Programación Declarativa

Matrilog

Creación de una librería para manipular matrices en Prolog escrita en C

Lucía Calzado
26-4-2020

Índice

1. Introducción

En este documento se muestra el desarrollo de Matrilog, una librería escrita en C para Prolog, con el objetivo de implementar en él algunas operaciones con matrices.

Se hablará de las decisiones que se han tenido que tomar para diseñar las matrices en C, de la librería que se ha usado, proporcionada por Prolog[1] y de la estructura de esta librería, que podría servir como guía para muchas otras.

Para conseguir esto, se ha hecho uso de las listas que brinda el lenguaje Prolog, las cuales se han pasado a matrices en C para poder realizar las operaciones requeridas (véase apartado 5. Funciones implementadas para echar un vistazo a las operaciones que se han codificado).

Las operaciones con matrices implementadas han sido: sumar todos los elementos de una matriz, multiplicar una matriz por un número entero, sumar dos matrices y multiplicar dos matrices.

Todas las capturas de pantalla de este documento, a excepción de las de los anexos, son del código fuente, que se puede encontrar junto a este documento y en el repositorio de GitHub https://github.com/Geadalu/DeclarativeProg/tree/master/TrabajoC.

2. Decisiones de diseño

En este apartado se enumerarán las distintas decisiones que se han tomado para el diseño de la librería.

Vectores unidimensionales

En lugar de usar una matriz bidimensional en C, declarada como matriz[][], se ha decidido usar una cadena (de aquí en adelante, vector) unidimensional. Gracias al control de este vector que podremos ver más adelante en el código fuente de la librería, a pesar de que dentro del programa se trata la lista de listas de Prolog como un vector, esto para el usuario es completamente transparente y simple de utilizar.

Por ejemplo, la lista de Prolog [[1, 3], [5, 2], [1, 1]], correspondería a la siguiente matriz:

$$\begin{pmatrix} 1 & 3 \\ 5 & 2 \\ 1 & 1 \end{pmatrix}$$

Y el programa la interpretaría como [1, 3, 5, 2, 1, 1].

Encapsulamiento del vector en un struct

También se ha decidido encapsular los atributos del vector en una estructura de C struct a la que se ha llamado Matrix_C.

```
typedef struct {
    int rows;
    int columns;
    double* matrix;

}Matrix_C;
```

Esto permite acceder a ellos en cualquier parte del programa, y se evita tener que pasar constantemente por argumentos las filas y las columnas, así como la propia matriz.

Operaciones en float

Una manera rápida y eficaz de evitar conflictos entre tipos de datos cuando el usuario meta enteros y números de punto flotante (de ahora en adelante, *float*) en una misma matriz es convertir todos los números al mismo tipo, y hacer así las operaciones. Debido a que, si al menos un número de la matriz fuera float, el resultado debiera ser también float, se ha decidido pasar todos los números a tipo float y hacer así las operaciones, independientemente de si es una matriz de enteros, de puntos flotantes o mixta.

Restricciones

Debido a la decisión explicada en el punto a, al usuario se le impone la restricción de meter en los predicados el número de filas y de columnas que tienen las matrices

bidimensionales que está introduciendo. Esto es necesario, en mayor parte, para la necesaria y correcta reserva de memoria en C que necesita cualquier vector o matriz para inicializarse.

```c
//Inicializamos el espacio de memoria necesario para la matriz
Matrix_C* createMatrix(int rows, int columns){
    Matrix_C* matrix = calloc(1, sizeof(Matrix_C));
    matrix->rows = rows;
    matrix->columns = columns;
    matrix->matrix = calloc(rows*columns, sizeof(double));
    return matrix;
}
```

3. Librería SWI-Prolog

Para realizar este proyecto se ha hecho uso de la librería SWI-Prolog para C[1]. Para introducirla en la librería, se debe añadir como cabecera: `#include <SWI-Prolog.h>`.

