Lógica Computacional 2018-2, nota de clase 11
Resolución binaria, fundamentos de programación lógica y PROLOG

Favio Ezequiel Miranda Perea Araceli Liliana Reyes Cabello
Lourdes Del Carmen González Huesca Pilar Selene Linares Arévalo

26 de abril de 2018

1. Resolución binaria con unificación

La regla de resolución binaria es de primordial importancia para los sistemas de programación lógica y razonamiento automatizado, al proporcionar un método mecánico para decidir la consecuencia lógica $\Gamma \vdash \varphi$, mediante la obtención de la cláusula vacía $\square$ a partir de las formas clausulares de las fórmulas del conjunto $\Gamma \cup \{\neg\varphi\}$.

La regla de resolución para la lógica de predicados toma como premisas dos cláusulas $\mathcal{C}, \mathcal{D}$ con variables ajenas tales que existe una literal ℓ en una de ellas y una literal ℓ' en la otra de manera que ℓ^c y ℓ' son **unificables** mediante un σ. La regla devuelve como conclusión la disyunción de las literales de $\mathcal{C}, \mathcal{D}$ después de eliminar las literales contrarias $\ell\sigma$ y $\ell'\sigma$, pero aplicando a cada una de ellas el unificador σ.

Esquemáticamente tenemos lo siguiente:

$$\frac{\mathcal{C} =_{def} \mathcal{C}_1 \vee \ell \qquad \mathcal{D} =_{def} \mathcal{D}_1 \vee \ell' \qquad Var(\mathcal{C}) \cap Var(\mathcal{D}) = \varnothing \qquad \sigma \text{ umg de } \{\ell^c, \ell'\}}{(\mathcal{C}_1 \vee \mathcal{D}_1)\sigma}$$

En tal caso decimos que $(\mathcal{C}_1 \vee \mathcal{D}_1)\sigma$ es un **resolvente** de $\mathcal{C}$ y $\mathcal{D}$.
Obsérvese que la restricción acerca de las variables siempre puede obtenerse al renombrar las variables de alguna de las cláusulas, lo cual es válido pues las variables de una cláusula en realidad estaban cuantificadas universalmente en una forma normal de Skolem.
Veamos un ejemplo

$$\frac{\mathcal{C} = Pxy \vee \neg Qax \qquad \mathcal{D} = Rz \vee Qzb}{Pby \vee Ra}$$

en tal caso $\ell = \neg Qax$, $\ell' = Qzb$ y $\sigma = [x := b, z := a]$.

Veamos ahora un ejemplo de decisión de consecuencia lógica utilizando resolución binaria:

Ejemplo 1.1 Verificar que $\forall x \exists y(Cx \to Py \wedge Axy) \models \forall z(Cz \to \exists w(Pw \wedge Azw))$.
La transformación a formas clausulares de la premisa y la negación de la conclusión resulta en el siguiente conjunto:

$$\{\neg Cx \vee Pfx,\ \neg Cx \vee Axfx,\ Ca,\ \neg Pw \vee \neg Aaw\}$$

La derivación mediante resolución es:

1.	$\neg Cx \vee Pfx$	$Hip.$
2.	$\neg Cx \vee Axfx$	$Hip.$
3.	Ca	$Hip.$
4.	$\neg Pw \vee \neg Aaw$	$Hip.$
5.	Pfa	$Res(1, 3, [x := a])$
6.	$\neg Aafa$	$Res(5, 4, [w := fa])$
7.	$\neg Ca$	$Res(6, 2, [x := a])$
8.	$\square$	$Res(3, 7, \varnothing)$

Dado que la cláusula vacía fue obtenida podemos concluir que la consecuencia lógica original es válida.

Es importante mencionar que la restricción acerca de que las variables de las dos cláusulas sobre las que se va a aplicar la resolución sean ajenas es primordial para demostrar la completud refutacional, sin tal restricción tendríamos por ejemplo que el conjunto insatisfacible $\{\forall xPx, \forall x\neg Pfx\}$ cuyas formas clausulares son $\{Px, \neg Pfx\}$, no permite derivar la cláusula vacía, pues el conjunto $\{Px, Pfx\}$ no es unificable.
El renombre de variables nos lleva al conjunto $\{Py, Pfx\}$ unificable mediante $\sigma = [y := fx]$.

1.1. Correctud y completud refutacional

Justificamos el método de semidecisión mediante resolución binaria con los teoremas correspondientes de correctud y completud.

Dado un conjunto de cláusulas Γ y una cláusula $\mathcal{C}$ decimos que $\mathcal{C}$ es derivable a partir de Γ mediante resolución, y escribimos en tal caso $\Gamma \vdash_{\mathcal{R}} \mathcal{C}$, si $\mathcal{C}$ se obtuvo a partir de Γ usando resolución binaria y equivalencias simplificativas (eliminación de literales repetidas bajo unificación, proceso llamado factorización).
En el ejemplo anterior tenemos en particular $\Gamma \vdash_{\mathcal{R}} \neg Ca$, donde Γ consta de las cláusulas 1 a 4.

Teorema 1 (Correctud de la Resolución) *Si* $\Gamma \vdash_{\mathcal{R}} \mathcal{C}$ *entonces* $\Gamma \models \mathcal{C}$.

