



UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

PARADIGMAS DE PROGRAMACIÓN

PARADIGMA LÓGICO

Dr. Pablo Vidal

Facultad de Ingeniería
Universidad Nacional de Cuyo

2023

- 1 Fundamentos Teóricos
- 2 Principio SLD
- 3 Estrategia de Búsqueda
- 4 Corte y Negación

Fundamentos Teóricos

Objetivos

❶ Principio de Resolución - SLD

- Tipos de derivaciones - SLD
- Árbol de derivación - SLD

❷ Estrategia de búsqueda BPP-RC.

Introducción

En esta clase desarrollaremos el **procedimiento** mediante el cual el **intérprete** de Prolog **evalúa un objetivo** con respecto a un programa.

El procedimiento se basa en una regla de inferencia denominada **Principio de Resolución**, la que es utilizada en asociación con el proceso de **unificación** ya estudiado para computar los valores de las variables en un objetivo.

Reglas de inferencia deductiva

Constituyen el método fundamental de generación y justificación lógica de pasos en una demostración y tienen la siguiente estructura:

. **Premisas**
.
.
.
 Conclusión

donde **Premisas** denota un conjunto de pasos previos, por lo mismo ya justificados, que ocurren en una demostración y **Conclusión** denota el nuevo paso que se inserta en la demostración al aplicar la regla.

Modus Ponens (MP)

Todo proceso de deducción consta al menos implícitamente de una regla de inferencia deductiva denominada *regla de separación* o *modus ponens* cuyo esquema general es el siguiente:

$$\begin{array}{l}
 \cdot \qquad \qquad \mathbf{T} \mid - \mathbf{A} \\
 \cdot \qquad \qquad \mathbf{T} \mid - \mathbf{A} \Rightarrow \mathbf{B} \\
 \cdot \qquad \qquad \hline
 \cdot \qquad \qquad \mathbf{T} \mid - \mathbf{B}
 \end{array}$$

donde \mathbf{T} denota una teoría y \mathbf{A} , \mathbf{B} son fórmulas y cuya lectura puede ser la siguiente: A partir de que en una demostración exista un paso de la forma $\mathbf{T} \mid - \mathbf{A}$ y otro paso de la forma $\mathbf{T} \mid - \mathbf{A} \Rightarrow \mathbf{B}$, entonces se puede insertar $\mathbf{T} \mid - \mathbf{B}$ como un nuevo paso en la demostración. Note que $\mathbf{T} \mid - \mathbf{A}$ y $\mathbf{T} \mid - \mathbf{A} \Rightarrow \mathbf{B}$ son las premisas de la regla y $\mathbf{T} \mid - \mathbf{B}$ su conclusión.

Principio de Resolución (PR)

En el área de la Ciencia de la Computación que estudia la automatización de la demostración y el razonamiento en general fue definida por **Robinson [1966]** una regla de inferencia denominada **Principio de Resolución (PR)**, la cual se aplica a fórmulas en forma clausal. Su versión más simple para la lógica proposicional es la siguiente (siendo A un literal, B y C cláusulas):

$$\begin{array}{l}
 \cdot \qquad \qquad \mathbf{T} \mid - \mathbf{A} \vee \mathbf{B} \\
 \cdot \qquad \qquad \mathbf{T} \mid - \neg \mathbf{A} \vee \mathbf{C} \\
 \cdot \qquad \qquad \hline
 \cdot \qquad \qquad \mathbf{T} \mid - \mathbf{B} \vee \mathbf{C}
 \end{array}$$

Considere el programa familia

```
(P1) padre(luis,alicia).
(P2) padre(luis,josé).
(P3) padre(jose,ana).
(M1) madre(alicia,dario).
(A1) abuelo(X,Y) :- padre(X,Z),madre(Z,Y).
(A2) abuelo(X,Y) :- padre(X,Z),padre(Z,Y).
```

Observe que ...

- (P1)-(P3) definen **extensionalmente** la relación padre/2 . Se afirma entre que pares de individuos se mantiene la relación en este programa (lo mismo puede decirse de la relación madre/2)
- (A1)-(A2) definen **intencionalmente** la relación abuelo/2 . Se define la relación abuelo/2 en términos de la relación padre/2 y madre/2 que están definidas extensionalmente en el programa

Intérprete de Prolog

Como lenguaje de programación basado en la lógica, Prolog es eminentemente **declarativo**, como se ha dicho, es uno de los exponentes fundamentales de la **programación declarativa**.

Toda fórmula de LPO es un enunciado o proposición declarativa. Las cláusulas de un programa definen predicados o relaciones, es decir, responde al **Qué?**.