Esta librería contiene herramientas para escribir predicados de Prolog en C, convertir términos de Prolog a tipos de datos de C y unificar tipos de datos de C a términos de Prolog, entre muchas otras operaciones. Aquí se describirán las que se han usado para escribir Matrilog.

a. Términos y tipos de llamadas utilizados

`term_t`

Este tipo de variable representa cualquier tipo de término en Prolog. Puede ser un entero, un float, una lista… Si se necesita operar directamente con estos términos o hacer cualquier tipo de referencia a ellos, se deben usar las funciones `PL_new_term_ref()` y `PL_copy_term_ref(+term)`. El primero crea una nueva referencia para un término nuevo, y el segundo copia una ya existente. Podemos ver un claro ejemplo del uso de estas llamadas en el manejo de las listas en la librería, donde a partir de una lista (en la librería, un dato term_t llamado list), construimos referencias a su cabecera y su cola:

```
term_t head = PL_new_term_ref();
term_t tail = PL_copy_term_ref(list);
```

Para, posteriormente, crear la lista con la función `PL_get_list()`:

```
if(!PL_get_list(tail, head, tail))
    return 0;
```

`foreign_t`

Este tipo de definición de función es el que permite escribir el predicado para Prolog. Los métodos de la librería que lleven este tipo, son los que posteriormente se registrarán como predicados en la función `install()`, que veremos más adelante.

```
foreign_t pl_total_matrix (term_t list_ofLists, term_t rows_, term_t columns_, term_t result){
```

`install_t`

Dentro de esta función, que en la biblioteca se encuentra al final, por cada predicado escrito se tendrá que definir, mediante `PL_register_foreign`(), el nombre del predicado, el número de argumentos y cómo se llama el predicado dentro de la librería. De esta forma se queda registrado y listo para usar en la consola de Prolog. Como ejemplo, en esta función, se registra un solo predicado llamado `sum_matrix`, al cual se le tienen que pasar 4 argumentos y que corresponde con el predicado de la librería `pl_sum_matrix`:

```
install_t install() {
    PL_register_foreign("total_matrix", 4, pl_total_matrix, 0);
}
```

PL_fail y PL_succeed

Estas dos llamadas permiten la comunicación para que Prolog devuelva `false`, en el caso de `PL_fail` y `true`, en el caso de `PL_succeed`. Su utilidad es muy simple: controlar si el programa, en algún punto debería fallar, y en ese caso, la operación entera. De esta forma, por ejemplo, hemos controlado que el usuario no pueda meter caracteres en vez de números para operar las matrices.

Funciones de comprobación de tipos

Hay varias funciones que proporciona la librería para comprobar el tipo de dato del término en cuestión. Las que se han usado han sido:

`PL_is_number(+term_t)`, que comprueba que un término es un número, pero sin comprobar el tipo del número.

`PL_is_float(+term_t)`, que comprueba que un término es un número de punto flotante.

`PL_is_integer(+term_t)`, que comprueba que un término es un número entero.

Todas estas funciones retornan un 0 en caso de fallo, y 1 en caso de terminar correctamente, por eso, en caso de estar dentro del método del predicado de Prolog, se debe controlar su salida y añadir `PL_fail` en los que retornen un 0.

```
if (PL_is_integer(rows)){
    if(!PL_get_integer(rows, &rows_)){
        fprintf(stderr, "La variable filas no es válida\n");
        PL_fail;
    }
}
```

Funciones de asignación de tipos de Prolog a C

Estas funciones pasan el término de Prolog elegido a un tipo de dato de C.

`PL_get_float(+term_t, -float_number)`, que coge el término `term_t` y lo intenta convertir a un float de C. El término `float_number` tiene que ser una referencia a memoria.

`PL_get_integer(+term_t, -integer_number)`, que coge el término term_t y lo intenta convertir a un integer de C. El término integer_number tiene que ser una referencia a memoria.

Hay una función especial que opera con términos en Prolog, pero no los pasa a C, y es esta:

`PL_get_list(+term_t_tail, +term_t_head, -term_t_tail)`, que coge la lista y la divide en su cabeza y su cola.

Colocación de datos en C a los términos de Prolog

De igual manera que se pueden pasar los términos de Prolog a datos en C con las funciones vistas en el apartado anterior, también se puede hacer lo contrario: pasar de datos en C a tipos `term_t` de Prolog.

Esto es verdaderamente útil a la hora de la unificación del resultado final, por ejemplo.

`PL_put_float(-term_t_r, +float_number)`, donde `float_number` es un número float de C, y `term_t_r` es un término de Prolog del tipo `term_t`.

`PL_put_integer(-term_t_r, +integer_number)`, donde `integer_number` es un número integer de C, y `term_t_r` es un término de Prolog del tipo `term_t`.

`PL_cons_list(+term_t_List, +term_t_List2, -term_t_List)`, donde se añade la lista `term_t_List2` a la lista `term_t_List`.

