Taller de programación funcional

Verano 2019

Fecha límite de entrega: 7 de febrero de 2019 a las 16:00

1. Introducción

En este taller trabajaremos con máquinas de estados no determinísticas (MEN), que modelan el funcionamiento de un sistema en base a estados y símbolos de entrada que disparan transiciones entre dichos estados. De esta manera, consideraremos que el sistema se encuentra en algún estado y al leer un símbolo de entrada, cambia a algún otro estado. Dado que la máquina no es determinística, puede ocurrir que para un mismo estado y un mismo símbolo existan dos o más estados posibles a los que transitar.

En general, estos autómatas se modelan utilizando un conjunto de estados, un alfabeto (que se suele denominar Σ) y una función de transición (que solemos llamar δ) que, dado un estado `q` y un símbolo `s`, devuelve el conjunto de estados a los que puede transitar el sistema desde el estado `q` al leer el símbolo `s`. Si el conjunto es vacío no existen transiciones posibles para esa combinación.

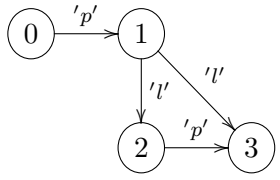
En este taller modelaremos estos autómatas mediante el siguiente tipo:

```
data MEN a b = AM {sigma :: [b], delta :: (a -> b -> [a])}
```

Si el sistema representado por `m :: MEN a b` se encuentra en un estado `q :: a` , después de recibir una entrada `s :: b` tal que `s` $\in$ `sigma m`, se encontrará en alguno de los estados de la lista `delta m q s`[1]. En todo el taller, podemos desentendernos de que la función de transición se aplique sobre símbolos que no pertenecen al alfabeto.

Un ejemplo de máquina de estados:

$mPLP$ =




```
mPLP :: MEN Int Char
mPLP = AM ['l','p'] tran
   where tran q s | q == 0 && s == 'p'= [1]
                  | q == 1 && s == 'l'= [2, 3]
                  | q == 2 && s == 'p'= [3]
                  | otherwise = []
```

Con esta forma de definir el tipo `MEN`, automáticamente se definen las funciones

- `sigma :: MEN a b -> [b]` que dada una máquina de estados, devuelve su alfabeto, por ejemplo: `sigma mPLP = ['l', 'p']`;
- `delta :: MEN a b -> (a -> b -> [a])` que dada una máquina de estados, devuelve su función de transición.

[1] Una manera alternativa de interpretarlo es pensar que después de transicionar el autómata se encuentra en *todos* los estados indicados por la función de transición.

El taller contiene dos archivos:

- `Util.hs`: para poder trabajar con autómatas (tipo de dato, funciones, y definición de $mPLP$). Este archivo no podrá ser modificado.
- `Main.hs`: funciones para implementar.

2. Ejercicios

Ejercicio 1

(a) Definir `vacio :: [b] -> MEN a b` que dado un alfabeto devuelve una máquina sin transiciones.

(b) Definir `agregarTransicion :: (Eq a, Eq b) => MEN a b -> a -> b -> a -> MEN a b` que, dado un autómata `m`, un estado q_0, un símbolo del alfabeto `s` y otro estado q_1, devuelve un autómata igual al original, pero con una transición agregada desde q_0 a q_1, por el símbolo `s`. Se puede suponer que la transición no se encuentra previamente definida y que si q_0 no pertenece a los estados de `m`, entonces `delta m` q_0 `s = []`.

(c) Definir `aislarEstado :: Eq b => MEN a b -> b -> MEN a b` que, dado un autómata y un estado, devuelve un autómata igual al original, pero con el estado aislado, es decir, sin transiciones entrantes ni salientes.

Ejercicio 2

Se dice que un estado es una *trampa* cuando no tiene transiciones que lo lleven a otros estados más que a sí mismo.

(a) Definir `trampaUniversal :: a -> [b] -> MEN a b` que, dado un estado y un alfabeto, devuelve un autómata que, para cualquier estado y cualquier símbolo, transiciona al estado recibido como parámetro.

Se dice que un autómata es *completo* cuando todos los estados tienen transiciones por todos los símbolos, es decir, la función de transición es no nula para todas las combinaciones de estados y símbolos del alfabeto.

(b) Definir `completo :: Eq a => MEN a b -> a -> MEN a b` que, dado un autómata y un estado, completa el autómata con transiciones hacia este estado. Es decir, las transiciones nulas en el primer autómata ahora dirigen al estado pasado como parámetro.

Ejercicio 3

Si bien la función de transición está definida para leer de a un símbolo, se puede definir una noción de transición para palabras o cadenas de símbolos del alfabeto. Simplemente, el autómata comienza en un estado y va leyendo, de izquierda a derecha, los símbolos de la cadena, de a uno por vez, comenzando en cada caso por cualquiera de los estados en los que terminó tras leer el símbolo anterior.

Definir la función `consumir :: Eq a => MEN a b -> a-> [b] -> [a]` que, dado un autómata, un estado y una cadena de símbolos, devuelve los estados en los que se puede encontrar el autómata después de haber consumido la cadena.

En el autómata de ejemplo, `consumir mPLP 0 "pl"` $\rightsquigarrow$ `[2,3]` y `consumir mPLP 0 "plp"` $\rightsquigarrow$ `[3]`.

La lista devuelta puede estar en cualquier orden pero no debe contener repetidos. Prestar atención al esquema de recursión elegido.

Útil: `union :: Eq a => [a] -> [a] -> [a]`.