Pero para computar soluciones, es decir, para evaluar un objetivo tal como :- abuelo(X,darío) (¿Quién es el abuelo de Darío?) a partir del programa Familia el intérprete de Prolog se comporta como una caja negra a la pregunta **Cómo evaluar?** Revelemos esta caja negra.

Principio de Resolución-SLD (PR-SLD)

Desarrollaremos la versión específica del principio de resolución que utiliza Prolog, denominada Resolución-SLD.

Sea G_i el objetivo : $-A_1, \dots, A_m, \dots, A_k$.

C_{i+1} la cláusula $A : -B_1, \dots, B_q$

R una regla de selección,

entonces un nuevo objetivo G_{i+1} se deriva de G_i y de C_{i+1} usando el umg σ_{i+1} mediante la regla de selección R, si se cumplen las siguientes condiciones:

i) A_m es el átomo seleccionado por la regla de selección R,

ii) $A_m \sigma_{i+1} = A \sigma_{i+1}$ (siendo σ_{i+1} un umg de A_m y A),

iii) G_{i+1} es el objetivo : $-(A_1, \dots, A_{m-1}, B_1, \dots, B_q, A_{m+1}, \dots, A_k) \sigma_{i+1}$

G_{i+1} es un resolvente de G_i y C_{i+1} . G_i y C_{i+1} se denominan las premisas o cláusulas padres y G_{i+1} la conclusión o resolvente de la derivación.

Ejemplo de aplicación de la Resolución-SLD:

$$G_0 : :- \textcolor{blue}{p}(\textcolor{red}{a}), \textcolor{blue}{q}(\textcolor{red}{b}).$$

$$C_1 : \textcolor{blue}{p}(X) :- \textcolor{red}{r}(X, \textcolor{red}{a}).$$

$$G_1 : :- \textcolor{red}{r}(\textcolor{red}{a}, \textcolor{red}{a}), \textcolor{blue}{q}(\textcolor{red}{b}).$$

A partir de C_1 seleccionando mediante R el sub-objetivo a la extrema izquierda de G_0 y usando el umg $\sigma_1 = \{a/X\}$.

Principio SLD

Principio de Resolución SLD

Las siglas “**SLD**” se refieren a las siguientes características de la especialización del principio de resolución en la programación lógica:

S: señala la utilización de una **regla de selección** R del sub-objetivo en el objetivo como premisa. La regla elimina el indeterminismo en la selección del sub-objetivo.

L: la resolución es **lineal**, en el sentido de que siempre se toma como cláusula padre para la próxima aplicación de resolución el último resolvente obtenido, en nuestro caso, siempre el último objetivo generado.

D: apunta a la utilización sólo de **cláusulas definidas** en la aplicación de resolución.

Observación

Al seleccionar una cláusula es necesario renombrar todas las variables que ocurren en la misma con nuevas variables que no han ocurrido en el desarrollo de la derivación, con lo cual se evita principalmente un injustificable fracaso del proceso de unificación por chequeo de ocurrencia.

Para evitar que cada lector cree su propio método de renombrar variables, adoptaremos el siguiente método para así lograr uniformidad en los futuros procesos de interpretación de un objetivo.

Método de renombramiento de variables

Dada una cláusula C de un programa Prolog, la cual ha sido identificada con un nombre N , la primera variante de dicha cláusula se llamará N_1 , es decir agregar al nombre N el número 1 y a toda variable que aparezca en C se le adiciona este nuevo identificador. Posteriormente, cada vez que se necesite una i -ésima variante de C ($i > 1$) la nueva variante se denominará N_i y a toda variable que aparezca en C se le adiciona este nuevo identificador.

Ejemplos:

(A2) `abuelo(X,Y) :- padre(X,Z),padre(Z,Y).`

La primera variante de cláusula de A2 es:

(A21) `abuelo(XA21,YA21) :- padre(XA21,ZA21),padre(ZA21,YA21).`

La segunda variante de cláusula de A2 es:

(A22) `abuelo(XA22,YA22) :- padre(XA22,ZA22),padre(ZA22,YA22).`

Derivación-SLD

Definición: Sea P un programa, sea G un objetivo y sea R una regla de selección. Una *derivación-SLD* a partir de $P \cup G$ mediante R es una sucesión finita o infinita $G = G_0, G_1, \dots$ de objetivos, una sucesión C_0, C_1, \dots de variantes de cláusulas en P que no comparten variables y una sucesión $\sigma_0, \sigma_1, \dots$ de umg's, tales que cada G_{i+1} se deriva por resolución-SLD de G_i y de $C_i + 1$ usando $\sigma_i + 1$ mediante R .

