Fundamentos de programación lógica e Introducción a Prolog

Nota complementaria para el curso de Inteligencia Artificial

Proyecto UNAM-DGAPA PAPIME 102723

Lourdes Del Carmen González Huesca*

Última versión: 31 de marzo de 2024

Resumen

En esta nota se revisarán los fundamentos de programación lógica y el lenguaje más representativo de este estilo de programación: Prolog [1]. Se da un breve resumen de nociones preliminares como unificación y resolución binaria para después pasar a la programación lógica. Finalmente se incluye parte de la teoría o modelos de Herbrand para comprender este método para verificar la validez de fórmulas en Lógica de Primer Orden o de Predicados.

1. Preliminares

1.1. Breve introducción a la programación lógica: características e historia

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

- Un programa es una secuencia de instrucciones.
- Las principales estructuras de control son los ciclos: `while, repeat, for`, etc.
- Estas estructuras permiten seguir paso a paso las acciones que debe realizar un programa.
- Hay un manejo explícito de la memoria.
- La operación de asignación `x:=a` es imprescindible.

Estas características hacen que el programa especifique *cómo* se calculan los resultados.

En contraste, los lenguajes de programación funcional como Haskell, Lisp, Scheme, ML y lógica como 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.

*Material realizado con las notas de curso de Lógica Computacional utilizados durante 2017 a 2023 a cargo de Favio Ezequiel Miranda Perea, Araceli Liliana Reyes Cabello, Lourdes Del Carmen González Huesca y Pilar Selene Linares Arévalo.

[1] `https://www.swi-prolog.org/`

La programación declarativa es de mucho más alto nivel ya que no tiene manejo de memoria explícito y no se enfoca en un algoritmo que describe paso a paso las instrucciones a ejecutarse. Debido a lo anterior podemos decir que los programas declarativos son elegantes matemáticamente. Si bien los programas imperativos pueden ser rápidos y especializados, un programa declarativo es más general, corto y legible. De esto podemos decir que es más fácil *verificar* si un programa cumple una especificación o simplemente verificar las propiedades deseables del programa. Aprender programación declarativa permite a cualquier programador(a) a desarrollar una forma de programación abstracta, rigurosa y disciplinada que puede ser usada ventajosamente sin importar el lenguaje de programación utilizado. Este estilo genera programas con una mejor ingenieria, más fáciles de depurar, mantener y modificar.

Fue alrededor de las décadas de 1920 y 1930 que Jacques Herbrand [2] propuso en su tesis un método para reducir la Lógica de Primer Orden a la Lógica de Proposiciones. En este método utiliza 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 de discurso es el universo formado por términos que representan objetos
2. las fórmulas que involucran predicados describen un problema

En el estilo de programación lógica, las fórmulas son usadas para expresar conocimiento, en particular, son descripciones o algoritmos (recursivos) en forma de funciones parciales. Estas descripciones crean un programa lógico como un conjunto de cláusulas.

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 respuestas están 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 "$\mathcal{C}$ se sigue lógicamente de $\mathbb{P}$", $\mathbb{P} \models \mathcal{C}$. En particular, un predicado no se ejecuta ni devuelve un valor como resultado sino que se relacionan los argumentos de la función junto con el resultado: en lugar de "programar una función" de n argumentos, se "programa un predicado" de $n + 1$ argumentos.

Los fundamentos del estilo 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, basado en el Teorema de Herbrand, que sirvió para la implementación de Prolog. El algoritmo que se utiliza en esta nota es el de Martelli y Montanari.

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.

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

1.2. Lógica de Predicados

En contraposición al lenguaje de la lógica proposicional que está determinado de manera única, no es posible hablar de un solo lenguaje para la lógica de predicados. Dependiendo de la estructura semántica que tengamos en mente será necesario agregar símbolos particulares para denotar objetos y relaciones entre objetos. De esta manera el alfabeto de esta lógica o lenguaje consta de dos partes ajenas entre sí: la parte común a todos los lenguajes está determinada por los símbolos lógicos y auxiliares y la parte particular, llamada tipo de semejanza o signatura del lenguaje que determina los objetos y sus relaciones.

La parte común a todos los lenguajes consta de:

- Un conjunto infinito de variables $\mathsf{Var} = \{x_1, \ldots, x_n, \ldots\}$
- Constantes lógicas: $\bot, \top$
- Conectivos u operadores lógicos: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- Cuantificadores: $\forall, \exists$.
- Símbolos auxiliares: (,) y , (coma).
- Si se agrega el símbolo de igualdad $=$, decimos que el lenguaje tiene o maneja la igualdad.

La signatura de un lenguaje en particular está dada por:

- Un conjunto $\mathcal{P}$, posiblemente vacío, de símbolos o letras de predicado para representar relaciones: $P_1, \ldots, P_n, \ldots$
  A cada símbolo se le asigna un índice [3] o número de argumentos m, el cual se hace explícito escribiendo $P_n^{(m)}$ que significa que el símbolo P_n requiere m argumentos.
- Un conjunto $\mathcal{F}$, posiblemente vacío, de símbolos o letras de función para representar transformaciones de objetos: $f_1, \ldots, f_n, \ldots$
  Análogamente a los símbolos de predicado cada símbolo de función tiene un índice asignado, $f_n^{(m)}$ significará que el símbolo f_n necesita de m argumentos.
- Un conjunto $\mathcal{C}$, posiblemente vacío, de símbolos de constantes para representar objetos particulares, bien definidos: $c_1, \ldots, c_n, \ldots$
  En algunos libros los símbolos de constante se consideran como parte del conjunto de símbolos de función, puesto que pueden verse como funciones de índice cero, es decir, funciones sin argumentos.

Dado que un lenguaje de primer orden $\mathcal{L}$ queda determinado de manera única por su signatura, abusaremos de la notación y escribiremos $\mathcal{L} = \mathcal{P} \cup \mathcal{F} \cup \mathcal{C}$ para denotar al lenguaje dado por tal signatura.

Términos

Los términos del lenguaje son aquellas expresiones que representarán objetos en la semántica y su definición es:

- Los símbolos de constantes $c_1, \ldots, c_n, \ldots$ son términos.
- Las variables $x_1, \ldots, x_n, \ldots$ son términos.
- Si $f^{(m)}$ es un símbolo de función y $t_1, \ldots, t_m$ son términos entonces $f(t_1, \ldots, t_m)$ es un término.
- Son todos.

[3] Este índice suele llamarse también "aridad", pero dado que tal palabra no existe en el diccionario de la lengua española, trataremos de evitar su uso.

Es decir, los términos están dados por la siguiente gramática:

$$t ::= x \mid c \mid f(t_1, \ldots, t_m)$$

Y el conjunto de términos de un lenguaje dado se denota con $\mathsf{TERM}_{\mathcal{L}}$, o simplemente TERM si es claro cuál es el lenguaje.

Fórmulas

Las expresiones o fórmulas atómicas están dadas por:

- Las constantes lógicas $\bot, \top$.
- Las expresiones de la forma: $P_1(t_1, \ldots, t_n)$ donde $t_1, \ldots, t_n$ son términos
- Las expresiones de la forma $t_1 = t_2$, si el lenguaje cuenta con igualdad

El conjunto de expresiones atómicas se denotará con $\mathsf{ATOM}_{\mathcal{L}}$.

Así el conjunto $\mathsf{FORM}_{\mathcal{L}}$ de expresiones compuestas (bien formadas) en un lenguaje $\mathcal{L}$, llamadas usualmente fórmulas, se define recursivamente como sigue:

- Si $\varphi \in \mathsf{ATOM}_{\mathcal{L}}$ entonces $\varphi \in \mathsf{FORM}_{\mathcal{L}}$. Es decir, toda fórmula atómica es una fórmula.
- Si $\varphi \in \mathsf{FORM}_{\mathcal{L}}$ entonces $(\neg\varphi) \in \mathsf{FORM}_{\mathcal{L}}$.
- Si $\varphi, \psi \in \mathsf{FORM}_{\mathcal{L}}$ entonces $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \to \psi), (\varphi \leftrightarrow \psi) \in \mathsf{FORM}_{\mathcal{L}}$.
- Si $\varphi \in \mathsf{FORM}_{\mathcal{L}}$ y $x \in \mathsf{Var}$ entonces $(\forall x\varphi), (\exists x\varphi) \in \mathsf{FORM}_{\mathcal{L}}$ [4].
- Son todas.

