Conferencia 4: Programación Declarativa

M.Sc. Dafne García de Armas

Curso 2018-2019

Tema: El corte y la negación en la Prolog

El corte: estructura de control explícita

Como vimos la programación lógica pura en su realización en Prolog resuelve el problema del control en la ejecución de algoritmos adoptando BPP-RC como la estrategia de navegación en el espacio de búsqueda. En verdad, el lector se puede percatar de que el árbol de derivación de un objetivo con respecto a un programa puede contener muchas ramas que terminan en fracasos, las que BPP-RC no puede evitar generar. No obstante, nuestro conocimiento sobre cómo el intérprete ejecuta nuestro programa puede ser utilizado para realizar "podas" de tales ramas estériles en la deducción del objetivo, lo cual resultaría en una mayor eficiencia.

Prolog suministra un predicado primitivo que permite al programador afectar la generación del árbol de deducción de un objetivo eliminando ramificaciones que a juicio del programador den lugar a ramas fracasos. Se trata de un predicado 0-ario, denominado *corte(cut)* y denotado por !/0. El corte puede ser introducido como un sub-objetivo más en el cuerpo de cualquier cláusula o de un objetivo, su valor como constante proposicional es verdadero, por lo tanto, como objetivo a evaluar en cualquier derivación el corte siempre triunfa, de ahí que no afecte para nada la lectura y semántica declarativas de un programa si es correctamente utilizado. Su efecto sobre la ejecución del intérprete es relevante: actúa como un mecanismo de control que reduce el espacio de búsqueda podando dinámicamente ramas del árbol de deducción de un objetivo con respecto a un programa que supuestamente no conducirían a soluciones. Veamos a través de un ejemplo la mecánica del corte.

Ejemplo: Considere el siguiente programa donde se define el predicado el_menor/3 para hallar el menor entre dos números (note que el predicado fracasa si los números son iguales):

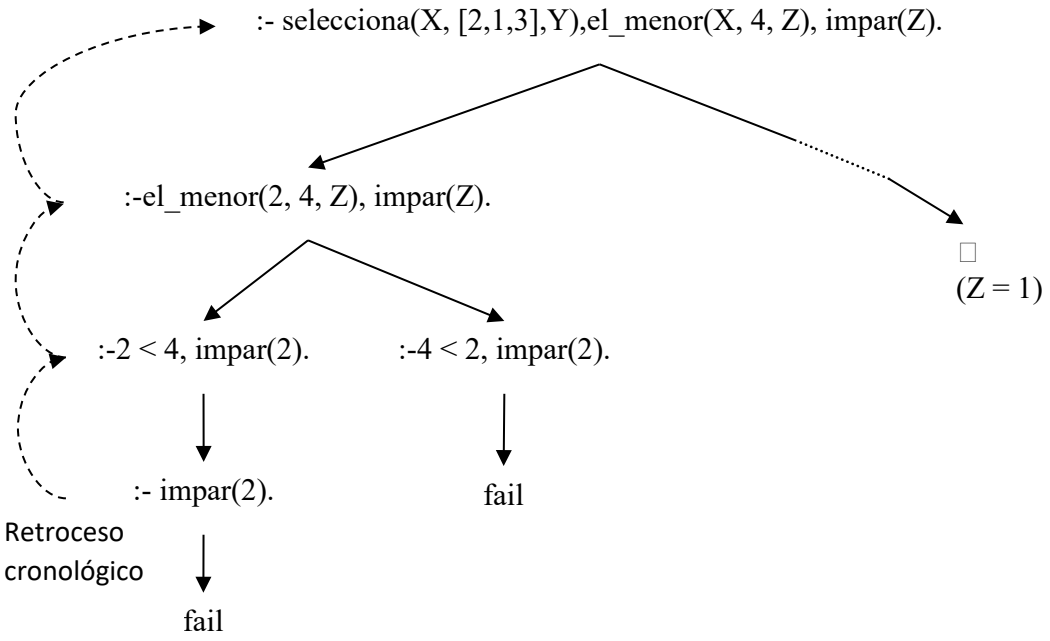
```
% el_menor(N1, N2, M) denota que M es el menor entre N1 y N2.

el_menor(X, Y, X) :- X < Y.      % </2 es un predicado numérico que es una
                                 % primitiva de la aritmética de Prolog con la
                                 %interpretación % usual.
el_menor(X,Y,Y) :- Y < X.
```