Encontramos tres tipos fundamentales de derivaciones-SLD:

- Refutación-SLD
- Derivación-SLD fracaso
- Derivación-SLD infinita

Refutación-SLD

Considere el programa Familia $\cup \{:- \text{abuelo}(\text{luis}, \text{darío.})\}$

(1.1) $:- \text{abuelo}(\text{luis}, \text{darío.})$.

(1.2) A11: $\text{abuelo}(\text{XA11}, \text{YA11}) :- \text{padre}(\text{XA11}, \text{ZA11}), \text{madre}(\text{ZA11}, \text{YA11})$.

$\sigma_1 = \{\text{luis}/\text{XA11}, \text{darío}/\text{YA11}\}$ –unificador entre el objetivo (1.1) y la variante A11 (1.2)

(1.3) $:- \text{padre}(\text{luis}, \text{ZA11}), \text{madre}(\text{ZA11}, \text{darío.})$ –resolvente de (1.1) y (1.2)

(1.4) P11: $\text{padre}(\text{luis}, \text{alicia.})$.

$\sigma_2 = \{\text{alicia}/\text{ZA11}\}$ –unificador entre (1.4) y el primer sub-objetivo de (1.3)

(1.5) $:- \text{madre}(\text{alicia}, \text{darío.})$ –resolvente de (1.3) y (1.4)

(1.6) M11: $\text{madre}(\text{alicia}, \text{darío.})$.

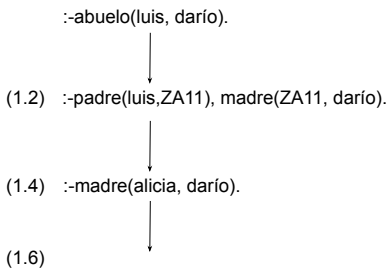
$\sigma_3 = \epsilon$ –unificador entre (1.5) y (1.6): cuando no hay variables, el unificador es la sustitución idéntica.

(1.7) –resolvente de (1.5) y (1.6), es la cláusula vacía

Refutación-SLD

Como no hay variables que instanciar en el objetivo el intérprete se limita a dar como respuesta un “sí”.

La deducción en forma de árbol muestra la característica lineal de la aplicación de la resolución-SLD:



Sustitución (respuesta) computada de una derivación-SLD

Considere el conjunto de cláusulas $\text{Familia} \cup \{:- \text{abuelo}(\text{luis}, X).\}$. Nuestro objetivo con este programa no se satisface al demostrar que $\exists(X)\text{abuelo}(\text{luis}, X)$ es una consecuencia lógica de familia. En verdad nuestro objetivo es una solicitud (query) al programa para que nos proporcione como valor de X el nombre de algún nieto de Luis. Como podemos verificar en la siguiente derivación, la unificación computa este valor para X .

Ejemplo

Computar el valor X del objetivo $:-\text{abuelo}(\text{luis}, X)$.

(2.1) $:- \text{abuelo}(\text{luis}, X)$.

(2.2) $A11: \text{abuelo}(XA11, YA11) :- \text{padre}(XA11, ZA11), \text{madre}(ZA11, YA11)$.

$\sigma_1 = \{\text{luis}/XA11, X/YA11\}$ –unificador entre (2.1) y (1.2), recordar que la sustitución $X/YA11$ se lee: “YA11 será sustituida por X ”

(2.3) $:- \text{padre}(\text{luis}, ZA11), \text{madre}(ZA11, X)$. –resolvente de (2.1) y (2.2)

(2.4) $P11: \text{padre}(\text{luis}, \text{alicia})$.

$\sigma_2 = \{\text{alicia}/ZA11\}$ –unificador entre (2.4) y el primer sub-objetivo de (2.3)

(2.5) $:-\text{madre}(\text{alicia}, X)$. –resolvente de (2.3) y (2.4)

(2.6) $M11: \text{madre}(\text{alicia}, \text{darío})$.

$\sigma_3 = \{\text{dario}/X\}$ –unificador entre (2.5) y (2.6)

(2.7) –resolvente de (2.5) y (2.6)

Sustitución computada de la derivación-SLD

Hallando la composición de los tres umg generados en la derivación se tiene la sustitución computada de la derivación-SLD

$$\sigma_1\sigma_2\sigma_3 = \{\text{luis}/XA11, \mathbf{darío}/YA11, \text{alicia}/ZA11, \text{dario}/X\}$$

Luego, además de suministrar la deducción de `abuelo(luis, X)`, la derivación-SLD a través del proceso de unificación que se desencadena ofrece un valor para las variables que ocurran en el objetivo G_0 , mediante su sustitución computada, en el caso que nos ocupa, $\mathbf{X} = \mathbf{darío}$.