La gramática para las fórmulas en forma de Backus-Naur es:

$$\begin{array}{rcl} \varphi & ::= & at \mid (\neg\varphi) \mid (\varphi \wedge \psi) \mid (\varphi \vee \psi) \mid (\varphi \to \psi) \mid (\varphi \leftrightarrow \psi) \mid (\forall x\varphi) \mid (\exists x\varphi) \\ at & ::= & \bot \mid \top \mid P(t_1, \ldots, t_m) \mid t_1 = t_2 \end{array}$$

Cada lenguaje definido a partir de una signatura y conforme a la gramática recién descrita será un *lenguaje de primer orden.*

El lenguaje de la Lógica de Predicados tiene ciertas convenciones y propiedades que no se estudian en esta nota. Para más detalles se pueden consultar [5, 7].

1.3. Unificación

El proceso de unificación consiste en encontrar, dado un conjunto de literales o términos W, una sustitución σ de tal forma que el conjunto resultante $W\sigma$ conste de un solo elemento.
En adición a sus aplicaciones en programación lógica, la unificación también es importante para los sistemas de reescritura de términos, el razonamiento automátizado y los sistemas de tipos. A continuación estudiamos el caso general así como un algoritmo de unificación junto con las definiciones de las operaciones involucradas.

Definición 1. *Una sustitución en un lenguaje de predicados $\mathcal{L}$ es una tupla de variables y términos denotada como $[x_1, x_2, \ldots, x_n := t_1, \ldots, t_n]$ donde*

- *$x_1, \ldots, x_n$ son variables distintas.*

[4] En ocasiones, estas fórmulas tienen otra sintaxis: $\forall x.\varphi$, $\exists x.\varphi$ donde el punto indica que la fórmula a la derecha está bajo la dependencia del cuantificador correspondiente.

- $t_1, \ldots, t_n$ *son términos de* $\mathcal{L}$.
- $x_i \neq t_i$ *para cada* $1 \leqslant i \leqslant n$

Por lo general denotaremos a una sustitución con $[\vec{x} := \vec{t}]$.

La aplicación de una sustitución $[\vec{x} := \vec{t}]$ a un término r, denotada $r[\vec{x} := \vec{t}]$, se define como el término obtenido al reemplazar **simultáneamente** todas las presencias de x_i en r por t_i. Este proceso de define recursivamente como sigue:

$$
\begin{aligned}
x_i[\vec{x} := \vec{t}] &= t_i \qquad 1 \leqslant i \leqslant n \\
z[\vec{x} := \vec{t}] &= z \qquad \text{si } z \neq x_i \; 1 \leqslant i \leqslant n \\
c[\vec{x} := \vec{t}] &= c \qquad \text{si } c \in \mathcal{C}, \text{ es decir, } c \text{ constante} \\
f^m(t_1, \ldots, t_m)[\vec{x} := \vec{t}] &= f^m(t_1[\vec{x} := \vec{t}], \ldots, t_m[\vec{x} := \vec{t}]) \qquad \text{con } f^{(m)} \in \mathcal{F}.
\end{aligned}
$$

Obsérvese entonces que la aplicación de una sustitución a un término es simplemente una sustitución textual tal y como sucede en lógica de proposiciones, más importante aun, podemos ver que esta función es parcial cuando se aplica a una fórmula de primer orden. Para convertirla en una función total se debe considerar la α-equivalencia al convenir en identificar fórmulas que sólo difieren en sus variables ligadas. De esta manera, al aplicar una sustitución no se verá modificado el significado original de la fórmula.

Definición 2. *Decimos que dos fórmulas* φ_1, φ_2 *son* α*-equivalentes lo cual escribimos* $\varphi_1 \sim_\alpha \varphi_2$ *si y sólo si* φ_1 *y* φ_2 *difieren a lo más en los nombres de sus variables ligadas.*

Las fórmulas α-equivalentes también son lógicamente equivalentes y por lo tanto son intercambiables en cualquier contexto o bajo cualquier operación.

Definición 3. *La aplicación de una sustitución* $[\vec{x} := \vec{t}]$ *a una fórmula* φ*, denotada* $\varphi[\vec{x} := \vec{t}]$ *se define como la fórmula obtenida al reemplazar simultáneamente todas las presencias libres de* x_i *en* φ *por* t_i*, verificando que este proceso no capture posiciones de variables libres.*

La aplicación de una sustitución a una fórmula $\varphi[\vec{x} := \vec{t}]$ se define **recursivamente** como sigue

$$
\begin{aligned}
\bot[\vec{x} := \vec{t}] &= \bot \\
\top[\vec{x} := \vec{t}] &= \top \\
P(t_1, \ldots, t_m)[\vec{x} := \vec{t}] &= P\big(t_1[\vec{x} := \vec{t}], \ldots, t_m[\vec{x} := \vec{t}]\big) \\
(t_1 = t_2)[\vec{x} := \vec{t}] &= t_1[\vec{x} := \vec{t}] = t_2[\vec{x} := \vec{t}] \\
\\
(\neg\varphi)[\vec{x} := \vec{t}] &= \neg\big(\varphi[\vec{x} := \vec{t}]\big) \\
(\varphi \wedge \psi)[\vec{x} := \vec{t}] &= \big(\varphi[\vec{x} := \vec{t}] \wedge \psi[\vec{x} := \vec{t}]\big) \\
(\varphi \vee \psi)[\vec{x} := \vec{t}] &= \big(\varphi[\vec{x} := \vec{t}] \vee \psi[\vec{x} := \vec{t}]\big) \\
(\varphi \to \psi)[\vec{x} := \vec{t}] &= \big(\varphi[\vec{x} := \vec{t}] \to \psi[\vec{x} := \vec{t}]\big) \\
(\varphi \leftrightarrow \psi)[\vec{x} := \vec{t}] &= \big(\varphi[\vec{x} := \vec{t}] \leftrightarrow \psi[\vec{x} := \vec{t}]\big) \\
\\
(\forall y\varphi)[\vec{x} := \vec{t}] &= \forall y\big(\varphi[\vec{x} := \vec{t}]\big) \;\; \textbf{si } y \notin \vec{x} \cup Var(\vec{t}) \\
(\exists y\varphi)[\vec{x} := \vec{t}] &= \exists y\big(\varphi[\vec{x} := \vec{t}]\big) \;\; \textbf{si } y \notin \vec{x} \cup Var(\vec{t})
\end{aligned}
$$

Definición 4. *Sea* W *un conjunto no vacío de términos. Un unificador de* W *es una sustitución* σ *tal que* $|W\sigma| = 1$*, es decir tal que el conjunto* $W\sigma$ *resultante de aplicar* σ *a todos los elementos de* W *consta de un mismo elemento. Si* W *tiene un unificador decimos que* W *es unificable.*

Ejemplo 1.1. Sea $W = \{g(x, f(y)), g(x, f(x)), g(u, v)\}$ un conjunto de términos, entonces la sustitución

$$\sigma = [x := a,\ y := a,\ u := a,\ v := f(a)]$$

es un unificador de W ya que $W\sigma = \{g(a, f(a))\}$

Un conjunto de términos puede tener una infinidad de unificadores o ninguno. Dado un conjunto finito de fórmulas W, es decidible mediante un algoritmo si W es unificable o no; si W es unificable el algoritmo proporciona un unificador llamado **unificador más general**.

Definición 5. *Un unificador σ de un conjunto de términos W, se llama unificador más general (**umg**) si para cada unificador τ de W, existe una sustitución ϑ, tal que $\sigma\vartheta = \tau$.*

La importancia de los unificadores más generales es que nos permiten representar de manera finita un número infinito de sustituciones.

De aquí en adelante omitiremos los paréntesis en funciones y predicados dado es claro el número de argumentos de cada uno de ellos a partir de la signatura dada.

Ejemplo 1.2. Sean $W = \{fgaxgyb, fzguv\}$ con $f^{(2)},\ g^{(2)}$ y

$$\tau = [x := a, z := gaa, y := u, v := b] \qquad \sigma = [z := gax, y := u, v := b].$$

Entonces τ y σ son unificadores de W y es fácil ver que σ resulta ser el **umg** y en particular si $\vartheta = [x := a]$ entonces $\sigma\vartheta = \tau$.

