Universidad Nacional de Rosario

Facultad de Ciencias Exactas Ingeniería y Agrimensura

Informe sobre Intérprete de Funciones de Lista

Licenciatura en Ciencias de la Computación

Estructuras de Datos y Algoritmos I

Franco Bramucci

Introducción

En el presente documento se comentará la implementación de un intérprete de funciones de listas con operaciones para definir listas y funciones (`defl` y `deff`), aplicar funciones a listas (`apply`) y buscar funciones que dada una cierta entrada devuelvan una salida (`search`).

Diseño

Estructuras de datos utilizadas

THash (`thash.h`): Para almacenar las listas y funciones se utilizó una tabla hash para cada una. De esta forma se permite un acceso de $O(1)$ a cualquier función o lista. El hasheo se realiza sobre el nombre de cada función o lista que se encuentran en formato `string`. La función hash utilizada es la presentada en *The C Programming Language - 2nd Edition.* El método de resolución de colisiones utilizado es llamado coalesced hashing o de listas mezcladas.

La tabla hash se encuentra dividida en dos regiones, una región de direcciones que ocupa aproximadamente un 86% de la capacidad total (ajustado a un número primo y una región de colisiones que ocupa el 14% restante. Estos números son en base al paper de Vitter, *J.S. Implementations for Coalesced Hashing, 1982* donde se muestra que es la forma más efectiva cuando se realizan búsquedas fructuosas en la tabla.

El hasheo se realiza sobre sobre la región de direcciones. Cuando ocurre una colisión se inserta el elemento colisionado en la región de colisiones. Para llevar registro de dónde se colocó el último elemento colisionado se utiliza un índice que almacena dicha posición. El mismo comienza en el final de la tabla y va decrementando en cada inserción hasta encontrar una casilla vacía.

En el momento que la región de colisiones se llena, el indice se ubica en la región de direcciones y se insertan los elementos allí. Cuando el índice llega a -1 es dónde la tabla se encuentra llena y se procede a realizar un rehasheo. Es decir que se rehashea cuando el factor de carga es del 100%. Esto es así ya que con esta implementacion no hay diferencias significativas de rendimiento con un factor de carga alto.

Lista (`lista.h`): Las listas se representan con el tipo `Lista` que fue implementado utilizando `DList`, es decir listas doblemente enlazadas generales. Se eligió esta estructura ya que permite acceder en $O(1)$ al comienzo y al final de la lista. De esta forma se pueden aplicar las funciones de lista primitivas

también en tiempo constante.

FLista (`flista.h`): Las funciones de listas se representan con el tipo `FLista` que fue implementado usando la estructura `Vector` general. Esta estructura provee indexación, lo que resulta muy útil para aplicar el operador repetición en funciones de lista. En `FLista` se almacenan los identificadores de las funciones y los símbolos `<` y `>` que representan el inicio y el fin de una repetición respectivamente. Se verá más adelante que para aplicar repeticiones se almacena la posición del comienzo de la repetición y se accede mediante dicho índice.

Otras estructuras (`DList Vector Token Pila`): Algunas de estas estructuras ya fueron mencionadas en la definición de las anteriores. `Token` se utiliza para tokenizar la entrada estándar y posteriormente parsearla. `Pila` se utiliza en `apply` para almacenar los inicios de las repeticiones como se verá más adelante.

Algoritmos

Apply (`apply.c apply.h`)

En `apply.c` se encuentran definidas principalmente 2 funciones.

`apply_flista`: Recibe una funcion de tipo `FLista` y una lista de tipo `Lista`. Se define una pila de enteros para almacenar las posiciones donde comienzan las repeticiones.

Un bucle `for` recorre los elementos de la función. En caso de que el elemento sea un `<` significa que comienza una repetición y se deben aplicar todas las funciones comprendidas entre `<` y `>` hasta que los extremos de la lista sean iguales.

En primer lugar se comprueba si la lista cuenta con más de dos elementos, ya que la repetición se encuentra definida para listas que cumplan esa condición. Si esto último no se cumple, se devuelve un error de dominio, es decir que la lista provista no pertenece al dominio de la función dada.

Luego se corrobora si los extremos son iguales, en caso de que lo sean entonces no se debe ingresar a la repetición y se debe avanzar hasta el fin de la repetición sin ejecutar ninguna función comprendida en la misma.

En caso contrario, se almacena la posición del comienzo de la repetición en la pila, se continua con el bucle y se aplican todas las funciones que se encuentran hasta llegar al final de la misma. Al llegar al final de la repetición se comprueba nuevamente si los extremos son iguales. Si lo son, se continua

con el resto de la función. Si no, entonces se obtiene el comienzo de la repetición correspondiente del tope de la pila y se actualiza el índice del bucle a dicha posición. De esta forma se ejecuta la función dentro de la repetición nuevamente.

El uso de la pila es elemental ya que permite almacenar los comienzos de repetición a medida que el bucle los recorre. De esta forma, cuando se llega a un fin de repetición se sabe que el inicio correspondiente a dicho fin es el tope de la pila.

En caso de que el elemento de la función sea un identificador, entonces se llama a

`aplicacion_singular`: Recibe un `char*` que representa al identificador de una función, si es una función primitiva la aplica. Si es una función definida por el usuario entonces se busca su definición en la tabla hash de funciones y se llama a `apply_flista` con esa definición.