Derivación-SLD fracasada o fracaso

Considere el programa Familia $\cup \{:- \text{abuelo}(\text{luis}, \text{alicia})\}$.

(3.1) $:- \text{abuelo}(\text{luis}, \text{alicia})$.

(3.2) A11: $\text{abuelo}(\text{XA11}, \text{YA11}) :- \text{padre}(\text{XA11}, \text{ZA11}), \text{madre}(\text{ZA11}, \text{YA11})$.

$\sigma_1 = \{\text{luis}/\text{XA11}, \text{alicia}/\text{YA11}\}$ –unificador entre (3.1) y (3.2)

(3.3) $:- \text{padre}(\text{luis}, \text{ZA11}), \text{madre}(\text{ZA11}, \text{alicia})$. –resolvente de (3.1) y (3.2)

(3.4) P11: $\text{padre}(\text{luis}, \text{alicia})$.

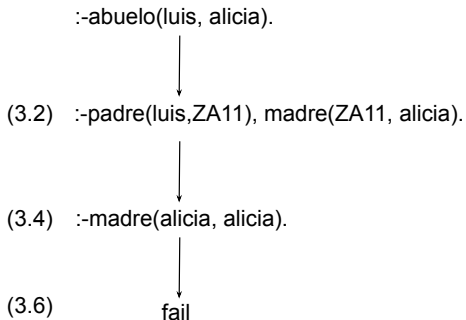
$\sigma_2 = \{\text{alicia}/\text{ZA11}\}$ –unificador entre (3.4) y el primer sub-objetivo de (3.3)

(3.5) $:- \text{madre}(\text{alicia}, \text{alicia})$. –resolvente de (3.3) y (3.4)

(3.6) fail –no existe una variante de cláusula que unifique con (3.5) para continuar la derivación

Derivación-SLD fracasada o fracaso

La deducción representada en forma de árbol muestra la característica lineal de la aplicación de resolución:



Derivación-SLD infinita

Si una derivación-SLD no es una refutación-SLD o un fracaso, entonces es una derivación-SLD infinita. Más adelante veremos casos de derivaciones infinitas.

Espacio de búsqueda: árbol de derivación-SLD

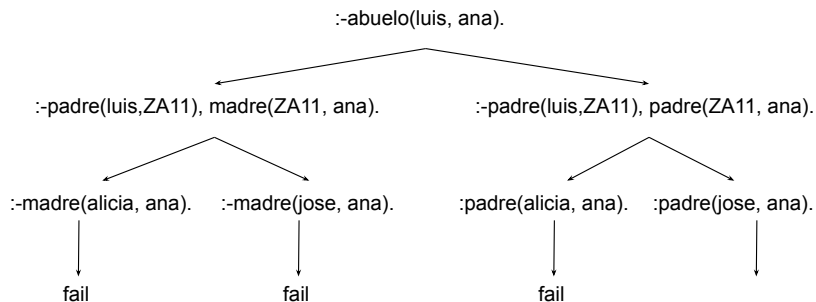
Dado $P \cup G$, lo primero es tener una representación en potencia del espacio de búsqueda del objetivo G . La siguiente definición ofrece una solución a este problema.

Definición: $P \cup G$, el *árbol de derivación-SLD* de G es un árbol etiquetado que satisface las siguientes condiciones:

- El nodo raíz tiene como etiqueta a G , que denominaremos G_0 .
- Si el árbol contiene un nodo etiquetado por G_i y existe una variante de cláusula $C_i + 1$ de $C \in P$ tal que $G_i + 1$ es un resolvente de G_i y $C_i + 1$ mediante la regla de selección R , entonces existe un nodo hijo de G_i etiquetado por $G_i + 1$. El arco que conecta a ambos nodos tiene como etiqueta a $C_i + 1$.

Ejemplo

Árbol de derivación-SLD para Familia $\cup \{:- \text{abuelo}(\text{luis}, \text{ana})\}$.



Espacio de búsqueda: árbol de derivación-SLD

Se puede adoptar, una **regla de selección fija** del sub-objetivo a expandir. En el árbol dado la regla adoptada es **expandir el primer sub-objetivo o el sub-objetivo a la extrema izquierda**.

