INSTITUTO TECNOLÓGICO DE BUENOS AIRES

Memotest en Elm

Programación funcional

Juan Martín Buireo

18/07/2016

Informe sobre el desarrollo de un Memotest utilizando el lenguaje de programación funcional Elm para el final de la materia “Programación funcional”

Índice

Introducción

Este informe trata acerca de las decisiones de diseño e implementación adoptadas para la realización del tradicional juego Memotest utilizando el lenguaje de programación funcional Elm. Este juego fue realizado para el final de la materia "Programación funcional". Este tema fue elegido entre otros propuestos por la cátedra debido a que la sintaxis era muy similar a la vista en clase (Haskell) aunque se presentan algunas diferencias mínimas. Por otro lado, luego de investigar un poco, se llegó a la conclusión de que dicho lenguaje era apropiado para realizar este juego.

El lenguaje Elm

Elm es un lenguaje de programación funcional para crear páginas web estáticas y dinámicas. Utiliza el estilo de programación funcional reactivo y el diseño gráfico es puramente funcional permitiendo construir la interfaz de usuario sin actualizaciones destructivas.

Fue diseñado por Evan Czaplicki como parte de su tesis en 2012. Elm puede compilarse en archivos HTML, CSS y JavaScript. A su vez cuenta con un compilador online, un debugger y una comunidad con distintas librerías creadas por los usuarios.

Este trabajo práctico se realizó utilizando la versión más reciente de Elm (0.17). Dicha versión tiene algunos cambios de nomenclaturas en relación a las versiones anteriores.

Elm utiliza el patrón de model, view y update. A continuación se explica cada una de las funciones:

- Model: define el estado de la aplicación, es decir, la estructura de datos que almacenará lo necesario para el funcionamiento de la aplicación.
- Update: permite actualizar el estado de la aplicación acorde a ciertas acciones implementadas por el usuario.
- View: permite visualizar el estado de la aplicación en formato HTML.

Estas tres partes se encuentran claramente separadas y son necesarias para el funcionamiento de la aplicación.

Existen también otros dos conceptos importantes que pueden ser utilizados opcionalmente según lo requiera la aplicación. Ellos son:

- Commands: es una forma de exigir algún efecto, es decir, solicitar algún valor y que la respuesta pueda ser diferente dependiendo lo que esté pasando en el mundo. Por ejemplo, realizar una petición HTTP.
- Subscriptions: permite registrarse a la espera de algo, es decir, estar a la espera y recibir las actualizaciones sólo cuando existan. Por ejemplo, escuchar los mensajes que entran por un web socket.

Reglas del Memotest

El Memotest es un juego individual que tiene como objetivo encontrar pares de cartas iguales en el menor tiempo posible.

El juego comienza con todas las cartas “tapadas”, es decir, con las cartas boca abajo:

