Isolating State with Monads
An Introduction to Reader, Writer and State

Ben Kolera

bfpg.org

April 23, 2013

Me

Me

- Scala and Perl Dev at iseek Communications

Me

- Scala and Perl Dev at iseek Communications
- Lots of diverse codebases coded by a small team

Me

- Scala and Perl Dev at iseek Communications
- Lots of diverse codebases coded by a small team
- Often have to modify old bitrotted code.

Me

- Scala and Perl Dev at iseek Communications
- Lots of diverse codebases coded by a small team
- Often have to modify old bitrotted code.
- ~~Hate~~ Suck at writing tests

Me

- Scala and Perl Dev at iseek Communications
- Lots of diverse codebases coded by a small team
- Often have to modify old bitrotted code.
- ~~Hate~~ Suck at writing tests
- Need to be able to easily reason about code and be confident in changes.

Me

- Scala and Perl Dev at iseek Communications
- Lots of diverse codebases coded by a small team
- Often have to modify old bitrotted code.
- ~~Hate~~ Suck at writing tests
- Need to be able to easily reason about code and be confident in changes.
- Love how types can keep my refactorings honest and correct

Me

- Scala and Perl Dev at iseek Communications
- Lots of diverse codebases coded by a small team
- Often have to modify old bitrotted code.
- ~~Hate~~ Suck at writing tests
- Need to be able to easily reason about code and be confident in changes.
- Love how types can keep my refactorings honest and correct
- Slowly trying to convince my team that types are awesome

Goals of this Talk

Goals of this Talk

- To take you on a refactoring journey!

Goals of this Talk

- To take you on a refactoring journey!
- To show you all of the little baby steps on the path to ReaderWriterState.

Goals of this Talk

- To take you on a refactoring journey!
- To show you all of the little baby steps on the path to ReaderWriterState.
- To highlight reasons why unrestricted side effects are bad.

Goals of this Talk

- To take you on a refactoring journey!
- To show you all of the little baby steps on the path to ReaderWriterState.
- To highlight reasons why unrestricted side effects are bad.
- To show to that purifying code $\neq$ removing state.

Goals of this Talk

- To take you on a refactoring journey!
- To show you all of the little baby steps on the path to ReaderWriterState.
- To highlight reasons why unrestricted side effects are bad.
- To show to that purifying code $\neq$ removing state.
- To show that types and structures can be used to build and restrict computations as well as data.

Goals of this Talk

- To take you on a refactoring journey!
- To show you all of the little baby steps on the path to ReaderWriterState.
- To highlight reasons why unrestricted side effects are bad.
- To show to that purifying code $\neq$ removing state.
- To show that types and structures can be used to build and restrict computations as well as data.
- To convince you that all of this isn't hard; its just (brain alteringly) different!

Goals of this Talk

- To take you on a refactoring journey!
- To show you all of the little baby steps on the path to ReaderWriterState.
- To highlight reasons why unrestricted side effects are bad.
- To show to that purifying code $\neq$ removing state.
- To show that types and structures can be used to build and restrict computations as well as data.
- To convince you that all of this isn't hard; its just (brain alteringly) different!
- To convince you that these weird and wonderful changes actually make your code easier to reason about, not the other way around.

Scalaz

Scalaz

- Library with lots of FP goodies missing from the std lib.

Scalaz

- Library with lots of FP goodies missing from the std lib.
- Focus is on composable typeclass based programming.

Scalaz

- Library with lots of FP goodies missing from the std lib.
- Focus is on composable typeclass based programming.
- Really awesome!

Scalaz

- Library with lots of FP goodies missing from the std lib.
- Focus is on composable typeclass based programming.
- Really awesome!
- Fair few code committers are familiar BFPG faces. :)

Scalaz

- Library with lots of FP goodies missing from the std lib.
- Focus is on composable typeclass based programming.
- Really awesome!
- Fair few code committers are familiar BFPG faces. :)
- Code from this talk depends on it.

Monad Typeclass/Interface

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`
  - flatMap: `5.point[Id].flatMap( x => 6.point[Id] )`

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`
  - flatMap: `5.point[Id].flatMap( x => 6.point[Id] )`
- We get the map method for free.

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`
  - flatMap: `5.point[Id].flatMap( x => 6.point[Id] )`
- We get the map method for free.
  - `5.point[Id].map( _ + 2 )`

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`
  - flatMap: `5.point[Id].flatMap( x => 6.point[Id] )`
- We get the map method for free.
  - `5.point[Id].map( _ + 2 )`
  - `5.point[Id].flatMap( x => 6.point[Id].map( _ + x ) )`

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`
  - flatMap: `5.point[Id].flatMap( x => 6.point[Id] )`
- We get the map method for free.
  - `5.point[Id].map( _ + 2 )`
  - `5.point[Id].flatMap( x => 6.point[Id].map( _ + x ) )`
- Lots of other machinery written around the typeclass.