`PL_put_nil(+term_t_List)`, que añade al término `term_t_List` la constante de terminación de una lista.

Unificación

Para unificar el resultado y retornarlo en Prolog, usamos la función `PL_unify`:

`PL_unify(-term_t_resultado, -term_t_variable)`, donde `term_t_resultado` es el resultado que da el programa, convertido a `term_t`, y `term_t_variable` es la variable que se le pasa al predicado para obtener el resultado.

```
if(PL_put_float(r, sum)){
    return PL_unify(r, result);
} else {
    PL_fail;
}
```

En esta imagen podemos ver cómo se intenta meter en el término `r` el float `sum`, y unificar este término `r` con `result`, que es la variable que se le pasa por argumentos al predicado.

b. Idea general de construcción

Para construir una librería en C para Prolog, generalmente, se han de seguir tres pasos:

- Pasar de términos de Prolog a estructuras y datos en C

Todo lo que el usuario introduce mediante el predicado de Prolog, la librería necesita interpretarlo. Para ello, hacemos uso de las funciones anteriormente mencionadas para pasar tanto términos numéricos a integers o float, como listas a matrices.

- Operar con esos datos y estructuras directamente en lenguaje C

Cuando se tengan todos los datos en lenguaje C, se opera con ellos en cada predicado. Dependiendo del tipo o uso de ese predicado, se harán unas operaciones u otras, controlando siempre el flujo del programa con `PL_succeed` o `PL_fail.`

- Pasar de estructuras y datos en C a términos de Prolog

Finalmente, para poder devolver la respuesta del predicado (notada en todos los predicados de la librería como *result*), se necesita pasar el resultado a término en Prolog y unificarlo con el término del resultado.

c. Cómo compilar una librería

Para compilar el archivo .c, usaremos este comando:

```
swipl-ld -c  nombre_archivo.c
```

Para convertir el archivo resultante en librería de Prolog, lanzaremos este comando:

```
swipl-ld -shared -o nombre_libreria.so nombre_archivo.o
```

Finalmente, dentro de la consola de Prolog solo tendríamos que cargar la librería:

```
load_foreign_library(nombre_libreria).
```

Y ya se podrían usar los predicados según se han escrito en la función `install()`. Se pueden ver ejecuciones de estos comandos en el anexo, ejecutados mediante un Makefile.

4. Contenido de la librería

En este apartado se hablará de la estructura que ha seguido la librería: las funciones que tiene implementadas, tanto esenciales como de apoyo, y los predicados escritos para Prolog instalados.

Estructura creada para la matriz

La estructura creada para la matriz es lo primero que se lee en la librería.

```
typedef struct {
  int rows;
    int columns;
    double* matrix;

}Matrix_C;
```

De esta estructura se ha hablado en el apartado de Decisiones de diseño.

Reserva de memoria para la matriz

En cualquier parte de la librería donde es necesario crear una matriz, se llama a este método, puesto que en C se necesita reservar memoria para las matrices (al no saber el compilador, en el momento de la creación, cuánta memoria tiene que reservar para ella).

```
//Inicializamos el espacio de memoria necesario para la matriz
Matrix_C* createMatrix(int rows, int columns){
    Matrix_C* matrix = calloc(1, sizeof(Matrix_C));
    matrix->rows = rows;
    matrix->columns = columns;
    matrix->matrix = calloc(rows*columns, sizeof(double));
    return matrix;
}
```

Liberación de la memoria

La función `destroyMatrix` libera la memoria utilizada por la matriz. Se llama al final de cada predicado para liberar la memoria de todas las matrices que se han usado.

```
//destroyMatrix
//Liberamos la memoria del sistema
void destroyMatrix(Matrix_C* matrix){
    free(matrix);
    matrix = NULL;
}
```

Función `checkInteger`

Esta función comprueba si el term_t del parámetro es un integer, en cuyo caso lo asigna al int integer. He decidido construirla para ahorrar líneas, puesto que hay que llamarla cada vez que se quieren recoger el número de filas y columnas de cada matriz.

```
//checkInteger
//Comprueba si el term_t introducido es un integer
int checkInteger(int* integer, term_t term){
    if(PL_is_integer(term)){
        if(!PL_get_integer(term, integer)){
            return 0;
        }
        return 1;
    } else {
        return 0;
    }
}
```

