Если вы видите что-то необычное, просто сообщите мне.

Elm Part 1: Language Basics

Haskell has a number of interesting libraries for frontend web development. Yesod and Snap come to mind. Another option is Reflex FRP which uses GHCJS under the hood.

Each of these has their own strengths and weaknesses. But there are other options as well if we want to write frontend web code while keeping a functional style. This series is all about the Elm language!

I love Elm for a few reasons. Elm builds on my strong belief that we can take the principles of functional programming and put them to practical use. The language is no-nonsense and the documentation is quite good. Elm has a few syntactic quirks. It also lacks a few key Haskell features. And yet, we can still do a lot with it.

In this first part, we'll look at basic installation, and usage, as well as some differences from Haskell. If you're already a little familiar with Elm, you can move onto part 2, where we compose a simple Todo list application in Elm. This will give us a feel for how we architect our Elm applications. We'll wrap up by exploring how to add more effects to our app, and how to integrate Elm types with Haskell.

Frontend is, of course, only part of the story. To learn more about using Haskell for backend web, check out our Haskell Web Series! You can also download our Production Checklist for more ideas!

Also, be sure to check out our Github repository to see some of this example Elm code! This part's code is mostly under the ElmProject folder.

BASIC SETUP

As with any language, there will be some setup involved in getting Elm onto our machine for the first time. For Windows and Mac, you can run the installer program provided here. There are

separate instructions for Linux, but they're straightforward enough. You fetch the binary, tar it, and move to your bin.

Once we have the elm executable installed, we can get going. When you've used enough package management programs, the process gets easier to understand. The elm command has a few fundamental things in common with stack and npm.

First, we can run elm init to create a new project. This will make a src folder for us as well as an elm.json file. This JSON file is comparable to a .cabal file or package.json for Node.js. It's where we'll specify all our different package dependencies. The default version of this will provide most of your basic web packages. Then we'll make our .elm source files in /src.

RUNNING A BASIC PAGE Elm development looks different from most normal Javascript systems I've worked with. While we're writing our code, we don't need to specify a specific entry point to our application. Every file we make is a potential web page we can view. So here's how we can start off with the simplest possible application:

```
import Browser
import HTML exposing (Html, div, text)

type Message = Message

main : Program () Int Message
main =
  Browser.sandbox { init = 0, update = update, view = view }

update : Message -> Int -> Int
update _ x = x

view : Int -> Html Message
view _ = div [] [text "Hello World!"]
```

Elm uses a model/view/controller system. We define our program in the main function. Our Program type has three parameters. The first relates to flags we can pass to our program. We'll ignore those for now. The second is the model type for our program. We'll start with a simple integer. Then the final type is a message. Our view will cause updates by sending messages of this type. The sandbox function means our program is simple, and has no side effects. Aside from passing an initial state, we also pass an update function and a view function.

The update function allows us to take a new message and change our model if necessary. Then the view is a function that takes our model and determines the HTML components. You can read the type of view as "an HTML component that sends messages of type Message.

We can run the elm-reactor command and point our browser at localhost:8000. This takes us to a dashboard where we can examine any file we want. We'll only want to look at the ones with a main function. Then we'll see our simple page with the div on the screen. (It strangely spins if we select a pure library file).

As per the Elm tutorial we can make this more interesting by using the Int in our model. We'll change our Message type so that it can either represent an Increment or a Decrement. Then our update function will change the model based on the message.

```
type Message = Increment | Decrement

update : Message -> Int -> Int
update msg model = case msg of
  Increment -> model + 1
  Decrement -> model - 1

view : Int -> Html Message
view model = div [] [String.fromInt model]
```

As a last change, we'll add + and - buttons to our interface. These will allow us to send the Increment and Decrement messages to our type.

```
view model = div []
  [ button [onClick Decrement] [text "-"]
  , div [] [ text (String.fromInt model) ]
  , button [onClick Increment] [text "+"]
  ]
```

Now we have an interface where we can press each button and the number on the screen will change! That's our basic application!

THE MAKE COMMAND

The elm reactor command builds up a dummy interface for us to use and examine our pages. But our ultimate goal is to make it so we can generate HTML and Javascript from our elm code. We would then export these assets so our back-end could serve them as resources. We can do this with the elm make command. Here's a sample:

```
elm make Main.elm --output=main.html
```

We'll want to use scripting to pull all these elements together and dump them in an assets folder. We'll get some experience with this in a couple weeks when we put together a full Elm + Haskell project.

DIFFERENCES FROM HASKELL

There are a few syntactic gotchas when comparing Elm to Haskell. We won't cover them all, but here are the basics.

