Tarea Chica 4: pregunta 2

Diego Andai

7 de junio de 2016

1. Descripción

En esta pregunta se busca fundamentar sobre la mejor forma de implementar la relación de ancestro en el lenguaje ProLog. Para esto se usará el siguiente archivo:

```
1  progenitor(a,b).
2  progenitor(c,d).
3
4  ancestro(X,Y) :- progenitor(X,Y).
5
6  ancestro(X,Y) :- ancestro(X,Z),
7                   progenitor(Z,Y).
8
9  ancestro(X,Y) :- progenitor(X,Z),
10                  ancestro(Z,Y).
11
12 ancestro(X,Y) :- ancestro(X,Z),
13                  ancestro(Z,Y).
14
```

Para esto analizaremos principalmente lo finito de una ejecución, es decir, en que momento el programa se da cuenta que ya no quedan más ancestros y termina su ejecución.

2. Dos formas fallidas

2.1. Forma 1

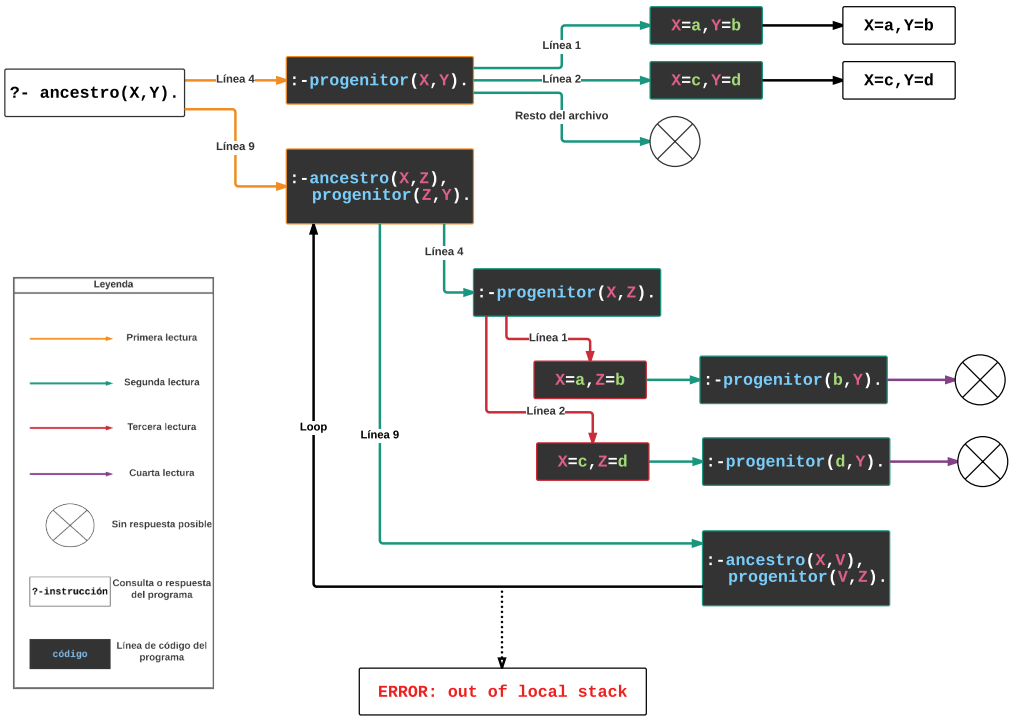
Estudiamos primero la implementación 1, o sea el siguiente código:

```
1  progenitor(a,b).
2  progenitor(c,d).
3
4  ancestro(X,Y) :- progenitor(X,Y).
5
6  ancestro(X,Y) :- ancestro(X,Z),
7                   progenitor(Z,Y).
8
9  %ancestro(X,Y) :- progenitor(X,Z),
10 %                  ancestro(Z,Y).
11
12 %ancestro(X,Y) :- ancestro(X,Z),
13 %                  ancestro(Z,Y).
14
```

Si corremos este programa, la respuesta que nos entrega al preguntar por todos los ancestros existentes es:

```
?- ancestro(X,Y).
X = a,
Y = b ;
X = c,
Y = d ;
ERROR: Out of local stack
```

Notamos que, después de entregar dos respuestas correctas, el programa tiene una falla, quedándose sin espacio en memoria. Para analizar la razón de esto, nos apoyamos en el siguiente gráfico:



Como podemos ver, los pasos que sigue el programa son los siguientes:

1. Encuentra en la línea 4 (amarillo) que para probar ancestro(X,Y), es suficiente probar progenitor(X,Y). Hace entonces una segunda lectura del archivo (verde) encontrando las líneas 1 y 2 que prueban progenitor(X,Y), y por lo tanto entrega esas respuestas. No encuentra otra forma de probar progenitor (X,Y) y por lo tanto vuelve a la lectura 1 (amarillo).

