Programación Declarativa

Interfaz SWI-Prolog $\longleftrightarrow$ C/C++ (2017/2018)

Javier Córdoba Romero
Juan José Corroto Martín

Índice

1 Introducción

Este documento explica cómo utilizar las interfaces de SWI-Prolog con el lenguaje C[1] y con el lenguaje C++[2], para crear funciones en dichos lenguajes que puedan ser llamados desde un programa Prolog, del mismo modo que se llamaría a un predicado de Prolog y de forma transparente.

Estas funciones y predicados externos se realizan sobre la interfaz que SWI-Prolog nos proporciona, tanto en C como en C++, con una gran cantidad de tipos y funciones para manipular los tipos de Prolog, así como comunicarse con el núcleo de Prolog.

Este documento se centrará especialmente en la librería interfaz con el lenguaje C, y después se hará una comparación menos exhaustiva con la librería interfaz de C++, centrándonos especialmente en sus diferencias, ventajas y desventajas de cada una de ellas.

1.1 Bibliotecas estáticas y dinámicas

Los predicados externos que se quieran utilizar en Prolog deben agruparse en una librería externa. Estas librerías pueden ser:

- **Librería estática**. Las librerías estáticas se enlazan al fichero Prolog en tiempo de compilación, de manera que el archivo resultado de compilar los predicados externos consiste en un ejecutable que contiene tanto a dichos predicados como el núcleo de Prolog. De esta forma todos los predicados externos que hayamos definido en nuestro lenguaje de elección ya forman parte del entorno, y no es necesario cargarlos a posteriori.
- **Librería dinámica**. Las librerías dinámicas se enlazan en tiempo de ejecución, y no de compilación. Esto quiere decir que el resultado de compilar nuestros predicados externos consistirá en un archivo con extensión de librería dinámica (En un Sistema Operativo Linux tendrán la extensión *.so*, y en un Sistema Operativo tendrán extensión *.dll*). Este fichero deberá cargarse desde un entorno de Prolog en funcionamiento con el predicado de SWI-Prolog *load_foreign_library/1*, cuyo argumento es el nombre de la librería.

Desde el punto de vista de la implementación de nuestros predicados externos, no influye la elección del tipo de librería que queramos desarrollar. El código fuente no cambiará.

En este documento se discutirán solo la creación de librerías dinámicas para SWI-Prolog.

1.2 Predicados deterministas y no deterministas

SWI-Prolog permite la implementación de predicados externos deterministas y no deterministas. Los predicados deterministas son aquellos que no permiten puntos de elección y sólo tienen éxito una vez, mientras que los no deterministas permiten varios puntos de elección, y pueden tener éxito al ser reevaluados en otro punto de elección. Sin embargo, nosotros sólo nos centraremos en la implementación de predicados deterministas, puesto que los predicados externos no deterministas son mucho más complejos, puesto que necesitan mantener información del contexto entre las diferentes llamadas al predicado externo para computar todas sus soluciones.

2 Compilación, ejecución y depuración de bibliotecas dinámicas

Antes de mostrar cómo se implementan predicados externos en C y C++, vamos a ver primero el proceso de compilado y enlazado de las librerías generadas.

2.1 C

2.1.1 Compilación

El proceso de compilar una librería dinámica para usarla en `SWI-Prolog` pasa por usar el comando `swipl-l` que por defecto es instalado junto a `SWI-Prolog` cuya función es compilar y enlazar el código que le proporcionemos. Para compilar será necesario ejecutar el siguiente comando

```
swipl-ld -Iinclude/ -c code.c -o out.o
```

En esencia, este comando es una interfaz con `gcc` el cual ayuda a localizar e incluir los archivos de cabecera de `SWI-Prolog`.

Para enlazar los archivos obtenidos de la compilación se usará el siguiente comando

```
swipl-ld -shared -o library.so code1.o code2.o
```

Donde `codeX.o` son todos los archivos compilados usando el comando anterior.

El resultado de ejecutar este comando es un archivo `.so` que utilizaremos para cargar la librería dentro de la línea de comandos de `SWI-Prolog`.

2.1.2 Ejecución

Para ejecutar la librería compilada anteriormente es necesario ejecutar `SWI-Prolog` en el mismo directorio en el que se encuentra la librería y ejecutar el predicado `load_foreign_library(+library)` donde `library` es la ruta de la librería externa ya compilada.

Una vez ejecutado el predicado todos los predicados definidos dentro de la librería están disponibles dentro de `SWI-Prolog` como si de un predicado nativo se tratase.

El predicado `load_foreign_library` se puede incluir en una directiva dentro de un fichero de código `Prolog` para automatizar la carga de la librería.

2.2 C++11

2.2.1 Compilación

En el caso de C++, la compilación se realiza de forma similar. En este caso, es necesario usar el enlazador de SWI-Prolog a la vez del compilador de C++ para que la compilación se realice correctamente.

swipl-ld -shared -ld g++ -cc-options[" -std=c++11"

En este caso se usa la flag -cc-options del enlazador de SWI-Prolog para indicarlo al compilador de C++ que use el estándar C++11, para poder utilizar sus librerías y sus punteros inteligentes.

2.2.2 Ejecución

La ejecución usando la interfaz de C++ es similar a la de C, porque ésta interfaz está basada en la de C. Aunque, como luego veremos, la interfaz de C++ no tenga función de instalación, la macro que se va a usar para definir predicados externos, realiza procesos transparentes al programador que generan dicho función de instalación. Debido a esto, el flujo de ejecución es similar.

2.3 Depuración

Para poder depurar la librería externa se debería compilar el núcleo de `SWI-Prolog` con símbolos de depuración lo cual no es trivial y depende en gran medida del sistema operativo y plataforma usada.

La solución que se ha alcanzado es implementar una funcion de depuración a nivel de preprocesador, similar a la función `printf`.

Esta función imprime por la salida estándar de error el mensaje que le pasemos como argumentos e incluso valores en tiempo de compilación.

