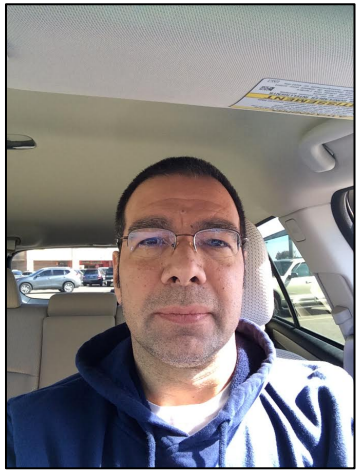
Writer Monad

Learn how to use the Writer monad to log (trace) the execution of functions through the work of

Bartosz Milewski
@BartoszMilewski



Alvin Alexander
@alvinalexander

slides by @philip_schwarz

Bartosz Milewski's explanation of how to use a **Writer monad** to model the side effects of functions that log or trace their execution

```
bool negate(bool b) {
    return !b;
}
```

```
string logger;

bool negate(bool b) {
    logger += "Not so! ";
    return !b;
}
```

Not a **pure function** – has **side effects**. **In modern programming, we try to stay away from global mutable state as much as possible**. Fortunately for us, it's possible to make this function pure. You just have to pass the log explicitly, in and out. Let's do that by adding a string argument, and pairing regular output with a string that contains the updated log

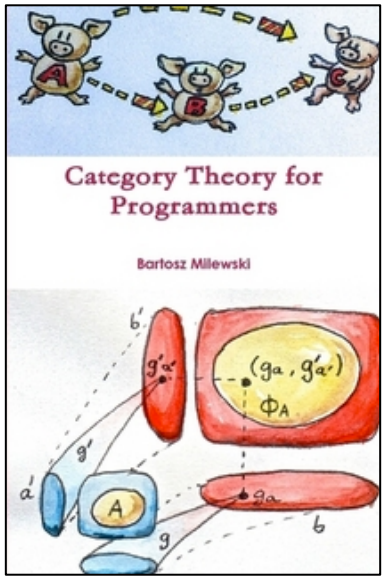
```
pair<bool, string> negate(bool b, string logger) {
    return make_pair(!b, logger + "Not so! ");
}
```

YouTube Category Theory 3.2 – Kleisli Category

@BartoszMilewski

This function is pure, it has no side effects, it returns the same pair every time it's called with the same arguments, and it can be memoized if necessary. However, considering the cumulative nature of the log, you'd have to memoize all possible histories that can lead to a given call. There would be a separate memo entry for: `negate(true, "It was the best of times. ");` and `negate(true, "It was the worst of times. ");` and so on.

It's also **not a very good interface for a library function**. The callers are free to ignore the string in the return type, so that's not a huge burden; but they are forced to pass a string as input, which might be inconvenient.



Is there a way to do the same thing less intrusively? Is there a way to separate concerns? In this simple example, **the main purpose of the function negate is to turn one Boolean into another**. The **logging is secondary**. Granted, the message that is logged is specific to the function, but the task of aggregating the messages into one continuous log is a **separate concern**. **We still want the function to produce a string, but we'd like to unburden it from producing a log**.

```
pair<bool, string> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

So here's the **compromise solution**: The idea is that **the log will be aggregated between function calls**.

Bartosz Milewski's example of two embellished functions and how to compose them

