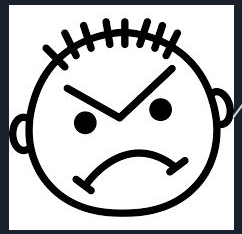
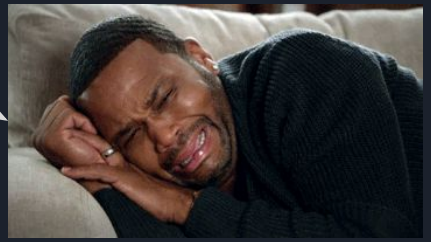

Free Your Monads and the Rest Will Follow
Wherein our heroes use the Free Monad in production

Motivations

Me, a simple diagram





Me and Category Theory

Practical Free Monadery

What IS the free monad?

What IS the free monad?

Get your verbs right first

```
type Interpreter = Action ~> TwitterFuture
```

```
trait Action[A]

case class TakeInput() extends Action[Unit]
case class ValidateInput() extends Action[Unit]
case class FetchOtherData() extends Action[Unit]
case class ValidateOtherData() extends Action[Unit]
case class TransformData() extends Action[Unit]
case class OutputData() extends Action[Unit]
```

Translate your verbs to lifted methods

```
type FreeAction[A] = Free[Action, A]

def takeInput(): FreeAction[Unit] = liftF[Action, Unit](TakeInput())
def validateInput(): FreeAction[Unit] = liftF[Action, Unit](ValidateInput())
def fetchOtherData(): FreeAction[Unit] = liftF[Action, Unit](FetchOtherData())
def validateOtherData(): FreeAction[Unit] = liftF[Action, Unit](ValidateOtherData())
def transformData(): FreeAction[Unit] = liftF[Action, Unit](TransformData())
def outputData(): FreeAction[Unit] = liftF[Action, Unit](OutputData())
```

Wire together in a for comp

```
def makeItGo(): FreeAction[Unit] = for {
  _ <- takeInput()
  _ <- validateInput()
  _ <- fetchOtherData()
  _ <- validateOtherData()
  _ <- transformData()
  _ <- outputData()
} yield Unit
```

Make an interpreter

```
class BareInterpreter extends Interpreter {
  def apply[A](action: Action[A]): TwitterFuture[Unit] = action match {
    case TakeInput() => TwitterFuture.Unit
    case ValidateInput() => TwitterFuture.Unit
    case FetchOtherData() => TwitterFuture.Unit
    case ValidateOtherData() => TwitterFuture.Unit
    case TransformData() => TwitterFuture.Unit
    case OutputData() => TwitterFuture.Unit
  }
}
```

Make it go

```
def makeAThing(id: Int): TwitterFuture[Unit] = {
  makeItGo().foldMap(mapOfInterpreters.getOrElse(id, new UnimplementedInterpreter))
}
```

Now with types!

Start with your DSL

```
case class TakeInput(i: Int) extends Action[String]
case class ValidateInput(s: String) extends Action[Unit]
case class FetchOtherData(s: String) extends Action[List[_]]
case class ValidateOtherData(l: List[_]) extends Action[Unit]
case class TransformData(l: List[_]) extends Action[String]
case class OutputData(s: String) extends Action[Unit]
```

Now with types!

Adjust your lifts

```
def takeInput(i: Int): FreeAction[String] = liftF[Action, String](TakeInput(i))
def validateInput(s: String): FreeAction[Unit] = liftF[Action, Unit](ValidateInput(s))
def fetchOtherData(s: String): FreeAction[List[_]] = liftF[Action, List[_]](FetchOtherData(s))
def validateOtherData(l: List[_]): FreeAction[Unit] = liftF[Action, Unit](ValidateOtherData(l))
def transformData(l: List[_]): FreeAction[String] = liftF[Action, String](TransformData(l))
def outputData(s: String): FreeAction[Unit] = liftF[Action, Unit](OutputData(s))
```

Now with types!

Wire it into the for comp

```
def makeItGo(id: Int): FreeAction[Unit] = for {
  a <- takeInput(id)
  _ <- validateInput(a)
  b <- fetchOtherData(a)
  _ <- validateOtherData(b)
  c <- transformData(b)
  d <- outputData(c)
} yield d
```

Now with types!