2. Encuentra en la línea 9 que otra forma de probar ancestro(X,Y) es probar simultáneamente ancestro(X,Z),progenitor(Z,Y). Para esto primero busca probar ancestro(X,Z), lo que lo lleva a probar progenitor(X,Z) y que finalmente no termina en ninguna respuesta (ver figura para detalle completo). Hay otra forma de probar ancestro(X,Z), que es la que causa el problema, ya que vuelve al inicio de este ítem: vuelve a encontrar la relación de la línea 9, y para probarla vuelve a hacer este desarrollo, creando un loop infinito y una posterior falla.

Este loop es entonces la causa de que la forma 1 no sea la correcta.

2.2. Forma 3

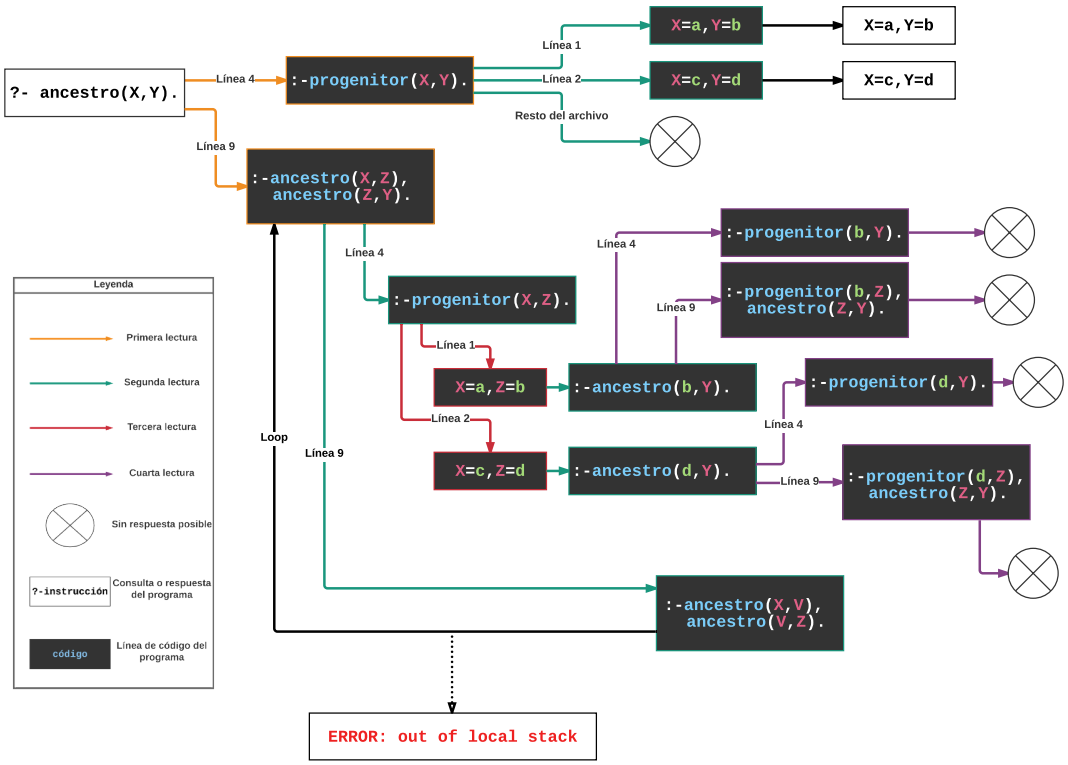
Pasamos ahora a la tercera forma propuesta, que, como nos daremos cuenta, presenta un error muy parecido al anterior. El código es:

```
1  progenitor(a,b).
2  progenitor(c,d).
3
4  ancestro(X,Y) :- progenitor(X,Y).
5
6  %ancestro(X,Y) :- ancestro(X,Z),
7  %                 progenitor(Z,Y).
8
9  %ancestro(X,Y) :- progenitor(X,Z),
10 %                 ancestro(Z,Y).
11
12 ancestro(X,Y) :- ancestro(X,Z),
13                  ancestro(Z,Y).
14
```

De nuevo corremos el programa, y la respuesta que nos entrega al preguntar por todos los ancestros existentes esta vez es:

```
?- ancestro(X,Y).
X = a,
Y = b ;
X = c,
Y = d ;
ERROR: Out of local stack
```

De nuevo, podemos graficar la ejecución de la siguiente manera:



Nos damos cuenta entonces que esta forma también entra en un loop, solo que esta vez prueba con un par de demostraciones antes (ver el gráfico para más detalle). El principal problema de estas implementaciones es que recaen en una definición que se define a si misma, es como intentar buscar el valor de **x** diciendo *'x es x'*. En este ejemplo, quiero saber x, entonces encuentro que x es x, ahora necesito saber que es x para saber que es x, y de nuevo encuentro que x es x, por lo tanto necesito saber que es x para saber que es x para saber que es x, y así sucesivamente.