De esta manera podemos habilitar y deshabilitar todos los mensajes de depuración cambiando el comando de compilación. Definiendo una macro en tiempo de compilación.

El código que habilitaría/deshabilitaría los mensajes en tiempo de compilación sería el siguiente:

```
#ifdef DEBUG
       #define DEBUG_PRINT(...) dbgprintf(__FILE__, __LINE__, __VA_ARGS__)
#else
       #define DEBUG_PRINT(...) do {} while(0)
#endif
```

La función `dbgprintf` está definida de la siguiente forma:

```
void dbgprintf(const char* source_file, int source_line, const char* format_string
    ↪ , ...)
{
    va_list args;
    va_start(args, format_string);

    fprintf(stderr, "%s:%d: ", source_file, source_line);
    vfprintf(stderr, format_string, args);

    va_end(args);

}
```

De esta forma, para habilitar los mensajes de depuración solo tendríamos que añadir al comando de compilación la opción `-DDEBUG`

La forma de imprimir un mensaje sería llamar a la función `DEBUG_PRINT` como si de otra función de C se tratase.

3 Implementación de predicados externos

En esta sección se presentan las características que deben tener nuestras funciones para ser consideradas predicados externos, además de algunas de las funciones C más importantes de C y C++ para comunicarse con Prolog y manejar sus términos. Más información en los manuales de referencia [1][2].

Aunque no se discutirán todas las operaciones implementadas, el listado completo de todas las operaciones que la librería contiene, tanto en C como en C++, es el siguiente:

- **is_squared(+Matriz)**. Devuelve verdadero si la matriz es cuadrada, y falso en caso contrario.
- **same_dimensions(+Matriz1, +Matriz2)**. Devuelve verdadero si matriz1 y matriz2 tienen el mismo número de filas y columnas, y falso en caso contrario.

- **mat_sum(+Matriz, -sum)**. Devuelve la suma de todos los elementos de la matriz.
- **sum_mat(+Matriz1, +Matriz2, -MatrizResultado)**. Devuelve la matriz resultado de sumar matriz1 con matriz2.
- **res_mat(+Matriz1, +Matriz2, -MatrizResultado)**. Devuelve la matriz resultado de restar matriz1 con matriz2.
- **mult_mat_factor(+Matriz, +factor, -MatrizResultado)**. Devuelve la matriz resultado de multiplicar matriz1 por un número.
- **mult_mat(+Matriz1, +Matriz2, -MatrizResultado)**. Devuelve la matriz resultado de multiplicar matriz1 con matriz2.
- **transpose_mat(+Matriz, -MatrizResultado)**. Devuelve la matriz resultado de transponer la matriz de entrada.
- **determinant_mat(+Matriz, -Determinante)**. Devuelve el determinante de la matriz de entrada, si lo tiene.
- **inv_mat(+Matriz, -MatrizInversa)**. Devuelve la matriz inversa de la matriz de entrada, si la tiene.

3.1 C

3.1.1 Definición de predicados externos

Para definir un predicado externo es necesario definir una función que devuelva un valor del tipo `foreign_t` y acepte valores del tipo `term_t` los cuales son los argumentos del predicado que serán proporcionados dentro del entorno de `SWI-Prolog`.

Cada función que definamos se corresponderá con un predicado externo en `SWI-Prolog`.

La forma de devolver el control a `SWI-Prolog` es usar las macros `PL_succeed` y `PL_fail` que a su vez devuelven las macros `TRUE` y `FALSE`, cuyos valores son 1 y 0.

3.1.2 Función de instalación

Para que los predicados definidos en la librería externa estén disponible en `SWI-Prolog` es necesario definir una función de instalación que registre todos los predicados definidos en el núcleo de `SWI-Prolog`.

Esta función :

- Tiene un tipo de retorno `install_t`.
- No tiene parámetros de entrada.
- El nombre de la función es `install` o `install_` seguido por el nombre de la biblioteca.

La función para registrar los predicados externos es

```
PL_register_foreign(+name, +arity, +function, 0)
```

El significado de estos argumentos son: Esta función :

- **+name** El nombre del predicado con el cual será llamado dentro de `SWI-Prolog`
- **+arity** El número de argumentos del predicado externo, que debe coincidir con el número de argumentos de la función definida en C
- **+function** El nombre de la función en código C, que no tiene por qué coincidir con el nombre del predicado que se usará en `SWI-Prolog`

Un ejemplo de definición de predicados y de función de instalación podría ser la implementación de un predicado que sumase dos números enteros:

```
foreign_t
pl_sum_num(term_t n1, term_t n2, term_t res)
{
        int num1, num2 = 0;

        if (!PL_get_integer(n1, &num1)
                PL_fail;
        if (!PL_get_integer(n2, &num2)
                PL_fail;

        term_t r = PL_new_term_ref();

        return PL_put_integer(res, num1 + num2);
}

install_t
install()
{
        PL_register_foreign("sum_num", 2, pl_sum_num, 0);
}
```

3.1.3 El tipo `term_t`

El tipo de datos `term_t` es el principal tipo de datos en la librería externa de `SWI-Prolog`.

Este tipo de datos representa un puntero a un término de `SWI-Prolog` aunque este puntero no se puede manipular con los operadores habituales de C.

Los valores posibles de `term_t` son:

- Una variable.
- Un átomo.
- La constante `[]`, que representa el final de una lista.
- Datos binarios.
- Una cadena de texto.
- Un valor numérico.
- Un término compuesto
- Una lista.
- Un diccionario.

Las funciones más usadas para manejar términos son:

- **PL_new_term_ref()** Crea una referencia a un nuevo término.
- **PL_new_term_refs(+count)** Crea `count` referencias a un nuevo término, estas referencias se pueden acceder sumando 1,2,3... al primer término.
- **PL_copy_term_ref(+term)** Crea una referencia a un nuevo término el cual apunta inicialmente a `term`.