```
bool isEven(int n) {
    return n % 2 == 0;
}

bool negate(bool b) {
    return !b;
}
```

functions we want to add logging to

We want to modify these functions so that they **piggyback** a message string on top of their regular return values.

We will "**embellish**" the return values of these functions.

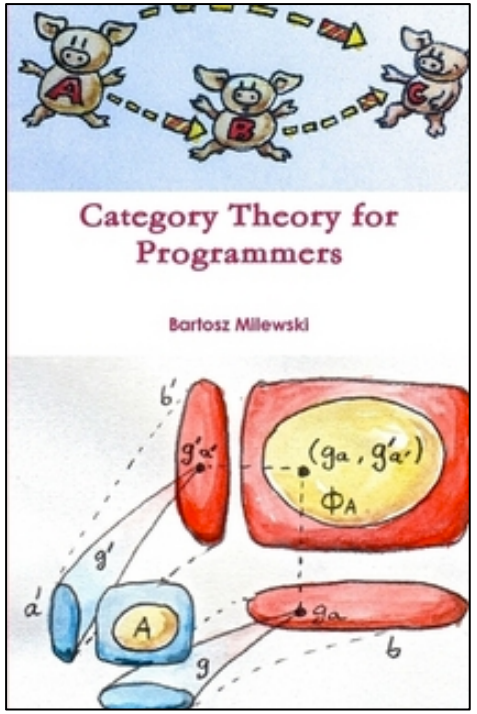
@BartoszMilewski

```
pair<bool, string> isEven(int n) {
    return make_pair(n % 2 == 0, "isEven ");
}
pair<bool, string> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

embellished functions

```
pair<bool, string> isOdd(int n) {
    pair<bool, string> p1 = isEven(n);
    pair<bool, string> p2 = negate(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

composing the embellished functions



YouTube Category Theory 3.2 – Kleisli Category

```
bool isEven(int n) {
    return n % 2 == 0;
}

bool negate(bool b) {
    return !b;
}
```

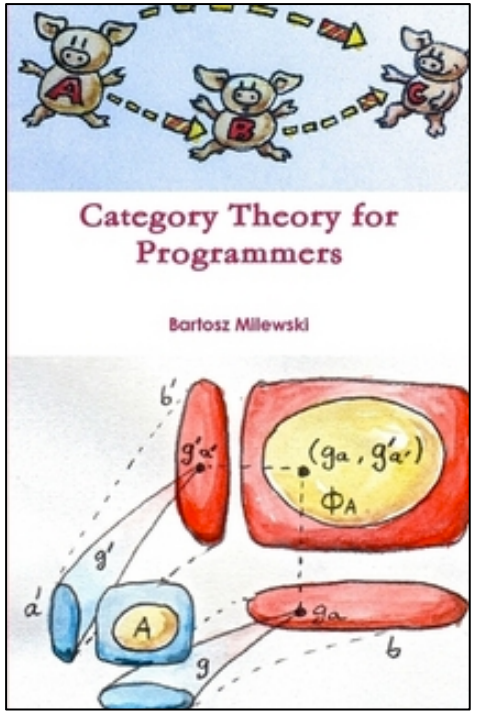
Here we "**embellish**" the return values of functions **isEven** and **negate in a generic way** by defining a template **Writer** that **encapsulates** a **pair** whose first component is a value of **arbitrary type A** and the second component is a **String**.

@BartoszMilewski

```
template<class A>
using Writer = pair<A, string>;
```

```
Writer<bool> isEven(int n) {
    return make_pair(n % 2 == 0, "isEven ");
}
Writer<bool> negate(bool b) {
    return make_pair(!b, "Not so! ");
}
```

piggybacking a message string on top of regular return values using a **Writer** template.




```
Writer<bool> isOdd(int n) {
    Writer<bool> p1 = isEven(n);
    Writer<bool> p2 = negate(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

YouTube Category Theory 3.2 – Kleisli Category

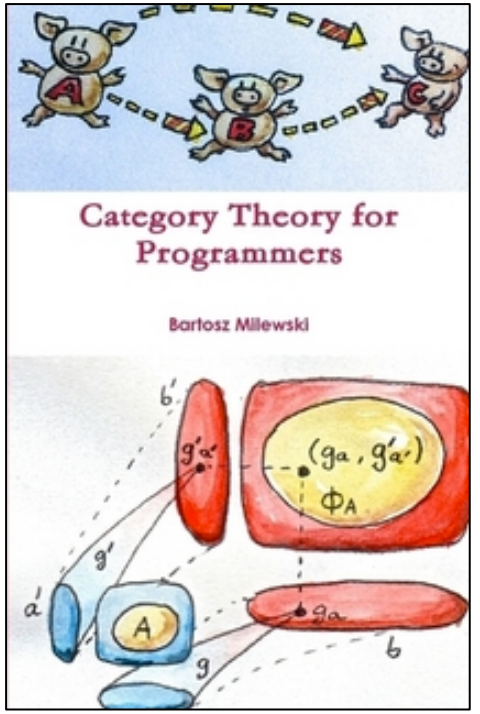
```
Writer<bool> isOdd(int n) {
    Writer<bool> p1 = isEven(n);
    Writer<bool> p2 = negate(p1.first);
    return make_pair(p2.first, p1.second + p2.second);
}
```

