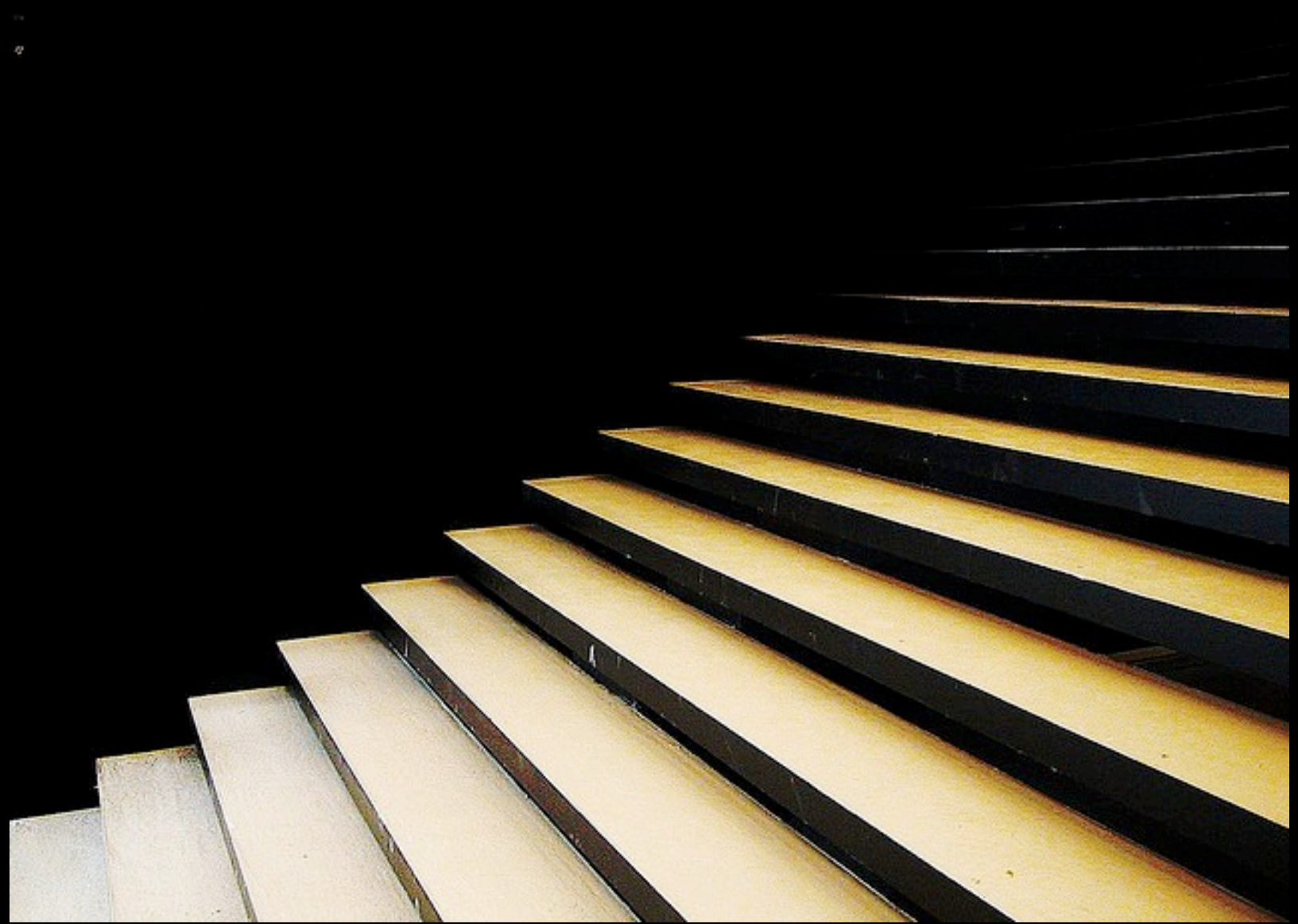
Is an endofunctor, together with two natural transformations

