Programación lógica e inteligencia artificial

M. Vilares[1], V.M. Darriba[1], C. Gómez-Rodríguez[2], and J. Vilares[2]

[1] Department of Computer Science, University of Vigo
Campus As Lagoas s/n, 32004 Ourense, Spain
{vilares,darriba}@uvigo.es
[2] Department of Computer Science, University of A Coruña
Campus de Elviña s/n, 15071 A Coruña, Spain
{cgomezr,jvilares}@udc.es

1 Introducción

Si preguntásemos a un usuario experto, que encuentra en la programación lógica y que echa de menos en otros paradigmas de programación, probablemente su respuesta se basase en algunas ideas extremadamente sencillas. La primera podría ser la práctica ausencia de sintaxis impuesta por el lenguaje. Unos pocos operadores lógicos y la noción de árbol dejan toda la fuerza expresiva en manos de la lógica de Horn.

Otro factor a menudo esgrimido en favor de la programación lógica es su potencia de cálculo, fruto de la combinación de los mecanismos de unificación y resolución, simples, eficaces y transparentes al programador.

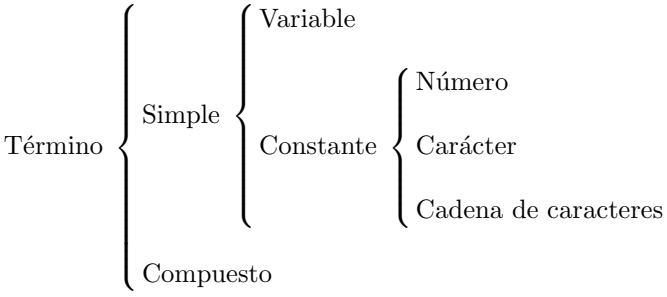
En conjunto, estas características permiten centrar nuestra atención en los aspectos propios al desarrollo algorítmico, que denominamos declarativos, limitando el esfuerzo de programación asociado a factores operacionales; al contrario de lo que ocurre en los lenguajes imperativos. En la práctica, ello se traduce en un código más compacto, lo que ha contribuido a extender la idea de que unas pocas líneas de Prolog equivalen habitualmente a páginas enteras de código tradicional.

Desde un punto de vista más concreto, la programación lógica define un tratamiento extraordinariamente flexible de símbolos y estructuras, a la vez que permite poner en marcha de forma sencilla y rápida estrategias deductivas. Todo ello hace que a menudo se le considere como un paradigma de programación especialmente adaptado al tratamiento de problemas en el dominio de la inteligencia artificial. En este capítulo repasaremos los conceptos básicos, que ilustraremos mediante ejemplos prácticos cuyo objetivo será el de transmitir al lector el particular estilo de programación requerido si queremos exprimir todas las posibilidades asociadas a la combinación de la unificación y la resolución lógica.

2 Léxico y sintaxis

La clase más general de objeto manejado en programación lógica es el *término*. La jerarquía de datos a partir de dicha estructura viene indicada por el siguiente

diagrama:



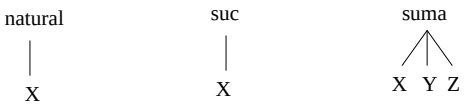
Las *variables* suelen indicarse mediante un identificador cuyo primer carácter es una letra mayúscula. Los *términos compuestos* son los objetos estructurados del lenguaje. Se componen de un *funtor*[3] y una secuencia de uno o más términos llamados *argumentos*. Un funtor se caracteriza por su *firma*, a saber, el par formado por su *nombre*, que es un átomo, y su *aridad* o número de argumentos. Cuando el término compuesto no expresa ningún tipo de relación lógica[4], suele dársele el nombre de *función*. Cuando si expresa ese tipo de relación, lo denominamos *predicado*. Una *constante lógica* no es otra cosa que una función de aridad cero.

Estructuralmente, un objeto se representa mediante un árbol que responde al esquema indicado por su firma, y en el que la raíz está etiquetada con el nombre del funtor y los hijos con los argumentos del mismo. Desde el punto de vista semántico, un término establece una relación entre el funtor y el conjunto de sus argumentos. Esta relación, junto con el término al que se asocia, viene fijada por el programador y constituirá el elemento básico dotado de significado de un programa, es lo que denominaremos *átomo*.

Ejemplo 21 *Consideraremos tres términos diferentes, uno asociado al concepto de* "número natural"*, otro al concepto de* "número siguiente a otro" *y el tercero asociado a la noción de* "suma de dos números para obtener un tercero"*. Resumimos el conjunto de firmas y relaciones definidas en el diagrama:*

Sintaxis	**Semántica**	**Firma**
natural(X)	*X es un número natural*	*natural/1*
suc(X)	*El número que sigue a X*	*suc/1*
suma(X,Y,Z)	*Z = X + Y*	*suma/3*

términos que estructuralmente se corresponden con el conjunto de árboles que sigue



[3] llamado el *funtor principal del término.*

[4] ello dependerá tan solo del contexto expresado por el programa lógico.

y que determinarán los procesos de unificación y resolución que introduciremos más tarde.