Una función muy interesante es `PL_copy_term_ref` ya que permite recorrer una lista al estilo de `SWI-Prolog` un ejemplo es:

```
foreign_t
pl_print_list(term_t list)
{
       if (!PL_is_list(list))
              PL_fail;

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

       while(PL_get_list(tail, head, tail))
       {
              char* str;
              PL_get_atom_chars(head, &str);

              printf("%s, ", str);
       }

       PL_succeed;
}
```

3.1.4 Análisis y construcción de términos

En la librería externa de `SWI-Prolog` existen diversas funciones que, permiten conocer qué tipo de variable está almacenado en una variable `term_t` y que permiten relacionar esas variables con los tipos de C.

3.1.4.1 Comprobación de tipos

La librería cuenta con una serie de funciones que permiten conocer el tipo de variable que contiene una variable `term_t`. Algunos de estos métodos son:

- `PL_is_variable(+term)`
- `PL_is_atom(+term)`
- `PL_is_number(+term)`
- `PL_is_list(+term)`
- `PL_is_compound(+term)`
- `PL_is_string(+term)`

3.1.4.2 Términos simples

La librería externa cuenta con funciones para obtener valores de los términos de `SWI-Prolog`, algunas de estas funciones son:

- `PL_get_atom_chars(+term, -char_pointer)`
- `PL_get_integer(+term, -int_pointer)`
- `PL_get_float(+term, -double_pointer)`

También podemos usar funciones para guardar tipos de datos de C en un término de `SWI-Prolog`, algunas de estas funciones son:

- **`PL_put_variable(-term)`:** Que guarda una variable no inicializada en el término.

- **PL_put_atom_chars(-term, +char_pointer):** Que guarda una cadena de caracteres en el término.
- **PL_put_float(-term, +double):** Que guarda un número de punto flotante en el término.

Un ejemplo de operaciones con estas funciones sería la de sumar todos los elementos numéricos de una lista

```
foreign_t
pl_sum_list(term_t list, term_t result)
{
	if (!PL_is_list(list))
		PL_fail;

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

	double sum = 0;

	while(PL_get_list(tail, head, tail))
	{
		double d = 0;
		int i = 0;
		if (!PL_is_number(head))
			continue;

		if (PL_is_integer(head))
		{
			PL_get_integer(head, &i);
			d = (double) i;
		}
		else
		{
			PL_get_float(head, &d);
		}

		sum += d;
	}

	term_t r = PL_new_term_ref();
	PL_put_float(result, sum);
}
```

3.1.4.3 Listas

En la librería también existen funciones para crear, modificar y analizar listas de `SWI-Prolog`, algunas de estas funciones son:

- **PL_get_list(+list, -head, -tail):** Que guarda en dos términos la cabeza y la cola de la lista correspondiente.
- **PL_get_nil(+list):** Comprueba si el término representa la constante de final de lista.
- **PL_cons_list(-list, +head, +tail):** Construye una lista nueva formada por la unión de la cabeza y la cola. La lista objetivo y la cola pueden ser el mismo término, lo que ayuda a construir listas en un bucle.
- **PL_skip_list(+list, -tail, -length):** Permite conocer la longitud de la lista y, a la vez, obtener un término que hace referencia al último elemento de la lista.

3.1.5 Otras funciones

. Otra función interesante dentro de la librería es la posibilidad de unificar dos términos de la misma forma que se produce la unificación en `SWI-Prolog`

Esta función es `PL_unify(term1, term2)` que intenta unificar term2 con term1.

Para ejemplificar este apartado junto con el de listas se implementará un predicado externo que construya una lista de los que contenga todos los números naturales que hay entre los dos argumentos pasados al predicado, la lista resultante se unificará para que el predicado tenga también un propósito de comprobar si una lista genérica es la lista de elementos consecutivos entre dos números:

```
foreign_t
pl_gen_list(term_t low, term_t high, term_t list)
{
	if (!PL_is_integer(low) || !PL_is_integer(high))
		PL_fail;

	int l, h;

	PL_get_integer(low, &l); // Get the numeric values of the terms
	PL_get_integer(high, &h);

	if (l > h || l < 0) // Check if the range is valid
		PL_fail;

	term_t res = PL_new_term_ref();
	PL_put_nil(res); // Put the empty list constant in the new term

	term_t num = PL_new_term_ref();

	for (int i = l; i <= h; i++)
	{
		PL_put_integer(num, i); // Put the numeric value on the term
		PL_cons_list(list, num, list); // Build the list tail-to-head
	}

	return PL_unify(list, res); // Unify with the list
}
```

3.2 C++11

3.2.1 Definición de predicados externos

Al igual que en C, una biblioteca externa puede tener tantos predicados externos como se desee. Cada uno de ellos estará asociado a un bloque de código. Ésta es la primera diferencia que tenemos con respecto a la interfaz con C, nuestros predicados no serán funciones C++, si no bloques de código. Sin embargo, el flujo de ejecución es el mismo que en la interfaz de C: después de cargar la librería dinámica mediante el predicado *load_foreign_library/1*, e invocar uno de los predicados externos, se cede el control al bloque de código C++, que al finalizar, devuelve el control a Prolog.

Los bloques de código de C++ asociados a un predicado externo estarán marcados mediante la macro PREDICATE(name, arity). En esta macro se establece:

- El nombre del predicado que se invocará en Prolog.
- La aridad, es decir, el número de argumentos que tomará el predicado.

Con esta macro ya hemos realizado toda la instalación del bloque de código, y ya puede ser llamado desde Prolog. En realidad, la macro lo que está haciendo por nosotros es envolver todo el boque de código designado mediante una función con tipo de retorno *foreignt_t*, con el número de argumentos que le hemos establecido de tipo PlTerm (más adelante aclararemos qué es este tipo), y también genera la función de instalación para este bloque. Esto quiere decir que no tenemos que preocuparnos por ninguno de estos temas cuando usemos dicha macro.

Estos bloque de código deberán retornar siempre TRUE, si el predicado ha tenido un resultado exitoso, o FALSE si ha fallado.

