Escuela de Ingeniería y Gestión

Trabajo Práctico Final

Sokoban

Alumno: Giuliano Scaglioni (57244)

Fecha: 08/02/2020

Índice

1. Introducción **1**

2. Planteo del problema **2**

2.1. Sokoban 2

2.2. Funcionalidades de la implementación 3

3. Aspectos técnicos **4**

3.1. Arquitectura 4

3.2. Elementos del juego 5

3.2.1. Tablero 5

3.2.2. Juego 5

3.3. Funciones relevantes 6

3.4. Traversals 6

3.5. Archivos de tableros 7

3.5.1. Parser 8

3.6. Tests Unitarios 10

3.7. Interfaz gráfica y entrada del usuario 11

4. Conclusiones **12**

Referencias **13**

1. Introducción

En el presente trabajo se comenzará explicando cuál es la idea a desarrollar, la motivación de la misma y las dificultades que se espera puedan llegar a presentarse. En la sección siguiente, *Planteo del problema*, se desarrollará en mayor detalle las características de la aplicación desarrollada en cuanto a funcionalidad brindada al usuario. Luego, en la sección *Aspectos técnicos* se explicarán las decisiones tomadas vinculadas a la implementación, es decir, cómo se decidió organizar el programa, la arquitectura planteada, funciones principales, bibliotecas utilizadas entre otros detalles importantes. Finalmente se presenta la sección de *Conclusiones* donde se explica que se aprendió, observaciones con respecto al trabajo y posibles mejoras a futuro.

El problema a resolver consiste en implementar el conocido juego japonés *Sokoban*. La elección del mismo se debe principalmente a que, en este juego, la evolución de un estado a otro del mismo se realiza ante los movimientos (acciones) que realiza el jugador. Esto lo hace adecuado para modelarlo como transformaciones de estado a acción a estado (`State->Action->State`). Por otro lado, ofrece la oportunidad de aprender otros temas como *Monadic Parsing* útiles para cargar niveles desde archivos para tal fin.

Se espera que las dificultades en el desarrollo se presenten sobre todo en temas desconocidos, como son el manejo de la librería de gráficos y el aprendizaje y entendimiento de los *Monadic Parsers* (Hutton y Meijer, 1998).

2. Planteo del problema

En esta sección se introducirá el juego a desarrollar, las características del mismo y las funcionalidades presentes en la implementación.

2.1. Sokoban

Sokoban es un rompecabezas japonés. El juego consiste de un tablero de celdas donde las mismas pueden ser piso, paredes o almacén. Por otro lado, sobre esta grilla se sitúa el jugador y una o más cajas. El objetivo es utilizar al jugador para desplazar las cajas desde sus posiciones iniciales a los almacenes. El juego finaliza cuando todas las cajas del tablero se encuentran ubicadas en los objetivos.

Las acciones que puede realizar el jugador son desplazarse hacia arriba, abajo, izquierda y derecha; y desplazar cajas empujándolas. Esta última acción se da cuando el jugador avanza hacia la posición de una caja y la misma tiene espacio para desplazarse en esa dirección, caso contrario, el jugador permanece en su celda.

En la Figura 1 se puede observar los distintos tipos de celdas. Las celdas correspondientes a las paredes, en color azul. El piso en color blanco, y los lugares de almacenamiento (almacenes) marcados con un punto gris. Por otro lado, sobre el tablero se sitúa una caja, en color rojo, y el jugador, representado por un circulo violeta.

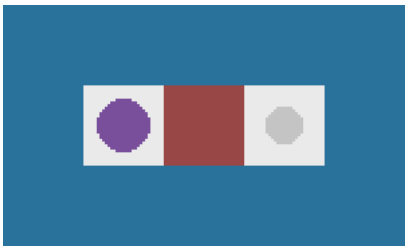

Figura 1: Ejemplo de tablero de Sokoban

En este ejemplo, el jugador debería hacer un movimiento hacia la derecha para desplazar la caja a su lugar de almacenamiento. Si bien este ejemplo es trivial, el juego se puede complejizar en otros casos. Por ejemplo, notar que una caja que se sitúa en una esquina no puede ser retirada de ese lugar lo cual dificulta más la resolución cuando se tienen varias cajas.