Función `list_toRow`

Esta función es muy importante, pues es la que pasa una fila de la lista (esto es, una de las listas dentro de la lista) a un vector de C, para posterior uso en la función `list_toMatrix`(), que se explicará más adelante.

```
//list_toRow
//para poner una lista en una fila de una array
int list_toRow(int currentIndex, term_t list, Matrix_C* matrix){
    double number_d = 0.0;
    int number_i = 0;
    int i = 0;
    double aux = 0;

    if (!PL_is_list(list))
        return 0;

    term_t head = PL_new_term_ref();
    term_t tail = PL_copy_term_ref(list);

    for (i=0; i<matrix->columns; i++){

        if(!PL_get_list(tail, head, tail))
            return 0;

        if (!PL_is_number(head))
            return 0;

        if (PL_is_float(head)){
            if (PL_get_float(head, &number_d)){
                matrix->matrix[i+currentIndex*matrix->columns] = number_d;
            }
        } else {
            if (PL_get_integer(head, &number_i)){
                aux = (double) number_i;
                matrix->matrix[i+currentIndex*matrix->columns] = aux;
            }
        }
    }
    return 1;
}
```

Retorna 0 si ha fallado, y 1 si ha tenido éxito.

Función `list_toMatrix`

Esta función rellena un vector llamando varias veces a `list_toRow()`. Se llama en cada predicado de Prolog, para pasar la lista que introduce el usuario a vector en C.

```c
//list_toMatrix
//Pasar de lista a matriz en C
int list_toMatrix(term_t list_ofLists, Matrix_C* matrix){
    term_t head = PL_new_term_ref();
    term_t tail = PL_copy_term_ref(list_ofLists);
    int i;

    for (i=0; i<matrix->rows; i++){

        if(!PL_get_list(tail, head, tail)){
            fprintf(stderr, "Error recogiendo la lista.\n");
            return 0;
        }

        if (list_toRow(i, head, matrix) == 0){
            fprintf(stderr, "Error pasando la fila %d a matriz\n", i);
            return 0;
        }
    }
 return 1;
}
```

Función `matrix_toList`

Esta función es la opuesta a la anterior. Se llama cuando se necesita pasar una matriz resultado a lista, para unificarla y devolverla en el predicado.

```c
//matrix_toList
//Pasar de matriz en C a lista de Prolog
int matrix_toList(Matrix_C* matrix, term_t term){
    int i, j;

    if(!PL_put_nil(term)){
        fprintf(stderr, "Error pasando la matriz a lista\n");
        return 0;
    }

    for (i=matrix->rows-1; i>=0; i--){
        term_t currentRow = PL_new_term_ref();
        if(!PL_put_nil(currentRow)){
            fprintf(stderr, "Error pasando la matriz a lista\n");
            return 0;
        }
        for(j=matrix->columns-1; j>=0; j--){
            term_t aux_t = PL_new_term_ref();
            if(!PL_put_float(aux_t, matrix->matrix[j+i*matrix->columns])){
                fprintf(stderr, "Error recogiendo un número float de la matriz.\n");
                return 0;
            } else if(!PL_cons_list(currentRow, aux_t, currentRow)){
                fprintf(stderr, "Error construyendo una fila de la lista.\n");
                return 0;
            }
        }
        if(!PL_cons_list(term, currentRow, term)){
            fprintf(stderr, "Error añadiendo una fila a la lista.\n");
            return 0;
        }
    }
    return 1;
}
```

Predicado `multiply_matrix_by`

Este predicado multiplica una matriz por un número dado. Los argumentos son:

- `term_t multiplier_`: el número por el que se quiere multiplicar la matriz
- `term_t list_ofLists`: la matriz
- `term_t rows_`: las filas que tiene la matriz
- `term_t columns`: las columnas que tiene la matriz
- `term_t result`: el resultado que se devolverá

```
//Multiplicar número por matriz
foreign_t pl_multiply_matrix_by (term_t multiplier_, term_t list_ofLists, term_t rows_, term_t columns_, term_t result){
    int i=0;
    int multiplier;
    int rows, columns;

    if(!checkInteger(&multiplier, multiplier_)){
        fprintf(stderr, "La variable multiplier no es válida\n");
        PL_fail;
    }

    if(!checkInteger(&rows, rows_)){
        fprintf(stderr, "La variable rows no es válida\n");
        PL_fail;
    }

    if(!checkInteger(&columns, columns_)){
        fprintf(stderr, "La variable columns no es válida\n");
        PL_fail;
    }

    Matrix_C* matrix = createMatrix(rows, columns);
    if (list_toMatrix(list_ofLists, matrix) == 0){
        fprintf(stderr, "Error pasando la lista a matriz\n");
        PL_fail;
    }

    for (i=0; i<matrix->rows*matrix->columns; i++){
        matrix->matrix[i] = matrix->matrix[i]*multiplier;
    }

    term_t aux = PL_new_term_ref();
    if (!matrix_toList(matrix, aux)){
        fprintf(stderr, "Error pasando la matriz a lista\n");
        PL_fail;
    }
    return PL_unify(aux, result);
}
```

Se llama desde Prolog mediante `mult_matrix_by`.