Antes de discutir un algoritmo de unificación es conveniente hacer un análisis intuitivo del problema. Como punto de partida, basta analizar un conjunto de dos términos digamos $W = \{t_1, t_2\}$, y ver si el conjunto W es unificable. Hay que analizar varios casos:

1. Los términos t_1 y t_2 son constantes.
   En este caso $t_i\sigma = t_i$ para cualquier sustitución σ, de manera que W será unificable si y sólo si $t_1 = t_2$.

2. Alguno de los términos es una variable.
   Supongamos que $t_1 = x$, entonces si x figura en t_2 entonces W no es unificable, en caso contrario $\sigma = [x := t_2]$ unifica a W.

3. t_1 y t_2 son términos funcionales.
   En este caso W es unificable si y sólo si se cumplen las siguientes dos condiciones:

   a) Los símbolos principales (es decir los primeros de izquierda a derecha), de t_1 y t_2 son el mismo.
   b) Cada par correspondiente de subexpresiones de t_1 y t_2 deben ser unificables.

Ejemplo 1.3. Veamos los siguientes ejemplos:

- El conjunto $\{c, d\}$ no es unificable pues consta de dos constantes diferentes.
- El conjunto $\{x, fy\}$ es unificable mediante $\sigma = [x := fy]$.
- El conjunto $\{gxw, hya\}$ no es unificable pues g y h son símbolos de función distintos.
- El conjunto $\{fxgyw, fagbhw\}$ no es unificable pues los subtérminos w y hw no son unificables.

Para el caso general en que $W = \{t_1, \ldots, t_n\}$ el unificador se obtiene aplicando recursivamente el caso para dos términos como sigue:

- Hallar μ **umg** de t_1, t_2.
- Hallar ν unificador de $\{t_2\mu, t_3\mu_1, \ldots, t_n\mu\}$.
- El unificador de W es la composición $\nu\mu$.

1.4. El algoritmo de unificación de Martelli-Montanari

A continuación se describe el algoritmo de unificación de Martelli-Montanari para términos [6].

> **Entrada**: un conjunto de ecuaciones $\{s_1 = r_1, \ldots, s_k = r_k\}$ tales que se desea unificar simultáneamente s_i con r_i para $1 \leqslant i \leqslant k$.
>
> **Salida**: un unificador más general μ tal que $s_i\mu = r_i\mu$ para toda $1 \leqslant i \leqslant n$.

Para unificar un conjunto de términos:

- se colocan como ecuaciones las expresiones a unificar
- escoger una de ellas de manera *no determinística* que tenga la forma de alguna de las siguientes reglas y realizar la acción correspondiente:

	Nombre de la regla	$t_1 = t_2$	Acción
[DESC]	Descomposición	$fs_1 \ldots s_n = ft_1 \ldots t_n$	cambiar $t_1 = t_2$ por $\{s_i = t_i\}$
[DFALLA]	Desc. fallida	$fs_1 \ldots s_n = gt_1 \ldots t_n$ donde $g \neq f$	falla
[ELIM]	Eliminación	$x = x$	eliminar
[SWAP]	Intercambio	$t = x$ donde t no es una variable	cambiar por $x = t$
[SUST]	Sustitución	$x = t$ donde x no figura en t	eliminar $x = t$ y aplicar la sustitución $[x := t]$ a las ecuaciones restantes
[SFALLA]	Sust. fallida	$x = t$ donde x figura en t y $x \neq t$	falla

- el algoritmo termina cuando no se puede llevar a cabo alguna acción o cuando falla.
  En caso de éxito se obtiene el conjunto vacío de ecuaciones y el unificador más general se obtiene al componer todas las sustituciones usadas por la regla de sustitución en el orden en que se generaron.

Observaciones: El caso en que se deba unificar a dos constantes iguales a, se genera la ecuación $a = a$ que por la regla de descomposición se sustituye por el conjunto vacío de ecuaciones dado que constantes se consideran funciones sin argumentos, así cualquier ecuación de la forma $a = a$ se elimina. En el caso de una ecuación $a = b$ (con dos constantes distintas) falla por la regla de descomposición fallida.

Para la unificación de cualquier número de términos en un conjunto, se debe usar el algoritmo anterior para unificar un par de ellos generando un unificador que debe ser aplicado al resto del conjunto y continuar con el proceso tomando otros dos términos a unificar y así sucesivamente hasta que el conjunto sólo contenga un elemento o algún proceso de unificación falle.

Ejemplo 1.4. *Sean $f^{(2)}, g^{(1)}, h^{(2)}$ y $W = \{fgxhxu,\ fzhfyyz\}$. Mostramos el proceso de ejecución del algoritmo de manera similar al razonamiento de verificación de correctud de argumentos por interpretaciones.*

1.	$\{fgxhxu = fzhfyyz\}$	*Entrada (ecuación con cualesquiera dos términos de W)*
2.	$\{gx = z,\ hxu = hfyyz\}$	DESC,*1*
3.	$\{z = gx,\ hxu = hfyyz\}$	SWAP,*2*
4.	$\{hxu = hfyygx\}$	SUST,*3*, $[z := gx]$
5.	$\{x = fyy,\ u = gx\}$	DESC,*4*
6.	$\{u = gfyy\}$	SUST,*5*, $[x := fyy]$
7.	$\varnothing$	SUST,*6*, $[u := gfyy]$

El unificador se obtiene al componer las sustituciones utilizadas desde el inicio:

$$\begin{aligned} \mu &= [z := gx][x := fyy][u := gfyy] \\ &= [z := gfyy, x := fyy][u := gfyy] \\ &= [z := gfyy, x := fyy, u := gfyy] \end{aligned}$$

Veamos ahora un ejemplo de un conjunto no unificable:

Ejemplo 1.5. *Sean* $f^{(3)}, g^{(1)}$ *y* $W = \{fxyx,\ fygxx,\ faza\}$.

1.	$\{fxyx = fygxx\}$	*Entrada (ecuación con cualesquiera dos términos de* W*)*
2.	$\{x = y,\ y = gx,\ x = x\}$	DESC,*1*
3.	$\{x = y,\ y = gx\}$	ELIM,*2*
4.	$\{y = gy\}$	SUST,*3,* $[x := y]$
5.	X	SFALLA,*4*

Por lo tanto todo el conjunto W *no es unificacble.*

Si por otro lado se tomara la siguiente:

1.	$\{fxyx = faza\}$	*Entrada (ecuación con cualesquiera dos términos de* W*)*
2.	$\{x = a,\ y = z,\ x = a\}$	DESC,*1*
3.	$\{y = z\}$	SUST,*2,* $[x := a]$ *y* ELIM
4.	$\{\ \}$	SUST,*3,* $[y := z]$ *y* ELIM

se tiene el unificador $\mu_1 = [x := a, y := z]$ *que se aplica a* W *para obtener* $W' = \{fzgaa,\ faza\}$ *y se continúa con la unificación:*

1.	$\{fzgaa = faza\}$	*Entrada (ecuación con cualesquiera dos términos de* W'*)*
2.	$\{z = a,\ ga = z,\ a = a\}$	DESC,*1*
3.	$\{z = a,\ ga = z\}$	ELIM,*2*
4.	$\{ga = a\}$	SUST,*3,* $[z := a]$ *y* ELIM
5.	X	SFALLA,*4*

para finalmente llegar al mismo resultado: el conjunto no es unificable.

Para terminar enunciamos el teorema de correctud total del algoritmo.

Teorema 1 (Correctud total del algoritmo de Martelli-Montanari)**.** *Dado un conjunto finito de expresiones* W*, el algoritmo MM termina, dando como resultado el mensaje "*W *no es unificable", en el caso en que* W *no sea unificable, y un unificador más general* μ *de* W *en el caso en que* W *sea unificable.*

1.5. Unificación de literales

El algoritmo de unificación puede generalizarse para unificar literales (predicados) de manera sencilla simplemente adaptando las reglas de descomposición y descomposición fallida considerando los casos para símbolos de predicado de manera análoga a los símbolos de función.