2.2. Funcionalidades de la implementación

La implementación permite al usuario jugar distintos niveles utilizando una interfaz gráfica. Al iniciar el juego, se presenta una pantalla de título y se cargan los niveles a jugar desde un archivo de texto.

El desarrollo del juego consiste en ir resolviendo los distintos tableros presentados. A medida que se van finalizando los niveles se muestra una pantalla de finalización de nivel y se procede al próximo. Al finalizar todos los niveles termina el juego.

La posibilidad de cargar niveles desde un archivo de texto es lo que permite que el usuario pueda extender el juego de forma casi ilimitada y definiendo el nivel de dificultad en base a los niveles que decida jugar. Para simplificar la obtención de niveles se decidió utilizar un formato de archivo comúnmente utilizado, esto permite conseguir niveles en internet fácilmente.

3. Aspectos técnicos

3.1. Arquitectura

La aplicación se separó en tres componentes o módulos principales, como se observa en la Figura 2. El primero, *Game*, define el estado del juego y las transiciones del mismo. Por ejemplo, si se finaliza un tablero, se encarga de hacer la transición a la pantalla de finalización de nivel y luego al próximo tablero. Luego se encuentra *Sokoban*, que define los elementos del tablero y las transformaciones del mismo. Finalmente está *Parser*, encargado de realizar el parseo de niveles.

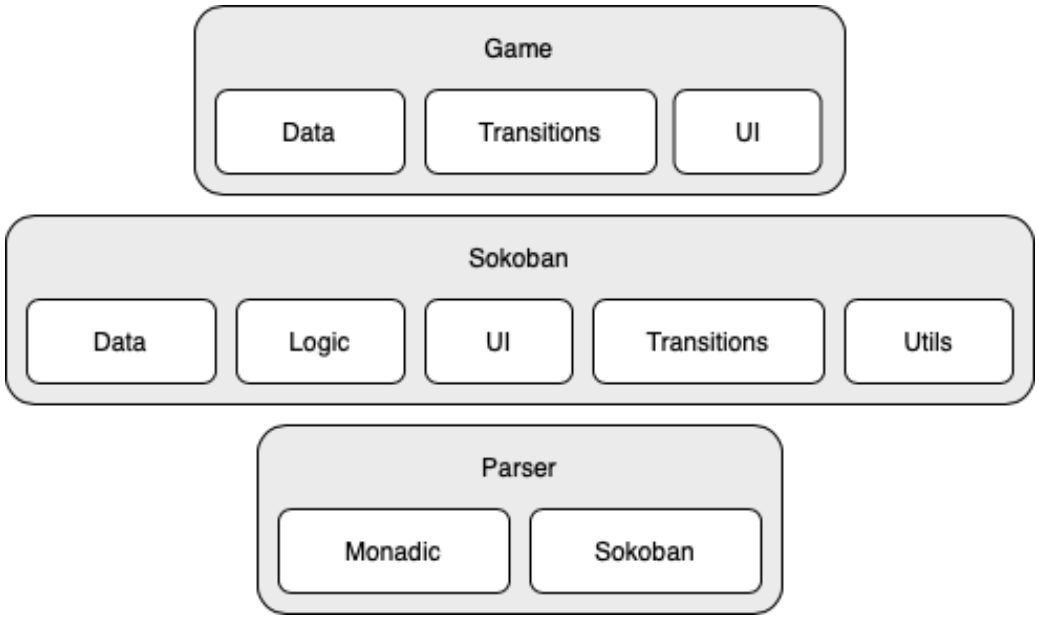

Figura 2: Módulos

Para el caso de los módulos *Sokoban* y *Game*, se definen submódulos *UI* que dibujan los distintos elementos de cada uno. En el primero se indica cómo se debe dibujar cada celda, cada fila y el tablero completo. En el segundo se toma en cuenta el estado del juego para saber si dibujar el tablero, la pantalla de título o de finalización de nivel.

Esta separación de los módulos y de las funciones presentes en cada uno permitió realizar tests unitarios de forma sencilla. Mas adelante se explica en detalle este punto.

