Meet Elm

Marco Paolo Valerio Vezzoli[1]

<2017-10-01 Sun>

[1] mvezzoli@micron.com

let's begin with the most easy program

```
main = div [] [text "Hello World"]
```

when we save it the formatter changes a little the spaces: it is useful for

- reading consistently stuff
- verify trivial sintax errors (as it parse the code in order to format it)

The formatter nicely integrates all programmer' editors (no notepad)

Import and Export

It also added a new line with the module declaration: we change it a little

```
module Hello exposing (..)
```

now let's add some import that define the html functions

```
import Html exposing (div, text, h1)
```

Before compiling we need to install some packages: you can find a complete list at `http://package.elm-lang.org`

```
elm-package install elm-lang/html
```

this will install the html package and its dependencies
elm creates a file `elm-package.json` with all ours specifications and a directory `elm-stuff` where all packages will be downloaded To compile the package we can use

```
elm-make Hello.elm --output index.html
```

this will create an html file with the compiled program We can add some warning to the compiler

```
elm-make --warn Hello.elm --output index.html
```

this gives us a hint about the type of our value: we can write a type declaration like this

```
main : Html.Html Never
```

immediately before the code
Never is a type with no values: this means that our HTML is not dynamic We can also generate a json library which can help us integrating with other HTML

```
elm-make --warn Hello.elm --output main.js
```

we just need to add this in the body

```
<body>
  <H1>This is the external HTML<H1>
  <script type="text/javascript" src="main.js"></script>
  <div id="main"></div>
  <script type="text/javascript">
    var d = document.getElementById('main');
    Elm.Hello.embed(d);
  </script>
</body>
```

- Compiler infer types
  - this helps us a lot in debugging and development
  - a lot of effort has been put since the beginning to get useful error messages
  - the language support parametric polymorphism
- Semantic packaging and modules
  - modules are the mean to provide encapsulation
  - published module are enforced to change major version if the exposed signatures change
  - this means that module upgrades don't hurt

From version 0.17 of Elm, the previous language has been greatly simplified removing the whole "reactive" infrastructure for a less general but clearer pattern

We start this journey with two buttons which can increase or decrease an integer value

The Main Function

the main function accepts a record containing 4 functions: we will go through each one

```
main : Program Never Model Msg
main =
    Html.program
{ init = init
, subscriptions = subscriptions
, update = update
, view = view
}
```

The type means: this is a `Program` that has no startup input and an inner state of type `Model` which is modified by event of type `Msg`

The Model

this type represents the state of the application

```
type alias Model = Int
```

Alias are useful to read the code and the error messages
We start simple this time, but we will refactor later

The Init

we need to set up the initial state at the beginning of the application

```
init : ( Model, Cmd Msg )
init = ( 0, Cmd.none )
```

init returns a pair whose first element is the status and the second is an "effect" value or a command to execute something (e.g. send a message to a server).
We start with 0 and no effect.
Cmd is a PARAMETRIC type which accepts another type as parameter

Messages

this data type should represent the asynchronous signals coming from the application

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

this is called a UNION type; in this case it has exactly 2 values

Update

this function will change the state of the application according to signals

```
update : Msg -> Model -> ( Model, Cmd Msg )
update message model =
    case message of
Increase ->
    ( model + 1, Cmd.none )

Decrease ->
    ( model - 1, Cmd.none )
```

the type tells that this function expect the event as first input, the old status as second and gives out the modified status paired with a command (or "effect")

View

this function shows the page starting from the model

```
view : Model -> Html Msg
view model =
    div []
[ button [ onClick Increase ] [ text "Add 1" ]
, div [] [ text <| "Buy " ++ (toString model) ++ " bananas"
, button [ onClick Decrease ] [ text "Remove 1" ]
]
```

update Index.html

a little change is needed in the html call to find the module

```
<script type="text/javascript">
  var d = document.getElementById('main');
  Elm.Form.embed(d);
</script>
```

we will now add to the view a password field and its confirmation; we want to be able to signal to the users the quality of the password (weak, strong) and if both fields match