Demostración. Basta observar que la regla de resolución preserva validez. ⊣

Aquí la consecuencia lógica realmente significa $\forall\Gamma \models \forall\mathcal{C}$, es decir se restauran las formas normales de Skolem usando las cerraduras universales de las fórmulas.
Como corolario obtenemos la correctud refutacional:

Corolario 1 (Correctud Refutacional de la Resolución) *Si* $\Gamma \vdash_{\mathcal{R}} \square$ *entonces* Γ *es insatisfacible.*

Este corolario es el que permite que funcione el método del ejemplo anterior. El recíproco de este teorema, llamado completud refutacional también es cierto, aunque su demostración es compleja y no se abordará en este curso.

Teorema 2 (Completud Refutacional de la Resolución) *Si* Γ *es insatisfacible entonces existe una derivación de la cláusula vacía* $\Gamma \vdash_{\mathcal{R}} \square$.

2. Breve introducción a la programación lógica

Los lenguajes de programación más ampliamente usados, como C o Java, forman parte del paradigma de programación imperativa o procedimental cuyas características principales son:

- Un programa es una secuencia de instrucciones.
- La principal estructura de control son los ciclos: *while, repeat, for* etc.
- La operación de asignación $x := a$ es imprescindible.
- Las estructuras de control permiten seguir paso a paso las acciones que debe realizar un programa.
- Es decir, el programa especifica *cómo* se calculan los resultados.

En contraste los lenguajes de programación funcional (Haskell, Lisp, Scheme, ML) y lógica (Prolog) conforman la llamada programación declarativa cuyas características principales son:

- Un programa es una sucesión de definiciones.
- La principal estructura de control es la recursión.
- No existen ni ciclos ni operación de asignación
- El programa especifica *qué* se debe calcular, es decir, las propiedades que debe cumplir el resultado o solución a calcular.
- El *cómo* es irrelevante.

2.1. Ventajas de la programación declarativa

La programación declarativa no depende del lenguaje en particular.

- Si bien los programas imperativos pueden ser rápidos y especializados, un programa declarativo es más general, corto y legible.
- Debido a lo anterior podemos decir que los programas declarativos son elegantes matemáticamente. Lo cual implica que es más fácil *verificar* si el programa cumple su especificación.
- Aprender programación declarativa permite al programador desarrollar un estilo de programación riguroso y disciplinado que puede ser usado ventajosamente sin importar el lenguaje de programación utilizado.
- Este estilo genera programas con una mejor ingenieria, más fáciles de depurar, mantener y modificar.

2.2. Lenguajes de programación lógica

Fue alrededor de las décadas de 1920 y 1930 que Jacques Herbrand [1] propuso en su tesis un método para verificar la validez de fórmulas en lógica de predicados utilizando un procedimiento para unificar fórmulas. Esta tesis es el fundamento de la programación lógica que modela al cómputo a través de los llamados modelos de Herbrand donde:

1. el dominio es el universo formado por términos que representan objetos
2. las fórmulas que involucran predicados describen un problema

[1] Jacques Herbrand (1908-1931), lógico prominente que murió a la edad de 23 años en un accidente en los alpes.

Las fórmulas son usadas para expresar conocimiento, en particular, descripciones o algoritmos en forma de funciones parciales creando un programa lógico.

Realizar cómputos a partir de un programa lógico es obtener respuestas a partir de la información descrita. Las respuestas también llamadas metas son exactamente las sustituciones que asocian valores del universo de Herbrand a las variables de la meta.

Así, el significado (declarativo) de un programa y sus respuetas estan bien definidas a través de un modelo matemático que ofrece el universo de Herbrand.

Lo anterior permite establecer las características de un lenguaje de programación lógica: un lenguaje declarativo en el que los programas constan de definiciones plasmadas en fórmulas; en particular los predicados **especifican** información acerca de lo que se desea calcular, expresada mediante ciertos hechos y reglas, es decir, al establecer relaciones que describan propiedades de la información.

La evaluación de un programa en un lenguaje de programación lógica es **interactiva**: para activar el mecanismo de ejecución se necesita de una pregunta relacionada con la información dada en el programa, es decir, el programa $\mathbb{P}$ se activa al preguntar cierta información $\mathcal{C}$ lo cual formalmente requiere **verificar** si $\mathbb{P} \models \mathcal{C}$.

Los fundamentos del paradigma de programación lógica se sirven básicamente de la lógica de primer orden, en particular el mecanismo de ejecución se basa en la regla de resolución binaria con unificación de Robinson. Esta regla (propuesta alrededor de 1960 por Alan Robinson) ofrece un algoritmo para unificar términos que sirvió para la implementación de Prolog. El algoritmo que nosotros estudiamos y utilizamos es el de Martelli y Montanari descrito en la nota 9.

El poder expresivo de la lógica inspirado en la tesis de Herbrand y el proceso refinado para la resolución de Robinson, fueron usados en combinación por Robert Kowalski, Alan Colmerauer y Philippe Roussel alrededor de 1970 para crear el primer lenguaje de programación lógico: Prolog.

3. Resolución Binaria en Programación Lógica

3.1. Notaciones para cláusulas

Además de la notación de cláusulas utilizada hasta ahora, se pueden utilizar otras, aquí mencionamos dos de ellas.