Used in theory of pairs of adjoint functions


Carrier
12:39 PM


Carrier
12:39 PM
You Are Late


Carrier
12:39 PM
Monad
Symposium
in 15 min



$$
\begin{aligned}
&\textbf{type}\ \ M\ a = a \\
&\\
&unit \quad\quad\quad :: a \to I\ a \\
&unit\ a \quad\quad\quad = a \\
&\\
&(\star) \quad\quad\quad :: M\ a \to (a \to M\ b) \to M\ b \\
&a \star k \quad\quad\quad = k\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{type}\ \ M\ a = a \\
&unit \qquad\quad :: a \to I\ a \\
&unit\ a \qquad\quad = a \\
&(\star) \qquad\quad :: M\ a \to (a \to M\ b) \to M\ b \\
&a \star k \qquad\quad = k\ a
\end{aligned}
$$

A monad is just a monoid in the category of endofunctors,

What's the problem?

$$\textbf{type}\ \ M\ a = a$$

$$unit \qquad :: a \to I\ a$$

$$unit\ a \qquad = a$$

$$(\star) \qquad \to M\ b$$

$$a \star k$$

Purity has its regrets.

onad is just a
d in the category
endofunctors,
what's the problem?


Say
Hello
:F

Is it true that this could be a monadic return operation?

```
(defn return [v] (fn [] v))
```

Is it true that this could be a monadic return operation?

```
(defn return [v] (fn [] v))

 (return “jelly”)
 ;=> #<user$return$fn__1669 user$retur
 $fn__1669@314382c6>
```

Is it true that this could be a monadic return operation?

```
(defn return [v] (fn [] v))

(return “jelly”)
;=> #<user$return$fn__1669 user$return
$fn__1669@314382c6>

((return “jelly”)) ;=> “jelly”
```

Yes,

because the return object is a function (container/ monadic value) that when executed is “jelly”

```
(defn return [v] (fn [] v))

((return “jelly”)) ;=> “jelly”
```

Is it true that this could be a monadic return operation?

```
(defn return [v] v)
```

Is it true that this could be a monadic return operation?

```
(defn return [v] v)

(return “jelly”) ;=> jelly
```

No,

because the return object "jelly" is not a function (container/ monadic value)

```
(defn return [v] v)

(return "jelly") ;=> jelly
```

Is it true that this could be a monadic bind operation?

```
(defn bind [mv f] (f (mv)))
```

Is it true that this could be a monadic bind operation?

```
(defn bind [mv f] (f (mv)))

(defn with-toast [s]
  (return (str "toast & " s)))
```

Is it true that this could be a monadic bind operation?

```
(defn bind [mv f] (f (mv)))

(defn with-toast [s]
  (return (str "toast & " s)))

(bind (return "jelly") with-toast)
;=> container/ monadic value
```

Is it true that this could be a monadic bind operation?

```
(defn bind [mv f] (f (mv)))

(defn with-toast [s]
  (return (str "toast & " s)))

(bind (return "jelly") with-toast)
;=> container/ monadic value

((bind (return "jelly") with-toast))
;=> “toast & jelly”
```

Yes,
because it takes a monadic value, and applies a function and returns a monadic value

```
(defn bind [mv f] (f (mv)))

(defn with-toast [s]
  (return (str "toast & " s)))

(bind (return "jelly") with-toast)
;=> container/ monadic value

((bind (return "jelly") with-toast))
;=> “toast & jelly”
```

Can you write a monadic return function for yourself?

Can you write a monadic return function for yourself?

```
(defn return [v]
  (fn [] v))
```

```
(return "me")
;=> monadic value

((return "me"))
;=> “me”
```

Can you write a function that makes you grow and returns a monadic value?

Can you write a function that makes you grow and returns a monadic value?

```
(defn grow [s]
  (return (str s (last s))))

(grow "me")  ;=> monadic value

((grow "me"))  ;=> “mee"
```

Now, can you use a monadic bind operation to grow?