Además, podemos acceder a los argumentos de nuestro predicado mediante las macros A_n, siendo n el número del argumento que accedemos. Por ejemplo, si queremos acceder al primer argumento, podemos acceder a él mediante A_1, variable de tipo PlTerm.

Ejemplo de predicado externo en el que se devuelve FALSE si la matriz pasada como argumento no es cuadrada, y TRUE si sí lo es:

```
PREDICATE(is_squared, 1)
{
	std::unique_ptr<Matrix> m1(list_to_Matrix(A1));

	if (m1.get() == nullptr || !m1->isSquared())
	{
		return FALSE;
	}

	return TRUE;
}
```

Algunas de las funciones que aparecen en el ejemplo serán discutidas más adelante (como por ejemplo *list_to_Matrix*).

3.2.2 El tipo `PlTerm`

En la interfaz de C++, que recordamos es totalmente compatible con la interfaz de C, tenemos a nuestra disposición un tipo más abstracto de datos que nos permite manipular términos de Prolog. Éste es el tipo *PlTerm*, una clase definida en la interfaz de C++. Las macros de nuestros argumentos A_n son de este tipo.

Los tipos *PlTerm* contienen una referencia a un término de prólog, al igual que el tipo *term_t*, pero nos da un nivel mayor de abstracción que más tarde discutiremos.

En caso de que queramos construir un nuevo término de Prolog, podemos hacerlo mediante múltiple constructores a nuestra disposición, entre los que se incluyen:

- PlTerm::PlTerm(). Crea una nueva referencia a un término de Prolog sin inicializar.
- PlTerm::PlTerm(*term_t*. Con el que podemos convertir un tipo de término de la interfaz de C a un tipo de la interfaz C++.
- PlTErm::PlTerm(double d). Con el que podemos crear una referencia a un término de prolog que contiene el número en punto flotante d.

Además también tenemos el tipo *PlTermv*, que consiste en un vector de *PlTerm*. Este tipo tiene sobrecargado el operador ( ) para poder acceder a cualquier de los términos que contiene, empezando en 0. Sin embargo, este tipo es principalmente usado para pasar argumentos en llamadas desde C++ a Prolog. Esto será discutido más adelante.

Una de las cosas más importante que ofrece la clase *PlTerm* es la sobrecarga del operador =. En el caso de que usemos el operador = con un *PlTerm*, no estaremos asignando, sino unificando un valor con dicho término. Por lo tanto la operación = devuelve un valor booleano verdadero si la

unificación fue un éxito, o uno falso si la unificación falló. Esto es importante porque, al igual que en Prolog, una vez unificado un término dentro de nuestro predicado, no se puede volver a unificar con algo distinto. En este caso devolvería un valor falso y el bloque de código continuaría.

A continuación un ejemplo de creación de términos, así como uno de unificación.

```
PlTerm term1(); //Crea una referencia vacía a un término

PlTerm term2(2.0); //Crea una referencia al término 2.0
```

```
PREDICATE(mat_sum, 2)
{
    std::unique_ptr<Matrix> m1(list_to_Matrix(A1));

        if (m1.get() == nullptr)
        {
                return FALSE;
        }

        return A2 = m1->sum(); //Devolveremos verdadero si el término A2 se puede
            unificar con la suma de la matriz m1
}
```

3.2.3 Casteo de tipos y excepciones

Una de las cosas más potentes que nos ofrece la interfaz de C++ con respecto a la de C es el casteo de tipos. Esto consiste en convertir uno de nuestros términos *PlTerm* en una valor de C++, como por ejemplo un valor en punto flotante. En el siguiente ejemplo se puede ver una instrucción perfectamente válida.

```
PREDICATE(sum, 2)
{

        double value = (double) A1 + 2.0;

        PlTerm ret_term(value);

        return A2 = ret_term;
}
```

Sin embargo, los términos no tienen por qué ser un valor en punto flotante, por lo que el casteo fallaría. En este caso, las excepciones entran en juego. En el caso anterior, si llamaramos al predicado sum/2 , siendo el primer argumento a, la instrucción $(double)A1$ lanzaría una excepción de tipo *PlTypeError*, excepción que se lanza cuando cualquier casteo falla. La interfaz además implementa más tipos de excepciones:

- PlTypeError: excepción lanzada cuando un termino no satisface el tipo esperado por Prolog. Como se ha definido antes, es el tipo que se lanza en el casteo fallido, pero también en otras llamadas a la interfaz que veremos más adelante.
- PlDomainError: excepción lanzada cuando un termino satisface el tipo esperado de Prolog, pero no es aceptable para el dominio esperado por una operación. Por ejemplo, la llamada al predicado de Prolog open/3 espera un io_mode(leer, escribir, etc). Si se pasa como argumento un entero, es un error de tipo y se lanzaría la excepción PlTypeError. Sin embargo, si se pasa un átomo distinto a los io_mode, se lanza una PlDomainError.

- PlException: Esta subclase de PlTerm actúa como clase padre de las demás excepciones, y nos permite definir nuestras propias excepciones. También la podemos usar como excepción general para capturar las excepciones lanzadas por Prolog en un bloque try-catch.

Es importante conocer que todas estas excepciones son lanzadas desde C++, y por tanto pueden ser controladas desde el mismo bloque de instrucciones que forma el predicado externo.

Ahora procedemos a realizar el mismo ejemplo que antes, pero esta vez cerciorándonos que controlamos todas los posibles escenarios.

```
PREDICATE(sum, 2)
{
        try{
                double value = (double) A1 + 2.0;
        }
        catch (PlException &ex)
        {
                std::cout << "El primer termino debe ser un numero" << std::endl;
                return FALSE;
        }
        PlTerm ret_term(value);

        return A2 = ret_term;
}
```

En este nuevo predicado, si el usuario introduce como primer argumento un término que no sea un número, el casteo lanzará una excepción que capturaremos. Le comunicamos al usuario el error y devolvemos false, en lugar de dejar que Prolog cese su ejecución.