Intuitivamente, el lector puede pensar en este tipo de términos como si de un *tipo registro* en los lenguajes imperativos se tratara. De hecho, el principio de gestión es el mismo con la salvedad de que aquí la recuperación y asignación de lo que serían los campos del registro no se realiza mediante funciones de acceso, sino utilizando el concepto de *unificación* que introduciremos más adelante.

La unidad fundamental de un programa lógico es el *átomo*, un tipo especial de término compuesto, distinguido sólo por el contexto en el cual aparece en el programa[5]. A partir del concepto de átomo, definimos el de *literal*, que no es otra cosa que un átomo o su negación.

En este punto, podemos ya introducir la noción de *cláusula* como un conjunto de literales ligados por conectores y cuantificadores lógicos. En nuestro caso, nos limitaremos a un tipo especial de cláusulas denominadas de Horn. Una *cláusula de Horn* es una estructura compuesta por una *cabeza* y un *cuerpo*. La cabeza consiste o bien de un simple átomo, o bien está vacía. El cuerpo está formado por una secuencia de cero o más literales ligados mediante conectivas conjuntivas o disyuntivas. El final del cuerpo suele indicarse mediante un punto, y todas las variables de la cláusula están cuantificadas universalmente y son locales a la misma. Cuando la cabeza está vacía se dice que la cláusula es una *pregunta*.

La cabeza y el cuerpo de una misma cláusula están separados por un símbolo de implicación lógica cuya representación notacional puede variar según el autor considerado. Esto es, en general representaremos una cláusula en la forma :

$$P \ :- \ Q_1, Q_2, \ldots, Q_n.$$

que podemos leer bien *declarativamente*:

"P es cierto si Q_1 es cierto, Q_2 es cierto, ..., y Q_n es cierto."

bien *operacionalmente*:

"Para satisfacer P, es necesario satisfacer los átomos Q_1, Q_2, ..., y Q_n."

donde el átomo de cabeza y los literales de la cola suelen referirse como *objetivos*. En este contexto, un *programa lógico* se define simplemente como una secuencia de cláusulas de Horn.

Ejemplo 22 *El siguiente programa define recursivamente el conjunto de los números naturales:*

```
natural(0).
natural(suc(X)) :- natural(X).
```

[5] esto es, expresa una relación lógica.

Se puede observar que hemos utilizado los términos `natural` *y* `suc` *previamente definidos en el ejemplo 21, donde resulta obvio que* `suc` *es una simple facilidad notacional que no expresa ningún tipo de relación lógica, en contra de lo que ocurre con el predicado* `natural/1`*. Por tanto,* `suc` *refiere a una función y no a un predicado, que si sería el caso de* `natural`*. Declarativamente, podemos leer ambas cláusulas en la forma:*

"El cero es natural."
"El número `X` es natural, si `X` también lo es."

Observar que la notación considerada para los números naturales es puramente simbólica y basada en la función `suc`*, dada por la tabla 1 que sigue:*

Natural	**Notación**
0	`0`
1	`suc(0)`
2	`suc(suc(0))`
⋮	⋮
n	`suc(`$\overset{n}{\ldots}$`(suc(0))...)`

Table 1. Notación para la representación de los números naturales.

Ello nos permitirá, más tarde, explotar plenamente la potencia deductiva de la unificación y de la resolución. Como preguntas para este programa podemos considerar, por ejemplo, las siguientes:

```
:- natural(suc(0)).      :- natural(0).
:- natural(suc(suc(X))). :- natural(X).
```

cuya interpretación declarativa es, respectivamente, la siguiente:

¿ es suc(0) natural ? ¿ es 0 natural ?
¿ existe X, tal que suc(suc(X)) sea natural ? ¿ existe X, tal que sea natural ?
Observar que las dos últimas preguntas tienen una infinitud de posibles respuestas.

3 Semántica

Una vez introducida brevemente la sintaxis y el léxico utilizados en programación lógica, necesitamos ahora dotar de significado a los programas. Lo haremos desde dos puntos de vista complementarios, aunque no los únicos posibles, las semánticas declarativa y procedural. En el primer caso centraremos nuestra atención en los programas como teorías en lógica de primer orden; mientras que en el segundo lo haremos en relación a la forma en que explícitamente se

resuelven las preguntas. Formalmente, necesitamos primero introducir algunos conceptos previos, básicos para el entendimiento de la mecánica del motor de un intérprete lógico y que determinan inequívocamente el estilo de programación propio de los lenguajes de este tipo.

