6 CONTROL DEL RAZONAMIENTO

Hemos visto como el programador puede controlar la ejecución de un programa ordenando sus cláusulas y sus metas. Ahora revisaremos otra herramienta de control conocida como corte, para prevenir la reconsideración característica de Prolog (*backtracking*). Este operador extiende además la expresividad de lenguaje, permitiendo la definición de una forma de negación, llamada negación por fallo finito (NAF), asociada al supuesto del mundo cerrado (CWA). En lo que sigue revisaremos estos dos conceptos en detalle.

6.1 CORTE

El árbol-SLD de una meta definitiva puede tener muchas ramas que conducen al fallo de la meta y muy pocas, ó una sola rama, que conducen al éxito. Por ello, el programador podría querer incluir información de control en sus programas, para evitar que el intérprete construya ramas fallidas. Observen que esta meta-información se basa en la semántica operacional del programa, por lo que el programador debe saber como se construyen y se recorren los arboles-SLD. El predicado !/0 denota la operación de **corte**, y puede utilizarse como una literal en las metas definitivas. Su presencia impide la construcción de ciertos sub-arboles.

Corte

Un intérprete de Prolog recorre los nodos de un árbol-SLD primero en profundidad. El orden de las ramas corresponde al orden textual de las cláusulas en el programa. Cuando una hoja es alcanzada, el proceso de **backtracking** es ejecutado. El proceso termina cuando no es posible hacer backtracking (todos los sub-arboles de la raíz del árbol han sido visitados).

Backtrackig

Ejemplo 6.1. *Asumamos el siguiente programa que define que el padre de una persona es su antecesor hombre:*

$$\begin{aligned} padre(X,Y) &\leftarrow progenitor(X,Y), hombre(X). \\ progenitor(benjamin, antonio). & \\ progenitor(maria, antonio). & \\ progenitor(samuel, benjamin). & \\ progenitor(alicia, benjamin). & \\ hombre(benjamin). & \\ hombre(samuel). & \end{aligned}$$

El árbol-SLD de la meta $\leftarrow padre(X, antonio)$ se muestra en la Figura 6.1. Bajo la función de selección implementada en Prolog, encontrará la solución $X/benjamin$. El intento por encontrar otra solución con $X/maria$, mediante el backtracking, fallará puesto que maria no satisface el predicado $hombre/1$.

Para detallar la semántica del corte, es necesario introducir algunos conceptos auxiliares. En un árbol-SLD, cada nodo n_i corresponde a una meta G_i de una derivación-SLD y tiene un átomo seleccionado asociado α_i:

$$G_0 \overset{\alpha_0}{\rightsquigarrow} G_1 \dots G_{n-1} \overset{\alpha_{n-1}}{\rightsquigarrow} G_n$$

Asumamos que para cierto nodo n_k, α_k no es una sub-meta de la meta inicial. Entonces α_k es un átomo β_i del cuerpo de una cláusula de la forma $\beta_0 \leftarrow$

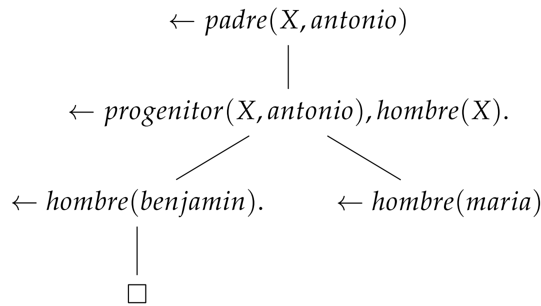

Figura 6.1: Arbol de derivación-SLD para la meta $\leftarrow padre(X, antonio)$

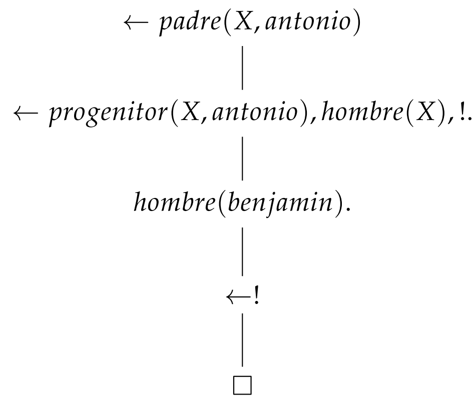

Figura 6.2: Arbol de derivación-SLD para la meta $\leftarrow padre(X, antonio)$ con las ramas fallidas podadas.

$\beta_1, \dots, \beta_i, \dots, \beta_n$ cuya cabeza β_0 unifica con la sub-meta seleccionada en algún nodo $n_{0<j<k}$, es decir un nodo entre la raíz del árbol y el nodo n_k. El nodo n_j se conoce como el **origen**, o padre, de α_k y se denota como $origen(\alpha_k)$.

Nodo origen

El predicado de corte ! se procesa como un átomo ordinario situado en el cuerpo de una cláusula. Sin embargo, cuando el corte es seleccionado para computar la resolución, éste tiene éxito inmediatamente (con la substitución vacía ϵ como resultado). El nodo donde ! fue seleccionado es llamado el **nodo de corte**. Un nodo de corte puede ser visitado nuevamente durante el backtracking. En este caso, el curso normal del recorrido del árbol es alterado (por definición el recorrido continua en el nodo superior a $origen(!)$. Si el corte ocurre en la meta inicial, la ejecución simplemente termina.

Nodo de corte

Ejemplo 6.2. *La formulación del problema padre, nos dice que a lo más existe una solución para nuestra meta. cuando la solución se encuentra, la búsqueda puede detenerse pues ninguna persona tiene más de un padre. Para forzar esta situación, el predicado de corte se agrega al final de* $padre/2$*:*

$$padre(X, Y) \leftarrow progenitor(X, Y), hombre(X), !.$$

Observen que el programa modificado en el ejemplo anterior sólo puede computar un elemento de la relación $padre/2$. El corte detendrá la búsqueda después de encontrar la primer respuesta para la meta $\leftarrow padre(X, Y)$. El origen del corte es la raíz del árbol, por lo que la búsqueda termina después de hacer backtracking al nodo de corte. La otra rama del árbol no es recorrida. El árbol-SLD del programa que incluye el corte se muestra en la figura 6.2.

Observen que la versión modificada con el corte, no puede usarse para computar más de un elemento de la relación “es padre de”. El corte detendrá la búsqueda después de encontrar la primer respuesta a la meta definitiva.

A partir de la definición del corte, se sigue que los efectos del operador son:

1. Divide el cuerpo de la meta en dos partes –Después de éxito de `!/0` , no es posible hacer backtracking hacía las literales a la izquierda del corte. Sin embargo, a la derecha del corte todo funciona de manera usual.
2. Poda las ramas sin explorar directamente bajo $origen(!)$. En otras palabras, no habrá más intentos de unificar la sub-meta seleccionada de $origen(!)$ con el resto de las cláusulas del programa.

El corte es controvertido. La intención al introducir el corte, es poder controlar la ejecución de los programas, sin cambiar su significado lógico. Por tanto, la lectura lógica del corte es `true`. Operacionalmente, si el corte remueve sólo ramas fallidas del árbol-SLD, no tiene influencia en el significado lógico de un programa. Pero el corte puede remover también ramas exitosas del árbol-SLD, atentando contra la completitud de los programas definitivos, o la correctez de los programas generales.