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`
  - flatMap: `5.point[Id].flatMap( x => 6.point[Id] )`
- We get the map method for free.
  - `5.point[Id].map( _ + 2 )`
  - `5.point[Id].flatMap( x => 6.point[Id].map( _ + x ) )`
- Lots of other machinery written around the typeclass.
  - Sequence is the most important for us tonight.

Monad Typeclass/Interface

- Typeclass for a type constructor that is 'flatmappable'
  - We write a monad for `Option[_]`, not `Option[String]`
- The typeclass has two methods:
  - return: `5.point[Id]`
  - flatMap: `5.point[Id].flatMap( x => 6.point[Id] )`
- We get the map method for free.
  - `5.point[Id].map( _ + 2 )`
  - `5.point[Id].flatMap( x => 6.point[Id].map( _ + x ) )`
- Lots of other machinery written around the typeclass.
  - Sequence is the most important for us tonight.
  - `(1 to 5).toList.map( _.point[Id] ).sequenceU`

Why is this useful?

Why is this useful?

- Each monad describes a different computational context.

Why is this useful?

- Each monad describes a different computational context.
- We're extremely restricted in how we can interact with a context:

Why is this useful?

- Each monad describes a different computational context.
- We're extremely restricted in how we can interact with a context:
  - We can transform what is inside with map.

Why is this useful?

- Each monad describes a different computational context.
- We're extremely restricted in how we can interact with a context:
  - We can transform what is inside with map.
  - Join it to another context with flatmap.

Why is this useful?

- Each monad describes a different computational context.
- We're extremely restricted in how we can interact with a context:
  - We can transform what is inside with map.
  - Join it to another context with flatmap.
- This is useful to describe/enforce sequential computations.

Why is this useful?

- Each monad describes a different computational context.
- We're extremely restricted in how we can interact with a context:
  - We can transform what is inside with map.
  - Join it to another context with flatmap.
- This is useful to describe/enforce sequential computations.
- Each monad defines flatmap/join in its own special way:

Why is this useful?

- Each monad describes a different computational context.
- We're extremely restricted in how we can interact with a context:
  - We can transform what is inside with map.
  - Join it to another context with flatmap.
- This is useful to describe/enforce sequential computations.
- Each monad defines flatmap/join in its own special way:
  - We can make lots of different kinds of chained computations.

Why is this useful?

- Each monad describes a different computational context.
- We're extremely restricted in how we can interact with a context:
  - We can transform what is inside with map.
  - Join it to another context with flatmap.
- This is useful to describe/enforce sequential computations.
- Each monad defines flatmap/join in its own special way:
  - We can make lots of different kinds of chained computations.
  - We're going to use different monads to serialize different kinds of state between our computations.

Our (toy) problem

Read XML from a file and:

Our (toy) problem

Read XML from a file and:

- Pretty print it to standard out
  - Configurable indentation step (2 spaces,4 spaces, tabs)

Our (toy) problem

Read XML from a file and:

- Pretty print it to standard out
  - Configurable indentation step (2 spaces,4 spaces, tabs)
- Validating the XML and print warnings to stderr
  - We only have one rule: <msg> may only include text

Example Execution

Input

```
<hello><msg><error>This should not be here.</error>World</msg></hello>
```

Output

```
WARN: Msg should only contain text, contains :<error>
<hello>
  <msg>
    <error>
      This should not be here.
    </error>
    World
  </msg>
</hello>
```

Our Imperative Beginning

Our Globals

- Global configuration of the indentation character(s).
- Mutable buffer of errors to append to in the program.
- Mutable stack of element names to track what we've seen.

```
object Start {

  case class Config ( indentSeq: String )
  val config = Config( "  " )
  val errors:mutable.ListBuffer[String] = mutable.ListBuffer()
  val foundElems = mutable.Stack.empty[String]
```

Indenter

- Takes an indent level and appends level amount of the indentSeq to the text.

```
def indented( indentLevel: Int , text: String ) = {
  ( config.indentSeq * indentLevel ) + text
}
```

Verifier

- Examines the new event and the top of the global stack.
- Appends error to errors if get a ElemStart inside of <msg>

```
def verifyNewElement( event: XMLEvent ) = {
  (foundElems.headOption,event) match {
    case (Some("msg"),EvElemStart( _ , l , _ , _ )) => errors += (
      s"WARN: Msg should only contain text, contains: <$l>"
    )
    case _ => ()
  }
}
```

Stream Processor

```
def main(filename: String) = {
  val lines = for ( event <- getStream( filename ) ) yield {
    verifyNewElement( event )
    event match {
      case EvElemStart( _ , l , a , scope ) => {
        val out = indented( foundElems.size , s"<$l>" )
        foundElems.push( l )
        out
      }
      case EvElemEnd( _ , l ) => {
        foundElems.pop()
        indented( foundElems.size , s"</$l>" )
      }
      case EvText(t) => indented( foundElems.size , t )
    }
  }
```

Output

```
    errors.foreach( System.err.println _ )
    lines.foreach( println _ )
  } //end main
```

Output

```
    errors.foreach( System.err.println _ )
    lines.foreach( println _ )
  } //end main
```

- Bzzt! This has a subtle bug

Output

```
  errors.foreach( System.err.println _ )
  lines.foreach( println _ )
} //end main
```

- Bzzt! This has a subtle bug
- The stream processing is lazy and isn't evaluated till the last line.