Adding more Widgets

First we add the password fields in the view and add an event to detect change

```
div  []
  [ div []
      [ label [ for "pass1" ] [ text "type your password" ]
      , input [ id "pass1"
  , onInput UpdatePass1
  , type_ "password"
  , value model.pass1
  ]
  []
      ]
  , -- some thing for pass2
  ]
```

Adding more Messages

the two messages are added to the union type to take into account each chenge in the textfields

```
type Msg
    = Increase
    | Decrease
    | UpdatePass1 String
    | UpdatePass2 String
```

the new messages now carry a value of type string

Extending the State

In order to store the new information we tranform the state into a record

```
type alias Model =
    { counter : Int
    , pass1 : String
    , pass2 : String
    }


init : ( Model, Cmd Msg )
init =
    ( { counter = 1
      , pass1 = ""
      , pass2 = ""
      }
    , Cmd.none
    )
```

Extending the Update

If we try to compile it now we get an error about the case switch not to be complete; we can add two cases; this demosntrates also type matching and decomposition

```
UpdatePass1 value ->
( { model
    | pass1 = value
  }
, Cmd.none
)

UpdatePass2 value ->
( { model
    | pass2 = value
  }
, Cmd.none
)
```

Adding Password Matching and Security: the Model

We want to feedback the user about the level of security; we can capture this in a type

```
type PassSecurity
    = Weak
    | Minimal
    | Good
```

and put into our model

```
type alias Model =
    { counter : Int
    , pass1 : String
    , pass2 : String
    , passMatching : Bool
    , passSecurity : PassSecurity
    }
```

Extending the code setup

in our init code we add initial values

```
, pass2 = ""
, passMatching = False
, passSecurity = Weak
}
```

Completing the view : security level

now we can show the value

```
, div []
     [ label [ for "pass1" ] [ text "type your password" ]
     , input
[ id "pass1"
, onInput UpdatePass1
, type_ "password"
, value model.pass1
]
[]
     , text <| "Security " ++ (toString model.passSecurity)
     ]
, hr [] []
```

Completing the view: password matching

we also show a colored status for pattern matching

```
, div []
    [ label [ for "pass2" ] [ text "retype your password" ]
    , input
[ id "pass2"
, onInput UpdatePass2
, type_ "password"
, value model.pass2
]
[]
    , let
( message, color ) =
    if model.passMatching then
( "Matching", "green" )
    else
( "Not Matching", "red" )
      in
div [ style [ ( "color", color ) ] ] [ text message ]
    ]
```

Completing the update: security logic

as an example we may decide that security is given by length alone

```
UpdatePass1 value ->
    let
passLength =
    String.length value

security =
    if passLength < 4 then
Weak
    else
(if passLength < 6 then
    Minimal
 else
    Good
)
    in
( { model | pass1 = value, passSecurity = security }, Cmd.n
```

Completing the update: password matching

We also can add a comparison between passwords

```
UpdatePass2 value ->
    let
match =
    value == model.pass1
    in
( { model
    | pass2 = value
    , passMatching = match
  }
, Cmd.none
)
```

this check must be added on the other case in order to make everything correct

Elm code does not have a lot of problems that you can meet in common javascript; but this is not enough: we want the program to work as it was intended. This may be due to an incorrect logic.
You can check unintended behaviours using the debugger; it is also fun to see how the program actually works.

Compiling with debugger option

You can compile the form with the `--debug` option

```
elm-make Form.elm --warn --debug --output main.js
```

this is useful in development: production code should not be compiled in this way
In the page now a new control appears which counts every event recorded

Time Travelling

through the debugging interface it is possible to:

- move to any recorded event and see the internal state while the GUI updates
- load and save all events list: this is great to report problems and reproduce each step

- The Type System support refactoring

- The Purity support debug

In this example we are creating a simple clock in elm

Subscriptions and messages

Elm handles those asynchronous events which are not related to the UI with subscriptions
A typical example would be server answers to http request; getting the time is another.

```
type Msg = Tick Time

subscriptions : Model -> Sub Msg
subscriptions model =
    Time.every (1 * Time.second) Tick
```