3. La forma correcta

Buscamos entonces una implementación que tenga la capacidad de notar cuando ya no hay más posibilidad de respuestas y en ese momento deje de buscarlas.

3.1. Forma 2

El código que buscamos es:

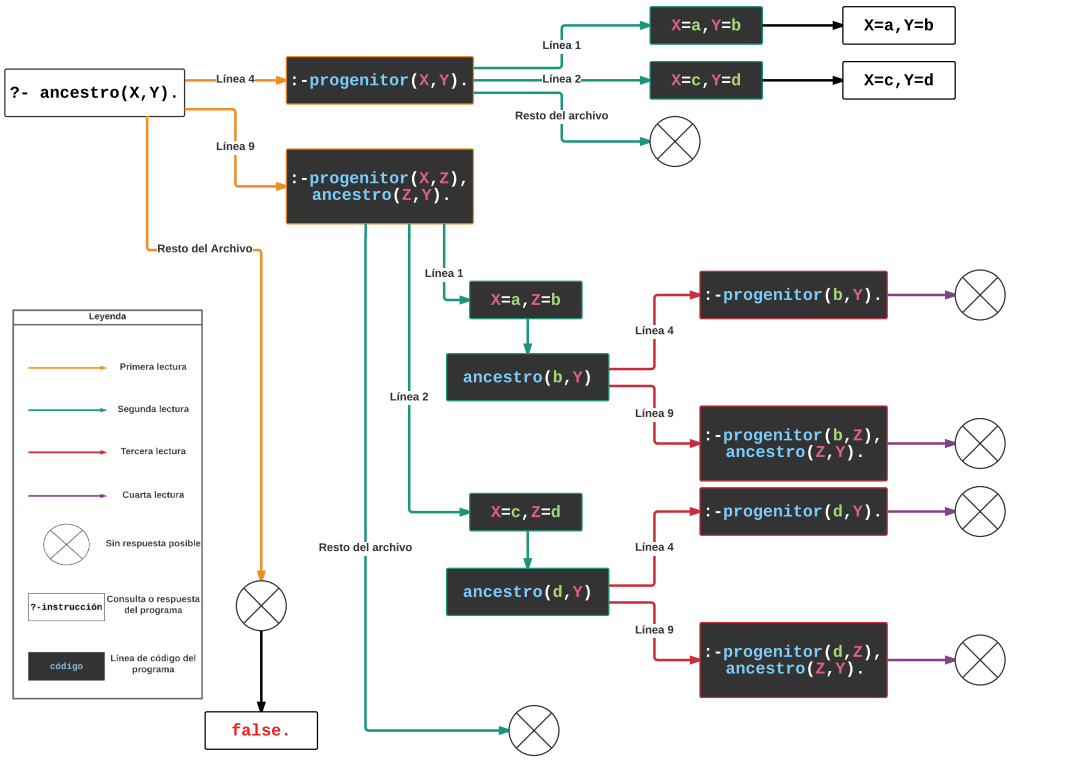
```
1  progenitor(a,b).
2  progenitor(c,d).
3
4  ancestro(X,Y) :- progenitor(X,Y).
5
6  %ancestro(X,Y) :- ancestro(X,Z),
7  %                 progenitor(Z,Y).
8
9  ancestro(X,Y) :- progenitor(X,Z),
10                  ancestro(Z,Y).
11
12 %ancestro(X,Y) :- ancestro(X,Z),
13 %                 ancestro(Z,Y).
14
```

Este nos entrega la siguiente respuesta:

```
?- ancestro(X,Y).
X = a,
Y = b ;
X = c,
Y = d ;
false.

?-
```

La ejecución la podemos graficar de la siguiente manera:



Como podemos notar, cada intento de demostración de este código termina en una respuesta, esta puede ser X e Y tales que cumplen ancestro(X,Y) o bien puede ser la respuesta vacia. Ya que esta ejecución efectivamente llega al final de la primera lectura, podemos decir que representa un algoritmo finito y por lo tanto es la mejor de las tres implementaciones.