Notación conjuntista Una notación para cláusulas utilizada por diversos autores es la notación conjuntista que representa a la cláusula $\mathcal{C} = \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$ como el conjunto

$$\mathcal{C} = \{\ell_1, \ell_2, \ldots, \ell_n\}$$

La justificación de esta notación está en el hecho de que el orden de las literales no importa, ni tampoco el hecho de que haya literales repetidas. Esta notación **no** será utilizada en nuestro curso.

Notación para programación lógica La programación lógica utiliza cláusulas escritas de una manera particular que ahora describimos.
Consideremos una cláusula $\mathcal{C}$ de la forma

$$\mathcal{C} = \ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$$

mediante conmutatividad del $\vee$ podemos reescribir a C de la siguiente forma, agrupando las literales positivas y negativas:

$$\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_j \vee Q_1 \vee Q_2 \vee \ldots \vee Q_k$$

donde $j + k = n$, los P_i y los Q_l son predicados, llamados átomos en programación lógica (por el momento omitimos los argumentos de cada átomo pues no son relevantes).
Ahora mediante las leyes de De Morgan podemos hacer la siguiente transformación:

$$\neg(P_1 \wedge P_2 \wedge \ldots \wedge P_j) \vee (Q_1 \vee Q_2 \vee \ldots \vee Q_k)$$

Enseguida podemos escribir esta expresión como una implicación:

$$P_1 \wedge P_2 \wedge \ldots \wedge P_j \rightarrow Q_1 \vee Q_2 \vee \ldots \vee Q_k$$

Convenimos en sustituir las conjunciones y las disyunciones del consecuente con comas. Siempre debe recordarse que las comas del antecedente representan conjunciones mientras que las del consecuente representan disyunciones.

$$P_1, P_2, \ldots, P_j \rightarrow Q_1, Q_2, \ldots, Q_k$$

Finalmente convenimos en escribir la implicación al revés obteniendo:

$$Q_1, Q_2, \ldots, Q_k \leftarrow P_1, P_2, \ldots, P_j$$

que es la presentación de $\mathcal{C}$ en notación de programación lógica.
En tal caso los átomos $Q_1, Q_2, \ldots, Q_k$ forman la **cabeza** de la cláusula $\mathcal{C}$ y los átomos $P_1, P_2, \ldots, P_j$ constituyen el **cuerpo** de la cláusula $\mathcal{C}$.

3.2. Clasificación de cláusulas

Las diversas formas de una cláusula $\mathcal{C} = Q_1, Q_2, \ldots, Q_k \leftarrow P_1, P_2, \ldots, P_j$. según sean k y j son las siguientes:

1. Si $k = 1$ y $j = 0$, entonces $\mathcal{C}$ es de la forma

   $$Q_1 \leftarrow$$

   en este caso el cuerpo está vacío y la cabeza es un átomo. Tal cláusula se conoce como **hecho**.

2. Si $k \geq 1$ y $j = 0$, entonces $\mathcal{C}$ es de la forma

   $$Q_1, \ldots, Q_k \leftarrow$$

   El cuerpo está vacío y la cabeza es una disyunción de átomos. Esta cláusula se llama **cláusula positiva**.

3. Si $k = 1$ y $j \geq 0$, entonces $\mathcal{C}$ es de la forma

   $$Q_1 \leftarrow P_1, \ldots, P_j$$

   Esta cláusula se llama **cláusula de Horn**, *cláusula definida* o **regla**.

4. Si $k > 1$ y $j \geq 0$, entonces $\mathcal{C}$ es de la forma

   $$Q_1, Q_2, \ldots, Q_k \leftarrow P_1, P_2, \ldots, P_j$$

   Este es el caso general y se llama **cláusula disyuntiva** o cláusula que no es de Horn.

5. Si $k = 0$ y $j \geq 1$, entonces $\mathcal{C}$ es de la forma

   $$\leftarrow P_1, \ldots, P_j$$

   La cabeza esta vacía. Este tipo de cláusula se llama **meta**, objetivo o cláusula negativa.

6. Si $k = 0$ y $j = 0$, entonces $\mathcal{C}$ es de la forma

   $$\leftarrow$$

   Esta cláusula representa a la **cláusula vacía**.

3.3. Resolución binaria y programas lógicos

Con la nueva notación la regla de resolución binaria se escribe como sigue:

$$\frac{\begin{array}{c}\mathcal{C} = Q_1, \ldots, Q_k, \ell \leftarrow P_1, \ldots, P_j \\ \mathcal{D} = S_1, \ldots, S_m \leftarrow \ell', R_1, \ldots, R_n \\ \mu \text{ un umg de } \{\ell, \ell'\} \qquad Var(\mathcal{C}) \cap Var(\mathcal{D}) = \varnothing\end{array}}{(Q_1, \ldots, Q_k, S_1, \ldots, S_m \leftarrow P_1, \ldots, P_j, R_1, \ldots R_n)\mu}$$

Definición 1 *Un programa lógico $\mathbb{P}$ es un conjunto finito de cláusulas no negativas.*

De ahora en adelante nos interesa hacer resolución principalmente sobre programas lógicos definidos, que son aquellos que el lenguaje de programación PROLOG entiende.

