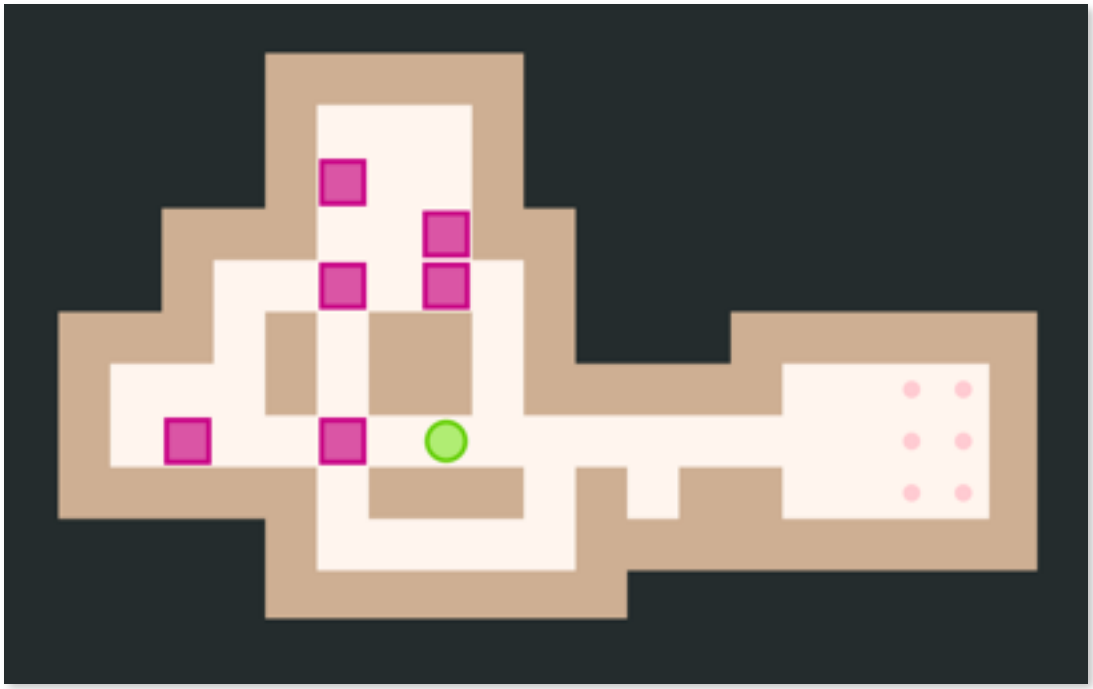
Trabajo final de Programación Funcional

Sokoban



Docentes

- Pablo Ernesto Fidel Martinez López
- Valeria Verónica Pennella

Alumna

- Sakuda, María Eugenia - Legajo: 53191

Indice

Introducción

La realización de este trabajo tiene como objetivo poner el práctica los conceptos desarrollados durante el transcurso de la materia. Con este fin se eligió desarrollar una implementación de Sokoban, un juego de rompecabezas originario de Japón.

Lenguaje

Para el desarrollo del juego se eligió utilizar Elm (0.17). La elección de este lenguaje fue basada por un lado en su similitud con Haskell, lenguaje utilizado durante la cursada de la materia. Por otro lado, brinda herramientas para poder construir de forma sencilla una aplicación web dado que fue creado con este fin, brindando clases para la construcción de un entorno gráfico donde visualizarla. Además, cuenta con una comunidad activa que contribuye con la creación de Packages[1], además de posts e información, aportando soluciones a problemas comunes. Y finalmente, ya se contaba con información sobre el mismo y su arquitectura y dentro del entorno de Swift se menciona con frecuencia por las buenas prácticas establecidas.

Antes de decidir utilizar Elm, se consideró la posibilidad de desarrollar una versión mobile en Swift. Si bien este lenguaje tiene una fuerte base en el Paradigma Funcional, sigue permitiendo el uso de objetos y la mutabilidad de los mismo lo que hizo que quedara descartado.