Definición 31 *Una* sustitución *es una lista de pares* (variable, término)*. Utilizaremos la notación*

$$\Theta \equiv \{X_1 \leftarrow T_1, \ldots, X_n \leftarrow T_n\}$$

para representar la sustitución Θ que asocia las variables X_i a los términos

$$T_i,\ i \in \{1, 2, \ldots, n\}$$

La aplicación de una sustitución Θ a un término lógico T será denotada $T\Theta$ y se dirá que es una instancia *del término T.*

3.1 Semántica declarativa

Ahora ya podemos considerar una definición formal recursiva de la *semántica declarativa* de las cláusulas, que sirve para indicarnos aquellos objetivos que pueden ser considerados ciertos en relación a un programa lógico:

> *"Un objetivo es cierto si es una instancia de la cabeza de alguna de las cláusulas del programa lógico considerado y cada uno de los objetivos que forman el cuerpo de la cláusula instanciada son a su vez ciertos."*

En este punto, es importante advertir que la semántica declarativa no hace referencia al orden explícito de los objetivos dentro del cuerpo de una cláusula, ni al orden de las cláusulas dentro de lo que será el programa lógico. Este orden será, sin embargo, fundamental para la semántica operacional de PROLOG. Ello es la causa fundamental de las divergencias entre ambas semánticas en las implementaciones prácticas de dicho lenguaje y fuente de numerosos errores de programación, que además no siempre son fáciles de detectar.

Las sustituciones son utilizadas en programación lógica para, mediante su aplicación a las variables contenidas en una cláusula, obtener la expresión de la veracidad de una relación lógica particular a partir de la veracidad de una relación lógica más general incluida en el programa. Más formalmente, dicho concepto se conoce con el nombre de *unificación*, que pasamos a definir inmediatamente.

Definición 32 *Un* unificador *de dos términos lógicos T_1 y T_2 es una sustitución Θ, tal que $T_1\Theta = T_2\Theta$. Cuando al menos existe un unificador para dos términos lógicos T_1 y T_2, existe un unificador particular Θ llamado el* unificador más general (umg) *de T_1 y T_2, tal que para cualquier otro unificador Θ', existe una sustitución σ tal que $\Theta' = \Theta\sigma$.*

Intuitivamente, el $umg(T_1, T_2)$ representa el número mínimo de restricciones a considerar sobre dos términos para hacerlos iguales.

Ejemplo 31 *Dados los términos lógicos:*

$$T_1 = f(X, g(X, h(Y)))$$
$$T_2 = f(Z, g(Z, Z))$$

un conjunto de posibles unificadores es el siguiente:

$$\Theta_1 \equiv \{X \leftarrow h(1), Z \leftarrow h(1), Y \leftarrow 1\}$$
$$\Theta_2 \equiv \{X \leftarrow Z, Z \leftarrow h(Y)\}$$
$$\Theta_3 \equiv \{X \leftarrow Z, Z \leftarrow h(1), Y \leftarrow 1\}$$

donde el $umg(T_1, T_2)$ *es* Θ_2.

Desde un punto de vista práctico, el umg nos permitirá efectuar nuestro razonamiento lógico conservando la mayor generalidad posible en las conclusiones. Es por ello que el umg es el unificador utilizado en el proceso de demostración que constituye la interpretación lógica. Para su obtención, la mayoría de los dialectos PROLOG actuales consideran el algoritmo de Robinson [**?**], que pasamos a describir inmediatamente.

Algoritmo 31 *Sean* T_1 *e* T_2 *dos términos lógicos, el siguiente seudocódigo describe el* método de unificación de Robinson*, calculando el* $umg(T_1, T_2)$.

Entrada: *Dos términos lógicos* T_1 *y* T_2.

Salida: $\Theta = umg(T_1, T_2)$*, si existe; en otro caso* `fail`.

inicio
$\Theta := \emptyset$;
`meter (`$T_1 \equiv T_2$`, Pila) ;`
mientras `Pila` $\neq \emptyset$ **hacer**
  $(X \equiv Y)$ `:= sacar(Pila) ;`
  caso
    X `e` Y `constantes,` $X = Y$: **nada**
    $X \notin Y$, X `variable` : `sustituir (`X, Y`);`
      : `añadir (`$\Theta, X \leftarrow Y$`)`
    $Y \notin X$, Y `variable` : `sustituir (`Y, X`);`
      : `añadir (`$\Theta, Y \leftarrow X$`)`
    $(X \leftarrow f(X_1, \ldots, X_n))$ **y**
    $(Y \leftarrow f(Y_1, \ldots, Y_n))$: **para** i `:=` n **hasta** *1* **paso** *-1* **hacer**
      `meter(`$X_i \leftarrow Y_i$`, Pila)`
      fin para
    sino : **devolver** `fail`
  fin caso
fin mientras ;
devolver Θ
fin

donde $X \notin Y$ expresa que el valor de X no forma parte de la estructura correspondiente a Y. Es lo que se conoce como test de ciclicidad en unificación[6]. *La función* `meter`$(T_1 \equiv T_2$, `Pila`$)$ *introduce la ecuación lógica $T_1 \equiv T_2$ y la función* `sacar(Pila)` *extrae la ecuación lógica $X \equiv Y$ de la estrucura* LIFO[7] `Pila`. *La función* `sustituir`(X, Y) *sustituye la variable X por Y en la pila y en la sustitución Θ. Finalmente, la función* `añadir`(Θ, σ) *añade al unificador Θ la sustitución σ.*

Para ilustrar el algoritmo anterior, consideramos dos ejemplos. Uno de ellos, el último, incluye un ciclo de unificación y, por tanto la unificación debiera abortarse.

