TP de Programación Funcional: Reversi

Fecha de entrega: 20 de septiembre

6 de septiembre de 2007

Índice

1. Módulo Reversi

El objetivo de este TP es construir un programa capaz de jugar al Reversi (u "Othello"). La primera parte del TP consiste en la programación de las reglas de este juego. La segunda parte requiere implementar el algoritmo *minimax*, para elegir la siguiente movida en un juego.

1.1. Reglas del juego

Se enuncian brevemente las reglas del Reversi que se considerarán en este TP.

- Juegan alternadamente dos jugadores, sobre un tablero de $n \times m$, con $n, m \geq 2$.
- Inicialmente, hay cuatro piezas en el tablero, acomodadas de acuerdo a una cierta disposición fija.

- En su turno, cada jugador debe poner en el tablero una pieza de su color.
- La pieza debe ser ubicada de forma tal que una fila de una o más piezas contiguas del color oponente quede encerrada por dos piezas del color del jugador. Observar que las "filas" de piezas pueden ser ortogonales y diagonales.
- Inmediatamente después, todas las piezas que hayan quedado encerradas se deben "dar vuelta" (cambian de color, pasando a ser del jugador que colocó la última pieza).
- Si corresponde el turno a un jugador, pero éste no puede jugar en ninguna coordenada del tablero, automáticamente se pasa el turno al oponente.
- El juego termina cuando ambos jugadores no pueden jugar en ninguna coordenada del tablero.
- Gana el jugador que tiene más piezas de su color en el tablero.

1.2. Tipos de datos

Las reglas del juego se reflejarán en el módulo `Reversi`, donde ya se encuentran definidos tipos de datos para:

- Representar el color de las piezas y diferenciar a los jugadores:

  ```
  data Color = Negro | Blanco
  ```

- Representar la dimensión de un tablero:

  ```
  type Dimension = (Int, Int)
  ```

- Representar una coordenada en el tablero:

  ```
  type Coordenada = (Int, Int)
  ```

  Si la dimensión del tablero es (dx, dy), las coordenadas válidas son de la forma (x, y) con $0 \leq x < dx, 0 \leq y < dy$.

- Representar un juego de Reversi:

  ```
  data Juego = Comenzar Dimension
             | Poner Juego Coordenada
  ```

  Los constructores tienen el siguiente significado:

  - `Comenzar d`: comenzar un juego sobre un tablero de dimensión d.
  - `Poner j c`: continuar el juego j, poniendo una pieza (del jugador al que le toque) en la coordenada c. La coordenada c debe ser válida (debe estar dentro de las dimensiones del tablero, y debe respetar las condiciones dadas por las reglas del Reversi).

- Representar un tablero:

  ```
  data Tablero = Tablero Dimension (Coordenada -> Maybe Color)
  ```

  Donde `Tablero d t` corresponde a un tablero de dimensión d que, en toda coordenada c dentro de esas dimensiones, tiene:

  - Una pieza de color k, si `t c == Just k`.
  - Nada, si `t c == Nothing`.

1.3. Funciones

Dentro del módulo `Reversi`, ya se encuentra implementada la función
`tableroInicial :: Dimension -> Tablero`
que devuelve un tablero de la dimensión dada, con las piezas ubicadas de acuerdo a la disposición inicial tradicional en el Reversi.

Ejercicio 1 Definir la función:
`enRango :: Coordenada -> Dimension -> Bool`
que determina si la coordenada está dentro de la dimensión dada.

Ejercicio 2 Definir y dar el tipo del esquema de recursión `foldJuego` asociado al tipo `Juego`.

Ejercicio 3 Usando el fold para `Juego`, definir la función:
`dimension :: Juego -> Dimension`
que dado un juego, devuelve la dimensión del tablero sobre el que se juega.

Ejercicio 4 Definir la función:

```
coordsQueInvierte :: Color -> Tablero -> Coordenada ->
                                          [Coordenada]
```

donde `coordsQueInvierte k t c` devuelve una lista con las coordenadas que deben "darse vuelta" si se juega una pieza de color k en la coordenada c de t. Pueden utilizarse funciones auxiliares, pero no recursión explícita.

No es necesario que las coordenadas devueltas por la implementación respeten algún orden particular.

