Capítulo 1

Fundamentos teóricos

La Programación Lógica nos permite definir un problema especificándolo en «alto nivel». Describimos el problema mediante relaciones, que pueden combinarse mediante conectivos. Luego, es posible realizar pruebas sobre esta especificación, originadas por el deseo de saber si determinadas consultas se satisfacen o no.

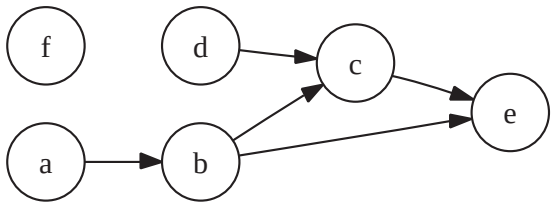

Figura 1.1: grafo dirigido

Por ejemplo, un grafo se describe determinando cuáles son sus aristas y sus vértices (por si existe algún nodo aislado). Así, el grafo de la figura 1.1 se representa por el siguiente conjunto de predicados:

vértice(a)	vértice(b)	vértice(c)
vértice(d)	vértice(e)	vértice(f)
arista(a,b)	arista(b,c)	arista(d,c)
arista(b,e)	arista(c,e)	

¿Cómo representamos el hecho de que hay un camino entre dos vértices?

hay_camino(X,Y) **si** arista(X,Y)
hay_camino(X,Y) **si** arista(X,Z) **y** hay_camino(Z,Y).

La definición anterior establece bajo qué condiciones existe un camino entre dos vértices cualesquiera. Esto implica una cuantificación universal implícta en todas las reglas.

Por otro lado, estamos dando dos condiciones para que haya un camino entre

dos vértices en el grafo; pero entonces, existe una disyunción implícita entre las dos reglas.

Ahora, querríamos saber si existe una arista o algún camino entre un par de nodos dados, o qué caminos existen partiendo de un nodo, etc. Por ejemplo:

Ejemplo
? arista(a,b)
? arista(a,f)
? hay_camino(a,e)

Las primeras consultas se contestan simplemente revisando la información que hay en la base. Sin embargo, para la tercer consulta, es necesario «inferir» nuevo conocimiento a partir de la información que se tiene. En otras palabras, se debe probar la existencia o inexistencia de un camino entre los vértices *a* y *e*, según la especificación dada.

Para esto puedo utilizar la propia definición de la relación *hay_camino* y leerla *proceduralmente*: para probar que hay un camino, primero encuentro un vértice con el que el vértice *a* tiene una arista, y luego pruebo que hay un camino entre este vértice y el vértice *e*

El objetivo de este curso es ver cómo se puede definir un problema como un conjunto de fórmulas lógicas, y el modo en que *automáticamente* se puede contestar las consultas que se realizan sobre el programa.

Para esto vamos a ver que:

- La especificación de un problema (elementos y relaciones) es una fórmula de primer orden (programa lógico).
- La consulta también es una fórmula de primer orden.
- La respuesta a una consulta es afirmativa si es una consecuencia lógica de la especificación del problema.
- La consulta se responde por un método de inferencia: dado un conjunto de axiomas, el programa lógico, realizamos una demostración utilizando reglas de inferencia. Esta inferencia debe realizarse *automáticamente*.

En definitiva, vamos a ver cómo un intérprete *Prolog* responde a una consulta.

1.1. Lógica de $1^{er.}$ orden

1.1.1. Sintaxis

Estas definiciones se extraen del Lloyd [2]

Definición Un *alfabeto de $1^{er.}$ orden* consiste de siete clases de símbolos:

- variables (x, y, z,...)

- constantes (a, b, c,...)
- functores (f, g, h, ...)
- símbolos de predicados (p, q, r,...)
- conectivos: {↔, →, ∧, ∨, ¬}
- cuantificadores: {∀, ∃}
- puntuación: { ')', '(', ',' }

Definición Término:

1. Una variable es un término.
2. Una constante es un término.
3. Si f es una función n-aria y $t_1 \ldots t_n$ términos, f(t_1,...,t_n) es un término.
4. Estos son todos los términos.

Definición Fórmula bien formada (f.b.f.):

1. Si p es una predicado n-ario y $t_1 \ldots t_n$ términos, p(t_1,...,t_n) es una fórmula (*átomo*).
2. Si F y G son fórmulas, también lo son:
   - (¬F)
   - (F∨G)
   - (F∧G)
   - (F→G)
   - (F↔G)
3. Si F es una fórmula y x una variable, también son fórmulas:
   - (∀x F)
   - (∃x F)
4. Estas son todas las fórmulas.

Definición Lenguaje de $1^{er.}$ orden (dado un alfabeto): es el conjunto de fórmulas que se pueden construir con los símbolos del alfabeto.

Ejemplo Retomando el ejemplo del grafo de la figura 1.1, se tiene lo siguiente:

- constantes: {a, b, c, d, e, f}
- functores: ∅
- variables: {x, y, z}
- predicados: {vértice, hay_camino}

- términos: {a, . . . , f, x, y, z}
- fórmulas atómicas: {vértice(a), vértice(x), hay_camino(x,y),. . . }
- fórmulas: { vértice(a), vértice(x), (∀x hay_camino(a,x)),. . . }
- lenguaje:{vértice(a), vértice(x), (∀x hay_camino(a,x)),. . . }

Ejemplo Podemos definir los naturales de la siguiente forma (representación unaria):

nat(0)
nat(suc(x)) ← nat(x)

mayor(suc(x),0)
mayor(suc(x),suc(y)) ← mayor(x,y)

En este caso, tenemos lo siguiente:

- constantes: {0}
- functores: {suc}
- variables: {x, y, z}
- predicados: {nat, mayor}
- términos: {0, suc(0), suc(suc(0)),. . . }
- fórmulas atómicas: {mayor(0,suc(0)), nat(suc(suc(0))) . . . }
- fórmulas: {nat(y), nat(x) → nat(suc(suc(x))), . . . }

Definición El *alcance* de $\forall x$ en $\forall x$ F es F. (idem para $\exists$)

Definición Una variable esta *ligada* si está inmediatamente luego de un cuantificador o es una ocurrencia dentro del alcance de un cuantificador. En caso contrario, diremos que la ocurrencia se encuentra *libre*.

