Elm

FP - Guest lecture

Wouter Swierstra

University of Utrecht

Front-end development

In the course on Webtechnology, you will typically have seen a variety of programming languages for front-end development:

- HTML – a language for describing the structure of webpages;
- CSS – a language for customizing how to render HTML;
- Javascript – a language for manipulating HTML.

Almost all modern webapps are built using some variation of these languages.

Front-end development

In the course on Webtechnology, you will typically have seen a variety of programming languages for front-end development:

- HTML – a language for describing the structure of webpages;
- CSS – a language for customizing how to render HTML;
- Javascript – a language for manipulating HTML.

Almost all modern webapps are built using some variation of these languages.

Do we really need another language?

Elm

http://www.elm-lang.org

Javascript vs Elm

Javascript and Elm are both languages for writing webapplications.

- Javascript is run in your browser; Elm is *compiled* to Javascript.
- Javascript manipulates the DOM directly; Elm does not.
- Javascript is untyped; Elm is statically typed.
- Javascript is imperative; Elm is purely functional.

They are very, very different languages – even if they can be used for the same kind of programs.

Writing applications in Elm

Elm forces you to structure your program in a particular way – sometimes referred to as The Elm Architecture (TEA).

Every program consists of at least three parts:

- The **Model** – describing the current state of your application;
- The **View** – describing how to generate HTML from your model;
- The **Controller** – describing how user interaction updates the model.

Separating these three concerns is A Good Thing.

Hello World!

```elm
type alias Model = String

initialModel : Model
initialModel = "Hello world!"

view : Model -> Html
view m = div [ id "content"]
             [ h1 [] [ text "Hello world!" ]
             , p [] [ text m ]
             ]

main : Html
main = view initialModel
```

Generating HTML

Elm provides a feature rich library for generating HTML.

There are functions for creating every HTML tags, such as `img`, `div`, `p` or `h1`.

Each of these functions takes two arguments:

- a list of attributes.
- a list of child nodes.

You ‘never’ have to write separate HTML pages (although you can if you want).

Elm Types 101

Elm has many built in types such as `String`, `Int`, or `Bool`.

Like C#, you cannot use a `String` when you're expecting a `Bool`. Trying to do so will result in an error at *compile time*.

Javascript will **not** rule out incorrect usage of your types until you run your program.

Using `type alias` you can create a new name for an existing type:

```
type alias Model = String
```

This is similar to Haskell's `type` synonyms.

Generating a webpage

```
> elm make hello.elm
Success! Compiled 1 module.
Successfully generated index.html
> firefox index.html
...
```

Generating a webpage

```
> elm make hello.elm
Success! Compiled 1 module.
Successfully generated index.html
> firefox index.html
...
```

Demo: Let’s have a look!

So far, so boring

We can change the welcome string in our Elm file, recompile and generate a new website.

Or we can have the same string show up in many different places in our HTML document.

But there isn’t much interaction going on…

Reversing strings: model

Let’s write a simple application that reverses the string stored in the model.

Our Model types can stay the same;

```
type alias Model = String


initialModel : Model
initialModel = "Hello world!"
```

Reversing strings: view

- The view should *reverse* the current model

```
view : Model -> Html msg
view m = div [ id "content"]
             [ h1 [myStyle]
                  [ text "My first Elm app" ]
             , p [myStyle]
                 [ text (String.reverse m) ]
             ]
```

- To make it look good, I'll add some CSS given by the attribute `myStyle`...

CSS and Elm

```
myStyle : List (Attribute msg)
myStyle =
  [ style "width" "100%"
  , style "height"  "40px"
  , style "padding" "10px 10px"
  , style "font-size" "2em"
  , style "text-align" "center"
  ]
```

If you want, you can specify all kinds of CSS from Elm.

Usually, it’s good practice to keep this separate in a different .css file.

Demo

Slightly more interesting…

This example shows that we can generate more interesting HTML from a certain model.

But we're still not interacting with the user…

That is there is still no **Controller** in our MVC.

Reversing text the user types

- Let’s add a textfield to our view;
- As the user enters text, we’ll generate new events;
- These events will update the model – setting it to the current text in the text area.