Ejemplo 32 *Calcularemos el umg para los términos T_1 y T_2 del anterior ejemplo 31, y al que denominaremos Θ. Lo haremos de forma incremental, estudiando cada una de las ramas de los términos implicados, dos a dos, y actualizando continuamente el contexto de trabajo con los nuevos valores asignados a las variables durante el proceso. En la misma línea, usaremos una pila como estructura de priorización del tratamiento de unos u otros términos en cada momento. Así, podemos sintetizar el cálculo del $umg(T_1, T_2)$ como sigue:*

$$\underset{\boxed{T_1 = T_2}}{\Theta \equiv \{\}} \vdash \underset{\boxed{g(Z,Z) = g(X,h(Y))}}{\boxed{X = Z}}^{\Theta \equiv \{\}} \vdash \underset{\boxed{g(Z,Z) = g(Z,h(Y))}}{\Theta \equiv \{X \leftarrow Z\}} \vdash \underset{\boxed{Z = h(Y)}}{\boxed{Z = Z}}^{\Theta \equiv \{X \leftarrow Z\}} \vdash$$

$$\vdash \underset{\boxed{Z = h(Y)}}{\Theta \equiv \{X \leftarrow Z\}} \vdash \underline{\underline{\Theta \equiv \{X \leftarrow Z,\ Z \leftarrow h(Y)\}}}$$

Donde $\vdash$ simplemente es una conectiva que enlaza las diferentes fases del proceso. Esto es, el resultado final es el dado por la sustitución

$$\Theta \equiv \{X \leftarrow Z, Z \leftarrow h(Y)\}$$

Ejemplo 33 *Para ilustrar el efecto de las estructuras circulares en el proceso de unificación, consideraremos ahora los dos términos lógicos siguientes:* `igual(X,X)` *e* `igual(Y,f(Y))`. *Veremos que nuestro algoritmo de unificación, si no se aplica el test de ciclicidad, entra en un ciclo sin fin. En efecto, la secuencia de configuraciones en la pila que sirve de sustento al algoritmo, es la que sigue:*

$$\underset{\boxed{igual(X,X) = igual(Y,f(Y))}}{\Theta \equiv \{\}} \vdash \underset{\boxed{X = f(Y)}}{\boxed{X = Y}}^{\Theta \equiv \{\}} \vdash \underset{\boxed{Y = f(Y)}}{\Theta \equiv \{X \leftarrow Y\}} \vdash \text{fail}$$

[6] *occur-check* en terminología anglosajona.
[7] **L**ast **I**nput **F**irst **O**utput.

Observemos que en la última de las configuraciones Y aparece en f(Y). *Si continuamos con el proceso ocurrirá que sustituiremos toda ocurrencia de* Y *por* f(Y)*, y como consecuencia obtendremos:*

$$X \leftarrow Y \leftarrow f(Y) \leftarrow f(f(Y)) \leftarrow f(f(f(Y))) \leftarrow f(f(f(f(Y)))) \leftarrow \ldots$$

Esto es, la unificación ha entrado en un ciclo.

En este punto podemos ya introducir la noción de resolución que proporciona capacidad deductiva a la programación lógica. Lo haremos sobre la base del algoritmo más popular en las implementaciones PROLOG, la resolución SLD[8] [**?**]

Algoritmo 32 *El siguiente seudocódigo describe el* método de resolución SLD.

Entrada: *Un programa lógico P y una pregunta Q.*

Salida: *$Q\Theta$, si es una instancia de Q deducible a partir de P; en otro caso* `fail`.

inicio
`Resolvente :=` $\{Q\}$;
mientras `Resolvente` $\neq \emptyset$ **hacer**
  $A \in$ `Resolvente` ;
  si $\exists\, P : -Q_1, \ldots, Q_n$ *tal que* $\exists\, \Theta = mgu(A, P)$ **entonces**
    `borrar (Resolvente,` A`) ;`
    `añadir (Resolvente,` $Q_1\Theta, \ldots, Q_n\Theta$`) ;`
    `aplicar (`Θ`, Resolvente)`
  sino devolver `fail`
  fin si
fin mientras ;
devolver Θ
fin

donde la función `borrar(Resolvente, A)` *borra el objetivo* `A` *de* `Resolvente`*, mientras que la función* `añadir(Resolvente,` $Q_1\Theta, \ldots, Q_2\Theta$`)` *añade los objetivos indicados a* `Resolvente`*. La función* `aplicar(`Θ`, Resolvente)` *aplica sobre el conjunto de objetivos de* `Resolvente` *la restricción representada por la sustitución Θ. En general consideraremos que una* resolvente *contiene en cada instante el conjunto de objetivos a resolver.*

El proceso de resolución SLD puede representarse en forma arboresecente, de forma que la existencia de una rama cuya última hoja es la resolvente vacía se traduce en la existencia de una instancia que es consecuencia lógica del programa, esto es, de una respuesta.

[8] por **S**electing a literal, using a **L**inear strategy and searching the space of possible deductions **D**epth-first.