Considere el siguiente objetivo que triunfa, si puede seleccionar de una lista de números un número que cumpla con la condición de ser menor que 4 y ser impar:

```
:-selecciona(X, [2, 1, 3], Y), el_menor(X, 4, Z), impar(Z).
```

El árbol de deducción simplificado es el siguiente:



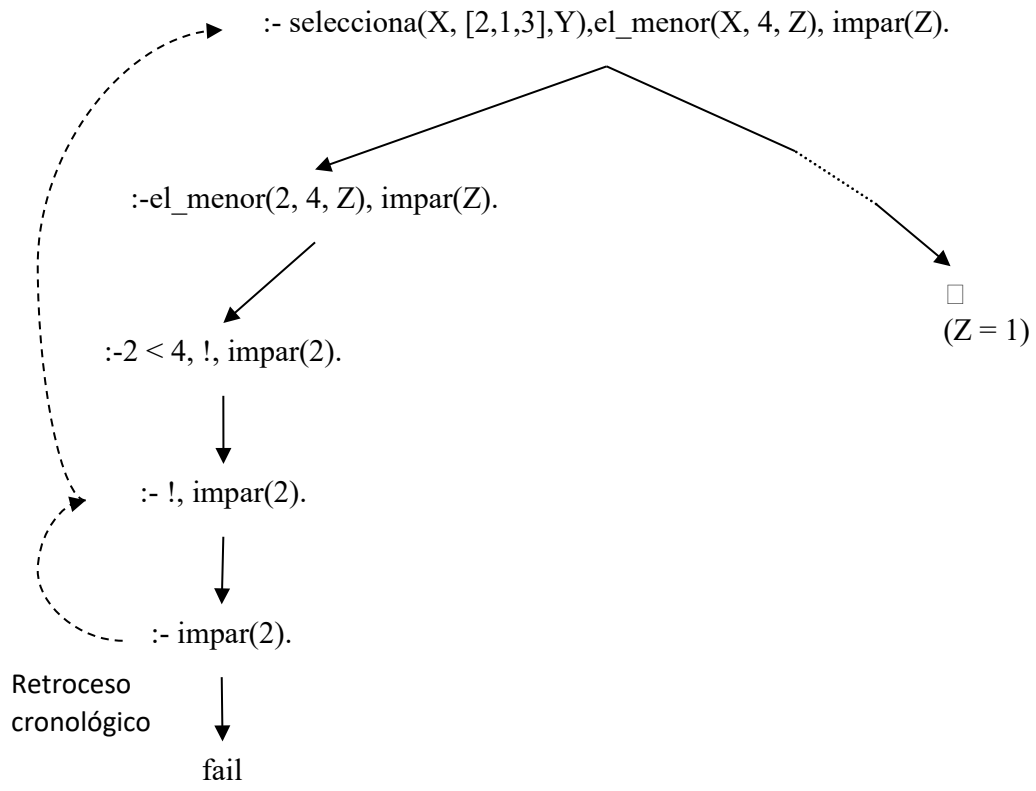
Se puede apreciar que el intérprete genera innecesariamente la rama intermedia que es un fracaso.

Considere ahora el siguiente programa del predicado el_menor/3, en cuya definición ocurre el corte.

```
el_menor(X,Y,X) :- X < Y, !.
el_menor(X,Y,Y) :- Y < X.
```

y evalúese nuevamente con este programa el objetivo

```
:-selecciona(X, [2,1,3],Y),el_menor(X, 4, Z), impar(Z).
```



Analizando el nuevo árbol de deducción se verifica lo siguiente:

Cuando algún objetivo a la derecha del corte fracasa (en el ejemplo: impar(2)), si el corte se alcanza por retroceso cronológico (como sucede en el ejemplo), esto trae como consecuencia que el intérprete:

1. No reevalúe ninguno de los sub-objetivos a la izquierda del corte (en el ejemplo: 2 < 4)
2. Tampoco selecciona ninguna de las restantes cláusulas que pudiesen aplicarse del predicado que introdujo el corte (en el ejemplo: la segunda cláusula de el_menor/3).
3. El intérprete reinicia el proceso de deducción en el sub-objetivo inmediatamente a la izquierda del sub-objetivo que introdujo el corte (en el ejemplo selecciona(X, [2, 1, 3], Y)).