Definición 2 *Un programa lógico definido $\mathbb{P}$ es un conjunto finito de cláusulas de Horn, es decir, un conjunto finito de hechos y reglas.*

Omitiremos el adjetivo "definido" pues no trataremos con otro tipo de programas. Obsérvese que las metas nunca son parte de un programa lógico, sino que sirven para interactuar con un programa mediante un intérprete que se encarga de buscar la cláusula vacía mediante resolución binaria. Veamos un par de ejemplos.

Ejemplo 3.1 Considérese el programa para la suma de naturales

$$\mathbb{P}_+ = \{Px0x \leftarrow,\ Pxsysz \leftarrow Pxyz\}$$

Queremos saber si $\mathbb{P}_+ \models Ps0s0w$, es decir, cuánto es $1 + 1$.
El principio de refutación y la correctud de la resolución nos dice que basta agregar la meta $\leftarrow Ps0s0w$ a $\mathbb{P}_+$ y obtener la cláusula vacía.

1.	$Px0x$	$\leftarrow$		
2.	$Pxsysz$	$\leftarrow$	$Pxyz$	
3.		$\leftarrow$	$Ps0s0w$	
4.		$\leftarrow$	$Ps00z$	$res(2, 3, [x, y, w := s0, 0, sz])$
5.		$\leftarrow$		$res(4, 1, [x, z := s0])$

De manera que $Ps0s0w$ es consecuencia de $\mathbb{P}_+$. Más aún, la composición de los unificadores utilizados nos devuelve $[w := ss0]$ que es la respuesta buscada.

Ejemplo 3.2 Veamos ahora un programa para el producto de naturales:

$$\mathbb{P}_\times = \mathbb{P}_+ \cup \{M0x0 \leftarrow, Msxyz \leftarrow Mxyv, Pvyz\}$$

Nos preguntamos cuánto es 1×2, la consecuencia lógica asociada es $\mathcal{P}_\times \models Ms0ss0w$

1.	$M0u0$	$\leftarrow$		
2.	$Msxyz$	$\leftarrow$	$Mxyv, Pvyz$	
3.	Px_10x_1	$\leftarrow$		
4.	$Px_2sy_2sz_2$	$\leftarrow$	$Px_2y_2z_2$	
5.		$\leftarrow$	$Ms0ss0w$	
6.		$\leftarrow$	$M0ss0v, Pvss0w$	$res(2, 5, [x, y, z := 0, ss0, w])$
7.		$\leftarrow$	$P0ss0w$	$res(1, 6, [u, v := ss0, 0])$
8.		$\leftarrow$	$P0s0z_2$	$res(4, 7, [x_2, y_2, w := 0, s0, sz_2])$,
9.		$\leftarrow$	$P00z_3$	$res(4$ renombrando con $[z_2 := z_3], 8, [x_3, y_3, z_2 := 0, 0, sz_3])$
10.		$\leftarrow$		$res(3, 9, [x_1, z_3 := 0, 0])$

Para obtener el valor de w componemos los valores necesarios de los unificadores utilizados, esto se conoce como *sustitución de respuesta*:

$$[w := sz_2][z_2 := sz_3][z_3 := 0], \text{ es decir }, [w := ss0]$$

Obsérvese que en la cláusula 6 existe un no determinismo, podemos resolver cualquiera de las dos literales. Si hubiéramos elegido resolver la segunda la derivación sería:

7.	$\leftarrow$	$M0ss0v, Pvs0z_2$	$res(4, 6, [x_2, y_2, w := v, s0, sz_2])$
8.	$\leftarrow$	$M0ss0v, Pv0z_3$	$res(4 \text{ renombrando con } [z_2 := z_3], 7, [x_2, y_2, z_2 := v, 0, sz_3])$
9.	$\leftarrow$	$M0ss0v$	$res(3, 8, [x_1, z_3 := v, v])$
10.	$\leftarrow$		$res(1, 9, [u, v := ss0, 0])$

La sustitución de respuesta es:

$$[w := sz_2][z_2 := sz_3][z_3 := v][v := 0], \text{ es decir }, [w := ss0]$$

En este caso se obtuvo la misma respuesta en ambos casos, sin embargo, la programación lógica es sensible al orden de las cláusulas y al orden en que se eligen las literales para resolver. A continuación ejemplificamos este fenómeno.

Ejemplo 3.3 Modificamos el programa para la suma de manera que la recursión sea en la primera variable.

$$\mathbb{P}'_+ = \{P0xx \leftarrow,\ Psxysz \leftarrow Pxyz\}$$

Semánticamente los resultados deben ser iguales, y lo son, pero operacionalmente existen grandes diferencias.

Veamos qué sucede ante la meta $\leftarrow Pws0ss0$. La respuesta buscada es $w := s0$.

1.	$Psx_2y_2sz_2$	$\leftarrow$	$Px_2y_2z_2$	
2.	$P0xx$	$\leftarrow$		
3.		$\leftarrow$	$Pws0ss0$	
4.		$\leftarrow$	Px_2s0s0	$res(1, 3, [w, y_2, z_2 := sx_2, s0, s0])$

En este punto hay dos elecciones para resolver 4, la regla 1 ó el hecho 2; como 1 se lista primero, se intenta con dicha regla:

$$5. \quad \leftarrow \quad Px_3s00 \quad res(1 \text{ renombrando con } [x_2 := x_3], 4, [x_2, y_2, z_2 := sx_3, s0, 0])$$

En este punto no es posible resolver 5 y la búsqueda de $\leftarrow$ ha fallado. Regresamos al último punto donde hubo una elección y elegimos de manera distinta, en este caso 4 con 2:

$$5. \quad \leftarrow \quad res(2, 4, [x_2, x := 0, s0])$$

La búsqueda tiene éxito y la sustitución de respuesta es $[w := s0]$.