Definición 33 *Sean P un programa lógico y Q una pregunta, entonces el* árbol de resolución *correspondiente se construye en la forma siguiente:*

1. *El nodo raíz del árbol es una resolvente con Q.*
2. *Seleccionado un objetivo A en la resolvente, la construcción del árbol en cada nodo continúa como sigue:*
   (a) *Si A se puede unificar con la cabeza de cada una de las cláusulas*

$$\begin{array}{l} P_1 : -Q_1^1, \ldots, Q_1^{m_1}. \\ P_2 : -Q_2^1, \ldots, Q_2^{m_2}. \\ \vdots \\ P_n : -Q_n^1, \ldots, Q_n^{m_n}. \end{array}$$

mediante las sustituciones $\{\Theta_i,\ i=1,2,\ldots,n\}$, entonces construimos n ramas para ese nodo. En cada una escribimos la nueva resolvente derivada de la cláusula C_i y de la sustitución Θ_i, renombrando[9] *automáticamente las variables de los nuevos objetivos incorporados a la resolvente. Los átomos de la cola de la cláusula unificada se añaden a la resolvente, mientras que el objetivo a resolver es eliminado de la misma.*

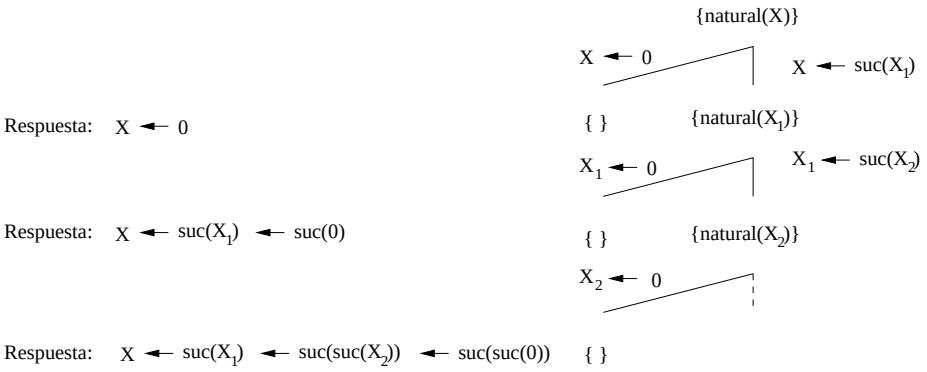


Fig. 1. Resolución para la pregunta `:- natural(X)`.

   (b) *Si el átomo seleccionado no se puede unificar con la cabeza de ninguna de las cláusulas de P, se paraliza la construcción de la rama del árbol y se devuelve* `fail` *como respuesta.*
   (c) *Si la resolvente resultante está vacía es porque no queda ningún objetivo por resolver. En este caso, también se paraliza la construcción de dicha*

[9] dicho renombramiento puede efectuarse simplemente mediante la incorporación de un subíndice a los nombres de las variables. Es importante indicar que esta técnica es necesaria para indicar qué variables de igual nombre en cláusulas distintas son diferentes, evitando de este modo confusiones en las futuras sustituciones.

rama y se devuelve `true`*, además del resultado de la composición de las sustituciones* $\tilde{\Theta}_1\tilde{\Theta}_2 \ldots \tilde{\Theta}_l$ *que han sido consideradas desde la raíz hasta el nodo en cuestión. Ello constituye la respuesta a la pregunta* Q*.*

Es importante observar aquí que, ni la resolución SLD ni el árbol asociado determinan orden alguno en el tratamiento de las cláusulas del programa lógico, ni tampoco en relación a los objetivos dentro de las propias cláusulas.

Ejemplo 34 *Para ilustrar el concepto de árbol de resolución, retomamos el programa del ejemplo 22,y suponemos que nuestra pregunta es la dada por:*

$$:-\texttt{natural(X)}.$$

que declarativamente se interpretaría como:

"¿ Existen valores para `X`, tal que `X` es natural ?"

un posible árbol de resolución asociado es el mostrado en la figura 1, pero también lo es el mostrado en la figura 2. Observar que las respuestas encontradas son las mismas, pero que los árboles son diferentes. Las líneas discontinuas indican que el árbol es, en cada caso, infinito.

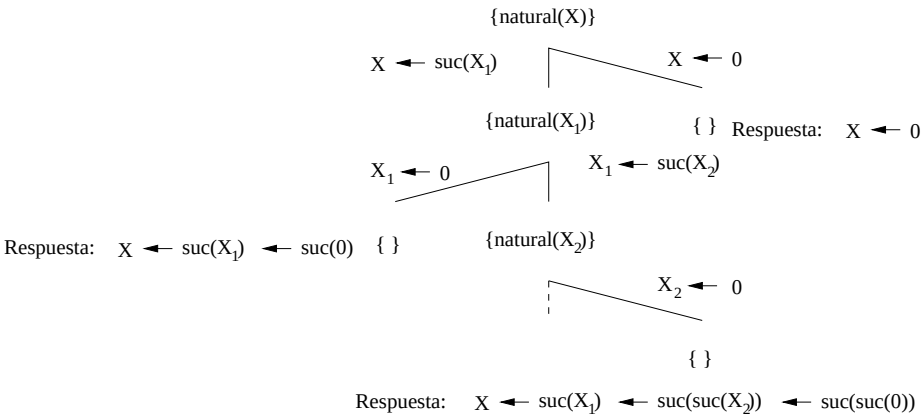


Fig. 2. Resolución alternativa para la pregunta `:- natural(X)`.