Definición Una *fórmula cerrada* es una fórmula que no presenta ocurrencias de variables libres.

Definición ∀(F) es la *clausura universal* de F, y se obtiene agregando un cuantificador universal por cada variable con una ocurrencia libre en F.

Análogamente se define la *clausura existencial* ∃(F), la cual se obtiene agregando un cuantificador existencial por cada variable con una ocurrencia libre en F.

Definición Un *literal* es un átomo (literal positivo) o su negado (literal negativo).

Definición Una *cláusula* es una fórmula (cerrada) de la siguiente forma:

$$\forall x_1 \ldots \forall x_k (L_1 \vee \ldots \vee L_p)$$

donde cada L_i es un literal y $x_1 \ldots x_k$ son todas las variables que ocurren en $L_1 \vee \ldots \vee L_p$

Vamos a utilizar indistintamente las siguientes notaciones para las cláusulas:

- $\forall x_1 \ldots \forall x_k (A_1 \vee \ldots \vee A_p \vee \neg B_1 \vee \ldots \vee \neg B_q)$
- $(A_1 \vee \ldots \vee A_p \leftarrow B_1 \wedge \ldots \wedge B_q)$
- $\{A_1, \ldots, A_p, \neg B_1, \ldots, \neg B_q\}$

Definición Una *cláusula definida* es una cláusula que contiene exactamente un literal positivo:

- regla: $A_1 \leftarrow B_1 \wedge \ldots \wedge B_q$
- cláusula unitaria (o hecho): $A_1 \leftarrow$

Definición Una *programa definido* es un conjunto de cláusulas definidas.

Retomando el ejemplo del grafo, podemos ver que la formulación hecha para *hay_camino* responde a un programa lógico:

$vertice(a) \leftarrow$
$\ldots$
$arista(a, b) \leftarrow$
$\ldots$
$hay\_camino(X, Y) \leftarrow arista(X, Y).$
$hay\_camino(X, Y) \leftarrow arista(X, Z), hay\_camino(Z, Y)$

¿Qué sucede con las consultas? Si las formulamos en lógica de primer orden, no son cláusulas:

$$\exists X \quad hay\_camino(X, b) \wedge vertice(X)$$

Sin embargo, la negación de una consulta sí es una cláusula:

$$\forall X \quad \neg hay\_camino(X, b) \vee \neg vertice(X)$$

O sea:

$$\leftarrow hay\_camino(X, b), vertice(X)$$

Definición Una *objetivo definido* es una cláusula sin literales positivos:

$$\leftarrow B_1 \wedge \ldots \wedge B_q$$

Definición Una *cláusula de Horn* es una cláusula definida o un objetivo definido:

Definición La *cláusula vacía* (contradicción) se denota por $\square$.

En otras palabras, tanto los programas lógicos, como las consultas, van a ser modelados con cláusulas de Horn: los primeros son conjuntos de cláusulas definidas (reglas o hechos), mientras que las consultas van a ser negadas para así obtener objetivos definidos.

1.1.2. Semántica

Para poder establecer si una fórmula es verdadera o falsa es necesario darle algún significado a sus símbolos. Los conectivos y cuantificadores tienen un significado predefinido, pero no así las constantes, variables, símbolos de función y de predicado, dado que pueden variar de acuerdo a la interpretación que les demos.

Ejemplo La siguiente fórmula es verdadera si consideramos que el dominio de variación de las variables es el conjunto de los naturales y que el símbolo de predicado p se interpreta mediante la relación menor entre naturales.

$$\forall x \exists y \quad p(x, y)$$

Parafraseando sería: *para todo natural x existe un natural y tal que x es menor que y (o sea, el conjunto de naturales no tiene extremo superior).*

En cambio, la fórmula

$$\exists y \forall x \quad p(x, y)$$

con la misma interpretación es falsa.

Definición Sea L un lenguaje de $1^{er.}$ orden. Una *interpretación* I para L consiste de:

I. Un conjunto D no vacío (dominio de la interpretación)

II. Un mapeo M_c de las constantes de L en D

III. Un mapeo M_f para los símbolos de función de L, tal que a f n-ario le corresponde una función $D^n \rightarrow D$

IV. Un mapeo M_p para los símbolos de predicado de L, tal que a p n-ario le corresponde una función $D^n \rightarrow \{v, f\}$ (o sea, una relación en D^n).

Una interpretación nos permite asignar un valor de verdad a cualquier fórmula de L. Observar que, si la fórmula no es cerrada, necesitamos además fijar un valor en D para las variables libres:

Definición Sea L un lenguaje de $1^{er.}$ orden e I una interpretación para L de dominio D. Una *asignación de valores a variables* V (según I) es una asignación de valores en D a las variables de L.

Definición Se define *valor de verdad de una fórmula* F según una interpretación I y una asignación V por recursión estructural en los términos y en las f.b.f.

Los términos se mapean a elementos del dominio mediante una función h:

1. A cada variable v se le da un valor $h(v)$ según V
2. A cada constante c se le da un valor $h(c)$ según I, $M_c(c)$
3. El valor asociado a un término $f(t_1, \ldots, t_n)$ es $h(f(t_1, \ldots, t_n) = M_f(h(t_1), \ldots, h(t_n))$ ($h(f)$ en D).

Luego, el valor de verdad está dado por:

1. Si la fórmula F es atómica: $p(t_1, ..., t_n)$ su valor es $M_p(h(t_1), ..., h(t_n))$.
2. Si F es construida con conectivos, su valor de verdad está dado por la siguiente tabla:

F_1	F_2	$\neg F_1$	$F_1 \wedge F_2$	$F_1 \vee F_2$	$F_1 \rightarrow F_2$	$F_1 \leftrightarrow F_2$
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

3. Si F es de la forma $\exists x G$, F es verdadera si existe $d \in D$, tal que G es verdadera para I y $V(x/d)$, donde $V(x/d)$ es V excepto para la variable x, a la que se le asigna el valor d; en caso contrario es falsa.
4. Si F es de la forma $\forall x G$, F es verdadera si para cada $d \in D$, G es verdadera para I y $V(x/d)$; en caso contraria es falsa.