Predicado `pl_total_matrix`

Este predicado suma todos los números de la matriz. Los argumentos son:

- `term_t list_ofLists`: la matriz
- `term_t rows_`: las filas de la matriz
- `term_t columns_`: las columnas de la matriz
- `term_t result`: el resultado que se devolverá

```c
//pl_total_matrix
//sumar todos los números de la matriz
foreign_t pl_total_matrix (term_t list_ofLists, term_t rows_, term_t columns_, term_t result){
    int i;
    int rows = 0;
    int columns = 0;

    if(!checkInteger(&rows, rows_)){
        fprintf(stderr, "La variable rows no es válida\n");
        PL_fail;
    }

    if(!checkInteger(&columns, columns_)){
        fprintf(stderr, "La variable columns no es válida\n");
        PL_fail;
    }

    //Convertir la lista de listas a Matrix_C
    Matrix_C* matrix = createMatrix(rows, columns);
    if (list_toMatrix(list_ofLists, matrix) == 0){
        fprintf(stderr, "Error pasando la lista a matriz\n");
        PL_fail;
    }

    double sum = 0;

    for (i=0; i<matrix->rows*matrix->columns; i++){
        sum += matrix->matrix[i];
    }

    term_t r = PL_new_term_ref();

    if(PL_put_float(r, sum)){
        return PL_unify(r, result);
    } else {
        PL_fail;
    }
}
```

Se llama desde Prolog mediante `total_matrix`.

Predicado `pl_sum_matrixes`

Este predicado realiza la operación suma sobre dos matrices. Dado que la restricción para sumar dos matrices es que ambas tengan las mismas dimensiones, solo se pide al usuario que introduzca una vez las dimensiones. Los argumentos son:

- `term_t list_ofLists1`: la primera matriz
- `term_t list_ofLists2`: la segunda matriz
- `term_t rows_`: las filas de las matrices
- `term_t columns_`: las columnas de las matrices
- `term_t result`: la matriz resultado que se devolverá

```
//pl_sum_matrixes
//Sumar dos matrices
foreign_t pl_sum_matrixes (term_t list_ofLists1, term_t list_ofLists2, term_t rows_, term_t columns_, term_t result){
    int columns, rows, i;

    if(!checkInteger(&rows, rows_)){
        fprintf(stderr, "La variable rows no es válida\n");
        PL_fail;
    }

    if(!checkInteger(&columns, columns_)){
        fprintf(stderr, "La variable columns no es válida\n");
        PL_fail;
    }

    Matrix_C* matrix1 = createMatrix(rows, columns);
    if (list_toMatrix(list_ofLists1, matrix1) == 0){
        fprintf(stderr, "Error pasando la primera lista a matriz.\nRecuerde que, "
                        "para sumar dos matrices, ambas tienen que tener las mismas dimensiones.\n");
        PL_fail;
    }

    Matrix_C* matrix2 = createMatrix(rows, columns);
    if (list_toMatrix(list_ofLists2, matrix2) == 0){
        fprintf(stderr, "Error pasando la segunda lista a matriz.\nRecuerde que,
                        "para sumar dos matrices, ambas tienen que tener las mismas dimensiones.\n");
        PL_fail;
    }

    Matrix_C* resultMatrix = createMatrix(rows, columns);

    for (i=0; i<resultMatrix->rows*resultMatrix->columns; i++){
        resultMatrix->matrix[i] = matrix1->matrix[i] + matrix2->matrix[i];
    }

    term_t aux = PL_new_term_ref();
    if (!matrix_toList(resultMatrix, aux)){
        fprintf(stderr, "Error pasando la matriz a lista\n");
        PL_fail;
    }
    return PL_unify(aux, result);
}
```

Se llama desde Prolog mediante `sum_matrixes`.