3.2 Semántica procedural

Aunque declarativamente el proceso de resolución pueda definirse en términos propios de la lógica de primer orden, expresando la elección de cláusulas y objetivos en términos de cuantificadores existenciales y universales, las implementaciones prácticas no tienen otro remedio que establecer órdenes de exploración del espacio de cálculo.

Definición 34 *Sean P un programa lógico y Q una pregunta, entonces el* árbol de resolución PROLOG *para el programa P, dada la pregunta Q, se construye de la misma forma que en la anterior definición 33, salvo que:*

1. *El objetivo a resolver es siempre el primer átomo A de la resolvente Q, si consideramos esta como una lista ordenada de izquierda a derecha.*
2. *En el proceso de unificación del objetivo a resolver, con las cabezas de las cláusulas del programa, se impondrá un orden de exploración de éstas que será de arriba hacia abajo. Este ordenamiento se trasladará en una exploración asociada de las ramas del árbol de izquierda a derecha, en profundidad.*
3. *Al añadir a la resolvente los átomos de la cola de la cláusula unificada con el objetivo a resolver, estos sustituyen la posición que tenía en la resolvente el objetivo resuelto, a la vez que conservan el orden local que mantenían en la cola de la cláusula.*

Ejemplo 35 *Retomando los términos del ejemplo 22, sólo el árbol de resolución para la pregunta:*

```
:- natural(X).
```

que se muestra en la figura 1 refleja la semántica operativa del intérprete lógico SLD*, no así la que se reflejaba en la figura 2.*

Al construirse el árbol de resolución en profundidad es necesario articular un protocolo para que, una vez agotada o truncada la exploración de una rama, podamos localizar la siguiente rama a explorar. Para ello remontaremos la estructura arborescente ya construida, revisitando cada nodo y aplicando el proceso siguiente:

- Si existe una rama no explorada por la derecha, esto es, queda alguna cláusula aplicable como alternativa a la anteriormente escogida para resolver el primer objetivo de la resolvente de ese nodo, entonces esa nueva posibilidad es estudiada.
- En otro caso, remontamos un nodo más en nuestra rama y recomenzamos el proceso para el mismo, hasta haber agotado todas las posibilidades. Si esto último ocurre devolvemos `fail` como respuesta.

en un proceso que denominamos *retroceso*[10].

Ejemplo 36 *Para ilustrar el concepto de retroceso, consideremos el siguiente programa, cuyo objeto es implementar el concepto de suma de números naturales:*

```
suma(0,N,N).
suma(suc(N_1),N_2,suc(N_3)) :- suma(N_1,N_2,N_3).
```

[10] *backtracking* en terminología anglosajona.

donde `suc(X)` *es la función ya considerada en el ejemplo 22, y* `suma(S_1,S_2,R)` *es la notación que determina la semántica declarativa del predicado* `suma/3`*, cuyo significado será:*

> "El resultado de la adición de los sumandos `S_1` y `S_2` es `R`."

a partir de la cual hemos construido las dos cláusulas del predicado `suma/3` *en la forma:*

> "La suma del cero y de un número cualquiera, es dicho número si éste es natural."
> "La suma del sucesor del número `N_1` y un número `N_2`, es el sucesor del número `N_3`, si `N_3` es el resultado de sumar `N_1` y `N_2`."

Entonces la pregunta `:- suma(X,Y,Z).` *se interpreta como:*

> "¿ Existen valores para las variables `X`, `Y` y `Z`; tales que `X + Y = Z`."

y el proceso de resolución debiera, por tanto, proveer la capacidad de cálculo para una infinitud de posibles soluciones. Ello implicará, en este caso, la aplicación de retrocesos sobre cada nodo del árbol de resolución, tal y como se ilustra en la figura 3, donde las respuestas se obtienen por encadenamiento de las sustituciones aplicadas y estas sustituciones aparecen asociadas a la rama correspondiente.

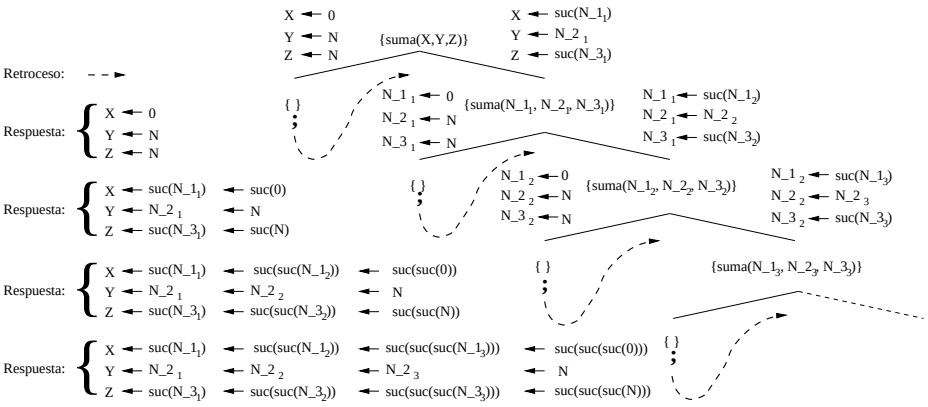


Fig. 3. Resolución para la pregunta `:- suma(X,Y,Z).`

Dado que todas las ramas se construyen con éxito, y por tanto ninguna falla, el retroceso se fuerza por el usuario, lo que habitualmente se realiza mediante la introducción de un carácter "`;`" desde el teclado.

