

Paradigmas de Programación

Resolución SLD Prolog

2do cuatrimestre de 2024

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Resolución SLD

Semántica operacional de Prolog

Aspectos extra-lógicos

Resolución general

Recordemos

Para determinar si una fórmula de primer orden σ es válida:

1. Pasar su negación $\neg\sigma$ a forma clausal.

Se obtiene un conjunto \mathcal{C} de cláusulas tal que $\neg\sigma$ es satisfactible sii \mathcal{C} es satisfactible.

2. Aplicar repetidamente la regla de resolución:

$$\frac{\begin{array}{c} \{\sigma_1, \dots, \sigma_p, \ell_1, \dots, \ell_n\} \quad \{\neg\tau_1, \dots, \neg\tau_q, \ell'_1, \dots, \ell'_m\} \quad (p, q > 0) \\ \mathbf{S} = \text{mgu}(\{\sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{?}{=} \dots \stackrel{?}{=} \sigma_p \stackrel{?}{=} \tau_1 \stackrel{?}{=} \tau_2 \stackrel{?}{=} \dots \stackrel{?}{=} \tau_q\}) \end{array}}{\mathbf{S}(\{\ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_m\})}$$

3. Si se alcanza la cláusula vacía, $\neg\sigma$ es insatisfactible y σ válida.
4. El método puede no terminar.

Resolución general

Aplicar el método de resolución general puede ser muy costoso.
Cada paso requiere usar criterios de búsqueda y selección:

Búsqueda. Elegir dos cláusulas.

Selección. Elegir un subconjunto de literales de cada cláusula.

La cantidad de opciones es exponencial en el tamaño del problema.

Además, cada paso agrega una nueva cláusula.

Además, en cada paso se deben resolver ecuaciones de unificación.

Además, el método requiere usar *breadth-first search* (BFS).

Resolución SLD

Veremos una variante de la resolución general, la **resolución SLD**.
Es un *tradeoff*: menor generalidad a cambio de mayor eficiencia.

Menor generalidad

No se puede aplicar sobre fórmulas de primer orden arbitrarias.
Sólo se puede aplicar sobre **cláusulas de Horn**.

Mayor eficiencia

Se reducen las opciones de búsqueda/selección.

Cláusulas de Horn

Recordemos que una cláusula es un conjunto de literales:

$$\{\ell_1, \dots, \ell_n\}$$

donde cada literal es una fórmula atómica posiblemente negada:

$$\ell ::= \underbrace{\mathbf{P}(t_1, \dots, t_n)}_{\text{literal positivo}} \mid \underbrace{\neg \mathbf{P}(t_1, \dots, t_n)}_{\text{literal negativo}}$$

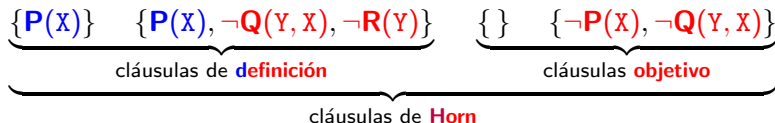
Definición (Cláusulas de Horn)

Las cláusulas son de los siguientes tipos, dependiendo del número de literales positivos/negativos que contienen:

| | #positivos | #negativos |
|-------------------------------|------------|------------|
| cláusula objetivo | 0 | * |
| cláusula de definición | 1 | * |
| cláusula de Horn | ≤ 1 | * |

Cláusulas de Horn

Ejemplo — cláusulas de Horn



Observación

Hay fórmulas que no se pueden escribir como cláusulas de Horn.

Por ejemplo:

$$P \vee Q$$

Regla de resolución SLD

La regla de resolución SLD involucra siempre a una cláusula de **definición** y una cláusula **objetivo**:

$$\frac{\begin{array}{c} \{\mathbf{P}(t_1, \dots, t_k), \neg\sigma_1, \dots, \neg\sigma_n\} \quad \{\neg\mathbf{P}(s_1, \dots, s_k), \neg\tau_1, \dots, \neg\tau_m\} \\ \mathbf{S} = \text{mgu}(\{\mathbf{P}(t_1, \dots, t_k) \stackrel{?}{=} \mathbf{P}(s_1, \dots, s_k)\}) \end{array}}{\mathbf{S}(\{\neg\sigma_1, \dots, \neg\sigma_n, \neg\tau_1, \dots, \neg\tau_m\})}$$

Caso particular de la regla de resolución general.