Obsérvese en este ejemplo además que estamos usando un mismo programa para una tarea distinta a aquella por la cual se diseñó. El programa fue diseñado para sumar, pero también sirve para restar, el predicado $Rxyz$ tal que $z = x - y$ puede programarse como:

$$Rxyz \leftarrow Pyzx$$

Esto se conoce como uso no estándar de un programa lógico y es una característica exclusiva de este tipo de programas.

En el ejemplo anterior vimos que una elección inadecuada de una clásula para resolver puede causar que falle la búsqueda de $\leftarrow$. Veamos ahora otro ejemplo que causa no terminación de la búsqueda.

Ejemplo 3.4 Calculemos nuevamente 1×2, esta vez con el segundo programa para la suma

$$\begin{array}{lrcl}
1. & M0u0 & \leftarrow & \\
2. & Msxyz & \leftarrow & Mxyv, Pvyz \\
3. & P0x_1x_1 & \leftarrow & \\
4. & Psx_2y_2sz_2 & \leftarrow & Px_2y_2z_2 \\
5. & & \leftarrow & Ms0ss0w \\
6. & & \leftarrow & M0ss0v, Pvss0w \quad res(2,5,[x,y,z := 0, ss0, w])
\end{array}$$

Para continuar, si elegimos para resolver ahora la segunda literal en 6, como no hay información para v, w, ambas cláusulas para la suma son aplicables. Si elegimos la cláusula 4 obtenemos:

$$7. \quad \leftarrow \quad M0ss0sx_2, Px_2ss0z_2 \quad res(6,4,[v,w,y_2 := sx_2, sz_2, ss0])$$

Como se observa obtuvimos la misma segunda literal con variables renombradas. Si nuevamente elegimos resolver la segunda literal de 7 con 4 obtenemos:

$$\begin{array}{lrcl}
8. & \leftarrow & M0ss0ssx_3, Px_3ss0z_3 & res(7, 4 \text{ renombrando con} \\
 & & & [x_2, z_2 := x_3, z_3], [x_2, z_2, y_2 := sx_3, sz_3, ss0])
\end{array}$$

Si seguimos usando la misma elección la búsqueda no terminará jamás aún cuando $\square$ sí puede obtenerse. Más aún, al cambiar la elección a la primera literal de la meta, la respuesta será *No* pues la meta no se puede resolver. En conclusión, un cambio en el orden de las cláusulas puede causar no terminación o bien una respuesta negativa a pesar de que una respuesta afirmativa existe.

4. Resolución en Prolog

Iniciamos la sección revisando la notación de cláusulas que PROLOG utiliza. Recordemos que sólo se permiten cláusulas definidas o de Horn en donde $\leftarrow$ se sustituye por `:-` y cada cláusula termina en punto. Éstas son de alguna de las siguientes formas:

- Reglas: una literal positiva y al menos una literal negativa:

$$P \text{ :- } Q_1, \ldots, Q_m.$$

- Hechos: una literal positiva y ninguna literal negativa:

$$P.$$

- Metas: ninguna literal positiva y al menos una literal negativa:

$$\text{?- } Q_1, \ldots, Q_m$$

Es importante observar que las metas son el medio para interactuar con un programa, el cual consta únicamente de hechos y reglas. Más aún el símbolo ?- es el prompt de PROLOG.

PROLOG usa una versión especial de la regla de resolución binaria, esta versión específicamente se adapta a programas con cláusulas de Horn. En un cómputo en PROLOG tenemos un programa lógico $\mathbb{P}$ y una cláusula meta $\mathcal{C}$ que expresa el problema que queremos resolver o más bien la información que queremos confirmar a partir del programa. En la versión de resolución implementada en PROLOG, una de las dos cláusulas que se resuelven debe ser siempre la cláusula meta, y el resolvente siempre se convierte en la nueva cláusula meta, esto se llama resolución lineal. Detalladamente se procede como sigue:

1. Se tiene dada una cláusula meta ?- $G_1, \ldots, G_k$.
2. Buscar en el programa una cláusula o una variante de una cláusula P :- $Q_1, \ldots, Q_n$ tal que
   - G_1 se unifica con P.
   - μ es el unificador más general de $\{G_1, P\}$

   si no hay tal cláusula terminar con falla.
3. Reemplazar G_1 con $Q_1, \ldots, Q_n$.
4. Aplicar μ al resultado, obteniendo

$$?- Q_1\mu, \ldots, Q_n\mu, G_2\mu, \ldots, G_k\mu.$$

5. Si el resultado es la cláusula vacía $\square$ (que en PROLOG se ve como ?-) entonces terminar, reportando éxito, y devolver como solución la composición de todos los unificadores μ aplicados a las variables de la meta original.