Output

```
  errors.foreach( System.err.println _ )
  lines.foreach( println _ )
} //end main
```

- Bzzt! This has a subtle bug
- The stream processing is lazy and isn't evaluated till the last line.
- Thus there aren't any errors in the buffer when we print them.

Output

```
  errors.foreach( System.err.println _ )
  lines.foreach( println _ )
} //end main
```

- Bzzt! This has a subtle bug
- The stream processing is lazy and isn't evaluated till the last line.
- Thus there aren't any errors in the buffer when we print them.
- Our harmless-looking variable is already causing us troubles!

Output

```
  errors.foreach( System.err.println _ )
  lines.foreach( println _ )
} //end main
```

- Bzzt! This has a subtle bug
- The stream processing is lazy and isn't evaluated till the last line.
- Thus there aren't any errors in the buffer when we print them.
- Our harmless-looking variable is already causing us troubles!
- Sure, here we don't need or gain benefit from the stream, but:

Output

```
  errors.foreach( System.err.println _ )
  lines.foreach( println _ )
} //end main
```

- Bzzt! This has a subtle bug
- The stream processing is lazy and isn't evaluated till the last line.
- Thus there aren't any errors in the buffer when we print them.
- Our harmless-looking variable is already causing us troubles!
- Sure, here we don't need or gain benefit from the stream, but:
  - Some degree of laziness is inevitable in fp.

Output

```
  errors.foreach( System.err.println _ )
  lines.foreach( println _ )
} //end main
```

- Bzzt! This has a subtle bug
- The stream processing is lazy and isn't evaluated till the last line.
- Thus there aren't any errors in the buffer when we print them.
- Our harmless-looking variable is already causing us troubles!
- Sure, here we don't need or gain benefit from the stream, but:
  - Some degree of laziness is inevitable in fp.
  - Laziness and unconstrained side-effects don't play nice.

Output

```
  errors.foreach( System.err.println _ )
  lines.foreach( println _ )
} //end main
```

- Bzzt! This has a subtle bug
- The stream processing is lazy and isn't evaluated till the last line.
- Thus there aren't any errors in the buffer when we print them.
- Our harmless-looking variable is already causing us troubles!
- Sure, here we don't need or gain benefit from the stream, but:
  - Some degree of laziness is inevitable in fp.
  - Laziness and unconstrained side-effects don't play nice.
  - Even if you discount laziness, non referentially transparent functions don't compose, which is the point of FP!

Death to the Global Config!

Before we go crazy fixing our side effects...

- With or without FP, global config is a code smell. Lets fix it!

Death to the Global Config!

Before we go crazy fixing our side effects...

- With or without FP, global config is a code smell. Lets fix it!
- In OO, we'd use DI.

Death to the Global Config!

Before we go crazy fixing our side effects...

- With or without FP, global config is a code smell. Lets fix it!
- In OO, we'd use DI.
- In scala, one way to DI is to use curried functions.

Curried Functions

- Just a function that takes a config and returns another function.

Curried Functions

- Just a function that takes a config and returns another function.
- We can store that configured function and call it later.

Curried Functions

- Just a function that takes a config and returns another function.
- We can store that configured function and call it later.

Old Indenter

```
def indented( indentLevel: Int , text: String ) = {
  ( config.indentSeq * indentLevel ) + text
}
```

New Indenter

```
def indented( config: Config )( level: Int, text: String ) = {
  (config.indentSeq * level) + text
}
```

Change to the Stream Processor

Old Processor

```
val lines = for ( event <- getStream( filename ) ) yield {
  verifyNewElement( event )
  event match {
    case EvElemStart( _ , l , a , scope ) => {
      val out = indented( foundElems.size , s"<$l>" )
```

New Processor

```
val indenter = indented( Config( "  " ) ) _
val lines = for ( event <- getStream( filename ).toList ) yield {
  verifyNewElement( event )
  event match {
    case EvElemStart( _ , l , a , scope ) => {
      val out = indenter( foundElems.size , s"<$l>" )
```

Wait! That sucks!

Wait! That sucks!

- We need to inject every function that needs the config!

Wait! That sucks!

- We need to inject every function that needs the config!
- Also need to thread the config through every level, as well.

Wait! That sucks!

- We need to inject every function that needs the config!
- Also need to thread the config through every level, as well.
- This idea is the right direction, but the idea wont scale.

Introducing the Reader Monad

Introducing the Reader Monad

- Same concept as currying, but the other way around:

Introducing the Reader Monad

- Same concept as currying, but the other way around:
  - We create a function that is waiting for configuration.