Ejemplo 6.3. *Es bien sabido que los padres de un recién nacido están orgullosos. La proposición puede representarse con la siguiente cláusula definitiva:*

$$orgulloso(X) \leftarrow padre(X,Y), recienNacido(Y).$$

consideren las siguiente cláusulas adicionales:

$$\begin{aligned} padre(X,Y) &\leftarrow progenitor(X,Y), hombre(X). \\ progenitor(juan, maria). & \\ progenitor(juan, cristina). & \\ hombre(juan). & \\ recienNacido(cristina). & \end{aligned}$$

La respuesta a la meta $\leftarrow orgulloso(juan)$ *es* `true`*, puesto que como describimos, juan es padre de cristina, que es un recién nacido. Ahora, si remplazamos la primera cláusula, con su versión que utiliza corte:*

$$padre(X,Y) \leftarrow progenitor(X,Y), hombre(X), !.$$

Y preguntamos nuevamente a Prolog, si

$$\leftarrow orgulloso(juan).$$

la respuesta será `false`*. Esto se debe a que la primer hija de juan en el programa es maria. Una vez que esta respuesta se ha encontrado, no habrá más intentos de satisfacer la meta en* $origen(!)$*. No se considerarán más hijos de juan en la solución computada.*

El programa del ejemplo anterior se ha vuelto incompleto, algunas respuestas correctas no pueden ser computadas. Más grave aún es el caso de las metas generales, donde se puede llegar a resultados incorrectos, por ejemplo, $\leftarrow \neg orgulloso(juan)$ tendría éxito en la versión de nuestro programa que utiliza corte.

Hasta ahora hemos distinguido dos usos del corte: eliminar ramas fallidas en el árbol-SLD; y podar ramas exitosas. Eliminar ramas fallidas se considera una práctica sin riesgo, porque no altera las respuestas producidas durante la ejecución de un programa. Tales cortes se conocen como **cortes verdes**. Sin embargo, este uso del operador corte, esta ligado al uso particular de un programa. Como se ilustra en los ejemplos anteriores, para algunas metas, el operador solo eliminará ramas fallidas; pero para otras podará ramas exitosas. Cortar ramas exitosas se considera una práctica de riesgo. Por eso, tales cortes se conocen como **cortes rojos**.

Corte verde

Corte rojo

Ejemplo 6.4. *Consideremos un ejemplo de corte verde. Si en el ejemplo anterior maria es una recién nacida, agregaríamos la cláusula* $recienNacido(maria)$ *a nuestro programa. Entonces la meta* $\leftarrow orgulloso(X)$ *nos diría que* $X/juan$ *está orgulloso. Esto es, juan tiene una doble razón para estar orgulloso. Pero a nosotros nos basta con saber sólo una vez, que orgulloso está juan. Para evitar que Prolog nos de la respuesta dos veces, definiríamos:*

$$orgulloso(X) \leftarrow padre(X,Y), recienNacido(Y), !.$$

Ejemplo 6.5. *Ahora consideren un ejemplo de corte rojo:*

$$\begin{aligned} min(X,Y,X) &\leftarrow X < Y, !. \\ min(X,Y,Y). \end{aligned}$$

Aparentemente nuestro programa es correcto. De hecho, el programa respondería de manera correcta a metas como $\leftarrow min(2,3,X)$ *respondiendo que "Si" para* $X/2$*; y para* $\leftarrow min(3,2,X)$ *respondería que "Si" para* $X/2$*. Sin embargo el programa no es correcto. Consideren la meta* $\leftarrow min(2,3,3)$ *y verán que Prolog respondería "Si". La razón de esto es que la segunda cláusula dice: el menor de* X *e* Y *es siempre* Y*. El corte está eliminando algunas ramas fallidas, que serían útiles en la definición de* min*. La definición correcta, usando corte, sería:*

$$\begin{aligned} min(X,Y,X) &\leftarrow X < Y, !. \\ min(X,Y,Y) &\leftarrow X \geq Y. \end{aligned}$$

6.1.1 Caso de estudio

Prolog reconsidera automáticamente si esto es necesario para satisfacer una meta. Esto es útil en el sentido que le evita al programador contender explícitamente con la reconsideración, pero si no tenemos ninguna forma de control sobre el *backtracking*, éste puede volverse la causa de programas ineficientes. Comencemos por estudiar un programa muy simple que involucra reconsideración, para identificar los puntos donde el *backtracking* es inútil y condice a la ineficiencia.

Consideren una regulación sobre el nivel de alerta de la contaminación en una ciudad. La Figura 6.3 muestra la relación entre la concentración de los contaminantes en el aire X y el estado de la alarma Y a la manera de un semáforo. La relación entre X e Y se puede establecer mediante tres reglas:

REGLA 1. Si $X < 3$ entonces $Y = verde$

REGLA 2. Si $3 \leq X$ y $X < 6$ entonces $Y = amarilla$

REGLA 3. Si $6 \leq X$ entonces $Y = roja$

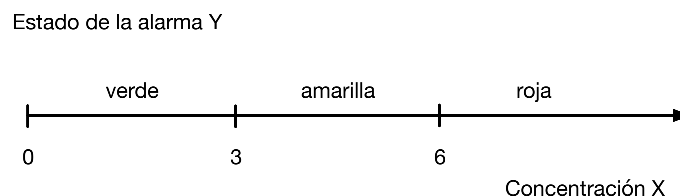

Figura 6.3: Estado de la alarma ambiental en función del nivel de contaminación.

Lo cual puede escribirse en Prolog de la siguiente manera:

```
1 % Concentración X de contaminantes en el aire
2 % f(X,Y), Y estado de la alarma
3
4 f(X,verde) :- X<3.     % Regla 1
5 f(X,amarilla) :- 3=< X, X<6.   % Regla 2
6 f(X,roja) :- 6 =< X.     % Regla 3
```

El programa asume que X será instanciada antes de la llamada a $f/2$, tal y como lo requieren los operadores de comparación utilizados en su definición. A continuación ejecutaremos dos experimentos, donde la ineficiencia de nuestra implementación se hará evidente; pero evitable con la ayuda del operador de corte.

Experimento 1

Supongamos que el nivel de contaminación en la ciudad es $X = 2$. Asumamos que el usuario del sistema no está muy al tanto de las normativas correspondientes y se pregunta si tal nivel de contaminación es inseguro y debiese generar una alarma amarilla. Para ello, podría preguntarle a Prolog si:

```
1 ?- f(2,Y), Y=amarilla.
```

¿Cómo computa Prolog su respuesta? Al resolver la primer meta `f(2,Y)`, Y unificará con `verde`; de forma que la segunda meta toma la forma `verde=amarilla`, lo cual es falso y en consecuencia la lista completa de metas falla. Sin embargo, antes de rendirse, Prolog evaluará vía su reconsideración dos alternativas que resultan inútiles. Invoquen la meta bajo el modo de traza de Prolog para observar esto en detalle:

```
1  [trace]  ?- f(2,Y), Y=amarilla.
2     Call: (9) f(2, _5824) ?
3     Call: (10) 2<3 ?
4     Exit: (10) 2<3 ?
5     Exit: (9) f(2, verde) ?
6     Call: (9) verde=amarilla ?
7     Fail: (9) verde=amarilla ?
8     Redo: (9) f(2, _5824) ?
9     Call: (10) 3=<2 ?
10    Fail: (10) 3=<2 ?
11    Redo: (9) f(2, _5824) ?
12    Call: (10) 6=<2 ?
13    Fail: (10) 6=<2 ?
14    Fail: (9) f(2, _5824) ?
15 false.
```

Las tres reglas de nuestro programa son mutuamente excluyentes, de forma que solo una de ellas tendrá éxito. Por ello, sabemos que tan pronto como una de ellas tiene éxito, no tiene caso intentar probar las otras, puesto que fallarán. Pero Prolog no sabe esto y para ello será necesario indicarle que no reconsidere en ciertos sitios del programa, en este caso, al final de las primeras dos reglas:

```
1 % Concentración X de contaminantes en el aire
2 % f(X,Y), Y estado de la alarma
3 
4 f(X,verde) :-  X<3.     % Regla 1
5 f(X,amarilla) :-  3=< X, X<6.   % Regla 2
6 f(X,roja) :-  6 =< X.     % Regla 3
```

cuyo efecto en la traza del programa puede verse a continuación:

```
1 [trace]  ?- f(2,Y), Y=amarilla.
2    Call: (9) f(2, _9422) ?
3    Call: (10) 2<3 ?
4    Exit: (10) 2<3 ?
5    Exit: (9) f(2, verde) ?
6    Call: (9) verde=amarilla ?
7    Fail: (9) verde=amarilla ?
8 false.
```

Observen que el comportamiento de ambos programs, con y sin corte, es el mismo. La única diferencia es que el primero tardará un poco más en dar su respuesta. En estos casos se dice que el operador corte solo cambia el significado procedural del programa. Como veremos más adelante, éste no es siempre el caso.