Ejemplo A continuación se presentan tres posibles casos:

1. En lógica proposicional no tenemos variables, constantes, funciones ni cuantificadores. Además, los predicados son 0-arios, y los mapeos son constantes. Luego, el valor de verdad de una fórmula se calcula según la tabla de verdad.

2. Sean las fórmulas $\forall x\ p(x)$, $\exists x\ \neg p(x)$, $\exists x\ q(x,y)$, $\forall x\ (p(x) \rightarrow q(f(x,a)))$

   Una posible interpretación I está dada por:

   $D = 1, 2$
   $M_c(a) = 1$
   $M_f(1) = 1,\ M_f(2) = 1$
   $M_p(1) = V,\ M_p(2) = F$
   $M_q(1,1) = \ldots$

3. $\forall x \exists y\ p(x,y)$, $\exists y \exists x\ p(x,y)$

   siendo I:
   $D = Nat$,
   $M_p = \leq$

En el último ejemplo, los valores de verdad se obtienen utilizando resultados conocidos de los naturales.

Para cada una de estas fórmulas existe al menos una interpretación que las hace verdaderas y otra que las hace falsa. Esto implica que no constituyen *verdades lógicas* (fórmulas que son siempre verdaderas).

Por otra parte, el valor de verdad de una fórmula cerrada es independiente de la asignación de valores a variables. Esto es de destacar, ya que en los programas lógicos (teorías de $1^{er.}$ orden) sólo se manejan fórmulas cerradas.

Definición Sea F una fórmula cerrada e I una interpretación para F. I es un *modelo* si F es verdadera según I. También decimos que I *satisface* a F.

Esta definición se extiende directamente para un conjunto de fórmulas cerradas S sobre un lenguaje de $1^{er.}$ orden L:

Definición I es un modelo para S si todas las fórmulas de S son verdaderas según I.

Definición Sea S un conjunto de fórmulas cerradas sobre un lenguaje de primer orden L.

- S es *satisfactible* si existe al menos un modelo para S.
- S es *válido* (lógicamente válido) si toda interpretación es modelo.
- S es *insatisfactible* si no tiene ningún modelo.
- S es *inválido* si existe al menos una interpretación que no es modelo.

De las definiciones se desprende que si un conjunto es válido, debe ser satisfactible; y si es insatisfactible, debe ser inválido. Claramente, no se cumplen los recíprocos de estos postulados.

A las fórmulas válidas se les llama también *tautología*; son los teoremas de la lógica de primer orden.

Suponiendo la satisfactibilidad de nuevas fórmulas se puede deducir que otras son también satisfactibles. Postulando nuevos axiomas podemos deducir nuevas fórmulas. Esto es lo que hacemos al escribir un programa lógico (fórmulas que se satisfacen).

Estamos interesados, justamente, en ver qué otras cosas son verdaderas suponiendo verdaderas las fórmulas postuladas, o sea, en las consecuencias de estas fórmulas. Esta noción es capturada por la siguiente definición:

Definición Sea S un conjunto de fórmulas cerradas y F una fórmula cerrada. F es una *consecuencia lógica* de S (o se *deduce lógicamente* de S) si todo modelo de S es modelo de F. **Notación:** $S \models F$

Teorema 1.1.1 *Sea* $S = \{F_1, \ldots, F_n\}$ *un conjunto de fórmulas cerradas y* F *una fórmula cerrada.* $S \models F \Leftrightarrow (F_1 \wedge \ldots \wedge F_n) \rightarrow F)$ *es una fórmula válida.*

Prueba Se deja como ejercicio (basta ver que si el consecuente es falso, también debe serlo el antecedente).

Teorema 1.1.2 *Sea* $S = \{F_1, \ldots, F_n\}$ *un conjunto de fórmulas cerradas y* F *una fórmula cerrada.* $S \models F \Leftrightarrow S \cup \{\neg F\}$ *es insatisfactible.*

Prueba ($\Leftarrow$) Sea I un modelo de S.

Como $S \cup \{\neg F\}$ es insatisfactible, I no es modelo de $\neg F$. Pero entonces, I es modelo de F. En conclusión, todo modelo de S es modelo de F. Por definición de consecuencia lógica, $S \models F$.

($\Rightarrow$) Sea I una interpretación cualquiera.

Si I no es modelo de S, al menos no satisface una cláusula de S, por lo que tampoco puede ser modelo de $S \cup \{\neg F\}$.

En cambio, si I es un modelo de S, como $S \models F$, I es modelo de F. Pero entonces, I no satisface $\neg F$. De donde, tampoco satisface $S \cup \{\neg F\}$.

En conclusión, cualquiera sea la I elegida, I no satisface $S \cup \{\neg F\} \Rightarrow S \cup \{\neg F\}$ es insatisfactible.

Nosotros sabemos que la negación de una consulta C es una cláusula. Pero entonces, para probar que la consulta es consecuencia lógica de un programa lógico P, basta con probar que el conjunto $P \cup \{\neg C\}$ es insatisfactible. A este

tipo de demostración se le denomina *prueba por refutación*: se niega lo que se quiere probar, llegándose a un absurdo.

En principio, debo demostrar que *ninguna* interpretación I satisface el conjunto de cláusulas. Claramente, no podemos probar una a una las interpretaciones... Debemos buscar formas automáticas de resolver este problema: se introducen, entonces, los métodos de resolución.

1.2. Resolución

Supongamos que sabemos lo siguiente:

1. Pablo estudia prolog o funcional.
2. Pablo no estudia funcional

Claramente, se concluye que, si las premisas son ciertas, Pablo debe estudiar prolog. Este esquema de derivación se llama *resolución*, y puede escribirse como:

$$\{p \vee q, \neg q\} \Rightarrow p$$

Todo modelo del conjunto {p $\vee$ q, $\neg$ q} es también modelo de p (notar que no se cumple el recíproco). Es decir, podemos «quitar» literales de una cláusula aplicando resolución. Si llegamos a la cláusula vacía, podemos afirmar que las clásulas originales no tenían modelo.