Introducing the Reader Monad

- Same concept as currying, but the other way around:
  - We create a function that is waiting for configuration.
- Joining two readers will thread the conf between them.

Introducing the Reader Monad

- Same concept as currying, but the other way around:
  - We create a function that is waiting for configuration.
- Joining two readers will thread the conf between them.
- Only need to put the config in the top and the monad will do the rest.

```
type PpReader[+A] = Reader[Config,A]

def getIndentSeq(): PpReader[String] = Reader { (conf) => conf.indentSeq }

def indented( level: Int, text: String ):PpReader[String] =
  getIndentSeq().map( indentSeq => ( indentSeq * level) + text )
```

List of Readers

No change to the stream processor!

```
val readers = for ( event <- getStream( filename ).toList ) yield {
  verifyNewElement(event)
  event match {
    case EvElemStart( _ , l , _ , _ ) => {
      val out = indented( foundElems.size , s"<$l>" )
      foundElems.push( l )
      out
    }
    case EvElemEnd( _ , l ) => {
      foundElems.pop()
      indented( foundElems.size , s"</$l>" )
    }
    case EvText(t) => indented( foundElems.size , t )
  }
}
```

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

- Transform our `List[PpReader[String]]` to `PpReader[List[String]]`

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

- Transform our `List[PpReader[String]]` to `PpReader[List[String]]`
- Inject config into our final program to get a `List[String]`

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

- Transform our `List[PpReader[String]]` to `PpReader[List[String]]`
- Inject config into our final program to get a `List[String]`
- We only have to thread in the config once.

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

- Transform our `List[PpReader[String]]` to `PpReader[List[String]]`
- Inject config into our final program to get a `List[String]`
- We only have to thread in the config once.
- If our conf is an immutable structure, no one can change it on us.

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

- Transform our `List[PpReader[String]]` to `PpReader[List[String]]`
- Inject config into our final program to get a `List[String]`
- We only have to thread in the config once.
- If our conf is an immutable structure, no one can change it on us.
- It's almost like we have a reusable Config => Output program.

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

- Transform our `List[PpReader[String]]` to `PpReader[List[String]]`
- Inject config into our final program to get a `List[String]`
- We only have to thread in the config once.
- If our conf is an immutable structure, no one can change it on us.
- It's almost like we have a reusable Config => Output program.
- Almost: since there is state that is not contained in the program, this is impossible right now.

Injecting the config

```
readers.sequenceU.apply( Config( "  " ) ).foreach( println _ )
```

- Transform our `List[PpReader[String]]` to `PpReader[List[String]]`
- Inject config into our final program to get a `List[String]`
- We only have to thread in the config once.
- If our conf is an immutable structure, no one can change it on us.
- It's almost like we have a reusable Config => Output program.
- Almost: since there is state that is not contained in the program, this is impossible right now.
- This state is a pain! Lets move onto moving it into our monad.

Death to the error buffer!

- The mutable error buffer is easiest to fix next.

Death to the error buffer!

- The mutable error buffer is easiest to fix next.
- What about putting something mutable into the config?

Death to the error buffer!

- The mutable error buffer is easiest to fix next.
- What about putting something mutable into the config?
  - That's better than what we have, but not good enough!

Death to the error buffer!

- The mutable error buffer is easiest to fix next.
- What about putting something mutable into the config?
  - That's better than what we have, but not good enough!
  - If a config was reused for two different monad executions, state may leak across them.

Death to the error buffer!

- The mutable error buffer is easiest to fix next.
- What about putting something mutable into the config?
  - That's better than what we have, but not good enough!
  - If a config was reused for two different monad executions, state may leak across them.
  - We need to keep our state sealed completely inside the monad

Death to the error buffer!

- The mutable error buffer is easiest to fix next.
- What about putting something mutable into the config?
  - That's better than what we have, but not good enough!
  - If a config was reused for two different monad executions, state may leak across them.
  - We need to keep our state sealed completely inside the monad
- We're already collecting a list of readers:

Death to the error buffer!

- The mutable error buffer is easiest to fix next.
- What about putting something mutable into the config?
  - That's better than what we have, but not good enough!
  - If a config was reused for two different monad executions, state may leak across them.
  - We need to keep our state sealed completely inside the monad
- We're already collecting a list of readers:
  - Why not collect errors and readers at the same time?

Changes to Stream Processor - Fold Initialisation

```
def main(filename: String) = {
  val result = getStream( filename ).toList.foldLeft[PpProgram](
    (Nil,Nil).point[PpReader]
  )( (reader,event) => reader.flatMap{ case (errors,outputs) => {
```

- verifyNewElement now returns a list instead of Unit.