3.2.4 El tipo `PlTail`

La clase que la interfaz nos ofrece para manipular listas de prolog es *PlTail*. Este clase nos permite asignar una referencia una lista e iterar sobre ella. Algo importante a tener en cuenta es que el constructor:

```
PlTail list(term);
```

Admite como parámetro un término del tipo PlTerm, pero este término debe ser ya una lista. Esto quiere decir que la interfaz no nos permite construir nuevas listas.

La clase PlTail contiene las siguientes operaciones:

- PlTail::append(PlTerm &term). Añade un término a la lista, poniéndose este nuevo término como nueva cabeza de la lista.
- PlTail::close(). Ésta función intenta unificar la lista con la lista vacía, devolviendo verdadero si la unificación ha tenido éxito, y falso en caso contrario.
- PlTail::next(PlTerm &term). Éste método nos permitirá iterar sobra la lista, asignando al término que pasamos como argumento una referencia a la cabeza de la lista, y haciendo que *PlTail* apunte al siguiente elemento. Esto quiere decir que cuando iteramos sobre la lista, estamos consumiendo los elementos, por lo que al final la variable PlTail quedará apuntando a una lista vacía.

A continuación se presenta un ejemplo característico del uso del tipo PlTail.

```
PlTail list(term);
```

```
PlTerm e1;

while (list.next(e1)) {
       content.push_back((double) e1);
}
```

En el ejemplo anterior, se puede ver como se construye una lista a partir de un término, y después se itera sobre la lista, asignando la referencia de la cabeza de la lista en cada iteración a un término sin inicializar. Después casteamos dicho término a un número en punto flotante y lo agregamos a un vector.

Es importante tener en cuenta que si, en el ejemplo anterior, el término *term* no fuese una lista, el constructor lanzaría una PlTypeError (excepción de tipo).

3.2.5 Llamadas a predicados Prolog

La interfaz de C++ también nos permite hacer llamadas a predicados de Prolog desde C++, pudiéndose así establecer una conversación entre sendos lenguajes.

Para esto, la interfaz nos da la clase *PlQuery*, que nos permite definir una consulta a un predicado Prolog, así como obtener sus soluciones.

los distintos constructores para la clase son los siguientes:

- PlQuery::PlQuery(const char* name, PlTermv &av). Crea una consulta al predicado de nombre *name*, y le pasa como argumentos aquellos del vector *av*. La aridad del predicado que llamamos es deducida del vector de términos *av*, por lo que no tenemos que decírsela.
- PlQuery::PlQuery(const char* module, const char* name, PlTermv &av). Realiza la misma construcción que el anterior, pero esta vez podemos indicarle el nombre del módulo en el que el predicado se encuentra.

La función que utilizaremos para obtener soluciones es $PlQuery :: next\_solution()$, que llama al predicado y obtiene en el vector de argumentos la siguiente solución, si la hay.

A continuación se presenta un ejemplo típico de uso:

```
PREDICATE(average, 3)
{ long sum = 0;
  long n = 0;

  PlQuery q("call", PlTermv(A2));
  while( q.next_solution() )
  { sum += (long)A1;
    n++;
  }
  return A3 = (double)sum/(double)n;
}
```

En este ejemplo utilizamos el predicado de orden superior call/1, de tal modo que podemos calcular la media de las soluciones que el segundo argumento, un predicado, nos de.

3.2.6 Construcción de una lista

Aunque se ha discutido anteriormente, es importante resaltar que la interfaz de C++ no nos ofrece la posibilidad de construir nuevas listas. Esto es debido a que la clase de la que disponemos para manipular listas, *PlTail*, admite en su constructor un término que ya sea una lista. Para solventarlo, hemos aprovechado la compatibilidad tanto de los lenguajes C y C++ como de las interfaces de dichos lenguajes, y hemos invocado a la interfaz de C. De este modo, conseguimos un término $term\_t$ que

contiene la lista, y éste es el que usamos para construir nuestro PlTerm que vamos a unificar con la solución.

Las funciones utilizadas para la creación de esta lista ya han sido discutidas en la parte de C de este documento.

4 Ejemplos

En esta sección se va a presentar un ejemplo concreto y concreto de la biblioteca dinámica que hemos desarrollado. El ejemplo constará de los siguientes predicados externos:

- **sum_mat(+Matriz1, +Matriz2, -MatrixResultado)**. Suma cada uno de los componentes de ambas matrices, y nos devuelve el resultado en una nueva matriz. Ejemplo:

```
?- sum_mat([[1,1],[2,2]], [[3,3],[1,1]], [A, B]).
A = [4.0, 4.0],
B = [3.0, 3.0].
```

- **mult_mat(+Matriz1, +Matriz2, -MatrixResultado)**. Realiza la multiplicación de matriz1 y matriz2 y nos devuelve el resultado en una nueva matriz. Ejemplo:

```
?- mult_mat([[1,3],[2,4]], [[1],[1]], X).
X = [[4.0], [6.0]].
```

Además de estas operaciones, se han implementado multitud más de operaciones con matrices que se pueden encontrar en el código fuente adjunto, tanto en C como en C++.

El mayor esfuerzo ha sido situado en la comprobación de tipos, de tal modo que los predicados sean capaces de admitir cualquier tipo de término de Prolog y lograr una respuesta, aunque se evalúe el predicado a falso, en vez de cortar la ejecución de Prolog.

4.1 C

Debido a la naturaleza de bajo nivel de C la lógica de la librería dinámica está mezclada con las comprobaciones de errores y el tratamiento de memoria dinámica, aun así se podrían distinguir 4 partes bien diferencias en un predicado externo:

- **Comprobación-Conversión:** De los tipos de `SWI-Prolog` a tipos y estructuras de C.
- **Obtención de memoria dinámica:** Donde se guardará la matriz resultado.
- **Obtención:** Del resultado del predicado mediante operaciones definidas previamente
- **Conversión:** De tipos de datos C a valores de `SWI-Prolog`.
- **Liberación de memoria dinámica:** Para prevenir las fugas de memoria.