Now **imagine a whole program written in this style**. It's **a nightmare of repetitive, error-prone code**. But we are programmers. We know how to deal with repetitive code: **we abstract it!** This is, however, **not your run of the mill abstraction — we have to abstract function composition itself**.

@BartoszMilewski

If we want to **abstract this composition as a higher order function** in C++, we have to use a template parameterized by three types corresponding to three objects in our category. It should take two **embellished functions** that are **composable** according to our rules, and return a third **embellished function**:

```
template<class A, class B, class C>
function<Writer<C>(A)> compose(function<Writer<B>(A)> m1,function<Writer<C>(B)> m2)
{
    return [m1, m2](A x) {
        auto p1 = m1(x);
        auto p2 = m2(p1.first);
        return make_pair(p2.first, p1.second + p2.second);
    };
}
```



Now we can go back to our earlier example and implement the **composition** of **isEven** and **negate** using this new template:

```
Writer<bool> process(string s) {
    return compose<string, string, bool>(isEven, negate)(s);
}
```

The particular monad that I used as the basis of the category in this post is called the **writer monad** and it's **used for logging or tracing the execution of functions**. **It's also an example of a more general mechanism for embedding effects in pure computations**

YouTube Category Theory 3.2 – Kleisli Category

So it turns out, this is really a miracle, I would say, that everything that can be computed using impure functions, state, exceptions, input/output et cetera, they can all be converted into pure calculation as long as you replace regular functions with these functions that return embellished results.

So all these side effects can be translated into some kind of embellishment of the result of a function, and the function remains a pure function, but it produces more than the result, it produces a result that's hidden, embellished, encapsulated in some way, in some weird way.

So this is the interesting part: you have a computation that normally an imperative programmer would implement as an impure function and the functional programmer comes along and says I can implement this as a pure function, it's just that the output type will be a little bit different.

And it works. And this still has nothing to do with the monad.

It just says: impure computation that we do in imperative programming can be transformed into pure computations in functional programming, but they return these embellished results.

And where does the monad come in? Monads come in when we say ok, but I have these gigantic function that starts with some argument a and produces this embellished result, and do I have to just write them inline, for a 1000 lines of code, to finally produce this result?

No, I want to split it into pieces, chop it into little pieces. I want to chop a function that produces side effects into 100 functions that produce side effects and combine them, compose them.

So this is where monads come in. The monad says, well you can take this gigantic computation, pure computation, and split it into smaller pieces and then when you want to finally compose this stuff, well then use the monad.

So this is what the monad does: it just glues together, it lets you split your big computation into smaller computations and glue them together.

@BartoszMilewski

YouTube Category Theory 10.1: Monads

So this is **what the monad does**: **it just glues together**, it lets you split your big computation into smaller computations and **glue them together**.

The hard part of functional programming involves how you glue together all of your pure functions to make a complete application. Because this process feels like you're writing "**glue**" code, I refer to this process as "**gluing,**" and as I learned, **a more technical term for this is called "binding."** This process is what the remainder of this book is about…**in Scala/FP this binding process involves the use of for expressions**.

