SilverFOCS Incubator

S-FOCS (tech)

Elegant Coding in Elm

July 12, 2023

Yiming Xiang

Table of Contents

Discussion

Why is duplicated code generally considered bad? (2 min)

Discussion

Why is duplicated code generally considered bad? (2 min)

- **Maintenance**: When the same logic is repeated in multiple places, any changes or bug fixes need to be made in every instance, which increases the risk of introducing inconsistencies or errors. It becomes time-consuming and error-prone to keep the duplicated sections in sync.
- **Code readability and understandability**: It leads to unnecessary code bloat, increases complexity, and reduces the overall clarity of the code. Developers who work on the codebase may have to spend more time deciphering duplicated sections, which slows down the development process.

Design patterns are mainly used to improve code safety and maintainability, including reducing duplicated code.

In this workshop, I will introduce a famous design pattern called *Monad* which can reduce redundant code. Moreover, this design pattern can also be used in imperative programming languages like Javascript and Python.

Table of Contents

Scenario 1: The Sheep Family

One intern wrote the following code to find the maternal grandfather of a sheep. Can you find some issues with respect to the code quality?

```
father : Sheep -> Maybe Sheep -- Omit implementation
mother : Sheep -> Maybe Sheep -- Omit implementation
maternalGrandfather : Sheep -> Maybe Sheep
maternalGrandfather s =
    case mother s of
        Just mom ->
            case father mom of
                Just fa ->
                    fa

                Nothing ->
                    Nothing

        Nothing ->
            Nothing
```

Chaining Maybes

Yes! The problem is obvious: the `Nothing` branch is **duplicated**. Is there any way to avoid this nested pattern matching?

Chaining Maybes

Yes! The problem is obvious: the `Nothing` branch is **duplicated**. Is there any way to avoid this nested pattern matching?

Read Maybe.andThen.

How to use it to make the code better?

Chaining Maybes

Yes! The problem is obvious: the `Nothing` branch is **duplicated**. Is there any way to avoid this nested pattern matching?

Read Maybe.andThen.

How to use it to make the code better?

```
maternalGrandfather : Sheep -> Maybe Sheep
maternalGrandfather s =
    mother s
        |> Maybe.andThen father

-- The same as `Maybe.andThen father (mother s)`
```

Chaining Maybes

Yes! The problem is obvious: the `Nothing` branch is **duplicated**. Is there any way to avoid this nested pattern matching?

Read Maybe.andThen.

How to use it to make the code better?

```
maternalGrandfather : Sheep -> Maybe Sheep
maternalGrandfather s =
    mother s
        |> Maybe.andThen father

-- The same as `Maybe.andThen father (mother s)`
```

What is `|>`?

The Ap Operators

Operators can help us improve code clarity and readability.

```
(|>) : a -> (a -> b) -> b -- Right Application (apR)
(|>) x f =
  f x

(<|) : (a -> b) -> a -> b -- Left Application (apL)
(<|) f x =
  f x
```

The Ap Operators

Operators can help us improve code clarity and readability.

```
(|>) : a -> (a -> b) -> b -- Right Application (apR)
(|>) x f =
  f x

(<|) : (a -> b) -> a -> b -- Left Application (apL)
(<|) f x =
  f x
```

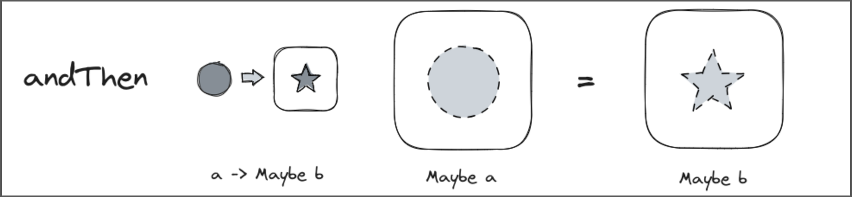
Both operators are used to **reduce parentheses**.
For example, `f1 (f2 (f3 x))` can be rewritten as:

```
f1 <| f2 <| f3 x -- Using <|
x |> f3 |> f2 |> f1 -- Using |>
```

`<|` is equivalent to inserting a left parenthesis here and adding a right parenthesis at the end of the expression.