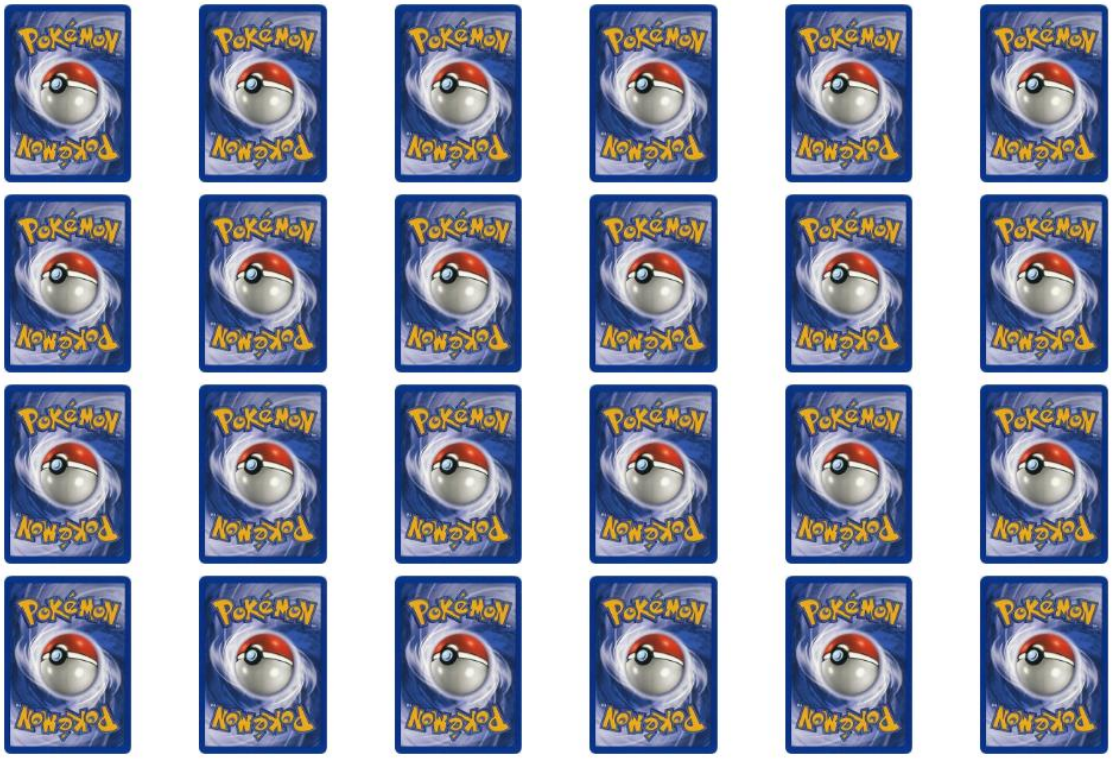

Imagen 1: *Cartas dispuestas al comienzo del juego*

Al cliquear sobre una carta, la misma se volteará:

Imagen 2: *Carta descubierta luego de ser clickeada*

En todo momento puede haber 0, 1 o 2 cartas descubiertas. En caso de encontrar un par de cartas iguales, las mismas quedan marcadas de manera que quedan descubiertas hasta el final del juego sin la posibilidad de taparlas de vuelta. Si al descubrir dos cartas no hay coincidencia entre ellas y se intenta descubrir una tercera, las dos cartas anteriores se taparán automáticamente.

Imagen 3: *Dos pares de cartas encontradas y dos cartas descubiertas diferentes. Al cliquear una tercera, estas dos se cubrirán*

El juego finaliza al encontrar todos los pares de cartas iguales.

Implementación del juego

Implementación de una carta

Model

Para implementar el juego se comenzó por desarrollar algo básico; se desarrolló una única carta la cual al ser clickeada se da vuelta.

Lo primero que había que definir era el estado de la carta. Había que distinguir dos estados diferentes en la carta, si estaba cubierta o descubierta. Para ello se creó un nuevo tipo con dos estados posibles:

```
type Face = Back | Front
```

Por otro lado, cada carta va a tener asociada una imagen. Por eso se agregó un String para representar el nombre de dicho archivo de imagen. Finalmente el modelo quedó definido de la siguiente manera:

```
type alias Model = { face: Face, image: String }
```

Con estas dos variables alcanzaba para saber en todo momento si la carta se encontraba tapada (Back) o destapada (Front) y cuál era la imagen que correspondía a esa carta (en caso de estar destapada ya que tapadas todas eran iguales).

Update

En cuanto a las acciones (llamadas Msg en la nueva versión de Elm), la única posible era la de girar la carta. Por lo tanto se definió:

```
type Msg = Flip
```

Luego se definió la actualización (la función update en Elm). La misma se encarga de actualizar el modelo cambiando la cara de la carta:

```
update msg model = case msg of
                       Flip -> flipCard model


flipCard model = case model.face of
                     Front -> { model | face = Back }
                     Back -> { model | face = Front }
```

