Teoría de los Lenguajes de Programación
Práctica curso 2018-2019

Enunciado

Fernando López Ostenero y Ana García Serrano

Índice

1. Introducción: Transformar una cadena en otra.

El problema que vamos a tratar en esta práctica consiste en **transformar una cadena de caracteres en otra aplicando transiciones carácter a carácter**. Queremos encontrar todas las posibles formas de transformar la cadena origen en la cadena destino aplicando el menor número de transiciones posibles.

Las transiciones que vamos a considerar son:

1. Se borra un carácter de la cadena origen.
2. Se inserta un carácter de la cadena destino en la cadena origen.
3. Se transforma un carácter de la cadena origen en otro de la cadena destino.

La práctica vamos a resolverla en dos partes. En la primera parte vamos a construir una tabla bidimensional mediante Programación Dinámica que nos dirá cuál es el número mínimo de transiciones que se necesitan para transformar la cadena origen en la cadena destino.

Una vez dispongamos de esa tabla, en la segunda parte aplicaremos el esquema de Backtracking para explorar la tabla y así obtener todas las posibles secuencias (de tamaño mínimo) de transiciones que se pueden aplicar para realizar la transformación.

Los esquemas algorítmicos de Programación Dinámica y Backtracking son materia de estudio de la asignatura “Programación y Estructuras de Datos Avanzadas”, que no es un prerrequisito de esta. Por ello, en este enunciado se explica su funcionamiento y se presentan ejemplos de los mismos aplicados a esta práctica.

2. Enunciado de la práctica

La práctica consiste en elaborar un programa en Haskell que reciba dos cadenas de caracteres (en adelante origen y destino) y devuelva todas las posibles secuencias minimales de transiciones que transforman la cadena origen en la cadena destino. Diremos que una secuencia es minimal si no existe una secuencia con menor número de transiciones que pueda transformar origen en destino.

Ejemplo 1: queremos transformar la cadena “casa” en la cadena “casco”. Una posible ejecución de la práctica sería:

```
*TLP2019> mostrarSoluciones (obtenerTransiciones "casa" "casco")
Cambiar 'a' en la posicion 4 por 'c'
Insertar 'o' en la posicion 5

Insertar 'c' en la posicion 4
Cambiar 'a' en la posicion 5 por 'o'
```

es decir, las secuencias minimales tienen todas tamaño 2 y se han encontrado dos diferentes secuencias que se ilustran a continuación mostrando cómo “casa” se transforma en “casco”:

- casa → cas**c** → casc**o**
- casa → cas**c**a → casc**o**

Ejemplo 2: si ahora queremos transformar “casa” en “arbol” obtendremos 11 posibles secuencias de 5 transiciones, las cuales ilustramos a continuación:

- casa → asa → a**r**a → ar**b** → arb**o** → arbo**l**
- casa → asa → a**r**a → ar**b**a → arb**o** → arbo**l**
- casa → asa → a**r**sa → ar**b**a → arb**o** → arbo**l**
- casa → **a**asa → a**r**sa → ar**b**a → arb**o** → arbo**l**
- casa → asa → a**r**a → ar**b**a → arb**o**a → arbo**l**
- casa → asa → a**r**sa → ar**b**a → arb**o**a → arbo**l**
- casa → **a**asa → a**r**sa → ar**b**a → arb**o**a → arbo**l**
- casa → asa → a**r**sa → ar**b**sa → arb**o**a → arbo**l**
- casa → **a**asa → a**r**sa → ar**b**sa → arb**o**a → arbo**l**
- casa → **a**asa → a**r**asa → ar**b**sa → arb**o**a → arbo**l**
- casa → **a**casa → a**r**asa → ar**b**sa → arb**o**a → arbo**l**

2.1 Tipos de datos (I)

En este apartado vamos a introducir el tipo de datos que vamos a utilizar en la primera parte de la práctica. Está definido en el fichero `CodeDataTypesTLP2019.hs` que proporciona el Equipo Docente.