Predicado pl_multiply_matrixes

Este predicado multiplica dos matrices. Los argumentos son:

- `term_t list_ofLists1`: la primera matriz
- `term_t list_ofLists2`: la segunda matriz
- `term_t rows1_`: las filas de la primera matriz
- `term_t columns1_`: las columnas de la primera matriz
- `term_t rows2_`: las filas de la segunda matriz
- `term_t columns2_`: las columnas de la segunda matriz
- `term_t result`: la matriz resultado que se devolverá

```c
//pl_multiply_matrixes
//Multiplicar dos matrices
//multiply_matrixes([[1,2,1],[2,1,2]],[[1, 1],[1,1],[1,1]], 2, 3, 3, 2, R).
foreign_t pl_multiply_matrixes (term_t list_ofLists1, term_t list_ofLists2, term_t rows1_,
        term_t columns1_, term_t rows2_, term_t columns2_, term_t result){
    int columns1, rows1, columns2, rows2;

    if(!checkInteger(&columns1, columns1_)){
        fprintf(stderr, "La variable columns1 no es válida\n");
        PL_fail;
    }

    if(!checkInteger(&rows2, rows2_)){
        fprintf(stderr, "La variable rows2 no es válida\n");
        PL_fail;
    }

    if(columns1 != rows2){
        fprintf(stderr, "No se pueden multiplicar dos matrices si el "
        "número de columnas de la primera no es igual al número de filas de la segunda\n");
        PL_fail;
    }

    if(!checkInteger(&rows1, rows1_)){
        fprintf(stderr, "La variable rows1 no es válida\n");
        PL_fail;
    }


    if(!checkInteger(&columns2, columns2_)){
        fprintf(stderr, "La variable columns2 no es válida\n");
        PL_fail;
    }

    Matrix_C* matrix1 = createMatrix(rows1, columns1);
    if (list_toMatrix(list_ofLists1, matrix1) == 0){
        fprintf(stderr, "Error pasando la primera lista a matriz.\n");
        PL_fail;
    }
```

```c
    Matrix_C* matrix2 = createMatrix(rows2, columns2);
    if (list_toMatrix(list_ofLists2, matrix2) == 0){
        fprintf(stderr, "Error pasando la segunda lista a matriz.\n");
        PL_fail;
    }

    Matrix_C* resultMatrix = createMatrix(rows1, columns2);
    int i, iRow, j, jRow, k, index_1, index_2;

    for(i=0; i<rows1+1; i++){
        for(j=0; j<columns2+1; j++){
            for (k=0; k<rows1+1; k++){
                resultMatrix->matrix[i*columns2+j] += matrix1->matrix[i*columns1+k] *
                    matrix2->matrix[k*columns2+j];
            }
        }
    }


    term_t aux = PL_new_term_ref();
    if (!matrix_toList(resultMatrix, aux)){
        fprintf(stderr, "Error pasando la matriz a lista\n");
        PL_fail;
    }
    return PL_unify(aux, result);
}
```

Se llama desde Prolog mediante `multiply_matrixes`.

Función `pl_help_matrix`

Esta función consiste en unos mensajes que muestran ayuda sobre la biblioteca.

Esta pequeña ayuda incluye los nombres de las funciones y los argumentos que necesitan.

```
//pl_help_matrix()
//ayuda de la librería
foreign_t pl_help_matrix (){
    fprintf(stderr, "Bienvenid@ a la librería Matrilog.\nLas operaciones disponibles
    fprintf(stderr, "  - total_matrix(list_ofLists:list, rows:integer, columns:intege
    fprintf(stderr, "  - mult_matrix_by(multiplier:integer, list_ofLists:list, rows:i
    fprintf(stderr, "  - sum_matrixes(list_ofLists:list, list_ofLists:list, rows:inte
    fprintf(stderr, "  - multiply_matrixes(list_ofLists:list, list_ofLists:list, rows
    PL_succeed;
}
```

Se llama desde Prolog mediante `help_matrix`.

