Trabajo Práctico: Sistema de Control de Versiones

Taller de Álgebra I
Departamento de Computación
FCEyN - UBA

Primer cuatrimestre de 2016

1. Introducción

En este trabajo práctico nos proponemos ejercitar los conceptos vistos en las clases del Taller de Álgebra I, en particular, la construcción e implementación en Haskell de funciones recursivas y tipos de datos. Para ello se pide implementar un Sistema de Control de Versiones[1] en Haskell, basándonos en el algoritmo de distancia de Levenshtein[5].

2. Enunciado

Un **Sistema de Control de Versiones**[1] (a partir de ahora SCV) es un programa que permite manejar los cambios en el contenido de uno o varios archivos a lo largo del tiempo. Los SCV realizan un seguimiento de las modificaciones que hacemos, y en caso que nos equivoquemos, es posible volver atrás y comparar el código actual con versiones anteriores para ayudar a arreglar el error. Algunos ejemplos populares son Git[2], Subversion[4] y Mercurial[3].

Lo que tienen en común todos estos SCV es que utilizan un algoritmo que *dados dos* String $str1$ *y* $str2$*, devuelve las modificaciones necesarias, tales que si las aplicamos a* $str1$*, obtenemos como resultado* $str2$.

Cada Modificación puede ser:

- *Insertar* un caracter en alguna posición del String.
- *Borrar* el caracter en alguna posición del String.
- *Substituir* el caracter en alguna posición del String por otro.

Uno de los algoritmos posibles, y el que vamos a usar para este trabajo práctico, es una modificación del algoritmo de **distancia de Levenshtein**[5]. Dicho algoritmo, en su versión estándar, devuelve la *cantidad de modificaciones que hacen falta para transformar un String en otro* y se puede formular de manera recursiva como:

$$\mathsf{levenshtein}(str1, str2) = \begin{cases} \mathsf{max}(\mathsf{length}(str1), \mathsf{length}(str2)) & \text{si } \mathsf{min}(\mathsf{length}(str1), \mathsf{length}(str2)) == 0 \\ \mathsf{min} \begin{cases} \mathsf{levenshtein}(\mathsf{init}(str1), str2) + 1 \\ \mathsf{levenshtein}(str1, \mathsf{init}(str2)) + 1 \\ \mathsf{levenshtein}(\mathsf{init}(str1), \mathsf{init}(str2)) + 1_{(\mathsf{last}(str1) \neq \mathsf{last}(str2))} \end{cases} & \text{en caso contrario} \end{cases}$$

Donde $1_{(\mathsf{last}(str1) \neq \mathsf{last}(str2))}$ vale 1 cuando $\mathsf{last}(str1) \neq \mathsf{last}(str2)$, y 0 en caso contrario.

La versión modificada que usaremos nosotros nos permite saber cuáles son efectivamente esas modificaciones que podemos aplicar para transformar el primer String en el segundo. Podemos definirla de la siguiente forma:

$$\mathsf{levenshtein2}(str1, str2) = \begin{cases} \text{Insertar todos los caracteres de } str2 & \text{si } \mathsf{length}(str1) == 0 \\ \text{Borrar todos los caracteres de } str1 & \text{si } \mathsf{length}(str2) == 0 \\ \mathsf{minLength} \begin{cases} \mathsf{Borrar}(\mathsf{length}(str1)) \ : \ \mathsf{levenshtein2}(\mathsf{init}(str1), str2) \\ \mathsf{Insertar}(\mathsf{length}(str1), \mathsf{last}(str2)) \ : \ \mathsf{levenshtein2}(str1, \mathsf{init}(str2)) \\ \mathsf{levenshtein2}(\mathsf{init}(str1), \mathsf{init}(str2)) \end{cases} & \text{si } \mathsf{last}(str1) == \mathsf{last}(str2) \\ \mathsf{minLength} \begin{cases} \mathsf{Borrar}(\mathsf{length}(str1)) \ : \ \mathsf{levenshtein2}(\mathsf{init}(str1), str2) \\ \mathsf{Insertar}(\mathsf{length}(str1), \mathsf{last}(str2)) \ : \ \mathsf{levenshtein2}(str1, \mathsf{init}(str2)) \\ \mathsf{Substituir}(\mathsf{length}(str1), \mathsf{last}(str2)) \ : \ \mathsf{levenshtein2}(\mathsf{init}(str1), \mathsf{init}(str2)) \end{cases} & \text{si } \mathsf{last}(str1) \neq \mathsf{last}(str2) \end{cases}$$

