Notación formalizada para la representación de árboles de seguimiento de algoritmos en Prolog

Nieves Pavón Pulido Omar Sánchez Pérez
Departamento de Ingeniería Electrónica Sistemas Informáticos y Automática
Palos de la Frontera, Huelva.
Universidad de Huelva
email: npavon@diesia.uhu.es
omar@diesia.uhu.es

Resumen

En el siguiente artículo se presenta un método de diseño y representación formal de árboles de ejecución de programas lógicos escritos en Prolog. Este método se ha aplicado con éxito en el desarrollo de ejercicios en una asignatura de programación declarativa que se imparte en tercer curso de Ingeniería Técnica en Informática de Gestión en la Universidad de Huelva. Además, se ha desarrollado una herramienta que permite generar automáticamente estos árboles de ejecución con la notación formalizada para la comprobación sencilla de ejercicios que consistan en un seguimiento del funcionamiento de código fuente escrito en Prolog.

1. Introducción

En la Universidad de Huelva, los alumnos de Ingeniería Técnica en Informática de Gestión se enfrentan al paradigma declarativo lógico en tercer curso [1].

Acostumbrados a las técnicas de programación imperativa, la comprensión de las técnicas declarativas lógicas de programación resultan verdaderamente complicadas.

Cuando se enseña una metodología de programación pueden seguirse los siguientes pasos:

- Descripción de aspectos teóricos.
- Descripción de cómo realizar el diseño del código fuente.
- Descripción de cómo realizar el seguimiento y depuración del código fuente.

El primer paso es, obviamente, el más sencillo. En el segundo paso se adquiere destreza cuando se han realizado muchos ejemplos y ejercicios. Por tanto, se puede asegurar que la realización de un número elevado de ejercicios es fundamental para comprender una técnica de programación. Sin embargo, el segundo paso no está completo si para cada programa diseñado no aplicamos el conjunto de operaciones que se exponen en el tercer paso, es decir, el seguimiento detallado del conjunto de instrucciones implementado.

En un lenguaje imperativo, el seguimiento de un programa consiste en la escritura del resultado de un conjunto de instrucciones secuenciales o encapsuladas en estructuras de control de sobra conocidas como *"repetir"*, *"si"*, *"caso de"*, etc., para las cuales ya existen técnicas de descripción y representación formal, sin embargo, para los lenguajes declarativos lógicos el proceso de seguimiento no es trivial, pues consiste en la generación de un árbol de ejecución cuyos nodos son originados por los procesos de recursión y backtracking intrínsecos a la implementación de este tipo de programas.

En los puntos siguientes del artículo se muestra un método para formalizar la representación de estos seguimientos, de modo que a los alumnos de la asignatura de Programación Declarativa les resulte sencillo realizarlos y comparar los resultados obtenidos con los expuestos por el profesor.

2. Descripción de elementos de un programa Prolog.

Un programa en Prolog consta de varios predicados que se implementan mediante un conjunto de cláusulas de tipo "*hecho*" o "*regla de inferencia*".

Los hechos se pueden definir como estructuras de la forma:

nombreclausula(arg1, arg2, ... , argN).

Donde "*nombreclausula*" describe la relación que existe entre los argumentos *arg1*, *arg2*, ... , *argN*.

Las reglas de inferencia se pueden definir como estructuras de la forma:

nombreclausula(arg1, arg2, ... , argN):-
c1(args), c2(args), ... ,cN(args).

Donde "*nombreclausula*" describe la relación que existe entre los argumentos *arg1*, *arg2*, ... , *argN*, y las cláusulas de la parte derecha del predicado deben ser ejecutadas (demostradas), para obtener el resultado de la estructura "*nombreclausula*".

