Trabajo Práctico Final

Blob Wars

Programación Funcional

Alumno
Argentino Ducret 52194

Profesores
Martinez Lopez, Pablo Ernesto Fidel
Pennella, Valeria Verónica

Introducción

El objetivo del trabajo práctico es implementar el juego Blob Wars en el lenguaje de programación Elm, el cual se encuentra dentro del paradigma funcional.

Este juego es para dos jugadores, consta de un tablero 8x8 que inicialmente se parte con dos fichas o manchas para cada jugador ubicadas en las esquinas del tablero, el juego finaliza al completar el tablero o cuando un jugador pierde todas sus fichas.

El lenguaje Elm es un lenguaje con una curva de aprendizaje rápida para la mayoría de los conceptos básicos y esenciales, además cuenta con una arquitectura muy particular que vale la pena conocer y aplicar a otros proyectos que no necesariamente estén desarrollados en lenguajes funcionales puros.

Se comienza explicando las reglas de juegos, luego se desarrolla la estructura o arquitectura que el lenguaje Elm necesita para una correcta implementación, se detallaran los modelos adoptados para representar el juego, se analizaran algunas tecnicas de debug y por ultimo instrucciones de compilación.

Reglas de juego

El objetivo del juego es dominar el tablero y obtener la mayor cantidad de manchas. El juego finaliza cuando el tablero esté completo o un jugador se quede sin fichas y el que más manchas tenga será el ganador.

Hay dos colores azul y rojo que representan a los dos jugadores, inicialmente se colocan de una lado del tablero las dos fichas azules en las esquinas y del mismo modo se colocan las fichas rojas del otro lado del tablero. Siempre comienza a jugar el jugar de color azul.

Los posibles movimientos son seleccionar una mancha del color del jugador y moverla 1 o 2 casilleros en cualquier dirección (la manchas propias o del rival no bloquean el movimiento). Si el movimiento es de 1 casillero la mancha original seleccionada se queda en su lugar y genera una ficha nueva en el casillero seleccionado. En caso contrario, si el movimiento es de 2 casillas la mancha original se mueve al lugar destino.

La forma de conquistar o manchar la fichas enemigas es realizar un movimiento y automáticamente se manchas las fichas rivales que estén a radio 1 de la ficha que se movió.

Arquitectura del lenguaje Elm

En esta sección se detallara la arquitectura básica que propone este lenguaje funcional.

Para comenzar es importante notar que Elm es un lenguaje estáticamente tipado, esto quiere decir que en tiempo de compilación se detectan o infieren todos los tipos que se van a utilizar ya sean en variables, funciones, etc. Es decir que no es posible que tengamos algún tipo de error de casteo o crash al aplicar alguna función como suele suceder en lenguaje imperativos.

Además Elm es un lenguaje funcional puro, en otras palabras no se pueden realizar cambios de estados o mutar datos. Será necesario perder la costumbre de los lenguajes imperativos tanto de declarar variables, tener objetos con estados, entre otras prácticas.

Arquitectura basica

La lógica de los programas desarrollados en Elm se pueden distribuir en tres secciones principales y bien diferenciadas:

1. Model: En esta sección se declaran todos los tipos que se usarán y se inicializa el modelo que utiliza la arquitectura de Elm. Es muy importante tener bien definido el modelo para poder trabajar con este en las siguientes secciones. En pocas palabras, representa el estado de la aplicación.
2. Update: Es la función encargada de actualizar el modelo al recibir algún mensaje. Los mensajes son otro actor importante en esta arquitectura, normalmente los mensajes son enviados por las acciones que realiza el usuario en la UI.
3. View: Es una funciona que dado un modelo lo traduce a HTML o a una vista con la cual el usuario interactuara.

Existe también una arquitectura un poco más compleja que se podría usar en Elm, la cual agrega señales, subscripciones, comandos, puertos para comunicarse con codigo JS, entre otras cosas que no serán abarcadas en este trabajo.

Ejemplos de la arquitectura básica

Todo programa debe indicar las 3 partes básica antes mencionadas, de esta forma Elm internamente llamara a las funciones de Update y View con el modelo correspondiente.

```
main = Html.beginnerProgram
  {
    model   = model,
    view    = view,
    update  = update
  }
```

El modelo es un estructura definida por el programador la cual deberá tener todo lo necesario para poder representar el estado de la aplicación en todo momento, se lo puede pensar como una foto del programa en un cierto momento.