Donde minLength es una función que toma una lista de listas y devuelve aquella que tiene menor longitud. Por ejemplo, si utilizamos la versión estándar del algoritmo en los String "auto" y "automata":

$$\mathsf{levenshtein}(\text{“auto”}, \text{“automata”}) = 4$$

Pero, utilizando la versión modificada, obtenemos cuáles son efectivamente esos 4 cambios:

$$\begin{aligned} \mathsf{levenshtein2}(\text{“auto”}, \text{“automata”}) = [&\text{Insertar el caracter ‘a’ en la posición 4,} \\ &\text{Insertar el caracter ‘t’ en la posición 4,} \\ &\text{Insertar el caracter ‘a’ en la posición 4,} \\ &\text{Insertar el caracter ‘m’ en la posición 4}] \end{aligned}$$

Mirando el ejemplo anterior, podemos definir un Paquete de modificaciones como una lista cuyos elementos son Modificaciones (Insertar, Borrar o Substituir).

Luego, habiendo implementado el algoritmo modificado de Levenshtein, podemos pensar que un Archivo no es más que una serie de Paquete de modificaciones, de tal forma que:

- La primera versión de un Archivo es el String vacío.
- La segunda versión de un Archivo, es aplicarle el primer Paquete de modificaciones a la primera versión.
- La tercer versión de un Archivo, es aplicarle el segundo Paquete de modificaciones a la segunda versión.
- ...
- La última versión de un Archivo es aplicar todos los Paquete de modificaciones disponibles en el orden adecuado.

Nótese que cada Paquete de modificaciones solo guarda las diferencias (modificaciones) con la versión anterior. Veámoslo con un ejemplo:

- Creamos un Archivo vacío, por lo que no tiene ningún Paquete de modificaciones.
- Al Archivo le agregamos una versión “auto”, por lo que ahora tiene un Paquete de modificaciones que es: [Insertar el caracter ‘a’ en la posición 0, Insertar el caracter ‘u’ en la posición 1, Insertar el caracter ‘t’ en la posición 2, Insertar el caracter ‘o’ en la posición 3].
- Al Archivo le agregamos una versión “automata”, por lo que ahora se agrega un nuevo Paquete de modificaciones que es: [Insertar el caracter ‘a’ en la posición 4, Insertar el caracter ‘t’ en la posición 4, Insertar el caracter ‘a’ en la posición 4, Insertar el caracter ‘m’ en la posición 4].
- Si agregamos una última versión “automovil”, se agrega otro Paquete de modificaciones que es: [Insertar el caracter ‘l’ en la posición 8, Substituir el caracter en la posición 8 por ‘i’, Substituir el caracter en la posición 7 por ‘v’, Substituir el caracter en la posición 6 por ‘o’].

3. Archivos suministrados

Además del presente enunciado se suministra el archivo **esqueleto.hs**, que contiene un esqueleto del trabajo para que utilicen como base. En este archivo encontrarán:

- La definición de los tipos de datos que utilizaremos:

- `Modificacion`: con las posibilidades mencionadas en el enunciado.
- `PaqueteModificaciones`: como una lista de `Modificacion`.
- `Archivo`: que bien puede ser vacío, o bien puede ser el resultado de agregar un `PaqueteModificaciones` a otro `Archivo`.
- `SCV`: que bien puede ser vacío, o bien puede ser el resultado de agregar un `Archivo` a otro `SCV`.

- Los encabezados de las funciones que deben implementar (a veces con algún ejemplo).
- Funciones brindadas por la cátedra que pueden utilizar.
- Ejemplos para que puedan hacer pruebas. Recomendamos fuertemente agregar más ejemplos propios, distintos a los de la cátedra, para entender mejor los problemas.