Tabla

Necesitamos manejar una tabla bidimensional, lo cual se realiza fácilmente mediante arrays bidimensionales. Pero como en Haskell no tenemos arrays, vamos a simularlos utilizando listas. Al ser una tabla bidimensional, tendremos que usar una lista de listas, por lo que tendremos los tipos:

```
type Fila = [Int]
type Tabla = [Fila]
```

2.2 Creación de la tabla: funciones a implementar

Para crear la tabla vamos a seguir el esquema algorítmico de Programación Dinámica. Este esquema parte de la definición de una recurrencia múltiple (recordemos que es aquella en la que por cada llamada recursiva se generan múltiples llamadas recursivas) y la transforma en una tabla, de forma que se elimina la necesidad de hacer un algoritmo recursivo con más de una llamada recursiva, lo cual supone un gran ahorro de tiempo (a costa de un mayor uso de memoria).

En nuestro caso, la idea para la recurrencia es la siguiente: queremos transformar la cadena origen (de tamaño n) en la cadena destino (de tamaño m) y vamos a considerar la función NMT(i,j) como el número mínimo de transiciones para transformar los primeros i caracteres de la cadena origen en los primeros j caracteres de la cadena destino. Entonces podemos afirmar que:

NMT(0,j) = j	pues hace falta insertar los j primeros caracteres de la cadena destino ya que la cadena origen es vacía
NMT(i,0) = i	pues hace falta borrar los i primeros caracteres de la cadena origen ya que la cadena destino es vacía
NMT(i,j) = 1 + NMT(i-1,j)	si borramos el carácter i-ésimo de la cadena origen
NMT(i,j) = 1 + NMT(i,j-1)	si insertamos el j-ésimo carácter de la cadena destino en la posición i de la cadena origen
NMT(i,j) = 1 + NMT(i-1,j-1)	si cambiamos el i-ésimo carácter de la cadena origen por el j-ésimo carácter de la cadena destino porque eran diferentes
NMT(i,j) = NMT(i-1,j-1)	si el i-ésimo carácter de la cadena origen es igual que el j-ésimo carácter de la cadena destino

Dado que lo que queremos es obtener el mínimo (de ahí la M del nombre de la función), podríamos implementar la función NMT en Haskell como sigue:

```
nmt origen destino = nmtAux (reverse origen) (reverse destino)
  where nmtAux [] destino = length destino
        nmtAux origen [] = length origen
        nmtAux (o:os) (d:ds) = min3 b i c
          where b = 1 + (nmtAux os (d:ds))
                i = 1 + (nmtAux (o:os) ds)
                c
                  | o == d = nmtAux os ds
                  | otherwise = 1 + (nmtAux os ds)

min3 a b c
  | (a <= b) && (a <= c) = a
  | (b <= a) && (b <= c) = b
  | otherwise            = c
```

Si nos fijamos, esta implementación (disponible a modo de ejemplo en el fichero `nmt.hs`) está usando directamente la recurrencia y realiza tres llamadas recursivas por cada ejecución, lo que supone un tiempo muy grande. Para cadenas pequeñas, puede encontrar el número mínimo de transiciones en un tiempo razonable, pero os animamos a probar qué pasa con:

```
> nmt "esternocleidomastoideo" "semicircunferencia"
```

y eso sólo para averiguar el tamaño de las secuencias minimales (que, por cierto, es 19). Ahora imaginemos cuánto tiempo tardaríamos en encontrar las 180 posibles secuencias que realizan la transformación.

La idea de la Programación Dinámica consiste en construir un array NMT[i,j] (nótese el cambio de paréntesis por corchetes) en la que NMT[i,j] = NMT(i,j). Vamos a ver el aspecto de la tabla para la transformación “`casa`” → “`arbol`”:

	“”	a	r	b	o	l
“”	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	3	4	5
s	3	2	2	3	4	5
a	4	3	3	3	4	5

La creación de esta tabla se hace de arriba hacia abajo y de izquierda a derecha, comprobando en cada momento cuál es la transición que minimiza el tamaño de la secuencia de transiciones.