Changes to Stream Processor - Fold Initialisation

```
def main(filename: String) = {
  val result = getStream( filename ).toList.foldLeft[PpProgram](
    (Nil,Nil).point[PpReader]
  )( (reader,event) => reader.flatMap{ case (errors,outputs) => {
```

- verifyNewElement now returns a list instead of Unit.
- We now build a `PpReader[(List[String],List[String])]`
  - Aliased to `PpProgram`

Changes to Stream Processor - Fold Initialisation

```
def main(filename: String) = {
  val result = getStream( filename ).toList.foldLeft[PpProgram](
    (Nil,Nil).point[PpReader]
  )( (reader,event) => reader.flatMap{ case (errors,outputs) => {
```

- verifyNewElement now returns a list instead of Unit.
- We now build a `PpReader[(List[String],List[String])]`
  - Aliased to `PpProgram`
- We now fold over the stream instead of mapping

Changes to Stream Processor - Fold Step

```
)( (reader,event) => reader.flatMap{ case (errors,outputs) => {
  val newErrors = verifyNewElement(event)
  val newReader = event match {
    case EvElemStart( _ , l , _ , _ ) => {
      val out = indented( foundElems.size , s"<$l>" )
      foundElems.push( l )
      out
    }
    case EvElemEnd( _ , l ) => {
      foundElems.pop()
      indented( foundElems.size , s"</$l>" )
    }
    case EvText(t) => indented( foundElems.size , t )
  }
  newReader.map(newOutput => (errors ++ newErrors , newOutput :: outputs))
}})
```

Changes to Output

```
val (errors,output) = result( Config( "  " ) )
errors.foreach( System.err.println _ )
output.reverse.foreach( println _ )
```

- No more sequencing since our fold is doing that for us.

Changes to Output

```
val (errors,output) = result( Config( "  " ) )
errors.foreach( System.err.println _ )
output.reverse.foreach( println _ )
```

- No more sequencing since our fold is doing that for us.
- Get back a tuple of both output lists

Changes to Output

```
val (errors,output) = result( Config( "  " ) )
errors.foreach( System.err.println _ )
output.reverse.foreach( println _ )
```

- No more sequencing since our fold is doing that for us.
- Get back a tuple of both output lists
- Need to remember to reverse the output list due to cons-ing

Collecting two lists - Results

- We moved our error list into the monad. Win!

Collecting two lists - Results

- We moved our error list into the monad. Win!
- Each level awkwardly needs to explicitly know about and append the errors.

Collecting two lists - Results

- We moved our error list into the monad. Win!
- Each level awkwardly needs to explicitly know about and append the errors.
- This would be a real pain with lots of nesting to the code.

Introducing the Writer Monad!

- `Writer[L,A]` where there must be a `Monoid[L]`

Introducing the Writer Monad!

- `Writer[L,A]` where there must be a `Monoid[L]`
  - `List("msg").tell` to log and return unit

Introducing the Writer Monad!

- `Writer[L,A]` where there must be a `Monoid[L]`
  - `List("msg").tell` to log and return unit
  - `5.set(List("msg"))` to log and return 5

Introducing the Writer Monad!

- `Writer[L,A]` where there must be a `Monoid[L]`
  - `List("msg").tell` to log and return unit
  - `5.set(List("msg"))` to log and return 5
  - Monoid.mappend used to append log state together

Introducing the Writer Monad!

- `Writer[L,A]` where there must be a `Monoid[L]`
  - `List("msg").tell` to log and return unit
  - `5.set(List("msg"))` to log and return 5
  - Monoid.mappend used to append log state together
- We just log and forget! Just like a logger.warn("msg") !

Introducing the Writer Monad!

- `Writer[L,A]` where there must be a `Monoid[L]`
  - `List("msg").tell` to log and return unit
  - `5.set(List("msg"))` to log and return 5
  - Monoid.mappend used to append log state together
- We just log and forget! Just like a logger.warn("msg") !

```
type PpWriter[+A] = Writer[List[String],A]
```

Introducing the Writer Monad!

- `Writer[L,A]` where there must be a `Monoid[L]`
  - `List("msg").tell` to log and return unit
  - `5.set(List("msg"))` to log and return 5
  - Monoid.mappend used to append log state together
- We just log and forget! Just like a logger.warn("msg") !

```
type PpWriter[+A] = Writer[List[String],A]
```

```
def verifyNewElement( event: XMLEvent ): PpWriter[Unit] = {
  (foundElems.headOption,event) match {
    case (Some("msg"),EvElemStart( _ , l , _ , _ )) => List(
      s"WARN: Msg should only contain text, contains: <$l>"
    ).tell
    case _ => Nil.tell
  }
}
```

Changes to our stream fold

- Abstract out the event pattern match into a separate function

Changes to our stream fold

- Abstract out the event pattern match into a separate function

```
def indentEvent( event: XMLEvent ): PpReader[String] = event match {
  case EvElemStart( _ , l , _ , _ ) => {
    val out = indented( foundElems.size , s"<$l>" )
    foundElems.push( l )
    out
  }
  case EvElemEnd( _ , l ) => {
    foundElems.pop()
    indented( foundElems.size , s"</$l>" )
  }
  case EvText(t) => indented( foundElems.size , t )
}
```

Changes to our stream fold

- For each event:

Changes to our stream fold

- For each event:
  - All we really want to do is append the new output string to the output.