[1] http://package.elm-lang.org/

El juego

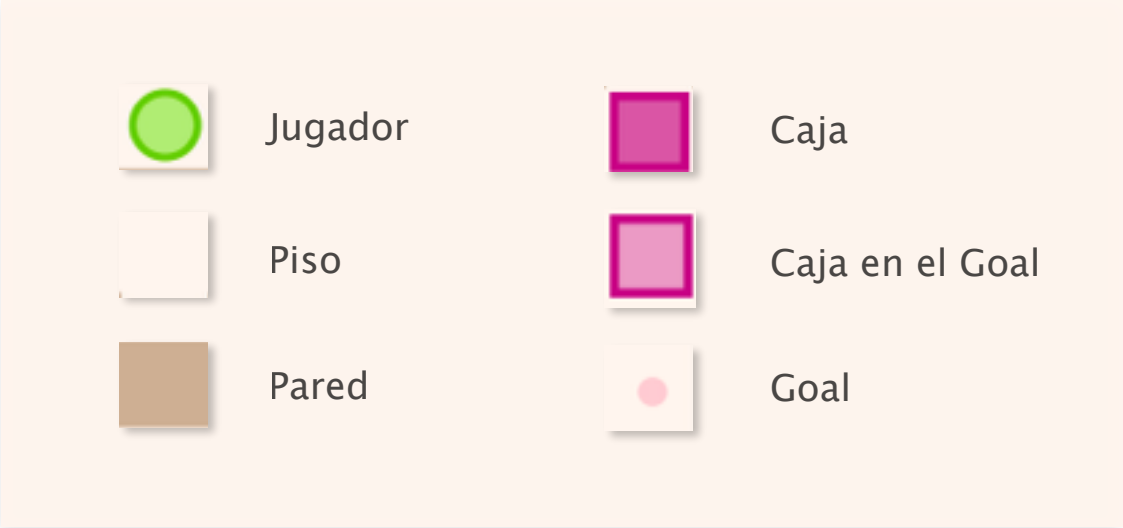
El Sokoban, en japonés “turno de almacén”, tiene como objetivo que el jugador acomode todas las cajas en los lugares indicados (*Goals*). Para llevar a cabo esta tarea puede moverse por el tablero en las cuatros direcciones principales (arriba, abajo, derecha e izquierda) y sólo tiene permitido empujar las cajas dentro del mismo.

El objetivo del juego es dejar cada caja en un goal, en la menor cantidad de movimientos, tanto del jugador (*Moves*) como de las cajas (*Pushes*).

La creación de los escenarios es considerado un arte y hay personas destinadas a este fin. Los algoritmos conocidos que generan este tipo de tableros suelen demorar entre minutos y horas dependiendo de los requerimientos pedidos (cantidad de cajas, tamaño del tablero, etc), dado que analizan diversas posibilidades. Lo mismo sucede para analizar si todavía existen movimientos disponibles o ya no tiene solución. Por este motivo, estas dos última ideas fueron descartadas de la implementación que se detallara posteriormente.

Elementos del tablero

Los elementos que se encuentran presentes en el tablero son:

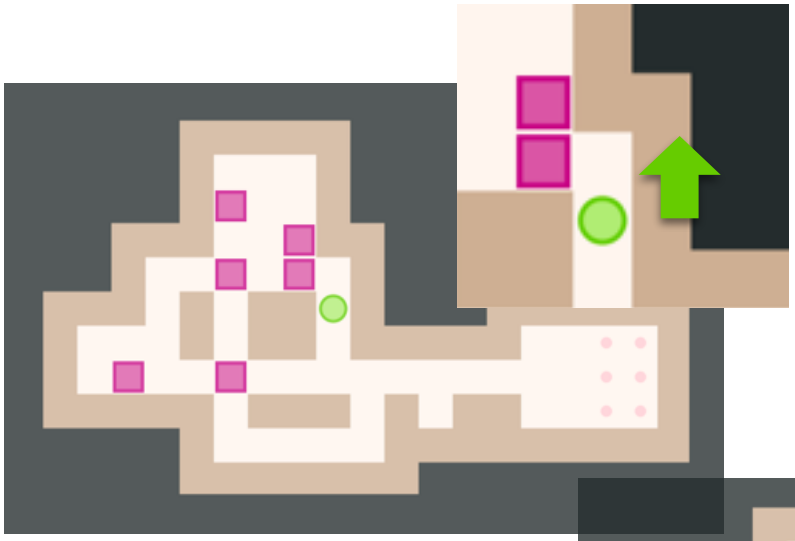


Movimientos permitidos

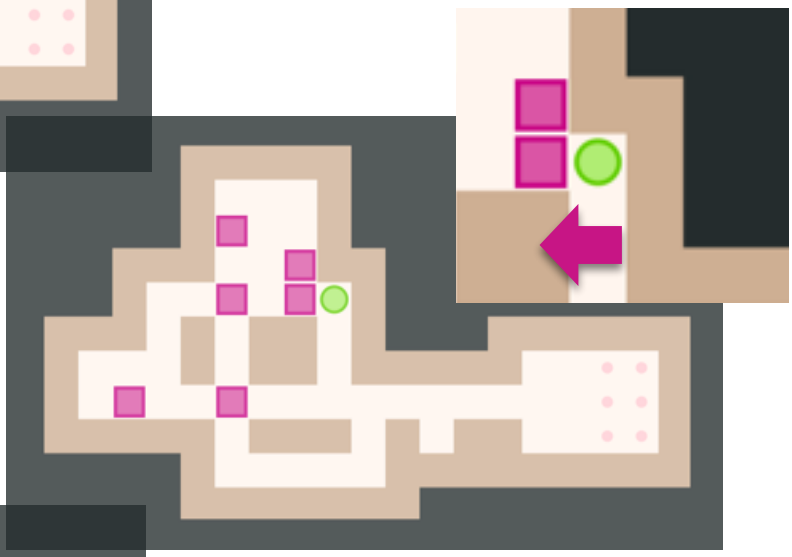
El movimiento de las cajas se encuentra restringido y el del jugador se encuentra restringido a su vez por este. No están permitidos los siguientes movimientos:

- Se quiere mover a una posición donde se encuentra otra caja
- Se quiere mover a una posición donde hay otra pared.

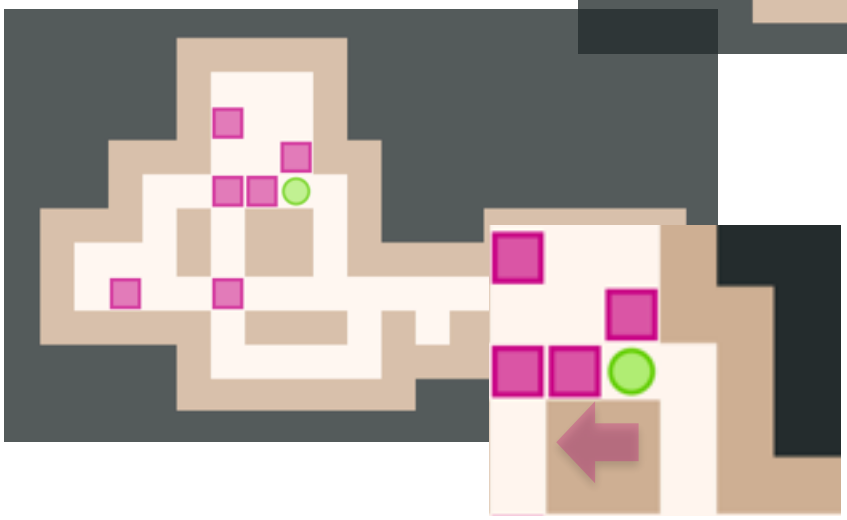
A continuación se describen gráficamente algunos de los movimientos permitidos y no permitidos dentro del juego.



En este caso el jugador puede moverse hacia arriba, donde el casillero se encuentra libre.



En este caso el jugador no puede moverse hacia arriba, porque se encuentra la pared. Si puede moverse hacia la izquierda, desplazando la caja hacia el casillero libre.



Finalmente, la caja de la izquierda no puede moverse dado que el casillero de destino no cuenta con espacio albergarla (hay una caja que la frena).

Prueba del Juego

El juego se encuentra disponible en: https://eugis.github.io/Sokoban/

Implementación

En esta sección se detalla cada una de las secciones que componen el desarrollo del juego así como también las decisiones de diseño que se llevaron a cabo.

Arquitectura de Elm

Elm tiene una arquitectura definida que es necesaria seguir para poder utilizar este lenguaje. Se destacan 5 partes principales:

- Model: es el encargado de mantener el estado de la aplicación.
- View: asigna una vista para un determinado estado
- Update: modifica el estado actual de acuerdo a la señal, comando o acción que recibe
- Subscriptions: es el encargado de determinar que señales, comandos o acciones necesitamos e indicar que se espera que se haga con la información que las mismas traen.
- Main: unifica todos los componentes de arriba para darle sentido al programa.