Así, partimos de una primera fila `[0,1,2,3,4,5]` y en base a ella creamos la siguiente mediante el proceso descrito a continuación:

1. El primer elemento será el primero de la fila anterior más uno.
2. El resto de elementos deberá tener en cuenta los valores de los elementos situados sobre él y a su izquierda. Por ejemplo el 2 marcado en rojo se calcula en base a los valores de las casillas marcadas en verde:

	“”	a	r	b	o	l
“”	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	3	4	5
s	3	2	2	3	4	5
a	4	3	3	3	4	5

Ya que se calcula el mínimo entre el borrado 1+1 (casilla `[‘a’,‘a’]`), la inserción 3+1 (casilla `[‘s’,“”]`) y el cambio 2+1 (casilla `[‘a’,“”]`). Dado que el carácter `‘s’` es diferente del carácter `‘a’`, cambiar uno por otro añade una transición y eso explica que en el caso de la transición “Cambiar” se sume uno. Un ejemplo de transición “Cambiar” que no suma uno lo tenemos en:

	“”	a	r	b	o	l
“”	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	3	4	5
s	3	2	2	3	4	5
a	4	3	3	3	4	5

Pues el `1` de la casilla en `[‘a’,‘a’]` viene del `1` en la casilla `[‘c’,“”]` porque `‘a’ == ‘a’` y cambiar un carácter por el otro es igual que no hacer nada.

Así, habrá que programar una función:

```
creaTabla :: String -> String -> Tabla
```

tal que reciba la cadena origen y la cadena destino y construya la tabla mediante el procedimiento anteriormente descrito. Para ello podemos apoyarnos en una función auxiliar:

```
creaSiguienteFila :: Char -> String -> Fila -> Fila
```

que construye la siguiente fila recibiendo como parámetros el carácter de la fila, la cadena destino y la fila superior. Por ejemplo, la fila de la `‘c’` se construiría con la llamada:

```
> creaSiguienteFila 'c' "arbol" [0,1,2,3,4,5]
[1,1,2,3,4,5]
```

y así sucesivamente, ya que la siguiente fila se construiría con la llamada:

```
> creaSiguienteFila 'a' "arbol" [1,1,2,3,4,5]
[2,1,2,3,4,5]
```

Nótese que este procedimiento tiene un coste muchísimo menor que el de la función NMT que describimos con anterioridad y además nos da la información necesaria para poder encontrar todas las secuencias minimales de transformaciones (algo que haremos en la segunda parte de la práctica),

cuyo tamaño coincide con el último valor calculado de la tabla (que es 5 en el ejemplo que hemos visto, como ya habíamos adelantado).

Para facilitar el acceso a los datos en la segunda parte de la práctica, las filas de la tabla deberán añadirse como si se tratase de una pila. Así, el aspecto de la tabla que hemos visto sería:

```
> creaTabla "casa" "arbol"
[[4,3,3,3,4,5],[3,2,2,3,4,5],[2,1,2,3,4,5],[1,1,2,3,4,5],[0,1,2,3,4,5]]
```

2.3 Tipos de datos (II)

En este apartado vamos a introducir los tipos de datos sobre los que vamos a trabajar para la segunda parte de la práctica. Todos ellos están definidos en el fichero `CodeDataTypesTLP2019.hs` que proporciona el Equipo Docente.

Transición

Como indicamos en la introducción, tenemos tres tipos de transiciones: borrar, insertar y cambiar. Las dos primeras nos indican qué carácter se borra (inserta) de (en) la cadena origen y en qué posición, mientras que la última nos indica, además, cuál es el carácter de la cadena destino por el que se cambia el carácter elegido de la cadena origen.

Así pues, construimos un tipo de datos `Transicion` de la siguiente forma:

```
data Transicion = Borrar Char Int| Insertar Char Int | Cambiar Char Int Char
```

ya que nos permite almacenar sin problemas toda la información necesaria para identificar correctamente la transición aplicada.