4.1.1 sum_mat

El código que realiza la suma de matrices es:

```
int add_matrices(const struct Matrix_t* m1, const struct Matrix_t* m2, struct
    ↪ Matrix_t* result)
{
    if (!m1)
        return 0;
    if (!m1->size)
        return 0;
    if (!m1->rows)
```

```
        return 0;

    if (!m2)
        return 0;
    if (!m2->size)
        return 0;
    if (!m2->rows)
        return 0;

    if (!result)
        return 0;
    if (!result->size)
        return 0;
    if (!result->rows)
        return 0;


    int same_dims = 0;
    if (!is_same_dimensions(m1, m2, &same_dims))
        return 0;

    if (!same_dims)
        return 0;

    for (int i = 0; i < result->size[0]; i++)
    {
        for (int j = 0; j < result->size[1]; j++)
        {
            result->rows[i][j] = m1->rows[i][j] + m2->rows[i][j];
        }
    }

    return 1;
}
```

El código que realiza la traducción SWI-Prolog $\longleftrightarrow$ C:

```
foreign_t
pl_add_matrices(term_t m1, term_t m2, term_t result)
{
        struct Matrix_t* matrix1;
        struct Matrix_t* matrix2;
        struct Matrix_t* mresult = NULL;

        matrix1 = list_to_matrix(m1);
        if (!matrix1)
                PL_fail;

        matrix2 = list_to_matrix(m2);
        if (!matrix2)
        {
                free_matrix(matrix1);
                PL_fail;
        }


        mresult = alloc_matrix(matrix1->size);
```

```
    if (!mresult)
    {
            free_matrix(matrix1); free_matrix(matrix2);
        PL_fail;
    }

        if (!add_matrices(matrix1, matrix2, mresult))
        {
            free_matrix(matrix1); free_matrix(matrix2); free_matrix(mresult);
        PL_fail;
        }

        term_t mlist = PL_new_term_ref();
        if (!matrix_to_list(mresult, mlist))
        {
            free_matrix(matrix1); free_matrix(matrix2); free_matrix(mresult);
        PL_fail;
        }

        if (!free_matrix(matrix1) | !free_matrix(matrix2) | !free_matrix(mresult))
            PL_fail;

        return PL_unify(result, mlist);
}
```

4.1.2 mult_mat

El código que realiza la multiplicación de matrices es:

```
int multiply_matrices(const struct Matrix_t* m1, const struct Matrix_t* m2, struct
    ↪  Matrix_t* result)
{
    if (!m1)
        return 0;
    if (!m1->size)
        return 0;
    if (!m1->rows)
        return 0;

    if (!m2)
        return 0;
    if (!m2->size)
        return 0;
    if (!m2->rows)
        return 0;

    if (!result)
        return 0;
    if (!result->size)
        return 0;
    if (!result->rows)
        return 0;


    if (m1->size[1] != m2->size[0])
    {
```

```
        return 0;
    }

    for (int i = 0; i < result->size[0]; i++)
    {
        for (int j = 0; j < result->size[1]; j++)
        {
            double res = 0;
            for (int k = 0; k < m1->size[1]; k++)
            {
                res += m1->rows[i][k] * m2->rows[k][j];
            }
            result->rows[i][j] = res;
        }
    }

    return 1;
}
```

El código que realiza la traducción SWI-Prolog $\longleftrightarrow$ C:

```
foreign_t
pl_multiply_matrices(term_t m1, term_t m2, term_t result)
{
        struct Matrix_t* matrix1;
        struct Matrix_t* matrix2;
        struct Matrix_t* mresult = NULL;

        matrix1 = list_to_matrix(m1);
        if (!matrix1)
        {
                PL_fail;
        }

        matrix2 = list_to_matrix(m2);
        if (!matrix2)
        {
                free_matrix(matrix1);
                PL_fail;
        }

        size_t nsize[2] = {0};
        nsize[0] = matrix1->size[0];
        nsize[1] = matrix2->size[1];
        mresult = alloc_matrix(nsize);
    if (!mresult)
    {
                free_matrix(matrix1); free_matrix(matrix2);
        PL_fail;
    }

        if (!multiply_matrices(matrix1, matrix2, mresult))
        {
                DEBUG_PRINT("Cannot multiply matrices\n");
                free_matrix(matrix1); free_matrix(matrix2); free_matrix(mresult);
        }
```

```
        term_t mlist = PL_new_term_ref();
        if (!matrix_to_list(mresult, mlist))
        {
               free_matrix(matrix1); free_matrix(matrix2); free_matrix(mresult);
        PL_fail;
        }

        if (!free_matrix(matrix1) | !free_matrix(matrix2) | !free_matrix(mresult))
        {
               PL_fail;
        }

        return PL_unify(result, mlist);
}
```

4.1.3 Matrix struct

A continuación se detalla la la estructura que se usa para representar una matriz de `nxn` elementos:

```
struct Matrix_t
{
    size_t size[2]; // Matrix dimensions

    double** rows; // Matrix rows
};
```

4.1.4 Funciones auxiliares

4.1.5 SWI-Prolog $\longrightarrow$ C

Para convertir una lista de `SWI-Prolog`, que representa una fila, en un array de C contamos con la siguiente funcion:

```
int list_to_row(const term_t list, size_t length, double* row)
{
    if (!row)
        return 0;

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

    for (int i = 0; i < length; i++)
    {
        if (!PL_get_list(tail, head, tail))
            return 0;

        int ival;
        double dval;
        switch (PL_term_type(head))
        {
            case PL_INTEGER:
                if (!PL_get_integer(head, &ival))
                    return 0;
```

```
                *(row + i) = (double) ival;
                break;

            case PL_FLOAT:
                if (!PL_get_float(head, &dval))
                    return 0;

                *(row + i) = dval;
                break;

            default:
                return 0;
        }
    }

    return 1;
}
```