3.2. Elementos del juego

3.2.1. Tablero

Para la representación del tablero se probaron dos alternativas. La primera aproximación separaba los elementos estáticos como son las paredes, el piso y los lugares de almacenamiento, del jugador y las cajas. El tablero estaba formado por filas de celdas, donde cada celda podía ser alguno de los elementos fijos mencionados. Luego se contaba con una estructura que contenía esta definición del tablero, la posición (fila y columna) del jugador y una lista de posiciones de las cajas. Esta definición permitió implementar el juego pero se encontraron algunas dificultades al construir esta estructura al parsear los niveles desde un archivo.

La segunda alternativa representa de forma mas natural al tablero y permite construirse mas simple de forma recursiva. Se define un objeto como el jugador o una caja. Una celda puede ser una pared, un piso con tal vez un objeto o un lugar de almacenamiento con tal vez un objeto. De esta forma, el tablero esta formado por filas de celdas. Se puede notar además, que la explicación de la estructura es mucho mas simple y fácil de entender.

```
data Object = Player | Box
data Cell   = Floor (Maybe Object)
            | Storage (Maybe Object)
            | Wall
type Row    = [ Cell ]
type Board  = [ Row ]
```

Código 1: Representación de los elementos utilizada.

3.2.2. Juego

Por otro lado, existe la estructura juego que contiene el estado actual del juego. Esto es, la pantalla en donde se encuentra el jugador (título, tablero, fin de nivel o fin del juego) y una lista con los tableros o niveles restantes.

3.3. Funciones relevantes

Como se mencionó en la introducción, el tablero va cambiando de estado como resultado de la aplicación de transformaciones sobre el mismo. Estas transformaciones a su vez son funciones de las acciones del jugador. Por ejemplo, si el jugador presiona la tecla izquierda, se aplica la transformación "mover a la izquierda". En otras palabras, *los cambios de un estado a otro son funciones de alto nivel que reciben una dirección y devuelven una función que recibe un tablero y devuelve un tablero.*

Para desplazar un objeto se utiliza una función que recibe una posición y devuelve una función que recibe una dirección y devuelve una función que recibe un tablero y devuelve un tablero. Aplicando dicha función a una posición del tablero, se obtiene una función que dada una dirección devuelve una función que permite transformar el tablero en otro en donde el objeto (si es que hay uno) se desplazó en esa dirección siempre que sea posible el movimiento. Por otro lado, se cuenta con una función que dado un tablero devuelve, tal vez, la posición del jugador. Mediante la composición de ambas funciones es posible realizar la transición de un tablero en otro.

La idea de transformación de estados, se aplica también al estado del juego. El mismo evoluciona como resultado de eventos de entrada del usuario (teclado) y el paso del tiempo. A su vez, la función de transición de estado del juego aplica la función de transición del tablero en caso de que el estado actual del mismo sea la pantalla de juego.

Es importante remarcar que es posible definir todo el juego como funciones puras y dejar las partes impuras para el final, esto es, la entrada del usuario y el tiempo.

3.4. Traversals

Para aprender sobre el tema Lens (Meijer, Fokkinga, y Paterson, 1991), se decidió implementar algunas funciones utilizando este concepto. Como el tablero esta representado como una lista de listas de celdas, se pensó mas adecuado utilizar Traversals (Lämmel y Jones, 2003), estos permiten recorrer la estructura de forma similar a un fold. A su vez se pueden componer entre si, de esta forma es mucho mas sencillo escribir una función que dada una posición y un tablero devuelva la celda en esa posición. Para realizar esto, se utilizó el traversal `ix`, este

recorre el valor en un determinado índice si es que existe. Componiendo este traversal aplicado a una columna j con el mismo traversal aplicado a una fila i podemos obtener un traversal que permite recorrer la celda en la posición (i, j). Luego usamos este traversal, aplicándolo a la función `preview` para obtener (tal vez) el valor de la celda. Esto se puede observar en el Código 2.

```
cellAt :: Position -> Board -> Maybe Cell
cellAt (row, col) = preview (ix row . ix col)
```

Código 2: Definición de `cellAt` usando traversals.