Como ejemplo se deja al lector unificar el siguiente conjunto

$$W = \{Qxaz, Qyahy, Qxahgb\}$$

1.6. Resolución binaria con unificación

La regla de resolución binaria de Robinson, que veremos más adelante, proporciona un método sintáctico para decidir la correctud de un argumento formalizado en lógica de predicados.

Si bien la idea del método de prueba resolución es la misma que en la lógica proposicional, es decir llegar a la cláusula vacía para mostrar que un argumento es correcto o que una fórmula es válida, en lógica de predicados el método se complica debido a la presencia de variables. Además, las fórmulas deben pasar por un proceso que las convierta en fórmulas en forma normal clausular para poder aplicar resolución.

Recordemos que en lógica proposicional, la cláusula vacía se obtiene al resolver dos literales contrarias [5], por ejemplo $p, \neg p$. Las literales en lógica de predicados son las fórmulas atómicas que son los predicados.

[5] Las literales proposicionales son fórmulas atómicas, las constantes lógicas o variables proposicionales o negaciones de fórmulas atómicas.

Pero a diferencia de las literales en lógica proposicional, resolver dos literales en predicados requiere de más cuidado. Por ejemplo, el par de literales $P(a,x), \neg P(z,b)$ no puede resolverse aunque tengan el mismo símbolo de predicado pues no son contrarias estrictamente dado que no comparten las mismas variables y/o constantes en el mismo orden.

Sin embargo en ciertos casos, como veremos después, podemos aplicar una sustitución para *unificar* las literales y aplicar resolución. Del ejemplo anterior, se puede utilizar la sustitución $\sigma = [x, z := b, a]$ para $P(a,x)$ y $\neg P(z,b)$ y obtener como resultado $P(a,b)$ y $\neg P(a,b)$ que ahora sí son literales contrarias.

Por lo tanto la búsqueda y uso de sustituciones que permitan obtener literales contrarias será de primordial importancia para usar resolución en lógica de predicados. Esta clase de sustituciones se conocen como *unificadores* y hemos visto anteriormente cómo es posible hallarlos algorítmicamente.

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 y ℓ' son **unificables** mediante un σ. La regla devuelve como conclusión la disyunción de las literales de $\mathcal{C}$ y $\mathcal{D}$ 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.

Ejemplo 1.6. Veamos un ejemplo sencillo:

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

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 1.7. Verificar que $\forall x \exists y \big(Cx \to Py \wedge Axy\big) \models \forall z \big(Cz \to \exists w (Pw \wedge Azw)\big)$.
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 $\{\forall x Px, \forall x \neg Pfx\}$ 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 $\sigma = [y := fx]$ y por tanto a la cáusula vacía para concluir que el conjunto es insatisfacible.

1.7. 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 2 (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 esta nota.

Teorema 3 (Completud Refutacional de la Resolución)**.** *Si* Γ *es insatisfacible entonces existe una derivación de la cláusula vacía* $\Gamma \vdash_{\mathcal{R}} \square$*.*

2. Resolución Binaria en Programación Lógica

2.1. 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 para fines de esta nota.

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 operador $\vee$ 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 o términos 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 $\mathcal{C}$.

2.2. 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 $k \geqslant 1$ 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 $j \geqslant 0$, 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 $j \geqslant 0$, 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 $j \geqslant 1$, 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**.

2.3. 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 6. *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 maneja.

Definición 7. *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 no son nunca 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 2.1 Considérese el programa para la suma de naturales, recordemos que los predicados relacionan objetos, en este caso el predicado $P(x, y, z)$ se lee como *la suma de x, y es z.*

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

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 2.2 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 hubieramos elegido resolver la segunda la derivación será:

7.	$\leftarrow$	$M0ss0v, Pvs0z_2$	$res(4, 6, [x_2, y_2, w := v, s0, sz_2])$
8.	$\leftarrow$	$M0ss0v, Pv0z_3$	$res(4$ 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 muy sensible al orden de las cláusulas y al orden en que se eligen literales para resolver. A continuación ejemplificamos este fenómeno.

Ejemplo 2.3 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 $[w := s0]$.

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

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

Si elegimos ahora en 6 la segunda literal para resolver como no hay información para v, w ya que ambas cláusulas para la suma son aplicables y 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:

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

Si seguimos usando la misma elección la búsqueda no terminará jamás aún cuando $\leftarrow$ si 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.

Obsérvese en el programa para la suma se puede interpretar el predicado 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.

3. 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. Las clásulas de un programa en PROLOG tienen 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

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

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.

3.1. Resolución SLD

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 SL-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 no se reemplaza con nada (puesto que los hechos son cláusulas sin cuerpo). Esto es, 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 tomar 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 3.1. 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.

3.2. Árboles asociados a un programa lógico

Definición 8. *Una SLD-derivación* [6] Δ *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:*

- *Para toda* n*,* G_{n+1} *es un SLD-resolvente de* G_n *y* $\mathcal{C}_m$ *via el umg* μ_k*.*
- *Las variables que figuran en* $\mathcal{C}_{n+1}$ *no figuran en ninguna de las cláusulas* $G_0, \ldots, G_n, \mathcal{C}_0, \ldots, \mathcal{C}_m$ *ni en las sustituciones* $\mu_0, \ldots, \mu_k$*.*

Si la derivación Δ *es finita digamos* $G_0, \ldots, G_i$ *decimos que* i *es la longitud de* Δ*.*

Definición 9. *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* $G_n = \square$*.*
- *Para toda* $1 \leqslant i \leqslant n$*,* $\mathcal{C}_i$ *es variante de alguna cláusula de* $\mathbb{P}$*.*

Definición 10. *Un árbol de prueba es un árbol binario que tiene en los nodos metas o cláusulas de programa y se construye mediante aplicaciones de la regla de resolución binaria:*
si C_R *es la cláusula resolvente a partir de la cláusula de programa* C *y de la meta* G *entonces el árbol tendra a* C_R *como hijo de* C *y* G*.*

Definición 11. *Un árbol de búsqueda es un árbol n-ario posiblemente infinito cuyos nodos tienen metas exclusivamente y donde el hijo de un nodo* G *es la nueva meta* G_1 *obtenida a partir de* G *y de una cláusula de programa* C *mediante resolución binaria.*

Este árbol 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.

Una SLD-derivación se presenta como un árbol n-ario finito o infinito, llamado árbol de resolución SLD. Adicionalmente el árbol de búsqueda de una meta es un árbol no necesariamente binario cuyos nodos son metas exclusivamente. Los hijos de un nodo son las metas obtenidas al resolver la meta del nodo actual con alguna de las cláusulas de un programa lógico. Las ramas pueden ser infinitas, en cuyo caso el proceso de búsqueda de $\square$ se cicla; si son finitas hay de dos tipos, las que se llaman ramas de éxito cuando terminan en $\square$ o las ramas de falla en otro caso.

Ejemplo 3.1 Considera el siguiente programa que describe tres propiedades básicas (P, Q, R) y una cuarta que depende de las tres anteriores:

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

[6] Esta derivación también se llama (en inglés) SLD-resolution tree o SLD-tree.

Analizaremos tres árboles diferentes:

- La meta $? - S(W)$ busca obtener los elementos que cumplen con la propiedad S, para ello se calcula el árbol SLD.
- Por otro lado, el árbol de la derecha para la meta $?-S(Y)$ busca obtener una derivación particular, en este caso es una que no obtiene la cláusula vacía y por lo tanto no tiene éxito.
- Y el tercer árbol es una SLD refuctación para la misma meta, es decir que sí tiene éxito al encontrar la cláusula vacía.

De estos árboles, los de las figuras 2 y 3 son subárboles del primero.

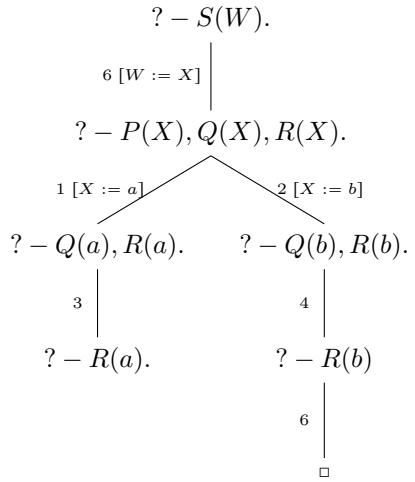


Figura 1: Árbol SLD para la meta $? - \texttt{S(W)}$.