Además, para facilitar la lectura de las transiciones, instanciaremos este tipo de datos dentro de la clase `Show` (que agrupa a todos los tipos de datos que son visualizables). Para ello basta con definir la función `show` (en otros casos la instanciación de un tipo en una clase puede ser más compleja que simplemente definir una función, pero eso queda fuera de los propósitos de la asignatura):

```
instance Show Transicion where
  show (Borrar c n) = "Borrar "++show(c)++" en la posicion "++show(n)++"\n"
  show (Insertar c n) = "Insertar "++show(c)++" en la posicion "++show(n)++"\n"
  show (Cambiar a n b) = "Cambiar "++show(a)++" en la posicion "++show(n)++" por
                         "++show(b)++"\n"
```

Los caracteres ‘`\n`’ representan saltos de línea, que serán correctamente interpretados por la función que se encarga de mostrar todas las soluciones.

Solución

Una solución será una secuencia minimal de transiciones, así que la podemos representar directamente como una lista:

```
type Solucion = [Transicion]
```

Cuando apliquemos el esquema de Backtracking, obtendremos todas las secuencias minimales, es decir, todas las soluciones. Algo que podemos representar mediante una lista de soluciones. Para poder visualizar la lista de soluciones se entrega la función:

```
mostrarSoluciones :: [Solucion] -> IO ()
mostrarSoluciones ls = putStr (mS ls)
  where mS [] = ""
        mS (s:ss) = (concat (map show s))++"\n"++(mS ss)
```

donde `IO ()` es la mónada input/output (nuevamente queda fuera del ámbito de la asignatura una explicación más profunda de su funcionamiento) que es el tipo devuelto por la función `putStr`, encargada de interpretar los saltos de línea (caracteres `'/n'`) que hemos ido introduciendo en la cadena de salida.

Nodo

El esquema de Backtracking (también conocido como "vuelta atrás") consiste en explorar un grafo implícito (es decir, que se va construyendo a medida que se explora) de soluciones en el que cada nodo intermedio contiene una proto-solución aún no completa y cada uno de sus hijos completa un poco más esa proto-solución hasta llegar a los nodos hoja que ya contendrán una solución completa.

En nuestro problema, vamos a definir los nodos como sigue:

```
data Nodo = Nodo { orig :: String , i :: Int , dest :: String , j :: Int ,
                   tabla :: Tabla , solucion :: Solucion }
  deriving Show
```

Es decir, en cada nodo almacenaremos la cadena origen que nos queda por procesar, su tamaño, la cadena destino que nos queda por procesar, su tamaño, la tabla (no entera) que hemos calculado en la primera parte de la práctica y la proto-solución que tenemos calculada hasta ese momento.

Recordemos que gracias a que estamos nombrando cada uno de los componentes del tipo `Nodo`, Haskell nos crea automáticamente las funciones:

```
orig     :: Nodo -> String
i        :: Nodo -> Int
dest     :: Nodo -> String
j        :: Nodo -> Int
tabla    :: Nodo -> Tabla
solucion :: Nodo -> Solucion
```

para acceder directamente a cada componente.

2.4 Identificación de soluciones: funciones a implementar

En esta segunda parte de la práctica vamos a identificar todas las posibles soluciones, gracias al Backtracking, consistentes en secuencias minimales de transiciones. Por construcción todas tendrán el mismo tamaño (recordemos que coincide con el último valor calculado para la tabla).

Pero antes de explicar cómo funciona el esquema de Backtracking y qué tendremos que programar para aplicarlo a nuestro problema, vamos a ver cómo podemos identificar una posible solución. Una vez entendido este paso, aplicando Backtracking será sencillo obtener todas las soluciones.

Si la construcción de la tabla se hizo de arriba hacia abajo y de izquierda a derecha, la identificación de soluciones se hará al revés: de abajo hacia arriba (de ahí que la tabla se deba construir como se indicó anteriormente) y de derecha a izquierda.