Experimento 2

Ejecutemos otro experimento con la versión con cortes de nuestro programa. Preguntémosle la siguiente meta:

```
1  ?- f(7,Y).
2  Y=roja
```

Revisemos la traza del programa:

```
1  [trace]  ?- f(7,Y).
2     Call: (8) f(7, _10108) ?
3     Call: (9) 7<3 ?
4     Fail: (9) 7<3 ?
5     Redo: (8) f(7, _10108) ?
6     Call: (9) 3=<7 ?
7     Exit: (9) 3=<7 ?
8     Call: (9) 7<6 ?
9     Fail: (9) 7<6 ?
10    Redo: (8) f(7, _10108) ?
11    Call: (9) 6=<7 ?
12    Exit: (9) 6=<7 ?
13    Exit: (8) f(7, roja) ?
14 Y = roja.
```

Las tres reglas son intentadas antes de poder computar la respuesta. Esto revela otra fuente de ineficiencia: algunos de los tests que se llevan a cabo son redundantes. Una tercer versión de nuestro programa sería como sigue:

```
1  % Tercer versión
2
3  f(X,verde) :-  X<3, !.
4  f(X,amarilla) :-  X<6, !.
5  f(_,roja).
```

Observen la traza de la meta:

```
1  [trace]  ?- f(7,Y).
2     Call: (8) f(7, _13718) ?
3     Call: (9) 7<3 ?
4     Fail: (9) 7<3 ?
5     Redo: (8) f(7, _13718) ?
6     Call: (9) 7<6 ?
7     Fail: (9) 7<6 ?
8     Redo: (8) f(7, _13718) ?
9     Exit: (8) f(7, roja) ?
10 Y = roja.
```

La respuesta es la misma, pero esta versión del programa es más eficiente que las dos anteriores. Pero, ¿Qué sucede si removemos los operadores de corte? Entonces la salida de la meta sería:

```
1  Y=verde;
2  Y=amarilla;
3  Y=roja;
4  false
```

Esto sugiere que a diferencia de nuestro segundo programa, los cortes aquí ¡cambian el resultado del programa! ¿A qué se debe la respuesta de nuestro tercer programa a las siguientes metas?

```
1  ?- f(2,amarilla).
2  true
3  ?- f(2,Y), Y=amarilla.
4  false
```

La traza de la primer meta explica lo sucedido:

```
1  [trace]  ?- f(2,amarilla).
2     Call: (8) f(2, amarilla) ?
3     Call: (9) 2<6 ?
```

```
4     Exit: (9) 2<6 ?
5     Exit: (8) f(2, amarilla) ?
6  true.
```

Si no tenemos cuidado con el corte, podemos cambiar la semántica de nuestros programas inadvertidamente.

Una descripción más precisa del funcionamiento del corte es como sigue: Llamaremos **meta padre** a la meta que unifica con la cabeza de la cláusula que contiene al operador corte. Cuando el corte es encontrado como una meta, tiene éxito inmediatamente, pero compromete al sistema con todas las elecciones hechas entre el momento en que la meta padre fue invocada y el momento en que el corte fue encontrado. Todas las alternativas entre el nodo padre y el corte son descartadas.

Meta padre

Ejemplo 6.6. *Consideren la siguiente cláusula:*

$$H : -B_1, B_2, \ldots, B_m, !, B_{m+2}, B_n$$

Asumamos que H unifica con la meta G, por lo que G es su meta padre. Al alcanzar el corte, la solución para las metas $B_1, \ldots B_m$ se fija, todas sus alternativas son descartadas. No así las metas $B_{m+2}, \ldots, B_n$.

6.1.2 Otros ejemplos

Implementaremos una función miembro determinista, e.g., solo computa una respuesta:

```
1  % single member
2
3  smember(X,[X|L]) :- !.
4  smember(X,[_,L]) :- smember(X,L).
```

De esta forma, su comportamiento es como sigue:

```
1  ?- smember(X,[1,2,3]).
2  X=1;
3  false
```

A veces queremos agregar un elemento a una lista, solo en el caso de que éste no sea ya parte de ella. La siguiente función hace este trabajo:

```
1  % agregar sin duplicar
2
3  add(X,L,L) :- smember(X,L), !.
4  add(X,L,[X|L]).
```

Al igual que en el caso de $max/3$ se asume que el tercer argumento de $add/3$ no está instanciado en las llamadas. En caso contrario, su comportamiento es inesperado, por ejemplo:

```
1  ?- add(a,[a],[a,a]).
2  true
```

Nuestro último ejemplo tiene que ver con clasificar objetos en categorías que cumplen ciertas condiciones y son excluyentes entre si. Por ejemplo, asumamos las siguientes relaciones entre equipos de futbol:

```
1  % clasificando en categorias
2
3  derrota(barcelona, real_madrid).
4  derrota(roma, barcelona).
5  derrota(cruz_azul, real_madrid).
```

La idea es clasificar a los equipos mediante una relación $clase/2$ siguiendo los siguientes criterios:

GANADOR. Es un equipo que gana todos sus encuentros.

LUCHADOR. Es un equipo que gana algunos de sus encuentros, pero pierde otros.

DEPORTISTA. Un equipo que pierde todos sus encuentros.

La definición de *clase*/2 en Prolog es como sigue:

```
% luchador es quien gana y pierde algunos partidos
% ganador es quien gana siempre
% deportista es quien pierde siempre

clase(X,luchador) :-
    derrota(X,_),
    derrota(_,X), !.

clase(X, ganador) :-
    derrota(X,_), !.

clase(X, deportista) :-
    derrota(_, X).
```

La consulta al programa es como sigue:

```
?- clase(barcelona,X).
X = luchador;
false
?- clase(barcelona, deportista).
false
```

6.2 NEGACIÓN POR FALLO FINITO

¿Cómo podemos expresar la siguiente proposición en Prolog: A Adriana le gustan todos los animales, excepto las serpientes? La primer parte de la proposición es sencilla:

```
le_gusta(adriana,X) :-
    animal(X).
```

Pero la segunda parte es más complicada. Deberíamos expresar que si X es una serpiente, entonces es falso que X le gusta a *adriana*; y que en cualquier otro caso X le gusta a *adriana*. La traducción a Prolog es la siguiente:

```

% A Adriana le gustan todos los animales, excepto las
% serpientes

le_gusta(adriana,X) :-
    serpiente(X), !, fail.

le_gusta(adriana,X) :-
    animal(X).
```

De hecho, las dos cláusulas puede escribirse juntas de manera más compacta:

```
le_gusta(adriana,X) :-
  serpiente(X), !, fail
  ;
  animal(X).
```

Podemos usar la misma idea, para escribir un predicado *diferente*/2:

```
diferente(X,Y) :-
    X=Y, !, fail
    ;
    true.
```

Estos ejemplos sugieren que sería conveniente tener un predicado *not*/1 con la siguiente definición:

```
not(P) :-
  P,!,fail
  ;
  true.
```

Esta definición descansa enteramente en la semantica operacional de Prolog. Esto es, las sub-metas se deben resolver de izquierda a derecha, y las cláusulas se buscan en el orden en que aparecen en el texto del programa.

En lo que sigue asumiremos que *not*/1 es un predicado predefinido en Prolog definido como un operador prefijo. De hecho SWI-Prolog, define el operador $\backslash+$ para ello. De forma que los ejemplos anteriores, se pueden definir ahora de la siguiente forma:

```
%%%Versiones con not

le_gusta2(adriana,X) :-
    animal(X),
    \+ serpiente(X).

diferente2(X,Y) :-
    \+ (X=Y).
```

6.2.1 CWA, NAF y corte

Los ejemplos de la sección anterior ilustran las ventajas y desventajas de usar el operador de corte. Las primeras incluyen:

1. Usando el operador de corte se puede mejorar la eficiencia de los programas lógicos. La idea es decirle a Prolog explícitamente –No intentes otra alternativa, pues están condenadas al fracaso.
2. Usando el operador de corte podemos definir cláusulas mutuamente exclusivas, por lo que podemos expresar reglas de la forma: Si la condición P entonces conclusión Q, y en cualquier otro caso R. Mejorando así la expresividad del lenguaje.