Illustration of Maybe.andThen

In case you don't understand `Maybe.andThen` well, here is an illustration:



More Chaining

How to get the paternal grandfather of the maternal grandfather (PGMG)?

More Chaining

How to get the paternal grandfather of the maternal grandfather (PGMG)?

```
pgmg : Sheep -> Maybe Sheep
pgmg s =
    mother s
        |> Maybe.andThen father
        |> Maybe.andThen father
        |> Maybe.andThen father
```

More Chaining

How to get the paternal grandfather of the maternal grandfather (PGMG)?

```
pgmg : Sheep -> Maybe Sheep
pgmg s =
    mother s
        |> Maybe.andThen father
        |> Maybe.andThen father
        |> Maybe.andThen father
```

However, there is a problem with `Maybe.andThen`.

More Chaining

How to get the paternal grandfather of the maternal grandfather (PGMG)?

```
pgmg : Sheep -> Maybe Sheep
pgmg s =
    mother s
        |> Maybe.andThen father
        |> Maybe.andThen father
        |> Maybe.andThen father
```

However, there is a problem with `Maybe.andThen`.
How to use previous result in the subsequent chaining? For instance, how to use the maternal grandfather in the last chaining?

Bind Operator

To solve this problem, we introduce the **bind** operator.

Bind Operator

To solve this problem, we introduce the **bind** operator.
The bind function for `Maybe` is:

```
bind : Maybe a -> (a -> Maybe b) -> Maybe b
bind x f =
    Maybe.andThen f x -- Reverse the order of arguments
```

It only **reverses the order** of arguments of `Maybe.andThen`.

Bind Operator

To solve this problem, we introduce the **bind** operator.
The bind function for `Maybe` is:

```
bind : Maybe a -> (a -> Maybe b) -> Maybe b
bind x f =
    Maybe.andThen f x -- Reverse the order of arguments
```

It only **reverses the order** of arguments of `Maybe.andThen`.
Now we can solve the previous problem elegantly:

```
pgmg : Sheep -> Maybe Sheep
pgmg s =
    bind (mother s) <| \x -> -- bind x to mother s
    bind (father x) <| \y -> -- bind y to father s
    bind (father y) <| \z -> -- bind z to father y
    father z
```

Bind Operator

To solve this problem, we introduce the **bind** operator.
The bind function for `Maybe` is:

```
bind : Maybe a -> (a -> Maybe b) -> Maybe b
bind x f =
    Maybe.andThen f x -- Reverse the order of arguments
```

It only **reverses the order** of arguments of `Maybe.andThen`.
Now we can solve the previous problem elegantly:

```
pgmg : Sheep -> Maybe Sheep
pgmg s =
    bind (mother s) <| \x -> -- bind x to mother s
    bind (father x) <| \y -> -- bind y to father s
    bind (father y) <| \z -> -- bind z to father y
    father z
```

What is the type of `x`, `y`, `z`?

Exercise 1: 5 min

Suppose we have a nested list of integers:

```
[ [ [1, 3] ] ] : List (List (List Int))
```

Write a function to get the first element of the nested list (in our example, it's 1).

Try using the bind operator to make your code more elegant.

Table of Contents

Scenario 2: Calculation with Logs

To start with, let's first do some calculations in a function:

```elm
square : Int -> Int
square x =
    x * x

addOne : Int -> Int
addOne x =
    x + 1

simpleCall : Int
simpleCall =
    3 |> square |> addOne -- (3 * 3) + 1
```

Scenario 2: Calculation with Logs

To start with, let's first do some calculations in a function:

```
square : Int -> Int
square x =
    x * x

addOne : Int -> Int
addOne x =
    x + 1

simpleCall : Int
simpleCall =
    3 |> square |> addOne -- (3 * 3) + 1
```

How to add logs while calculating? (5 min)
Expected result:

```
( 10, [ "Square 3", "AddOne 9" ] )
```

First Attempt

```
type alias IntWithLogs =
    ( Int, List String )

square : IntWithLogs -> IntWithLogs
square ( x, o ) =
    ( x * x, o ++ [ "Sqaure " ++ fromInt x ] )

addOne : IntWithLogs -> IntWithLogs
addOne ( x, o ) =
    ( x + 1, o ++ [ "AddOne " ++ fromInt x ] )

callWithLogs : IntWithLogs
callWithLogs =
    ( 3, [] ) |> square |> addOne
```

