TEORÍA DE LOS LENGUAJES DE PROGRAMACIÓN

Curso 2015 - 16

Memoria de la Práctica

Misrraim Suárez Pérez – misrraimsp@gmail.com

Descripción de las funciones en Haskell

Se presenta en este apartado una breve descripción de las funciones implementadas en Haskell, tanto para resolver el problema base de la práctica, como para dar solución a la cuestión 1a.

resuelveSkyline :: [Edificio] -> Skyline

Esta es la función principal que resuelve el problema planteado en la práctica. Toma una lista de edificios y devuelve una lista de coordenadas como skyline. Si la lista de edificios posee más de un elemento entonces divide el problema y se llama a sí misma dos veces con los problemas divididos a la mitad como nuevos argumentos, para luego combinar las soluciones. Esta estrategia se prolonga hasta llegar al caso trivial, en que la lista solo tenga un elemento, convirtiendo entonces el edificio a skyline directamente.

edificioAskyline :: Edificio -> Skyline

Función simple que transforma un edificio (tripla de enteros) en una lista de coordenadas que representa su skyline asociado.

divide :: [Edificio] -> ([Edificio],[Edificio])

Tomando una lista de edificios genera una tupla compuesta de dos listas de edificios, donde cada una de ellas contiene la mitad de los edificios contenidos en la lista original. Si la lista original tiene un número impar de edificios, una de las listas resultado tendrá un edificio más que la otra.

combina :: Skyline -> Skyline -> Skyline

Esta función es la encargada de combinar los resultados de los subproblemas que va generando resuelveSkyline. Cede todo el trabajo a la función auxiliar *combina2*.

combina2 :: Skyline -> Skyline -> Int -> Int -> Int -> Skyline

Esta función se ha embebido de la función 'combina' mediante una clausula 'where', tal como se especifica en el enunciado de la práctica.

La razón de ser de esta función es poder conservar la información de la última altura eliminada de las dos listas argumento, así como de la altura en curso. Esto se hace mediante el paso de argumentos.

Tiene dos casos simples en los que una de las dos listas argumento está vacía, en los que devuelve directamente la lista de coordenadas no vacía. El caso general en que los dos argumentos lista son no vacíos se gestiona teniendo en cuenta la posición relativa de las coordenadas x de los dos skylines. Así, se generan tres situaciones diferentes: mayor, menor o igual. Además, en cada uno de los casos se comprueba que la altura de la nueva coordenada sea diferente de la altura en curso.

dibujaSkyline :: Skyline -> [Char]

Toma un skyline como entrada y devuelve, usando las funciones auxiliares *calculaAltura* y *escribe*, la cadena de caracteres asociada al skyline.

calculaAlturas :: Skyline -> Int -> Int -> [Int]

Función auxiliar que toma una lista de coordenadas *lista*, y usando los parámetros de coordenada en curso *x* y altura en curso *h*, genera una lista de alturas asociada al skyline. La estrategia es ir haciendo llamadas recursivas mientras se va consumiendo la lista de coordenadas argumento.

escribe :: [Int] -> Int -> Int -> [Char]

Función auxiliar encargada de generar la cadena de caracteres asociada al skyline. Para ellos toma como argumento la lista de alturas *lista*, la coordenada *x* en curso y la altura *h* en curso. La filosofía es ir recorriendo todas las coordenadas para cada altura, decidiendo en cada caso en función de la lista de alturas el carácter que corresponda.

Descripción de los predicados en Prolog

Siguiendo la línea del apartado anterior, se muestra ahora una breve descripción de los predicados implementados en Prolog, tanto para resolver el problema base de la práctica, como para solucionar la cuestión 1b.

Se puede apreciar que existe una cierta correspondencia entre las funciones descritas y los siguientes predicados, debido ello al carácter declarativo de ambos lenguajes.