La idea, nuevamente, va a ser acudir a nuestra (muy ineficiente) función nmt. En ella vemos que para calcular el valor NMT(i,j) necesitamos calcular los valores de NMT(i-1,j), NMT(i,j-1) y NMT(i-1,j-1). Pero ahora, gracias a la tabla que hemos construido en la primera parte de la práctica ya tenemos los cuatro valores, lo que nos permite invertir el proceso, siempre que supongamos que i,j > 0:

- Si NMT(i,j) == NMT(i-1,j) + 1 significa que la última transición aplicada ha sido un borrado.
- Si NMT(i,j) == NMT(i,j-1) + 1 significa que la última transición aplicada ha sido una inserción.
- Si NMT(i,j) == NMT(i-1,j-1) + 1 y el carácter i-ésimo de la cadena origen es distinto del carácter j-ésimo de la cadena destino, significa que la última transición aplicada ha sido una conversión.
- Si NMT(i,j) == NMT(i-1,j-1) y el carácter i-ésimo de la cadena origen coincide con el carácter j-ésimo de la cadena destino, significa que en ese caso no se ha aplicado ninguna transición.

En el caso de que i sea 0, sólo podríamos realizar la segunda comprobación y si j fuera 0 sólo podríamos realizar la primera.

Y ahora cambiemos (mentalmente) los paréntesis anteriores por corchetes. ¿Qué tenemos? ¡Consultas a la tabla construida! Si partimos de la casilla NMT[i,j] y comparamos su valor con el de las casillas NMT[i-1,j], NMT[i,j-1] y NMT[i-1,j-1]. La primera de las comparaciones anteriores que sea verdadera nos dará la última transición aplicada (que meteremos en una pila) para llegar a la casilla NMT[i,j]. Nos trasladaremos a la casilla de la que partía esa transición y repetiremos el proceso hasta llegar a la casilla NMT[0,0], momento en el que tendremos en la pila una secuencia de transiciones que nos transforma la cadena origen en la cadena destino.

Para encontrar **todas** las soluciones posibles, debemos emplear un Backtracking en el que se exploren todas las posibles transiciones que hubieran podido dar en el paso que estamos examinando. Así, en la tabla que hemos considerado como ejemplo, tendríamos que el 5 situado en la última casilla (desde la que comenzaremos la exploración inicial) podría venir del 4 de la casilla a su izquierda, lo que supondría una transición "`Insertar 'l' en la posición 5`", pero también podría venir del 4 de la casilla superior izquierda, lo que supondría una transición "`Cambiar 'a' en la posición 5 por 'l'`".

Eso hace que el primer nodo que exploramos en el Backtracking tenga dos posibles hijos. En cada uno de ellos se modificarán los componentes adecuados (definidos en el tipo `Nodo` comentado en el apartado anterior).

Así, el esquema de Backtracking se aplicará recursivamente sobre todos los hijos de un nodo y luego volverá hacia atrás al programa que lo llamó. Aplicando este esquema a un nodo inicial que defina nuestro problema nos devolverá **todas** las soluciones del mismo.

De esto se encarga la función bt que se encuentra en el fichero `CodeDataTypesTLP2019.hs`:

```
bt :: (a -> Bool) -> (a -> [a]) -> a -> [a]
bt esSol compleciones nodo
  | esSol nodo = [nodo]
  | otherwise = concat (map (bt esSol compleciones) (compleciones nodo))
```

Esta función recibe tres parámetros, que se corresponden con tres funciones que hay que programar para resolver esta parte:

- **Función `esSol`**: que recibe un nodo (nótese que este nodo es de tipo genérico, es decir: la función es válida para cualquier problema de Backtracking) y nos dice si el nodo contiene una solución completa o no. En nuestro problema, el nodo debería estar situado en la posición `[0,0]` de la tabla.
- **Función `compleciones`**: que recibe un nodo que no es solución y nos devuelve la lista de nodos que lo completan hacia una solución. Aunque en general no se cumple, en nuestro problema todo nodo que no contiene una solución es completable. Es decir: ante un nodo que no tiene una solución completa, la función compleciones siempre debe devolver una lista no vacía.
- **`Nodo`**: que será el nodo a partir del cual se va a realizar la exploración. Por lo tanto, se deberá programar una función que genere el nodo inicial que defina nuestro problema al completo.