Función `install`

Esta función, como se ha explicado anteriormente, registra los predicados y les asigna un nombre para ejecutarlos desde Prolog.

```
install_t install() {
    PL_register_foreign("mult_matrix_by", 5, pl_multiply_matrix_by, 0);
    PL_register_foreign("total_matrix", 4, pl_total_matrix, 0);
    PL_register_foreign("sum_matrixes", 5, pl_sum_matrixes, 0);
    PL_register_foreign("multiply_matrixes", 7, pl_multiply_matrixes, 0);
    PL_register_foreign("help_matrix", 0, pl_help_matrix, 0);
}
```

5. Conclusiones

Escribir esta librería ha sido tanto desafiante como interesante. Saber cómo se pueden escribir predicados para Prolog en otro lenguaje es muy útil y, en ocasiones como esta, prácticamente necesario, ya que Prolog, al ser un lenguaje declarativo, se puede usar perfectamente para realizar modelos y operaciones matemáticas con matrices. Sin embargo, al no tener ninguna estructura "matriz", estas operaciones se hacen complicadas y tediosas de escribir, al contrario que en otro lenguaje que posea las facilidades de una estructura de datos.

Además, este trabajo me ha ayudado a reforzar tanto mis conocimientos en C como en Prolog, y hasta el momento en el que elegí hacerlo no sabía que era posible escribir librerías para un lenguaje de programación, en otro lenguaje de programación.

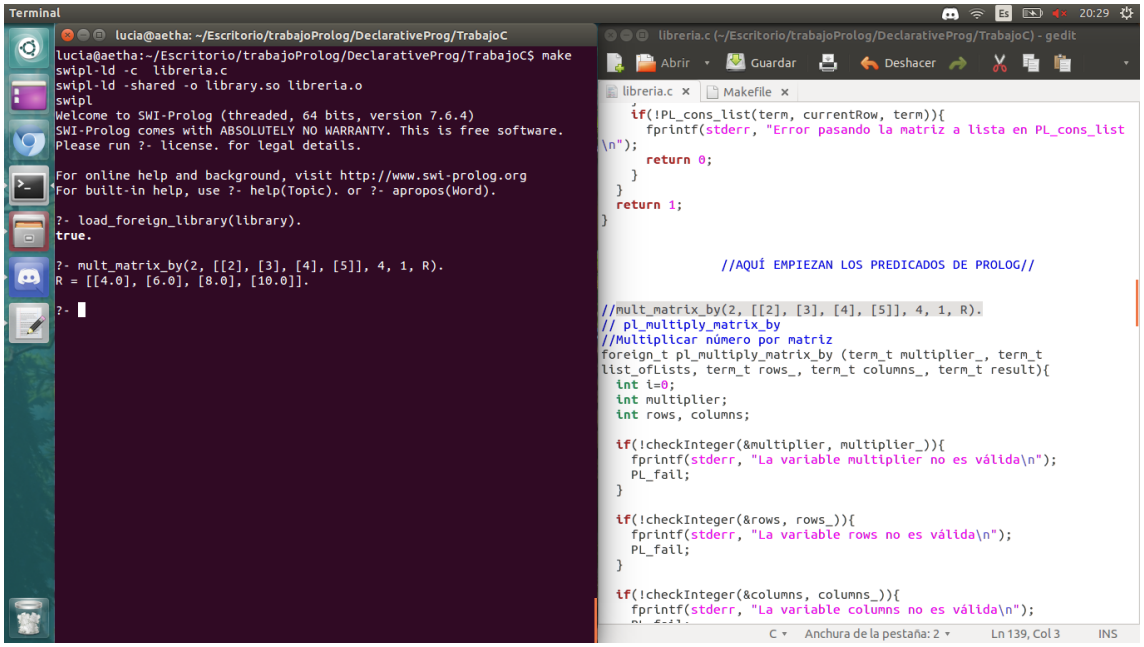
6. Referencias

[1]. Foreign Language Interface – SWIProlog https://www.swi-prolog.org/pldoc/man?section=foreign