Es de destacar que los problemas propuestos en el enunciado se han resuelto de forma que el programa Prolog escrito encuentre únicamente la solución buscada, y una vez lo haga pare, sin búsquedas infinitas. Esto es, se ha hecho hincapié en mejorar la eficiencia en la ejecución del código a través de la poda del espacio de búsqueda.

resuelveSkyline(Edificios, Skyline)

Predicado principal de la práctica. Verdad si 'Skyline' es la lista de coordenadas correspondiente al skyline de la lista de edificios 'Edificios'.

edificioAskyline(Edificio, Skyline)

Predicado verdadero si ‘Skyline’ representa las coordenadas del skyline del edificio pasado como primer argumento, ‘Edificio’.

divide(Fuente, Destino1, Destino2)

Predicado verdadero si las listas ‘Destino1’ y ‘Destino2’ representan mitades de la lista ‘Fuente’, o a lo sumo con un elemento de diferencia entre ambas.

combina(Skyline1, Skyline2, Resultado)

Predicado verdadero si 'Resultado' es el skyline obtenido de combinar los 'Skyline1' y 'Skyline2'. Este predicado cede todo el trabajo al predicado ‘combina2’.

combina2(Skyline1, Skyline2, Hsky1, Hsky2, Hactual, Resultado)

Predicado auxiliar que lleva a la práctica el trabajo del predicado 'combina'. Con los parámetros ‘Hsky1’, ‘Hsky2’ y ‘Hactual’ se controlan las diferentes llamadas recursivas, permitiendo obtener el resultado deseado.

dibujaSkyline(Skyline)

Predicado principal que resuelve la cuestión 1b de la práctica. Presenta en pantalla el dibujo del skyline pasado como argumento.

son_las_alturas(Skyline, Xactual, Hactual, Alturas)

Predicado auxiliar de 'dibujaSkyline', verdad si el último argumento representa la lista de alturas asociada al skyline pasado como primer argumento. Además, usa los argumentos ‘Xactual’ y ‘Hactual’ para controlar las llamadas recursivas.

escribe(Alturas, Xactual, Hactual)

Predicado auxiliar de 'dibujaSkyline', encargado de sacar por pantalla los caracteres que representan el skyline asociado a la lista de alturas pasada como primer parámetro. Nuevamente cuenta con ‘Xactual’ y ‘Hactual’ para controlar la recursividad.

maximo2(X, Y, Maximo)

Predicado auxiliar, verdad si el tercer argumento es el máximo entre el primer y el segundo argumento.

maximoN(Lista, Maximo)

Predicado auxiliar, verdad si el segundo argumento es el máximo de la lista pasada como primer argumento.

es_el_elemento(Lista, Posicion, Elemento)

Predicado auxiliar, verdad si el tercer argumento se encuentra en la posición dada por el argumento ‘Posicion’ dentro de la lista dada como primer argumento, entendiendo que la cabeza de la lista ocupa la posición '0'.

es_la_longitud(Lista, Longitud)

Predicado auxiliar, verdad si el segundo argumento es la longitud de la lista pasada en el primer argumento.

Cuestiones sobre la práctica

En este apartado se tratará las cuatro cuestiones planteadas sobre la práctica.

- ***Cuestión 1***

Los apartados a y b de la cuestión primera se dan por resueltos en el código fuente adjunto y en los comentarios de funciones y predicados realizados en los dos apartados anteriores.
Es de resaltar, nuevamente, que el código Prolog usa el predicado corte para una vez dibujado el skyline correspondiente concluya la ejecución.
También comentar que las funciones auxiliares implementadas en Haskell se han escrito a parte, y no embebidas en una clausula *where* como se especificaba para la primera parte de la práctica.