La selección es binaria (un literal de cada cláusula).

La resolvente es una nueva cláusula **objetivo**.

Derivaciones SLD

Una derivación SLD comienza con $n \geq 0$ cláusulas de **definición** y una cláusula **objetivo**:

$$D_1 \quad \dots \quad D_n \quad G_1$$

En cada paso:

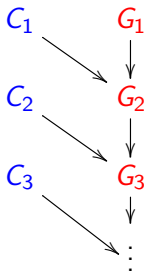
- ▶ Se elige una cláusula definición D_j con $1 \leq j \leq n$.
- ▶ Se aplica la regla de resolución SLD sobre D_j y G_i .
- ▶ La resolvente es una nueva cláusula objetivo G_{i+1} .

Derivaciones SLD

Es decir, dado un conjunto de cláusulas:

$$D_1 \quad \dots \quad D_n \quad G_1$$

una derivación SLD es de la forma:



Cada C_i debe ser alguna de las cláusulas originales $\{D_1, \dots, D_n\}$.
La cláusula G_{i+1} se obtiene aplicando resolución SLD sobre C_i y G_i .

Derivaciones SLD

Observaciones:

- ▶ La **búsqueda** se simplifica.
Se limita a elegir C_i como una de las n cláusulas D_1, \dots, D_n .
La cláusula objetivo G_i está fijada, no hay alternativas.
- ▶ La **selección** se simplifica.
Se limita a elegir uno de los literales **negativos** de G_i .
La cláusula de definición C_i tiene un único literal **positivo**.

Derivaciones SLD

Ejemplo

Dadas las hipótesis:

$$\boxed{1} \quad \forall X. a(0, X, X)$$

$$\boxed{2} \quad \forall X. \forall Y. \forall Z. (a(X, Y, Z) \Rightarrow a(s(X), Y, s(Z)))$$

Queremos demostrar:

$$\boxed{3} \quad \exists X. a(s(0), X, s(s(s(0))))$$

Es decir, queremos probar que $(\boxed{1} \wedge \boxed{2}) \Rightarrow \boxed{3}$ es válida.

Basta ver que $\neg((\boxed{1} \wedge \boxed{2}) \Rightarrow \boxed{3})$ es insatisfactible.

Es decir, basta ver que $\boxed{1} \wedge \boxed{2} \wedge \neg \boxed{3}$ es insatisfactible.

Derivaciones SLD

Escribiendo $\boxed{1} \wedge \boxed{2} \wedge \neg \boxed{3}$ en forma clausal, tenemos:

$$\begin{array}{ll} \boxed{1} & \{a(0, X, X)\} \\ \boxed{2} & \{\neg a(X, Y, Z), a(s(X), Y, s(Z))\} \\ \boxed{3} & \{\neg a(s(0), X, s(s(s(0))))\} \end{array}$$

$\boxed{1}$ y $\boxed{2}$ son cláusulas de **definición**.

$\boxed{3}$ es la cláusula **objetivo**.

Derivaciones SLD

Busquemos una **refutación SLD** (derivación SLD que llega a $\{\}$):

$$\boxed{3} = \{\neg a(s(0), X, s(s(s(0))))\}$$

$$\blacktriangleright \boxed{3} \text{ y } \boxed{2} = \{\neg a(X_4, Y_4, Z_4), a(s(X_4), Y_4, s(Z_4))\}$$

$$\mathbf{S}_4 = \{X_4 := 0, X := Y_4, Z_4 := s(s(0))\}.$$

$$\boxed{4} = \{\neg a(0, Y_4, s(s(0)))\}.$$

$$\blacktriangleright \boxed{4} \text{ y } \boxed{1} = \{a(0, X_5, X_5)\}$$

$$\mathbf{S}_5 = \{Y_4 := s(s(0)), X_5 := s(s(0))\}$$

$$\boxed{5} = \{\}$$

Derivaciones SLD

Definición (Sustitución respuesta)

Dada una refutación SLD, con pasos:

$$\begin{array}{ccccccc} G_1 & \xrightarrow{S_1} & G_2 & \xrightarrow{S_2} & \dots & G_{n-1} & \xrightarrow{S_{n-1}} & G_n \\ C_1 & \nearrow & C_2 & \nearrow & & C_{n-1} & \nearrow & \end{array}$$

la **sustitución respuesta** es la composición $S_{n-1} \circ \dots \circ S_1$.