PROLOG proporciona el operador de corte ! para interrumpir el backtracking, es decir, para cambiar la estrategia de control y no seguir buscando por ciertas ramas.

```
Obj :- Obj1, Obj2, Obj3, !, Obj4, Obj5, Obj6
```

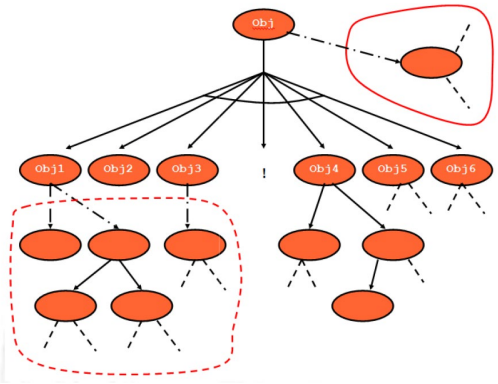
- Si falla Obj3, se vuelve atrás y se intenta Obj2 (backtracking).
- Si tiene éxito Obj3, pasa el corte y se satisface el objetivo Obj4, pero ya no se volverá a intentar Obj3
- Si falla Obj4, falla la regla.

Por tanto, ahora la regla puede fallar en dos situaciones: cuando Obj1 falla y cuando Obj4 falla

Una segunda consecuencia de haber pasado por el corte es que no se intenta aplicar ninguna otra regla con cabecera Obj.

Supongamos que Obj3 tiene éxito:

Las ramas que el corte impide explorar son las encerradas en círculo.



NOTA: Hasta que se consigue que Obj3 sea éxito, sí se explora parte del grafo: El backtracking está permitido antes de que Obj3 sea éxito

Control del no-determinismo

Como hemos visto una de las características más interesantes de la PL es la posibilidad de definir varias alternativas que puede tener un predicado (procedimiento) y el no determinismo asociado. Sin embargo, como hemos visto en el ejemplo desarrollado, pueden definirse predicados en los que la aplicación de una de las cláusulas por el intérprete para evaluar un objetivo excluye la aplicación de otras cláusulas del mismo predicado.

Considere como caso más convincente el siguiente predicado fusion_ordenada(X, Y, Z) donde Z es la lista ordenada de enteros que se obtiene al fundir las listas ordenadas de enteros X y Y:

```
fusion_ordenada(X, [], X).
fusion_ordenada( [], X, X).

fusion_ordenada([X |X1],[Y |Y1],[X |Z]):- X < Y,
fusion_ordenada(X1, [Y |Y1], Z).
fusion_ordenada([X |X1],[Y |Y1],[X, Y |Z]):-    X = Y,
                                                fusion_ordenada(X1, Y1, Z).
fusion_ordenada([X |X1],[Y |Y1],[Y |Z]):- X > Y,
                                          fusion_ordenada([X |X1],Y1,Z).
```

Observe que en las cláusulas que definen este predicado no sólo la unificación del objetivo con las cabezas de las cláusulas es determinante en su aplicación, también lo es el primer sub-objetivo en el cuerpo de cada regla que, como puede apreciarse, es una comparación de números. Se verifica entonces que una vez realizada la comparación, si esta resulta verdadera, se continúa con la aplicación del predicado fusion_ordenada/3 y resulta innecesario y por demás deficiente que el intérprete debido a su estrategia BPP-RC intente aplicar otra de las cláusulas si hubiese un fracaso posterior. Esto se debe además a que el predicado fusion_ordenada/2 es esencialmente funcional y la derivación de un objetivo de la forma:

```
:- fusion_ordenada([1, 3, 5], [2, 4], X).
```