4.1.6 SWI-Prolog ⟵ C

Para convertir un array de C, que representa una fila, en una lista de `SWI-Prolog` contamos con la siguiente funcion:

```
int row_to_list(double* row, size_t length, term_t list)
{
    if (!row)
        return 0;

    PL_put_nil(list);

    term_t num = PL_new_term_ref();

    for (int i = length - 1; i >= 0; i--)
    {
        int res = PL_put_float(num, row[i]);
        if (res != TRUE)
            return 0;

        res = PL_cons_list(list, num, list);
        if (res != TRUE)
            return 0;
    }
    return 1;
}
```

También tenemos funciones que no convierten los tipos, si no que manejan la memoria que se asigna y se libera para mejorar la modularización y a prevenir errores relaciones con la gestión de la memoria:

la siguiente función reserva memoria para una matriz de tamaño `size`:

```
struct Matrix_t* alloc_matrix(const size_t size[2])
{
    if (!size)
        return NULL;

    if (size[0] < 1 || size[1] < 1)
```

```
        return 0;

    struct Matrix_t* mat = calloc(1, sizeof(struct Matrix_t));
    if (!mat)
        return NULL;

    memcpy(mat->size, size, sizeof(size_t) * 2);


    int rows = size[0];
    int cols = size[1];

    double** rptr = (double**) calloc(rows, sizeof(double*));

    if (!rptr)
    {
        free(mat);
        mat = NULL;
        return NULL;
    }
    else
    {
        mat->rows = rptr;
    }

    for (int i = 0; i < rows; i++)
    {
        double* cptr = (double*) calloc(cols, sizeof(double));
        if (!cptr)
        {
            for (int j = 0; j < i; j++)
            {
                free(*(mat->rows + i));
                *(mat->rows + i) = NULL;
            }
            free(rptr);
            rptr = NULL;
            free(mat);
            mat = NULL;
            return NULL;
        }
        else
        {
            *(mat->rows + i) = cptr;
        }
    }

    return mat;
}
```

La siguiente función libera de la memoria un puntero a una estructura que almacena una matriz, es notable la comprobación para evitar liberar la misma dirección de memoria varias veces: :

```
int free_matrix(struct Matrix_t* matrix)
{
    if (!matrix)
        return 0;
```

```
    if (!matrix->size)
        return 0;

    for (int i = 0; i < matrix->size[0]; i++)
    {
        if (*(matrix->rows + i))
        {
            free(*(matrix->rows + i));
            *(matrix->rows + i) = NULL;
        }
    }

    free(matrix->rows);
    matrix->rows = NULL;

    free(matrix);
    matrix = NULL;

    return 1;
}
```

4.2 C++

En la librería dinámica de C++, se ha logrado una gran modularización de las responsabilidades, de tal manera que todos los predicados constan de tres partes bien diferenciadas:

- **Conversión** los tipos de Prolog en una clase definida en C++ de matrices.
- **Obtención** de matriz resultado (u obtención de resultado en caso de que sea un predicado que devuelva un número), mediante operaciones propias de C++, utilizando la clase definida en C++.
- **Contrucción** de una nueva lista de Prolog a partir de la matriz resultado.

Esto provoca que el código de todos los predicados sea prácticamente igual, salvo por la operación a realizar con las matrices.

4.2.1 sum_mat

```
PREDICATE(sum_mat, 3)
{
    std::unique_ptr<Matrix> m1(list_to_Matrix(A1));
    std::unique_ptr<Matrix> m2(list_to_Matrix(A2));

    if (m1.get() == nullptr || m2.get() == nullptr)
    {
        return FALSE;
    }

    //Get new matrix from the sum

    std::unique_ptr<Matrix> m_result(new Matrix(m1->getrows(), m1->getcols(), *m1 +
        ↪ *m2));

    if (!m_result->isValid())
    {
```

```
        return FALSE;
    }

    //C code to build the new list

    term_t nlist = PL_new_term_ref();

    matrix_to_list(nlist, m_result.get());

    //End of C code

    //Unify result

    PlTerm result(nlist);

    return A3 = result;
}
```

4.2.2 mult_mat

```
PREDICATE(mult_mat_factor, 3)
{
    std::unique_ptr<Matrix> m(list_to_Matrix(A1));

    if (m.get() == nullptr)
    {
        return FALSE;
    }

    double factor;

    try
    {
        factor = (double) A2;
    }
    catch(...)
    {
        std::cout << "Second argument must be a double" << std::endl;
        return FALSE;
    }

    std::unique_ptr<Matrix> m_result(new Matrix(m->getrows(), m->getcols(), *m *
        ↪ factor));

    if (!m_result->isValid())
    {
        return FALSE;
    }

    //C code to build the new list

    term_t nlist = PL_new_term_ref();

    matrix_to_list(nlist, m_result.get());

    //End of C code
```

```
    //Unify result

    PlTerm result(nlist);

    return A3 = result;
}
```

4.2.3 Matrix.hpp

A continuación se detalla el archivo de cabecera de la clase definida para almacenar una matriz. La implementación de cada una de las operaciones está contenida en el archivo Matrix.cpp.

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>
#include <limits>
#include <math.h>

class Matrix
{
public:
    Matrix(int rows, int cols, std::vector<double> content);
    Matrix(int rows, int cols);

    int getcols() const;
    int getrows() const;
    std::vector<double> getcontent() const;

    int addRow(std::vector<double> row);

        const double sum() const;
        std::vector<double> transpose() const;
        double determinant() const;
        std::vector<double> inverse() const;

        double adjoint(int row, int col) const;

    double operator()(size_t i, size_t j) const;
    std::vector<double> operator+(const Matrix m2) const;
    std::vector<double> operator-(const Matrix m2) const;
    std::vector<double> operator*(const double factor) const;
    std::vector<double> operator*(const Matrix m2) const;

    void getrow(int row, double retval[]) const;

    int isValid() const;
        int isSquared() const;
        int same_dimensions(const Matrix m2) const;

        void print() const;
private:
    std::vector<double> content;
    const int cols;
    const int rows;
```

```
};
```

4.2.4 Funciones auxiliares

A continuación se detallan las funciones que nos ayudan a convertir los tipos de Prolog en tipos de C++ y viceversa.

Las siguientes dos funciones transformarán términos de Prolog en instancias de nuestra clase Matrix.