Si las funciones anteriores son correctas, una llamada a `bt` con el nodo inicial nos devolverá una lista con todos los nodos solución. Por ello, a estas tres funciones hay que añadir una función principal:

```
obtenerTransiciones :: String -> String -> [Solucion]
```

que será la que reciba las cadenas origen y destino, construya el nodo inicial y llame a la función `bt` para obtener toda la lista de nodos solución, de la cual deberá extraer la lista de soluciones.

Sólo queda ver qué información se debe almacenar en cada uno de los componentes de los nodos, la cual deberá modificar adecuadamente la función `compleciones` al crear los hijos de cada nodo:

- `orig`: aquí almacenaremos la cadena origen que nos queda por procesar. Al igual que en la función `nmt`, la cadena estará invertida. Es decir, si la cadena origen es “`casa`”, el valor inicial de este componente sería “`asac`”.
- `i`: para evitar repetir cálculos, aquí almacenaremos el tamaño de la cadena origen que nos queda por procesar.
- `dest`: aquí almacenaremos la cadena destino que nos queda por procesar, también invertida como en el caso de `orig`.
- `j`: almacenará el tamaño de la cadena destino que nos queda por procesar.
- `tabla`: en este componente almacenaremos la tabla que hemos construido en la primera parte, pero sólo desde la posición `[0,0]` hasta la posición `[i,j]` y además con cada fila almacenada de forma invertida. ¿Por qué? Porque de esta forma podremos acceder en tiempo constante a los valores de las tres (como máximo) casillas que han podido influir en el valor de la casilla `[i,j]`.
- `solucion`: aquí iremos almacenando las transiciones que vayamos identificando. Este componente se tratará como una pila, ya que las transiciones se identifican en el orden inverso al que se aplican.

Vamos a ver cuál sería el aspecto del nodo inicial para convertir “`casa`” en “`arbol`”:

```
> nodoInicial "casa" "arbol"
Nodo {orig = "asac", i = 4, dest = "lobra", j = 5, tabla = [[5,4,3,3,3,4],
[5,4,3,2,2,3],[5,4,3,2,1,2],[5,4,3,2,1,1],[5,4,3,2,1,0]], solucion = []}
```

Aquí queda justificada la decisión de añadir las filas a la tabla como si fuera una pila, ya que con eso nos ahorramos el proceso de invertir la lista. Sin embargo, nótese que cada una de las filas de la tabla sí que está invertida con respecto a como se construyó en la primera parte.

3. Cuestiones sobre la práctica

La respuesta a estas preguntas es optativa. Sin embargo, si el estudiante no responde a estas preguntas, la calificación de la práctica sólo podrá llegar a 6 puntos sobre 10.

1. (*1'5 puntos*). Supongamos una implementación de la práctica en un lenguaje no declarativo (como Java, Pascal, C…). Comente qué ventajas y qué desventajas tendría frente a la implementación en Haskell. Relacione estas ventajas desde el punto de vista de la eficiencia con respecto a la programación y a la ejecución. ¿Cuál sería el principal punto a favor de la implementación en los lenguajes no declarativos? ¿Y el de la implementación en Haskell?

2. (*1'5 puntos*). Indique, con sus palabras, qué permite gestionar el predicado predefinido no lógico, corte (!), en **Prolog**. ¿Cómo se realizaría este efecto en **Java**? Justifique su respuesta.

3. (*1 punto*). Para los tipos de datos del problema definidos en **Haskell** (en el fichero `CodeDataTypesTLP2019.hs`), indique qué constructores de tipos se han utilizado en cada caso.

4. Documentación a entregar

Cada estudiante deberá entregar la siguiente documentación a su tutor de prácticas:

- Código fuente en **Haskell** que resuelva el problema planteado. Para ello se deberá entregar el fichero `TLP2019.hs`, con las funciones descritas en este enunciado, así como todas las funciones auxiliares que sean necesarias.
- Una memoria con:
    - Una pequeña descripción de las funciones programadas.
    - Las respuestas a las cuestiones sobre la práctica.