Now, can you use a monadic bind operation to grow?

```
(defn bind [mv f] (f (mv)))

(defn m-grow [mv]
  (bind mv grow))

(m-grow (return "me")) ;=> m. val

((m-grow (return "me"))) ;=> “mee”
```

Grow Bigger!

```
((m-grow
    (m-grow
      (m-grow (return "me")))))
;=> “meeee”
```

Grow Bigger!

```
((m-grow
    (m-grow
      (m-grow (return "me"))))))
;=> “meeee”
```

```
((-> (return "me") m-grow m-grow m-grow))
;=> “meeee”
```

Where you are

Destination

Directions

```
(defn directions [start]
  (return
    (.concat start
      (if (> 0.5 (rand 1))
        ": right"
        ": left"))))
```

```
(defn directions [start]
  (return
    (.concat start
      (if (> 0.5 (rand 1))
        ": right"
        ": left"))))

(defn m-directions [mv]
  (bind mv directions))
```

```
(defn directions [start]
  (return
    (.concat start
      (if (> 0.5 (rand 1))
        ": right"
        ": left"))))

(defn m-directions [mv]
  (bind mv directions))

((-> (return “here”) m-directions m-directions))
;=> "here: right: left"
```

```
(defn directions [start]
  (return
    (.concat start
      (if (> 0.5 (rand 1))
        ": right"
        ": left"))))

(defn m-directions [mv]
  (bind mv directions))

((-> (return “here”) m-directions m-directions))
;=> "here: right: left"

((-> (return nil) m-directions m-directions))
;=> NullPointerException!
```

```
(defn bind [mv f] (f (mv))
```

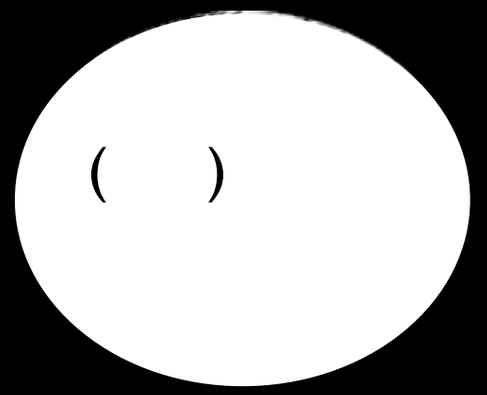
```
(defn bind [mv f]
  (let [v (mv)]
    (if (nil? v)
      (return nil)
      (f v))))
```

```
(defn bind [mv f] (f (mv))
```

```
(defn bind [mv f]
  (let [v (mv)]
    (if (nil? v)
      (return nil)
      (f v))))

((-> (return nil) m-directions m-directions))
;=> nil
```

That way ->



That way ->

Monday, March 12, 12

```
(defn return [v]
  (fn [] v))

(defn bind [mv f]
  (f (mv)))


(defn m-tea [mv name]
  (bind mv (fn [v]
             (return (str v " and " name)))))
```

```
(defn return [v]
  (fn [] v))

(defn bind [mv f]
  (f (mv)))


(defn m-tea [mv name]
  (bind mv (fn [v]
             (return (str v " and " name)))))


((-> (return "me") (m-tea "you")))
 ;=> "me and you"
```

```
(defn return [v]
  (fn [s] [v s]))
```

```
(defn return [v]
  (fn [s] [v s]))

(defn bind [mv f]
  (fn [s]
    (let [[v sn] (mv s)]
      ((f v) sn))))
```

```
(defn return [v]
  (fn [s] [v s]))

(defn bind [mv f]
  (fn [s]
    (let [[v sn] (mv s)]
      ((f v) sn))))

(defn m-tea [mv name]
  (bind mv (fn [v]
             (return (str v " and " name)))))

((-> (return "me") (m-tea "you")) 10)
;=> ["me and you" 10]
```

```
(defn take-sugar [mv]
  (bind mv (fn [v]
             (fn [s] [v (dec s)]))))
```

```
(defn take-sugar [mv]
  (bind mv (fn [v]
             (fn [s] [v (dec s)]))))


((-> (return "me")
 (take-sugar) (m-tea "you")) 10)
;=> ["me and you" 9]

((-> (return "me")
 (take-sugar) (take-sugar) (m-tea "you")) 10)
;=> ["me and you" 8]
```