Ejercicio 4

Definimos un *lenguaje* como un conjunto de palabras (o cadenas) de símbolos de un alfabeto. Luego, podemos hablar del lenguaje de un autómata como el conjunto de las palabras de símbolos de su alfabeto que son *aceptadas* por el autómata. Para esto, definimos algunos estados del autómata como estados de aceptación y decimos que una palabra es aceptada si es posible que el autómata, después de leerla, se encuentre en alguno de estos estados.

(a) Definir `acepta :: Eq a => MEN a b -> a -> [b] -> [a] -> Bool` que, dado un autómata, un estado de partida, una cadena de símbolos del alfabeto y una lista de estados de aceptación, decide si el autómata acepta la palabra en esas condiciones.

Por ejemplo: `acepta mPLP 0 "pl"[3]` $\rightsquigarrow$ `True` (pues el autómata puede alcanzar el estado 3 leyendo *pl*) y también `acepta mPLP 0 "plp"[3]` $\rightsquigarrow$ `True`.

(b) Definir `lenguaje :: Eq a => MEN a b -> a -> [a] -> [[b]]` que, dado un autómata, un estado de partida y una lista de estados de aceptación, devuelve el lenguaje del autómata en esas condiciones de aceptación. Se puede asumir que el lenguaje es infinito[2].

Sugerencia: definir la función `kleene :: [b] -> [[b]]` que devuelve la *clausura de Kleene* de un conjunto de símbolos. Esta operación, usualmente denotada como `*`, representa todas las palabras finitas que se pueden formar con los símbolos del conjunto. En definitiva:

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$$

donde Σ^i es el conjunto de todas las palabras de longitud exactamente i formadas con símbolos de Σ.

Ejercicio 5

Se definen las *trazas* de una máquina de estados a partir de un estado como todas las cadenas de símbolos que la pueden llevar desde dicho estado a algún otro mediante transiciones no nulas.

Definir `trazas :: MEN a b -> a -> [[b]]` que, dado un autómata y un estado, devuelve la lista con todas las trazas posibles del autómata a partir del estado dado.

[2] Esto quiere decir que, en el caso de que el lenguaje sea finito, no es necesario determinarlo ni producir una lista finita.

Notar que si existe al menos un ciclo en el autómata, la lista resultante es infinita, y que la función de transición siempre devuelve listas finitas. Una misma traza puede aparecer repetida si representa distintos "caminos" en el autómata.

En el autómata de ejemplo, `trazas mPLP 0` $\rightsquigarrow$ `["p", "pl", "pl", "plp"]`.

Útil: `takeWhile :: (a -> Bool) -> [a] -> [a]`.

Ejercicio 6

Decimos que un estado es *alcanzable* desde otro, si existe una cadena que lleva al autómata de un estado al otro.

Definir la función `alcanzables :: Eq b => MEN a b -> a -> [a]` que, dado un autómata y un estado, devuelve una lista con todos los estados alcanzables desde el recibido como parámetro.

Para eso, definir primero en orden las siguientes funciones:

- `deltaS :: Eq a => MEN a b -> [a] -> [a]` que, dado un autómata `m` y una lista `ls`, devuelve la lista con todos los estados alcanzables con cadenas de longitud 1 a partir de cualquiera de los estados de `ls`. En algún sentido, es la generalización de la función de transición para conjuntos de estados.
- El esquema de recursión `fixWhile :: (a -> a) -> (a -> Bool) -> a -> a` que, dada una función, un predicado y un elemento, devuelve la función aplicada sobre el elemento tantas veces como sea necesario para que se verifique el predicado. O sea, `fixWhile f p x` $\rightsquigarrow$ `f (f (...f (f x)))`, donde `f` está aplicada la menor cantidad de veces tal que `p $ f (f (...f (f x)))` $\rightsquigarrow$ `True`. Para esta función está permitido el uso de recursión explícita.
- Finalmente, `fixWhileF`, que es `fixWhile` pero definido sin utilizar recursión explícita, en términos de `fix :: (a -> a) -> a` cuya definición se encuentra en Util.hs.

3. Pautas de evaluación y entrega

Los objetivos a evaluar con este taller son:

- Corrección.
- Declaratividad.
- Prolijidad. Se deberá evitar repetir código innecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).
- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión. Es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas; por ejemplo, deberán usar `zip`, `map`, `filter`, `take`, `takeWhile`, `dropWhile`, `foldr`, `foldl`, listas por comprensión, etc., cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del Taller es aprender a aprovechar las características enumeradas en el ítem anterior. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `List`, `Maybe`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`.

Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Recomendamos que aprovechen las clases destinadas al taller para intentar terminar los ejercicios. Aquellos grupos que no cuenten con las correcciones al finalizar la clase del 6 de febrero, deberán entregar el código antes de la fecha límite de entrega utilizando el Campus Virtual.

Para hacerlo, *uno* de lxs dos integrantes del grupo deberá ingresar a la tarea "Taller de programación funcional" en la sección "Talleres" de la página del Campus. Allí deberá subir un archivo `.zip` o `.tar.gz` que contenga el código Haskell correspondiente. En el comentario que acompaña a la entrega, deberá especificar nombre y apellido de su compañero de grupo.

Importante:

- El código entregado **debe** incluir tests que permitan probar las funciones definidas.
- El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código).
- El sistema dejará de aceptar envíos pasada la fecha límite de entrega, y no se harán excepciones. Por favor, planifiquen el trabajo para llegar a tiempo con la entrega.

Para instalar HUnit usar: `> cabal install hunit`

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en: http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos parciales, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquellos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.