La ejecución de un programa Prolog se basa en la realización de dos procesos diferenciados pero fuertemente relacionados entre sí [3][4]:

- *Unificación*: Se intenta demostrar que un objetivo planteado es exactamente igual a un hecho o a la parte izquierda de una regla de inferencia, en todos sus componentes. En el caso de una regla de inferencia, además de que el objetivo planteado se unifique con la parte izquierda de la regla, es necesario que todas las llamadas a las cláusulas situadas a la derecha de la regla puedan ser unificadas.
- *Backtracking*: Si no es posible unificar un objetivo con una cláusula de tipo hecho o parte izquierda de una regla de inferencia, el sistema, automáticamente, intenta buscar la solución con la cláusula siguiente del conjunto que implementa un determinado predicado. Si la llamada a una cláusula de la parte derecha de una regla de inferencia no se puede demostrar, se intenta ejecutar de nuevo la llamada a la cláusula inmediatamente anterior a la misma, de modo que la nueva solución obtenida permita satisfacer la llamada a la cláusula que no pudo ser demostrada.

El modo de ejecución de un programa Prolog genera, no un conjunto secuencial de resultados de operaciones, sino un árbol de intentos de operaciones de unificación [5]. Cuando un objetivo se puede demostrar con varias cláusulas, se fija lo que se denomina punto de backtracking, es decir, un lugar al que podemos regresar para buscar una solución alternativa en el caso de que la cláusula que se ejecuta posteriormente no pueda ser demostrada. Se observa, además, que en Prolog no se dispone de estructuras de control, tales como *"repetir"* o *"si"*, entre otras. El control lo lleva el propio intérprete del código. El único modo de *controlar* dicho control es mediante dos predicados: !(*predicado corte*) y *fail*.

El primero no produce ninguna operación visible para el usuario salvo que elimina la posibilidad de hacer backtracking con alguna cláusula anterior (borrado del punto de backtracking más próximo). El segundo produce un resultado falso (no demostrado), es decir, que falla siempre.

Para comprender todo lo expuesto anteriormente, veamos el siguiente bloque de código que implementa dos relaciones de parentesco: "*padre*" y "*abuelo paterno*":

```
padre("juan", "pedro").
padre("pedro", "pepe").
abuelo(X,Y):-
padre(X,Z),padre(Z,Y).
```

El programa implementa el siguiente conocimiento:

- *Juan es padre de Pedro y éste, a su vez, es padre de Pepe.*
- Por otro lado, la relación abuelo se define mediante la demostración de la sentencia siguiente: *si X es padre de Z y Z es padre de Y entonces X es abuelo de Y.*

Si se especifica como objetivo la cláusula:

```
abuelo("juan", Y).
```

Se intenta que el programa Prolog obtenga como resultado en la variable *Y* al nieto de "*juan*", es decir, *Y="pepe"*.

Si se especifica como objetivo la cláusula:

```
abuelo("juan", "alvaro").
```

El programa Prolog da como resultado *no*, ya que no es posible encontrar ningún nieto de *Juan* que se llame *Álvaro*.

Si se utilizase este ejemplo como ejercicio en la asignatura de Programación Declarativa no tendría valor pedagógico ya que es posible obtener la solución razonadamente sin llevar a cabo el seguimiento del código. Por tanto, es necesario

que el alumno, no sólo exponga la solución final del programa, sino que también escriba el árbol completo de ejecución del mismo. Sin embargo, si estamos realizando un ejercicio tipo test [2], la única pregunta que parece razonable es, simplemente, *¿cuál es el valor final en la variable Y?*

Puede suceder que si se pide el desarrollo del árbol en un ejercicio no de tipo test el alumno realice su propia versión, que aunque correcta, puede diferir mucho de la del profesor, con lo que al profesor puede resultarle difícil comprender la notación establecida por el alumno y viceversa.

De ahí la necesidad de establecer una notación formal que solucione la problemática expuesta.

3. Notación formal para construir árboles de seguimiento

En el seguimiento de un programa en Prolog nos interesa conocer la solución final del algoritmo y comprender como se ha conseguido dicha solución.

Por tanto, en un árbol que represente el seguimiento de un programa Prolog deben aparecer los siguientes elementos:

1. Caja de texto para la especificación de la cláusula que se está ejecutando. (En el caso de que se trate de una regla de inferencia provocará el procesamiento de varios subobjetivos que quedarán representados mediante cajas).
2. Líneas de seguimiento hacia adelante.
3. Líneas de seguimiento del retroceso cuando se hace backtracking.
4. Líneas de valores devueltos por los procesos de backtracking o recursión.
5. Puntos de backtracking activos.
6. Puntos de backtracking borrados.

