Resolución SLD y Prolog

Paradigmas de Lenguajes de Programación

Resolución lineal

- Si bien el método de resolución general es completo, hallar refutaciones es un proceso muy caro en el caso general
- El espacio de búsqueda producido puede ser enorme
- Hay un alto grado de no-determinismo
  - ¿Qué cláusulas elegimos? Regla de búsqueda
  - ¿Qué literales eliminamos? Regla de selección
- Se precisan restricciones (regla de búsqueda y selección) para reducir el espacio de búsqueda (utilidad práctica)
- Es deseable que dichas restricciones no renuncien a la completitud del método

Resolución lineal

Una secuencia de pasos de resolución a partir de S es lineal si es de la forma:

$$
\begin{array}{lcc}
B_1 & C_0 & \\
 & | & \\
B_2 & C_1 & \sigma_1 \\
 & | & \\
B_3 & C_2 & \sigma_2 \\
 & | & \\
 & C_3 & \sigma_3 \\
 & & \\
B_{p-1} & \dots & \\
 & | & \\
B_p & C_{p-1} & \sigma_{p-1} \\
 & | & \\
 & C_p = \square & \sigma_p
\end{array}
$$

donde C_0 y cada B_i es un elemento de S (o algún C_j con $j < i$)

Resolución lineal

- En general, reduce el espacio de búsqueda considerablemente
- Preserva completitud
- Sin embargo sigue siendo altamente no-determinístico
  - El criterio de búsqueda deja espacio para refinamientos
  - No se especificó ningún criterio de selección

Cláusulas de Horn

Cláusulas de Horn

Una cláusula de Horn es de la forma $\forall x_1 \dots \forall x_m C$ tal que la disyunción de literales C tiene a lo sumo un literal positivo

Nota: C puede tomar una de las formas:

- $\{B, \neg A_1, \dots, \neg A_n\}$
- $\{B\}$
- $\{\neg A_1, \dots, \neg A_n\}$ (cláusula "goal", objetivo o negativa)

Relación con fórmulas de primer orden

- No toda fórmula de primer orden puede expresarse como una cláusula de Horn
- Ejemplos
  - $\forall x.(P(x) \vee Q(x))$
  - $(\forall x.P(x) \vee \forall x.Q(x)) \supset \forall x.(P(x) \vee Q(x))$
- Sin embargo, el conjunto de cláusulas de Horn es suficientemente expresivo para representar programas, en la visión de resolución como computación.
  - Más detalles por venir.

Resolución SLD

- Cláusula de definición ("Definite Clause")
  - Cláusula de la forma $\forall x_1 \ldots \forall x_m C$ tal que la disyunción de literales C tiene exactamente un literal positivo
- Sea $S = P \cup \{G\}$ un conjunto de cláusulas de Horn (con nombre de variables disjuntos) tal que
  - P conjunto de cláusulas de definición y
  - G un cláusula negativa
- $S = P \cup \{G\}$ son las cláusulas de entrada
  - P se conoce como el programa o base de conocimientos y
  - G el goal, meta o cláusula objetivo

Resolución SLD

Un secuencia de pasos de resolución SLD para S es una secuencia

$$< N_0, N_1, \ldots, N_p >$$

de cláusulas negativas que satisfacen las siguientes dos condiciones.

1. N_0 es el goal G
2. *sigue en transparencia siguiente*

Resolución SLD

2. para todo N_i en la secuencia, $0 < i < p$, si N_i es

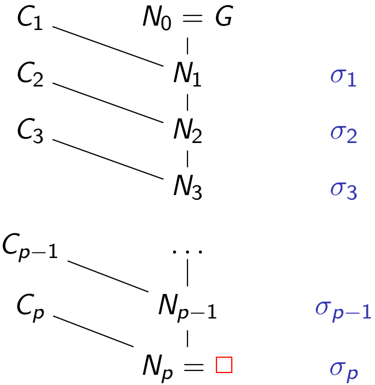
$$\{\neg A_1, \ldots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \ldots, \neg A_n\}$$

entonces hay alguna cláusula de definición C_i de la forma $\{A, \neg B_1, \ldots, \neg B_m\}$ en P tal que A_k y A son unificables con MGU σ, y si