Alvin Alexander @alvinalexander

Life is good when the output of one function matches the input of another

```
def f(a: Int): Int = ???
def g(a: Int): Int = ???
```

Because the output of **f** is a perfect match to the input of **g**, you can write this code:

```
def f(a: Int): Int = a * 2
def g(a: Int): Int = a * 3
val x = g(f(100))
println(x)
```

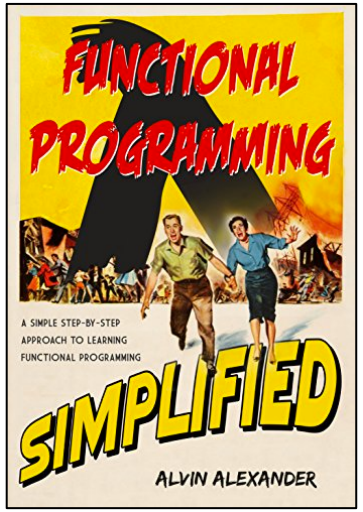
Because **f**(100) is 200 and **g** of 200 is 600, this code prints 600. So far, so good.

A new problem

Next, imagine a slightly more complicated set of requirements where **f** and **g** still take an Int as input, but **now they return a String in addition to an Int**. With this change their signatures look like this:

```
def f(a: Int): (Int, String) = ???
def g(a: Int): (Int, String) = ???
```

f and **g** are functions which, in addition to returning their result (the **Int**), return some information (the **String**). e.g. in a rules engine, the information returned by a function could be a logical explanation of how it came up with its result.



While it's nice to get a log message back from the functions, this also creates a **problem: I can no longer use the output of f as the input to g because g takes an Int input parameter, but f now returns (Int, String)**

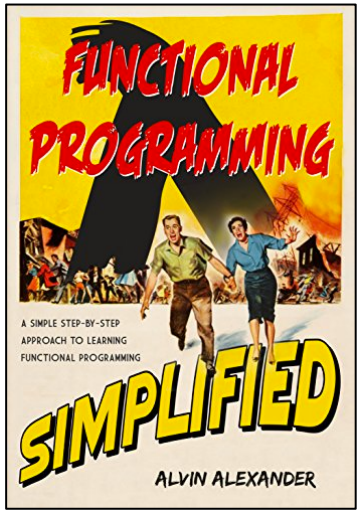
Here is an example of how we can solve the problem manually:

```
def f(a: Int): (Int, String) = {
  val result = a * 2
  (result, s"\nf result: $result.")
}

def g(a: Int): (Int, String) = {
  val result = a * 3
  (result, s"\ng result: $result.")
}

// get the output of `f`
val (fInt, fString) = f(100)
// plug the Int from `f` as the input to `g`
val (gInt, gString) = g(fInt)
// create the total "debug string" by manually
// merging the strings from f and g
val debug = fString + " " + gString
println(s"result: $gInt, debug: $debug")
```

The code prints this output:

```
result: 600, debug:
f result: 200.
g result: 600.
```

Alvin Alexander @alvinalexander



Starting to Glue Functions Together

While this approach works for this simple case, **imagine what your code will look like when you need to string many more functions together. That would be an awful lot of manually written (and error-prone) code. We can do better.**

Because Scala supports **higher-order functions** (HOFs), you can improve this situation by **writing a bind function to glue f and g together a little more easily**. For instance, with a properly written **bind function** you can write code like this to **glue together f, g,** and **h** (a new function that has the same signature as **f** and **g**):

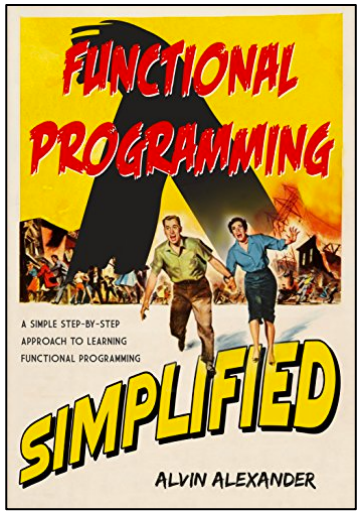
Alvin Alexander @alvinalexander

```
val fResult = f(100)
val gResult = bind(g, fResult)
val hResult = bind(h, gResult)
```

```
def f(a: Int): (Int, String) = {
  val result = a * 2
  (result, s"\nf result: $result.")
}
def g(a: Int): (Int, String) = {
  val result = a * 3
  (result, s"\ng result: $result.")
}
def h(a: Int): (Int, String) = {
  val result = a * 4
  (result, s"\nh result: $result.")
}

// bind, a HOF
def bind(fun: (Int) => (Int, String), tup: (Int, String)): (Int, String) =
{
  val (intResult, stringResult) = fun(tup._1)
  (intResult, tup._2 + stringResult)
}
```