Ejemplo — sustitución respuesta

En el ejemplo anterior, la sustitución respuesta es $S_5 \circ S_4$.

El valor de X en la cláusula objetivo original 3 es $s(s(0))$.

Esto dice que $X = s(s(0))$ verifica nuestra consulta original:

$$\exists X. a(s(0), X, s(s(s(0))))$$

Completitud del método de resolución SLD

El método de resolución es completo para cláusulas de Horn.

Más precisamente, si D_1, \dots, D_n son cláusulas de **definición** y G una cláusula **objetivo**:

Teorema

Si $\{D_1, \dots, D_n, G\}$ es insatisfactible, existe una refutación SLD.

Resolución SLD

Semántica operacional de Prolog

Aspectos extra-lógicos

Semántica de Prolog

Un programa en Prolog es una *lista* de cláusulas de definición.

Una consulta en Prolog es una cláusula objetivo.

La notación cambia ligeramente.

Ejemplo

| Cláusulas | Prolog |
|---|---|
| $\{a(0, X, X)\}$ | <code>a(0,X,X) .</code> |
| $\{a(s(X), Y, s(Z)), \neg a(X, Y, Z)\}$ | <code>a(s(X),Y,s(Z)) :- a(X,Y,Z) .</code> |
| $\{\neg a(s(0), X, s(s(s(0))))\}$ | <code>?- a(s(0),X,s(s(s(0)))) .</code> |

Las cláusulas son *listas*: el orden y la multiplicidad son relevantes.

Semántica de Prolog

La ejecución se basa en la regla de resolución SLD.
Escrita con la notación de Prolog:

$$\frac{\begin{array}{l} ?- \textcolor{red}{p}(t_1, \dots, t_k), \sigma_1, \dots, \sigma_n. \quad \textcolor{blue}{p}(s_1, \dots, s_k) :- \textcolor{red}{\tau}_1, \dots, \textcolor{red}{\tau}_m. \\ \mathbf{S} = \text{mgu}(\textcolor{red}{p}(t_1, \dots, t_k) \stackrel{?}{=} \textcolor{blue}{p}(s_1, \dots, s_k)) \end{array}}{\mathbf{S} (?- \textcolor{red}{\tau}_1, \dots, \textcolor{red}{\tau}_m, \sigma_1, \dots, \sigma_n.)}$$

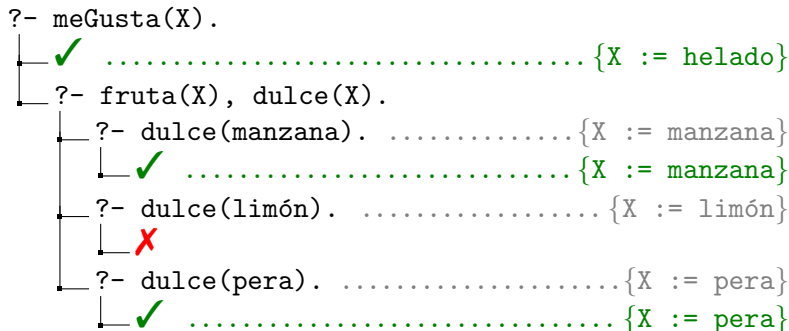
El orden de los literales en la cláusula objetivo es relevante.
Criterio de **selección**: elegir siempre el **primer literal** de la cláusula.

Semántica de Prolog

Prolog busca sucesivamente todas las refutaciones haciendo DFS.
Criterio de **búsqueda**: las reglas se usan en orden de aparición.

Ejemplo — árbol de refutación

```
fruta(manzana).      dulce(manzana).  
fruta(limón).        dulce(pera).  
fruta(pera).  
meGusta(helado).    meGusta(X) :- fruta(X), dulce(X).
```



Semántica de Prolog

La exploración *depth-first* (DFS) es **incompleta**.
Puede provocar que Prolog nunca encuentre refutaciones posibles.