Las reservas con respecto al uso del operador de corte tienen que ver con que fácilmente podemos perder la correspondencia entre el significado declarativo de nuestros programas y su significado procedural. Esto es, si cambiamos el orden de las cláusulas de nuestro programa, sin usar el operador de corte, afectamos la eficiencia o las condiciones de terminación de éste, sin cambiar su significado declarativo; pero si las cláusulas usan corte, un cambio en su orden si que afecta éste significado. Veamos un ejemplo:

Ejemplo 6.7. *Si queremos expresar en Prolog $p \Leftrightarrow (a \wedge b) \vee c$ podemos usar el siguiente programa:*

```
p :- a,b.
p :- c.
```

Podemos cambiar el orden de las cláusulas y el significado declarativo del programa sigue siendo el mismo. Introduzcamos un corte:

```
p :- a, !, b.
p :- c.
```

El significado declarativo del programa es ahora $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$. Pero si invertimos el orden de las cláusulas el significado declarativo se vuele $p \Leftrightarrow c \vee (a \wedge b)$.

Como se mencionó, los casos en que el corte cambia el significado declarativo del programa, se denominan cortes rojos; en contraste con los cortes verdes que no lo hacen.

Ahora bien, la definición de la negación usando `fail` y corte, tiene un problema más serio: No se corresponde con la definición lógica de la negación. Consideren el siguiente ejemplo:

Ejemplo 6.8. *Consideren el siguiente programa de una sola cláusula:*

```
1    pintor(artur_heras).
```

Si le preguntamos al programa si `artur_heras` *es un* `pintor`*:*

```
1 ?- pintor(artur_heras).
2 true
```

Lo cual es correcto pues esa respuesta se sigue del programa. Ahora, si preguntamos:

```
1 ?- pintor(juan_gris).
2 false
```

Lo cual también es correcto puesto que no se sigue del programa que `juan_gris` *sea un pintor. Ahora:*

```
1 ?- \+  pintor(juan_gris).
2 true
```

En este caso, el true de Prolog no significa que la meta se siga lógicamente del programa.

La última respuesta de Prolog se basa en el **supuesto del mundo cerrado** (CWA). i.e., un programa Prolog representa todo lo que es verdadero en el mundo que describe. De manera que aquello que no esté declarado en el programa, o sea derivable de éste lógicamente, se asume como falso; y su negación en consecuencia se asume verdadera. El CWA puede formularse como una pseudo-regla de inferencia que expresa:

CWA

$$\frac{\Delta \nvdash \alpha}{\neg \alpha} \quad (CWA)$$

Si una fbf atómica de base (sin variables) α, no puede derivarse del programa Δ siguiendo las reglas de inferencia del sistema, entonces puede derivarse $\neg\alpha$. En el caso de los sistemas correctos y completos, la condición $\Delta \nvdash \alpha$ es equivalente a $\Delta \not\models \alpha$. Como este es el caso para la resolución-SLD, la condición puede ser remplazada por $\alpha \notin M_\Delta$.

Ejemplo 6.9. *Dado el siguiente programa lógico:*

$$\begin{aligned} sobre(X,Y) &\leftarrow en(X,Y). \\ sobre(X,Y) &\leftarrow en(X,Z), sobre(Z,Y). \\ en(c,b). & \\ en(b,a). & \end{aligned}$$

la fbf $sobre(b,c)$ no puede ser derivada por resolución-SLD a partir del programa Δ (vean el árbol de derivación en la Figura 6.4). En realidad $sobre(b,c)$ no puede ser derivada por ningún sistema correcto, puesto que no es una consecuencia lógica de Δ. Dada la completitud de la resolución-SLD, se sigue que $\Delta \not\models sobre(b,c)$ y usando la CWA inferimos que $\neg sobre(b,c)$.

En contra de lo que podría ser nuestra primera intuición, existen problemas asociados a la CWA. Uno de ellos es que en la vida cotidiana existen muchas situaciones en donde el CWA no se puede asumir.

Otro problema, técnico en este caso, tiene que ver con que establecer que una meta no es derivable dado un programa definitivo es no decidible en el caso general. Esto es, no es posible determinar si la pseudo-regla asociada al CWA aplica o no. Una versión más débil de la suposición de mundo cerrado, se logra si asumimos que $\neg\alpha$ es derivable a partir del programa Δ si la meta $\leftarrow \alpha$ tiene un árbol-SLD finito que falla, i.e., NAF.

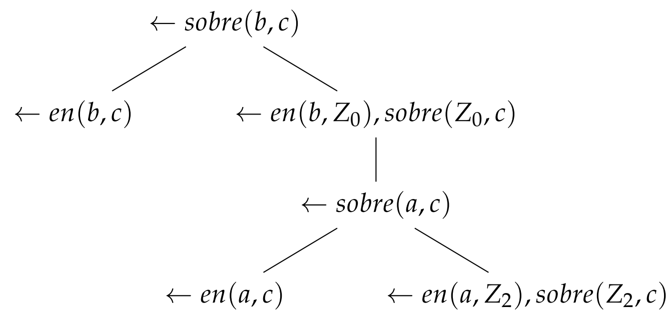

Figura 6.4: Árbol de derivación-SLD fallido

Es necesario contrastar la NAF con el CWA, que también puede verse como una negación por falla, pero infinita. Para ilustrar la diferencia entre los dos enfoques extendamos el programa Δ con la siguiente cláusula evidentemente verdadera $sobre(X,Y) \leftarrow sobre(X,Y)$.

Ejemplo 6.10. *El árbol-SLD de la meta $\leftarrow sobre(b,c)$ sigue sin contener refutaciones, pero ahora es infinito. Por lo tanto no podemos concluir que $\neg sobre(b,c)$ usando NAF, pero si usando CWA.*

Otro aspecto a considerar es el efecto de la negación en los cuantificadores que están bajo su alcance. Veamos un ejemplo:

Ejemplo 6.11. *Consideren el siguiente programa acerca de restaurantes:*

```
1 estrellas(diverxo).
2 estrellas(ricard_camarena).
3 
4 caro(diverxo).
5 
6 razonable(X) :-
7     \+ caro(X).
```

Una meta de interés sería:

```
1 ?- estrellas(X), razonable(X).
2 X = ricard_camarena.
```

Ahora bien, si preguntamos aparentemente la misma meta:

```
1 ?- razonable(X), estrellas(X).
2 false.
```

Usen la traza para entender este comportamiento de Prolog... y siempre prioricen calidad sobre precio :-)

Como recordarán, la diferencia entre estas dos metas que, aparentemente equivalentes, es que en el primer caso, la variable X está instanciada cuando $razonable/1$ es invocada; mientras que en el segundo caso esto no es así.

La cuestión es que la negación cambia la cuantificación de la variable bajo su alcance. La meta $caro(X)$ significa ¿Existe una X tal que $caro(X)$ es verdadera? Si ese es el caso ¿Cual X? La meta $notcaro(X)$ significa ¿Verdad que para todo X, no es el caso que $caro(X)$? ó ¿Es cierto que nadie es caro? En el ejemplo anterior, dado que $diverxo$ es caro, la respuesta es $false$. En general $notMeta$ es seguro si todas las variables en $Meta$ están instanciadas.

EL objetivo de advertir esos problemas con el operador de corte, e indirectamente con la negación, no es desalentar su uso. Por el contrario, pueden ser de gran utilidad. La cuestión es usarlos de manera adecuada, atendiendo sus singularidades.