A first approximation

```
view : Model -> Html msg

view m = div [ id "content"]

             [ h1 []

                  [ text "My first Elm app" ]

             , input [ placeholder "Reverse me" ]

                     []

             , p []

                 [ text (String.reverse m) ]

             ]
```

We add one new line to the view – an `input` field where you can enter text

(And I'll leave out the CSS styling from now on)

Recompile and test

If we now recompile our Elm file…

Recompile and test

If we now recompile our Elm file...

We can see a text field just under our title...

Recompile and test

If we now recompile our Elm file…

We can see a text field just under our title…

… but it doesn’t do anything yet.

Adding interactivity

How can we make our webapp **interactive**?

Adding interactivity

How can we make our webapp **interactive**?

We’ve seen the *model* and the *view* – where is the *controller*?

The `update` function

To modify the model, we need to define an update function:

```
update : Msg -> Model -> Model
```

- given the current model,
- and some event of type `Msg`,
- compute a new model.

Who picks the `Msg` type?

The `update` function

To modify the model, we need to define an update function:

```
update : Msg -> Model -> Model
```

- given the current model,
- and some event of type `Msg`,
- compute a new model.

Who picks the `Msg` type?

You do!

Messages and update

In our example, we're only interested in one kind of event: please reverse this string.

```
type alias Msg = String


update : Msg -> Model -> Model
update msg m = String.reverse msg
```

Extending the view

We need to change one line in the `view` function from:

```
input [ myStyle, placeholder "Reverse me" ]
```

to:

```
input [ myStyle, placeholder "Reverse me"
      , onInput identity ]
```

The `onInput` attribute expects a function of type `String -> Msg` as its argument – we pass on the entered input immediately without changing it using the `identity` function.

Putting it all together

We can write a simple interactive Elm program by providing:

- an initial model;
- a view function;
- an update function.

These three are grouped in a record and passed to the `sandbox` function that will initialize your program:

```
main =
  sandbox { model = "",
            view = view,
            update = update }
```

Demo

Elm vs Javascript

As you can see, we do **not** manipulate the DOM directly.

Instead, we only specify:

- how to turn a model into HTML (`view`)
- how events affect the current model (`update`)
- the initial model (`""`)

In the HTML generated by the `view` function, we can describe what kind of messages get generated by clicks, button presses, mouse hovers, etc.

Elm vs Javascript

To react to user events in Javascript, you typically attach a listener to an element of the DOM: ‘when the user clicks on this button call this function.’

In Elm, we simply specify what *messages* are sent to our update function.

We do this by specifying in the *view* what messages to raise when buttons are clicked, etc.

The type of `view`

The actual type of view is a bit more complicated:

```
view : Model -> Html Msg
```

The type `Html Msg` is a HTML document that may send messages of type `Msg`.

In C#, you might write something like `Html<Msg>`.

This ensures that the messages raised by the interactive Html components are of the same type as those that the update function can process.

A note on design choices

I’ve done a terrible job of playing to Elm’s strengths.

Both the model and the message types are `String`s.

It’s all too easy to confuse the two.

A note on design choices

I’ve done a terrible job of playing to Elm’s strengths.

Both the model and the message types are `String`s.

It’s all too easy to confuse the two.

I did this when writing the demo.

Elm’s type system lets us do much, much better.

Counter example

Let’s write a new example webapp that displays the current value of a counter.

We want to provide two buttons, labelled `+` and `-`, that increment and decrement the counter respectively.

How can we implement this in Elm?

Counter example

- Our `Model` can be a simple integer;
- We want to support two *different* kinds of messages:
    - increment the current counter;
    - decrement the current counter.
- And define a `view` function generating the desired HTML.

Counter example: model & view

```
type alias Model = Int

view : Model -> Html Msg
view m =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString m) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Counter example: messages and update

We want to support two different messages.

In Elm we can create an enumeration type as follows:

```elm
type Msg = Increment | Decrement
```

Deconstructing enumerations

To check which message the update has recieved, we use a `case` statement – similar to C#'s `switch`:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

Demo

Strings or enumerations?

Using a new data type `Msg` ensures that we can’t accidentally mix up our model and message.

We *only* have to worry about *exactly* two kinds of messages: `Increment` and `Decrement` when writing the `update` function.

If we allowed arbitrary strings as message, we would have much less precise information.

In general, Elm lets us use precise types to rule out junk values.

Elm: pickier than Haskell

Elm can be very picky about our function definitions.

Suppose, we ‘forget’ about the `Decrement` branch:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1
```

Calling `update Decrement m` will cause run-time crash.