Modelado

El modelado del juego está compuesto por varios modelos o estructuras que juntas forman el estado completo de la aplicación en un determinado momento.

El estado principal del juego viene dado por el SokobanState que determina si se está jugando en un determinado nivel (*InLevel*), se está en el menú (*Menu*), se completaron todos los niveles disponibles (*EndGame*) o se acaba de ganar el nivel actual (*Win*).

```
type SokobanState = InLevel Level | Win Stats | Menu | EndGame
```

Otra de las estructuras de gran importancia del juego son las Stats. Este modelo es el encargado de almacenar los movimientos, el nivel que se juega y el tiempo para cada uno de los niveles.

```
type alias Stats = { pushes: Int
                   , moves: Int
                   , time: Float
                   , level: Int
                   }
```

Otra de las estructuras importantes que es *Level*, que almacena toda la información referente al estado de un nivel. Para ello cuenta con una tablero (*Board*), la posición del jugador, una lista con las posiciones de todas las cajas, las estadísticas mencionadas arriba y un estado (*LevelState*).

El modelado de las posiciones se hizo mediante la estructura *Location* que forma parte del package Matrix[2]. Esta consta básicamente de una tupla que representan el *x* e *y*.

```
type alias Level = { board: Board
                   , player: List Matrix.Location
                   , player: Matrix.Location
                   , state: LevelState
                   , level: Stats
                   }

type LevelState = Win | WaitingForMove
```

El Board es una estructura que representa los objetos no movibles del la partida, es decir: goals, paredes y pisos. Estos son los elementos que deberían tener los tableros ya cargados y listos para utilizar. Durante el transcurso de composición de los mismos, suelen contar con componentes intermedios como ser: cajas en los goals, cajas solas y el jugador. Internamente se modela con una Matriz de *Component*s, donde los componentes pueden ser los elementos del juego o el vacío (Empty)

```
type alias Board = Matrix Component
```

Los *Component* son las estructuras que modelan cada uno de los elementos del tablero. Lo único que cabe destacar de esta estructura, es el *Bool* que recibe el tipo *Box*. Este booleano representa si el mismo se encuentra sobre un *Goal* o no que es utilizado sólo al momento de renderear el tablero. Dado que los componentes se crean a partir de *Chars* podría suceder que el carácter que se intenta convertir a un Component no sea válido por lo que se cuenta con la representación *Empty* para utilizar con este fin.

Los últimos modelos que restan mencionar son los utilizados para modelar la interacción del usuario, habilitando los cambios de estados: *Action* y *KeyboardInput*.

```
type Component = Box Bool | Floor | Wall | Goal | Player | Empty
```

- *KeyboardInput KeyoboardInput* representa el ingreso de una tecla en el sistema, el mismo sólo es tenido encuentra dentro del nivel (los menús no responden a la utilización de teclas).
- *Tick Time* es utilizado para modificar el tiempo transcurrido desde que se inició el nivel. Es la acción utilizada durante la suscripción que se ejecuta todos los segundos y recibe de parámetro la fecha en segundos.

---

[2] http://package.elm-lang.org/packages/chendrix/elm-matrix/latest/Matrix#

- BackMenu esta acción simplemente indica el paso al menú principal
- NextLevel esta acción se utiliza para indicar que el próximo nivel debe mostrarse. El valor que se pasa es el correspondiente al número del nuevo nivel que se desea ver.

```
type Action = KeyboardInput KeyboardInput
            | Tick Time
            | BackMenu
            | NextLevel Int
```

De todos los ingresos de teclados posibles sólo se consideran las cuatro flechas para desplazar al jugador, *Esc* para volver al menú principal y la *r* para reiniciar el nivel. Estos son los valores representados dentro del *KeyboardInput*.

```
type KeyboardInput = Up | Down | Left | Right | Esc | Restart | None
```

Vistas

Para el modelado de las vistas se utilizaron las librerías disponibles de Elm: *Html*, *Html.Event*, *Html.Attribute*, *Color* y *Graphics.Render*[3], siendo este último un Package perteneciente a un miembro de la comunidad. Esta última librería se utilizó únicamente para realizar la interpretación del tablero de juego, siendo útil para graficar los componentes sin la necesidad de utilizar tantos atributos de estilo, necesarios si se utiliza la librería de *Html* nativa pura. En el fondo, la librería termina pasando las estructuras a Html.