La función de update puede ser declarada como más convenga pero generalmente recibe como parámetros un mensaje y el modelo. Los mensajes deben ser definidos según las acciones que el usuario pueda realizar. Dados estos parámetros la función deberá evaluarlos y retornar el modelo actualizado.

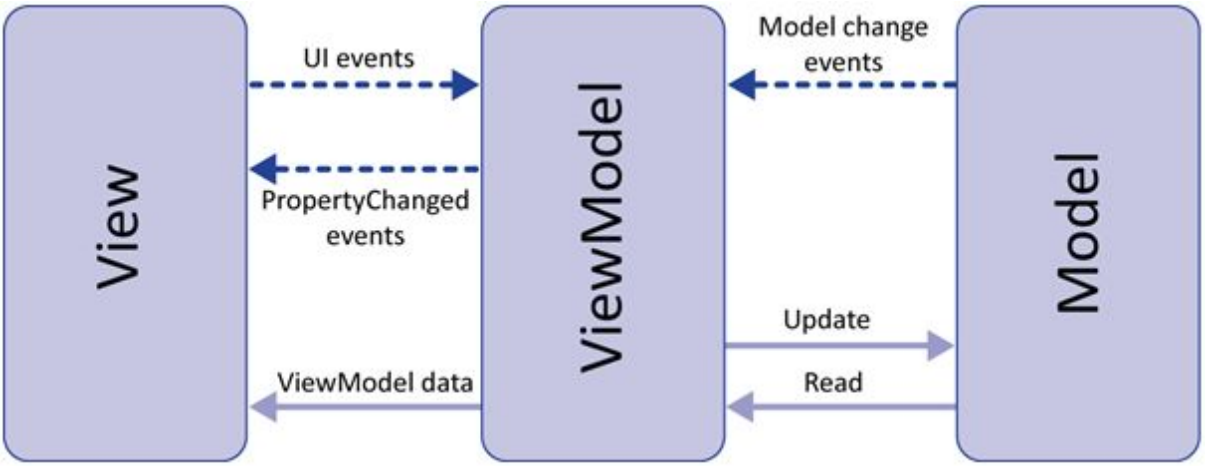
```
type Msg = Message1 | … | MessageN

update : Msg -> Model -> Model
update msg model =
  ...
```

Por último la función view, recibe como parámetro el modelo y lo traduce a una vista, particularmente en Elm lo transcribe a Html.

```
view : Model -> Html Msg
view model =
  ...
```

Experiencia personal

Hace dos años que desarrollo aplicaciones en Swift (iOS) aplicando el patrón MVVM y es muy común que en aplicaciones grandes donde hay varios niveles de profundidad al navegar se generen inconsistencias con el patrón, dificultad para coordinar eventos o acciones, se generan side effects no deseados o condiciones de carrera y se dificulta la comunicación entre view controllers padres e hijos.



En cambio si se siguiera un manejo de eventos o mensajes orientado al propuesto en Elm, se podrían evitar la mayoría de estos problemas.

Adoptando una estructura de mensajes de tipo árbol, es decir que los mensajes solo fluyen desde los padres hacia los hijos. Además, utilizando estado y datos inmutables se puede representar cada momento de la aplicación dado un mensaje. Tambien se podria guardar el estado en un stack para posteriormente analizarlo a modo de debug o agilizar tareas de QA.

Una de las librerías que comencé a utilizar para evitar los problemas antes mencionados es Portal la cual está inspirada en la arquitectura de Elm pero con sutiles cambios para poder adaptarla a aplicaciones para iOS. Con esta nueva forma de programar en Swift, se pueden observar todos los cambios de estados de la aplicación en un switch principal donde según qué mensaje se reciba se aplicarán ciertos cambios al estado o modelo y luego en la función de view se puede ver claramente que vista se va a visualizar para ese frame.

Buenas prácticas

1. **Siempre comenzar declarando los tipos a utilizar:** Por ejemplo al definir una función, primero pensar cuáles son los parámetros a utilizar y que retornara. Al hacer esto, ayuda a familiarizarse con el mundo funcional y a pensar antes de escribir una función.