Esto es lo que sucede en proposicional, pero ¿qué sucede en lógica de predicados?

$$\{\neg\exists x P(x) \vee q(a),\ q(b) \vee \exists x P(x)\} \Rightarrow q(a) \vee q(b)$$

En su caso más general, la regla se formula de la siguiente forma:

$$\frac{\alpha \vee \beta,\ \gamma \vee \neg\beta}{\alpha \vee \gamma}$$

Esta regla es aplicable a α, β y γ formulas de $1^{er.}$ orden cualesquiera. La regla es *correcta*: si ambas fórmulas son verdaderas en una interpretación I, al menos una de las fórmulas α y γ deben ser verdaderas en I. Se cumple entonces que $\alpha \vee \gamma$ es consecuencia lógica de las premisas.

En nuestro caso, vamos a aplicar la regla de resolución a cláusulas. La razón es que, en este caso, se puede probar resultados de completitud.

Veamos primero el caso de cláusulas proposicionales, para luego generalizar a cláusulas de primer orden.

1.2.1. Resolución para cláusulas proposicionales

Sean:

$$C_1 : L_1 \vee \ldots \vee L_n$$

$$C_2 : M_1 \vee \ldots \vee M_k$$

Si $L_i = \neg M_j$, la resultante es:

$$C_3 : L_1 \vee \ldots \vee L_{i-1} \vee L_{i+1} \vee \ldots \vee L_n \vee M_1 \vee \ldots \vee M_{j-1} \vee M_{j+1} \vee \ldots \vee M_k$$

Teorema 1.2.1 *La resolución es correcta* ($\{C_1, C_2\} \models C_3$)

Prueba Sea V una asignación de verdad a las letras proposicionales que hace verdadero a $\{C_1, C_2\}$. Al menos uno de los literales de C_1 y al menos uno de C_2 deben ser verdaderos según V.

Consideramos dos opciones para C_1:

1. V hace verdadero a un literal distinto de L_i. Entonces la resultante C_3 es verdadera, puesto que contiene a este literal.
2. V hace verdadero al literal L_i. Entonces, V hace falso al literal M_j, puesto que $M_j = \neg L_i$. Como V satisface a C_2 debe hacer verdadero a un literal de esta cláusula M_p. Entonces la resultante C_3 es verdadera, puesto que contiene a este último literal.

Pero entonces, todo modelo de $\{C_1, C_2\}$ es modelo de C_3, con lo que probamos este teorema.

Se deja como ejercicio probar que el recíproco del teorema anterior *no* es cierto, o sea, la resolvente puede tener modelos que no lo sean de las cláusulas premisa.

Ejemplo En la figura 1.2 se observa la deducción de q a partir de:

$$\{\{p, r\}, \{\neg r\}, \{r, \neg p, q\}\}$$

Si nuestro conjunto de cláusulas es un par de literales complementarios, en un paso de resolución llegamos a la cláusula vacía. Por ejemplo, resolviendo $\{p\}$ y $\{\neg p\}$ obtenemos como resultado una cláusula sin literales. La cláusula vacía ($\square$) no tiene modelos, puesto que no tiene literales que puedan hacerse verdaderos en una interpretación. Como la resolución mantiene los modelos (la resolvente es consecuencia lógica de las dos cláusulas que se resuelven), si de un conjunto de cláusulas se obtiene la cláusula vacía por aplicación de uno o más pasos de resolución, podemos deducir que el conjunto de cláusulas no tiene ningún modelo (o sea, es insatisfactible).

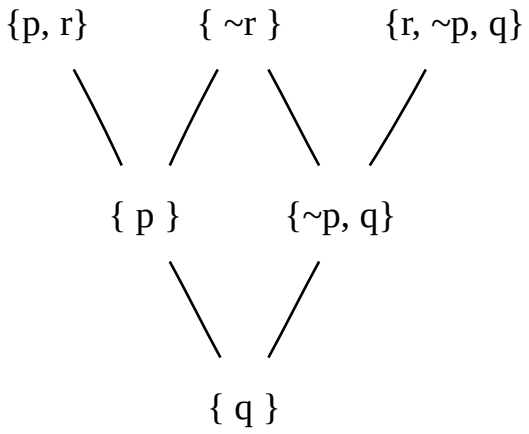

Figura 1.2: Ejemplo de resolución.

1.2.2. Resolución para cláusulas de primer orden

¿Qué sucede en el siguiente caso? Sean las siguientes cláusulas ($L_1 ... L_n, M_1 \dots M_k$ son literales):

$$\forall x(p(x) \vee L_1 \vee L_2 \dots \vee L_n)$$

$$\forall x(\neg p(x) \vee M_1 \vee M_2 \dots \vee M_k)$$

¿Cómo se aplica la resolución?

$$\forall x(L_1 \vee L_2 \dots \vee L_n \vee M_1 \vee M_2 \dots \vee M_k)$$

Para el resto del dominio DEBE valer la disyunción de literales de ambas cláusulas. Sea a un elemento del dominio. Si p(a) es verdadero, ¬ p(a) es falso, con lo cual alguno de los literales de la segunda cláusula deben ser verdadero. En cambio, si p(a) es falso, algun literal de la primera cláusula debe ser verdadero. Pero entonces, cualquier interpretación que sea modelo del conjunto de fórmulas debe ser modelo de la cláusula resultante.

Veamos un ejemplo concreto:

$$\forall x(p(x) \vee r(x))$$

$$\forall y(\neg p(a) \vee q(y))$$

Al aplicar resolución obtengo:

$$\forall y(r(a) \vee q(y))$$

pero también podría obtener, por ejemplo:

$$r(a) \vee q(b)$$

¿Cuál de las dos resoluciones conviene más? En el segundo caso necesito de la negación de q(b) para llegar a la cláusula vacía. La mejor es la primera, puesto que es más fácil buscar algo con lo cual «quitar» q(y) por ser «más general» (podría ser q(b), q(c) o cualquier otro literal de esa forma).

Entonces, se debe definir «formalmente» la sustitución de variables y encontrar un procedimiento que me permita encontrar aquella que sea la más general.

Definición Una *sustitución* θ es un conjunto finito de la forma: $\{v_1/t_1, \ldots, v_n/t_n\}$ donde, para $1 \leq i \leq n$, se cumple:

- v_i es una variable
- t_i es un término distinto de v_i
- $v_i \neq v_j$ si $i \neq j$

La sustitución vacía (no cambio ninguna variable) se denota por ε.