3.5. Archivos de tableros

Se utilizó un formato popular para representar tableros de Sokoban. En el Código 3 se muestra la gramática utilizada para representar un tablero como texto.

```
               <board> ::= <row><board> | <row>
               <row>   ::= <cell><row> | <cell>
               <cell>  ::= <wall> | <floor> | <storage> | <box>
                         | <box_at_storage> | <player_at_storage>
              <wall>   ::= '#'
              <floor>  ::= ' '
            <storage>  ::= '.'
                <box>  ::= '\$'
     <box_at_storage>  ::= '*'
  <player_at_storage>  ::= '+'
```

Código 3: Gramática del formato de archivo de un tablero.

Inicialmente se pensó en guardar un tablero por archivo, pero, para simplificar el código, se decidió utilizar la misma gramática y guardar todos los tableros en un mismo archivo separando cada uno por una línea con tres guiones (`---`). De esta forma sólo se lee un único archivo.

3.5.1. Parser

Para generar el tablero a partir de la gramática descripta anteriormente, se construyó un parser. Para esto se recurrió a los *Monadic Parsers*. Un parser es una función que recibe un string y devuelve una lista de resultados. Los elementos de esta lista corresponden a una tupla donde el primer valor es el producto del parseo y el segundo valor es el resto del string que quedó sin parsear.

Al ser mónadas, los parsers se pueden combinar utilizando la notación *do* como se puede ver en el Código 4. Lo que se hace es aplicar los parsers `p1` hasta `pn`, guardando los resultados en `a1` hasta `an`. Luego se utiliza una función `f` que permite combinar todos los resultados. A cada línea de la forma `ai <- pi` se la suele llamar generador.

```
parser = do a1 <- p1
            ...
            an <- pn
            return f a1 ... an
```

Código 4: Combinación de varios parsers.

Recordando que un parser es una función que recibe un string y devuelve una lista de resultados, podemos escribir un parser de caracteres que parsee un único carácter de la siguiente forma:

```
next :: Parser Char
next = Parser (\case
                [] -> []
                (c:cs') -> [(c, cs')]
              )
```

Código 5: Parser del próximo carácter.

Luego podemos reutilizar este parser, y esta es una de las ventajas de utilizar este tipo de parsers, para parsear un carácter en específico como se observa en el Código 6.

```haskell
char :: Char -> Parser Char
char c = do parsed <- next
            if parsed == c then return c else empty
```

Código 6: Parser de un carácter `c` específico.

En este caso se hace uso de un parser `empty` que para cualquier string no devuelve ningún resultado.

Luego podemos reutilizar el parser de carácteres para construir un parser de strings como se ve en el Código 7.

```haskell
string :: String -> Parser String
string ""     = return ""
string (c:cs) = do char c;
                   string cs;
                   return (c:cs)
```

Código 7: Parser de strings.

Entonces, por ejemplo, para parsear una pared del tablero, podemos escribir un parser como el del Código 8 que consume un carácter `#` y devuelve una celda pared.

```haskell
wall :: Parser Cell
wall = do char '#'; return Wall
```

Código 8: Parser de una celda pared.

En el paper de *Monadic Parsing* a su vez se introducen otros combinadores de parsers que permiten por ejemplo, combinar los resultados de varios parsers con un operador `++`. Es decir, si tenemos el parser `p1` y el parser `p2`, el parser `p1 ++ p2` parsea un string con `p1` y `p2` y devuelve la concatenación de los resultados. Luego podemos definir un nuevo combinador `<|>` (o `+++` en el paper) que haga lo anterior pero solo devuelva un resultado. De esta forma podemos escribir

un parser que parsee una celda como se observa en el Código 9. Notar cómo la definición del parser de celdas se asemeja mucho a la producción en la gramática presentada anteriormente si utilizamos el operador `<|>`.

```
cell :: Parser Cell
cell = wall <|> player <|>
       playerOnStorage <|> ... <|> floor
```

Código 9: Parser de una celda cualquiera.

Luego, utilizando recursión, de la misma forma que construimos el parser de strings a partir del parser de chars, podemos parsear una fila y luego todo el tablero.

Finalmente, es importante remarcar que existen bibliotecas que permiten realizar parsers de forma más sencilla pero se decidió implementarlos siguiendo los pasos del paper como forma de aprendizaje.