Primero, se toma la cláusula meta y se elige una de sus literales. En principio podemos seleccionar cualquier literal, pero PROLOG siempre elige la literal más a la izquierda de la cláusula meta. Este proceso se conoce como **resolución con función de selección**. La función de selección de PROLOG, al aplicarse a una sucesión de expresiones, siempre devuelve la que está más a la izquierda, es decir, la primera. Más aún, para seguir resolviendo siempre se utiliza la meta actual obtenida mediante el proceso anterior, es decir, está prohibido hacer un paso de resolución sin involucrar a la meta actual, esta restricción se conoce como **resolución lineal**. De manera que la combinación se conoce como **resolución lineal con función de selección** o SLD-resolución. Por lo tanto, dado que sólo se permiten cláusulas definidas, el método particular de resolución implementado en PROLOG se conoce también como **SLD-resolución** (en inglés Selected, Linear, Definite resolution).
El segundo paso, después de seleccionar una literal de la cláusula meta, es buscar en el programa una cláusula cuya cabeza se unifique con la literal seleccionada (ésta también se conoce como submeta). El resolvente de la meta con la cláusula que seleccionamos es el resultado de:

a) Eliminar la literal elegida de la cláusula meta.

b) Reemplazarla con el cuerpo de la cláusula del programa.

c) Tomar el unificador de la meta y la cabeza de la cláusula elegida en el programa y aplicarlo a la meta recien derivada.

Obsérvese que si resolvemos la meta con un hecho, entonces se reemplaza con nada (puesto que los hechos son cláusulas sin cuerpo). Es decir, simplemente removemos la submeta de la meta. Cuando la meta finalmente es la cláusula vacía, entonces hemos reducido la meta original a una colección de hechos, y por lo tanto hemos

terminado, probando así la meta original; solo resta considerar las variables que figuran en la meta original, y el resultado de aplicar a estas variables la composición de los unificadores que hemos usado durante el cómputo nos da la respuesta esperada.

Ejemplo 4.1 Considérese el siguiente programa, que representa a la operación de suma en los números naturales mediante la función sucesor $s(X)$.

```
suma(0,X3,X3).
suma(s(X2),Y2,s(Z2)):-suma(X2,Y2,Z2).
```

Supóngase que queremos saber cuánto es $1 + 2$, la pregunta correspondiente es:

```
?- suma(s(0),s(s(0)),X1).
```

Sólo hay una literal en la cláusula meta, así que la selección es única. Esta submeta se unifica con la cabeza de la regla del programa. El unificador más general es:

$$\mu = [X_2 := 0, Y_2 := s(s(0)), X_1 := s(Z_2)]$$

Por lo tanto nuestro resolvente será la proxima cláusula meta:

$$:-\texttt{suma}(\texttt{X2}, \texttt{Y2}, \texttt{Z2})\mu.$$

o, haciendo la sustitución,

$$:-\texttt{suma}(\texttt{0}, \texttt{s}(\texttt{s}(\texttt{0})), \texttt{Z2}).$$

esta meta se puede resolver con el hecho del programa con la sustitución

$$\tau = [X3 := s(s(0)), Z2 := s(s(0))]$$

La respuesta a la meta original es la aplicación de la composición de las sustituciones encontradas a las variables de la meta original, Esto es:

$$X_1\mu\tau = s(Z_2)\tau = s(s(s(0))).$$

Es decir, $X_1 = 3$ que es la respuesta esperada.

4.1. Árboles asociados a un programa lógico

Definición 3 *Sean $\mathbb{P}$ un programa lógico y G una meta. Un árbol SLD* [2] *es un árbol n-ario posiblemente infinito cuya raíz es la meta G y cuyos nodos tienen metas exclusivamente; el hijo de un nodo G_i es la nueva meta G_j obtenida a partir de la resolución binaria de G_i y una cláusula del programa.*

En el caso particular de un programa en Prolog, al árbol SLD se le llama árbol de búsqueda (Prolog search tree) y muestra todos los caminos que recorre el intérprete en la búsqueda por la cláusula vacía $\square$. Aquellas ramas finitas que terminan en la cláusula vacía $\square$ se llaman ramas de éxito, las que terminan en una cláusula no vacía se llaman ramas de fallo.

[2] En inglés SLD-resolution tree o SLD-tree.

Definición 4 *Una SLD-derivación es un árbol potencialmente infinito que consistente de metas $G_0, G_1, \ldots, G_n$, cláusulas de programa $\mathcal{C}_0, \ldots, \mathcal{C}_m$ y unificadores $\mu_0, \ldots, \mu_k$ tales que:*

- *G_{n+1} es un SLD-resolvente de G_n y $\mathcal{C}_m$ via el umg μ_k, para toda n.*

En este árbol si C_R es la cláusula resolvente a partir de la cláusula de programa C y de la meta G entonces el árbol tendrá a C_R como hijo de C y G. Si la derivación Δ es finita digamos $G_0, \ldots, G_i$ decimos que i es la longitud de Δ.

Definición 5 *Sean $\mathbb{P}$ un programa lógico y G una meta. Una SLD-refutación para $\mathbb{P} \cup \{G\}$ es una SLD-derivación finita Δ con metas $G_0, \ldots, G_n$ y cláusulas de programa $\mathcal{C}_0, \ldots, C_m$ tal que:*

- *$G_0 = G$ y $G_n = \square$.*

Ejemplo 4.1 Considera el siguiente programa:

1. $P(a)$.
2. $P(b)$.
3. $Q(a)$.
4. $Q(b)$.
5. $R(b)$.
6. $S(X) :- P(X), Q(X), R(X)$.