La figura 1 muestra el elemento descrito en el punto 1.

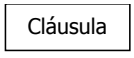

Figura 1. Representación de la ejecución de una cláusula

La figura 2 muestra los tres tipos de líneas que podemos encontrar, ya descritas en los puntos 2, 3 y 4.

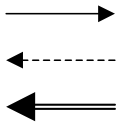

Figura 2. Representación de líneas de flujo de la información

La figura 3 muestra un punto de backtracking activo y un punto de backtracking borrado [5].

Figura 3. Representación de puntos de backtracking

Si se aplica el diseño formalizado al ejemplo descrito en el apartado anterior para el caso de la meta:

```
abuelo("juan", Y).
```

Queda el árbol de seguimiento que se muestra en la figura 4.

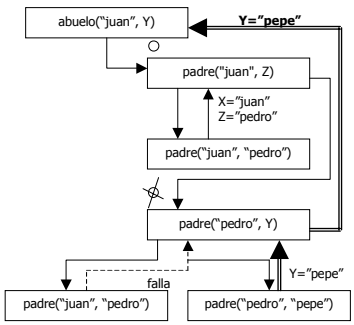

Figura 4. Árbol formalizado de seguimiento de un programa Prolog

Puede observarse que cuando se ejecuta la cláusula *abuelo("juan", Y)*, se generan dos subobjetivos que se han de satisfacer secuencialmente:

padre(X,Z)
padre(Z,Y)

Como *padre(X,Z)* puede resolverse mediante dos cláusulas, se ha de establecer un punto de backtracking. La solución es hallada con la utilización de la primera cláusula, luego el punto de backtracking permanece activo por si se necesita más adelante.

Tras la ejecución del primer subobjetivo planteado, es decir, *padre(X,Z)*, es necesario satisfacer el segundo subobjetivo *padre(Z,Y)*. Como en el caso anterior, se fija un punto de backtracking ya que el subobjetivo puede ser solucionado mediante dos cláusulas posibles. La primera de ellas falla, luego se utiliza el punto de backtracking y se intenta con la segunda y última cláusula. En este caso no quedan más cláusulas para nuevos intentos, por lo que el punto de backtracking se elimina. El subobjetivo queda satisfecho mediante el uso de esta segunda cláusula, por lo que el resultado final, almacenado en Y, se eleva hacia la raíz del árbol.

A través de este ejemplo podemos observar que debido a que en el gráfico se usa una notación formal que comparten tanto los alumnos como el profesor es fácil para ambos comparar resultados, realizar preguntas y especificar respuestas acerca del mismo.

A continuación se muestra un conjunto de preguntas tipo test que se puede elaborar teniendo en cuenta la representación formal del árbol de ejecución:

- ¿Cuántos puntos de backtracking se han generado en total durante el proceso de ejecución? Respuesta: 2 puntos de backtracking.
- ¿Cuántos puntos de backtracking se borran en total durante el proceso de ejecución? Respuesta: 1 punto de backtracking.
- ¿Cuántas veces se evalúa alguna de las cláusulas que componen el predicado padre? Respuesta: 3 veces.

La utilización de este método en la evaluación de esta clase de ejercicios en un examen tipo test revela los conocimientos del alumno de forma más exhaustiva que la simple elección del valor final devuelto por el programa que, sin duda, resulta evidente con un sencillo razonamiento, sin necesidad de llevar a cabo la ejecución del algoritmo.

Si el ejercicio se utiliza en un examen no de tipo test, donde se pide al alumno especificar manualmente el árbol, la notación formalizada proporciona al profesor un alto grado de comodidad en el momento de la corrección.

Además, en el proceso de construcción del árbol, el alumno nota como el intérprete Prolog lleva el control del algoritmo y puede considerar de una forma más inteligente el modo de optimizar el código mediante el uso del predicado corte o, simplemente, evitar errores de ejecución por el uso de puntos de backtracking que sobran.