Identity Monad

```
(defn return [v] (fn[] v))

(defn bind [mv f] (f (mv)))
```

```
(defn m-grow [w]
  (bind w grow))


((-> (return "me") m-grow m-grow m-grow))
;=> “meeee”
```

Maybe Monad

```
(defn return [v](fn[] v))


(defn bind [mv f]
  (let [v (mv)]
    (if (nil? v)
      (return nil)
      (f v))))


((-> (return nil) m-directions m-directions))
;=> nil
```

State Monad

```
(defn return [v]
  (fn [s] [v s]))

(defn bind [mv f]
  (fn [s]
    (let [[v sn] (mv s)]
      ((f v) sn))))
```

```
(defn take-sugar [mv]
  (bind mv (fn [v]
             (fn [s] [v (dec s)]))))

((-> (return "me")(take-sugar) (m-tea "you")) 10)
;=> ["me and you" 9]
```

Three Monad Laws

1) Left Unit - "return" acts as a neutral element of bind

```
(bind (return v) f) ≡ (f v)
```

Three Monad Laws

1) Left Unit - "return" acts as a neutral element of bind

`(bind (return v) f) ≡ (f v)`

```
(defn return [v] (fn [] v))
(defn bind [mv f] (f (mv)))

(defn grow [s](return (str s (str (last s)))))

((bind (return "me") grow))      ;=> “mee”

((grow "me"))                    ;=> “mee”
```

Three Monad Laws

2) Right Unit - "return" acts as a neutral element of bind

```
(bind mv return) ≡ mv
```

Three Monad Laws

2) Right Unit - "return" acts as a neutral element of bind

`(bind mv return)` ≡ `mv`

```
(defn return [v] (fn [] v))
(defn bind [mv f] (f (mv))

(defn grow [s](return (str s (str (last s)))))

((bind (return "me") return)) ;=> "me"

((return "me"))               ;=> "me"
```

Three Monad Laws

3) Associative - Binding two functions in succession is the same as binding one function that can be determined from them

```
(bind (bind mv f) g) ≡ (bind mv (fn [x] (bind (f x) g))
```

Three Monad Laws

3) Associative - Binding two functions in succession is the same as binding one function that can be determined from them