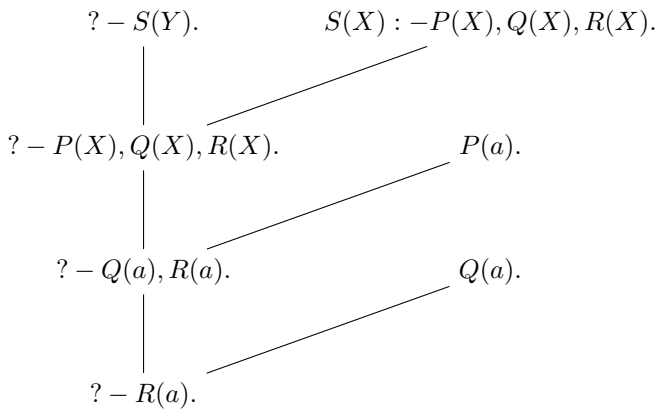


Figura 2: SLD derivación para $? - \texttt{S(Y)}$.

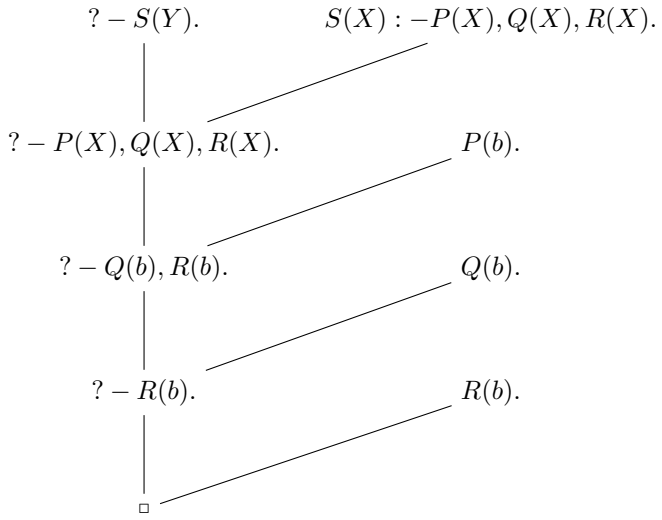
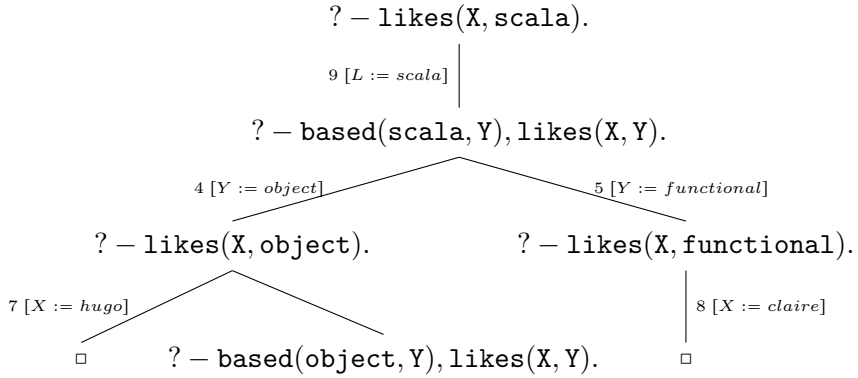


Figura 3: SLD refutación para $? - \texttt{S(Y)}$.

Ejemplo 3.2 Considera el siguiente programa que describe el estilo de programación en el que está basado un lenguaje de programación y los gustos de algunas personas por ciertos estilos. El árbol SDL corresponde a la meta $? - \texttt{likes(X, scala)}$.

1. based(prolog, logic).
2. based(java, object).
3. based(haskell, functional).
4. based(scala, object).
5. based(scala, functional).

6. likes(max, logic).
7. likes(hugo, object).
8. likes(claire, functional).
9. likes(X, L) : −based(L, Y), likes(X, Y).



4. 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 12 (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 13 (Respuesta correcta)**.** *Decimos que una respuesta σ para $\mathbb{P} \cup \{G\}$ es correcta si $\mathbb{P} \models G_i\sigma$ para toda $1 \leqslant i \leqslant m$, 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 4.1 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

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

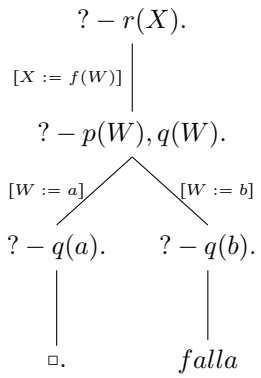
Definición 14 (Respuesta computada)**.** *Sean $\mathbb{P}$ un programa lógico y G una meta. Una sustitución σ es una respuesta computada para $\mathbb{P} \cup \{G\}$ si y sólo si existe una rama de éxito en el árbol de SLD-resolución (o árbol de búsqueda) de longitud n con unificadores más generales $\mu_0, \ldots, \mu_{n-1}$ de tal forma que $\sigma = \mu_0\mu_1 \ldots \mu_{n-1}\Uparrow_{Var(G)}$*

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 4.2 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.

5. Universo y Modelos de Herbrand

Definición 15 (Universo de Herbrand)**.** *Sea $\mathcal{L}$ un lenguaje de primer orden. El universo de Herbrand de $\mathcal{L}$ se define como el conjunto $\mathcal{H}_{\mathcal{L}}$ de los términos cerrados de $\mathcal{L}$. Es decir*

$$\mathcal{H}_{\mathcal{L}} = \{t \mid t \text{ es un } \mathcal{L}\text{-término cerrado } \}$$

Ejemplo 5.1.

1. Si $\mathcal{L}_1 = \{a, b, P^{(1)}, Q^{(1)}, R^{(2)}\}$ entonces $\mathcal{H}_{\mathcal{L}_1} = \{a, b\}$
2. Si $\mathcal{L}_2 = \{c, a, f^{(1)}, g^{(1)}, P^{(1)}\}$ entonces
   $\mathcal{H}_{\mathcal{L}_2} = \{c, a, f(c), f(a), g(c), g(a), f(f(a)), f(g(a)), f(f(c)), f(g(c)), g(g(a)), g(g(c)), \ldots\}$

Obsérvese que si el lenguaje NO contiene símbolos de función entonces el universo de Herbrand es finito y es simplemente el conjunto de símbolos de constante. Por otro lado si el lenguaje contiene al menos un símbolo de función y un símbolo de constante el universo de Herbrand se vuelve infinito. Si en un lenguaje no hubiera constantes se agrega una para poder formar el universo de Herbrand.

Definición 16 (Base de Herbrand)**.** *Sean $\mathcal{L}$ un lenguaje y $\mathcal{H}_{\mathcal{L}}$ su universo de Herbrand. La base de Herbrand $\mathcal{B}_{\mathcal{L}}$ de $\mathcal{L}$ se define como el conjunto de fórmulas atómicas cerradas, es decir, el conjunto de fórmulas atómicas cuyos argumentos pertenecen al universo de Herbrand de $\mathcal{L}$.*

Ejemplo 5.2. Con respecto a los lenguajes del ejemplo anterior se tiene

1. $\mathcal{B}_{\mathcal{L}_1} = \{Pa, Pb, Qa, Qb, Raa, Rbb, Rab, Rba\}$
2. $\mathcal{B}_{\mathcal{L}_2} = \{Pc, Pa, Pfc, Pfa, Pgc, Pga, Pffa, Pfga, Pffc, Pfgc, Pgga, Pggc, \ldots\}$

En particular nos interesará construir el universo y base de Herbrand de un programa lógico, definidos como sigue:

Definición 17. *Dado un programa lógico $\mathbb{P}$ definimos el universo $\mathcal{H}_{\mathbb{P}}$ y la base de Herbrand $\mathcal{B}_{\mathbb{P}}$ de $\mathbb{P}$ como:*

$$\mathcal{H}_{\mathbb{P}} =_{def} \mathcal{H}_{\mathcal{L}(\mathbb{P})} \qquad \mathcal{B}_{\mathbb{P}} =_{def} \mathcal{B}_{\mathcal{L}(\mathbb{P})}$$

donde $\mathcal{L}(\mathbb{P})$ es el lenguaje de los símbolos de constante, función y predicado que figuran en $\mathbb{P}$.