3.3 Incongruencias declarativo/procedurales

La sutil diferencia entre la semántica declarativa y la procedural que hemos introducido para la implementación de la resolución SLD, introduce incongruencias que el programador deberá tener en cuenta para asegurar la corrección de su código. Más exactamente, el alejamiento de la semántica declarativa puede llevar a situaciones en las que un programa sea declarativamente correcto, pero no así desde el punto de vista procedural.

Al origen del problema situaremos las alteraciones que en el tratamiento de los objetivos de la resolvente introducen las implementaciones prácticas del algoritmo SLD. Más concretamente, nos referimos a la fijación de un orden de exploración en el árbol de resolución que rompe la dinámica de tratamiento de conjunto de las resolventes para trasladarlas a un protocolo LIFO[11], extremadamente eficaz desde el punto de vista computacional, pero que compromete la completud de la resolución.

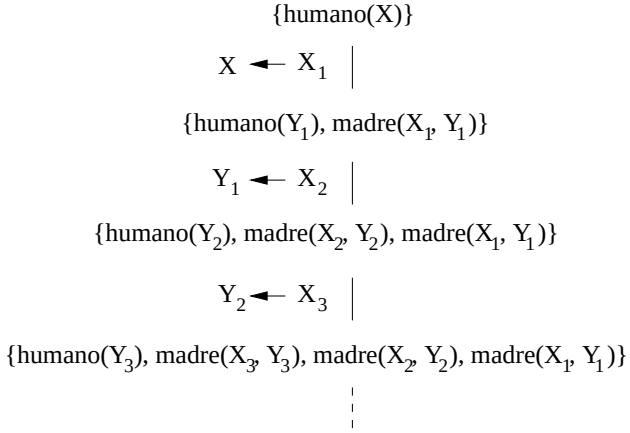


Fig. 4. Resolución para la pregunta `:- humano(X).`

Ejemplo 37 *Supongamos un conjunto de cláusulas que define la siguiente semántica declarativa:*

"Un individuo es humano si tiene una madre que es humana"

y que podríamos definir de la forma siguiente:

```
humano(X) :- humano(Y), madre(X,Y).
humano("Elena").
madre("Juan","Elena").
```

[11] por *Last Input First Output.*

donde `madre(X,Y)` *denota que la madre de* `X` *es* `Y`*, y* `humano(X)` *se interpreta como que* `X` *es humano. Consideremos en estas circunstancias la pregunta:*

```
:- humano(X).
```

cuya semántica declarativa sería:

" ¿Existen valores para la variable `X`, tal que `X` sea humano ?"

Resulta evidente que el programa refleja perfectamente la semántica declarativa considerada y, en consecuencia, el programa debiera ser correcto. En consecuencia, ante la pregunta:

```
:- humano(X).
```

y en función de la información disponible, debiéramos deducir que `"Juan"` *y* `"Elena"` *son humanos.*

Sin embargo, la semántica procedural considerada, nos lleva a construir el árbol de resolución de la figura 4, donde el proceso no lleva a respuesta alguna. El origen de tal comportamiento no es otro que el de la consideración del axioma `humano("Elena")` *en una posición no prioritaria y la introducción de una recursividad izquierda en la primera cláusula, incompatible con la construcción descendente del árbol. Consideremos, para comenzar, la siguiente variante del programa inicial, que mantiene intacta la semántica declarativa del mismo:*

```
humano("Elena").
humano(X) :- humano(Y), madre(X,Y).
madre("Juan","Elena").
```

lo que no varía la semántica declarativa, pero permite obtener una respuesta, asociada a la primera cláusula del predicado `humano/1`*, aún a pesar de la recursividad izquierda de la segunda, tal y como puede verse en la figura 5.*

Finalmente, consideremos ahora la siguiente variante del programa, en el que además hemos cambiado el orden, declarativamente irrelevante y que permite eliminar la recursividad izquierda antes comentada, de los objetivos de la cola en la cláusula recursiva del predicado `humano/1`*:*

```
humano("Elena").
humano(X) :- madre(X,Y), humano(Y).
madre("Juan","Elena").
```

tal y como podemos ver en la figura 6, en este caso si hemos obtenido todas las respuestas esperadas. Ello demuestra el impacto en la resolución tanto del orden de los objetivos en las resolventes como de las cláusulas del programa lógico.