Definición sea E una expresión (un término o una fórmula sin cuantificadores) y $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ una sustitución. La *instancia* de E por θ —Eθ— es el resultado de la aplicación *simultánea* del reemplazo de cada variable v_i por el término t_i en E.

Definición Si W=$\{E_1, \ldots, E_n\}$ es un conjunto de expresiones y θ una sustitución, diremos que Wθ, instancia del conjunto W por θ, es el conjunto $\{E_1\theta, \ldots, E_n\theta\}$

Ejemplo Aplicación de sustituciones:

1. Si E=p(x,y,f(b,x,z)) y θ={x/a, y/x, z/f(z)}, Eθ=p(a,x,f(b,a,f(z)))
2. Si W={p(x,y,f(b,x,z)), p(a,v,f(w,u,v))} y θ={u/a, w/b, x/a, y/v, z/v}, Wθ={p(a,v,f(b,a,v))}

Definición Composición de sustituciones: sea $\theta = \{v_1/s_1, \ldots, v_n/s_n\}$ y $\sigma = \{u_1/t_1, \ldots, u_m/t_m\}$ dos sustituciones. La sustitución $\theta \circ \sigma$, composición de θ con σ, se define como la sustitución:

$$\{v_1/(s_1\sigma), \ldots, v_n/(s_n\sigma), u_1/t_1, \ldots, u_m/t_m\}$$

luego de eliminar los pares $v_i/(s_i\sigma)$ tales que $v_i = (s_i\sigma)$, y los pares u_j/t_j tales que $u_j \in \{v_1, \ldots, v_n\}$.

[Obs.: Se eliminan aquellos pares que impiden que el resultado sea, a su vez, una sustitución.]

Ejemplo θ={x/f(x,y), y/z, w/f(t,a)} y σ={x/a, z/y, t/f(a,y)}

$\theta \circ \sigma$={x/f(a,y), w/f(f(a,y),a), z/y, t/f(a,y)}

Proposición 1.2.2 *Se cumple que:*

- $\theta \circ \varepsilon = \varepsilon \circ \theta = \theta$
- *(Eθ)σ = E($\theta \circ \sigma$)*
- $(\gamma \circ \theta) \circ \sigma = \gamma \circ (\theta \circ \sigma)$

Prueba Se deja como ejercicio.

Definición Las expresiones E_1 y E_2 son *variantes* si existen θ y σ sustituciones tales que ($E_1\theta$)=E_2 y E_1=($E_2\sigma$). En otras palabras, son idénticas a menos de nombres de variables.

Ejemplo ¿Son variantes las siguientes expresiones?

- f(x,a,y) es una variante de f(y,a,z); θ={x/y, y/z} y σ={y/x, z/y}
- f(x,y) no es una variante de f(z,z); θ={x/z, y/z} pero no puedo encontrar σ

Definición Sea E una expresión y V el conjunto de variables que ocurren en E. Una sustitución es un *renombre de variables* para E si es de la forma: $\{v_1/w_1, \ldots, v_n/w_n\}$, donde:

- $\{v_1, \ldots, v_n\} \subseteq V$
- $w_i \neq w_j$ si $i \neq j$
- $(V - \{v_1, \ldots, v_n\}) \cap \{w_1, \ldots, w_n\} = \emptyset$

Un renombre de variables genera variantes de expresiones.

Definición Una sustiución θ *unifica* un conjunto de expresiones W sii (Wθ) es un conjunto con un único elemento.

Ejemplo Sea W={ p(x,y,z), p(a,f(x),z) }. Las siguientes sustituciones son unificadores:

- θ = { x/a, y/f(a), z/b } ; (Wθ)={p(a,f(a),b)}
- σ = { x/a, y/f(a) } ; (Wσ)={p(a,f(a),z)}

Definición Una sustiución θ es el *unificador más general* (m.g.u.) de un conjunto de expresiones W si:

- θ es un unificador de W.

- Para cualquier unificador σ de W, $\exists$ una sustitución γ tal que $\sigma = \theta \circ \gamma$

Ejemplo ¿Cuáles son mgu de las siguientes expresiones?

- {p(f(a,x),g(y),y), p(z,w,a)} tiene como mgu θ={y/a, w/g(a), z/f(a,x)}
- {p(x), p(f(x))} no tiene unificador

Ejemplo El mgu no siempre es único: {p(x), p(y)} admite como mgu a {x/y} y a {y/x}.

Algoritmo de unificación: existe un algoritmo que determina el mgu de un conjunto de expresiones o reporta falla en caso de que el conjunto no sea unificable. Su pseudocódigo se presenta en el cuadro 1.1.

La idea es recorrer las expresiones de izquierda a derecha hasta encontrar un punto en el que sean distintas. Se toman todas las subexpresiones que comienzan en esa posición y se intenta unificarlas, sustituyendo una variable por el término que corresponda.

Este proceso se repite hasta que no queden más variables o hasta que en un paso queden expresiones imposibles de unificar. Este código no puede entrar en un ciclo infinito porque en cada iteración o bien reporta «no unificación» o bien quita una variable (y la cantidad de variables de una expresión es finita).

Definición El *conjunto de diferencias* de un conjunto de expresiones W esta compuesto por las subexpresiones que se extraen de la primera posición en la cual las expresiones de W difieren en sus símbolos.

Teorema 1.2.3 *Sea W un conjunto de expresiones. Si W es unificable, el algoritmo termina y da como resultado un mgu de W. En caso contrario, el algoritmo reporta que W no es unificable.*

Prueba Ver página 25 del Lloyd[2]

Aunque el algoritmo anterior es aplicable a cualquier número de expresiones, en los procedimientos de refutación siempre se unifican sólo dos. El algoritmo del cuadro 1.2, extraído de Sterling y Shapiro[3], utiliza un stack para realizar la unificación de dos expresiones.

