Introduzione a Elm

Marco Paolo Valerio Vezzoli[1]

<2017-10-01 Sun>

[1] mvezzoli@micron.com

Elm Architecture: Una semplice Form

Dalla versione 0.17 of Elm, il linguaggio precedente è stato notevolmente semplificato rimuovendo l'infrastruttura "reattiva" per un pattern meno generale ma piú chiaro
Iniziamo questo viaggio con due bottoni che possono incrementare o deceremente un valore intero

Elm Architecture: La funzione Main

la funzione main accetta un record che contiene quattro funzioni: andremo a conoscerle una per una

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

La segnatura di tipo significa: questo è un `Program` che non ha un imput a startup, ha uno stato interno di tipo `Model` che viene modificato da eventi di tipo `Msg`

Elm Architecture: il Model

questo tipo rappresenta lo stato interno dell'applicazionr

```
type alias Model = Int
```

Gli alias sono utili per leggere i codice ed i messaggi di errore
Cominciamo con un modello semplice che modificheremo in seguito

Elm Architecture: la Init

dobbiamo inizializzare lo stato del sistema quando viene avviato

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

La funzione `init` restituisce una coppia il cui primo elemento è lo stato dell'applicazione e il secondo rappresenta una "azione" da compiere (per esempio mandare un messaggio al server)
Iniziamo con stato 0 e nessun effetto.
Il tipo `Cmd` è parametrizzato, ed accetta un altro tipo come parametro

Elm Architecture: i messaggi

questo tipo di dato rappresenta i segnali asincroni che vengono mandati all'applicazione

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

In questo esempio utilizziamo uno union type che in questo caso ha soltanto due costruttori vuoti; in pratica il tipo puó avere solo due valori.

Elm Architecture: la Update

questa funzione restituisce lo stato aggiornato dopo l'arrivo di un segnale

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

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

Il tipo di questa funzione dice che ci aspettiamo un evento come primo valore passato alla funzione e lo stato corrente come secondo; il valore restitutito è una coppia che contiene lo stato modificato e un "effetto" da eseguire.

Elm Architecture: la View

questa funzione mostra la pagina a partire dal modello

```
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" ]
]
```

Elm Architecture: update Index.html

è necessario un piccolo cambiamento nell'html per proseguire

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

Estendere la Form

ora aggiungeremo alla vista un campo password e la sua conferma; vogliamo poter segnalare all'utente la qualità della password (forte o debole) e se entrambe i campi corrispondono

Estendere la Form: Aggiungere più Widgets

anzitutto aggiungiamo un campo password nella vista e aggiungiamo un evento per rilevare il cambiamento

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

Estendere la Form: Aggiungere più Messaggi

I due messaggi vengono aggiunti al tipo unione per considerare i cambiamenti in entrambe i campi password

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

the new messages now carry a value of type string

Estendere la Form: Estendere lo Stato

Per contenere maggiori informazioni estendiamo lo stato trasformandolo in un record

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


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

Estendere la Form: Estendere l'aggiornamento

Se provassimo a compilare ora otterremmo un errore dal comando `case` poichè il pattern matching non è completo; questi nuovi casi dimostrano anche come gestire il pattern matching e la decomposizione dei valori

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

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

Estendere la Form: Estendere il modello

Vorremmo restituire all'utente una informazione relativamente al livello di sicurezza: possiamo catturare questo concetto in un tipo

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

e metterlo nel nostro modello

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

Estendere la Form: 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

Debugger

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.

Debugger: 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

Debugger: 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

Takeaways

- The Type System support refactoring
- The Purity support debug