Luego, dada una regla de selección R fija, cada rama del árbol de derivación-SLD de G corresponde a una derivación-SLD de G a partir de P , existiendo una correspondencia biunívoca entre el conjunto de las derivaciones-SLD de G y las ramas del árbol de derivación-SLD de G .

Por lo anterior, dado $P \cup G$, **el árbol de derivación-SLD** de G constituye una **representación estructurada del espacio de búsqueda de G** .

Concatenar

Especificación: $\text{conc}(A,B,C)$ se verifica si C es la lista obtenida escribiendo los elementos de la lista B a continuación de los elementos de la lista A . Por ejemplo,

$?- \text{conc}([a,b],[b,d],C).$

$C = [a,b,b,d]$

Definición 1:

$\text{conc}(A,B,C) :- A=[], C=B.$

$\text{conc}(A,B,C) :- A=[X|D], \text{conc}(D,B,E), C=[X|E].$

Definición 2:

$\text{conc}([],B,B).$

$\text{conc}([X|D],B,[X|E]) :- \text{conc}(D,B,E).$

Ejemplo Lista

¿Cuál es el resultado de concatenar las listas [a,b] y [c,d,e]?

```
?- conc([a,b],[c,d,e],L).
```

```
L = [a, b, c, d, e]
```

¿Qué lista hay que añadirle a la lista [a,b] para obtener

```
[a,b,c,d]?
```

```
?- conc([a,b],L,[a,b,c,d]).
```

```
L = [c, d]
```

¿Qué dos listas hay que concatenar para obtener [a,b]?

```
?- conc(L,M,[a,b]).
```

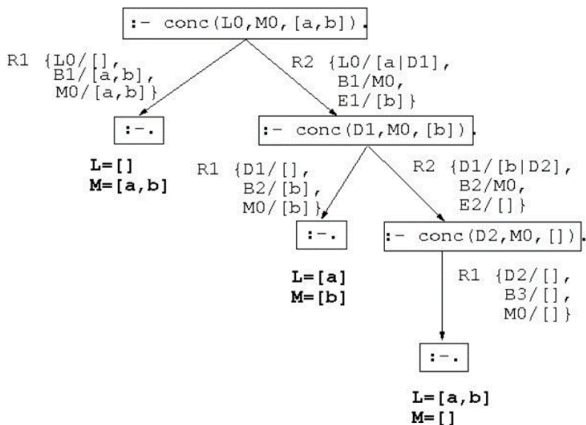
```
L = [] M = [a, b] ;
```

```
L = [a] M = [b] ;
```

```
L = [a, b] M = [] ;
```

No

Árbol de derivación



Estrategia de Búsqueda

Estrategia de Búsqueda

Dada la forma de árbol que presenta la estructura de un espacio de búsqueda de un objetivo, podemos navegar por dicho espacio generando paulatinamente el árbol de derivación-SLD, utilizando una estrategia de búsqueda en árbol que resulte conveniente.

Ejemplo:

- Búsqueda Primero-a-lo-Ancho (BPA) (Breadth-First Search)
- Búsqueda Primero-en-Profundidad (BPP) (Depth-First Search)

Estrategia de Búsqueda en PROLOG

La estrategia estándar utilizada en PROLOG para generar el árbol de búsqueda de un objetivo es una particularización de BPP que aplica conjuntamente la estrategia de búsqueda **retroceso cronológico (chronological backtrack)** (RC), la cual denotaremos por BPP-RC.

BPP-RC

- Se adopta como **regla de selección** la que selecciona el **sub-objetivo a la extrema izquierda** de un objetivo.
- Se toman las **cláusulas** del programa en un **orden fijo**, el orden lineal consecutivo en el cual aparecen.
- Se aplica **en cada oportunidad una sola cláusula** del programa al objetivo seleccionado para generar un nuevo resolvente cuyo primer sub-objetivo se convierte ahora en el nuevo objetivo a resolver.
- Si **no es posible expandir** el objetivo seleccionado, es decir el objetivo dado es un objetivo fracaso, entonces **se retrocede cronológicamente** a su objetivo padre el cual pasa a ser el objetivo a resolver utilizando una cláusula no anteriormente aplicada al mismo con la que pueda obtenerse un nuevo resolvente.
- El proceso bien termina, como se ha indicado, al obtenerse una **rama triunfo**, o porque G es un **objetivo fracaso**.