3.6. Tests Unitarios

Para la implementación de las funciones de transformación del tablero y de los parsers se realizó TDD (test-driven development). Esto agilizó mucho el desarrollo ya que, al definir primero la especificación (en forma de tests unitarios) y luego realizar la implementación, se pudieron identificar errores de forma mas rápida y probar las distintas partes de forma temprana.

Para realizar los tests se utilizó la biblioteca *HSpec* que permite definir los casos de forma clara. Por ejemplo, en el Código 10 se define un caso para el parser de una celda.

```
describe "cell" $ do
    it "returns wall when parsing #" $
        let [(result, remaining)] = parse cell "#"
        in (result, remaining) `shouldBe` (Wall, "")
```

Código 10: Ejemplo de caso de prueba.

3.7. Interfaz gráfica y entrada del usuario

Para la implementación de la interfaz gráfica y la lectura de la entrada de teclado, se recurrió a una biblioteca que provee ambas funcionalidades, *Gloss*. Se analizaron varias alternativas, se encontró que esta opción brinda ambas funcionalidades necesarias. Por otro lado, otra de las ventajas que provee es que permite separar las funciones para transformar el estado como respuesta a eventos (entrada de teclado) de las funciones para dibujar el estado en pantalla. Además permite combinar de forma sencilla las funciones de dibujo, obteniendo un código mas claro como se puede observar en el Código 11 donde se presenta como se dibujaría el tablero utilizando recursión explicita y como se dibujaría una fila con el esquema `foldr`.

```
drawBoard :: Board -> Picture
drawBoard [] = blank
drawBoard (r:rs) = drawRow r <> Translate 0 (-1*cellSize) (drawBoard rs)

drawRow :: Row -> Picture
drawRow = foldr (\c pic -> drawCell c <> Translate cellSize 0 pic)
          blank
```

Código 11: Funciones para dibujar el tablero.

En el Código 12 se muestra como se transforma el tablero ante distintos eventos. En este caso, solo se considera cuando se presionan las flechas del teclado, ante cualquier otro evento, la función de transformación es la identidad.

```
handleInput :: Event -> Board -> Board
handleInput (EventKey (SpecialKey KeyUp) Down _ _)    = tryMovePlayer U
handleInput (EventKey (SpecialKey KeyDown) Down _ _)  = tryMovePlayer D
handleInput (EventKey (SpecialKey KeyRight) Down _ _) = tryMovePlayer R
handleInput (EventKey (SpecialKey KeyLeft) Down _ _)  = tryMovePlayer L
handleInput _                                         = id
```

Código 12: Manejo de la entrada de teclado.

4. Conclusiones

Como primera conclusión, se considera que la puesta en práctica de los conocimientos aprendidos en la materia terminan de demostrar la expresividad y sencillez con la cual se pueden implementar programas utilizando el paradigma de la programación funcional. Una de las principales ventajas que trae es la cantidad de errores que se evitan al escribir funciones puras y solo introducir los efectos sobre el final de la implementación para comunicarse con el usuario (pantalla y teclado).

Por otro lado, la realización del trabajo brindó la oportunidad de aprender temas muy interesantes como *Lenses* y *Traversals*, y *Monadic Parsing*. Notándose en este último punto, un alto grado de reutilización de los componentes construidos.

Finalmente, haciendo referencia a posibles mejoras y extensiones del trabajo, se encuentra, en primer lugar, la posibilidad de implementar un resolvedor (o solver) utilizando árboles de juego, lo cual es muy adecuado para resolverse utilizando esquemas de recursión. Y en segundo lugar, mejorar la interfaz gráfica del usuario, agregando imágenes y mejorando la interacción.

Referencias

Hutton, G., y Meijer, E. (1998). Monadic parsing in haskell. *Journal of functional programming*, *8*(4), 437–444.

Lämmel, R., y Jones, S. P. (2003). Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, *38*(3), 26–37.

Meijer, E., Fokkinga, M., y Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. En *Conference on functional programming languages and computer architecture* (pp. 124–144).

Tutorial for the lens library. (s.f.). Descargado de `https://bit.ly/3a21YuX`