- $m = 0$, entonces N_{i+1} es $\{\sigma(\neg A_1, \ldots, \neg A_{k-1}, \neg A_{k+1}, \ldots, \neg A_n)\}$
- $m > 0$, entonces N_{i+1} es $\{\sigma(\neg A_1, \ldots, \neg A_{k-1}, \neg B_1, \ldots, \neg B_m, \neg A_{k+1}, \ldots, \neg A_n)\}$

Refutación SLD

Una refutación SLD es una secuencia de pasos de resolución SLD $< N_0, \dots, N_p >$ tal que $N_p = \square$



Sustitución respuesta

- En cada paso, las cláusulas $\{\neg A_1, \ldots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \ldots, \neg A_n\}$ y $\{A, \neg B_1, \ldots, \neg B_m\}$ son resueltas
- Los átomos A_k y A son unificados con MGU σ_i
- El literal A_k se llama átomo seleccionado de N_i
- Sustitución respuesta es la sustitución

$$\sigma_p \circ \ldots \circ \sigma_1$$

  - se usa en Prolog para extraer la salida del programa

Ejemplo

Consideremos las siguientes cláusulas de definición

- $C_1 = \{add(U, 0, U)\}$
- $C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$

y la cláusula goal G

$$\{\neg add(succ(0), V, succ(succ(0)))\}$$

- Deseamos mostrar que el conjunto de estas cláusulas (i.e. $\{C_1, C_2, G\}$) es insatisfactible
- Contamos con la siguiente refutación SLD

Ejemplo

$$C_1 = \{add(U, 0, U)\}$$
$$C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$$

Cláusula goal	**Cláusula de entrada**	**Sust.**
$\{\neg add(succ(0), V, succ(succ(0)))\}$	C_2	
$\{\neg add(succ(0), Y, succ(0))\}$	C_1	σ_1
$\square$		σ_2

donde

$$\begin{array}{rcl} \sigma_1 & = & \{X \leftarrow succ(0), V \leftarrow succ(Y), Z \leftarrow succ(0), \} \\ \sigma_2 & = & \{U \leftarrow succ(0), Y \leftarrow 0\} \end{array}$$

- La sustitución resultado es $\sigma_2 \circ \sigma_1 =$

$$\{X \leftarrow succ(0), V \leftarrow succ(0), Z \leftarrow succ(0), U \leftarrow succ(0), Y \leftarrow 0\}$$

Corrección y completitud

Corrección

Si un conjunto de cláusulas de Horn tiene una refutación SLD, entonces es insatisfactible

Completitud

Dado un conjunto de cláusulas de Horn $P \cup \{G\}$ tal como se describió, si $P \cup \{G\}$ es insatisfactible, existe una refutación SLD cuya primera cláusula es G.

Resolución SLD en Prolog

- Prolog utiliza resolución SLD con las siguientes restricciones
  - Regla de búsqueda: se seleccionan las cláusulas de programa de arriba hacia abajo, en el orden en que fueron introducidas
  - Regla de selección: seleccionar el átomo de más a la izquierda
- La suma de regla de búsqueda y regla de selección se llama estrategia
- Cada estrategia determina un árbol de búsqueda o árbol SLD

Ejemplo

Cláusulas de Def.

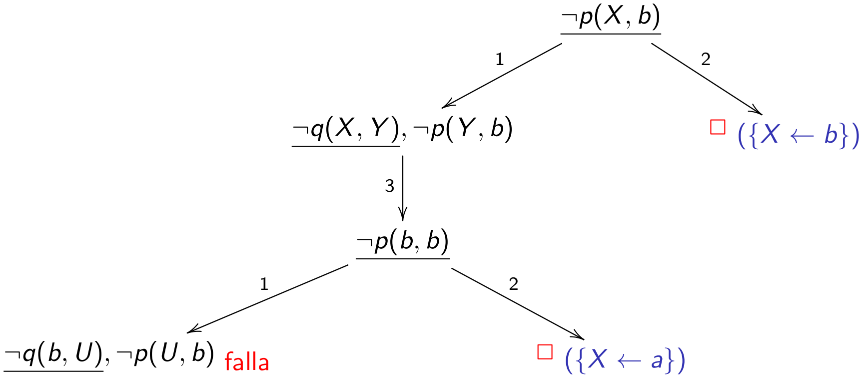
1. $\{p(X,Z), \neg q(X,Y), \neg p(Y,Z)\}$
2. $\{p(X,X)\}$
3. $\{q(a,b)\}$

Goal

$\{\neg p(X,b)\}$

El árbol de búsqueda, seleccionando el átomo que está más a la izquierda, es el siguiente:

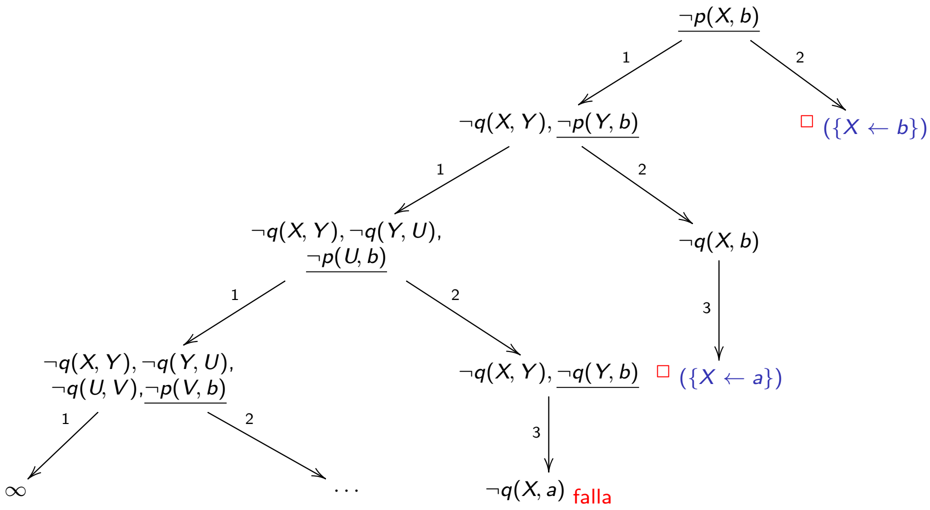
Ejemplo - árbol SLD



Variando la regla de selección

- Si variamos la regla de selección, varía el árbol SLD asociado
- Supongamos ahora que la regla de selección es "seleccionar el de más a la derecha"

Ejemplo - átomo de más la derecha



Variando la regla de búsqueda

- Si variamos la regla de búsqueda, también varía el árbol SLD asociado
- Ejercicio: suponer que ahora la regla de búsqueda es "seleccionar las cláusulas de programa de abajo hacia arriba" y armar el árbol para las cláusulas del ejemplo

Motivación

- En esta parte de la clase exploramos de qué manera resolución SLD puede usarse para computar
- Asimismo, vamos a enfatizar el rol de la sustitución respuesta como "resultado de cálculo"

Motivación

- Recordar el ejemplo de la suma
  - $C_1 = \{add(U, 0, U)\}$
  - $C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$
- Estas cláusulas pueden verse como una definición recursiva de la suma
- Supongamos que queremos saber si, dada esa definición,

$$\exists V.add(succ(0), V, succ(succ(0)))$$

- En otras palabras

"¿Existe V tal que $1 + V = 2$?"

Motivación

Podemos plantearlo como la validez de la fórmula

$$C_1 \wedge C_2 \supset \exists V.add(succ(0), V, succ(succ(0)))$$

Es decir,

$$\underbrace{(C_1 \wedge C_2)}_{\text{Define la suma}} \quad \supset$$

$$\underbrace{\exists V.add(succ(0), V, succ(succ(0)))}_{\text{Pide calcular } V} \text{ es válida}$$

Motivación

Esto es lo mismo que preguntarse por la insatisfactilidad de

$$C_1 \wedge C_2 \wedge \neg \exists V.add(succ(0), V, succ(succ(0)))$$

O lo que es lo mismo

$$C_1 \wedge C_2 \wedge \forall V.\neg add(succ(0), V, succ(succ(0)))$$

Motivación

Resolución SLD se dispara a partir del conjunto de cláusulas

$$\Big\{\{add(U,0,U), \{add(X,succ(Y),succ(Z)), \neg add(X,Y,Z)\}$$
$$\{\neg add(succ(0),V,succ(succ(0)))\}\Big\}$$

- En caso de tener éxito, va a hallar el V buscado
- Importante observar que
  - No solamente interesa saber que existe tal V (i.e. que existe la refutación)
  - Además, queremos una instancia del mismo
- La sustitución respuesta σ proveerá dichas instancias, las cuales pueden interpretarse como el resultado del cómputo

Programas lógicos - Notación Prolog

Recordar que la resolución SLD parte de un conjunto de cláusulas $S = P \cup \{G\}$ donde