Ejemplo 4.1 En la siguiente disposición del tablero, una implementación de `coordsQueInvierte` debería cumplir lo siguiente:

- La siguiente expresión:

  ```
  coordsQueInvierte Negro t (1, 3)
  ```

  debe evaluar a alguna permutación de:

  ```
  [(1, 1), (1, 2), (3, 1), (2, 2)]
  ```

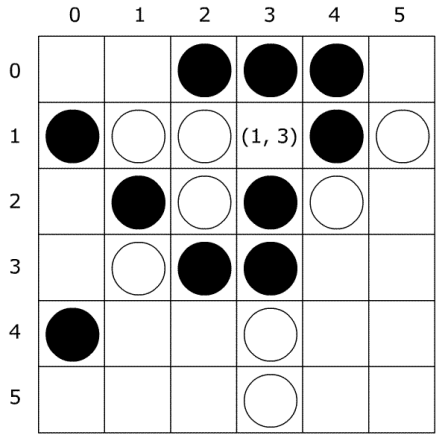

Figura 1: Tablero t de 6×6

- Mientras la siguiente expresión:

  `coordsQueInvierte Blanco t (1, 3)`

  debe evaluar a alguna permutación de:

  `[(1, 4), (2, 3), (3, 3)]`

Ejercicio 5 Definir la función:

```
puedeJugarEn :: Color -> Tablero -> Coordenada -> Bool
```

donde `puedeJugarEn k t c` determina si es posible jugar una pieza de color k en la coordenada c de t. No necesariamente la coordenada está dentro de las dimensiones del tablero.

Ejercicio 6 Definir las siguientes funciones:

- `terminoElJuego :: Juego -> Bool`
- `turno :: Juego -> Color`
- `tablero :: Juego -> Tablero`

Donde:

- `terminoElJuego j` determina si terminó el juego, es decir si ambos jugadores no pueden poner piezas en ninguna coordenada del tablero.
- `turno j` determina el color del jugador al que le toca mover, siempre que no haya terminado el juego. Recordar que los turnos se alternan, y que si un jugador no puede poner piezas, se pasa el turno automáticamente.
- `tablero j` devuelve el tablero en su situación actual.

Utilizar el fold para `Juego`. Pueden utilizarse funciones auxiliares, pero no recursión explícita.

Ejercicio 7 Definir la función:
`movidasValidas :: Juego -> [Juego]`
que, dado un juego, devuelve una lista con todos los posibles juegos a los que se puede llegar, en un paso, partiendo de la situación actual. Considerar todas las coordenadas en las que pueda jugar el color al que le toca. Considerar también el caso en el que el juego ya se encuentre terminado. No utilizar recursión explícita.

Ejercicio 8 Definir la función:
`quienGano :: Juego -> Maybe Color`
que, dado un juego terminado, determina quién es su ganador. Si la cantidad de piezas de cada color es la misma, se debe devolver `Nothing`. No utilizar recursión explícita.

Ejercicio 9 Usando el fold para `Juego`, definir la función:
`ultimaCoordJugada :: Juego -> Maybe Coordenada`
que devuelve la última coordenada en la que se jugó una pieza. Si el juego recién comienza, se debe devolver `Nothing`.

Ejercicio 10 Sin usar recursión explícita, definir la función:
`diferenciaNegrasBlancas :: Juego -> Int`
que devuelve la cantidad de piezas negras menos la cantidad de piezas blancas en la situación actual del juego.

2. Módulo Minimax

En el módulo `Minimax` se encuentra definido el siguiente tipo de datos, para árboles con cantidad no acotada de hijos.

```
data Arbol a = Nodo a [Arbol a]
```

Ejercicio 11 Definir y dar el tipo del esquema de recursión `foldArbol` asociado al tipo `Arbol`.

Ejercicio 12 Utilizando `foldArbol`, definir:
`mapArbol :: (a -> b) -> Arbol a -> Arbol b`
`mapArbol f a` debe devolver un árbol cuyos nodos resultan de aplicar f a cada nodo de a.

Ejercicio 13 Definir y dar el tipo del esquema de recursión `foldNat` sobre los naturales. Utilizar el tipo `Integer` de Haskell.

Ejercicio 14 Usando `foldNat`, definir la función:

```
podar :: Int -> Arbol a -> Arbol a
```

`podar n a` devuelve el árbol a podado a altura n. No utilizar recursión explícita.

