Trabajo Práctico de Programación Funcional

Introducción

Este trabajo consta de:

- La implementación de una librería para comunicación con dos jugadores a través de sockets TCP.
- La implementación servidor de un juego de cartas estilo Hearthstone que utiliza la librería de comunicación mencionada.

Planteo del problema

La aplicación es un servidor acepta dos conexiones y luego sirve una partida interactiva para usar
en un cliente tipo `telnet`. Los jugadores tienen tres acciones posibles durante su turno y la
accion a realizar la ingresan como un comando de texto.

Sección Técnica

Arquitectura del sistema

El trabajo contiene los siguientes 9 módulos de código Haskell:

- Board: Librería para trabajar con el tablero de juego.
- Deck: Librería de manejo del mazo de cartas.
- ExampleCards: Definiciones de instancias de cartas.
- ExampleDecks: Definiciones de instancias de mazos.
- Game: El modelo de datos y la lógica de negocio del juego.
- Main: El punto de entrada para el ejecutable, crea el controlador de red e inicia la

UI del juego
con el mismo.

- NetworkController: Una librería de comunicación utilizando sockets TCP. Expone una interfaz para
enviar y recibir mensajes con los jugadores.
- Shuffle: Una librería simple para mezclar una lista basada en
esta entrada de la wiki de haskell
- UI: La interfaz del juego, recibe acciones de las jugadores a través del controlador de red y las
aplica al estado del juego.

La API de la librería de comunicación es la siguiente:

`start` : Empieza a escuchar en un socket TCP e inicializa los worker threads.
`receive` : desencola del `Chan` de entrada y devuelve un par (jugador, mensaje).
`send` : Recibe el jugador y el mensaje y encola el mismo en un `Chan` para su envío.
`broadcast` : Encola un mensaje para ser enviado a todos los jugadores.
`waitForPlayers` : Bloquea la ejecución hasta que todos los jugadores estén conectados.

La API del juego es la siguiente:

`newGame deck1 deck2` : Crea el estado inicial del juego a partir de los mazos de los jugadores.

`runAction playerAction gameState` : Aplica una `PlayerAction` para obtener un nuevo estado del
juego.

`PlayerAction` : Las acciones que pueden realizar los jugadores: jugar una carta, terminar su turno, atacar con uno de sus monstruos.

```
data PlayerAction = Attack AttackSource AttackTarget
    | PlayCardFromHand Int
    | EndTurn
    deriving (Show, Eq, Read)
```

Decisiones técnicas

Librería de comunicación

Se decidió utilizar sockets TCP por la flexibilidad que otorgan.
Todos los mensajes se encolan en `Chan`s, uno de salida para cada jugador y uno de entrada.
Para cada `Chan` de salida hay un thread que lee de él y envía el mensaje al jugador correspondiente.

Para la funcionalidad de poder esperar a que todos los jugadores estén conectados se utilizó
señalización entre threads usando `MVar`.

Juego

El manejo de errores se hizo haciendo que las acciones de los jugadores retornen `Either`, en
algunos casos usando `Either` como mónada y la interfaz de usuario muestra los errores al usuario y
vuelve a pedir una acción.

Testing

No se agregó testing aún. QuickCheck es un muy
buen candidato para empezar a testear el modelo del juego sobre todo.

Mejoras futuras

Librería de comunicación

- Manejo de desconexiones/reconexiones.
- Generalización del tipo de los mensajes a cualquier cosa serializable.

Juego

- Aplicación cliente con interfaz de usuario más amigable.
- Implementación de “maná/energía” en el juego para que las acciones tengan un costo.
- El código de las acciones contiene mucha repetición, buscar una forma de no tener ramas tan
  grandes dependiendo de quien es el turno

Conclusiones

Haskell es un lenguaje muy poderoso y que conlleva pensar los programas de una forma diferente.
Para la realización del juego y librería de comunicación hizo falta mucha investigación sobre
distintas opciones para mejorar la arquitectura y calidad del código del mismo. Una opción
investigada que puede ayudar a mejorar la legibilidad del código es
Lenses.
Lenses aplica especialmente por utilizar record syntax y tener dos estructuras anidadas
(`GameState`
y `PlayerState`).