1. P es un conjunto de cláusulas de definición
   - Cláusulas con exactamente un literal positivo

$$\{B, \neg A_1, \ldots, \neg A_n\}$$
$$\{B\}$$

2. G es un goal
   - Cláusula negativa

$$\{\neg A_1, \ldots, \neg A_n\}$$

Notación Prolog para programas lógicos

$$
\begin{aligned}
B \vee \neg A_1 \vee \ldots \vee \neg A_n \quad & \Longleftrightarrow \quad \neg(A_1 \wedge \ldots \wedge A_n) \vee B \\
& \Longleftrightarrow \quad (A_1 \wedge \ldots \wedge A_n) \supset B
\end{aligned}
$$

Como consecuencia, las cláusulas en P se escriben

- B :- $A_1, \ldots, A_n$. para $\{B, \neg A_1, \ldots, \neg A_n\}$ (reglas)
- B. para B (hechos)

Ejemplo de la suma en notación Prolog

Volviendo al ejemplo de la suma, el programa Prolog es

```
add(U,0,U).
add(X,succ(Y),succ(Z)):-add(X,Y,Z).
```

Ingresamos el goal

```
?- add(succ(0),V,succ(succ(0))).
```

La respuesta es:

```
V=succ(0)
```

Corrección y completitud

Corrección

Si existe una refutación SLD de $P \cup \{G\}$ con la estrategia antes mencionada entonces es insatisfactible

Completitud

Si $P \cup \{G\}$ es insatisfactible, entonces una refutación SLD con la estrategia antes mencionada a partir del mismo

Búsqueda de refutaciones SLD en Prolog

- Recorre el árbol SLD en profundidad ("depth-first search")
- La ventaja del recorrido en profundidad es que puede ser implementado de manera muy eficiente
  - Se usa una pila para representar los átomos del goal
  - Se hace un push del resolvente del átomo del tope de la pila con la cláusula de definición
  - Se hace un pop cuando el átomo del tope de la pila no unifica con ninguna cláusula de definición más (luego, el átomo que queda en el tope se unifica con la siguiente cláusula de definición)
- Desventaja: ¡puede que no encuentre una refutación SLD aun si existe!

Más sobre Prolog

Veremos dos temas de Prolog que trascienden la lógica subyacente:

1. Cut
2. Deducción de información negativa: negation as failure (negación por falla)

Cut

- Es un precadicado 0-ario, notado !
- Solo tiene éxito la primera vez que se lo invoca.
- Brinda un mecanismo de control que permite podar el árbol SLD
- Es de carácter extra-lógico (i.e. no se corresponde con un predicado estándar de la lógica)
- Se encuentra presente por cuestiones de eficiencia
- Debe usarse con cuidado, dado que puede podarse una rama de éxito deseada

Ejemplo

```
1.p(a).
2.p(b).
3.p(c).
4.q(a,e).
5.q(a,f).
6.q(b,f).
```

Ejemplo

```
1.p(a).
2.p(b).
3.p(c).
4.q(a,e).
5.q(a,f).
6.q(b,f).
```

```
?- p(X).
X=a ; X=b ; X=c;
no

?- p(X),!.
X=a ;
no

?- p(X),q(X,Y).
X=a, Y=e ;
X=a, Y=f ;
X=b,Y=f ;
no

?- p(X),!,q(X,Y).
X=a Y=e ;
X=a Y=f ;
no
```

Ejemplo

```
1.p(a).
2.p(b).
3.p(c).
4.q(a,e).
5.q(a,f).
6.q(b,f).
7.r(X,Y):-p(X),!,q(X,Y).
```

Ejemplo

```
1.p(a).
2.p(b).
3.p(c).
4.q(a,e).
5.q(a,f).
6.q(b,f).
7.r(X,Y):-p(X),!,q(X,Y).
```

```
?- r(X,Y).
X=a Y=e ;
X=a Y=f ;
no

?- p(X),r(X,Y).
X=a, Y=e ;
X=a, Y=f ;
X=b,Y=f ;
no
```

Ejemplo

- Definición de `max`

```
max1(X,Y,Y) :- X =< Y.
max1(X,Y,X) :- X>Y.
```

- Es ineficiente. ¿Por qué?

Ejemplo

- Definición de `max`

```
max1(X,Y,Y) :- X =< Y.
max1(X,Y,X) :- X>Y.
```

- Es ineficiente. ¿Por qué? Pensar en `max(3,4,Z)`...