El código que se utiliza para renderear cada vista se encuentra en los archivos *View.elm* dentro de las carpetas pertenecientes a cada uno de los componentes del juego. A su vez hay tres archivos dentro de la carpeta de *general* que tienen la definición de los estilos (*General.Style*), definición de colores utilizados en el juego (*General.Colors*) y un render que tiene el background utilizado en toda la App así como también el layout básico (*General.Render*).

Como se mencionó anteriormente en la "Arquitectura de Elm" la App maneja todas sus vistas mediante el método ***view*** que se encuentra dentro del archivo principal (*App.elm*). De acuerdo al modelado tratado en la sección anterior, hay cuatro grandes divisiones de estado, representadas por cada una de las opciones dentro de *SokobanState* que se detallan a continuación.

VISTA DEL MENÚ Y VISTA DEL FIN DE JUEGO

Estas vistas son bastante simples y no tienen demasiados componentes. Ambas cuentan con un título y un "botón", la primera para iniciar el juego (en el nivel 1) y la otra para volver al menú principal.

---

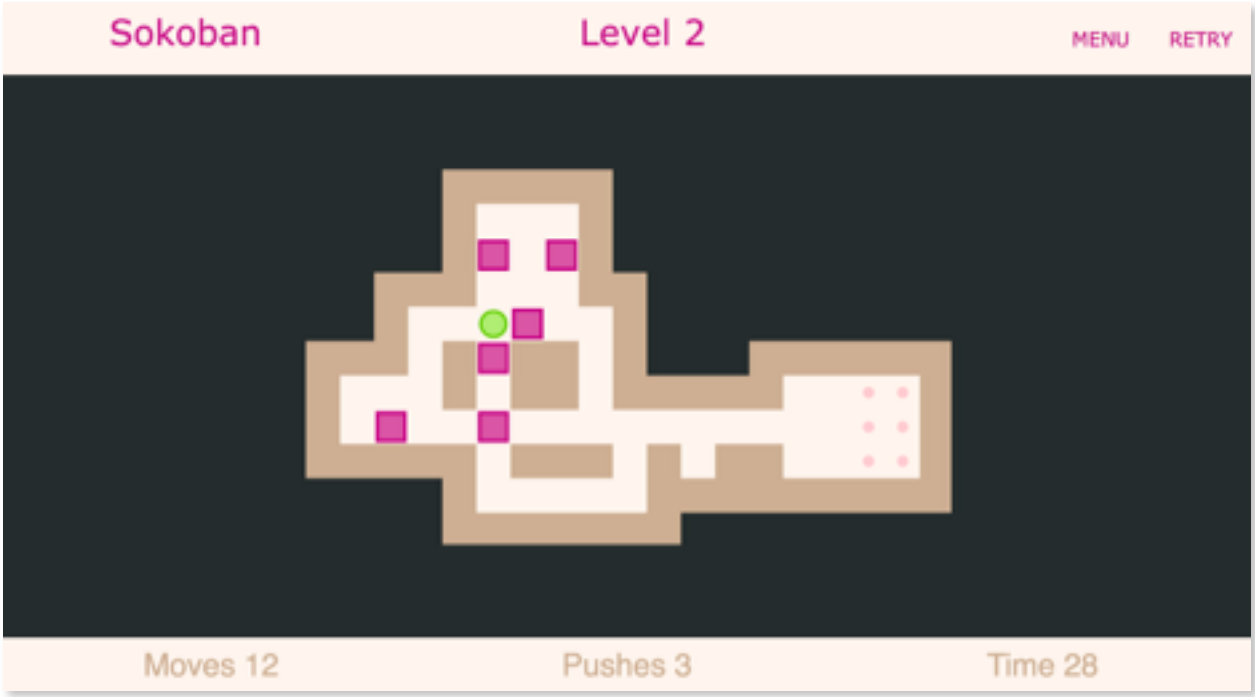
[3] Wrapper de la librería de Html de elm que permite diseñar formas geométricas de manera sencilla: http://package.elm-lang.org/packages/Kwarrtz/render/1.0.2/Graphics-Render#

VISTA DE WIN

Esta vista es la que se muestra al momento de completar el un nivel. Brinda las estadísticas del nivel completado, por este motivo requiere de *Stats* para poder dibujarse. Además de mostrar esa información, da la posibilidad de continuar al siguiente nivel, volver a jugar el mismo o volver al menú principal.