Existen dos aproximaciones a la solución de estos problemas: ver los programas como resúmenes de programas más extensos que validan las literales negativas; o

redefinir la noción de consecuencia lógica de forma que sólo algunos modelos del programa (el mínimo de Herbrand, por ejemplo) sean tomados en cuenta. En ambos casos, el efecto es descartar algunos modelos del programa que no son interesantes. Primero justificaremos la regla NAF en términos de la compleción de los programas definitivos y posteriormente, extenderemos el lenguaje de los programas definitivos para incluir en ellos literales negativas en la cabeza y cuerpo de las cláusulas.

6.3 LA COMPLECIÓN DE UN PROGRAMA

La idea que presentaremos a continuación se debe a Clark [25] y se basa en que cuando uno escribe un programa definitivo Δ, en realidad quiere expresar algo más que su conjunto de cláusulas definitivas. El programa deseado puede formalizarse como la compleción de Δ. Consideren la siguiente definición:

$$\begin{aligned} sobre(X,Y) &\leftarrow en(X,Y). \\ sobre(X,Y) &\leftarrow en(X,Z), sobre(Z,Y). \end{aligned}$$

Estas reglas especifican que un objeto está sobre un segundo objeto, si el primer objeto está encima del segundo (1) ó si el objeto está sobre otro objeto que a su vez está encima del segundo (2). Esto también puede escribirse como:

$$sobre(X,Y) \leftarrow en(X,Y) \vee (en(X,Z), sobre(Z,Y))$$

Ahora, ¿Qué sucede si remplazamos la implicación por la equivalencia lógica?

$$sobre(X,Y) \leftrightarrow en(X,Y) \vee (en(X,Z), sobre(Z,Y))$$

Está fbf expresa que X está sobre Y si y sólo si una de las condiciones es verdadera. Esto es, si ninguna de las condiciones se cumple, ¡se sigue que X no está sobre Y! Esta es la intuición seguida para justificar la negación por fallo.

Desafortunadamente, combinar cláusulas definitivas como en el ejemplo anterior, sólo es posible para cláusulas con cabezas idénticas. Por ejemplo:

$$\begin{aligned} &en(c,b). \\ &en(b,a). \end{aligned}$$

Por una simple transformación, el programa puede ser escrito como:

$$\begin{aligned} en(X_1,X_2) &\leftarrow X_1 = c, X_2 = b \\ en(X_1,X_2) &\leftarrow X_1 = b, X_2 = a \end{aligned}$$

Las cláusulas pueden combinarse en una sola fórmula, donde la implicación es remplazada por la equivalencia lógica.

$$en(X_1,X_2) \leftrightarrow (X_1 = c, X_2 = b) \vee (X_1 = b, X_2 = a)$$

La lectura lógica de esta fbf es que X_1 está en X_2 si y sólo si $X_1 = c$ y $X_2 = b$ o si $X_1 = b$ y $X_2 = a$. Esta transformación se puede realizar sobre un programa lógico definitivo Δ y el resultado se conoce como **compleción** de Δ.

Compleción

Definición 6.1 (Compleción). *Sea Δ un programa lógico definitivo. La compleción $comp(\Delta)$ de Δ es el conjunto de fórmulas obtenido a partir de las siguientes tres transformaciones:*

1. *Para cada símbolo de predicado ϕ remplazar la cláusula α de la forma:*

$$\phi(t_1,\ldots,t_m) \leftarrow \alpha_1,\ldots,\alpha_n \quad (n \geq 0)$$

por la fórmula:

$$\phi(X_1, \dots, X_m) \leftarrow \exists Y_1, \dots, Y_i(X_1 = t_1, \dots, X_m = t_m, \alpha_1, \dots, \alpha_n)$$

donde las Y_i son todas variables en α y las X_i son variables únicas que no aparecen en α.

2. *Para cada símbolo de predicado ϕ remplazar todas las fbf:*

$$\begin{aligned} \phi(X_1, \dots, X_m) &\leftarrow \beta_1 \\ &\vdots \\ \phi(X_1, \dots, X_m) &\leftarrow \beta_j \end{aligned}$$

por la fórmula:

$$\begin{aligned} \forall X_1, \dots, X_m(\phi(X_1, \dots, X_m) \leftrightarrow \beta_1 \vee, \dots, \vee \beta_j \quad &si \quad j > 0 \\ \forall X_1, \dots, X_m(\neg\phi(X_1, \dots, X_m)) \quad &si \quad j = 0 \end{aligned}$$

3. *Finalmente el programa se extiende con los siguientes axiomas de igualdad libre, que definen las igualdades introducidas en el paso 1:*

$$\begin{aligned} \forall(X = X) \\ \forall(X = Y \Rightarrow Y = X) \\ \forall(X = Y \wedge Y = Z \Rightarrow X = Z) \\ \forall(X_1 = Y_1 \wedge \dots \wedge X_n = Y_n \Rightarrow f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n)) \\ \forall(X_1 = Y_1 \wedge \dots \wedge X_n = Y_n \Rightarrow (\phi(X_1, \dots, X_n) \Rightarrow \phi(Y_1, \dots, Y_n)) \\ \forall(f(X_1, \dots, X_n) = f(Y_1, \dots, Y_n) \Rightarrow X_1 = Y_1 \wedge \dots \wedge X_n = Y_n) \\ \forall(\neg f(X_1, \dots, X_m) = g(Y_1, \dots, Y_n))(\text{Si } f/m \neq g/n) \\ \forall(\neg X = t)(\text{Si } X \text{ es un subtermino propio de } t) \end{aligned}$$

Estas definiciones garantizan que la igualdad (=) sea una relación de equivalencia; que sea una relación congruente; y que formalice la noción de unificación. Las primeros cinco definiciones se pueden abandonar si se especifica que = representa la relación de **identidad**.

Identidad

Ejemplo 6.12. *Consideremos la construcción de $comp(\Delta)$ tal y como se definió anteriormente. El primer paso produce:*

$$\begin{aligned} sobre(X_1, X_2) &\leftarrow \exists X, Y\ (X_1 = X, X_2 = Y, en(X, Y)) \\ sobre(X_1, X_2) &\leftarrow \exists X, Y, Z\ (X_1 = X, X_2 = Y, en(Z, Y), sobre(Z, Y)) \\ en(X_1, X_2) &\leftarrow (X_1 = c, X_2 = b) \\ en(X_1, X_2) &\leftarrow (X_1 = b, X_2 = a) \end{aligned}$$

dos pasos más adelante obtenemos:

$$\begin{aligned} \forall X_1, X_2(sobre(X_1, X_2 &\leftrightarrow \exists X, Y(\dots) \wedge \exists X, Y, Z(\dots)) \\ \forall X_1, X_2(en(X_1, X_2) &\leftrightarrow (X_1 = c, X_2 = b) \wedge (X_1 = b, X_1 = a)) \end{aligned}$$

y el programa se termina con las definiciones de igualdad como identidad y unificación.

La compleción $comp(\Delta)$ de un programa definitivo Δ preserva todas las literales positivas modeladas por Δ. Esto es, si $\Delta \models \alpha$ entonces $comp(\Delta) \models \alpha$. Tampoco se agrega información positiva al completar el programa: Si $comp(\Delta) \models \alpha$ entonces $\Delta \models \alpha$. Por lo tanto, al completar el programa no agregamos información positiva al mismo, solo información negativa.