Estrategia de Búsqueda

O bien transcurre infinitamente sin encontrar una refutación ... habiéndola, como se demuestra con el siguiente programa. $P \cup G$:

$p(a, b).$

$p(c, b).$

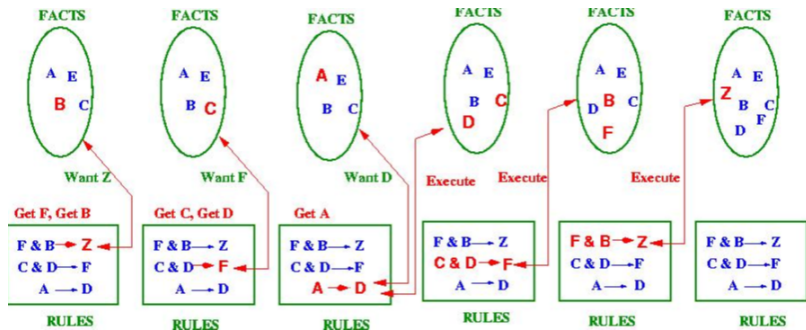
$p(X, Y) :- p(X, Z), p(Z, Y).$

$p(X, Y) :- p(Y, X).$

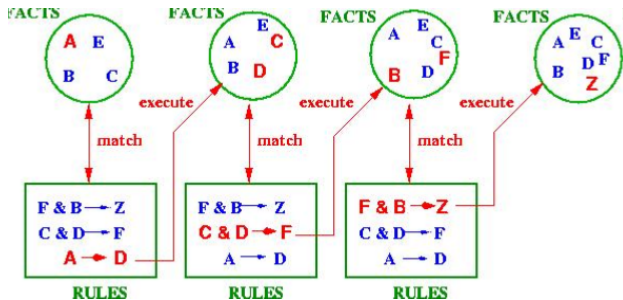
$:- p(a, c).$

El orden de las cláusulas y el orden de los sub-objetivos de las cláusulas requieren atención del programador en Prolog. La utilización de BPP-RC obliga a prestar atención al orden en el cual serán tomadas las cláusulas por el intérprete.

Backward Chaining



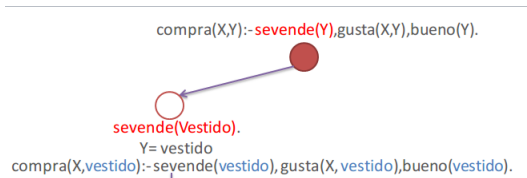
Forward Chaining



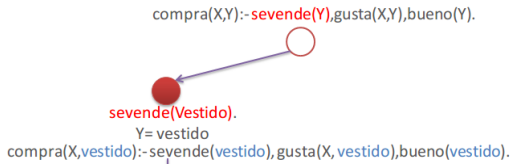
Ejemplo

```
1 % Regla
2 compra(X,Y):-
3     sevende(Y),
4     gusta(X,Y),
5     bueno(Y).
6
7 % Hechos
8 sevende(vestido).
9 sevende(sombrero).
10 sevende(zapatos).
11
12 % Relaciones
13 gusta(jaime,zapatos).
14 gusta(maria,vestido).
15 gusta(maria,sombrero).
16 bueno(sombrero).
```