VISTA DE UN NIVEL

La vista de un nivel se compone de tres partes fundamentales: la barra superior, el área central con el tablero propiamente dicho y una barra inferior con las estadísticas del nivel en tiempo real.



La vista utiliza *Flexbox*[4] para distribuir de forma uniforme cada uno de los componentes formando una columna donde cada fila es uno de los elementos mencionados inicialmente.

De las barras tanto superior como inferior, sólo resulta interesante destacar los "botones" de la parte superior (*"Menu"* y *"Restry"*) que técnicamente no son *Buttons* de *Html*. Son divs que tienen un atributo para triggerear las acciones que le corresponden a cada uno sin la necesidad de utilizar explícitamente las subscriptions.

El área central está compuesta por el tablero (*Board*) junto con las cajas y el jugador. El dibujado de todo el sector se basa en que cada uno sabe dibujarse a sí mismo, entonces la idea es llegar a que cada componente tenga la información necesaria para

[4] Flexbox: http://www.w3schools.com/css/css3_flexbox.asp

llevar a cabo esa tarea y luego la estructura que los contiene terminando uniendo cada uno de los pedazos en una vista más compleja.

Matrix tiene un *mapWithLocation* que resultó de utilidad para poder obtener una matriz de Html a partir del tablero de componentes. Como se mencionó durante el modelado, las cajas tiene un parámetro booleano para dibujarlas de forma diferente si se encuentran sobre un *Goal*. Dada esta condición, al momento de realizar el *map* que dibuje las cajas de la lista se checkea si el elemento de la posición indicada es un *Goal* o no.

```
renderBoxes: Level -> Form Action
renderBoxes level =
        List.map (\l -> Component.View.render l
                                        (Box (component level.board l
                                                |> Component.Type.isGoal)
                                        )
                ) level.boxes
        |> group
```

La función *group* que aparece algunas de las funciones "render" es la encargada de agrupar distintos *Form*s tomando una lista de *Form*s y devolviendo uno sólo. Este método pertenece a la librería *Graphics.Render* mencionada con anterioridad.

El tablero general, al realizar la transformación para obtener el Html y unificarlo con el resto de la vista, utiliza un factor de corrección, porque el posicionamiento quedaba desfasado.

Lógica del juego

Una vez dentro del nivel, para jugar deben utilizarse las flechas del teclado. Estos movimientos son captados en las *subscriptions* que al recibir la entrada de una tecla ejecutan la función *keyboardIntput*. Esta función devuelve una acción *KeyboardInput* (vistos en el modelado) basado en el *key code* de la tecla ingresada.

```
keyboardInput: KeyCode -> Action
keyboardInput keyCode =
                case keyCode of
                        27 -> KeyboardInput Esc
                        37 -> KeyboardInput Left
                        38 -> KeyboardInput Up
                        39 -> KeyboardInput Right
                        40 -> KeyboardInput Down
                        82 -> KeyboardInput Restart
                        otherwise -> KeyboardInput None
```

KeyboardInput KeyboardInput
KeyboardInput Esc

Una vez que se "dispara" la acción se verificar a través del *PatternMatching* qué tipo de *KeyboardInput* se está procesando. En caso de ser una *Restart* o *Esc* se ejecutan las acciones correspondientes a recargar el nivel o volver al menú. Si no se trata de ninguna de las teclas representativas para el juego se devuelve el mismo estado, sin modificación alguna. Por último, si se trata de alguna de flechas, se realiza el *move* del nivel transformando el K*eyboardInput* en una *Direction* (*tupla* cuyos valores representas las 4 direcciones posibles). Antes de devolver el estado, la función *newState* verifica que el nivel no haya sido contemplado en cuyo caso devuelve un *Win* con las *stats* necesarias.

```
updateLevelWithKeyboard: KeyboardInput -> Level -> SokobanState
updateLevelWithKeyboard keyboardInput level =
    case keyboardInput of
        Esc -> Menu
        None -> InLevel level
        Restart -> InLevel (LevelManager.restart level)
        otherwise -> newState (move (direction keyboardInput) level)


            newState: Level -> SokobanState
            newState level =
                case level.state of
                    Level.Type.Win -> Win level.stats
                    otherwise -> InLevel level
```