tiene una sola solución. La introducción conveniente del corte en las cláusulas de este predicado permite declarar la aplicación mutuamente excluyente o determinista:

```
fusion_ordenada(X, [], X).
fusion_ordenada( [], X, X).

fusion_ordenada([X |X1],[Y |Y1],[X |Z]):- X < Y, !,
fusion_ordenada(X1, [Y |Y1],Z).
fusion_ordenada([X |X1],[Y |Y1],[X, Y |Z]):-    X = Y, !,
                                                fusion_ordenada(X1, Y1, Z).
fusion_ordenada([X |X1],[Y |Y1],[Y |Z]):- X > Y,
                                          fusion_ordenada([X |X1],Y1,Z).
```

Es posible utilizando el corte modificar la definición de un predicado existente de tal manera que se pueda obtener un nuevo predicado con distinta funcionalidad.

Ejemplo: Analice la siguiente modificación del predicado member/2 y determine qué nuevo predicado resulta ser member_1/2 y cuál es su diferencia con member/2.

```
member_1(X, [X| _]):- !.
member_1(X, [_|Y]):- member_1(X,Y).
```

Estos usos del corte no afectan la lectura declarativa de un programa.

Usos incorrectos del corte

Sin embargo puede haber usos incorrectos del corte que pueden atentar contra la corrección de un programa. Por ejemplo, utilizándolo para omitir condiciones en la definición de un predicado.

Analice las siguientes dos definiciones del predicado min/3:

```
% min(N1,N2;M): M el mínimo de los números N1 y N2

min1(X,Y,X):- X =< Y, !.
min1(X,Y,Y):- Y < X.

min2(X,Y,X):- X =< Y, !.
min2(X,Y,Y).
```

El lector verificará que la primera definición (min1/3) hace un uso correcto del corte. La segunda definición (min2/3) tiene toda la apariencia de un *"if then else"* al omitirse la comparación en la segunda cláusula. Considere ahora el siguiente objetivo:

```
:- min(2,5,5).
```

El lector verificará que con min1/3 se obtienen la respuesta negativa correcta mientras que con min2/3 se obtiene una respuesta afirmativa incorrecta.

El corte, como estructura de control explícita que puede ser de utilidad en lograr una mayor eficiencia en la ejecución de un programa debe ser utilizado sin afectar la interpretación proyectada de un predicado.

Debe enfatizarse que el uso del corte no es necesario: nada agrega a la capacidad computacional de Prolog, su uso es en aras de la eficiencia en la búsqueda y su empleo debe reducirse a este fin.

Ejemplo:
Supongamos que, en una biblioteca, si una persona tiene prestado un libro, sólo se le permite consultar la base de datos y leer libros dentro del recinto. En caso contrario, también se le permite el préstamo a domicilio, así como préstamos de otras librerías.
Las consultas a la BD son servicios básicos:

```
servicio_basico(consulta_BD).
servicio_basico(consulta_interna).
```

Las consultas externas son servicios adicionales:

```
servicio_adicional(prestamo_interno).
servicio_adicional(prestamo_externo).
```

Los servicios completos son los básicos y los adicionales.

```
servicio_completo(X) :- servicio_basico(X).
servicio_completo(X) :- servicio_adicional(X).
```

Ahora, si el cliente tiene un libro prestado, se le ofrecen sólo los servicios básicos:

```
servicio(Cliente,Servicio) :- prestado(Cliente,_), ! , servicio_basico(Servicio).
```

A todos los clientes que no tengan prestado un libro, se les ofrecen todos los servicios:

```
servicio(_,Servicio) :- servicio_completo(Servicio).
```

Por ejemplo:

```
cliente('Juan').
cliente('Pedro').
cliente('Maria').
…
prestado('Juan',clave107).
prestado('Pedro',clave145).
?- servicio('Juan',Serv).
Serv = consulta_BD;
Serv = consulta_interna;
no.
?- servicio('Maria',Serv).
Serv = consulta_BD;
Serv = consulta_interna;
Serv = prestamo_interno;
```

```
Serv = prestamo_externo;
no.
```