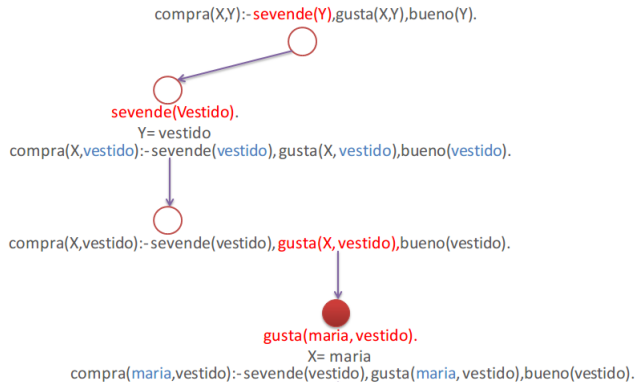
Ejemplo



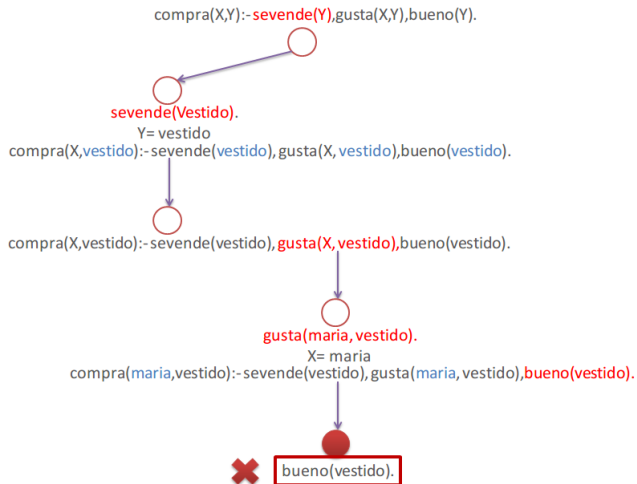
Ejemplo



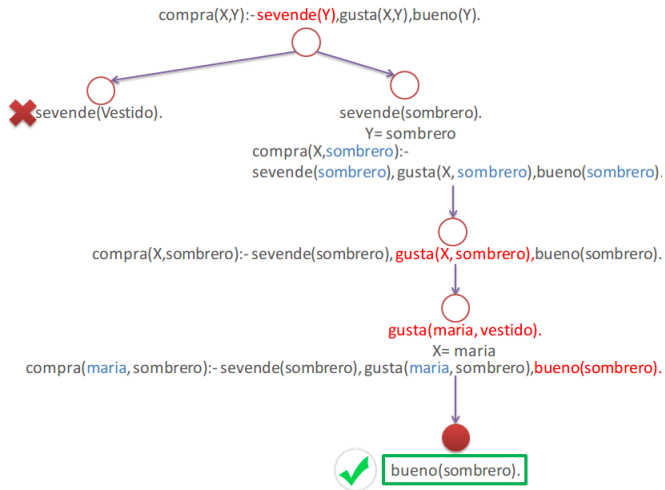
Ejemplo



Ejemplo



Ejemplo



Corte y Negación

Introducción

Recordemos que ... Prolog resuelve el **problema del control** en la ejecución de algoritmos adoptando **BPP-RC** como la **estrategia de navegación en el espacio de búsqueda**.

Pero ... 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 en la deducción del objetivo, lo cual resultaría en una **mayor eficiencia**.

El corte: estructura de control explícita.

- Prolog suministra un predicado primitivo denominado **corte(cut)** y denotado por **!/0**.
- 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** (como objetivo siempre triunfa).
- 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.

Ejemplo sin corte

Dado el siguiente programa para hallar el menor entre dos números:

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

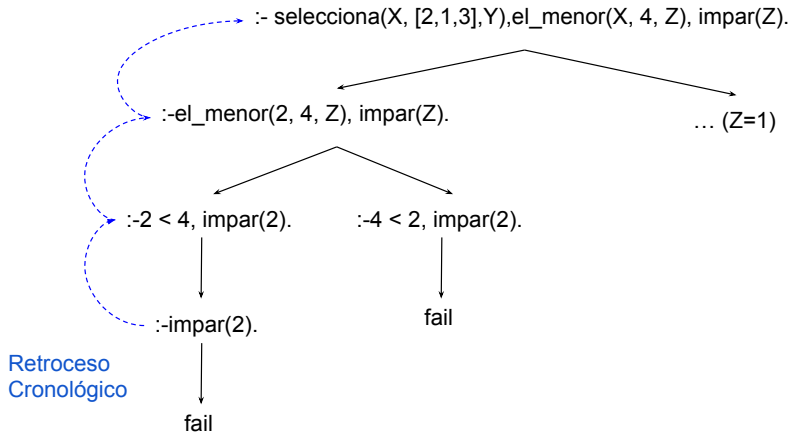
```
el_menor(X, Y, Y) :- Y < X.
```

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).
```


Ejemplo sin corte

El árbol de deducción simplificado es el siguiente:



Ejemplo con corte

Considere ahora el siguiente programa en cuya definición ocurre el corte.

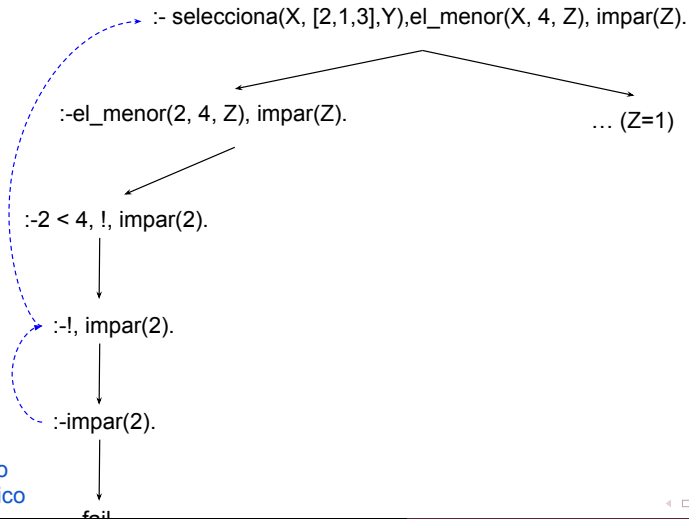
```
el_menor(X, Y, X) :- X < Y, !.  
el_menor(X, Y, Y) :- Y < X.
```

y evalúese nuevamente con este programa el objetivo:

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

Ejemplo sin corte

El árbol de deducción simplificado es el siguiente:



Retroceso
Cronológico

Funcionamiento del corte

Analizando el nuevo árbol de deducción se verifica lo siguiente:

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

- ❶ No reevalúe ninguno de los sub-objetivos a la izquierda del corte (en el ejemplo: $2 < 4$)
- ❷ 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`).
- ❸ 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`)).

Ejemplo

Considere el predicado $\text{fusion}(X, Y, Z)$ donde Z es la lista ordenada de enteros que se obtiene al fundir las listas ordenadas de enteros X y Y :

$\text{fusion}(X, [], X).$

$\text{fusion}([], X, X).$

$\text{fusion}([X | X1], [Y | Y1], [X | Z]) :- X < Y, \text{fusion}(X1, [Y | Y1], Z).$

$\text{fusion}([X | X1], [Y | Y1], [X, Y | Z]) :- X = Y, \text{fusion}(X1, Y1, Z).$

$\text{fusion}([X | X1], [Y | Y1], [Y | Z]) :- X > Y, \text{fusion}([X | X1], Y1, Z).$

Ejemplo

La introducción conveniente del corte en las cláusulas de este predicado permite declarar una aplicación mutuamente excluyente de sus cláusulas:

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

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

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

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

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

Ejemplo

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.

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).
```

Usos incorrectos del corte

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.

$\text{min1}(X,Y,X):- X = < Y, !.$

$\text{min1}(X,Y,Y):- Y < X.$

$\text{min2}(X,Y,X):- X = < Y, !.$

$\text{min2}(X,Y,Y).$

La primera definición (min1) hace un uso correcto del corte.

La segunda definición (min2) tiene toda la apariencia de un “if then else” al omitirse la comparación en la segunda cláusula. Considere ahora el siguiente objetivo :- **$\text{min}(2,5,5)$** .

Conclusiones sobre el Corte

- El corte es una estructura de control explícita para lograr mayor eficiencia en la ejecución de un programa.
- Debe ser utilizado sin afectar la interpretación proyectada del predicado.
- El uso del corte no agrega nada 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.

La negación en Prolog

Si se analiza un programa lógico, se verá que se trata de un conjunto de cláusulas (hechos, reglas) que enuncian solo conocimiento positivo.

Sin embargo, no es posible prescindir de la negación para representar y procesar conocimiento acerca de muchos problemas. Por ejemplo: la definición de listas disjuntas (no tienen elementos en común).

La negación en Prolog

Se introduce en Prolog la operación proposicional de la **negación** como un predicado unario **not/1**.

Los intérpretes modernos usan el símbolo `\+` (mnemotécnica para "no probable") aunque aceptan ambos por compatibilidad con códigos anteriores.

Ejemplo:

```
?- \+(2 = 4).
```

```
true.
```

```
?- not(2 = 4).
```

```
true.
```

La negación en Prolog

¿Cuál sería una definición admisible del predicado not/1?

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)).

Aprovechando los fracasos se puede introducir la HMC como una “regla de inferencia” más en el intérprete de Prolog que se ocupe de interpretar la negación.

Negación como Fracaso Finito (NFF)

Sea P un programa y G un literal, entonces

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

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



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

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 $\{:- \text{not}(G)\}$ es una consecuencia de un programa P por la regla NFF si G pertenece al conjunto de fracasos finitos de P .

Ejemplos Negación

- ❶ Se verifica que $\text{Familia} \cup \{:- \text{abuelo}(\text{a}, \text{e}).\}$ fracasa finitamente, luego aplicando la regla anterior se tiene que $\text{Familia} \cup \{:- \text{not } \text{abuelo}(\text{a}, \text{e}).\}$ triunfa.
- ❷ Analice la siguiente definición del predicado `disjuntas/2` que verifica si dos listas no tienen ningún elemento en común,
`disjuntas(L1, L2):-not((member(X, L1), member(X, L2))).`
- ❸ 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).`

Referencias

-  MAX BRAMER. 2014. **Logic Programming with Prolog (2nd. ed.)**. Springer Publishing Company, Incorporated.
-  CLOCKSIN, W. F., MELLISH, C. S. 2003. *Programming in Prolog*. Berlin: Springer. ISBN: 978-3-540-00678-7