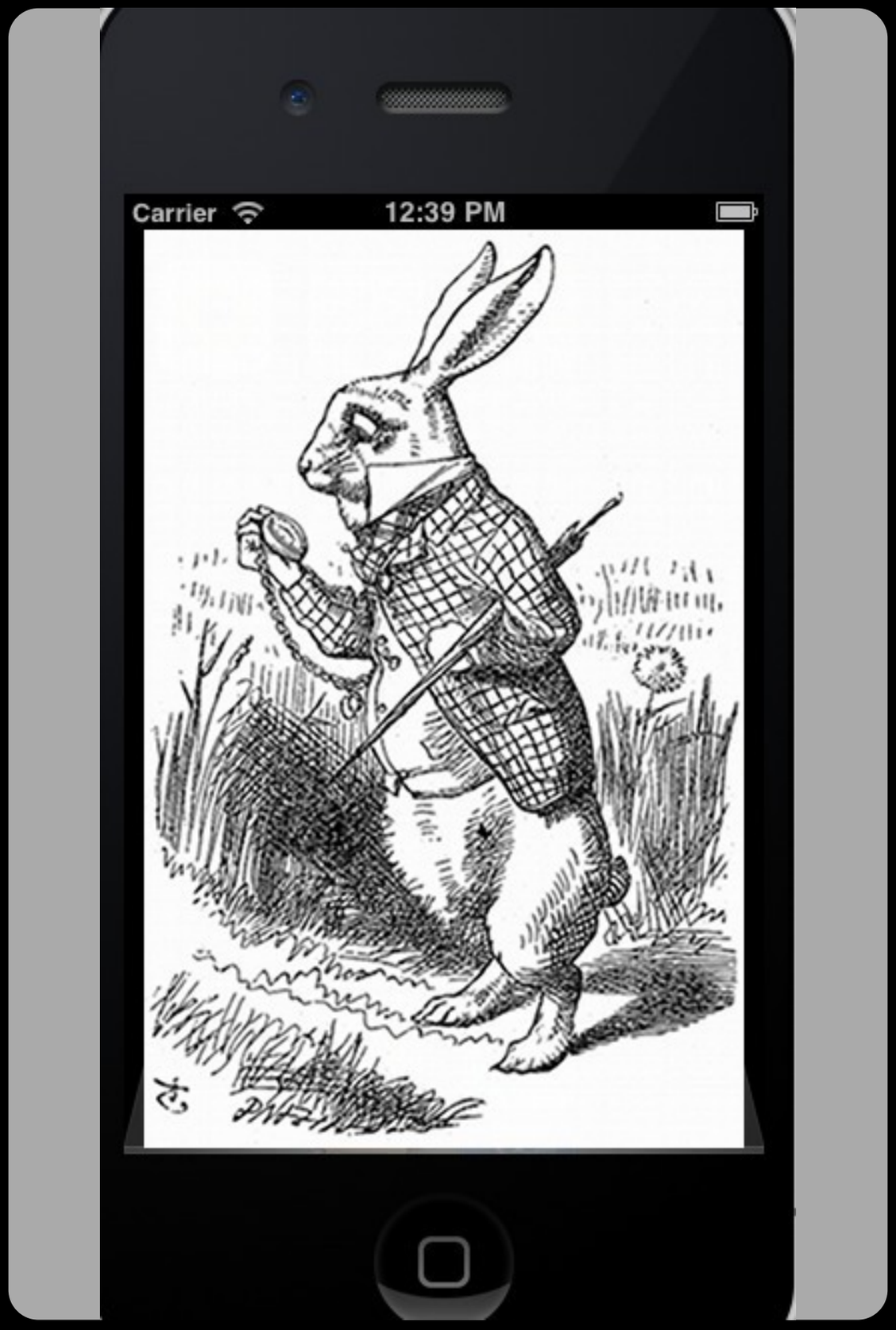
(bind (bind mv f) g) ≡ (bind mv (fn [x] (bind (f x) g))

```
(defn return [v] (fn [] v))
(defn bind [mv f] (f (mv))

(defn grow [s](return (str s (str (last s)))))


((bind (bind (return "me") grow) grow)) ;=> "meee"

((bind (return "me")
  (fn [v] (bind (grow v) grow))))       ;=> "meee"
```


Carrier
12:39 PM


Carrier
12:39 PM
Photos
Settings
Game Center
Contacts
Safari

Monday, March 12, 12

Summary

Many More Types of Monads

- IO
- Collections
- Continuation
- Writer

Etc ...

Summary

Have your cake and eat it too!

Inside functions pure and messy impure stuff on the outside

Summary

Practical Uses

- Sequential execution(parsing, fuzzing, web request processing)
- Reusable modular side effect processing (error handling/ logging)
- Keep your business logic clean in pure functions

Summary

Use Caution When Mixing Monads and Cheese

Resources

Philip Wadler:

Monads for functional Programming

Comprehending Monads

Adam Smyczek:

Introduction to Monads (Video)

http://www.youtube.com/watch?v=ObR3qi4Guys

Jim Duey:

Monads in Clojure

http://www.clojure.net/2012/02/02/Monads-in-Clojure/

Clojure Lib:

clojure.algo.monads

Konrad Hinsen

Photos

http://www.flickr.com/photos/jepoirrier/1387560191/sizes/z/in/photostream/ - blocks
http://www.flickr.com/photos/alspic/409365553/sizes/m/in/photostream/ - cake
http://www.flickr.com/photos/moyogo/4884992/sizes/z/in/photostream/ - stairs
http://www.flickr.com/photos/aeireono/369981290/ - forest path
http://www.flickr.com/photos/gillpoole/238722066/ - tree branch
http://www.flickr.com/photos/tofflerann/5936636472/sizes/m/in/photostream/ - high tea
http://www.flickr.com/photos/gsfc/4399423264/ - astronaught
http://www.flickr.com/photos/via/2115916059/ - sugar cubes