First Attempt

```elm
type alias IntWithLogs =
    ( Int, List String )

square : IntWithLogs -> IntWithLogs
square ( x, o ) =
    ( x * x, o ++ [ "Sqaure " ++ fromInt x ] )

addOne : IntWithLogs -> IntWithLogs
addOne ( x, o ) =
    ( x + 1, o ++ [ "AddOne " ++ fromInt x ] )

callWithLogs : IntWithLogs
callWithLogs =
    ( 3, [] ) |> square |> addOne
```

Issue: We need to manually create the initial `IntWithLogs`, which can cause duplicated code.

Second Attempt

We need a **wrapping function** that wraps the integer to `IntWithLogs` and then do the calculation.

```
wrapInt : Int -> IntWithLogs
wrapInt x =
    ( x, [] )
```

New calling function:

```
callWithLogs : IntWithLogs
callWithLogs =
    3 |> wrapInt |> square |> addOne
```

Second Attempt

We need a **wrapping function** that wraps the integer to `IntWithLogs` and then do the calculation.

```
wrapInt : Int -> IntWithLogs
wrapInt x =
    ( x, [] )
```

New calling function:

```
callWithLogs : IntWithLogs
callWithLogs =
    3 |> wrapInt |> square |> addOne
```

Issue: The concatenation of list is duplicated in both `square` and `addOne` functions.

Third Attempt

We want to move the logic of concatenation to a new function so that we can reuse it.

Third Attempt

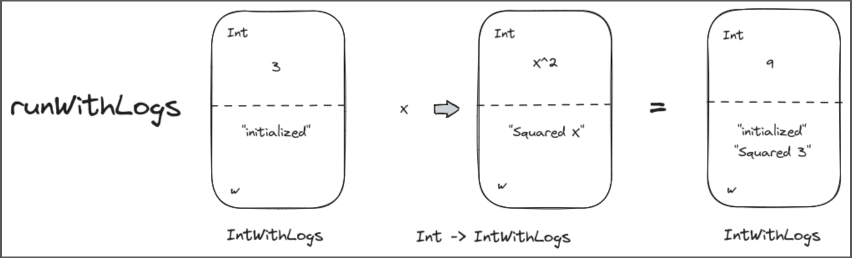
We want to move the logic of concatenation to a new function so that we can reuse it.

```
runWithLogs : IntWithLogs -> (Int -> IntWithLogs) -> IntWithLogs
runWithLogs ( x, o ) transform =
    let
        ( x_, o_ ) =
            transform x
    in
    ( x_, o ++ o_ )
```

Here the `transform` argument can be fed with `square` and `addOne`.

Illustration of runWithLogs

In case you don't understand `runWithLogs` well, here is an illustration:



Third Attempt

Change the remaining part of the code accordingly:

```
square : Int -> IntWithLogs
square x =
    ( x * x, [ "Sqaure " ++ fromInt x ] )

addOne : Int -> IntWithLogs
addOne x =
    ( x + 1, [ "AddOne " ++ fromInt x ] )

callWithLogs : IntWithLogs
callWithLogs =
    let
        x = runWithLogs (wrapInt 3) square

        y = runWithLogs x addOne
    in
        runWithLogs y square
```

Third Attempt

callWithLogs doesn't look very nice. We will change its style to make it more readable:

```
callWithLogs : IntWithLogs
callWithLogs =
    runWithLogs (square 3) <| \x ->
    runWithLogs (addOne x) <| \y ->
    square y
```

This is called the *Monadic* style.

Third Attempt

`callWithLogs` doesn't look very nice. We will change its style to make it more readable:

```
callWithLogs : IntWithLogs
callWithLogs =
    runWithLogs (square 3) <| \x ->
    runWithLogs (addOne x) <| \y ->
    square y
```

This is called the *Monadic* style.
An important difference between the monadic style and the original style is that `x` and `y` are of type `Int` in the former while `IntWithLogs` in the latter. The monadic style allows the direct use of previous result in the subsequent chaining, and is easier to modify.