La figura 5 muestra un árbol de backtracking que permite realizar el seguimiento del algoritmo definido a continuación.

```
fact(0,1).
fact(N,R):-
N1=N-1, fact(N1,R1), R=N*R1.
prueba(N,R):-fact(N,R), R=5.
```

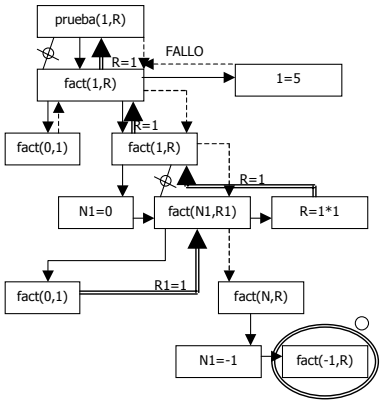

Figura 5. Árbol de ejecución para la meta prueba(1,R)

Como puede observarse, cuando *R=5* falla, regresamos al punto de backtracking que queda activo para buscar una nueva solución del factorial. Sin embargo, esta estrategia de control es errónea ya que el factorial es un problema determinista que genera una solución única para cada objetivo. Si se intenta buscar una nueva solución para el factorial, el algoritmo entra en un proceso de recursión infinito divergente del caso trivial. El alumno puede darse cuenta, a través del gráfico, de que es necesario eliminar el punto de backtracking que queda activo tras la demostración de la cláusula *fact(0,1)*. La solución, por tanto, consiste en modificar la cláusula:

```
fact(0,1).
```

sustituyéndola por:

```
fact(0,1):-!.
```

La figura 6 muestra el árbol de ejecución cuando se elimina el punto de backtracking conflictivo mediante la llamada al predicado corte.

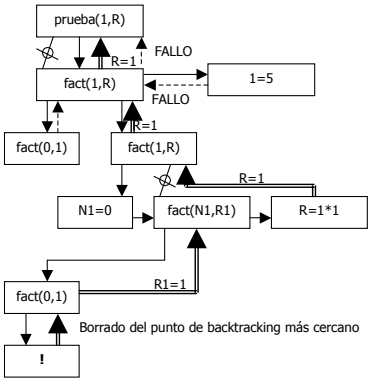

Figura 6. Árbol de ejecución para mostrar el resultado de aplicar el corte

La tarea de escribir correctamente los cortes es difícil ya que, en muchas ocasiones, no resulta obvia. Puede observarse que el gráfico es de gran ayuda en esta circunstancia.

4. Simulador de árboles de seguimiento

El método de representación formal de árboles de ejecución expuesto en los puntos anteriores se ha utilizado con éxito durante los dos últimos cursos impartidos.

Sin embargo, dicho método no resulta completo aún, puesto que el alumno no dispone de una herramienta de generación automática de árboles que le permita comparar su solución con la solución correcta para un determinado ejercicio, cuando el profesor no está presente para llevar a cabo esta corrección de forma personal.

Para solucionar este problema se ha implementado una herramienta software que genera el árbol de ejecución de un algoritmo escrito en Prolog.

El simulador ha sido construido por un alumno de Proyecto Fin de Carrera que, previamente, estudió la asignatura en tercer curso de carrera.

4.1. Características generales del simulador

El simulador es un programa que funciona en las siguientes plataformas: Windows 9X, Millenium y XP.

La aplicación consta de:

- Un editor de texto que permite la escritura de un algoritmo sencillo en Prolog.
- Un corrector de la sintaxis del algoritmo escrito.
- Una ventana para definir la meta u objetivo que se desea satisfacer.
- Una ventana para mostrar el árbol generado, con posibilidad de realizar operaciones para guardarlo o imprimirlo.

La figura 7 muestra el aspecto de la ventana principal de la aplicación. En el editor de texto está escrito el algoritmo expuesto en el apartado 3.