Sin embargo, si ponemos:

```
?- servicio(Cliente,Serv).
Cliente = 'Juan'
Serv = consulta_BD;
Cliente = 'Juan'
Serv = consulta_interna;
no.
```

Sólo aparecen los servicios disponibles para Juan (no salen los básicos de Pedro ni los completos de María) al no haber backtracking después del corte:

```
servicio(Cliente,Servicio) :- prestado(Cliente,_), ! , /* Una vez llegados
aquí, no hay vuelta atrás */
servicio_basico(Servicio).
```

El uso del corte en reglas cuya cabecera puede venir con variables no instanciadas, puede dar problemas.

Para solucionar el problema, utilizamos un predicado auxiliar que nos "separe" la aplicación del corte del resto del proceso de búsqueda:

```
servicioAUX(Cliente,Servicio) :- prestado(Cliente,_), ! , servicio_basico(Servicio).
servicioAUX(_,Servicio) :-
servicio_completo(Servicio).
servicio(Cliente,Servicio) :-
cliente(Cliente),
servicioAUX(Cliente,Servicio).
```

- Si Cliente viene no instanciado, el predicado cliente(Cliente) fijará un cliente y buscará sus servicios. Al hacer backtracking ,volverá a buscar otro cliente (y así con todos.)
- Si Cliente viene instanciado, el predicado cliente(Cliente) tendrá éxito si es un cliente fichado y fallará en otro caso.

EJEMPLO

Sobre una base de datos de contribuyentes, queremos definir los contribuyentes normales como aquellos que:
- No son extranjeros.
- Si están casados, su cónyuge no ingresa más de 2000.
- En otro caso, sus ingresos propios están entre 700 y 2000.

Primer intento:

```
cont_normal(X) :- extranjero(X),fail.
cont_normal(X) :- ingresosNormales(X).
?- cont_normal('John Wiley').
yes !!!! <--- Falla porque PROLOG aplica la segunda regla
```

Solución: Combinación corte-fallo.

```
cont_normal(X) :- extranjero(X), ! , fail.
cont_normal(X) :- casado(X,Y) , ingresos(Y,Ing), Ing>2000 , ! , fail.
cont_normal(X) :- ingresos(X,Ing) , Ing>700, Ing<2000.
```

La negación en Prolog (\+)

Conocimiento positivo y negativo

Considere los siguientes ejemplos de lo que puede denominarse *conocimiento positivo (negativo)*:

Ejemplo 1: El menú de un restaurante informa de las comidas que ofrece (conocimiento positivo) y no informa de las comidas que no ofrece (conocimiento negativo).

Ejemplo 2: La pizarra de una terminal de ómnibus anuncia las salidas que se realizan (conocimiento positivo) pero no informa de las salidas que no se realizan (conocimiento negativo).

Llámense tablas tanto al menú como a la pizarra. Diremos que en ambos casos las tablas muestran *explícitamente* sólo el conocimiento positivo. Tal forma de suministrar conocimiento (información) es típica ente los seres humanos, debido a que:

Los seres humanos son capaces de razonar y suministrar respuestas negativas a partir de conocimiento positivo, basándose en la presuposición socialmente aceptada de que el conocimiento positivo que expresan las tablas contiene implícitamente el conocimiento negativo, el cual puede hacerse explícito por el razonamiento.

Así, por ejemplo, si un cliente inspeccionase la tabla para verificar si consigna una determinada comida (salida) y no la encontrase haría explícito, es decir, *inferiría* el conocimiento negativo de que no hay tal comida (salida).

El razonamiento anterior se basa en la siguiente regla de inferencia que lo caracteriza: lo que no se declara como un hecho (conocimiento positivo) es un hecho falso, por lo tanto, la negación de un hecho (conocimiento negativo) es verdadera si el hecho no está declarado. (*Hipótesis del Mundo Cerrado (HMC)(Closed World Hipótesis (CWH*)).

A veces, con fines "pragmáticos" el principio se hace de alguna manera explícito mediante algún enunciado asociado a la tabla, tal como "comida (salida) que no está en la tabla no se ofrece" (fin pragmático: no pregunte por lo que no está en la tabla).