Adjust interpreter behavior

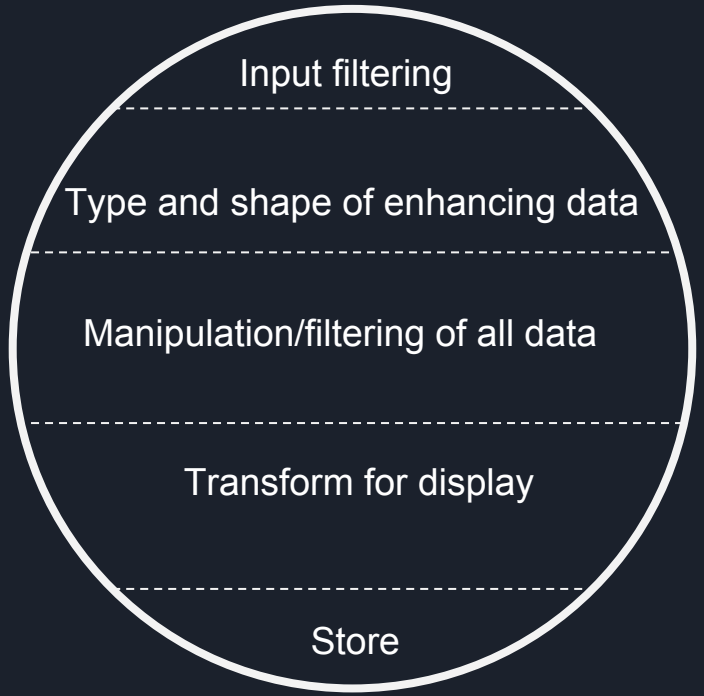
```
def apply[A](action: Action[A]): TwitterFuture[Unit] = action match {
  case TakeInput(i: Int) => TwitterFuture.value("")
  case ValidateInput(s: String) => TwitterFuture.Unit
  case FetchOtherData(s: String) => TwitterFuture.value(List.empty)
  case ValidateOtherData(l: List[_]) => TwitterFuture.Unit
  case TransformData(l: List[_]) => TwitterFuture.value("")
  case OutputData(s: String) => TwitterFuture.Unit
}
```

Why we chose the Free Monad

Our Business

- Our customers are large utilities
- We create reports for their end users about their home energy use
- Reports are whitelabled and customized for each tenant
- Common data model on the back-end

The ball of commonality



The legacy system

- Everything was a batch
- Workflow was done through XML
- Branching based on tenant names or ids
- Opaque, hard to work on
- Easy to break

What we wanted

- Programmatic limits in choices
- Clear, easy to understand workflow
- Tight contracts so everything HAD to happen in a similar way
- Low cognitive cost in working on different tenants
- Configuration over Customization
- Make things 90% the same, so we can focus on the details of what really makes something different.

Free Monad to the rescue

- Each interpreter can make different choices in:
  - Input data to fetch
  - Judgment
  - Display
- However, all interpreters must implement the same kinds of steps, with no variation of input or output types
- Focus on the differences knowing the commonalities are taken care of

Why it worked for us

- Already using monadic operations on twitter futures for composition
- Differing details are all easily discoverable by following a customer’s interpretation of a known flow
- Changes to one customer don’t affect another customer. They are similar in shape and rules, but independent.

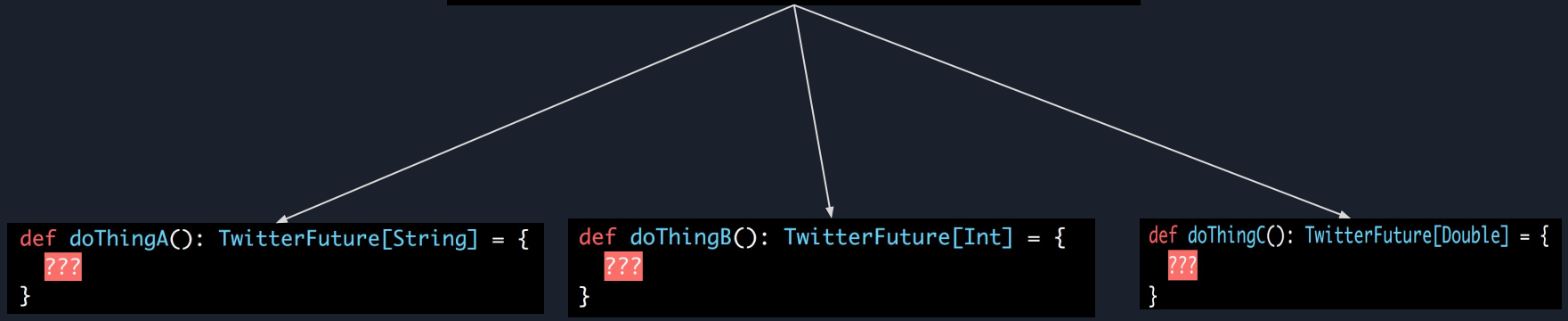
Why it worked for us

- Deterministic system, end-to-end
- The controlling flow can be tested independently of data.
- The interpretation is unit-testable and compiler enforced.
- It is easy to do the right thing and hard to do the wrong thing.