Search (`search.c search.h`)

`buscar_funcion`: La función principal del algoritmo. La idea principal es recorrer el universo de posibles funciones por composición hasta una cierta profundidad hasta encontrar aquella que aplicada al primer elemento de cada par, devuelva el segundo elemento respectivo. En este caso la profundidad es dada por `PROFUNIDAD_MAX n`. El universo de posibles funciones puede ser representado como un árbol m-ario, donde

$$m = \#funciones\_primitivas + \#funciones\_del\_usuario$$

Por lo que dada la profundidad máxima, la cantidad de posibles funciones es

$$\sum_{i=1}^{n} m^i = m^1 + m^2 + \ldots + m^n$$

Si se contasen solamente las primitivas y consideramos $n = 8$ serían $2015538 \approx 2 \cdot 10^6$ posibles funciones. Si además de las primitivas se añadiesen 4 funciones definidas por el usuario, la cantidad escala a $111111110 \approx 1 \cdot 10^8$.

Solo con fuerza bruta no es posible debido a la gran cantidad de funciones y el costo de aplicar cada una de ellas. Por lo tanto debemos probar funciones de forma inteligente. La estrategia empleada en este caso será la de backtracking.

`buscar_funcion` implementa backtracking para construir, paso a paso, una posible composición de funciones que transforme `listaInput` en `listaOutput`.

La condición de corte del algoritmo es si la cantidad de funciones compuestas ya alcanzó `PROFUNDIDAD_MAX`. Esto evita que el algoritmo explore recursivamente soluciones demasiado largas.

El vector `elementosTabla` contiene todas las funciones definidas. Se prueban una por una, siempre que `podar` no indique que conviene descartarla de entrada para evitar composiciones sin sentido como `Od Dd`. También se descartan en caso de que aplicadas a la `listaInput` arrojen un error.

En cada ejecución de `buscar_funcion` se insertará una subfunción a `funcion`, se llamará recursivamente a `buscar_funcion` con `funcion` compuesta con la subfunción y se pasara como `listaInput` a la lista con la aplicación de subfunción. De esta forma se ahorra recomputar cada posible función y se aprovechan las aplicaciones anteriores.

A modo de ejemplo haremos un paso a paso de una posible ejecución. Supongamos que `Od Oi Sd Si Dd Di` es el orden en que aparecen las funciones en `elementosTabla`, `listaInput = [1,2]` y `listaOutput = [1,2,1]`. Supongamos además que `PROFUNDIDAD_MAX = 2`

1. Recorremos `elementosTabla` y aplicamos `Od` a `listaInput` y la insertamos a `funcion`.
2. Como no es la función buscada se llama nuevamente a `buscar_funcion` con `listaInput = [1,2,0]` y `funcion = Od`.
3. Hacemos lo mismo que en 1.
4. Como no es la función buscada se llama nuevamente a `buscar_funcion` con `listaInput = [1,2,0,0]` y `funcion = Od Od`.
5. Como el largo de `funcion` es $\geq$ `PROFUNDIDAD_MAX`. Entonces se llega a la condición de corte. Luego se elimina la última subfunción `Od` de `funcion` y el estado de la lista vuelve a ser [1,2,0].
6. Insertamos la próxima función de `elementosTabla`, en este caso `Oi`.
7. Como no es la función buscada se llama nuevamente a `buscar_funcion` con `listaInput = [0,1,2,0]` y `funcion = Od Oi`.
8. Como el largo de `funcion` es $\geq$ `PROFUNDIDAD_MAX`. Entonces se llega a la condición de corte. Luego se elimina la última subfunción `Oi` de `funcion` y el estado de la lista vuelve a [1,2,0].
9. Insertamos la próxima función de `elementosTabla`, en este caso `Sd`.

10. Aplicamos `Sd` a nuestra copia de `listaInput` y obtenemos la lista [1,2,1] que es `listaOutput`. Si tuvieramos más listas probaríamos `funcion` con el resto de las listas usando `probar_funcion_con_resto_de_pares`.

11. Se retorna el valor 1 en las llamadas recursivas y se termina la ejecución.

Algunos puntos a considerar.

- Se copia `listaInput` a `copiaInput`, para no modificar el estado original.
- Se intenta aplicar la subfunción con `aplicacion_singular`. Si la aplicación produce un error (dominio o cantidad de ejecuciones), se descarta esta rama.
- Se utiliza una memorización para estados de lista que ya se hayan computado. De esta forma, si la función construída devuelve una lista que ya fue obtenida por otra función, se descarta la última subfunción y se continúa por otra rama. Una salvedad sobre esto es que si un estado de lista ya existe pero fue generado por una función de mayor profundidad (mayor cantidad de funciones compuestas) entonces no se descarta la rama y se continua con dicha función. Esta decisión fue tomada ya que si un estado fue generado por una función de mayor largo, entonces se explorarán menos niveles sobre ese estado de lista ya que la función estaría más cerca de la condición de corte. Otra salvedad es que no se almacena el estado de lista equivalente a `listaOutput` ya que si hay una función que genera dicho estado para la primera función del primer par pero no funciona para el segundo entonces el algoritmo dejaría de buscar posibles funciones.

Si analizamos la forma en la que se recorre el universo de funciones es la de una búsqueda en profundidad. Lo positivo de esto es que, además de utilizar el resultado de la llamada anterior y ahorrar tiempo de computo también se ahorra memoria ya que como máximo se están almacenando n subfunciones en simultáneo. Si intentáramos hacer una búsqueda por niveles, se deberían almacenar los niveles enteros, con lo que que se tendrían que almacenar m^n funciones en una cola, mucho más costoso en cuanto a memoria. Además, si quisiéramos aprovechar el resultado de haber aplicado la función a la `listaInput` deberíamos guardar la misma cantidad de listas en la cola, cada una asociada a la función correspondiente.

Referencias

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, 2nd Edition. Prentice Hall, 1988.

[2] Jeffrey Scott Vitter. "Implementations for Coalesced Hashing." *Communications of the ACM*, 25(2): 120–125, 1982.