Ejemplo

- Definición de `max`

```
max1(X,Y,Y) :- X =< Y.
max1(X,Y,X) :- X>Y.
```

- Es ineficiente. ¿Por qué? Pensar en `max(3,4,Z)`...
- Mejor así

```
max2(X,Y,Y) :- X =< Y, !.
max2(X,Y,X) :- X>Y.
```

- ¿Y esto?

```
max3(X,Y,Y) :- X =< Y, !.
max3(X,Y,X).
```

Ejemplo

- Definición de `max`

```
max1(X,Y,Y) :- X =< Y.
max1(X,Y,X) :- X>Y.
```

- Es ineficiente. ¿Por qué? Pensar en `max(3,4,Z)`...
- Mejor así

```
max2(X,Y,Y) :- X =< Y, !.
max2(X,Y,X) :- X>Y.
```

- ¿Y esto?

```
max3(X,Y,Y) :- X =< Y, !.
max3(X,Y,X).
```

  - Cambia la semántica de max
  - Probar `max(2,3,2)`

Cut - En general

- Cuando se selecciona un cut, tiene éxito inmediatamente
- Si, debido a backtracking, se vuelve a este cut, su efecto es el de hacer fallar el goal que le dio origen
  - El goal que unificó con la cabeza de la cláusula que contiene al corte y que hizo que esa cláusula se "activara"
- El efecto obtenido es el de descartar soluciones (i.e. no dar más soluciones) de
  1. otras cláusulas del goal padre
  2. cualquier goal que ocurre a la izquierda del corte en la cláusula que contiene el corte
  3. todos los objetivos intermedios que se ejecutaron durante la ejecución de los goals precedentes

Negación por falla

- Se dice que un árbol SLD falla finitamente si es finito y no tiene ramas de éxito
- Dado un programa P el conjunto de falla finita de P es $\{B \mid$ B átomo cerrado ('ground') y existe un árbol SLD que falla finitamente con B como raíz$\}$

Negation as failure

$$\frac{B \text{ átomo cerrado} \quad B \text{ en conjunto de falla finita de } P}{\neg B}$$

Predicado not

Negación por falla en Prolog

```
not(G) :- call(G), !, fail.
not(G).
```

Predicado not

Negación por falla en Prolog

```
not(G) :- call(G), !, fail.
not(G).
```

Ejemplo

Puede deducirse `not(student(mary))` a partir de

```
student(joe).
student(bill).
student(jim).
teacher(mary).
```

Predicado not

Negación por falla en Prolog

```
not(G) :- call(G), !, fail.
not(G).
```

Ejemplo

Puede deducirse `not(student(mary))` a partir de

```
student(joe).
student(bill).
student(jim).
teacher(mary).
```

Y también puede deducirse `not(student(anna))` a partir de esos hechos!

Negación por falla no es negación lógica

```
animal(perro).
animal(gato).
vegetal(X) :- not(animal(X)).
```

- La consulta `vegetal(perro)` da "no", tal como se espera
- La consulta `vegetal(pasto)` da "sí", tal como se espera
- ¿Resultado de la consulta `vegetal(X)`?

Negación por falla no es negación lógica

```
animal(perro).                    not(G) :- G, !, fail.
animal(gato).                     not(G).
vegetal(X) :- not(animal(X)).
```

- Observar: el goal `not(G)` nunca instancia variables de G
  - Si G tiene éxito, `fail` falla y descarta la sustitución
  - Caso contrario, `not(G)` tiene éxito inmediatamente (sin afectar G)
- En consecuencia, `not(not(animal(X)))` no es equivalente a `animal(X)`

Negación por falla no es negación lógica

```
firefighter_candidate(X) :-
              not( pyromaniac(X) ),
              punctual(X).
pyromaniac(attila).
punctual(jeanne_d_arc).
```

- ¿Resultado del query `firefighter_candidate(W)`?

Negación por falla no es negación lógica

```
firefighter_candidate(X) :-
              not( pyromaniac(X) ),
              punctual(X).
pyromaniac(attila).
punctual(jeanne_d_arc).
```

- ¿Resultado del query `firefighter_candidate(W)`?
- ¿Por qué `jeanne_d_arc` no es solución?
- Después de todo: ¡si se intercambian las dos cláusulas en la definición de `firefighter_candidate` sí da a `jeanne_d_arc` como solución!