View

Lo único que restaba definir era la vista (la función view en Elm). La misma presentaba una dificultad y era decidir si mostrar la carta destapada (mostrar el

archivo de la imagen) o mostrar la carta tapada (mostrar una imagen por defecto). Para ello se creó una función que según la cara en que se encuentre la carta muestre una u otra. Se utilizó el evento onClick de la librería de Elm para llamar a la función de update.

```
... img [onClick Flip, src (getImage model) ...]
```

```
getImage model = case model.face of
                    Back -> "images/cardBack.jpg"
                  _ -> model.image
```

Implementación del Memotest

Para implementar el Memotest teniendo ya el comportamiento de una única carta implementada se definió esta misma como módulo. Para ello se agregó en la implementación de la carta lo siguiente:

```
module PokeCard
```

De esta forma se puede reutilizar el comportamiento de una carta para extenderlo a varias cartas agregando lo siguiente en el Memotest:

```
import PokeCard as Card
```

Model

El primer paso fue definir un nuevo modelo para el Memotest. Para ello siguiendo el ejemplo de la guía de Elm (http://guide.elm-lang.org/architecture/modularity/counter_list.html) se definió un nuevo alias:

```
type alias IndexedCard = { id : Int, model : Card.Model }
```

Esto permite identificar cada carta con un id de manera que sea posible luego determinar sobre que carta se ejecutó una acción. Se puede notar la reutilización del modelo de la carta previamente definida aprovechando su comportamiento.

Se definió otro tipo nuevo que sirve para determinar el estado en el que se encuentra el juego:

```
type GameState = NewGame | InProgress | Finished
```

Con este estado se puede determinar que pantalla mostrar según el estado en que se encuentre el juego (se apreciará mejor más adelante cuando se explique la función view).

Con estos dos tipos nuevos definidos el modelo quedó de la siguiente manera:

```
type alias Model = { cards : List IndexedCard, startTime : Time, currentTime :
Time, gameState : GameState }
```

Para definir las cartas se utilizó una lista. Se tomó esta decisión para aprovechar la potencialidad de la misma con las funciones map, filter, etc.

Las variables startTime y currentTime se usan para poder medir el tiempo transcurrido. En la variable startTime se almacena el tiempo en que comenzó el juego y currentTime actualiza mediante subscripciones (se explicarán más adelante) el tiempo actual. Luego mediante la resta entre currentTime y startTime se obtiene el tiempo total de juego.

Por último, se definió el tipo gameState para poder saber en qué estado se encuentra el juego.

Update

Se definieron las siguientes acciones:

```
type Msg = Start | ShuffleCards (Array.Array IndexedCard) | FlipCard Int Card.Msg
| MainMenu | Tick Time | ChangeGameState GameState
```

A continuación se explica cada una de ellas.

START

En la primer pantalla que se muestra (el menú principal), hay una sola acción posible que es oprimir un botón para iniciar el juego. Al oprimirlo, se llama a la acción Start:

```
Start -> (Model [] 0 0 InProgress, shuffleCards indexedCardList)
```

Se puede ver que se crea un modelo vacío con el estado de juego en progreso (InProgress). En la función de inicialización (función init):

```
init = (Model [] 0 0 NewGame, Cmd.none)
```

, se creó el modelo con el estado NewGame. En lugar de cambiar solamente la variable GameState en el modelo de NewGame a InProgress se crea nuevamente el modelo ya que este botón es luego reusado para resetear el juego en caso de querer comenzar de nuevo mientras se está jugando.

Se puede ver también que se llama a una función shuffleCards y se le pasa como parámetro a indexedCardList. Esta función tiene el siguiente tipo:

```
shuffleCards : List IndexedCard -> Cmd Msg
```

Y generará una acción de tipo ShuffleCards que se explica más abajo. El llamado se realiza en forma de comando ya que shuffleCards devuelve un comando.