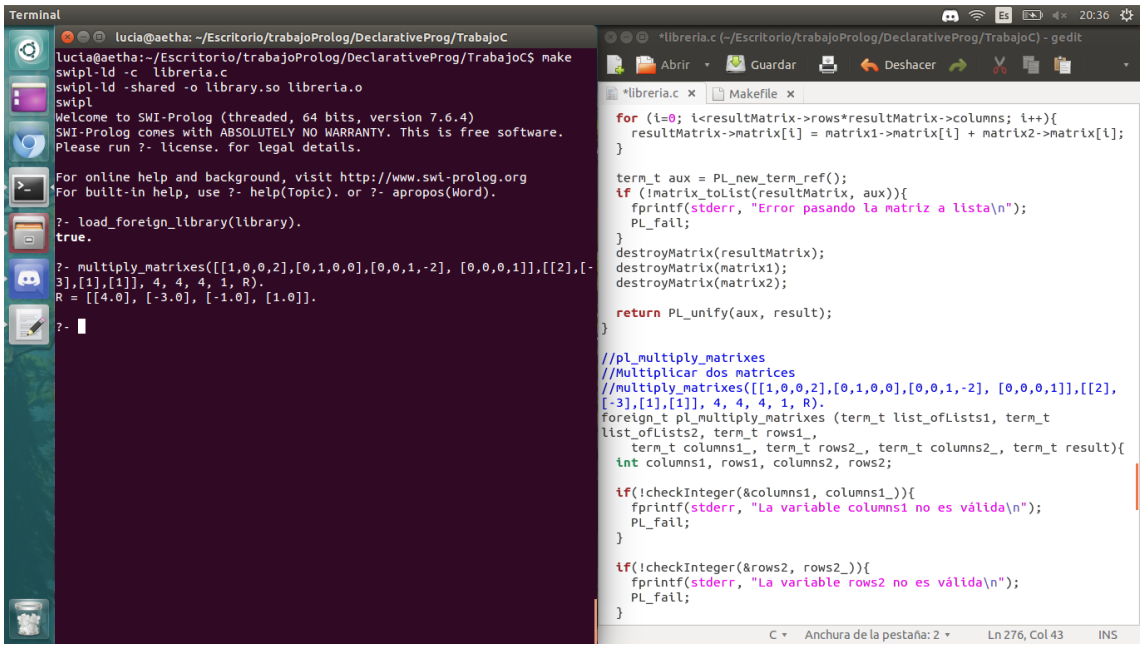
7. Anexo: Ejecuciones

A continuación, se muestran capturas de pantalla con las ejecuciones de los diferentes predicados escritos.

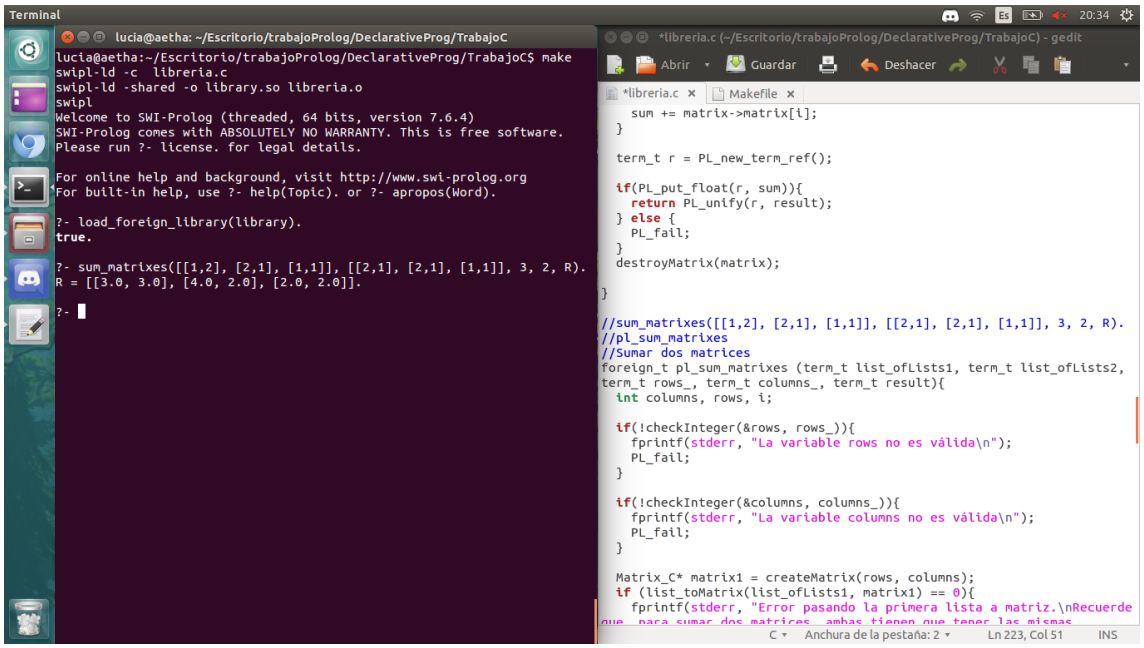
- Multiplicar una matriz por un número: `mult_matrix_by()`



- Multiplicación de dos matrices: `multiply_matrixes()`



- Suma de dos matrices: `sum_matrixes()`



- Sumar todos los elementos de una matriz: `total_matrix()`