Ejercicio 15 Definir la función:

```
arbolDeMovidas :: (a -> [a]) -> a -> Arbol a
```

`arbolDeMovidas f x` debe devolver un árbol cuya raíz es x, y su i-ésimo hijo es `arbolDeMovidas f (f x !! i)`.

Esta función puede considerarse una generalización de la función `iterate` para el tipo `Arbol`.

Si el tipo `a` representa la posición de un juego, y f devuelve todas las posiciones que pueden alcanzarse en un paso, el resultado es el árbol de todas las posiciones alcanzables desde x. Observar que el árbol devuelto es potencialmente infinito.

Ejemplo 15.1 Al evaluar la expresión:

```
podar 2 (arbolDeMovidas (\x -> [x - 1, x, x + 1]) 0)
```

se obtiene el siguiente árbol:

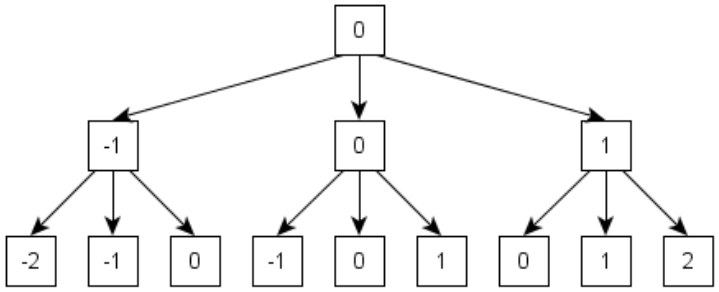

Figura 2: Resultado de la evaluación

Ejercicio 16 Sin utilizar recursión explícita, definir la función:

```
minimax :: Ord b => (a -> b) -> (a -> Bool) -> Arbol a -> a
```

Minimax es un algoritmo para determinar recursivamente la siguiente movida en un juego, asumiendo que se dispone de una función de evaluación. La función de evaluación analiza una posición del juego y estima cuán buena es dicha posición para uno de los jugadores.

Minimax se aplica sobre el árbol de posiciones de un juego, suponiendo que uno de los jugadores pretende maximizar el valor de la función de evaluación, mientras el oponente pretende minimizarla.

Seudocódigo de *minimax*:

```
minimax nodo =
```

```
    if el nodo no tiene hijos
      then evaluar nodo
      else optimizar (map minimax (hijos nodo))
  where
    optimizar = if le toca al jugador que maximiza
                  then maximize
                  else minimize
```

Si el tipo `a` representa la posición de un juego:
`minimax evaluar turnoMax arbol` debe devolver la posición elegida utilizando el algoritmo *minimax*.

- `evaluar` es la función de evaluación.
- `turnoMax` indica, dada una posición, si es el turno del jugador cuyo objetivo es maximizar el resultado.
- `arbol` es la parte del árbol de movidas que va a ser analizada por el algoritmo. La raíz del árbol corresponde a la posición actual. El resultado de `minimax` debe ser una de las posiciones en el segundo nivel del árbol.