IndexedCardList es la lista de cartas inicial sin ser mezclada. Esta lista de cartas fue creada de una manera un poco compleja pero que facilita el agregado de nuevas cartas sin tener que definir el id de la carta nueva y acordarse de agregarla dos veces (ya que debe existir por cada carta su par correspondiente). El primer paso fue llamar al constructor (la función init) de cada carta:

```
cardsList : List Card.Model
cardsList = List.map (\image -> Card.init image) ["images/blastoise.jpg",
"images/charmeleon.jpg", "images/raichu.jpg", "images/diglett.jpg",
"images/venusaur.jpg", "images/mewto.jpg", "images/ninetales.jpg",
"images/scyther.jpg", "images/wigglytuff.jpg", "images/pikachu.jpg",
"images/omanyte.jpg", "images/jynx.jpg"]
```

De esta manera en caso de querer agregar más cartas al juego basta con agregarla sólo una vez a esta lista. Luego, aprovechando la potencialidad de las listas y la función map2 que provee el core de Elm se crearon las siguientes dos funciones:

```
indexedCardList : List IndexedCard
indexedCardList = List.map2 (<|) indexList (cardsList ++ cardsList)


indexList : List (Card.Model -> IndexedCard)
indexList = List.map (\index -> IndexedCard index) [0..((List.length cardsList) *
2)]
```

Al concatenar dos veces la misma lista y en la función indexList utilizar la longitud de la misma, se puede agregar una sola vez la nueva carta sin pensar en tener su par correspondiente. Por otro lado, se puede ignorar el agregado del índice a la lista que se produce de manera automática.

SHUFFLECARDS

Sirve para mezclar las cartas al comienzo del juego. Se utilizó la librería “elm-community/random-extra” que tiene una función (shuffle) para mezclar un Array pero no para listas. De todas maneras, existe una función en el core de Elm para convertir de lista a array.

La función shuffle tiene la siguiente forma:

```
shuffle : Array a -> Generator (Array a)
```

Y la funcion Random.generate del core de Elm:

```
generate : (a -> msg) -> Generator a -> Cmd msg
```

Con lo cual, hubo que crear un mensaje ShuffleCards array tal como se ve en el ejemplo de la guía (http://guide.elm-lang.org/architecture/effects/random.html).

Luego en la acción ShuffleCards se actualiza la lista de cartas en el modelo de la siguiente manera convirtiendo nuevamente el Array a lista:

```
ShuffleCards array -> ({ model | cards = Array.toList array }, Cmd.none)
```

La parte de Cmd.none corresponde a los comandos previamente explicados. En este caso luego de mezclar las cartas no se lleva a cabo ningún comando razón por la cual se usa Cmd.none.

TICK

Se utiliza esta acción para poder actualizar las variables de tiempo (startTime y currentTime). Para ello se utilizó uno de los ejemplos de la guía de Elm (http://guide.elm-lang.org/architecture/effects/time.html).

En esta sección interviene el uso de las subscripciones para generar una acción de tipo Tick cada segundo:

```
subscriptions : Model -> Sub Msg
subscriptions model = case model.gameState of
                        InProgress -> Time.every Time.second Tick
                        _ -> Sub.none
```

Se puede ver que sólo se actualizará el tiempo mientras el juego se encuentre en progreso. En el resto de los casos (menú principal o juego finalizado), el tiempo no se actualiza.

Con respecto a la acción, sólo se actualiza la variable startTime al comienzo para registrar el tiempo en que comenzó el juego y luego siempre se actualiza currentTime:

```
Tick newTime -> if (startTime == 0) then ({ model | startTime = newTime,
currentTime = newTime }, Cmd.none) else ({ model | currentTime = newTime },
                                          Cmd.none)
```

FLIPCARD

En esta acción se lleva a cabo la parte más importante del juego, la de girar una carta y controlar si se encontró un par igual o taparlas nuevamente al haber dos cartas destapadas simultáneamente.

Para poder implementar esta acción hubo que realizar primero un pequeño cambio sobre la implementación de la carta. Antes se habían definido sólo dos estados posibles para la cara de una carta: Back y Front. Restaba definir un nuevo estado para cuando la carta se había descubierto junto con su mismo par. Para ello se modificó el tipo de la carta agregando un nuevo estado de la siguiente forma:

```
type Face = Back | Front | Found
```

Luego se revisaron las funciones ya implementadas en la carta agregando el nuevo caso (Found para una carta ya encontrada con su par).