Ejemplo 5.1 Si $\mathbb{P} = \{nat(0).,\ nat(s(X))\ \text{:-}\ nat(X).\}$ entonces $\mathcal{L}(\mathbb{P}) = \{nat^{(1)}, 0, s^{(1)}\}$ y

$$\mathcal{H}_{\mathbb{P}} = \{0, s(0), s(s(0)), \ldots\} \quad \mathcal{B}_{\mathbb{P}} = \{nat(0), nat(s(0)), nat(s(s(0))), \ldots\}$$

Ejemplo 5.2 Si $\mathbb{P}$ es el programa lógico para caminos en una gráfica (como en la nota 11) entonces el lenguaje de $\mathbb{P}$ es $\mathcal{L}(\mathbb{P}) = \{a, b, c, d, e, f, edge^{(2)}, path^{(2)}\}$ y $\mathcal{H}_{\mathbb{P}} = \{a, b, c, d, e, f\}$

$$\mathcal{B}_{\mathbb{P}} = \{edge(a,a), edge(a,b) \ldots, edge(f,e), edge(f,f), path(a,a), path(a,b) \ldots, path(f,e), path(f,f)\}$$

Definición 18 (Interpretación de Herbrand)**.** *Decimos que una interpretación $\mathcal{M} = \langle M, \mathcal{I} \rangle$ de un lenguaje dado $\mathcal{L}$ es una interpretación o modelo de Herbrand si y sólo si se cumple lo siguiente:*

1. *$M = \mathcal{H}_{\mathcal{L}}$ es decir, el universo de $\mathcal{M}$ es el universo de Herbrand de $\mathcal{L}$.*
2. *Para todo símbolo de constante c, se cumple $\mathcal{I}(c) = c$.*
3. *Para todo símbolo de función $f^{(n)}$, se cumple $\mathcal{I}(f(t_1, \ldots, t_n)) = f(t_1, \ldots, t_n)$.*
4. *Para todo símbolo de predicado $P^{(n)}$ es un subconjunto de $P^{\mathbb{N}} \subseteq \mathcal{H}_{\mathcal{L}}^n$*

Obsérvese que en una interpretación de Herbrand los símbolos de constante y de función se interpretan como ellos mismos y que lo único que no está determinado es la interpretación de los símbolos de predicado por esta razón a los modelos de Herbrand tambien se les llama modelos sintácticos.

5.1. Representación de modelos de Herbrand

Una manera útil de representar modelos de Herbrand es mediante subconjuntos de la base de Herbrand del lenguaje en cuestión. Dado un modelo de Herbrand $\mathcal{M}$, representamos a $\mathcal{M}$ mediante el subconjunto $B_{\mathcal{M}} \subseteq \mathcal{B}_{\mathcal{H}}$ de aquellas fórmulas atómicas que son verdaderas en $\mathcal{M}$. De esta manera, si la base es finita, es posible construir todos los modelos de Herbrand, y en el caso en que la base es infinita el número de modelos de Herbrand también, pero pueden enumerarse efectivamente como subconjuntos de la base.

Ejemplo 5.3 Si $\mathcal{L} = \{a, P^{(1)}, Q^{(1)}\}$ entonces

$$\mathcal{H}_{\mathcal{L}} = \{a\} \quad \mathcal{B}_{\mathcal{L}} = \{Pa, Qa\}$$

y existen cuatro interpretaciones o modelos de Herbrand para $\mathcal{L}$:

1. $\mathcal{M}_1$ tal que $\mathcal{M}_1 \models Pa$ y $\mathcal{M}_1 \models Qa$
2. $\mathcal{M}_2$ tal que $\mathcal{M}_2 \models Pa$ y $\mathcal{M}_2 \not\models Qa$
3. $\mathcal{M}_3$ tal que $\mathcal{M}_3 \not\models Pa$ y $\mathcal{M}_3 \models Qa$
4. $\mathcal{M}_4$ tal que $\mathcal{M}_4 \not\models Pa$ y $\mathcal{M}_4 \not\models Qa$

Cada modelo corresponde a un subconjunto $B_{\mathcal{M}_i}$ de la base como sigue:

1. $B_{\mathcal{M}_1} = \mathcal{B}_{\mathcal{L}} = \{Pa, Qa\}$ representa a $\mathcal{M}_1$
2. $B_{\mathcal{M}_2} = \{Pa\}$ representa a $\mathcal{M}_2$

3. $B_{\mathcal{M}_3} = \{Qa\}$ representa a $\mathcal{M}_3$
4. $B_{\mathcal{M}_4} = \varnothing$ representa a $\mathcal{M}_4$

Dado que cada subconjunto de $B \subseteq \mathcal{B}_\mathcal{L}$ determina de manera única a un modelo de Herbrand $\mathcal{M}$, identificamos a $\mathcal{M}$ con B. Por ejemplo, el modelo $\mathcal{M}_2$ del ejemplo anterior, se define como $\mathcal{M}_2 = \{Pa\}$.

Con esta idea en mente, si deseamos verificar que una fórmula atómica A es verdadera en $\mathcal{M}$ basta ver que $A \in B_\mathcal{M}$

El siguiente lema nos dice cómo se evaluan los términos en una interpretación de Herbrand:

Lema 1. *Sean $\mathcal{L}$ un lenguaje, $\mathcal{M} = \langle M, \mathcal{I}\rangle$ una interpretación de Herbrand para $\mathcal{L}$, t un término con variables $x_1, \ldots, x_n$ y ν un estado de las variables tal que $\nu(x_i) = r_i$ donde $r_i \in \mathcal{H}_\mathcal{L}$ es decir, r_i es un término cerrado, para cada $1 \leqslant i \leqslant n$. Entonces:*

$$\mathcal{I}_\nu(t) = t[x_1 := r_1, \ldots, x_n := r_n]$$

En particular si t es un término cerrado entonces $\mathcal{I}_\nu(t) = t$

Demostración. (Por inducción sobre los términos). Ejercicio. □

El lema anterior nos dice que la interpretación de términos en un modelo de Herbrand coincide con aplicar una sustitución al término y que los términos cerrados se interpretan como ellos mismos.

6. El Teorema de Herbrand

El teorema de Herbrand es esencial para describir la semántica de programas lógicos, a continuación lo estudiamos con detalle.

Definición 19. *Sean $\mathcal{L}$ un lenguaje y $\mathcal{K}$ un conjunto de $\mathcal{L}$-fórmulas cerradas y universales. El conjunto de instancias cerradas $\varphi\sigma$ o cerraduras universales, donde $\forall x_1 \ldots \forall x_n \varphi$ pertenece a $\mathcal{K}$, se llama conjunto de instancias cerradas de $\mathcal{K}$ y se denota $IC(\mathcal{K})$.*

A continuación presentamos el Teorema Clásico de Herbrand enunciado de forma adecuada a nuestras necesidades.

Teorema 4 (de Herbrand)**.** *Sean $\mathcal{L}$ un lenguaje con al menos una constante y $\mathcal{K}$ un conjunto de $\mathcal{L}$-enunciados universales. Las siguientes condiciones son equivalentes:*

1. *$\mathcal{K}$ tiene un modelo.*
2. *$\mathcal{K}$ tiene un modelo de Herbrand.*
3. *$IC(\mathcal{K})$ tiene un modelo.*
4. *$IC(\mathcal{K})$ tiene un modelo de Herbrand.*

Demostración. Debido a que todo modelo de Herbrand es un modelo, las implicaciones $b \Rightarrow a$ y $d \Rightarrow c$ son triviales.
La válidez universal de la fórmula $\forall x_1 \ldots \forall x_n \varphi \to \varphi\{x_1/t_1, \ldots, x_n/t_n\}$ hace inmediatas a las implicaciones $a \Rightarrow c$ y $b \Rightarrow d$. Así que basta probar la implicación $c \Rightarrow b$.
Sea $\mathcal{M}$ un modelo de $IC(\mathcal{K})$. Definimos una estructura de Herbrand $\mathcal{N} = \langle \mathcal{H}_\mathcal{L}, \mathcal{I}\rangle$, para lo cual basta decir como se interpretan los símbolos de predicado, puesto que por definición los símbolos de constante y de función se interpretan como ellos mismos. Si P es un símbolo de predicado n-ario, entonces definimos:

$$P^\mathcal{N} = \{(t_1, \ldots, t_n) \mid \mathcal{M} \models P(t_1, \ldots, t_n)\}$$