Ejemplo — incompletitud de DFS

```
esMaravilloso(X) :- esMaravilloso(suc(X)).  
esMaravilloso(cero).
```

```
?- esMaravilloso(cero).  
  |  
  |?- esMaravilloso(suc(cero)).  
    |  
    |?- esMaravilloso(suc(suc(cero))).  
      |  
      |?- esMaravilloso(suc(suc(suc(cero)))).  
        |  
        |...  
        |
```

El orden de las reglas se torna relevante.

Tradeoff: mayor eficiencia a cambio de menor declaratividad.
La exploración *breadth-first* (BFS) es completa pero muy costosa.

Semántica de Prolog

Al unificar, Prolog **no** usa la regla *occurs-check*.

Por ejemplo, X unifica con $f(X)$. Esto es **incorrecto**.

Puede provocar que Prolog encuentre una “refutación” incorrecta.

Ejemplo — refutación incorrecta por omisión de *occurs check*

`esElSucesor(X, suc(X)).`

`?- esElSucesor(Y, Y).`

`└─ ✓ {Y := X, X := suc(X)}`

Tradeoff: mayor eficiencia a cambio de incorrección lógica.

En muchos contextos la regla *occurs-check* es innecesaria.

La carga de probar corrección recae en los programadores.

Ejemplo: concatenación de listas

`c([], Ys, Ys).`

`c([X | Xs], Ys, [X | Zs]) :- c(Xs, Ys, Zs).`

`?- c([1, 2], [3, 4], Zs).`

└─ `?- c([2], [3, 4], Zs1).` `{Zs := [1 | Zs1]}`

└─ `?- c([], [3, 4], Zs2).` .. `{Zs := [1, 2 | Zs2]}`

└─ ✓ `{Zs := [1, 2, 3, 4]}`

`?- c([1, 2], Ys, [1, 2, 3, 4]).`

└─ `?- c([2], Ys, [2, 3, 4]).`

└─ `?- c([], Ys, [3, 4]).`

└─ ✓ `{Ys := [3, 4]}`

Ejemplo: concatenación de listas

`c([], Ys, Ys).`

`c([X | Xs], Ys, [X | Zs]) :- c(Xs, Ys, Zs).`

`?- c(Xs, [3], [1, 2, 3]).`

└─ `?- c(Xs1, [3], [2, 3]).` `{Xs := [1 | Xs1]}`

└─ `?- c(Xs2, [3], [3]).` `{Xs := [1, 2 | Xs2]}`

└─  `{Xs := [1, 2]}`

└─ `?- c(Xs3, [3], []).`

└─ `{Xs := [1, 2, 3 | Xs3]}`

└─ 

Ejemplo: concatenación de listas

`c([], Ys, Ys).`

`c([X | Xs], Ys, [X | Zs]) :- c(Xs, Ys, Zs).`

`?- c(Xs, [9], Zs).`

✓ {Xs := [], Zs := [9]}

└─ `?- c(Xs1, [9], Zs1).`

└─ ✓ {Xs := [_1 | Xs1], Zs := [_1 | Xs1]}

└─ `?- c(Xs2, [9], Zs2).`

└─ ✓ {Xs := [_1, _2 | Xs2], Zs := [_1, _2 | Xs1]}

└─ ...

Resolución SLD

Semántica operacional de Prolog

Aspectos extra-lógicos

Operador de corte (cut)

Consideremos el siguiente programa:

```
padre(zeus, atenea). % ...base de conocimiento...  
ancestro(X, Y) :- padre(X, Y).  
ancestro(X, Y) :- padre(X, Z), ancestro(Z, Y).
```

```
?- ancestro(zeus, atenea).  
  |  
  |?- padre(zeus, atenea).  
  |  |  
  |  | ✓  
  |  |  
  |  |?- padre(zeus, Z), ancestro(Z, atenea).  
  |  |  
  |  | ...  
  |  |
```

Nos gustaría tener una manera de podar el árbol de búsqueda.