Una vez hecho esto, se procedió a definir el estado en cuestión:

```
FlipCard id message -> ({ model | cards = checkCards (updateCards id message
    cards) }, Cmd.Extra.message (ChangeGameState gameState))
```

Se puede notar que se recibe en la acción el id de la carta cliqueada y la acción realizada sobre la carta. Con esto lo primero que hay que hacer es actualizar las cartas del tablero girando la carta que corresponda:

```
updateCards : Int -> Card.Msg -> List IndexedCard -> List IndexedCard
updateCards id message cards = List.map (updateCard id message) cards


updateCard : Int -> Card.Msg -> IndexedCard -> IndexedCard
updateCard targetId msg {id, model} = IndexedCard id (if targetId == id
then Card.update msg model else model)
```

La función updateCards se encarga de destapar o tapar la carta sobre la cual el usuario cliqueó independientemente de cuantas cartas hayan destapadas. Acá se puede notar la reutilización de la carta previamente definida (Card.update).

Una vez actualizado el estado de las cartas, se ejecuta la función checkCards que tiene como objetivo controlar la repercusión que pudo haber tenido en el progreso del juego el haber modificado una carta. Esta función básicamente chequea si al haber rotado la carta se encontró un par de cartas iguales para pasarlos al estado Found y dejarlas destapadas hasta el final del juego independientemente que el usuario cliquee sobre ellas. También chequea si al haber rotado la carta y no haber coincidencia quedan más de 3 cartas descubiertas tapando las anteriores ya que va contra las reglas del juego.

Se puede observar que se ejecuta un comando que llama a una nueva acción:

```
Cmd.Extra.message (ChangeGameState gameState)
```

Para poder usar esto se utilizó la librería “shmookey/cmd-extra” que provee una función para crear una acción. Esta función es muy útil cuando se quiere disparar un evento desde el código de Elm y no desde un evento externo. Se implementó de esta manera ya que recién se puede chequear el estado del juego (gameState) una vez finalizada la función checkCards y actualizada la lista de cartas.

CHANGEGAMESTATE

Esta acción se encarga de determinar si el juego sigue en progreso o finalizó:

```
ChangeGameState state -> ({ model | gameState = checkGameState cards
                          }, Cmd.none)
```

La función checkGameState se fija si la cantidad de cartas encontradas (en estado Found) es igual a la cantidad de cartas total del juego. De ser así, se cambia el estado a terminado (Finished).

MAINMENU

Esta acción sirve para comenzar un juego nuevo cuando se encuentra en la pantalla final del juego finalizado:

```
MainMenu -> init
```

Lo único que hace es llamar a la función init previamente mencionada:

```
init = (Model [] 0 0 NewGame, Cmd.none)
```

View

Para implementar la vista del juego, se aprovechó el uso de la variable gameState para determinar qué pantalla se debe dibujar en cada momento. También se aprovechó lo previamente definido en la carta y se llamó a la función Card.view para dibujar a la misma.

En este punto se aprovechó el uso de un framework llamado Bootstrap. Dicho framework es un CSS ya establecido que ya trae incorporado todo el manejo de valores porcentuales y los distintos tamaños de pantallas (laptop, Tablet, celular, etc.). Es muy utilizado hoy en día en el desarrollo de páginas o aplicaciones web y es open-source. Twitter aparte de otras páginas conocidas lo utiliza.

Al haber utilizado Bootstrap para definir el estilo CSS del juego, el mismo es responsive lo que quiere decir que los distintos objetos se redistribuyen según el

tamaño de pantalla que se utilice. Esto se logra utilizando las clases “col-lg-x”, “col-sm-x”, etc. en los divs, donde lg, sm, etc. determinan la pantalla sobre la cual se trabaja. Se considera que una pantalla a lo ancho está divida en 12 secciones con lo cual col-lg-2 dice que ese div ocupa 2 secciones de las 12.