Al momento de modificar el nivel en sí (función *move*), se tiene en cuenta dos opciones posibles para realizar el movimiento, el resto de las posibilidades hacen que el jugador no logre moverse:

- La posición de destino se encuentra libre, en cuyo caso se realiza el movimiento y el contador de movimientos aumenta en 1.
- La posición de destino tiene una caja y aplicando la transformación de la dirección en el mismo sentido que tuvo el jugados, el casillero se encuentra libre. En este caso, se aumenta tanto la cantidad de movimientos como el de *pushes*.

La validación la lleva a cabo la función isValidDirectionMovement. Si el movimiento no es posible devuelve el mismo nivel y en caso de poder efectuarlo mueve al jugador en la dirección requerida.

```
        move: Action.Type.Direction -> Level -> Level
        move direction level =
            if (isValidDirectionMovement direction level)
            then movePlayer direction level
            else level
```

El mover el jugador pude desencadenar el desplazamiento de una caja. Primero se calcula la nueva posición del jugador. Se verifica si en ese lugar hay alguna caja y en ese caso se mueve esta última. En ambos casos se modifican el parámetro *player* y *boxes* del nivel dejando el tablero cómo está (sólo posee la información estática una vez arrancado el juego). Se verifica el estado del juego, se valida si se cumple la condición de fin, donde las cajas deben estar en el mismo lugar que los *Goal*s. Finalmente se modifican las *stats* para incrementar los movimientos y las *pushes* en caso de ser necesario.

```
movePlayer: Action.Type.Direction -> Level -> Level
movePlayer direction level =
let
  updatedPlayer = updateLocation level.player direction
  updatedBoxes = moveBoxes level.boxes updatedPlayer direction
  newLevelState = updateLevelState level.board updatedBoxes updatedPlayer
pushes = updatedBoxes /= level.boxes
in
  { board = level.board
  , boxes = updatedBoxes
  , player = updatedPlayer
  , state = newLevelState
  , stats = incrementStats level.stats pushes
  }
```

Generación de los niveles

El juego cuenta con un LevelManager que se encarga de levantar los niveles de una lista de listas de *Char*s. Si bien hubiera sido más prolijo tener una lista de Matrices o *Board*s ya creados, la idea es que la estructura estuviera creada para que los usuarios puedan ingresar en un futuro los tableros que quisieran probar mediante texto. Por este motivo, esta implementación permite que la diferencia entre cargar los niveles que viene con el juego y los de los usuarios esté en el parseo de la entrada y no en generar el tablero.

Cuando el juego requiere un nivel determinado llama la función *level* con el número de nivel deseado. Esta función puede devolver un nivel o *Nothing*, dependiendo de si el nivel requerido se encuentra disponible así como también si al momento de crearlo no hay errores, es decir es un tablero válido.

Se mapea la lista para construir una lista de listas de *Component*s una vez conseguido esto se crea el Board, con la peculiaridad que incluye las cajas así como también al jugador. Este tablero se utiliza para sacar las posiciones del jugador y armar las listas de las cajas y una vez que ya se dispone de esa información por separado se borran estos últimos elementos del tablero reemplazándolos por *Floor*s o *Goal*s según corresponda (función *clear*). Por último se inicializan las *Stats* con el número de nivel correspondiente y se retorna el estado.

Problemas encontrados durante la implementación y posibles mejoras

Problemas encontrados

Durante el desarrollo de este juego se presentaron algunos problemas que fueron resueltos. Los mismos se relatan a continuación

TABS

Si bien la herramienta utilizada para desarrollas (Atom) es consciente que los Tabs no compilan en Elm y los reemplaza por espacios, al copiar información muchas veces se introducen estos *asciis* resultando en un error de compilación difícil de detectar. Esto sucedió al principio al incluir los *RGB* de los colores.

IMPORTS DE MODELOS

Al importar módulos con la intención de utilizar PatternMatching de tipos custom (como fue el caso de SokobanStates) es necesario aclarar que no sólo incluya el tipo, sino todo lo que involucra el mismo:

```
import Component.Type exposing (Component(..))
```

```
import Component.Type exposing (Component)
```

Si bien la diferencia en un principio es sutil, si se quiere verificar si el componente es un *Box* o un *Player*, es necesario incluirlo con la sentencia de arriba, caso contrario sólo podremos indicar que un parámetro es de tipo *Component*.