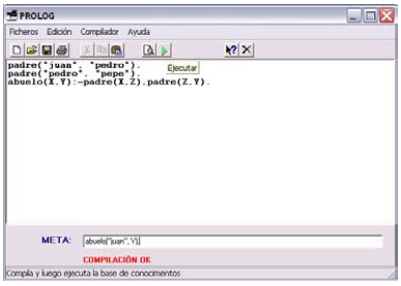

Figura 7. Pantalla principal del simulador

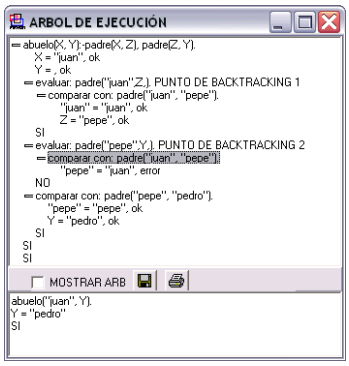

Figura 8. Diálogo que muestra un árbol de ejecución

La figura 8 muestra el diálogo que permite la visualización y las operaciones de "guardar" e "imprimir" el árbol de ejecución obtenido para el algoritmo descrito en el apartado 3.

Puede observarse, además, que el simulador permite operaciones de expansión y contracción de los nodos del árbol para facilitar su estudio.

4.2. Fase de implantación del simulador en la asignatura

Dado que el programa de generación de árboles de ejecución ha sido fruto del desarrollo que un alumno ha realizado durante su proyecto fin de carrera aún no ha sido posible implantar el sistema en la asignatura de Programación Declarativa.

Se están planeando dos formas de introducir el simulador en la asignatura:

- En las horas de prácticas, como una herramienta más a utilizar en el diseño e implementación de las soluciones incorporadas a las hojas de problemas prácticos a resolver.
- En las horas de teoría, como apoyo en la corrección de ejercicios. Para ello, sería conveniente que el profesor utilizase un ordenador y un cañón de vídeo para impartir sus clases, de modo que, para ejercicios propuestos por él mismo, o para dudas planteadas por los alumnos, pueda utilizar el simulador durante la clase teórica proyectando los resultados en tiempo real a modo de transparencia.

5. Conclusiones

La principal conclusión que se obtiene de todo lo expuesto anteriormente es que el uso de una estrategia formal de representación de árboles de ejecución de algoritmos en Prolog supone una mejora en la calidad de la docencia de la asignatura de Programación Declarativa.

Esta mejora se obtiene a partir de las ventajas que conlleva el uso de la notación formalizada:

- Permite una mayor versatilidad a la hora de proponer ejercicios para exámenes tipo test.
- Unifica el método de representación de árboles de seguimiento, de modo que profesor y alumnos utilizan unas normas comunes en la escritura de estos árboles, facilitando la compresión de las soluciones aportadas para los ejercicios propuestos.
- Facilita la comprensión de predicados como el *corte* y el *fail*.
- Permite al alumno conocer de forma exhaustiva el método de ejecución de un algoritmo escrito en Prolog.

Se ha comprobado experimentalmente que, desde que se está usando este método, los alumnos comprenden mejor el funcionamiento del intérprete Prolog.

6. Líneas Futuras

Las líneas futuras están encaminadas hacia la mejora de la aplicación de generación automática de árboles de seguimiento.

Se pretende que, en el curso que viene, la herramienta se encuentre disponible para su uso, principalmente, en las sesiones teóricas de la asignatura.

Agradecimientos

Al alumno Antonio Villarán Carrellán, autor de la aplicación de generación automática de árboles de ejecución de algoritmos en Prolog, por el eficiente trabajo realizado para su proyecto fin de carrera.

Referencias

[1] Pavón, N. *Actas de las VII Jornadas de Enseñanza Universitaria de Informática*, JENUI 2001, pp. 329-334, Palma de Mallorca, 2001.

[2] Pavón, N. et all. *Actas de las VIII Jornadas de Enseñanza Universitaria de Informática*, JENUI 2001, pp. 231-236, Cáceres, 2002.

[3] Sterling & Shapiro. *The Art of Prolog*. MIT Press, 1994.

[4] Adarraga & Zaccagnini. *Psicología e Inteligencia Artificial*. Editorial Trotta, 1994.

[5] Prolog Development Center A/S. *Language Tutorial. Visual Prolog version 5.X*. Copenhagen, Denmark.