Obsérvese que, por construcción de $\mathcal{N}$, se cumple que:

$$\mathcal{M} \models P(t_1, \ldots, t_n) \text{ si y sólo si } \mathcal{N} \models P(t_1, \ldots, t_n)$$

es decir, $\mathcal{M}$ y $\mathcal{N}$ validan a las mismas fórmulas atómicas cerradas. Dicho resultado se puede extender a cualquier fórmula cerrada libre de cuantificadores mediante inducción sobre fórmulas (ejercicio [7]).
Por último veamos que $\mathcal{N}$ es modelo de $\mathcal{K}$. Sea $\forall x_1 \ldots \forall x_n \varphi$ una fórmula de $\mathcal{K}$. Queremos demostrar que $\mathcal{N} \models \forall x_1 \ldots \forall x_n \varphi$, lo cual es equivalente a mostrar que para cualesquiera $t_1, \ldots, t_n \in \mathcal{H}_\mathcal{L}$, $\mathcal{N} \models \varphi[x_1 := t_1, \ldots, x_n := t_n]$. Pero esta última afirmación es equivalente, por lo recién observado a que para cualesquiera $t_1, \ldots, t_n \in \mathcal{H}_\mathcal{L}$, $\mathcal{M} \models \varphi[x_1 := t_1, \ldots, x_n := t_n]$, lo cual es cierto puesto que $\varphi[x_1 := t_1, \ldots, x_n := t_n]$ pertenece a $IC(\mathcal{K})$ y por hipótesis, $\mathcal{M}$ es modelo de $IC(\mathcal{K})$. De manera que $\mathcal{N} \models \forall x_1 \ldots \forall x_n \varphi$ por lo que $\mathcal{N}$ es un modelo de Herbrand de $\mathcal{K}$. □

Es importante observar que el teorema de Herbrand sólo es válido para enunciados universales. Por ejemplo si $\mathcal{K} = \{Pa, \exists x \neg Px\}$ entonces $\mathcal{K}$ es un conjunto de fórmulas cerradas pero no universales y claramente tiene un modelo, digamos $M = \{0, 1\}$ con $\mathcal{I}(a) = 0$ y $P^\mathcal{M} = \{0\}$, pero $\mathcal{K}$ no tiene un modelo de Herbrand pues el universo de Herbrand es $\mathcal{H}_\mathcal{L} = \{a\}$ de manera que los únicos modelos de Herbrand son los representados por $\varnothing$ que corresponde a una interpretación donde Pa es falsa y por lo tanto no es modelo de $\mathcal{K}$; y el representado por $\{Pa\}$ que tampoco es modelo de $\mathcal{K}$ pues $\exists x \neg Px$ es falsa.

Dado que las cláusulas son enunciados universales, se tiene el siguiente

Corolario 2. *Sea $\mathcal{C}$ un conjunto de cláusulas de Horn. Las siguientes condiciones son equivalentes:*

- *$\mathcal{C}$ tiene un modelo.*
- *$\mathcal{C}$ tiene un modelo de Herbrand.*

En lo que resta de esa nota presentamos algunos resultados semánticos para programas lógicos de Horn.

6.1. Semática declarativa de programas lógicos

Debido a las formas de sus cláusulas, un programa lógico $\mathbb{P}$ siempre tiene un modelo en particular debido al teorema de Herbrand basta estudiar los modelos de Herbrand del programa $\mathbb{P}$. El significado declarativo del programa $\mathbb{P}$ consiste de todas las consecuencias lógicas de $\mathbb{P}$ y puede representarse con un modelo particular llamado el modelo mínimo de Herbrand de $\mathbb{P}$.

Construir la base de Herbrand de $\mathbb{P}$ es importante debido a la siguiente

Proposición 1. *Si $\mathbb{P}$ es un programa lógico entonces $\mathbb{P}$ tiene un modelo de Herbrand*

Demostración. La base de Herbrand $\mathcal{B}_\mathbb{P}$ representa al modelo que hace verdaderas a todas las fórmulas atómicas del lenguaje lo cual implica que todas las cláusulas de $\mathbb{P}$ son verdaderas debido a su forma (hechos o reglas). □

De manera que cualquier programa lógico tiene al menos un modelo de Herbrand, enseguida veremos que existe un modelo de Herbrand mínimo con respecto a la contención.

Proposición 2. *Sea $\mathbb{P}$ un programa lógico. $\mathbb{P}$ tiene un modelo de Herbrand mínimo $\mathcal{M}_\mathbb{P}$ definido por:*

$$\mathcal{M}_\mathbb{P} = \bigcap \{\mathcal{M} \mid \mathcal{M} \text{ es modelo de Herbrand de } \mathbb{P}\}$$

[7] Hay que probar el siguiente

Lema 2. *Si φ es un enunciado sin cuantificadores entonces $\mathcal{M} \models \varphi$ syss $\mathcal{N} \models \varphi$.*

Demostración. Sabemos que $\mathbb{P}$ tiene al menos un modelo de Herbrand representado por la base de Herbrand $\mathcal{B}_\mathbb{P}$. Es fácil ver que si $\mathcal{M}_1, \mathcal{M}_2$ son modelos de Herbrand de $\mathbb{P}$ su intersección también lo es. De manera que el modelo mínimo de Herbrand $\mathcal{M}_\mathbb{P}$ está bien definido. □

Proposición 3. *Sea $\mathbb{P}$ un programa lógico. El modelo de Herbrand mínimo $\mathcal{M}_\mathbb{P}$ se representa con el conjunto de átomos que son consecuencia lógica de $\mathbb{P}$. Es decir,*

$$\mathcal{M}_\mathbb{P} = \{A \in \mathcal{B}_\mathbb{P} \mid \mathbb{P} \models A\}$$

Demostración. $\subseteq$). Sea $A \in \mathcal{M}_\mathbb{P}$. Veamos que $\mathbb{P} \models A$. Si esto no sucediera entonces $\mathbb{P} \cup \{\neg A\}$ tendría un modelo y por el teorema de Herbrand tendría un modelo de Herbrand $\mathcal{N}$. Pero entonces como $\mathcal{N} \models \neg A$ mos que $A \notin \mathcal{N}$ pero por minimalidad $\mathcal{M}_\mathbb{P} \subseteq \mathcal{N}$ y entonces también $A \in \mathcal{N}$ lo cual es absurdo.
$\supseteq$). Sea A tal que $\mathbb{P} \models A$. Queremos ver que $A \in \mathcal{M}_\mathbb{P}$. Como $\mathcal{M}_\mathbb{P}$ es modelo de $\mathbb{P}$ y $\mathbb{P} \models A$ entonces $\mathcal{M}_\mathbb{P} \models A$, es decir, $A \in \mathcal{M}_\mathbb{P}$.

Dado que $\mathcal{M}_\mathbb{P}$ contiene exactamente a todos los átomos que son consecuencia lógica de $\mathbb{P}$ entonces este modelo corresponde realmente al significado intensional o estandar de $\mathbb{P}$. Es decir, la semántica declarativa de $\mathbb{P}$ está dada por $\mathcal{M}_\mathbb{P}$.

6.2. Procedimiento para hallar el modelo mínimo de Herbrand

Desde el punto de vista puramente matemático la semántica declarativa de un programa lógico ha quedado definida mediante el modelo mínimo. Sin embargo, en la práctica no es claro cómo construir directamente a $\mathcal{M}_\mathbb{P}$, pues su definición involucra a una intersección generalmente infinita. A continuación veremos que existe una manera **mecánica** para construir el modelo mínimo.

Definición 20. *Dado un programa lógico $\mathbb{P}$, el operador de consecuencia inmediata $\mathcal{T}_\mathbb{P}$ se define como sigue*

$$\mathcal{T}_\mathbb{P}(\mathcal{K}) = \{P \in \mathcal{B}_\mathbb{P} \mid P \text{ :- } Q_1, \ldots, Q_m \text{ es instancia cerrada de una cláusula de } \mathbb{P} \text{ y } Q_1, \ldots, Q_m \in \mathcal{K}\}$$

donde $\mathcal{K} \subseteq \mathcal{B}_\mathbb{P}$.