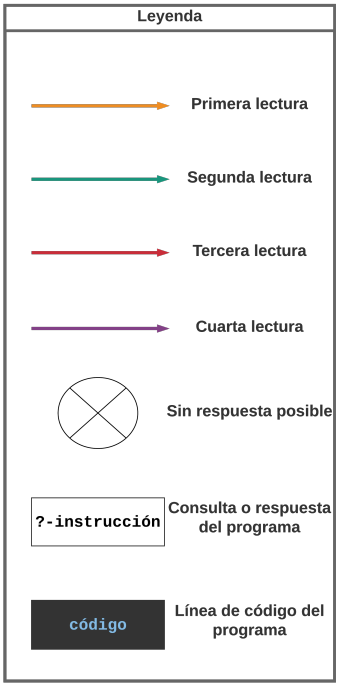
4. Conclusión

Inferimos de este análisis que al implementar reglas recursivas en el lenguaje Prolog, tenemos que referir al principio de las demostraciones a las reglas más fundamentales, y nunca poner como primera regla una demostración que se defina a si misma. En este caso la regla fundamental es progenitor(X,Y) ya que:

- No puede existir un ancestro si no hay progenitor.
- esta no se define a si misma, si no que se explicita al principio del archivo, por lo tanto tiene una cantidad finita de maneras en las que se puede probar.

5. Notas sobre los gráficos

Algunas observaciones sobre los gráficos, referirse a la leyenda a continuación para entender los símbolos.

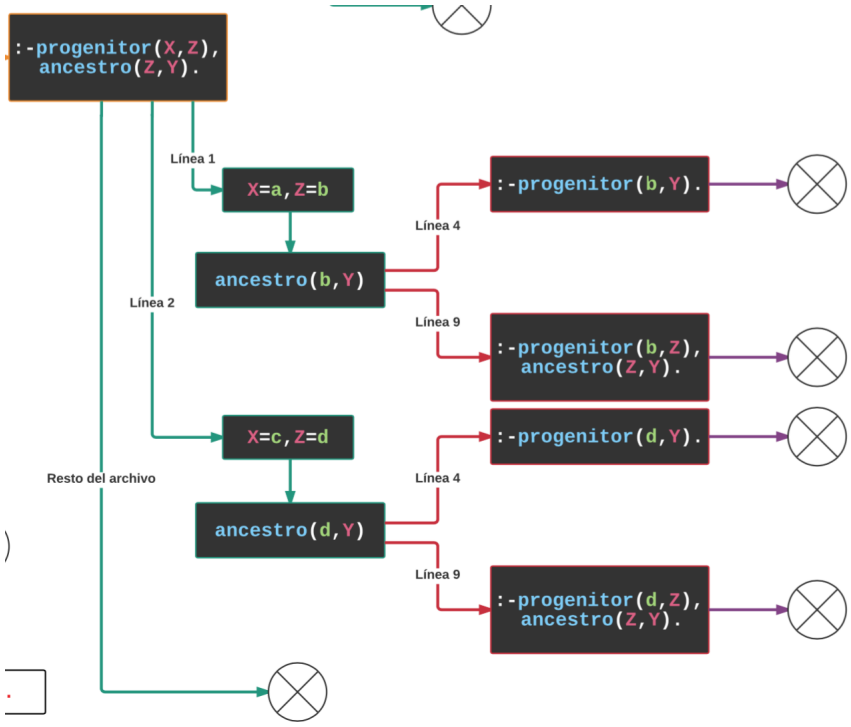


Cuando una lectura termina y no encuentra respuesta posible (*Sin respuesta posible*) se vuelve a la lectura anterior para seguir la ejecución. Cuando esto ocurre en la primera lectura, ha finalizado la ejecución y el programa retorna false ya que no hay más formas de probar lo preguntado.

En las demostraciones de dos lineas, por ejemplo:

```
ancestro(X,Y) :- progenitor(X,Y) , ancestro(X,Y).
```

si me encuentro en la lectura segunda, primero se busca una posible demostración del primer enunciado en la lectura tercera, cuarta, quinta, y las que necesitan, luego se vuelve a la lectura segunda y con lo encontrado para el primer enunciado se busca una demostración de la segunda regla con una lectura tercera, cuarta, quinta, etc. Cuando termino de buscar una demostración de la segunda regla, vuelvo a la segunda lectura para ver si hay mas demostraciones posibles de la primera, retomando en el punto en que deje las lecturas de la demostración de esta. Esto se puede ver por ejemplo en:



Aquí se encuentra una demostración para progenitor(primera regla) con X=a, Z=b y se busca ancestro (segunda regla) con el resultado de la primera, luego de finalizar esa búsqueda vuelvo a la lectura dos (ver colores leyenda) y encuentro otra demostracion X=c,Z=d, repitiendo el proceso.