Third Attempt

callWithLogs doesn't look very nice. We will change its style to make it more readable:

```
callWithLogs : IntWithLogs
callWithLogs =
    runWithLogs (square 3) <| \x ->
    runWithLogs (addOne x) <| \y ->
    square y
```

This is called the *Monadic* style.
An important difference between the monadic style and the original style is that x and y are of type Int in the former while IntWithLogs in the latter. The monadic style allows the direct use of previous result in the subsequent chaining, and is easier to modify.
It is easy to extend the functionalities in this design pattern. You can add calculating functions that only focuses on calculation. All things related to logging are hidden by the runWithLogs function.

Exercise 2: 2 + 5 min

- Find the similarities between this scenario and the first scenario.
- Add new functions `multiple` and `subtract` and test them.

Table of Contents

Monad

Okay, I noticed that there are some similarities between the `Maybe` and `IntWithLogs`. So what are they on earth?

Monad

Okay, I noticed that there are some similarities between the `Maybe` and `IntWithLogs`. So what are they on earth?

They are called *Monads*. It sounds spooky but once you understand it, it is very simple and useful. In FP, you can think of *monad as a strategy for combining computations into more complex computations*.

Monad

Okay, I noticed that there are some similarities between the `Maybe` and `IntWithLogs`. So what are they on earth?

They are called *Monads*. It sounds spooky but once you understand it, it is very simple and useful. In FP, you can think of *monad as a strategy for combining computations into more complex computations*.

You have already learned two monads: the **Maybe** monad and the **Writer** monad.

What is a monad?

A monad contains three things:

[1] An alternative equivalent definition is `join: M (M a) -> M a`.

What is a monad?

A monad contains three things:

- **A type constructor** `M` that takes a type `a` and returns a new type `M a`. You can think of this as a **container** of type `a`, e.g. `Maybe`.
- **A function** `return: a -> M a`. This is used to wrap a value of type `a` into the container `M`.
- **A function** `bind: M a -> (a -> M b) -> M b`[1]. This will bind the result of a computation to some symbol you can use in the second argument function.

[1] An alternative equivalent definition is `join: M (M a) -> M a`.

What is a monad?

A monad contains three things:

- **A type constructor** `M` that takes a type `a` and returns a new type `M a`. You can think of this as a **container** of type `a`, e.g. `Maybe`.
- **A function** `return: a -> M a`. This is used to wrap a value of type `a` into the container `M`.
- **A function** `bind: M a -> (a -> M b) -> M b`[1]. This will bind the result of a computation to some symbol you can use in the second argument function.

Elm doesn't support *Higher Kinded Types* (HKT), i.e. you cannot write the type of `Maybe: * -> *` explicitly in Elm. Therefore, we cannot define a general monad signature/type in Elm.

[1]An alternative equivalent definition is `join: M (M a) -> M a`.

What is a monad?

A monad contains three things:

- **A type constructor** `M` that takes a type `a` and returns a new type `M a`. You can think of this as a **container** of type `a`, e.g. `Maybe`.
- **A function** `return: a -> M a`. This is used to wrap a value of type `a` into the container `M`.
- **A function** `bind: M a -> (a -> M b) -> M b`[1]. This will bind the result of a computation to some symbol you can use in the second argument function.

Elm doesn't support *Higher Kinded Types* (HKT), i.e. you cannot write the type of `Maybe: * -> *` explicitly in Elm. Therefore, we cannot define a general monad signature/type in Elm.

However, we can still write monads in Elm by defining those functions separately for each monad.

[1] An alternative equivalent definition is `join: M (M a) -> M a`.

Some Mindset about Monad

Some important mindset before we go further:

- Monad is not FP only. It is a general design pattern in programming.
- You can think of Monad as an **interface** or abstract class and the `return` and `bind` functions are the abstract methods. Different monad instances have different implementations of those functions.
- One container type **may have** different `return` and `bind` functions. So there might be different monads for the same container type. However, there is usually only one monad for a container type and those functions are determined by the container itself.
- You may have additional helper functions for a monad.

Summary of the Maybe Monad

Computations which may return Nothing.

- **Useful for**: Building computations from sequences of functions that may return Nothing.
- **Type constructor**: `Maybe`.
- **Implementation**.