Como sabemos, no es posible que una literal negativa pueda ser consecuencia lógica de un programa definitivo. Pero al substituir las implicaciones en Δ por equivalencias en $comp(\Delta)$ es posible inferir información negativa a partir del programa completado. Esta es la justificación de la regla NAF, cuyas propiedades de consistencia se deben a Clark [25]:

Teorema 6.1 (Consistencia de la NAF)**.** *Sea Δ un programa definitivo y $\leftarrow \alpha$ una meta definitiva. Si $\leftarrow \alpha$ tiene un árbol-SLD finito fallido, entonces $comp(\Delta) \models \forall(\neg\alpha)$.*

La consistencia se preserva aún si α no es de base. Por ejemplo, $\leftarrow en(a, X)$ falla de manera finita y por lo tanto, se sigue que $comp(\Delta) \models \forall(\neg en(a, X))$. La completitud de la NAF también ha sido demostrada:

Teorema 6.2 (Completitud de la NAF)**.** *Sea Δ un programa definitivo. Si $comp(\Delta) \models \forall(\neg\alpha)$ entonces existe un árbol finito fallido para la meta definitiva $\leftarrow \alpha$.*

Observen que solo enuncia la existencia de un árbol-SLD finito fallido. Como se ha mencionado, un árbol-SLD puede ser finito bajo ciertas reglas de computación e infinito bajo otras. En particular, el teorema de completitud no es válido para las reglas de computación de Prolog. La completitud funciona para una subclase de derivaciones-SLD conocidas como **justas** (*fair*), las cuales o bien son finitas o garantizan que cada átomo en la derivación (u ocurrencia de éste), es seleccionado eventualmente por las reglas de computación. Un árbol-SLD es justo si todas sus derivaciones son justas. La NAF es completa para árboles-SLD justos. Este tipo de derivaciones se pueden implementar fácilmente: selecciona la sub-meta más a la izquierda y agrega nuevas submetas al final de esta (búsqueda en amplitud). Sin embargo, pocos sistemas implementan tal estrategia por razones de eficiencia.

Derivación justa

6.4 RESOLUCIÓN SLDNF PARA PROGRAMAS DEFINITIVOS

En el capítulo 5.4 presentamos el método de resolución-SLD, utilizado para probar si una literal positiva cerrada es consencuencia lógica de un programa. En la sección anterior afirmamos que también las literales negadas pueden derivarse a partir de la terminación de programas lógicos definitivos. Combinando la resolución SLD y la negación como fallo finito (NAF), es posible generalizar la noción de meta definitiva para incluir literales positivas y negadas. Tales metas se conocen como generales.

Definición 6.2 (Meta general)**.** *Una meta general tiene la forma:*

$$\leftarrow \alpha_1, \ldots \alpha_n \quad (n \geq 0)$$

donde cada α_i es una literal positiva o negada.

La combinación de la resolución SLD y la NAF se llama resolución SLDNF.

Definición 6.3 (Resolución SLDNF para programas definitivos)**.** *Sea Δ un programa definitivo, G_0 una meta general y $\mathcal{R}$ una función de selección (también conocida como regla de computación). Una derivación SLDNF de G_0 usando $\mathcal{R}$, es una secuencia finita o infinita de metas generales:*

$$G_0 \overset{\alpha_0}{\leadsto} G_1 \ldots G_{n-1} \overset{\alpha_{n-1}}{\leadsto} G_n$$

donde $G_i \overset{\alpha_i}{\leadsto} G_{i+1}$ puede ocurrir si:

1. *la literal $\mathcal{R}$-seleccionada en G_i es positiva y G_{i+1} se deriva de G_i y α_i por un paso de resolución SLD;*
2. *la literal $\mathcal{R}$-seleccionada en G_i es negativa ($\neg\alpha$) y la meta $\leftarrow \alpha$ tiene un árbol SLD fallido y finito y G_{i+1} se obtiene a partir de G_i eliminando $\neg\alpha$ (en cuyo caso α_i, corresponde al marcador especial FF).*

Cada paso en una derivación SLDNF produce una substitución, en el caso 1 un MGU y en el caso 2, la substitución vacía ϵ.

Entonces, una literal negativa $\neg\alpha$ es demostrada si $\leftarrow \alpha$ tiene un árbol SLD finito que falla. Por dualidad, $\neg\alpha$ falla de manera finita si α es demostrada. Además de la refutación y de la derivación infinita, existen dos clases de derivaciones SLDNF completas dada una función de selección:

1. Una derivación se dice (finitamente) **fallida** si (i) la literal seleccionada es positiva y no unifica con ninguna cabeza de las cláusulas del programa, o (2) la literal seleccionada es negativa y tiene un fallo finito.
2. Una derivación se dice **plantada** (*stuck*) si la sub-meta seleccionada es de la forma $\neg\alpha$ y $\leftarrow \alpha$ tiene un fallo infinito.

Ejemplo 6.13. *Considere el siguiente programa:*

$$en(c,b)$$
$$en(b,a)$$

La meta $\leftarrow en(X,Y), \neg en(Z,X)$ tiene una refutación-SLDNF con la substitución computada {X/c,Y/b}:

$$\begin{aligned}
G &= \leftarrow en(X,Y), \neg en(Z,X). \\
G_0 &= \leftarrow en(X,Y). \\
\alpha_0 &= en(c,b). \\
\theta_0 &= \{X/c, Y/b\} \\
\\
G_1 &= \neg en(Z,X)\theta_0 \;= \leftarrow en(Z,c) \\
\alpha_1 &= FF \\
\theta_1 &= \epsilon \\
G_2 &= \square \\
\\
\theta &= \theta_0\theta_1 \;=\; \{X/c, Y/b\}
\end{aligned}$$

En cambio, si la función de selección hubiera computado las cláusulas de abajo hacía arriba $\alpha_0 = en(b,a)$ la derivación hubiera sido fallida (a ustedes probarlo).

Como es de esperarse, la resolución-SLDNF es consistente, después de todo, la resolución-SLD y la NAF son consistentes.

Teorema 6.3 (Consistencia de la resolución-SLDNF)**.** *Sea Δ un programa definitivo y $\leftarrow \alpha_1, \ldots, \alpha_n$ una meta general. Si $\leftarrow \alpha_1, \ldots, \alpha_n$ tiene una refutación SLDNF con una substitución computada θ, $comp(\Delta) \models \forall(\alpha_1\theta, \ldots, \alpha_n\theta)$.*

Sin embargo, la resolución-SLDNF no es completa aunque pudiéramos haber esperado lo contrario. La resolución SLDNF no es completa a pesar de que la resolución-SLD y la NAF si lo son. Un simple contra ejemplo es $\leftarrow \neg en(X,Y)$ que corresponde a la consulta "¿Hay algunos bloques X e Y, tal que X no está en Y?"

Uno esperaría varias respuestas a esta consulta, por ejemplo, el bloque a no está encima de ningún bloque, etc.

Pero la derivación SLDNF de $\leftarrow \neg en(X,Y)$ falla porque la meta $\leftarrow en(X,Y)$ tiene éxito (puede ser demostrada). El problema es que nuestra definición de derivación fallida es demasiado conservadora. El éxito de $\leftarrow en(X,Y)$ no significa necesariamente que no halla un bloque que no esté en otro bloque, sólo que existe al menos un bloque que no está en otro.

El problema tiene su origen en que la NAF, en contraste con la resolución SLD, es sólo una prueba (*test*). Recuerden que dada la definición de la resolución SLDNF y la consistencia y completitud de la NAF, tenemos que $\neg en(X,Y)$ tiene éxito si y sólo si ($\equiv$) $en(X,Y)$ tiene asociado un árbol SLD fallido y finito; o si y sólo si $comp(\Delta) \models \forall(\neg en(X,Y))$. Por lo tanto, la meta general $\leftarrow en(X,Y)$ no debe leerse como una consulta cuantificada existencialmente, sino como una prueba universal: "Para todo bloque X e Y, ¿No está X en Y?".

Esta última consulta tiene una respuesta negativa en el modelo deseado del programa, puesto que el bloque b está en el bloque a. El problema anterior se debe a la cuantificación de las variables en la literal negativa. Si replanteamos la consulta anterior como $\leftarrow \neg en(a,b)$ entonces la resolución SLDNF alcanza una refutación puesto que $\leftarrow en(a,b)$ falla con una derivación finita.

Algunas veces se asume que la función de selección $\mathcal{R}$ permite seleccionar una literal negativa $\neg\alpha$ si la literal α no tiene variables libres o si α tiene asociada una substitución computada vacía. Estas funciones de selección se conocen como seguras (*safe*).