Si se analiza un programa lógico, por ejemplo, el programa Familia, se verá que se trata de un conjunto de cláusulas (hechos, reglas) que enuncian solo conocimiento positivo. Las cláusulas definidas de un programa lógico no permite expresar conocimiento negativo: los hechos son literales positivos cuya verdad se afirma en el programa. Las reglas del programa contienen sólo literales positivos en la cabeza y en el cuerpo, en esencia, las reglas son implicaciones que sólo permiten derivar hechos positivos.

Sin embargo, no es posible prescindir de la negación para representar y procesar conocimiento acerca de muchos problemas. No se puede por ejemplo preguntar negativamente al programa familia.

Tampoco es posible escribir un programa definitorio de listas disjuntas, es decir, es decir de listas que no tienen ningún elemento en común.

El hecho de que un programa lógico sólo representa conocimiento positivo, permite afirmar que *un programa lógico tiene siempre un modelo*, el más ubicuo, su base Herbrandiana. Finalmente, las preguntas (teoremas) que se someten a un programa lógico representan también conocimiento positivo (no confundir la pregunta con su transformación en objetivo negándola para que el intérprete proceda a su refutación), por lo tanto el programa solo suministra respuestas positivas cuando hay triunfo, no fracasa el objetivo o el proceso de su derivación es infinito.

Razonamiento monotónico y no monotónico

Analícese nuevamente las Teorías de Primer orden (TPO) como las estudiadas en la Conferencia 2: Si $\boldsymbol{T}$ son los axiomas de una TPO, los teoremas que pueden derivarse de $\boldsymbol{T}$ no se invalidan jamás: Eliminar axiomas de una TPO puede dar lugar a que algunos teoremas de la teoría no puedan demostrarse, no a que resulten falsos y, lo más importante, la extensión $\boldsymbol{T'}$ de una teoría $\boldsymbol{T}$ con un conjunto de nuevas proposiciones no invalida los teoremas que previamente se hayan demostrado o que pudieran demostrarse solamente con $\boldsymbol{T}$.

Ejemplo: Considere la teoría $\boldsymbol{T}$ expresada en el programa Familia. Si se adiciona un nuevo axioma familiar, como pudiera ser abuela/2 o hermano/2, se obtiene una nueva teoría $\boldsymbol{T'}$, donde nuevas proposiciones podrán ser demostradas pero ninguna proposición demostrable a partir de $\boldsymbol{T}$ podrá ser invalidada.

Esta característica de las TPOs estudiadas se debe a una propiedad del concepto de consecuencia lógica subyacente a las TPOs, denominada monotonicidad:

Formalmente, si el concepto de consecuencia lógica que una lógica define satisface la siguiente propiedad:

$$\boldsymbol{T} \mid\!\!- A \text{ entonces } \boldsymbol{T} \cup \boldsymbol{T'} \mid\!\!- A$$

se dice que la lógica es *monotónica*. La lógica hasta aquí estudiada en este curso y la que hemos desarrollado hasta ahora para la PL es monotónica.

Volviendo a los ejemplos de tablas de comida (salida), suponga, como suele ocurrir, que las tablas son regularmente actualizadas y se eliminan comidas (salidas) que aparecían previamente y se añaden nuevas comidas (salidas). El resultado es que en cada actualización (comidas) salidas que antes eran hechos falsos ahora son verdaderos y (comidas) salidas que antes eran hechos verdaderos ahora son falsos. Resulta entonces que las tablas son *teorías en constante revisión* a las cuales no las caracteriza una lógica monotónica, y es que el razonamiento ordinario o de sentido común de los seres humanos es esencialmente un razonamiento *no-monotónico*. Esa es la consideración más expandida entre los especialistas de IA, de ahí que en las últimas décadas se haya puesto una gran atención al estudio y formalización del razonamiento no-monotónico dando lugar a las denominadas lógicas no-monotónicas.

No es necesario argumentar aquí lo importante que resulta extender la PL de conocimiento positivo a una PL que permita de alguna manera la representación y el procesamiento lógico de conocimiento

negativo. Los desarrollos que siguen establecen la forma en que Prolog ha sido habilitado para esta extensión.