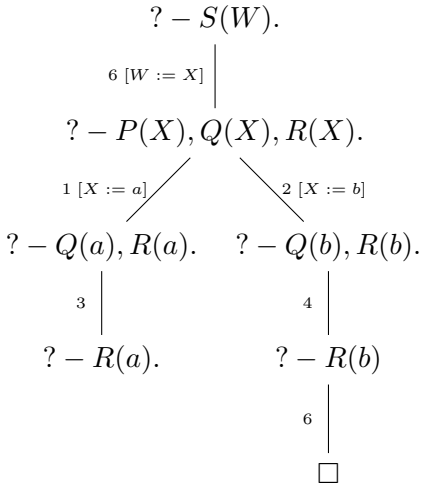


Figura 1: Árbol SLD para la meta $?. - S(W)$.

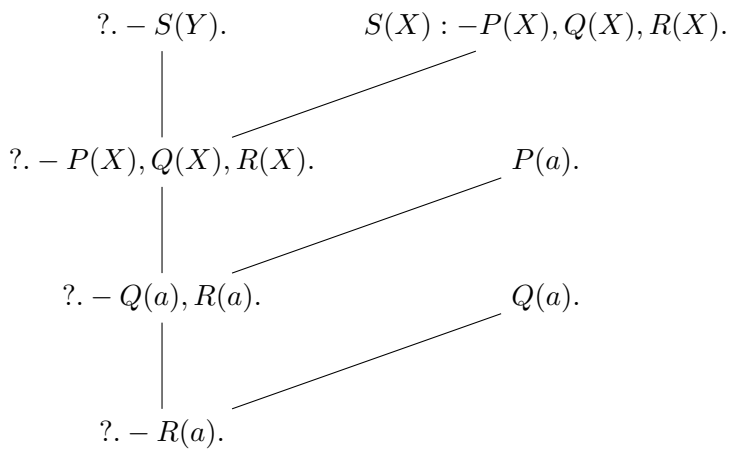


Figura 2: SLD derivación para $?. - S(Y)$.

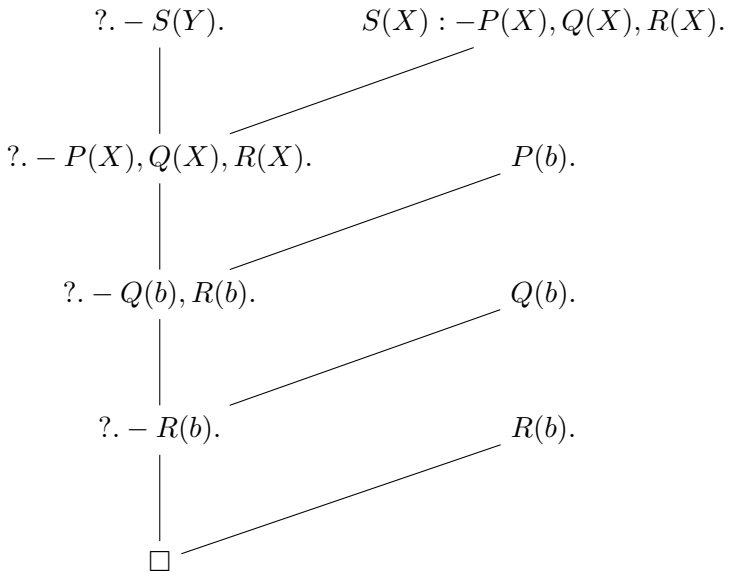


Figura 3: SLD refutación para $?. - S(Y)$.

5. Semántica de programas lógicos

Una parte importante de cada paradigma de programación es la semántica, por medio de la cual se le da significado a un programa, situación que nos permite describir formalmente lo que éste calcula. Al inicio de esta nota mencionamos una ventaja de la programación declarativa: la elegancia matemática resultado de la descripción del programa mediante enunciados precisos, así el significado del programa es claro y facilita su verificación.

En esta sección trataremos brevemente con dos clases de semántica para programas lógicos: la declarativa y la procedimental u operacional.

Comencemos observando que una cláusula `P :- Q1,Q2,...,Qn` de PROLOG tiene una interpretación declarativa y una interpretación procedimental:

- Declarativa: P es válida si Q_1 y Q_2 y ... y Q_n son válidas.
  La interpretación declarativa permite discutir la correctud de la cláusula.
- Operacional: Para ejecutar (probar) P basta ejecutar Q_1 y Q_2 y ... y Q_n.
  La interpretación operacional permite considerar a la cláusula como la definición de un proceso. Esta semántica Se genera con la ejecución del programa mediante las estrategias de control del intérprete de PROLOG.

Una pregunta importante es si ambas interpretaciones generan la misma información, para contestarla necesitamos hablar del concepto de respuesta de manera formal.