- ***Cuestión 2***

La respuesta a la pregunta planteada en esta cuestión segunda es, desde mi punto de vista, directa: **Haskell**.
Es verdad que a priori Prolog pueda parecer más eficiente en cuanto a programación ya que el hecho de ser un lenguaje declarativo no tipificado agiliza los trámites de trasladar la idea desde la mente del programador al código del programa. Pero debido al mecanismo de búsqueda automático de soluciones que Prolog implementa, el programador, ante un problema de solución única como el planteado en esta práctica, tiende a hacer suyo el objetivo adicional de parar la ejecución una vez se encuentra dicha solución única, y por supuesto evitar recorrer ramas infinitas. Para ello es necesario realizar un esfuerzo adicional de programación, ajeno a la semántica declarativa. En cambio, Haskell, además de contar con la expresividad otorgada por su semántica declarativa (pura), es ajeno a ese esfuerzo adicional. La solución Haskell es única siempre.
Java es el menos eficiente en cuanto a programación. Su naturaleza imperativa lo acerca demasiado a la máquina en comparación con los lenguajes declarativos.

- ***Cuestión 3***

Este predicado modifica el recorrido automático a través del árbol de búsqueda (que Prolog crea cuando se hace una consulta). El recorrido que Prolog implementa del árbol es en profundidad, de izquierda a derecha. Cuando el objetivo a satisfacer sea el corte (que se verifica siempre) se bloquean todos los caminos de los nodos superiores, en la rama del corte, que aún no hayan sido recorridas. El efecto es que esas vías del árbol son ‘cortadas’ y nunca exploradas. Se dice que el árbol ha sido podado. Este predicado es un ejemplo de mecanismo para controlar la ejecución del programa.

Esto es muy útil, por ejemplo, cuando se sabe que el resultado del cómputo es único, como es el caso en esta práctica. En estos casos, cuando se alcanza la solución es conveniente que el sistema no siga buscando. Como se puede ver en el código, y como se ha comentado al inicio del apartado de descripción de los predicados, allá donde se ha necesitado se ha usado el predicado corte para garantizar que una vez encontrada la solución en el árbol no se siga buscando.

En Java existen elementos en el lenguaje para controlar la ejecución: las sentencias condicionales y los bucles (con todo el azúcar sintáctico que los envuelve). Además, es de destacar la existencia de las sentencias 'break' (sale incondicionalmente del bucle) y 'continue' (salta incondicionalmente a la siguiente iteración del bucle), que aún añaden más flexibilidad al control de ejecución.

- ***Cuestión 4***

En cuanto al sistema de tipos, Haskell y Java son totalmente opuestos a Prolog. Haskell y, en menor medida, Java son *lenguajes fuertemente tipados*, mientras que en Prolog no existen tipos en absoluto, es un *lenguaje no tipado*.

En el problema de esta práctica se han usado los tipos de datos Edificio, Coordenada y Skyline, de una forma diferente en cada lenguaje.

En **Prolog**, por lo comentado anteriormente, no se han usado tipos de datos en su sentido estricto. Sin embargo, para representar los edificios y las coordenadas se han usados *estructuras*. Este término es el usado en el libro 'Programming in Prolog' 5th ed, de Clocksin y Mellish, aunque en Standard Prolog se le conoce como términos compuestos.

En **Haskell** se usan tipos sinónimos, por lo que no son nuevos tipos de datos, sino tipos predefinidos con nombre personalizado. Un edificio es una tripla de enteros, una coordenada una tupla de enteros, y un skyline una lista de coordenadas.

En **Java** los tipos de datos son primitivos (con semántica de almacenamiento) o referenciados (con semántica de puntero). Los nuevos tipos creados por el programador solo pueden ser referenciados, y se crean mediante el uso de clases. Skyline, Edificio y Coordenada son clases. Representan tipos abstractos de datos. Las operaciones que se pueden realizar con cada uno de estos TAD se pueden ver en el código Java del problema. La clase Skyline posee un campo, homónimo, que se representa como un ArrayList de objetos de la clase Coordenada. La clase Coordenada tiene dos campos enteros y la clase Edificio tres campos también de enteros. Los constructores de tipos usados son los constructores de las clases que implementan estos TAD.

En conclusión, en Prolog no existe el concepto de tipo de datos (lenguaje no tipado), en Haskell se han usado tipos sinónimos, y en Java se han usado clases para dar forma a los TAD.