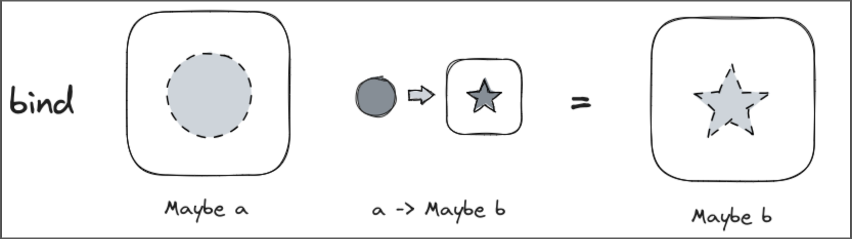


Figure: Binding Strategy of the Maybe monad

Summary of the Writer Monad

Computations which produce a stream of data in addition to the computed values.

- **Useful for**: Logging, or other computations that produce output "on the side".
- **Type constructor**: `Writer`.
- **Implementation**.

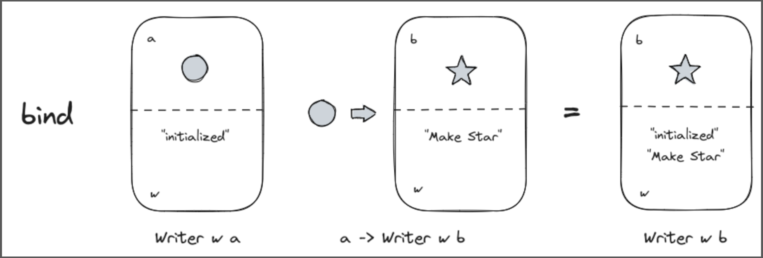


Figure: Binding Strategy of the Writer monad

The Writer Monad: Helper Functions

`runWriter` is a function that extracts the data in the Writer monad.
`tell` is a function that appends a log to the Writer monad.

The Writer Monad: Helper Functions

`runWriter` is a function that extracts the data in the Writer monad.
`tell` is a function that appends a log to the Writer monad.

Since end-users don't need to know the internal structure of the Writer monad, we will show you an example of using the Writer monad.

Try playing with Monad in Elm

elm-monad is an elm package that exposes several useful monads for Elm.

Try playing with Monad in Elm

elm-monad is an elm package that exposes several useful monads for Elm.

Try installing this package in a project:

```
elm install linsyking/elm-monad
```

Try playing with Monad in Elm

elm-monad is an elm package that exposes several useful monads for Elm.

Try installing this package in a project:

```
elm install linsyking/elm-monad
```

To use monads in your module, import related modules:

```elm
import Monad.Writer exposing (..)
```

If you want to use multiple monads in a module, you need to **rename** the module because their functions have conflicts.

Exercises

Exercise 3: 5 min

Use `elm-monad` to rewrite Scenario 1 in the previous example.

Writer Monad Example

Similar to the previous example, we want to add some logs during the calculation.

```
logNumber : Int -> Writer String Int
logNumber x =
    Writer ( x, ["logged " ++ fromInt x] )

mlog : Writer String Int
mlog =
    bind (logNumber 3) <| \a ->
    bind (tell ["hi1"]) <| \_->
    bind (logNumber 5) <| \b ->
    bind (tell ["hi2"]) <| \_->
    return <| a * b
```

Run `mlog` in `elm-repl`:

```
> import Test.WriterTest exposing (..)
> import Monad.Writer exposing (..)
> runWriter mlog
(15,["logged 3","hi1","logged 5","hi2"])
    : ( Int, List String )
```

Exercise 4: 5 min

Write monadic-styled `logNumber` and `multiply` functions by using `return` and `tell`, and test it.

Table of Contents

Summary & More Monads

In this workshop, you learned a new design pattern called Monad. Hopefully it doesn't sound any more mysterious and difficult for you.

If you find monad helpful and want to learn more about monad, you can start by looking at the *State* monad, which is a generalization of the Writer monad and is helpful for writing random generators. You may also refer to the references of this workshop.

- Youtube. The Absolute Best Intro to Monads For Software Engineers
- Haskell Wiki. All About Monads
- Wikipedia. Monad (functional programming)

Thank you!