```
Matrix* list_to_Matrix(PlTerm term)
{
    int rows = 0, cols = -1;
    int i = 0, j;
    std::vector<double> content;
    try {
        PlTail list(term);


        PlTerm e1;

        while (list.next(e1)) {
            j = 0;
            PlTerm e2;

            try
            {
                PlTail list_row(e1);
                while (list_row.next(e2)) {
                    content.push_back((double) e2);
                    j++;
                }
                //Check if matrix is well formed
                if (cols != -1 && cols != j)
                {
                    throw std::runtime_error(0);
                }
            }
            catch (PlTypeError &ex)
            {
                return list_to_matrix_one_row(term);
            }
            cols = j;
            i++;
        }
    }
    catch (...)
    {
        std::cout << "Invalid matrix" << std::endl;
        return nullptr;
    }

    rows = i;
    cols = j;

    return new Matrix(rows, cols, content);
}
```

```
Matrix* list_to_matrix_one_row(PlTerm term)
{
    int rows = 0, cols = 0;
    int j = 0;
    std::vector<double> content;
    try {
        PlTail list(term);


        PlTerm e1;

        while (list.next(e1)) {
            content.push_back((double) e1);
            j++;
        }
    }
    catch (...)
    {
        std::cout << "Invalid matrix" << std::endl;
        return nullptr;
    }

    rows = 1;
    cols = j;

    return new Matrix(rows, cols, content);
}
```

Es importante resaltar el empleo de la función *list_to_matrix_one_row*. Ésta se llama cuando nos salta una excepción en la función *list_to_matrix* y sabemos que estamos en la primera fila. Ésto se debe a que, si el usuario introduce una matriz fila (o lo que es lo mismo, una lista de Prolog), al recorrer la lista estamos esperando a que cada uno de esos elementos sea a su vez otra lista, porque el caso general es el de una matriz bidimensional. Si estos elementos no son listas y son elementos atómicos, el constructor de *PlTail* lanza una excepción de tipo, que controlamos para llamar a la función de construcción de matrices en caso de que sea una lista de una dimensión.

La siguiente función realiza el proceso contrario: recibe una referencia a una matriz definida en C++ y la transforma en una lista de Prolog. Aquí es donde hemos tenido que utilizar llamadas a la interfaz de C.

```
//Function written in C and using C interface
//Receives a prolog list and a Matrix
//Converts the reference of the list into the matrix
int matrix_to_list(term_t list, Matrix* matrix)
{
    size_t rows = matrix->getrows();
    size_t cols = matrix->getcols();

    term_t lists = PL_new_term_refs(rows);

    for (int i = 0; i < rows; i++)
    {
        double row_from_matrix[cols];
        matrix->getrow(i, row_from_matrix);
        if (!row_to_list(row_from_matrix, cols, lists + i))
        {
```

```
            return 0;
        }
    }

    PL_put_nil(list);

    for (int i = rows - 1; i >= 0; i--)
    {
        if(!PL_cons_list(list, lists + i, list))
        {
            return 0;
        }
    }

    return 1;
}

//Function written in C and using C interface
//Receives a matrix row and a prolog list, needs the length of the row
//Converts the list into the matrix row
int row_to_list(double *row, size_t length, term_t list)
{
    //Taken from matlog.c
    if (!row)
    {
        return 0;
    }

    PL_put_nil(list);

    term_t num = PL_new_term_ref();

    for (int i = length - 1; i >= 0; i--)
    {
        int res = PL_put_float(num, row[i]);
        if (res != TRUE){
            return 0;
        }

        res = PL_cons_list(list, num, list);
        if (res != TRUE)
        {
            return 0;
        }
    }

    return 1;
}
```

4.2.5 Sesión de ejemplo

```
?- load_foreign_library(matlog).
true.

?- sum_mat([[1,2,3],[1,2,3],[1,2,3]], [[1,1,1],[1,1,1],[1,1,1]], X).
X = [[2.0, 3.0, 4.0], [2.0, 3.0, 4.0], [2.0, 3.0, 4.0]].

?- sum_mat([[1,2,3],[1,2,3],[1,2,3]], [[1,1,1],[1,1,1]], X).
Incompatible dimensions: Trying to add 3 3 with 2 3
false.

?- mult_mat([[1,0,0,1],[0,1,0,0],[0,0,1,1],[0,0,0,1]], [[3],[4],[1],[1]],X).
X = [[4.0], [4.0], [2.0], [1.0]].

?- mult_mat([[1,0,0,1],[0,1,0,0],[0,0,1,1],[0,0,0,1]], [1,1,1,1],X).
Incompatible dimensions: Trying to multiplicate 4 4 with 1 4
false.

?- inv_mat([[1,0,1],[3,1,4],[2,1,0]], X).
X = [[1.3333333333333333, -0.3333333333333333, 0.3333333333333333], [-2.66666666
66666665, 0.6666666666666666, 0.3333333333333333], [-0.3333333333333333, 0.33333
33333333333, -0.3333333333333333]].

?- sum_mat([[1,1],[2,2]], [[2,2],[3,3]], [[3.0,3.0],[5.0,5.0]]).
true.
```

5 Referencias

[1] Foreign Language interface to C.

[2] Foreign Language interface to C++.

```
try {//In case the PlTail constructor fails
        PlTail list(term);
        PlTerm e1;

    while (list.next(e1)) { //Iterate over the list
       j = 0;
       PlTerm e2;
           PlTail list_row(e1);
           while (list_row.next(e2)) {
               content.push_back((double) e2);
               j++;
                }
    }
catch (...)
{
   std::cout << "Invalid matrix" << std::endl;
   return nullptr;
}
```