Ejemplo 16.1

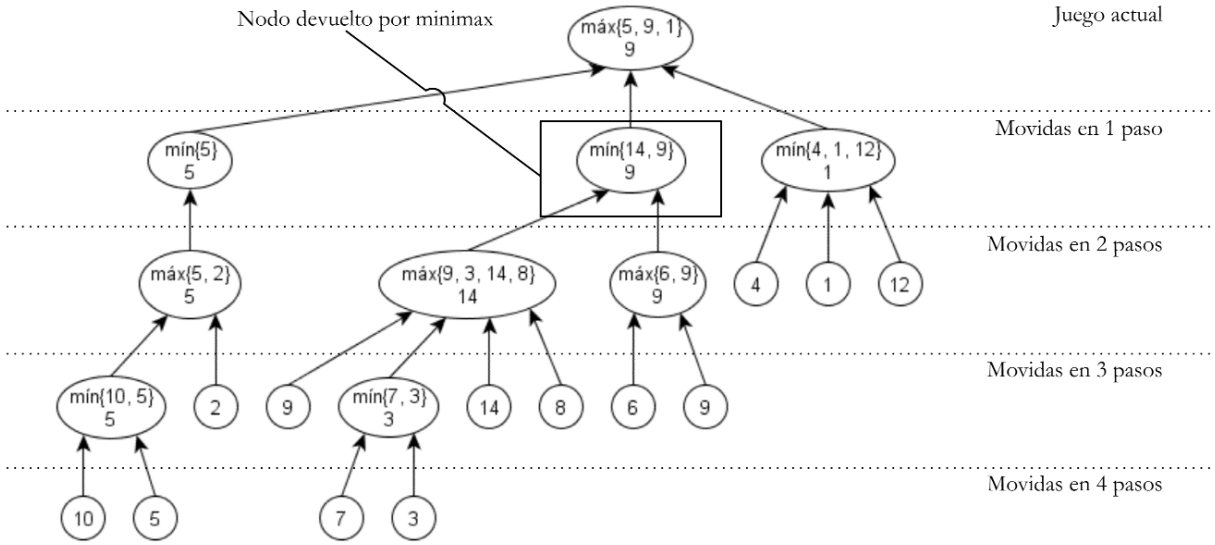

Figura 3: Ejemplo de aplicar *minimax* sobre un árbol

3. Módulo IA

Ejercicio 17 Definir la función:

```
estrategiaIA :: Juego -> Juego
```

que tome una posición del juego de Reversi y realice una movida, devolviendo la siguiente posición. Para elegir la siguiente movida, deberá aplicarse el algoritmo *minimax*. La función de evaluación será la diferencia entre la cantidad de piezas negras y piezas blancas (siendo negro el color del jugador que maximiza).

Considerar que utilizar *minimax* para explorar todo el árbol del juego de Reversi puede tomar demasiado tiempo. Antes de aplicar *minimax* se deberá podar el árbol para explorar una cantidad razonable de niveles.

Como observación más allá del objetivo del TP, tener en cuenta que la función de evaluación propuesta es demasiado *naïve*.

4. Opcional: alpha-beta pruning

El tiempo de ejecución de *minimax* puede disminuirse considerablemente realizando la optimización conocida como *alpha-beta pruning*.

La optimización consiste en mantener, a medida que se recorre el árbol, un valor α, que representa el mínimo valor que se sabe que puede conseguir el jugador que maximiza, y un valor β, que representa el máximo valor que se sabe que puede conseguir el jugador que minimiza.

Mantener estos valores permite descartar subárboles cuando se puede determinar, sin explorarlos, que no mejoran el resultado conseguido hasta el momento.

Ejercicio 18 Definir la función:

```
alfaBeta :: (Ord b, Num b) => (a -> b) -> (a -> Bool) ->
                                  b -> b -> Arbol a -> a
```

`alfaBeta evaluar turnoMax arbol alfa beta` debe devolver la posición elegida utilizando el algoritmo *minimax* con *alpha-beta pruning*.

- `evaluar` es la función de evaluación.
- `turnoMax` indica, dada una posición, si es el turno del jugador cuyo objetivo es maximizar el resultado.
- `alfa` es el valor inicial de α, que corresponde a $-\infty$. Si `b` es, por ejemplo, `Integer`, α corresponde a algún entero menor que todos los que pueda devolver la función de evaluación.
- `beta` es el valor inicial de β, que corresponde a $+\infty$. Igual que en el caso de `alfa`, β corresponde a algún valor mayor que todos los que pueda devolver la función de evaluación.
- `arbol` es la parte del árbol de movidas que va a ser analizada por el algoritmo. La raíz del árbol corresponde a la posición actual. El resultado de `alfaBeta` debe ser una de las posiciones en el segundo nivel del árbol.

Para este ejercicio se permite utilizar recursión explícita.

Links de interés:

- John Hughes, *Why functional programming matters*:
  `http://www.math.chalmers.se/~rjmh/Papers/whyfp.html`
- *Alpha-beta pruning* en Wikipedia:
  `http://en.wikipedia.org/wiki/Alpha-beta_pruning`
- Ejemplo de ejecución de *alpha-beta pruning*:
  `http://www.cs.swarthmore.edu/~meeden/Minimax/TypicalCase.html`

5. Pautas de entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función debe contar con un comentario donde se explique su funcionamiento y cada función asociada a los ejercicios deben contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Asimismo, se debe enviar un mail conteniendo el código fuente Haskell a la dirección `plp-docentes@dc.uba.ar`. Dicho mail debe cumplir con el siguiente formato:

- El subject debe ser "[PLP;TP-PF]" seguido inmediatamente del nombre del grupo sin acentos.
- Solamente el código Haskell ejecutable debe acompañar el mail y lo debe hacer en forma de **attachment**.

Importante: Se admitirá una única submisión, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.