Using Monad Transformers without understanding them

About Me

GitHub: @JordanMartinez

Core Team Member (as of Summer 2020)

Author of "PureScript: Jordan's Reference"

Work at Arista Networks, Inc.

Code, Slides, & Recording

https://github.com/JordanMartinez/pure-conf-talk

Outcome

Outcome

- Use them now
  - ExceptT
  - ReaderT
  - StateT

Agenda

- Why & What

Agenda

- Why & What
- Thought Process

Agenda

- Why & What
- Thought Process
- Usage & Mistakes

Audience Presumptions

Audience Presumptions

- Understand Monads

Audience Presumptions

- Understand Monads
- Understand "do notation"

What problem do monads solve?

What problem do monads solve?

```
foo();
const bar = baz();
return true;
```

What problem do monads solve?

```
foo
bar <- baz
pure true
```

What problem do monads solve?

Statements vs Expression

What problem do monads solve?

Analogy: Foreground vs Background

What problem do monads solve?

What problem do monads solve?

Foreground = "do notation" syntax

What problem do monads solve?

Background = `bind` implementation

What problem do monads solve?

Foreground = sequential computation

What problem do monads solve?

Background = boilerplate-y boxing and unboxing

Common Monadic Types

- `Identity`
- `Either error _`
- `Function input _`

A Monadic Type Example: `Identity _`

A Monadic Type Example: `Either error _`

A Monadic Type Example: `inputArg -> _`

What problem do monads solve?

What problem do monads solve?

Via "do notation," monads provide a **syntax** for running "computations" in a sequential order.

What problem do monads NOT solve?

What problem do monads NOT solve?

```
try {
  throw new Error("My bad!");
  console.log("I never get printed! 😭");
} catch (e) {
  console.log("Something went wrong...");
}
```

What problem do monads NOT solve?

```
const readOnlyConfig = ...;
const f = function () {
  console.log(readOnlyConfig.age);
};
f();
```

What problem do monads NOT solve?

```
let x = 1;
x += 1;
x = 5;
```

How do we implement these features?

Monad Transformers

Why use monad transformers?

Why use monad transformers?

- Add such "effects"

Why use monad transformers?

- Add such "effects"
- Decouple code

Why use monad transformers?

- Add such "effects"
- Decouple code
- Algebraic Effects (Tagless Final)

Agenda

- Why & What
- Thought Process
- Usage & Mistakes

Agenda

- ~~Why & What~~
- Thought Process
- Usage & Mistakes

How do monad transformers solve these problems?

How do monad transformers solve these problems?

Macro: **simulate** "effects"

How do monad transformers solve these problems?

Micro: make `<-` hide more boilerplate

Simulating Effects

`try ... catch`

Simulating Effects

`try ... catch`

`Either`

Simulating Effects

`readOnlyValue`

Simulating Effects

```
readOnlyValue
```

```
\readOnlyValue -> ...
```

Simulating Effects

```
let x = 0; x += 1;
```

Simulating Effects

```
let x = 0; x += 1;
```

```
Tuple output state
```

Simulating Effects

```
let x = 0; x += 1;
```

```
\oldState -> Tuple output (oldState + 1)
```

Simulating Effects

Simulating Effects

Have you seen that boilerplate?

Simulating Effects with Boilerplate

`try ... catch`

`Either`

Simulating Effects with Boilerplate

```
readOnlyValue
```

```
\readOnlyValue -> ...
```

Simulating Effects with Boilerplate

```
let x = 0; x += 1;
```

```
\oldState -> Tuple output (oldState + 1)
```

Simulating Effects with Boilerplate

What happens if we compose these effects?

Simulating Effects with Boilerplate

```
try {
  let x = 5;
  x = x + config.age;
  console.log(`x is ${x}`);
  if (x > config.max) throw new Error("x is too large");
  return "Everything is fine!";
} catch (e) {
  console.log(e);
  return "What did you do!?";
}
```

Simulating Effects with Boilerplate

```
try {
  let x = 5;
  x = x + globalRef1;
  console.log(`x is ${x}`);
  if (x > globalRef2) throw new Error("x is too large");
  return "Everything is fine!";
} catch (e) {
  console.log(e);
  return "What did you do!?";
}
```

```
readOnlyValue -> state -> Effect (Tuple (Either Error Output) State)
```

Simulating Effects with Boilerplate

```
.·´¯`(>▂<)´¯`·.
```

Simulating Effects with Boilerplate

If only...

Simulating Effects with Boilerplate

But wait! Don't monads...?

Simulating Effects with Boilerplate

Can we transform monads...?