Definición 6 (Respuesta) *Sean $\mathbb{P}$ un programa lógico y $G =$?- $G_1, \ldots, G_m$ una meta. Una respuesta para $\mathbb{P} \cup \{G\}$ es una sustitución σ tal que $Var(G_1) \cup \ldots \cup Var(G_m) \subseteq \mathsf{dom}(\sigma)$, es decir una sustitución que incluye a las variables de G.*

Definición 7 (Respuesta correcta) *Decimos que una respuesta σ para $\mathbb{P} \cup \{G\}$ es correcta si $\mathbb{P} \models G_i\sigma$ para toda $1 \leq i \leq m$, o equivalentemente si $\forall \mathbb{P} \models \forall\big((G_1 \wedge \ldots \wedge G_m)\sigma\big)$.*

Recordemos que $\forall\varphi$ denota a la cerradura universal de φ obtenida cuantificando universalmente todas las variables libres de φ. Análogamente $\forall\mathbb{P}$ se obtiene de $\mathbb{P}$ al cuantificar universalmente todas las variables de cada una de sus cláusulas. Intuitivamente una respuesta correcta para $\mathbb{P}\cup\{G\}$ corresponde a una consecuencia lógica particular del programa, y es entonces una significado declarativo del programa.

Ejemplo 5.1 Considérese el programa

$$\mathbb{P} = \{mq(0, suc(X)),\ mq(suc(Y), suc(X) \text{ :- } mq(Y, X)\}.$$

Entonces $\sigma = [Y := suc(suc(0))]$ es una respuesta correcta para $? - mq(0, Y)$ pues

$$\forall\mathbb{P} \models mq(0, Y)[Y := suc(suc(0))]$$

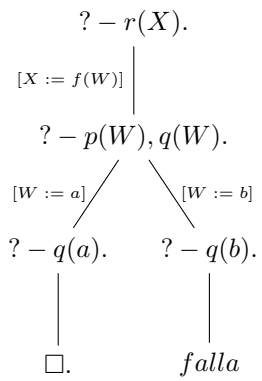
Definición 8 (Respuesta computada) *Sean $\mathbb{P}$ un programa lógico y G una meta. Una sustitución σ es una respuesta computada para $\mathbb{P}\cup\{G\}$ si y sólo si existe una rama de éxito en el árbol de SLD-resolución (o árbol de búsqueda) de longitud n con unificadores más generales $\mu_0, \ldots, \mu_{n-1}$ de tal forma que $\sigma = \mu_0\mu_1 \ldots \mu_{n-1}\restriction_{Var(G)}$*

Es decir σ es una respuesta computada para $\mathbb{P}\cup\{G\}$ si y sólo si σ es la restricción de la composición de los unificadores de una rama de éxito en el árbol de SLD-resolución para $\mathbb{P}\cup\{G\}$.

Intuitivamente una respuesta computada corresponde al resultado del proceso de inferencia por parte del sistema. Las respuestas computadas son aquellas que el intérprete de PROLOG devuelve al usuario.

Ejemplo 5.2 Si $\mathbb{P} = \{p(a),\ p(b),\ q(a),\ r(f(X)) \text{ :- } p(X), q(X)\}$ entonces tenemos la siguiente rama de éxito en el árbol de SLD-resolución para la meta $G =$?- $r(X)$:

1.	$p(a)$.	Hip.
2.	$p(b)$	Hip.
3.	$q(a)$.	Hip.
4.	$r(f(Y)) \text{ :- } p(Y), q(Y)$.	Hip.
5.	?- $r(X)$	$Meta$
6.	?- $p(Y), q(Y)$	$SLDRes(4, 5, [X := f(Y)])$
7.	?- $q(a)$	$SLDRes(6, 1, [Y := a])$
8.	$\square$.	$SLDRes(3, 7)$



La composición de los unificadores es $[X := f(a), Y := a]$ de manera que la sustitución $[X := f(a)]$ es una respuesta computada para $\mathbb{P}\cup\{$?- $r(X)\}$.

El significado de un programa lógico se debe dar mediante sus respuestas, intuitivamente el significado es el conjunto de respuestas. Por ejemplo el siguiente programa $\mathbb{P}$, implementa la búsqueda de caminos en una gráfica:

```
edge(a,b).
edge(b,c).
edge(b,d).
edge(c,d).
edge(e,f)
path(X,X).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

De manera que el significado intensional de $\mathbb{P}$ es el conjunto de todos los caminos posibles (especificando todo los vértices del camino) en la gráfica. Pero recordemos que la programación lógica tiene un uso no estándar, por ejemplo el programa para concatenar dos listas, sirve también para descomponer una lista en dos partes. Por lo que con esta definición informal no es tan claro cuál es el significado. Más aún, dado un programa lógico $\mathbb{P}$, podemos asignarle dos significados:

- Significado declarativo: el significado de un programa lógico $\mathbb{P}$ es el conjunto de todas las consecuencias lógicas del programa, noción asociada a la de respuesta correctas.
- Significado operacional: el significado de un programa lógico $\mathbb{P}$ es el conjunto de todas los éxitos del programa, noción asociada a la de respuesta computada.

Por supuesto que el significado debería ser único, pero no es claro que las dos nociones recién enunciadas sean equivalentes. De los ejemplos anteriores se observa que una respuesta correcta también es una respuesta computada y viceversa, ¿es esto válido en general?, es decir ¿Toda respuesta correcta puede computarse? y ¿Toda respuesta computada es correcta? esto ayudará a probar que las dos semánticas coinciden.
Resulta que en efecto ambos conceptos resultan equivalentes de cierta manera. Los enunciados formales para esta equivalencia, así como su demostración requieren de conceptos como los modelos de Herbrand o sintácticos.