Elm: pickier than Haskell

Elm can be very picky about our function definitions.

Suppose, we ‘forget’ about the `Decrement` branch:

```
update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1
```

Calling `update Decrement m` will cause run-time crash.

But Elm will refuse to compile your program until you have a branch covering every possible value.

How robust is Elm?

More than 100,000 lines of code running in production since 2015.

How robust is Elm?

More than 100,000 lines of code running in production since 2015.

Zero run time crashes.

How efficient is Elm?

If every time `update` is called, we need to recompute the entire DOM, isn't this expensive?

How efficient is Elm?

If every time `update` is called, we need to recompute the entire DOM, isn’t this expensive?

The Elm runtime implemented in Javascript computes the difference between the DOM before and after update.

It doesn’t replace the entire DOM, but rather applies small updates where necessary.

No I/O

This is nice for toy applications, like a the counter-example.

But any realistic front-end will need to communicate with a server.

And Elm does not have an IO monad...

No I/O

This is nice for toy applications, like a the counter-example.

But any realistic front-end will need to communicate with a server.

And Elm does not have an IO monad...

But the Elm architecture I have presented so far is not the whole story...

The Elm Architecture

The `sandbox` function is the simplest way to write your main Elm program. More often, you want to use the following function:

```
element :
    { init : flags -> ( model, Cmd msg )
    , view : model -> Html msg
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    }
    -> Program flags model msg
```

- `init` and `update` now also return a command;
- an extra `subscriptions` field

What do these fields mean?

Subscriptions & Commands

Up till now, events/messages were created once the user clicked a button, entered text, or somehow interacted with website.

Subscriptions and commands are two separate ways that new events can be raised.

- Subscriptions can be used to ‘listen’ for certain events – like the user typing, the clock ticking, etc.
- Commands are typically used to send an (asynchronous) request, for example over HTTP.

Crucially, both subscriptions and commands raise messages that are then processed by the `update` function.

Sending and receiving HTTP requests

To communicate with a server typically happens in two steps:

- At some point during the program's execution, a command is created, e.g., when the user interacts with the GUI, the website loads for the first time, etc. This command is then sent out.
- When sending out the command, you specify which message to create when the response returns. This then triggers another execution of the `update` function.

Example: github user profile

A little website to request a user's public github profile information:

```
type alias Model =
    { inputText : String
    , info : String}


type Msg =
      SendRequest
    | ReqReceived (Result Error String)
    | SetText String
```

The model consists of the current text entered in the input field and the user's info.

There are three kinds of messages: send a request, receive an answer, or update the current value of the inputfield.

Generating messages: reacting to changes to the textfield

We can add a button to lookup the user's data:

```
            ,  button  ([onClick SendRequest]) [ text "Lookup user" ]
```

And extend the update function to handle this request:

```
update msg m = case msg of
  SendRequest ->
    (m, Http.get
        { url = String.append "https://api.github.com/users/" m.inputText
        , expect = Http.expectString ReqReceived})
```

This sends out an HTTP GET request to the Github API.

Upon returning, we will raise the `ReqReceived` message.

Receiving requests

```
ReqReceived res -> ({m | info = Result.withDefault "ERROR" res}, Cmd.none)
```

In the `update` function, we add a new case to handle the `ReqReceived` message.

Upon receiving the request, we simply dump the string into our HTML.

Demo!

Extracting information

```
Http.get
  { url = String.append "https://api.github.com/users/" usr
  , expect = Http.expectJson ReqReceived (field "followers" int)}
```

Of course, it’s nicer to extract specific information from the JSON data we collect. For example, let’s figure out how many followers our user has.

We can also adapt the `view` and `Model` accordingly:

Demo!

Doing better…

There's lots missing from this example:

- we should parse the JSON response and display it better;
- there's no real error handling;
- this is just a single API call – usually we want to have more interesting interaction with the server
- …

There are even richer versions of the Elm Architecture that can let you observe URL changes, interact with Javascript, observe the user clicking on links, etc.