Changes to our stream fold

- For each event:
  - All we really want to do is append the new output string to the output.
  - To do this we need to join the inner and outer monads.

Changes to our stream fold

- For each event:
  - All we really want to do is append the new output string to the output.
  - To do this we need to join the inner and outer monads.

```
val reader = getStream( filename ).toList.foldLeft[PpProgram](
  ( Nil.point[PpWriter] ).point[PpReader]
)( (reader,event) => reader.flatMap( writer => {
  val newReader = indentEvent( event )
  val newWriter = verifyNewElement( event )

  newReader.map( newOutput =>
    writer.flatMap( outputs =>
      newWriter.map( _ => newOutput :: outputs )
    )
  )
}))
```

Changes to output

Changes to output

- When we inject the config into the reader we now get back a writer.

Changes to output

- When we inject the config into the reader we now get back a writer.
- `writer.run` runs the writer and returns a tuple of logged state and output.

Changes to output

- When we inject the config into the reader we now get back a writer.
- `writer.run` runs the writer and returns a tuple of logged state and output.

```
val (errors,lines) = reader( Config( "  " ) ).run
errors.foreach( System.err.println _ )
lines.reverse.foreach( println _ )
```

Writer Monad - Results

- The appending and passing around of logs is now hidden. Win!

Writer Monad - Results

- The appending and passing around of logs is now hidden. Win!
- The nested monads are very awkward, though. We can still do better.

Introducing Monad Transformers

- A way to stack monads on top of each other to feel like one monad.

Introducing Monad Transformers

- A way to stack monads on top of each other to feel like one monad.
- Gets rid of the nesting when mapping / flatmapping

Introducing Monad Transformers

- A way to stack monads on top of each other to feel like one monad.
- Gets rid of the nesting when mapping / flatmapping
- No general transformer. Written for each monad.

Introducing Monad Transformers

- A way to stack monads on top of each other to feel like one monad.
- Gets rid of the nesting when mapping / flatmapping
- No general transformer. Written for each monad.
- To mimic what we had before, we want these types:

```
type PpWriter[+A] = Writer[List[String],A]
type PpReaderWriter[+A] = ReaderT[PpWriter,Config,A]
type PpProgram = PpReaderWriter[List[String]]
```

Changes to indented and verifyNewElement

- Everything is in PpReaderWriter

Changes to indented and verifyNewElement

- Everything is in PpReaderWriter
- Note: Kleisli ≡ ReaderT

```
def indented( level: Int, text: String ):PpReaderWriter[String] = Kleisli{
  (conf) => ((conf.indentSeq * level) + text).point[PpWriter]
}

def verifyNewElement( event: XMLEvent ): PpReaderWriter[Unit] = Kleisli{ _ =>
   (foundElems.headOption,event) match {
     case (Some("msg"),EvElemStart( _ , l , _ , _ )) => List(
       s"WARN: Msg should only contain text, contains: <$l>"
     ).tell
     case _ => Nil.tell
   }
}
```

Changes to stream processor

- Interleaving is gone!

```
def main(filename: String) = {
  val readerWriter = getStream( filename ).toList.foldLeft[PpProgram](
    Nil.point[PpReaderWriter]
  )( (readerWriter,event) => for{
      output    <- readerWriter
      newOutput <- indentEvent( event )
      _         <- verifyNewElement( event )
    } yield newOutput :: output
  )

  val (errors,lines) = readerWriter( Config( "  " ) ).run
  errors.foreach( System.err.println _ )
  lines.reverse.foreach( println _ )

}
```

ReaderWriter Results

- We have our reader and writer enabled program as nice as we can.
- Lets move on to get rid of the mutable stack!

Removing the mutable stack

- We need to bring the stack into our monad so that we have a completely sealed monad.

Removing the mutable stack

- We need to bring the stack into our monad so that we have a completely sealed monad.
- Lets just try threading the stack through our fold with this structure:

Removing the mutable stack

- We need to bring the stack into our monad so that we have a completely sealed monad.
- Lets just try threading the stack through our fold with this structure:

```
type PpWriter[+A] = Writer[List[String],A]
type PpReaderWriter[+A] = ReaderT[PpWriter,Config,A]
type PpReaderWriterState[+A] = PpReaderWriter[(Stack[String],A)]
type PpProgram = PpReaderWriterState[List[String]]
```