Patterns we used

- Manager / Worker pattern
  - You have managers and you have workers
  - Workers do one unit of work, return one thing
  - Managers only coordinate work
  - Many layers of managers
  - Workers are unit testable, managers can be unit/integration tested

```
def doThing1: TwitterFuture[Unit] = for {
  a <- doThingA()
  b <- doThingB()
  c <- doThingC()
} yield collector(a, b, c)
```



```
def doThingA(): TwitterFuture[String] = {
  ???
}
```

```
def doThingB(): TwitterFuture[Int] = {
  ???
}
```

```
def doThingC(): TwitterFuture[Double] = {
  ???
}
```

Manager and worker

Patterns we used

- Composition EVERYWHERE
  - The operation of the entire system reads like lists
  - Everything is early return
  - Locks us into a small number of monadic shapes
  - Everything becomes a Future in the end
  - Filters are lists of functions with a distinct type

```
def bigManager: TwitterFuture[Unit] = for {
  x <- doThing1
  y <- doThing2
  z <- doThing3
} yield (x, y, z)
```

```
def doThing1: TwitterFuture[Unit] = for {
  a <- doThingA()
  b <- doThingB()
  c <- doThingC()
} yield collector(a, b, c)
```

```
def doThingA(): TwitterFuture[String] = {
  ???
}
```

```
def doThingB(): TwitterFuture[Int] = {
  ???
}
```

```
def doThingC(): TwitterFuture[Double] = {
  ???
}
```

So many levels, and all for comps

Patterns we used

- Free Monad as a Command pattern
  - DSL must be implemented
  - Well named DSL allows you to know generally what you need to do at any stage
  - Each verb in the DSL takes one type of input returns one type of output
  - Typed, definitive contracts at each stage of the workflow

The Path To Production

Guiding Principles

Any developer can work on it

- We are a mixed Java/Scala/Python shop
- People should be able to move to the project with low cognitive lift in the language
- No symbolic forms. If a non-symbolic form is not available (~>), use a type alias
- Settle early on the contract, then don’t change it unless you have to
- Document the heck out of it

Practical Functional programming without learning Category Theory

- Deeply functional operations (MonadT, LiftF) live in a syntax package and are required to be clearly and thoroughly commented
- Explicit agreement on style, without religion. Use it to teach, not to bludgeon
- Build a culture of clean code, readable functional style from the inception of the project
- Be disciplined about readability, especially early in the process
- Pointers to theoretical training in the documentation

Does what it says on the box

- All names (variables, methods, classes, objects, exceptions) must be direct and meaningful
- Libraries must be functional, terse and readable (cats Validated, json4s, twitter futures, etc)
- For comprehensions should read like a recipe
- At any level, I should know from the method name what its purpose is without drilling down

Non-biased data gathering

- In a multi-tenant system, we should gather all the data we are ABLE to
- Sort, format and filter data later, in tenant specific areas
- DO NOT manipulate data in the workers, do that in a manager controlled by a tenant specific interpreter

KISS

- Only use the monads you NEED (Twitter Futures, cats Validated)
- Only use the word form of operations
- Principle of Least Power
- One test format (we like FlatSpec because it reads as clear English)

Growing and Shrinking

First pass

- Each tenant had a full interpreter
- Adding a new tenant was copy-pasta on 3-4 code files with minor tweaks
- Each tenant handled everything from input to storage explicitly

Second pass

- Tenants are 90% the same in what they do
- Inheritance to the rescue
- Interpreter is inherited, all cases in the match statement go through inherited defs
- Tenants now only override only the exact things they want to change
- Most tenants are ~20 lines of code, including import statements
- End-to-end new tenant can happen in a day

The Scala Sweet Spot

Using FP and OOP together

Functional Programming

- Collections first
- Monadic consistency
- For-comps for readability
- Type aliases for String types for tighter contracts
- Type alias symbolic types for FUD reduction

Object Oriented Programming

- Interoperate with our Java platform
- Inheritance for DRY
  - Input and output types
  - Configuration management
  - Interpreters
- Modular structure to make unit testing more reliable

The Wrap Up

Effective Free Monadery

- Multi-tenant / multi-format system
- In the abstract, all the steps can be called the same thing, work on the same classes of data
- Differentiation of details of how the data gets manipulated
- Want differentiation with minimal branching in code
- Want a tight, understandable contract
- Want compiler enforced adherence to the contract
- Want shared, unit testable fundamentals to compose

Effective Free Monadery

- Use the manager / worker pattern
- Compose like a boss
- Name things clearly and intentionally
- Only use the monadic types you need
- Type alias symbolism

Thank you