Se introduce en Prolog la operación proposicional de la negación como un predicado unario *not/1* (suministrado con notaciones alternas en los intérpretes de Prolog en uso, casi siempre \+).

¿Cuál puede ser la interpretación que defina semánticamente a not/1 en Prolog?. Evidentemente no puede ser la interpretación de la negación en LPO ya estudiada dado que como se ha visto la sintaxis no permite la afirmación de hechos negativos y los predicados se definen intencionalmente mediante reglas que son implicaciones en un solo sentido, es decir, las reglas declaran la parte "si" de una definición, no la parte "solo si".

No obstante nada impide extender la sintaxis de Prolog introduciendo objetivos negados y también realizar definiciones de predicados utilizando la negación en el cuerpo de las cláusulas, teniendo en cuenta que cuando el intérprete aplica una cláusula de un programa, su cuerpo se convierte en parte del objetivo.

¿Cuál sería una definición admisible del predicado not/1? Aprovechando los fracasos se podría quizás introducir la HMC como una "regla de inferencia" más en el intérprete de Prolog que se ocuparía de interpretar la negación. Su definición como tal podría ser la siguiente:

Sea P un programa y G un literal, entonces

Existe refutación-SLD de $P \cup \{:\text{- not } G\}$ *si y sólo si* $P \cup \{:\text{-}G\}$ *fracasa*

Pero los resultados de decidibilidad de la LPO estipulan que $P \cup \{:\text{-}G\}$ *es en general no decidible,* y Prolog tiene una semántica operacional basada en el Principio de Resolución-SLD que es un semi-algoritmo, luego se adopta una versión más débil que la regla anterior a la cual se denomina *Negación como Fracaso Finito (NFF)*[Clark]:

Existe refutación-SLD de $P \cup \{:\text{- not } G\}$ *si y sólo si* $P \cup \{:\text{-}G\}$ *fracasa finitamente*

Además G deberá ser básico (recordatorio: no hay ocurrencia de variables en G).

Al añadir tal regla al intérprete denominaremos a su nuevo mecanismo de inferencia Resolución-SLDNFF.

Podemos decir que la definición de not sería como sigue:

```
not (X) :- X,!,fail.
not (_).
```

- Si X tiene éxito, la primera regla nos dice que not(X) debe fallar.
- Si X falla para todos los posibles valores de X, la segunda regla nos dice que not(X) tiene éxito

Ejemplos:

1. se verifica que FAMILIA ∪ {:- abuelo(a, e)} fracasa finitamente, luego aplicando la regla anterior se tiene que FAMILIA ∪ {:- not abuelo(a, e)} triunfa.

2. el lector analizará la siguiente definición del predicado disjuntas/2 que verifica si dos listas no tienen ningún elemento en común,

```
disjuntas(Lista1, Lista2):-not((member(X, Lista1), member(X, Lista2))).
```

3. analice la propuesta de una definición del predicado union/3 que halla la unión de dos listas

```
union([],X,X).
union([X|Y],Z,[X|W]) :- not(member(X,Z)) , union(Y,Z,W).
union([X|Y],Z,W) :- union(Y,Z,W).
```

Análisis de la regla NFF

Denominaremos *conjunto de fracasos finitos* de un programa lógico P al conjunto de objetivos G tal que G fracasa finitamente con respecto a P. Luego un objetivo :- not(G) es una consecuencia de un programa P por la regla NFF si G pertenece al conjunto de fracasos finitos de P.

Se verifica fácilmente que si un programa P consta solamente de hechos, la regla NFF es equivalente a HMC.

Se tiene en resumen la siguiente relación: *el conjunto de fracasos finitos de un programa P es un subconjunto del complemento del conjunto de los triunfos de P.*

La NFF convierte a Prolog en una cierta realización del razonamiento no-monotónico.

No es posible en esta conferencia introductoria desarrollar toda la problemática asociada a la introducción de la negación en la programación lógica y que se confunde con los problemas asociados en lógica e IA a la modelación del razonamiento monotónico.