Monads are Not Scary!

Manuel M. T. Chakravarty [Part I]
André Pang [Part II]

University of New South Wales

What are Monads?

Folklore has it that,

- Monads are scary!
- Monads are *only* needed to handle I/O, or other side effects, in lazy languages.

What are Monads?

Folklore has it that,

- Monads are scary!
- Monads are *only* needed to handle I/O, or other side effects, in lazy languages.

This is utter ~~bolloc~~. . . nonsense!

Let's Try Again: What are Monads?

Monads are a programming pattern for library APIs:

Let’s Try Again: What are Monads?

Monads are a programming pattern for library APIs:

- Another such pattern is, e.g., model-view-controller (MVC).
- All we need to know is,
    - in what situations is the monad pattern useful and
    - how does it look like?

Let's Try Again: What are Monads?

Monads are a programming pattern for library APIs:

- Another such pattern is, e.g., model-view-controller (MVC).
- All we need to know is,
  - in what situations is the monad pattern useful and
  - how does it look like?

What kind of libraries benefit from monads?

Let's Try Again: What are Monads?

Monads are a programming pattern for library APIs:

- Another such pattern is, e.g., model-view-controller (MVC).
- All we need to know is,
    - in what situations is the monad pattern useful and
    - how does it look like?

What kind of libraries benefit from monads?

- Answer: Libraries that manipulate contextual information.
- Contextual information is implicit and the monad hides it.
- Examples:
    - Stateful libraries (mutuable arrays, I/O, . . .)
    - Exception handling
    - Libraries using CPS (e.g., schedulers)
    - Libraries encapsulating search
    - Parser combinators

Monad 101

A monad you all know and love(?)

```
int compare_chars ()
  {
    int a, b;

    a = getchar ();
    b = getchar ();
    return (a < b);
  }
```

Monad 101

A monad you all know and love(?)

```
compare_chars =
  do {


    a <- getChar;
    b <- getChar;
    return (a < b);
  }
```

Monad 101

A monad you all know and love(?)

```
compare_chars :: IO Bool
compare_chars =
  do {



    a <- getChar;
    b <- getChar;
    return (a < b);
  }
```

- "IO t": monad encapsulating the state of the world:
  - perform operations depending on external or internal state
  - perform operations changing external or internal state
  - when done, return a value of type t

Monad 101

A monad you all know and love(?)

```
compare_chars :: IO Bool
compare_chars =
  do {


    a <- getChar;       – getChar :: IO Char
    b <- getChar;
    return (a < b);
  }
```

- “IO t”: monad encapsulating the state of the world:
  - perform operations depending on external or internal state
  - perform operations changing external or internal state
  - when done, return a value of type t

Monad 101

A monad you all know and love(?)

```
compare_chars :: IO Bool
compare_chars =
  do {


    a <- getChar;          – getChar :: IO Char
    b <- getChar;
    return (a < b);
  }
```

- “`IO t`”: monad encapsulating the state of the world:
  - perform operations depending on external or internal state
  - perform operations changing external or internal state
  - when done, return a value of type `t`

Hello World with Gtk2Hs

```
import Graphics.UI.Gtk
main :: IO ()
main =
  do {
    initGUI;
    window <- windowNew;
    button <- buttonNew;
    set window [ containerBorderWidth := 10,
                 containerChild := button ];
    set button [ buttonLabel := "Hello World" ];
    onClicked button (putStrLn "Hello World");
    onDestroy window mainQuit;
    widgetShowAll window;
    mainGUI;
  }
```

Hello World with Gtk2Hs

```
import Graphics.UI.Gtk
main :: IO ()
main =
  do {
    initGUI;
    window <- windowNew;
    button <- buttonNew;
    set window [ containerBorderWidth := 10,
                 containerChild := button ];
    set button [ buttonLabel := "Hello World" ];
    onClicked button (putStrLn "Hello World");
    onDestroy window mainQuit;
    widgetShowAll window;
    mainGUI;
  }
```

- `onClicked :: Button -> IO () -> IO ()`

Hello World with Gtk2Hs

```
import Graphics.UI.Gtk
main :: IO ()
main =
  do {
    initGUI;
    window <- windowNew;
    button <- buttonNew;
    set window [ containerBorderWidth := 10,
                 containerChild := button ];
    set button [ buttonLabel := "Hello World" ];
    onClicked button (putStrLn "Hello World");
    onDestroy window mainQuit;
    widgetShowAll window;
    mainGUI;
  }
```

- `onClicked ::  Button -> IO () -> IO ()`
- If you can write C programs, you can write programs in the `IO` monad

Value Added

So, programming in the `IO` monad is like programming in C.
Why bother?!?

Value Added

So, programming in the `IO` monad is like programming in C. Why bother?!?