2. **Usar typealias:** Abusar del uso de typealias puede ser algo bueno.
   Ejemplo:

```
type alias Row = Int
type alias Col = Int
type alias Size = Int
type alias Distance = Int
type alias Point = (Row, Col)
```

   Es mucho más intuitivo ver una definición de la siguiente forma:
   Size -> (Point -> Distance) -> Point
   Que en cambio usar:
   Int -> ((Int, Int) -> Int) -> Int

3. **Que los estados inválidos no sean posible:** Si bien a veces no es fácil o no es posible, no hay que generar casos extras que no van a representar un estado valido de la aplicación, siempre es mejor pensar mejor el modelo y como se va a representar la aplicación.

Modelo

La parte más importante es el modelo, siempre es conveniente dedicarle un tiempo a analizar y pensar esta sección antes de empezar a programar. Un error en el modelo y puede generar estados inconsistentes, o que el código sea mucho más engorroso simplemente por plantear mal el modelo.

En el caso de Blob wars, se eligió representar el modelo de la siguiente forma:

```
type alias Model =
  {
    board              : Board,
    selected           : Maybe Point,
    match              : Match,
    turn               : Turn,
    amountOfRed    : Amount,
    amountOfBlue   : Amount
  }

type alias Row = Int

type alias Col = Int

type alias Point = (Row, Col)

type alias Board = List(List(Tile))

type alias Amount = Int

type Tile = Empty
          | PossibleMovement
          | Selected Blob
          | Unselected Blob

type Blob = Blue
          | Red


type Match = BlueWin
           | RedWin
           | Tie
           | InProgress

type Turn = BlueTurn
          | RedTurn
```

Como es un juego de tablero lo ideal es representarlo con una lista de listas de los posibles elementos a representar en el tablero, en este caso Tiles.
Tiles se definió utilizando union types, es decir que pueden ser Empty, PossibleMovement, Selected Blob o Unselected Blob. Donde Blob puede ser Red o Blue, los colores de los jugadores.

Además se agregaron al modelo otros elementos para poder representar la vista, como la cantidad de fichas de cada jugador, el turno (rojo o azul), el estado del juego (en progreso, empate, ganó azul o gano rojo).

Por último, es necesario saber cual es la posición de la última ficha seleccionada para realizar el movimiento en el caso correspondiente, ya que el único dato que tenemos es la posición donde se hizo click.

Maquina de estados

Es útil pensar a la aplicación como una máquina de estados la cual se analizará en la función de update. Por ejemplo en nuestro caso solo soportamos dos posibles mensajes:

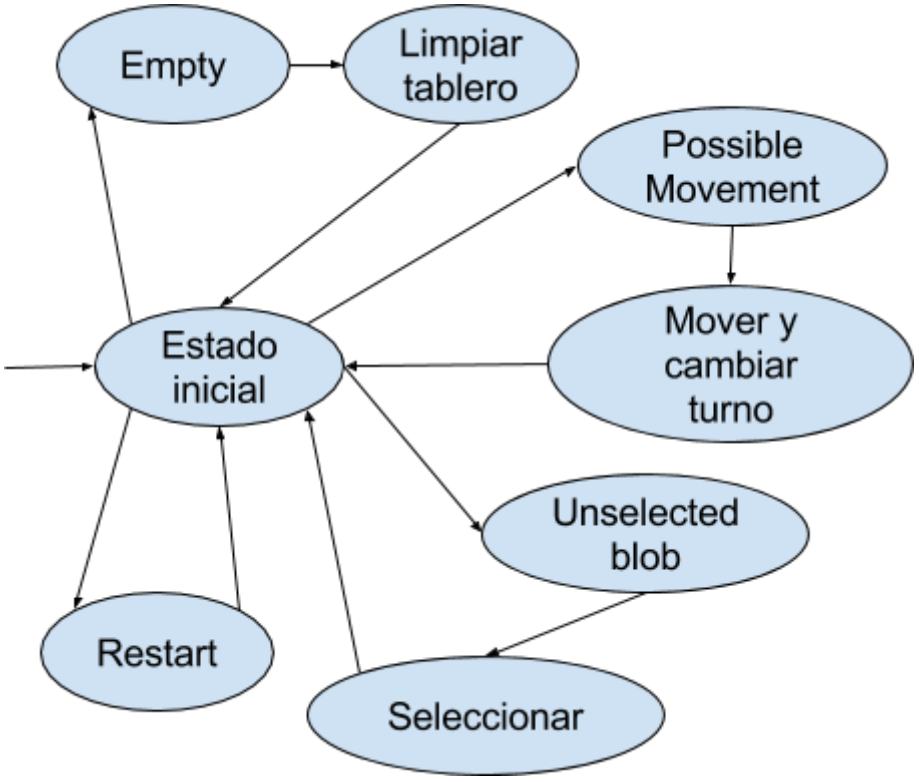
1. Click Point
2. Restart

Con el primero indicamos en qué casillero hizo click el usuario y con el segundo indica que hizo click en el botón de reiniciar el juego.

En el caso del mensaje de restart solo hace falta volver al estado inicial, en el caso del mensaje Click se debe analizar en qué Tile se hizo click:

- Unselected blob: Se hizo click en una mancha sin seleccionar es decir que hay que seleccionarla y mostrar los casilleros a donde se puede mover.
- Empty: Solo hace falta limpiar el tablero. (Deseleccionar las manchas y cambiar los PossibleMovement por Empty)
- PossibleMovement: Es importante aclarar que si se llegó a este estado es porque anteriormente se seleccionó una mancha y se mostraron las fichas de PossibleMovement, es decir que podemos realizar el movimiento de la ficha y cambiar de turno sin hacer ninguna validación. (Por esto es importante no representar casos inválidos)
- El resto de los casos no generan ningún cambio en el modelo.

La máquina de estado se la puede pensar de la siguiente forma:



Se puede ver que en este caso y con el modelo propuesto es muy fácil hacer la estructura de la función de update y aplicar la actualización que corresponda.

Repetición de código

Al tratarse de un juego de tablero es común repetir mucho código para recorrer el tablero y aplicar funciones como por ejemplo el uso de maps y recude o folds.

Por lo cual, se generó una serie de funciones para generalizar todo este comportamiento y así agilizar el desarrollo del resto de las funciones específicas del juego.

Map

```
type alias Transform a b = a -> b

matrixMap : Transform a b -> Matrix a -> Matrix b
matrixMap transform board =
  List.map (\tiles -> List.map transform tiles) board
```

Indexed map

```
type alias IndexedTransform a b = Row -> Col -> a -> b

matrixIndexedMap : IndexedTransform a b -> Matrix a -> Matrix b
matrixIndexedMap transform board =
   List.indexedMap (\row tiles -> List.indexedMap (transform row) tiles) board
```

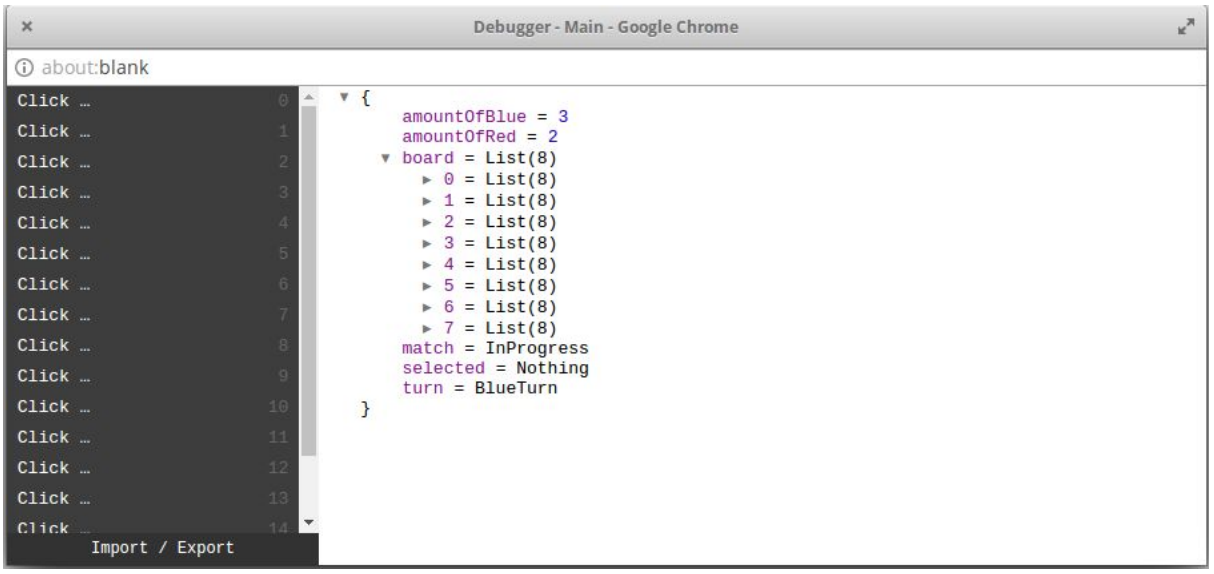
Reduce

```
type alias InnerReduce a b = a -> b -> b

type alias OuterReduce a = a -> a -> a

matrixReduce : InnerReduce a b -> OuterReduce b -> b -> Matrix a -> b
matrixReduce innerReduce outerReduce initialValue matrix =
  List.foldr (\row acum -> outerReduce (List.foldr innerReduce initialValue row) acum)
initialValue matrix
```