Simulating Effects with Boilerplate

Implementing `try ... catch`

`monad output`

Implementing `try ... catch`

`monad (Either error output)`

Implementing `try ... catch`

```
newtype ExceptT error monad output =
  ExceptT (monad (Either error output))
```

```
runExceptT (ExceptT ma) = ma
```

Implementing `try ... catch`

```
-- throw new Error
throwError :: forall m e o. Monad m =>
  e -> m (Either e o)
throwError e = pure $ Left e
```

Implementing `try ... catch`

```
-- try ... catch
catchError :: forall m e o. Monad m =>
  m (Either e o) -> (e -> m (Either e o)) -> m (Either e o)
catchError ma handler = do
  either <- ma
  case either of
    Left e -> handler e
    right -> pure right
```

Implementing `try ... catch`

Before vs After

Implementing `readOnlyValue`

`monad output`

Implementing `readOnlyValue`

```
readOnlyValue -> monad output
```

Implementing `readOnlyValue`

```
newtype ReaderT readOnlyValue monad output =
  ReaderT (readOnlyValue -> monad output)
```

```
runReaderT (ReaderT argToMa) arg = argToMa arg
```

Implementing `readOnlyValue`

```
ask :: forall m globalVal. Monad m => m globalVal
ask = (\arg -> pure arg)
```

Implementing `readOnlyValue`

Before vs After

Implementing `let x = 0; x += 1`

`monad output`

Implementing `let x = 0; x += 1`

```
state -> monad (Tuple output state)
```

Implementing `let x = 0; x += 1`

```
newtype StateT state monad output =
  StateT (state -> monad (Tuple output state))
```

```
runStateT (ReaderT sToMa) initialState = sToMa initialState
```

Implementing `let x = 0; x += 1`

```
get :: forall m state. Monad m => m state
get = (\currentState -> pure (Tuple currentState currentState))
```

Implementing `let x = 0; x += 1`

```
put :: forall m state. Monad m => state -> m Unit
put newState = pure (Tuple unit newState)
```

Implementing `let x = 0; x += 1`

Before vs After

Agenda

- ~~Why & What~~
- Thought Process
- Usage & Mistakes

Agenda

- ~~Why & What~~
- ~~Thought Process~~
- Usage & Mistakes

Usage & Mistakes

Usage & Mistakes

Don't!

Usage & Mistakes

One Transformer?

Usage & Mistakes

```
foo :: Effect (Either String SomeValue)
foo = runExceptT do
  resp1 <- callApiRequest1
  resp2 <- callApiRequest2
  resp3 <- callApiRequest3
  pure $ ...
```

Usage & Mistakes

Multiple Transformers?

Usage & Mistakes

Use type classes to encode business logic

Usage & Mistakes

```
npm run tc-stack-order
```

Usage & Mistakes

Stack and Stack Order

Usage & Mistakes

```
state -> monad (Either err (Tuple output state)
```

vs

```
state -> monad (Tuple (Either err output) state)
```

Usage & Mistakes

```
Right (Tuple output state)
```

vs

```
Tuple (Right output) state
```

Usage & Mistakes

`Left error`

vs

`Tuple (Left error) state`

Usage & Mistakes

```
forall monad. ....
```

Usage & Mistakes

```
npm run tc-different-monads
```

Agenda

- ~~Why & What~~
- ~~Thought Process~~
- Usage & Mistakes

Agenda

- ~~Why & What~~
- ~~Thought Process~~
- ~~Usage & Mistakes~~

Where do we go from here?

Where do we go from here?

- https://github.com/JordanMartinez/pure-conf-talk
  - `slides/CheatSheet.md`
    - Monad Transformers <-> Concrete types
    - Stack Order

Where do we go from here?

- https://github.com/JordanMartinez/pure-conf-talk
  - `slides/CheatSheet.md`
    - Monad Transformers <-> Concrete types
    - Stack Order
  - `src` directory - each transformer: boilerplate, nested bind, do notation, transformer

Where do we go from here?

- https://github.com/JordanMartinez/pure-conf-talk
  - `slides/CheatSheet.md`
    - Monad Transformers <-> Concrete types
    - Stack Order
  - `src` directory - each transformer: boilerplate, nested bind, do notation, transformer
  - Run examples of transformers (`package.json` scripts)

Where do we go from here?

- https://github.com/JordanMartinez/pure-conf-talk
  - `slides/CheatSheet.md`
    - Monad Transformers <-> Concrete types
    - Stack Order
  - `src` directory - each transformer: boilerplate, nested bind, do notation, transformer
  - Run examples of transformers (`package.json` scripts)
- Explore the Capability Pattern

Summary

- Expressions, not statements

Summary

- Expressions, not statements
- Add effects via transformers

Summary

- Expressions, not statements
- Add effects via transformers
- Stack Order & `forall monad`

Contact Me

Discord: @JordanMartinez

GitHub: @JordanMartinez