Definición 21. *Sea $\mathbb{P}$ un programa lógico. Definimos las iteraciones del operador de consecuencia inmediata $\mathcal{T}_\mathbb{P}$ para $\mathcal{K} \subseteq \mathcal{B}_\mathbb{P}$ recursivamente como sigue:*

$$\begin{aligned} \mathcal{T}_0(\mathcal{K}) &= \mathcal{K} \\ \mathcal{T}_{n+1}(\mathcal{K}) &= \mathcal{T}_\mathbb{P}\big(\mathcal{T}_n(\mathcal{K})\big) \\ \mathcal{T}_\omega(\mathcal{K}) &= \textstyle\bigcup_{n=0}^{\infty} \mathcal{T}_n(\mathcal{K}) \end{aligned}$$

El modelo mínimo de Herbrand de un programa lógico puede obtenerse mediante las iteraciones del operador de consecuencia inmediata iniciando en el conjunto vacío, como lo asegura la siguiente

Proposición 4. *Sean $\mathbb{P}$ un programa lógico y $\mathcal{M}_\mathbb{P}$ su mínimo modelo de Herbrand. Entonces $\mathcal{M}_\mathbb{P} = \mathcal{T}_\omega(\varnothing)$.*

Veamos un ejemplo.

Ejemplo 6.1 Sea $\mathbb{P}$ el siguiente programa $\{p(X, a) \text{ :- } q(X).\ p(X, Y) \text{ :- } q(X), r(Y).\ q(a).\ r(b).\ q(b).\ r(c).\}$

- El lenguaje de $\mathbb{P}$ es $\mathcal{L}(\mathbb{P}) = \{a, b, c, p^{(2)}, q^{(1)}, r^{(1)}\}$
- El universo de Herbrand de $\mathbb{P}$ es $\mathcal{H}_\mathbb{P} = \{a, b, c\}$
- La base de Herbrand de $\mathbb{P}$ es

$$\mathcal{B}_\mathbb{P} = \{q(a), q(b), q(c), r(a), r(b), r(c), p(a, a), p(a, b), p(a, c), p(b, a), p(b, b), p(b, c), p(c, a), p(c, b), p(c, c)\}$$

- El modelo mínimo de Herbrand de $\mathbb{P}$ es:
  - $\mathcal{T}_0(\varnothing) = \varnothing$
  - $\mathcal{T}_1(\varnothing) = \mathcal{T}_{\mathbb{P}}(\mathcal{T}_0(\varnothing)) = \mathcal{T}_{\mathbb{P}}(\varnothing) = \{q(a), r(b), q(b), r(c)\}$
  - $\mathcal{T}_2(\varnothing) = \mathcal{T}_{\mathbb{P}}(\mathcal{T}_1(\varnothing)) = \mathcal{T}_{\mathbb{P}}(\{q(a), r(b), q(b), r(c)\})$
    $= \{q(a), r(b), q(b), r(c), p(a,a), p(b,a), p(a,b), p(a,c), p(b,b), p(b,c)\}$
  - $\mathcal{T}_3(\varnothing) = \mathcal{T}_{\mathbb{P}}(\mathcal{T}_2(\varnothing)) = \mathcal{T}_2(\varnothing)$
- En el paso $n = 3$ la sucesión se estabiliza y concluimos

$$\mathcal{M}_{\mathbb{P}} = \mathcal{T}_\omega(\varnothing) = \{q(a), r(b), q(b), r(c), p(a,a), p(b,a), p(a,b), p(a,c), p(b,b), p(b,c)\}$$

6.3. Semática operacional de programas lógicos

El significado operacional de un programa lógico consiste en los resultados obtenidos al ejecutar el programa.

Definición 22. *El conjunto de éxito de un programa lógico $\mathbb{P}$ se define como*

$$\mathcal{E}_{\mathbb{P}} = \{A \in \mathcal{B}_{\mathbb{P}} \mid \mathbb{P} \cup \{?{-}A\} \text{ tiene una SLD-refutación } \}$$

La semántica operacional del programa $\mathbb{P}$ se define como el conjunto de éxitos $\mathcal{E}_{\mathbb{P}}$. Veamos un ejemplo

Ejemplo 6.2 Sea $\mathbb{P}$ el siguiente programa $\{p(f(X)) \text{ :- } p(X).,\ p(a).,\ q(b).\}$. Entonces

- El lenguaje de $\mathbb{P}$ es $\mathcal{L}(\mathbb{P}) = \{a, b, p^{(1)}, q^{(1)}, f^{(1)}\}$
- El universo de Herbrand de $\mathbb{P}$ es $\mathcal{H}_{\mathbb{P}} = \{a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots\}$
- La base de Herbrand de $\mathbb{P}$ es

$$\mathcal{B}_{\mathbb{P}} = \{p(a), p(b), q(a), q(b), p(f(a)), p(f(b)), q(f(a)), q(f(b)), \ldots\}$$

- Claramente las metas $?{-}p(a).$ y $?{-}q(b).$ tienen una SLD-refutación por lo tanto pertenecen a $\mathcal{E}_{\mathbb{P}}$.
- La meta $?{-}p(f(a)).$ tiene éxito pues tenemos una SLD-refutación a partir de las instancias $p(f(a)) \text{ :- } p(a).$ y $p(a).$ de las cláusulas del programa.
- Similarmente podemos verificar que $?{-}p(f^n(a)).$ tiene éxito para cualquier $n \geqslant 0$. Más aún si una meta es éxitosa entonces es de esta forma (salvo $? - q(b)$).
- Por lo tanto el conjunto de éxitos es:

$$\mathcal{E}_{\mathbb{P}} = \{p(a), q(b)\} \cup \{p(f^n(a)) \mid n \geqslant 1\}$$

7. Equivalencia de ambas semánticas

Finalmente observamos la equivalencia de ambas semánticas. Esta equivalencia es corolario de ciertos resultados teóricos que dependen del teorema de Herbrand.

Proposición 5. *Sea $\mathbb{P}$ un programa lógico definido. La semánticas declarativa y operacional para $\mathbb{P}$ coinciden. Es decir,*

$$\mathcal{M}_{\mathbb{P}} = \mathcal{E}_{\mathbb{P}}$$

De manera que toda consecuencia lógica atómica de $\mathbb{P}$ es un éxito de $\mathbb{P}$ y viceversa, todo éxito $G \in \mathcal{E}_{\mathbb{P}}$ es consecuencia lógica de $\mathbb{P}$, es decir, cumple $\mathbb{P} \models G$.

8. Incompletud e Incorrectud de Prolog

Debido a que las implementaciones de PROLOG no verifican las presencia de una variable en un término en la unificiación de $\{X, t\}$ las propiedades de correctud completud no son válidas como lo muestran los siguientes ejemplos:

- Incorrectud: PROLOG contesta que `true` a una meta que no es consecuencia lógica del programa:

```
test :- p(X,X).

p(Y,f(Y)).

--------------------
?- test.
true.

?- p(X,X).
X = f(X).
```

- Incompletud: en el siguiente ejemplo `q` es consecuencia lógica del programa pero PROLOG se cicla y queda sin recursos sin poder contestar afirmativamente.

```
p :- q.
p :- r.
q :- p.
r.

--------------------------
?- q.
ERROR: Out of local stack
   Exception: (4,193,042) p ? abort
% Execution Aborted
?-
```

Referencias

[1] Reference manual swi prolog. `https://www.swi-prolog.org/pldoc/doc_for?object=manual`.

[2] Krzysztof R. Apt. The logic programming paradigm and prolog, 2001.

[3] I. Bratko. *Prolog Programming for Artificial Intelligence.* International Computer Science Series. Addison-Wesley, 2011.

[4] M. Fernandez. *Models of computation - An introduction to computability theory.* Springer, 2009.

[5] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

[6] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, apr 1982.

[7] U. Schöning. *Logic for Computer Scientists*. Modern Birkhäuser Classics. Birkhäuser Boston, 2009.

[8] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. Logic Programming. MIT Press, United States, mar 1994.