Model and Init

Let's start modelling the status to be the seconds of the current minute

```
type alias Model =
    Int

init : ( Model, Cmd Msg )
init =
    ( 1, Cmd.none )
```

Update

every 1 s a Tick event is sent; its value is the number of ms from epoch; we transform it into the seconds of current minute

```
update : Msg -> Model -> ( Model, Cmd Msg )
update (Tick time) model =
    ( (floor (time / 1000)) % 60, Cmd.none )
```

SVN: Declaration

we use SVN to draw the clock

```
import Svg exposing (svg, circle, line)
import Svg.Attributes exposing ( --many things here
```

SVN: view part 1

first a little trigonometry

```
view : Model -> Html Msg
view model =
    let
size =
    { width = 300, height = 300 }

center =
    { x = size.width / 2, y = size.height / 2 }

radius =
    size.width / 2

angle =
    2 * pi * (toFloat model) / 60

hand =
    { x = center.x + radius * cos angle
    , y = center.y + radius * sin angle
```

SVG: view part 2

then the actual draw

```
in
    div []
[ svg
    [ width <| toString size.width
    , height <| toString size.height
    , viewBox <| "0 0 " ++ (toString size.width) ++ " " ++
    ]
    [ circle
[ cx <| toString center.x
, cy <| toString center.y
, r <| toString radius
, fill "blue"
]
[]
    , line
[ x1 <| toString center.x
, y1 <| toString center.y
, x2 <| toString hand.x
```

- Asynchronous events are all created equal
  - Dynamic DOM Update
  - Effects are accessible: type system aids checking
- Asynchronous events are all created equal (Again)
- Games anyone?
- what if we want to integrate more modules into one?
- is it possible to create reusable "widgets" (e.g. a calendar)?
- How can types be matched in order to reuse the status and message types?

we can import other modules from the main module like this

```
import Clock
import Form
import Hand
```

but what do they have to export?

- the Model and Msg types
- the init function
- the update function
- the subscription function

- the view function

to have a working model we must allow space for each of the sub models

```
type alias Model =
    { hand : Hand.Model
    , clock : Clock.Model
    , form : Form.Model
    }
```

the same is true for messages

```
type Msg
    = HandMsg Hand.Msg
    | ClockMsg Clock.Msg
    | FormMsg Form.Msg
```

the init also allows for each module to have side effects when initialized; it is useful to execute all of them in sequence; `Cmd.map` allows to remap messages into the common type

```
init =
    let
        ( handM, handC ) =
            Hand.init
```

```
        -- and so on for other modules
    in
        ( { hand = handM
          , clock = clockM
          , form = formM
          }
        , Cmd.batch
            [ Cmd.map HandMsg handC
            , Cmd.map ClockMsg clockC
            , Cmd.map FormMsg formC
            ]
        )
```

also subscriptions can be glued together; Sub.map is the function to remap the messages here

```
subscriptions : Model -> Sub Msg
subscriptions model =
    let
        clockS =
            Sub.map ClockMsg <| Clock.subscriptions model.c


        formS =
```

```
            Sub.map FormMsg <| Form.subscriptions model.for

        handS =
            Sub.map HandMsg <| Hand.subscriptions model.han
    in
        Sub.batch [ clockS, formS, handS ]

update : Msg -> Model -> ( Model, Cmd Msg )
update message model =
    case message of
        ClockMsg msg ->
            let
                ( clockNewM, clockNewC ) =
                    Clock.update msg model.clock
            in
                ( { model | clock = clockNewM }, Cmd.map Cl
-- more cases follow
```

`Html.map` provides the type mapping functionality also for the view part

```
view : Model -> Html Msg
view model =
    div []
```

```
    [ Html.map ClockMsg <| Clock.view model.clock
    , Html.map HandMsg <| Hand.view model.hand
    , Html.map FormMsg <| Form.view model.form
    ]
```

- The Elm architecture is scalable
- There is more behind Map than your eyes see now