Ahora podemos volver a nuestro objetivo: aplicar resolución en cláusulas de primer orden.

$$C_1 : \{p(x), q(x)\}$$

$$C_2 : \{\neg p(a), s(y)\}$$

Si aplicamos la sustitución $\{x/a\}$ obtenemos un par de literales complementarios y estamos en condiciones de aplicar resolución:

$$C_3 : \{q(a), s(y)\}$$

- Entrada: conjunto de expresiones W
- Salida: θ, mgu de las expresiones, o *no unifican* si W no es unificable.
- $\theta_0 = \epsilon$
- Mientras Wθ_k no sea un conjunto «unitario»

  Calcular el conjunto de diferencias D_k de Wθ_k

  Si en D_k hay una variable v y un término t tal que v no ocurre en t

  $\theta_{k+1} = \theta_k \circ \{v/t\}$

  en caso contrario

  retornar *no unifican*
- Retornar θ_k

Cuadro 1.1: Algoritmo de unificación de un conjunto W de expresiones

- Entrada: $T_1 = T_2$
- Salida: θ, mgu de las expresiones, o *no unifican* si T_1 y T_2 no son unificables.
- PUSH $T_1 = T_2$
- $\theta = \epsilon$
- Mientras el stack no se vacíe

  POP X=Y

  En caso de que

  1. X es una variable que no ocurre en Y:
     $\theta = \theta \circ \{X/Y\}$;
     sustituir Y por X en el stack, o sea, stack = stack $\{X/Y\}$.
  2. Y es una variable que no ocurre en X:
     $\theta = \theta \circ \{Y/X\}$; stack = stack $\{Y/X\}$.
  3. X e Y son el mismo término: skip
  4. X es $f(X_1, \ldots, X_n)$ e Y es $f(Y_1, \ldots, Y_n)$:
     PUSH $X_i = Y_i, i = 1 \ldots n$.
  5. en otro caso: retornar *no unifican*.
- Retornar θ

Cuadro 1.2: Algoritmo de unificación de dos expresiones

Definición Sean C_1 y C_2 dos cláusulas de primer orden, sin variables en común:

$$C_1 : \{L_1, \ldots, L_n\}$$

$$C_2 : \{M_1, \ldots, M_k\}$$

y θ un mgu de L_i y $\neg M_j$
Definimos como *resolvente binaria* a la siguiente cláusula C_3:

$$C_3 : \{L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n, M_1, \ldots, M_{i-1}, M_{j+1}, \ldots M_k\}\theta$$

Observaciones:

- Utilizamos el mgu para obtener la resolvente «más general» posible.
- Exigimos que no haya variables en común para que no suceda que $p(x) \vee q(x)$ sea la resolvente de $p(x) \vee r(y)$ con $q(x) \vee \neg r(x)$. Esto último se arregla con un cambio de variables.

Teorema 1.2.4 [Corrección] *Sean C_1 y C_2 dos cláusulas de primer orden y R una resolvente de ambas. Se cumple que $\{C_1, C_2\} \models R$*

Prueba El esquema de la demostración es el siguiente: se debe utilizar que $C \models (C\theta)$ (ej. de práctico 2), y considerar que cualquier modelo de las clásulas también lo es de las sustituidas. Luego, el razonamiento es análogo a la prueba proposicional.

Completitud: Nos gustaría que aplicando resolución en las cláusulas de S se llega a C ($S \vdash_R C$) siempre que $S \models C$. Sin embargo, esto no se cumple; para probarlo, basta con ver el siguiente contraejemplo:

$$S = \{p(x)\} \quad C = p(f(y))$$

No podemos derivar C a partir de S porque no podemos «inventar» el functor f. Sin embargo, aplicando *refutación* se prueba lo que queremos: todos los elementos que «precisamos» están presentes en las propias fórmulas.

Se puede probar que un conjunto de cláusulas es insatisfactible si y sólo si existe una derivación de la cláusula vacía a partir de S aplicando resolución.

1.3. Procedimientos automáticos de resolución

Contando con la regla de resolución, queda por definir cuál es la estrategia con que se aplicará esta regla para obtener la cláusula vacía. A continuación se presentan algoritmos, extraídos del Chang y Lee[1].

1.3.1. Saturación por niveles

Supongamos que deseamos probar que un conjunto S de cláusulas proposicionales es insatisfactible. Un primer procedimiento es generar todas las resolventes posibles entre todo par de cláusulas de nuestro conjunto, y luego utilizar éstas como entrada para nuevas resoluciones.

$$S^0 = S$$
$$S^{i+1} = \{resolventes\, de\, C_1 \in (S^0 \cup \ldots \cup S^i)\, y\, C_2 \in S^i\}$$

Este proceso continúa hasta que en algún momento se llegue a la cláusula vacía.

Ejemplo

$$S = \{p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q\}$$

$$S^0 = \{p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q\}$$

$$S^1 = \{q, p, \neg q \vee q, \neg p \vee p, \neg q \vee q, \neg p \vee p, \neg p, \neg q\}$$

$$S^2 = \{p \vee q, \ldots, \square\}$$

Repitiendo el procedimiento, se llega a la cláusula vacía en la 35^{va} resolución.

La saturación de niveles es muy ineficiente. Para empezar, en el ejemplo anterior se generan tautologías que podrían haberse detectado (y eliminando, dado que seguro no llegamos a la cláusula vacía a partir de ellas). Análogamente, se pueden eliminar las cláusulas subsumidas (ver práctico).

Se proponen *refinamientos* para evitar resolver todas las cláusulas contra todas las otras. El problema es si los refinamientos mantienen o no la completitud de los procedimientos de refutación.

1.3.2. Resolución lineal

Consiste en considerar siempre como una de las cláusulas de entrada a la resolvente del paso anterior. Esto hace que el árbol tome la forma de una línea (y de ahí el nombre), en donde las resolventes se llaman cláusulas centrales, y las que entran en cada paso de resolución, laterales.