6.5 PROGRAMAS LÓGICOS GENERALES

Con los desarrollos anteriores estamos en posición de extender el lenguaje de los programas definitivos para incluir cláusulas que contienen literales tanto positivas como negativas en su cuerpo. Estas fbf se llaman **cláusulas generales**, y en consecuencia, un conjunto de ellas conforma un programa general. En ocasiones, a los programas generales se les conoce a veces como **programas normales**.

Cláusulas Generales

Programas Normales

Definición 6.4 (Cláusula General)**.** *Una cláusula general es una fbf de la forma* $\alpha \leftarrow \alpha_1, \ldots, \alpha_n$ *donde* α *es una fbf atómica y* $\alpha_1, \ldots, \alpha_n$ *son literales (*$n \geq 0$*).*

Definición 6.5 (Programa General)**.** *Un programa lógico general es un conjunto finito de cláusulas generales.*

Ahora podemos extender nuestro programa del mundo de los bloques con las siguientes relaciones:

$$\begin{array}{rcl} base(X) & \leftarrow & en(Y,X), en_la_mesa(X). \\ en_la_mesa(X) & \leftarrow & \neg no_en_la_mesa(X). \\ no_en_la_mesa(X) & \leftarrow & en(X,Y). \\ en(c,b). & & \\ en(b,a). & & \end{array}$$

La primer cláusula especifica que un bloque es base si está sobre la mesa y tiene otro bloque encima. La segunda cláusula indica que cuando no es cierto que un bloque no está sobre la mesa, entonces está sobre la mesa. La tercera cláusula especifica que un bloque que está sobre otro, no está sobre la mesa.

Parece claro, pero la pregunta que deberíamos hacernos es qué tipo de sistema de prueba queremos para los programas lógicos generales y cuales serán las aproximaciones lógicas a las sutilezas, algunas ya discutidas, introducidas por este tipo de lenguajes.

Observen que aunque el lenguaje fue enriquecido, no es posible de cualquier forma que una literal negativa sea consecuencia lógica de un programa dado. La razón es la misma que para los programas definidos, la base de Herbrand de un programa Δ, B_Δ es un modelo de Δ en el que todas las literales negativas son falsas. Al igual que con los programas definidos, la pregunta es entonces como lograr inferencias negativas consistentes. Afortunadamente el concepto de compleción de programa puede aplicarse también a los programas lógicos generales.

Ejemplo 6.14. *La compleción de* $gana(X) \leftarrow mueve(X, Y), \neg gana(Y)$ *contiene la fbf:*

$$\forall X_1(gana(X_1) \leftrightarrow \exists X, Y(X_1 = X, mueve(X, Y), \neg gana(Y)))$$

Desafortunadamente, la compleción de los programas normales puede ocasionar paradojas.

Ejemplo 6.15. *Consideren la cláusula general* $p \leftarrow \neg p$*, su compleción incluye* $p \leftrightarrow \neg p$*. La inconsistencia del programa terminado se debe a que* $p/0$ *está definida en términos de su propio complemento.*

Una estrategia de programación para evitar este problema consiste en componer los programas por capas o estratos, forzando al programador a referirse a las negaciones de una relación hasta que ésta ha sido totalmente definida. Se entiende que tal definición se da en un estrato inferior a donde se presenta la negación. En la definición del **programa estratificado** usaremos Δ^p para referirnos al subconjunto de cláusulas en Δ que tienen a p como cabeza.

Programa Estratificado

Definición 6.6 (Programa Estratificado)**.** *Un programa general* Δ *se dice estratificado si y sólo si existe al menos una partición* $\Delta_1 \cup \dots \cup \Delta_n$ *de* Δ *tal que :*

1. *Si* $p(\dots) \leftarrow q(\dots), \dots \in \Delta_i$ *entonces* $\Delta^q \subseteq \Delta_1 \cup \dots \cup \Delta_i$;
2. *Si* $p(\dots) \leftarrow \neg q(\dots), \dots \in \Delta_i$ *entonces* $\Delta^q \subseteq \Delta_1 \cup \dots \cup \Delta_{i-1}$.

Por ejemplo, el siguiente programa está estratificado:

Δ_2: $base(X) \leftarrow en(Y, X), en_la_mesa(X)$.
$en_la_mesa(X) \leftarrow \neg no_en_la_mesa(X)$.
Δ_1: $no_en_la_mesa(X) \leftarrow en(X, Y)$.
$en(c, b)$.
$en(b, a)$.

Apt, Blair y Walker [2] demostraron que la compleción de un programa estratificado en consistente, de forma que la situación descrita anteriormente no puede ocurrir. Sin embargo, la estratificación es solo una condición suficiente para la **consistencia**: Determinar si un programa está estratificado es decidible, pero determinar si la compleción de un programa general es consistente, es indecidible. Por lo tanto, hay programas generales no estratificados cuya compleción es consistente.

Consistencia

6.6 RESOLUCIÓN-SLDNF PARA PROGRAMAS GENERALES

Hemos revisado el caso de la resolución-SLDNF entre programas definitivos y metas generales. Informalmente podemos decir que la resolución-SLDNF combina la resolución-SLD con los siguientes principios:

1. $\neg\alpha$ tiene éxito ssi $\leftarrow \alpha$ tiene un árbol-SLD finito que falla.
2. $\neg\alpha$ falla finitamente ssi $\leftarrow \alpha$ tiene una refutación-SLD.

El paso de programas definitivos a programas generales, es más complicado. Para probar $\neg\alpha$, debe de existir un árbol finito fallido para $\leftarrow \alpha$. Tal árbol puede contener nuevas literales negativas, las cuales a su vez deben pueden tener éxito o fallar finitamente. Esto complica un poco la definición de la resolución-SLDNF para programas generales.

Ejemplo 6.16. *Es posible llegar a situaciones paradójicas cuando los predicados están definidos en términos de sus propios complementos. Consideren el programa no estratificado:*

$$\alpha \leftarrow \neg\alpha$$

Dada la meta inicial $\leftarrow \alpha$, se puede construir una derivación $\leftarrow \alpha \rightsquigarrow \leftarrow \neg\alpha$. La derivación puede extenderse hasta una refutación si $\leftarrow \alpha$ falla finitamente. De manera alternativa, si $\leftarrow \alpha$ tiene una refutación, entonces la derivación falla. Helas! esto es imposible pues la meta $\leftarrow \alpha$ no puede tener una refutación y fallar finitamente al mismo tiempo.

En lo que sigue, definiremos las nociones de derivación-SLDNF y árbol-SLDNF, de manera similar a la derivación-SLD y a los arboles-SLD. La idea se concreta en el concepto de **bosque-SLDNF**: Un conjunto de árboles cuyos nodos está etiquetados con metas generales. Un **sub-bosque** se obtiene removiendo algunos nodos y sus hijos del bosque original. Dos sub-bosques B_1 y B_2 se consideran equivalentes si contieinen los mismos árboles considerando un posible renombrado de variables. B_1 es más pequeño que B_2 si es equivalente a algún sub-bosque de B_2.

Bosque-SLDNF
Sub-Bosque

Definición 6.7 (Bosque-SLDNF). *Sea Δ un programa general, G_0 una meta general, y $\mathcal{R}$ una función de selección. El bosque-SLDNF de G_0 es el bosque más pequeño (considerando el posible renombrado de variables), tal que:*

1. *G_0 es la raíz del árbol.*
2. *Si G es un nodo en el bosque cuya literal seleccionada es positiva, entonces para cada cláusula α tal que G' puede ser derivada de G y α (con UMG θ), G tiene un hijo etiquetado como G'. Si no existe tal cláusula, entonces G tiene un hijo etiquetado* **FF** *(falla finita);*
3. *Si G es un nodo del bosque cuya literal seleccionada es de la forma $\neg\alpha$ (G es de la forma $\leftarrow \alpha_1, \ldots, \alpha_{i-1}, \neg\alpha, \alpha_{i+1}, \ldots, \alpha_n$), entonces:*
   - *El bosque contiene un árbol cuya raíz está etiquetada como $\leftarrow \alpha$;*
   - *Si el árbol con raíz $\leftarrow \alpha$ tiene una hoja $\square$ con la substitución computada ϵ, entonces G tiene un sólo hijo etiquetado* **FF***;*
   - *Si el árbol con raíz $\leftarrow \alpha$ es finito y tiene todas sus hojas etiquetadas* **FF***, entonces G tiene un sólo hijo (con substitución asociada ϵ) etiquetado como $\leftarrow \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_n$.*

Observen que la literal negativa seleccionada $\neg\alpha$ falla sólo si $\leftarrow \alpha$ tiene una refutación con la substitución computada ϵ. Como veremos más adelante, esta condición que no era necesaria cuando definimos la resolución-SLDNF para programas definitivos, es vital para la **correctez** de la resolución en los programas generales.