Changes to indentEvent

- Each ReaderWriter returns a tuple of the new stack and output

Changes to indentEvent

- Each ReaderWriter returns a tuple of the new stack and output

```
def indentEvent(
  foundElems: Stack[String]
  , event: XMLEvent
):PpReaderWriterState[String] = event match {
  case EvElemStart( _ , l , _ , _ ) => {
    val out = indented( foundElems.size , s"<$l>" )
    out.map( ( foundElems.push( l ) , _ ) )
  }
  case EvElemEnd( _ , l ) => {
    val newStack = foundElems.pop
    indented( newStack.size , s"</$l>" ).map( ( newStack , _ ) )
  }
  case EvText(t) => indented( foundElems.size , t ).map( ( foundElems , _ ) )
}
```

Changes to processor

- Fold has to pass the stack through and append output

Changes to processor

- Fold has to pass the stack through and append output

```
val readerWriter = getStream( filename ).foldLeft[PpProgram](
  Kleisli{ _ => (Stack.empty , Nil).point[PpWriter] }
)( (rw,event) => {
  for{
    s1 <- rw
    s2 <- indentEvent( s1._1 , event )
    _  <- verifyNewElement( s1._1 , event )
  } yield (s2._1,s2._2::s1._2)
} )

val (errors,(_,lines)) = readerWriter.run( Config("  ") ).run
errors.foreach( System.err.println _ )
lines.reverse.foreach( println _ )
```

Explicit State Threading - Results

- Passing through the state each step is really annoying

Explicit State Threading - Results

- Passing through the state each step is really annoying
- Multiple levels of this state would be impractical.

Introducing the state monad

- This monad composes functions, much like reader.

Introducing the state monad

- This monad composes functions, much like reader.
- State functions are of the form `State(s => (newState,output))`

Introducing the state monad

- This monad composes functions, much like reader.
- State functions are of the form `State(s => (newState,output))`
- We can modify the state for the next thing in the chain.

Introducing the state monad

- This monad composes functions, much like reader.
- State functions are of the form `State(s => (newState,output))`
- We can modify the state for the next thing in the chain.
- Lets add it to our existing transformer stack in the pretty printer:

Introducing the state monad

- This monad composes functions, much like reader.
- State functions are of the form `State(s => (newState,output))`
- We can modify the state for the next thing in the chain.
- Lets add it to our existing transformer stack in the pretty printer:

```
type ElemStack = Stack[String]
type PpWriter[+A] = Writer[List[String],A]
type PpState[+A] = StateT[PpWriter,ElemStack,A]
type PpReaderWriterState[+A] = ReaderT[PpState,Config,A]
type PpProgram = PpReaderWriterState[List[String]]
```

Build our stateful primitives

Can reuse these anywhere in our program.

```
def stackHeight: PpReaderWriterState[Int] =
  Kleisli[PpState,Config,Int]{ _ =>
    StateT[PpWriter,ElemStack,Int]{ s => (s,s.size).point[PpWriter] }
  }

def popStack: PpReaderWriterState[Unit] =
  Kleisli[PpState,Config,Unit]{ _ =>
    StateT[PpWriter,ElemStack,Unit]{ s => (s.pop,()).point[PpWriter] }
  }

def pushStack( newElem:String ): PpReaderWriterState[Unit] =
  Kleisli[PpState,Config,Unit]{ _ =>
    StateT[PpWriter,ElemStack,Unit]{
      s => (s.push(newElem),()).point[PpWriter]
    }
  }
```

New Indenter

```
def getIndentSeq: PpReaderWriterState[String] =
  Kleisli.ask[PpState,Config].map( _.indentSeq )

def indented( text: String ):PpReaderWriterState[String] = for {
  level     <- stackHeight
  indentSeq <- getIndentSeq
} yield (indentSeq * level) + text
```

- Our primitives come into play here.

New Indenter

```
def getIndentSeq: PpReaderWriterState[String] =
  Kleisli.ask[PpState,Config].map( _.indentSeq )

def indented( text: String ):PpReaderWriterState[String] = for {
  level     <- stackHeight
  indentSeq <- getIndentSeq
} yield (indentSeq * level) + text
```

- Our primitives come into play here.
- Do notation nicely hiding the fact that there are monads at all!

New verifyNewElement

```
def verifyNewElement( event: XMLEvent ): PpReaderWriterState[Unit] =
  Kleisli[PpState,Config,Unit]{ _ =>
    StateT[PpWriter,ElemStack,Unit]{ foundElems =>
      (foundElems,()).set(
        (foundElems.headOption,event) match {
          case (Some("msg"),EvElemStart( _ , l , _ , _ )) => List(
            s"WARN: Msg should only contain text, contains: <$l>"
          )
          case _ => Nil
        }
      )
    }
  }
```

- The layers are making this pretty ugly.

New indentEvent

```
def indentEvent( event: XMLEvent ):PpReaderWriterState[String] =
  event match {
    case EvElemStart( _ , l , _ , _ ) => for{
      line <- indented( s"<$l>" )
      _    <- pushStack( l )
    } yield line
    case EvElemEnd( _ , l ) => for {
      _    <- popStack
      line <- indented( s"</$l>" )
    } yield line
    case EvText(t) => indented( t )
  }
```

- This is very easy to read, and is not too dissimilar from our original.