En las siguientes imágenes se puede ver la reorganización de los objetos según sea un celular o una laptop:

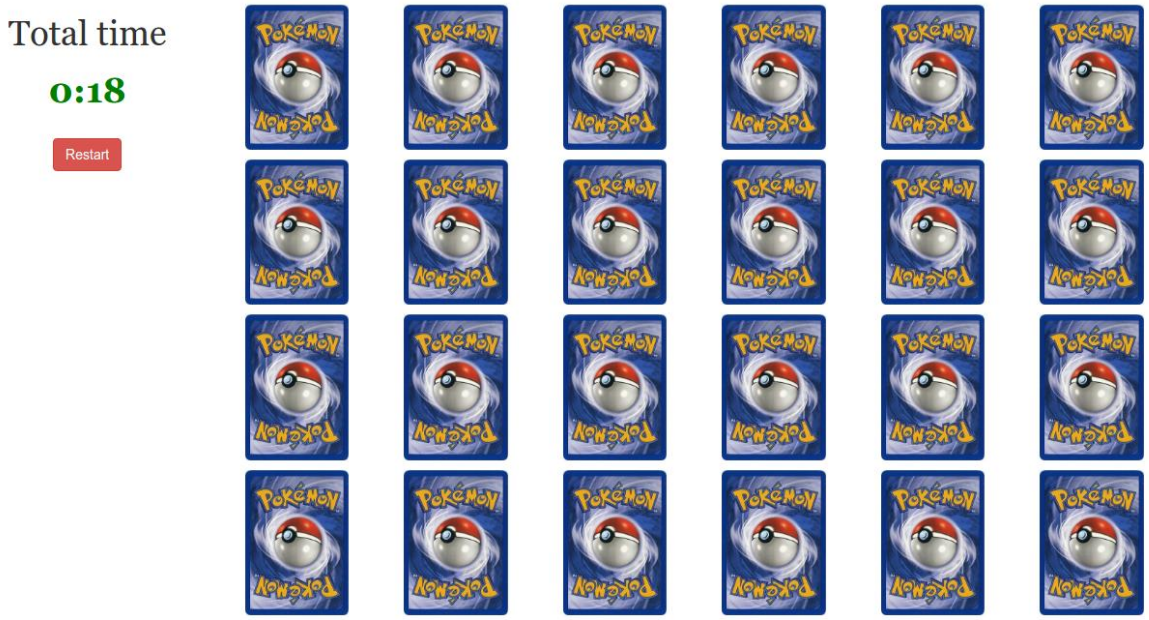

Imagen 4: *Vista del juego en una laptop*

Imagen 5: *Vista del juego en un celular*

Conclusiones

Aprender la sintaxis de Elm fue muy sencillo por ser muy similar a la sintaxis vista en clase (Haskell). La parte nueva es el modelo propuesto por Elm (model, view, update), el cual se comprende muy bien siguiendo la guía completa con ejemplos (http://guide.elm-lang.org/). Con lo cual aprender Elm teniendo incorporados los conceptos de la programación funcional es muy fácil.

Encontré el uso de Elm muy cómodo para el desarrollo de juegos orientados a máquinas de estado. Al ser un lenguaje funcional, termina siendo un lenguaje muy descriptivo en sí mismo donde se definen estados y se realizan acciones sobre esos estados. Realizar esto en un lenguaje imperativo con una vista hubiese sido mucho más engorroso en cuanto a código.

Una de las desventajas que encuentro es que en cada “update” la aplicación se redibuja completamente. Esto en juegos complejos de tiempo real puede llegar a causar problemas de performance.

Otra desventaja es la manera en que se define el HTML. Cuando se comienzan a concatenar divs dentro de divs, se confunde un poco la construcción del HTML al haber tantos corchetes.

No tuve oportunidad de probar, ya que no lo requerí, la interacción con JavaScript.