Doing better…

There’s lots missing from this example:

- we should parse the JSON response and display it better;
- there’s no real error handling;
- this is just a single API call – usually we want to have more interesting interaction with the server
- …

There are even richer versions of the Elm Architecture that can let you observe URL changes, interact with Javascript, observe the user clicking on links, etc.

But in principle, this gives a purely functional account of front-end development.

Maintaining the Elm architecture

- Do you want to modify the HTML structure of your webapp?

Maintaining the Elm architecture

- Do you want to modify the HTML structure of your webapp?

Modify the `view` function.

Maintaining the Elm architecture

- Do you want to modify the HTML structure of your webapp?

Modify the `view` function.

- Do you want to keep the HTML intact, but modify the styling?

Maintaining the Elm architecture

- Do you want to modify the HTML structure of your webapp?

Modify the `view` function.

- Do you want to keep the HTML intact, but modify the styling?

Modify the CSS (either specified in Elm or in a separate file).

Maintaining the Elm architecture

- Do you want to modify the HTML structure of your webapp?

Modify the `view` function.

- Do you want to keep the HTML intact, but modify the styling?

Modify the CSS (either specified in Elm or in a separate file).

- Do you want to add a new button?

Maintaining the Elm architecture

- Do you want to modify the HTML structure of your webapp?

Modify the `view` function.

- Do you want to keep the HTML intact, but modify the styling?

Modify the CSS (either specified in Elm or in a separate file).

- Do you want to add a new button?

Add a new type of `Msg`, handle it in the `update` function, and extend the `view` with your new button.

Maintaining the Elm architecture

- Do you want to change the behaviour of a button?

Maintaining the Elm architecture

- Do you want to change the behaviour of a button?

Modify the associated case in the `update` function.

Maintaining the Elm architecture

- Do you want to change the behaviour of a button?

Modify the associated case in the `update` function.

- Do you need further information from a server?

Maintaining the Elm architecture

- Do you want to change the behaviour of a button?

Modify the associated case in the `update` function.

- Do you need further information from a server?

Raise a command; add a new event to process the results.

The Elm Architecture

If you’ve followed MSO, you should be familiar with concepts such as *coupling* and *cohesion*.

The Elm architecture forces you to structure your code in a certain style: the model, view and controller.

Each part has a separate responsibility that is clearly delineated. They interact in a fairly predictable way.

When you want to modify or extend code, you know exactly where to look.

Embedding Elm

You can generate an `index.html` using the command `elm-make`.

But alternatively, you can embed Elm code anywhere in your website.

This allows you to write HTML/CSS/Javascript however you would like.

But add Elm components to an existing website.

```
<script>
  var main = document.querySelector("main")
  var app = Elm.Counter.embed(main)
</script>
```

Zero installation IDE

If you don’t want to install Elm, but want to play around with the language…

- Check out the demos on the Elm homepage `http://elm-lang.org/examples`
- Or try the Elm IDE Ellie: `https://ellie-app.com/new`

Additional Elm features

The language is much richer, providing:

- algebraic data types;
- higher order functions;
- polymorphism;
- pattern matching;
- … and many other ideas borrowed from Haskell and other functional languages.

Why Elm?

Elm provides a safe & robust alternative to Javascript.

If you enjoy functional programming language, such as Haskell, Elm offers a very familiar way to write frontend code running in your browser.

Great power to weight ratio!

Why Elm?

Elm provides a safe & robust alternative to Javascript.

If you enjoy functional programming language, such as Haskell, Elm offers a very familiar way to write frontend code running in your browser.

Great power to weight ratio!

It’s a lot more fun to program in than Javascript!

Alternatives

Elm isn't the only language in this design space.

- Purescript is another functional language, with a richer design than Elm.
- GHC2JS is a fork of the GHC compiler capable of generating Javascript.
- Other functional languages, such as Scala, F#, OCaml/ReasonML are all also exploring this space.

Learn more?

- Check out the Elm homepage `http://www.elm-lang.org`
- *Elm in Action* by Richard Feldman – recently published.
- Lots of meetups all over the world – including Utrecht.
- Active Reddit community;
- Industrial users including Prezi, TruQu, No Red Ink, ...