New Fold

```
val program = getStream( filename ).foldLeft[PpProgram](
  Kleisli{ _ => Nil.point[PpState] }
)( (s,event) =>
  for {
    output    <- s
    newOutput <- indentEvent( event )
    _         <- verifyNewElement( event )
  } yield newOutput :: output
)

val (errors,(_,lines)) = program.run( Config("  ") ).run( Stack.empty ).run
errors.foreach( System.err.println _ )
lines.reverse.foreach( println _ )
```

- Stack state is gone from our fold.
- Stack initialized at end like the config.

State Monad - Results

- All state is now encapsulated without our monad.

State Monad - Results

- All state is now encapsulated without our monad.
- Thanks to the primitives, our logic code is clean and 'monad free'

State Monad - Results

- All state is now encapsulated without our monad.
- Thanks to the primitives, our logic code is clean and 'monad free'
- The primitives,however, are a bit nasty to create.

State Monad - Results

- All state is now encapsulated without our monad.
- Thanks to the primitives, our logic code is clean and 'monad free'
- The primitives,however, are a bit nasty to create.
- Because our monads are function based, can't shortcut the layers very well with point and liftM.

State Monad - Results

- All state is now encapsulated without our monad.
- Thanks to the primitives, our logic code is clean and 'monad free'
- The primitives,however, are a bit nasty to create.
- Because our monads are function based, can't shortcut the layers very well with point and liftM.
- Would be nice to flatten this even more.

Introducing ReaderWriterState

- Instead of being layers of Reader, Writer and State, this is just one layer.

Introducing ReaderWriterState

- Instead of being layers of Reader, Writer and State, this is just one layer.
- Composes functions of the form:
  `ReaderWriterState( (r,s) => (newLog , output, newState) )`

Introducing ReaderWriterState

- Instead of being layers of Reader, Writer and State, this is just one layer.
- Composes functions of the form:
  `ReaderWriterState( (r,s) => (newLog , output, newState) )`
- Gives us the same functionality, but easier to construct our functions.

Introducing ReaderWriterState

- Instead of being layers of Reader, Writer and State, this is just one layer.
- Composes functions of the form:
  `ReaderWriterState( (r,s) => (newLog , output, newState) )`
- Gives us the same functionality, but easier to construct our functions.
- This shortcut exists because RWS stacks are very common and should be as nice as possible.

Primitive Improvements

```
def stackHeight: PpReaderWriterState[Int] = ReaderWriterState{
  (r,s) => ( Nil , s.size, s )
}
```

```
def verifyNewElement( event: XMLEvent ): PpReaderWriterState[Unit] =
  ReaderWriterState {
    (r,foundElems) => {
      val newLog = (foundElems.headOption,event) match {
        case (Some("msg"),EvElemStart( _ , l , _ , _ )) => List(
          s"WARN: Msg should only contain text, contains: <$l>"
        )
        case _ => Nil
      }
      (newLog,(),foundElems)
    }
  }
```

Conclusions

Conclusions

- Our pure code still carries state through the program.

Conclusions

- Our pure code still carries state through the program.
- All we have done is enforce that side effects must be isolated and serialised.

Conclusions

- Our pure code still carries state through the program.
- All we have done is enforce that side effects must be isolated and serialised.
- We're also being explicit about what kind of state a piece of code depends on.

Conclusions

- Our pure code still carries state through the program.
- All we have done is enforce that side effects must be isolated and serialised.
- We're also being explicit about what kind of state a piece of code depends on.
- Because the state is restricted and explicit, it is easier to reason about and is safer.

Conclusions

- Our pure code still carries state through the program.
- All we have done is enforce that side effects must be isolated and serialised.
- We're also being explicit about what kind of state a piece of code depends on.
- Because the state is restricted and explicit, it is easier to reason about and is safer.
- Even though we have these restrictions, our original algorithm still remains the same.

Conclusions

- Our pure code still carries state through the program.
- All we have done is enforce that side effects must be isolated and serialised.
- We're also being explicit about what kind of state a piece of code depends on.
- Because the state is restricted and explicit, it is easier to reason about and is safer.
- Even though we have these restrictions, our original algorithm still remains the same.
- In our imperative code, we were taking risks and burdening ourselves with flexibility that wasn't even necessary!

Three Points I hope I left with you

Three Points I hope I left with you

- The motivations for ReaderWriterState and how it is used.

Three Points I hope I left with you

- The motivations for ReaderWriterState and how it is used.
- Isolation of side effects is a good idea.

Three Points I hope I left with you

- The motivations for ReaderWriterState and how it is used.
- Isolation of side effects is a good idea.
- Types and FP don't get in your way: they keep you honest and help reduce bugs.

Further Reading

Further Reading

- LYAHFGG

Further Reading

- LYAHFGG
- FP in Scala

The end

- Thanks for listening!