What can we say about bind at this point? First, a few good things:

- It's a **useful higher-order function** (HOF)
- It gives us a way to **bind/glue** the functions **f**, **g**, and **h**
- It's **simpler** and **less error-prone** than the code at the end of the previous lesson



64

The "Bind" Concept

```
val fResult = f(100)
val gResult = bind(g, fResult)
val hResult = bind(h, gResult)
println(s"result: ${hResult._1}, debug: ${hResult._2}")
```

result: 2400, debug:
f result: 200.
g result: 600.
h result: 2400.

```
val (fInt, fString) = f(100)
val (gInt, gString) = g(fInt)
val (hInt, hString) = h(gInt)
val debug = fString + " " + gString + " " + hString
println(s"result: $hInt, debug: $debug")
```

Adapting Alvin Alexander's example by introducing Bartosz Milewski's `Writer` type alias for `(A,String)`

```scala
type Writer[A] = (A, String)

object Writer {
  def apply[A](a:A, log:String) = (a, log)
}

def f(a: Int): Writer[Int] = {
  val result = a * 2
  Writer(result, s"\nf result: $result.")
}

def g(a: Int): Writer[Int] = {
  val result = a * 3
  Writer(result, s"\ng result: $result.")
}

def h(a: Int): Writer[Int] = {
  val result = a * 4
  Writer(result, s"\nh result: $result.")
}

// bind, a HOF
def bind[A,B,C](fun: A => Writer[B], tup: Writer[A]): Writer[B] =
{
  val (intResult, stringResult): Writer[B] = fun(tup._1)
  Writer(intResult, tup._2 + stringResult)
}

val fWriter: Writer[Int] = f(100)
val gWriter: Writer[Int] = bind(g, fWriter)
val hWriter: Writer[Int] = bind(h, gWriter)
println(s"result: ${hWriter._1}, debug: ${hWriter._2}")
```

```scala
// bind is very similar to flatMap – we can go from one to the
// other with a few simple changes
def bind[A,B](fun: A => Writer[B], tup: Writer[A]): Writer[B]

// (1) swap parameters
def bind[A,B](tup: Writer[A], fun: A => Writer[B]): Writer[B]

// (2) rename tup and fun to ma and f
def bind[A,B](ma: Writer[A], f: A => Writer[B]): Writer[B]

// (3) replace Writer with F
def bind[A,B](ma: F[A], f: A => F[B]): F[B]

// (4) rename bind to flatMap
def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
```

Just like **flatMap** first **maps** and then **flattens**

```scala
def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B] = flatten(map(ma)(f))
```

Bind first **maps** the given function onto the given Writer, i.e. applies the function to the writer's value component, and then **flattens**, in that it returns a Writer[A] rather than a Writer[Writer[A]].