USO DE HTML Y CSS

Si bien ese fue un problema personal, por la falta de costumbre, la mayor parte del tiempo estuvo invertido para lograr acomodar cada pieza de la vista en el lugar deseado y aún así hay cosas con las que no me encuentro del todo conforme. Al buscar cómo realizar muchas de las cosas necesarias, las soluciones vienen dadas para CSS común y me resultó complicado poder pasarlo a la clase Html de Elm, si bien no encontré nada que no se pudiese hacer.

GENERACIÓN DE NIVELES Y EVALUACIÓN DE GAME OVER

La idea original de la implementación contaba con la generación de niveles y poder predecir el game over de un nivel. Durante el proceso de investigación para poder llevar a cabo esto se descubrió, como se mencionó en la sección de juego, que generar un

nivel no es un algoritmo sencillo que pueda entrar en un tiempo de espera razonable para generarlo en el inicio de cada partida.

Así mismo, validar el game over también requiere de tiempo para que sea confiable y pueda detectar cualquier tablero que ya no cuente con una solución. Para poder incluirlo en el análisis de cada jugada, debería poder resolverse de forma casi instantánea al menos en lo que respecta al tiempo de un jugador. Por este motivo, tampoco es viable para realizarse en tableros grandes o muy complejos.

Posibles mejoras

POSICIONES Y CÁLCULO DE PUNTAJE

Sería deseable que en un futuro se pudiese establecer una fórmula para calcular el puntaje obtenido a partir de las stats de cada nivel. De este modo se podría almacenar los puntajes de cada jugador en una base de datos para poder tener referencia de los mismos y armar una tabla con las mejores posiciones totales y por nivel.

CARGA DE NIVELES POR PARTE DE EL USUARIO

Dado que los niveles son un elemento predeterminado del juego y va a llegar un punto a partir del cual el jugador no va a poder seguir jugando, una opción es que pueda probar sus propios niveles o buscarlos en la red.

Parte de la lógica de validación y paseo del nivel se encuentra en el *LevelManager*, aunque no se contempla, en este momento, el ingreso de la información al programa. Haría falta agregar una vista donde poder cargar el nivel custom en forma de Springs con un formato predefinido y posteriormente modificar las *stats* para que pueda tener un número de nivel o que sea custom en cuyo caso no debería disponer la opción de *Next Level*.

Conclusiones

Más allá de haber trabajado con Haskell para realizar ejercicios sencillos, trabajar con un lenguaje funcional puro para la realización de un proyecto completo aunque pequeño, es una gran experiencia. Resulta muy distinto respecto a la utilización de lenguajes con un enfoque funcional, pero que siguen teniendo problemas en runtime y permiten introducir “vicios” de otros paradigmas.

Fue interesante poner en práctica la arquitectura de Elm que en varias oportunidades había escuchado nombrar a modo de ejemplo de buenas prácticas dentro del ambiente de Swift y entender un poco mejor las ventajas de la inmutabilidad de las estructuras.

Bibliografía

Documentación general

Página principal de Elm: http://elm-lang.org/docs

Página principal de Packages: http://package.elm-lang.org/

“How I Structure Elm Apps”: http://blog.jenkster.com/2016/04/how-i-structure-elm-apps.html

Tutorial básico de elm: https://www.elm-tutorial.org/en/03-subs-cmds/01-subs.html

“Quickly create github.io pages for your Elm projects”, https://jasonneylon.wordpress.com/2015/11/08/quickly-create-github-io-pages-for-your-elm-projects/

“Developing games in Elm Functional Programming”: http://gelatindesign.co.uk/developing-games-in-elm/functional-programming

“A complete guide to Flexbox": https://css-tricks.com/snippets/css/a-guide-to-flexbox/

Packages de Elm utilizados

Matrix: http://package.elm-lang.org/packages/chendrix/elm-matrix/latest/Matrix#

Graphics.Render: http://package.elm-lang.org/packages/Kwarrtz/render/1.0.2/Graphics-Render#

Guards: http://package.elm-lang.org/packages/Fresheyeball/elm-guards/latest