Debug

Una de las herramientas más útiles que tienen Elm es su debugger, el cual permite tener la lista de todos los estados que pasó nuestra aplicación con las opciones de exportarlo e importarlo en otro momento para evitarnos volver a generar el mismo estado solo para debuguear un caso en particular.

Para esto es necesario utilizar el comando Elm-Reactor y abrir en un explorador localhost:8000 seleccionando el archivo principal de nuestro código.



Como podemos ver en la imagen, en la sección lateral izquierda se encuentra la lista de todos los estados que fuimos pasando pudiendo seleccionarlos y cambiar el estado de la aplicación a el seleccionado para poder visualizarlo y ver los valores del modelo en ese momento.

Además, podemos ver las opciones de importar y exportar, lo cual nos permite guardar un archivo con todos los estados y cargarlos en otro explorador por ejemplo para ver si esa funcionalidad se comporta de forma correcta en distintos exploradores.

Build

Para compilar la aplicación es necesario ejecutar el siguiente comando:

```
elm-make --output target/elm.js src/Main.elm
```

O correr el escript ./build que se encuentra en la raíz del repositorio.

Este comando compila el archivo Main.elm y sus dependencias generando un archivo de salida en formato JS.

Se decidió compilar a JS, ya que se utilizó CSS y para esto es necesario escribir nuestro propia página index.html que se vincula a la hoja de estilo y carga por javascript la aplicación contenida en elm.js. Además, de cargar los recursos necesarios.

Ejemplo de index.html:

```
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="UTF-8">
  <title>Blob wars</title>
  <link href="https://fonts.googleapis.com/css?family=Coiny" rel="stylesheet">
  <link rel="stylesheet" type="text/css" href="style.css" media="screen"/>
</head>

<body id="body" style="-moz-user-select: none; -webkit-user-select: none;
-ms-user-select:none; user-select:none;-o-user-select:none;" unselectable="on"
onselectstart="return false;" onmousedown="return false;">
</body>

<script type="text/javascript" src="target/elm.js"></script>
<script type="text/javascript">
  var node = document.getElementById('body');
  var app = Elm.Main.embed(node);
</script>

</html>
```

Conclusión

La detección e indicación de errores en tiempo de compilación es muy amigable y fácil de corregir, además de la robustez que esto genera. Ya que Elm es estáticamente tipado, evita errores en tiempo de ejecución facilitando el desarrollo.

Se noto que es sencillo detectar repetición de código similar y abstraerse en funciones más generales de esta forma haciendo más fácil la construcción de otra funciones específicas del juego.

En este caso el modelo fue chico por lo cual ayuda a entender en cada momento lo que está sucediendo, además de que no se generan side effect por ser un lenguaje de programación funcional puro evitando así muchos problemas que como se mencionó en secciones anteriores en iOS son muy frecuentes.

Por último, la arquitectura propuesta por Elm genera un nivel de abstracción excelente para trabajar con programas chicos o juegos sencillos. Pudiendo así plantear una solución inicial en poco tiempo y solo es necesario enfocarse en la programación de funciones específicas del juego sin preocuparnos por la interacción con la UI.