Also like **flatMap**, **bind** carries out the extra logic, required to **compose functions embellished with logging** (functions returning a Writer). In this case the extra logic, is the concatenation of two log strings.

```
result: 2400, debug:
f result: 200.
g result: 600.
h result: 2400.
```

Taking Alvin Alexander's sample **f**,**g**,**h** functions and his **bind** function, and adding Bartosz Milewski's **Writer** type alias and his **compose** function

```
type Writer[A] = (A,String)

object Writer {
  def apply[A](a:A, log:String) = (a,log)
}
```

```
def f(a: Int): Writer[Int] = {
  val result = a * 2
  Writer(result, s"\nf result: $result.")
}

def g(a: Int): Writer[Int] = {
  val result = a * 3
  Writer(result, s"\ng result: $result.")
}

def h(a: Int): Writer[Int] = {
  val result = a * 4
  Writer(result, s"\nh result: $result.")
}
```

```
def bind[A,B,C](fun: A => Writer[B], tup: Writer[A]): Writer[B] =
{
  val (intResult, stringResult): Writer[B] = fun(tup._1)
  Writer(intResult, tup._2 + stringResult)
}
```

```
def compose[A,B,C](f: A => Writer[B], g: B => Writer[C]): A => Writer[C] =
(a:A) => {
  val (fVal, fLog) = f(a)
  val (gVal, gLog) = g(fVal)
  Writer(gVal, fLog + gLog)
}
```

Kleisli Composition – composition of embellished functions

```
val fWriter: Writer[Int] = f(100)
val gWriter: Writer[Int] = bind(g, fWriter)
val hWriter: Writer[Int] = bind(h, gWriter)
println(s"result: ${hWriter._1}, debug: ${hWriter._2}")
```

```
val fgh = compose(compose(f, g), h)
val result: Writer[Int] = fgh(100)
println(s"result: ${result._1}, debug: ${result._2}")
```

If there's anything bad to say about bind, it's that it looks like **it's dying to be used in a for expression**, **but because bind doesn't have methods like map and flatMap, it won't work that way**.

For example, wouldn't it be cool if you could write code that looked like this:

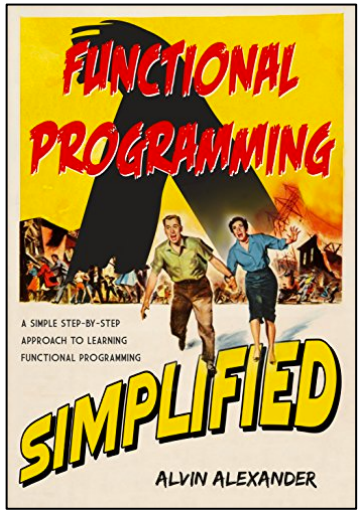
```
val finalResult = for {
  fResult <- f(100)
  gResult <- g(fResult)
  hResult <- h(gResult)
} yield hResult
```

Now we're at a point where we see that **bind is better than what I started with, but not as good as it can be**. That is, I want to use **f**, **g**, and **h** in a **for expression**, but I can't, because **bind** is a function, and therefore it has no way to implement **map** and **flatMap** so it can work in a **for expression**.

What to do?

Well, if we're going to succeed we need to figure out how to create a class that does two things:
1. Somehow works like **bind**
2. Implements **map** and **flatMap** methods so it can work inside **for expressions**

Alvin Alexander @alvinalexander



Instead of these functions returning a tuple, they could return … something else …**a type that implements map and flatMap**.

You could call it a TwoElementWrapper:

```
def f(a: Int): TwoElementWrapper(Int, String)
def g(a: Int): TwoElementWrapper(Int, String)
def h(a: Int): TwoElementWrapper(Int, String)
```

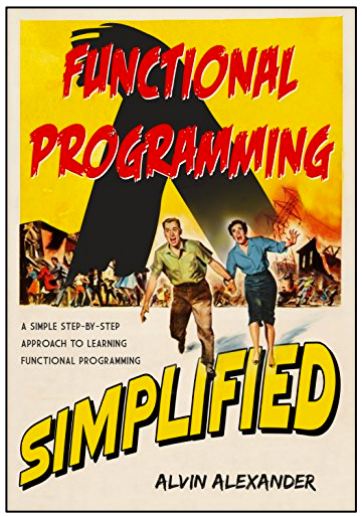
```
def f(a: Int): Debuggable = {
  val result = a * 2
  val message = s"f: a ($a) * 2 = $result."
  Debuggable(result, message)
}

def g(a: Int): Debuggable = {
  val result = a * 3
  val message = s"g: a ($a) * 3 = $result."
  Debuggable(result, message)
}

def h(a: Int): Debuggable = {
  val result = a * 4
  val message = s"h: a ($a) * 4 = $result."
  Debuggable(result, message)
}
```

```
val finalResult = for {
  fResult <- f(100)
  gResult <- g(fResult)
  hResult <- h(gResult)
} yield hResult

println(s"value: ${finalResult.value}\n")
println(s"message: \n${finalResult.message}")
```