Advantage 1: Control side effects

- Different signatures, different properties:

```
noSideEffects    :: Int -> Int
maybeSideEffects :: Int -> IO Int
```

- Checked by the compiler, simplifies debugging
- Encapsulated internal state
- Required for concurrency!

```
int compare_chars_bad ()
  {
    return (getchar () < getchar ());
  }
```

Value Added

So, programming in the `IO` monad is like programming in C.
Why bother?!?

Advantage 1: Control side effects

- Different signatures, different properties:

```
noSideEffects    :: Int -> Int
maybeSideEffects :: Int -> IO Int
```

- Checked by the compiler, simplifies debugging
- Encapsulated internal state
- Required for concurrency!

```
int compare_chars_bad ()
  {
    return (getchar () < getchar ());  // what order?
  }                              // same problem in ML
```

Value Added

So, programming in the `IO` monad is like programming in C.
Why bother?!?

Advantage 1: Control side effects

- Different signatures, different properties:

```
noSideEffects    :: Int -> Int
maybeSideEffects :: Int -> IO Int
```

- Checked by the compiler, simplifies debugging
- Encapsulated internal state
- Required for concurrency!

```
compare_chars_bad =
  do {
    return (getChar < getChar);   – Type error!
  }                               – Can’t compare (IO Char)
```

Advantage 2: Monads are first-class

Advantage 2: Monads are first-class

- Define your own monad! Here it gets slightly scary...

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

Advantage 2: Monads are first-class

- Define your own monad! Here it gets slightly scary…

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

- The `do` notation is just syntactic sugar:

  ```
    do {
      c <- getChar;
      return (c == ’ ’);
    }
  ```

  ≡

  ```
    getChar >>= \c ->
    return (c == ’ ’)
  ```

Advantage 2: Monads are first-class

- Define your own monad! Here it gets slightly scary…

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

- The `do` notation is just syntactic sugar:

```
  do {
    c <- getChar;
    return (c == ’ ’);
  }
≡
  getChar >>= (\c -> return (c == ’ ’))
```

Advantage 2: Monads are first-class

- Define your own monad! Here it gets slightly scary. . .

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

- The `do` notation is just syntactic sugar:

```
  do {
    c <- getChar;
    return (c == ’ ’);
  }
≡
  getChar >>= (\c -> return (c == ’ ’))
```

- Redefine `IO` to simplify debugging!

Advantage 2: Monads are first-class

- Define your own monad! Here it gets slightly scary. . .

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

- The `do` notation is just syntactic sugar:

```
  do {
    c <- getChar;
    return (c == ’ ’);
  }
≡
  getChar >>= (\c -> return (c == ’ ’))
```

- Redefine `IO` to simplify debugging!
- `foldl (>>) (return ())` Have fun!

```
  where m >> n = do {_ <- m; n}
```

Encapsulated state

External state versus internal state

- External state: external to the application (hard disks, networks, . . .); can only be manupilated by side effects
- Internal state: part of application data structure; manipulation by side effect or state threading

Encapsulated state

External state versus internal state

- External state: external to the application (hard disks, networks, . . .); can only be manupilated by side effects
- Internal state: part of application data structure; manipulation by side effect or state threading

Encapsulated state

- State with limited life time
- Example: marker array for graph traversal
  - Pure structure: threaded set of visited nodes
  - Mutable array: update by side effect

Encapsulated state

External state versus internal state

- External state: external to the application (hard disks, networks, . . .); can only be manupilated by side effects
- Internal state: part of application data structure; manipulation by side effect or state threading

Encapsulated state

- State with limited life time
- Example: marker array for graph traversal
  - Pure structure: threaded set of visited nodes
  - Mutable array: update by side effect
- Algorithmic choice should not affect graph interface (e.g., depth first traversal skeleton)

Encapsulated state

External state versus internal state

- External state: external to the application (hard disks, networks, . . .); can only be manupilated by side effects
- Internal state: part of application data structure; manipulation by side effect or state threading

Encapsulated state

- State with limited life time
- Example: marker array for graph traversal
  - Pure structure: threaded set of visited nodes
  - Mutable array: update by side effect
- Algorithmic choice should not affect graph interface (e.g., depth first traversal skeleton)
- State transformer monad

```
data ST s a
instance Monad ST
data STRef s a

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

Different Categories of Monads

Monads classified:

- State transformer monad
- Reader monad & writer monad
- Exception monad
- CPS monad
- Indeterminism monad
- Time-runs-backwards monad
- List monad
- Strictness monad
- Identity monad
- . . .

Different Categories of Monads

Monads classified:

- State transformer monad
- Reader monad & writer monad
- Exception monad
- CPS monad
- Indeterminism monad
- Time-runs-backwards monad
- List monad
- Strictness monad
- Identity monad
- ...
- There are also monad transformers

Different Categories of Monads

Monads classified:

- State transformer monad
- Reader monad & writer monad
- Exception monad
- CPS monad
- Indeterminism monad
- Time-runs-backwards monad
- List monad
- Strictness monad
- Identity monad
- …
- There are also monad transformers
- Parser monad — Hello André!