Operador de corte (cut)

Reescribimos el programa agregando un *corte* ("!"):

```
padre(zeus, atenea). % ...base de conocimiento...  
ancestro(X, Y) :- padre(X, Y), !.  
ancestro(X, Y) :- padre(X, Z), ancestro(Z, Y).
```

El operador **!** no tiene una interpretación declarativa/lógica.

Es un operador extra-lógico.

Su comportamiento se explica desde el punto de vista operacional.

El operador de corte indica que, si se lo alcanza, no se deben explorar alternativas a la regla en la que aparece.

Operador de corte (cut)

Semántica del operador de corte

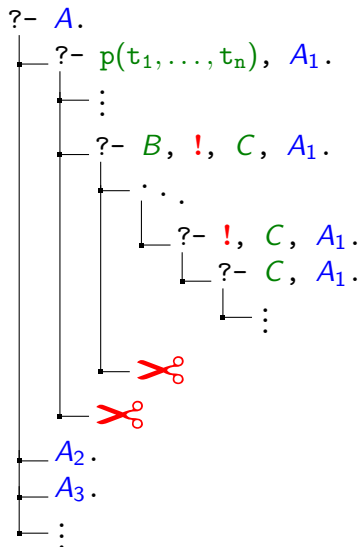
El predicado **!** tiene éxito inmediatamente.

Al momento de hacer *backtracking*:

- ▶ Se vuelve atrás hasta el punto en el que se eligió usar la regla que hizo aparecer el operador de corte.
- ▶ Se descartan todas las elecciones alternativas.
- ▶ Se continúa buscando hacia atrás.

Operador de corte (cut)

Gráficamente:



Operador de corte (cut)

Ejemplo — cortes “benignos” (green cuts)

En algunos casos, **!** no altera la semántica del programa.
Puede servir para construir programas equivalentes más eficientes.

```
add(N, zero, N) :- !.  
add(zero, N, N).  
add(suc(N), M, suc(P)) :- add(N, M, P).
```

```
?- add(suc(suc(suc(suc(...))), zero, P).
```

Ejemplo — cortes “riesgosos” (red cuts)

En otros casos, la semántica puede verse alterada.

```
maximo(A, B, A) :- A >= B, !.  
maximo(A, B, B).
```

```
?- maximo(2, 1, C).  
>> C = 2
```

```
?- maximo(2, 1, 1).  
>> true.
```

Negación por falla

Definición (operador de negación)



Si `fail` es un predicado que falla siempre, se puede definir la negación en Prolog así:



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

Observación. `not(P)` tiene éxito si y sólo si `P` falla.

Ejemplo — negación por falla

`fruta(pera).`

```
?- not(fruta(papa)).  
  |  
  | ?- fruta(papa), !, fail.  
  | |  
  | |   
  | |  
  | |   
  |  
  |  
  |
```

```
?- not(fruta(pera)).  
  |  
  | ?- fruta(pera), !, fail.  
  | |  
  | | ?- !, fail.  
  | | |  
  | | | ?- fail.  
  | | | |  
  | | | |   
  | | |  
  | |   
  |  
  |  
  |
```


Negación por falla

La negación por falla no coincide con la negación lógica.

Ejemplo — negación por falla \neq negación lógica

`fruta(pera). verdura(papa).`

```
?- verdura(X), not(fruta(X)).  
  |  
  |_- ?- not(fruta(papa)).....{X := papa}  
    |  
    |_- ...✓
```

```
?- not(fruta(X)), verdura(X).  
  |  
  |_- ?- fruta(X), !, fail, verdura(X).  
    |  
    |_- ?- !, fail, verdura(pera).....{X := pera}  
      |  
      |_- ?- fail, verdura(pera).....{X := pera}  
        |  
        |_- ✗  
    |  
    |_- ✂
```

El orden de los literales en la consulta se vuelve relevante.

Esto atenta contra la declaratividad.

Ejemplo

1. Definir `append(L1, L2, L3)`.
2. Definir `ocurreAlMenosUnaVez(X, L)` usando `append/3`.
3. Definir `ocurreAlMenosDosVeces(X, L)` usando `append/3`.
4. Definir `ocurreExactamenteUnaVez(X, L)` combinando las anteriores.

i i i i i i i i i i ? ? ? ? ? ? ? ?