4. Ejercicios

Los siguientes ejercicios están pensados para ir haciéndolos en el orden en el que están dados.

1. Crear la función `aplicarModificacion :: String -> Modificacion -> String` que aplica la modificación correspondiente (insertar, borrar o substituir) al `String` pasado por parámetro.
2. Crear la función `aplicarPaqueteModificaciones :: String -> PaqueteModificaciones -> String` que aplica todas las modificaciones en el paquete al `String` pasado por parámetro.
3. Crear la función `obtenerUltimaVersion :: Archivo -> String` que devuelve la última versión del `Archivo`.
4. Crear la función `cantVersiones :: Archivo -> Integer` que devuelve la cantidad de versiones que tiene un `Archivo`. En el caso de ser un `Archivo` vacío, decimos que este tiene 0 versiones.
5. Crear la función `obtenerVersion :: Integer -> Archivo -> String` que dado un `Archivo` *`file`* y un entero $0 \leq n \leq$ `cantVersiones` *`file`*, devuelve la n-ésima versión del `Archivo`.
6. Crear la función `levenshtein :: String -> String -> Integer` que devuelve la cantidad de modificaciones necesarias para transformar un `String` en el otro.
7. Crear la función `levenshtein2 :: String -> String -> PaqueteModificaciones` que devuelve las modificaciones necesarias para transformar el primer `String` en el segundo.
8. Crear la función `agregarVersion :: String -> Archivo -> Archivo` que dados un `String` *`str`* y un `Archivo` *`file`*, devuelve un `Archivo` *`nuevo_file`* resultado de agregar una nueva versión a *`file`*, de tal forma que `obtenerUltimaVersion` *`nuevo_file`* `==` *`str`* y la anteúltima versión de *`nuevo_file`* es igual a `obtenerUltimaVersion` *`file`*.

   Dicho de otra forma, el nuevo paquete de modificaciones que tiene *`nuevo_file`* con respecto a *`file`* es el resultado de utilizar el algoritmo de Levenshtein modificado para transformar `obtenerUltimaVersion` *`file`* en *`str`*.

Nota: Tener en cuenta que la función `length` devuelve un `Int` en vez de un `Integer`, como podríamos querer en algunos casos. Si es necesario, crear una función auxiliar `len :: [a] -> Integer`, de forma tal que `len xs = toInteger (length xs)`.

5. Normativa

- Fecha de **entrega**: 9 días después de la presentación:
  - Comisión Martes: jueves 30/6 hasta las 23:59 hs.
  - Comisión Miércoles: viernes 1/7 hasta las 23:59 hs.
  - Comisión Viernes: domingo 3/7 hasta las 23:59 hs.
- La entrega es en formato digital por e-mail a la dirección `algebra1-doc@dc.uba.ar`, con asunto "`[TP-Martes]`", "`[TP-Miercoles]`" o "`[TP-Viernes]`", dependiendo de la comisión en la que estén anotados. En el e-mail deben indicar los nombres completos, DNI o LU (libreta universitaria) y dirección de correo electrónico de **TODOS** los integrantes del grupo.

- Fecha de **coloquio**:
  - Comisión Martes: 5/7.
  - Comisión Miércoles: 6/7.
  - Comisión Viernes: 8/7.

  En los horarios de cursada de cada turno.
- El trabajo práctico deberá realizarse en grupos de TRES personas, sin excepción.
- El coloquio consistirá en preguntas sobre lo realizado a **todos** los integrantes del equipo.

6. Referencias

[1] Atlassian: What is version control.
URL: `https://www.atlassian.com/git/tutorials/what-is-version-control`.

[2] Git.
URL: `https://git-scm.com/`.

[3] Mercurial.
URL: `https://www.mercurial-scm.org/`.

[4] Subversion.
URL: `https://subversion.apache.org/`.

[5] Wikipedia: Levenshtein distance.
URL: `https://en.wikipedia.org/wiki/Levenshtein_distance`.