En la figura 1.3 se logra derivar p utilizando resolución lineal a partir de las cláusulas:

$$\{\{p, \neg q, r\}, \{q, s\}, \{\neg s\}, \{\neg r\}\}$$

Definición Sea S un conjunto de cláusulas y C una cláusula. Una *derivación lineal* de C a partir de S es una secuencia finita de cláusulas $\mathrm{R}^0 \ldots \mathrm{R}^k$ en donde $\mathrm{R}^0 \in$S, R^k=C y R^{i+1} es la resolvende de R^i con alguna cláusula de S $\cup$ $\{\mathrm{R}^0 \ldots R^{i-1}\}$

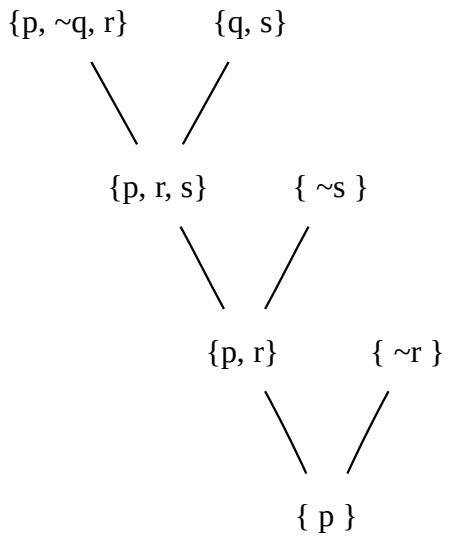

Figura 1.3: Ejemplo de resolución lineal.

Se demuestra que la resolución lineal es completa respecto a la refutación (ver Chang & Lee).

1.3.3. Resolución con cláusulas de entrada

La resolución con cláusulas de entrada es una resolución lineal en donde se restinge las cláusulas laterales a pertener únicamente a S; en otras palabras, no se puede utilizar resultados de resoluciones previas a la inmediatamente anterior en la cadena.

A diferencia de la resolución lineal, este método no es completo respecto a la refutación: para llegar a la cláusula vacía hay que resolver en el último paso dos cláusulas con un literal. Sin embargo, en:

$$S = \{p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q\}$$

todas las clásulas tienen dos literales, lo que hace que cualquier resolvente siempre tenga al menos un literal.

En cambio, la resolución con cláusulas de entrada cuando éstas son definidas *sí* es completa. Utilizaremos una variante llamada resolución SLD (selection linear definite) para programas definidos.

Buscaremos una refutación para $P \cup \{\neg C\}$ utilizando resolución SLD. Tomaremos como cláusula incial el objetivo, o sea, una cláusula con todos los literales negativos. El único modo de aplicar resolución es utilizando las cláusulas del programa, ya que sólo éstas tienen un (único) literal positivo.

Definición Sea P un programa definido y G un objetivo definido $\leftarrow A_1 \ldots A_n$. G' es la *resolvente SLD* de G y una cláusula $A \leftarrow B_1 \ldots B_n$ de P con mgu θ si:

- A_m es un átomo de G (átomo seleccionado)
- θ es un mgu de A_m y A
- G' es el objetivo $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_n, A_{m+1}, \ldots, A_n)\theta$

Definición Sea P un programa definido y G un objetivo definido $\leftarrow A_1 \ldots A_n$. Una *derivación SLD* de $P \cup \{G\}$ es una secuencia (finita o infinita) de objetivos de la forma $G \equiv G_0, G_1, \ldots, G_n, \ldots$; una secuencia de variantes de cláusulas de P: $C_0, C_1, \ldots, C_n, \ldots$; y una secuencia de mgus $\theta_0, \theta_1, \ldots, \theta_n, \ldots$ tales que G_{i+1} se deriva de G_i y C_i con mgu θ_i.

Se utilizan variantes de las cláusulas de los programas para evitar el problema de que variables de distintas cláusulas queden «enganchadas» por tener el mismo nombre. Por ejemplo, ¿qué sucedería con p(f(X)) y p(X) si no lo hiciéramos?

Definición Una *refutación SLD para* $P \cup \{G\}$ es una derivación SLD finita para $P \cup \{G\}$ cuyo último objetivo es $\square$.

Ejemplo ¿Qué sucede con los siguientes objetivos?
$\leftarrow$ camino(a,d).
$\leftarrow$ camino(a,X).
$\leftarrow$ camino(a,Z), camino(Z,e).

Ejemplo ¿Qué sucede con el siguiente programa cuando se tiene el objetivo $\leftarrow$ r(Y,X)?

r(X,Y)$\leftarrow$ r(Y,X).

Definición Una *respuesta computada para* $P \cup \{G\}$ es la sustitución obtenida de la composición de mgu utilizados en una refutación SLD de $P \cup \{G\}$ restringiéndola a las variables de G.

Definición Sea P un programa lógico y $G \equiv \leftarrow A_1 \ldots A_k$ un objetivo definido. Una *respuesta correcta para* $P \cup \{G\}$ es una respuesta θ que cumple:

$$P \models \forall((A_1 \wedge \ldots \wedge A_k)\theta)$$

La refutación SLD es completa y correcta, dado que es un caso particular de resolución con cláusulas definidas. Además, se puede probar la completitud y corrección respecto a las respuestas computadas:

- *Corrección*: toda respuesta computada es correcta. (dem. por IC en el largo de la refutación)
- *Completitud*: Si θ es una respuesta correcta, existe una respuesta computada σ y una sustitución γ tales que $\theta = \sigma \circ \gamma$.

Definición Sea P un programa definido y G un objetivo definido. Un *árbol SLD para* $P \cup \{G\}$ es un árbol que satisface:

- Todo nodo es un objetivo (eventualmente vacío).
- La raíz del árbol es G.
- Sea $\leftarrow A_1 \ldots A_k$ (con $k > 0$) un nodo del árbol y A_m el átomo seleccionado. Entonces, para toda cláusula $A \leftarrow B_1 \ldots B_n$ de P tal que A unifique con A_m con mgu θ, hay un hijo cuyo objetivo es la resolvente SLD de esas dos cláusulas: $\leftarrow(A_1, \ldots, A_{m-1}, B_1, \ldots, B_n, A_{m+1}, \ldots, A_k)\theta$
- Los nodos con objetivos vacíos no tienen hijos.

Cada rama del árbol SLD se corresponde con una posible derivación SLD; de esta forma se representan a todas las derivaciones posibles a partir de un objetivo G utilizando la regla de computación seleccionada.

A las ramas que se corresponden con refutaciones —la hoja es □— se les llama *ramas exitosas* o de éxito; a las que se corresponden con derivaciones infintas se les llama *ramas infinitas*; y a las que lo hacen con derivaciones fallidas se les llama *ramas de falla*.