We can already see that import and module syntax is a little different. We use the exposing keyword in an import definition to pick out specific expressions we want from that module.

```
import HTML exposing (Html, div, text)

import Types exposing (Message(..))
When we define our own module, we will also use the exposing keyword in place of where in the
module definition:

module Types exposing
  (Message(..))

type Message = Increment | Decrement
```

We can also see that Elm uses type where we would use data in Haskell. If we want a type synonym, Elm offers the type alias combination:

```
type alias Count = Int
```

As you can see from the type operators above, Elm reverses the : operator and ::. A single colon refers to a type signature. Double colons refer to list appending:

```
myNumber : Int
myNumber = 5

myList : [Int]
myList = 5 :: [2, 3]
```

Elm is also missing some of the nicer syntax elements of Haskell. For instance, Elm lacks pattern matching on functions and guards. Elm also does not have where clauses. Only case and let statements exist. And instead of the . operator for function composition, you would use <<. data-preserve-html-node="true" Here are a few examples of these points:

```
isBigNumber : Int -> Bool
isBigNumber x = let forComparison = 5 in x > forComparison

findSmallNumbers : List Int -> List Int
findSmallNumbers numbers = List.filter (not << isBigNumber) numbers
```

As a last note in this section, Elm is strictly evaluated. Elm compiles to Javascript so it can run in browsers. And it's much easier to generate sensible Javascript with a strict language.

ELM RECORDS

Another key difference with Elm is how record syntax works. It Elm, a "record" is a specific type. These simulation Javascript objects. In this example, we define a type synonym for a record. While we don't have pattern matching in general, we can use pattern matching on records:

```
type alias Point2D =
  { x: Float
  , y: Float
  }
```

```
sumOfPoint : Point2D -> Float
sumOfPoint {x, y} = x + y
```

To make our code feel more like Javascript, we can use the . operator to access records in different ways. We can either use the Javascript like syntax, or use the period and our field name as a normal function.

```
point1 : Point2D
point1 = {x = 5.0, y = 6.0}

p1x : Float
p1x = point1.x

p1y : Float
p1y = .y point1
```

We can also update particular fields of records with ease. This approach scales well to many fields:

```
newPoint : Point2D
newPoint = { point1 | y = 3.0 }
```

TYPECLASSES AND MONADS

The more controversial differences between Haskell and Elm lie with these two concepts. Elm does not have typeclasses. For a Haskell veteran such as myself, this is a big restriction. Because of this, Elm also lacks do syntax. Remember that do syntax relies upon the idea that the Monad typeclass exists.

There is a reason for these omissions though. The Elm creator wrote an interesting article about it.

His main point is that (unlike me), most Elm users are coming from Javascript rather than Haskell. They tend not to have much background with functional programming and related concepts. So it's not as big a priority for Elm to capture these constructs. So what alternatives are available?

Well when it comes to typeclasses, each type has to come up with its own definition for a function. Let's take the simple example of map. In Haskell, we have the fmap function. It allows us to apply a

function over a container, without knowing what the container is:

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

We can apply this same function whether we have a list or a dictionary. In Elm though, each library has its own map function. So we have to qualify the usage of it:

```
import List
import Dict

double : List Int -> List Int
double l = List.map (* 2) l

doubleDict : Dict String Int -> Dict String Int
doubleDict d = Dict.map (* 2) d
```

Instead of monads, Elm uses a function called andThen. This acts a lot like Haskell's >>= operator. We see this pattern more often in object oriented languages like Java. As an example from the documentation, we can see how this works with Maybe.

```
toInt : String -> Maybe Int

toValidMonth : Int -> Maybe Int
toValidMonth month =
    if month >= 1 && month <= 12
        then Just month
        else Nothing

toMonth : String -> Maybe Int
toMonth rawString =
    toInt rawString `andThen` toValidMonth
```

So Elm doesn't give us quite as much functional power as we have in Haskell. That said, Elm is a front-end language first. It expresses how to display our data and how we bring components together. If we need complex functional elements, we can use Haskell and put that on the back-end.

CONCLUSION

You're now ready to move onto part 2 of this series! There, we'll expand our understanding of Elm by writing a more complicated program. We'll write a simple Todo list application and see Elm's architecture in action.

To hear more from Monday Morning Haskell, make sure to Subscribe to our newsletter! That will also give you access to our awesome Resources page!

Revision #1
Created 2022-03-11 16:38:57 UTC by gasick
Updated 2022-03-11 17:11:16 UTC by gasick