Correctez

Los arboles del bosque-SLDNF son llamados **arboles-SLDNF** (completos); y la secuencia de todas las metas en una rama de un árbol-SLDNF con raíz G es llamada derivación-SLDNF completa de G (bajo un programa Δ y una función de selección $\mathcal{R}$). El árbol etiquetado por G_0 es llamado árbol principal. Un árbol con la raíz $\leftarrow \alpha$ es llamado árbol subsidiario si $\neg\alpha$ es una literal seleccionada en el bosque (el árbol principal puede ser a su vez subsidiario).

Arboles-SLDNF

Ejemplo 6.17. *Consideren el siguiente programa general estratificado Δ:*

$$
\begin{array}{rcl}
base(X) & \leftarrow & en(Y,X), en_la_mesa(X). \\
en_la_mesa(X) & \leftarrow & \neg no_en_la_mesa(X). \\
no_en_la_mesa(X) & \leftarrow & en(X,Y). \\
encima(X,Y) & \leftarrow & en(X,Y). \\
encima(X,Y) & \leftarrow & en(X,Z), encima(Z,Y). \\
en(c,b). & & \\
en(b,a). & &
\end{array}
$$

El bosque-SLDNF para la meta $\leftarrow base(X)$ se muestra en la Figura 6.5. El árbol principal contiene una derivación fallida y una refutación con la substitución computada $\{X/a\}$.

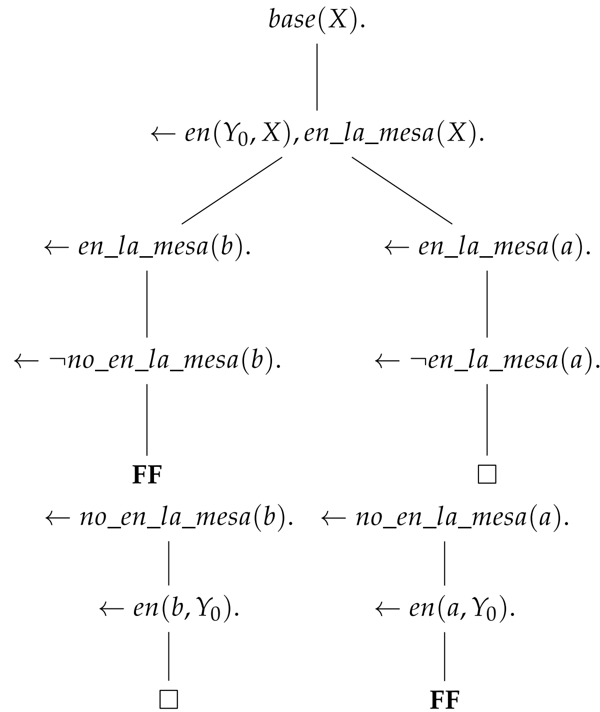

Figura 6.5: Bosque-SLDNF para la meta $\leftarrow base(X)$.

Las ramas de un árbol-SLDNF en un bosque-SLDNF representan todas las derivaciones-SLDNF completas de su raíz, con base en la función de selección dada. Hay cuatro clases de derivaciones-SLDNF completas:

1. Derivaciones infinitas;
2. Derivaciones finitas fallidas (terminan en **FF**);
3. Refutaciones (terminan en □); y
4. Derivaciones plantadas (si ninguno de los casos anteriores aplica).

Ejemplo 6.18. *Consideren el siguiente programa:*

$$
\begin{array}{rcl}
termina(X) & \leftarrow & \neg ciclo(X). \\
ciclo(X) & \leftarrow & ciclo(X).
\end{array}
$$

El bosque-SLDNF para el ejemplo anterior se muestra en la Figura 6.6. El bosque incluye una derivación plantada para $termina(X)$ y una derivación infinita para

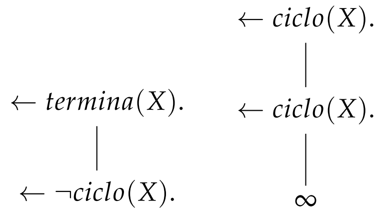

Figura 6.6: Bosque-SLDNF para la meta $\leftarrow paro(X)$.

$ciclo(X)$. Esto ilustra una de las razones por las cuales una derivación se planta: uno de sus arboles subsidiarios contiene sólo derivaciones fallidas o infinitas.

El siguiente programa también conduce a una derivación plantada (ciclo en el cómputo de la negación):

$$\begin{array}{rcl} paradoja(X) & \leftarrow & \neg ok(X). \\ ok(X) & \leftarrow & \neg paradoja(X). \end{array}$$

Intenten construir el bosque-SLDNF de este programa y observaran también que en este caso, la árbol principal es a su vez un árbol subsidiario.

La última razón para que una derivación quede plantada es ilustrada por el siguiente programa:

$$\begin{array}{rcl} tope(X) & \leftarrow & \neg bloqueado(X). \\ bloqueado(X) & \leftarrow & en(Y, X). \\ en(a, b). & & \end{array}$$

Es evidente que $tope(a)$ debería poder derivarse de este programa. Sin embargo, el árbol-SLDNF de la meta $\leftarrow tope(X)$ no contiene refutaciones. De hecho, esta meta se planta aún cuando $\leftarrow bloqueado(X)$ tiene una refutación. La razón de esto es que $\leftarrow bloqueado(X)$ no tiene ninguna derivación que termine con una substitución computada vacía. A la meta $\leftarrow \neg tope(X)$, Prolog no responde b, sino que ¡no todos los bloques están en el tope de la pila! De manera que la implementación de la resolución-SLDNF en la mayoría de los Prolog no es robusta, aunque nuestra definición si lo es.

Teorema 6.4 (Correctez de la resolución-SLDNF)**.** *Sea* Δ *un programa general y* $\leftarrow \alpha_1, \ldots, \alpha_n$ *una meta general. Entonces:*

- *Si* $\leftarrow \alpha_1, \ldots, \alpha_n$ *tiene una substitución de respuesta computada* θ*, entonces* $comp(\Delta) \models \forall(\alpha_1\theta \wedge \cdots \wedge \alpha_n\theta)$.
- *Si* $\leftarrow \alpha_1, \ldots, \alpha_n$ *tiene un árbol-SLDNF finito que falla, entonces* $comp(\Delta) \models \forall(\neg(\alpha_1 \wedge \cdots \wedge \alpha_n))$.

La definición de bosque-SLDNF no debe verse como una implementación de la resolución-SLDNF, sólo representa el espacio ideal de computación donde la correctez puede ser garantizada. Nada se ha dicho en cuanto al orden en que el bosque debe construirse. Al igual que en los demás casos, Prolog sigue una estrategia primero en profundidad: Ante una meta $\neg\alpha$, Prolog suspende la construcción del árbol de resolución en espera de la respuesta a la meta $\leftarrow \alpha$. El hecho de que Prolog no verifique si la substitución de respuesta de una refutación fue ϵ, al igual que la ausencia de chequeo de ocurrencias, contribuyen a que Prolog no sea robusto.

6.7 lecturas y ejercicios sugeridos

La discusión presentada aquí sobre corte y negación por fallo finito en Prolog, se basa en el libro de Bratko [19]. Los aspectos generales más técnicos sobre estos temas se basan en el libro de Nilsson y Maluszynski [88]. Apt y Bol [3] nos ofrecen una revisión de la negación en el contexto de la programación lógica.