La regla de computación determina las ramas del árbol SLD: para un mismo programa lógico y objetivo, el árbol SLD posiblemente cambie al cambiar la regla de computación.

Ejemplo Sea el siguiente programa lógico:

p(X,Z) ← q(X,Y), p(Y,Z)
p(X,X)
q(a,b)

y el siguiente objetivo: ← p(X,b)

El árbol SLD de la figura 1.4 se obtiene al aplicar la regla de computación que selecciona el átomo de más a la izquierda. Las siguientes son las resoluciones hechas:

1. $C_1 : p(X_1, Z_1) \leftarrow q(X_1, Y_1), p(Y_1, Z_1)$
   $\theta_1 = \{X_1/X, Z_1/b\}$

2. $C_2 : q(a, b)$
   $\theta_2 = \{X/a, Y_1/b\}$

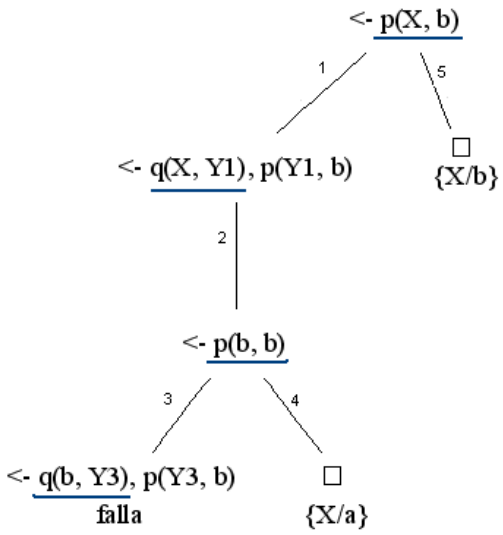

Figura 1.4: Resolución SLD eligiendo el átomo de más a la izquierda.

3. $C_3 : p(X_3, Z_3) \leftarrow q(X_3, Y_3), p(Y_3, Z_3)$
   $\theta_3 = \{X_3/b, Z_3/b\}$

4. $C_4 : p(X_4, X_4)$
   $\theta_3 = \{X_4/b\}$

5. $C_5 : p(X_5, X_5)$
   $\theta_3 = \{X_5/b, X/b\}$

En cambio, al aplicar la regla de computación que selecciona el átomo de más a la derecha, se obtiene el árbol de la figura 1.5. En este caso las resoluciones son:

1. $C_1 : p(X_1, Z_1) \leftarrow q(X_1, Y_1), p(Y_1, Z_1)$
   $\theta_1 = \{X_1/X, Z_1/b\}$

2. $C_2 : p(X_2, Z_2) \leftarrow q(X_2, Y_2), p(Y_2, Z_2)$
   $\theta_2 = \{X_2/Y_1, Z_2/b\}$

3. $C_3 : p(X_3, X_3)$
   $\theta_3 = \{X_3/b, Y_1/b\}$

4. $C_4 : q(a, b)$
   $\theta_4 = \{X/a\}$

5. $C_5 : p(X_5, X_5)$
   $\theta_5 = \{X_5/b, X/b\}$

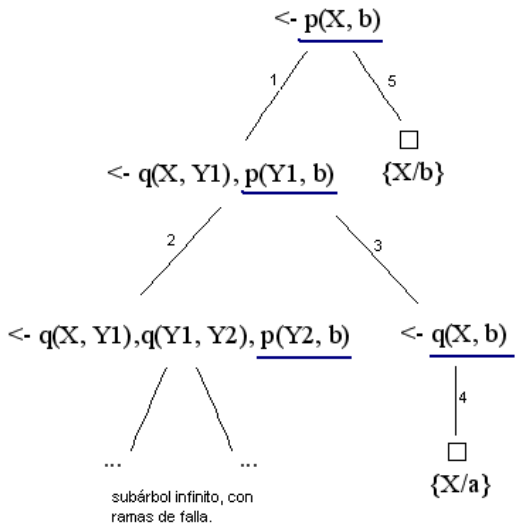

Figura 1.5: Resolución SLD eligiendo el átomo de más a la derecha.

Proposición 1.3.1 Independencia de la regla de computación*: si existe una refutación SLD para* $(P \cup \{G\})$ *con respuesta computada* θ*, para cualquier regla de computación* R *existe una refutación SLD con respuesta computada* σ *tal que* $G\sigma$ *es una variante de* $G\theta$*. (ver Lloyd[2])*

El resultado anterior me garantiza que no «pierdo» soluciones al fijar una regla de computación. Ahora bien, dada una regla de computación, cada objetivo tiene asociado un árbol SLD, el cual puedo recorrer en busca de las soluciones de distintas formas.

Definición Una regla de computación junto a una estrategia de búsqueda en un árbol SLD definen un *procedimiento de refutación SLD*.

Existen dos estrategias clásicas: búsqueda en amplitud (BFS) y búsqueda en profundidad (DFS). En el caso de que el árbol sea finito, ambas estrategias llegan a todos los nodos del árbol (posiblemente en distinto orden). Sin embargo, en el caso de un ábol infinito, mientras BFS sigue llegando a cualquier nodo del árbol en una cantidad finita de pasos, DFS no lo hace («cae» en ramas infinitas). A pesar que BFS parece ser la estrategia a elegir, se opta por DFS, que es más eficiente.

Un intérprete Prolog estándar utiliza el siguiente procedimiento de refutación SLD:

- regla de computación: elegir el átomo de más a la izquierda.
- estrategia de búsqueda: en profundidad, eligiendo las ramas según el orden de aparición en el programa de las cláusulas correspondientes.

Esto hace que Prolog sea incompleto: existen objetivos para los cuales hay alguna refutación SLD que el intérprete es incapaz de encontrar.

Bibliografía

[1] Chang y Lee. "Symbolic Logic and Mechanical Theorem Proving". Academic Press, US (1973).

[2] J.W. Lloyd. "Foundations of Logic Programming". Springer-Verlag, Berlin, segunda edición (1987).

[3] Leon Sterling y Ehud Shapiro. "The Art of Prolog". The MIT Press, Cambridge, US, segunda edición (1994).