value: 2400

message:
*f: a (100) * 2 = 200.*
*g: a (200) * 3 = 600.*
*h: a (600) * 4 = 2400.*

Alvin Alexander @alvinalexander



69

Using bind in a for Expression

But that's not very elegant. When you think about it, the purpose of **f, g**, and **h** is to **show how functions can return "debug" information in addition to their primary return value**, so a slightly more accurate name is **Debuggable**:

```
def f(a: Int): Debuggable(Int, String)
def g(a: Int): Debuggable(Int, String)
def h(a: Int): Debuggable(Int, String)
```

If **Debuggable** implements **map** and **flatMap**, this design will let **f**, **g**, and **h** be used in a **for expression**.

Now all that's left to do is to create **Debuggable**.

In **Scala**, a **monad** consists of a class with **map** and **flatMap** methods, along with some form of a "lift" function.

In **Scala**, a class built like this is intended to be used in **for expressions**.

```
case class Debuggable (value: Int, message: String) {

  def map(f: Int => Int): Debuggable = {
    val newValue = f(value)
    Debuggable(newValue, message)
  }

  def flatMap(f: Int => Debuggable): Debuggable = {
    val newValue: Debuggable = f(value)
    Debuggable(newValue.value, message + "\n" + newValue.message)
  }
}
```

```scala
case class Debuggable[A](value: A, log: List[String]) {

  def map[B](f: A => B): Debuggable[B] = {
    val nextValue = f(value)
    Debuggable(nextValue, this.log)
  }

  def flatMap[B](f: A => Debuggable[B]): Debuggable[B] = {
    val nextValue: Debuggable[B] = f(value)
    Debuggable(nextValue.value, this.log ::: nextValue.log)
  }

}

def f(a: Int): Debuggable[Int] = {
  val result = a * 2
  Debuggable(result, List(s"f: multiply $a * 2 = $result"))
}
```

```scala
val finalResult = for {
  fRes <- f(100)
  gRes <- g(fRes)
  hRes <- h(gRes)
} yield s"result: $hRes"
```

```scala
finalResult.log.foreach(l => println(s"LOG: $l"))
println(s"Output is ${finalResult.value}")
```

```
LOG: f: multiply 100 * 2 = 200
LOG: g: multiply 200 * 3 = 600
LOG: h: multiply 600 * 4 = 2400
Output is result: 2400
```

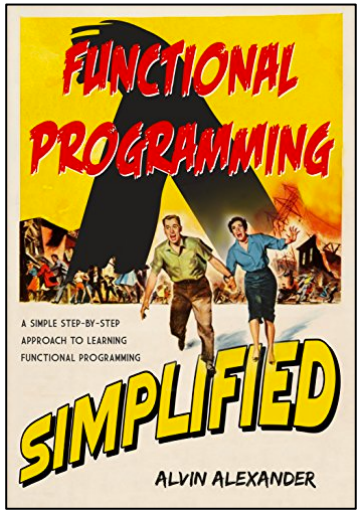
71

A Generic Version of Debuggable

72

One Last Debuggable: Using List Instead of String

Alvin Alexander @alvinalexander



The Writer monad

If you're interested in where the **Debuggable** class comes from, it's actually an implementation of something known as the **Writer monad** in **Haskell**.

As **Learn You a Haskell for Great Good!** states, "**the Writer monad is for values that have another value attached that acts as a sort of log value**. Writer allows us to do computations while making sure that all the log values are combined into one log value that then gets attached to the result."