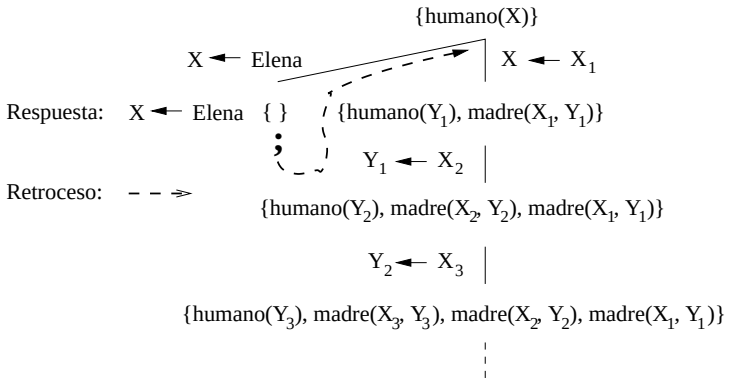


Fig. 5. Resolución alternativa para la pregunta `:- humano(X)`.

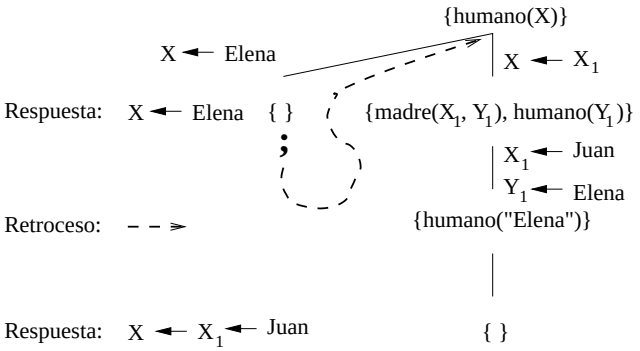


Fig. 6. Resolución alternativa para la pregunta `:- humano(X)`.

4 Control

Si en el paradigma imperativo la única referencia para la programación es la propia semántica procedural, en el caso lógico esto no ocurre. En particular, cualquier usuario que pretenda trasladar un programa imperativo en uno lógico, constatará la total ausencia de estructuras de control, omnipresentes en el primer caso.

4.1 El corte

El *corte* es el método más utilizado para establecer un control de la resolución en un programa lógico. Sin embargo, su utilización romperá la semántica declarativa del lenguaje, cercenando la potencia de cálculo derivadas de la unificación y resolución.

Formalmente, el *corte* es un predicado sin argumentos, que se verifica siempre y que se representa mediante la notación "`!`". Como efecto colateral, suprime en el árbol de resolución todas las alternativas que puedan quedar por explorar para los predicados que se encuentren entre la cabeza de la cláusula en la que aparece, y la posición en la que el corte se ha ejecutado.

Ejemplo 41 *Se trata de implementar la estructura de control por autonomasia, el* `if_then_else`*. Su semántica será la habitual. Así, interpretaremos la notación* `if(P, Q, R)` *en la forma:*

"Si `P` es cierto, entonces probar `Q`, sino probar `R`."

cuya implementación es la siguiente:

```
if(P,Q,R) :- P, !, Q.
if(P,Q,R) :- R.
```

El corte indica que una vez probado el objetivo `P`*, el único camino a seguir es el indicado por la primera cláusula, esto es, probar* `Q`*.*

4.2 El fallo

Este predicado se denota por `fail`, y como resultado de su ejecución se produce un fallo. Proceduralmente ello implica que la resolución se paraliza en la rama actual del árbol, forzándose un retroceso en busca de la siguiente rama a explorar.

Ejemplo 42 *Para ilustrar el concepto de fallo, consideramos la implementación de la relación* inferior que *en el conjunto de los números naturales:*

```
inferior(0,0) :- fail.
inferior(0, suc(Y)).
inferior(suc(X),suc(Y)) :- inferior(X,Y).
```

cuya semántica declarativa viene dada por:

"El cero no es inferior al cero"
"El cero es inferior a cualquier número que sea sucesor de otro"
"El sucesor de un natural `X` es inferior al sucesor de otronatural `Y`, si `X` es inferior a `Y`".

y para el que el cálculo de respuesta negativa para la pregunta:

```
:- inferior(suc(0), suc(0)).
```

se muestra en la figura 7.

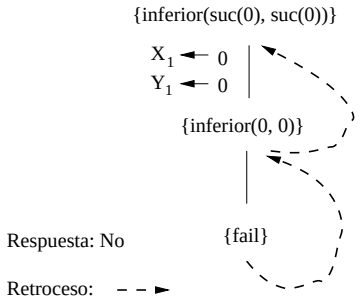


Fig. 7. Resolución para la pregunta `:- inferior(suc(0),suc(0))`.

Observar que en este caso el retroceso no es provocado por el usuario mediante la introducción de un "`;`" desde el teclado, sino forzado automáticamente al evaluarse el `fail`*. El hecho de que dicho retroceso sea infructuoso al no quedar ramas por explorar es lo que conlleva la respuesta negativa a la pregunta.*

4.3 La negación

La conjunción de los predicados corte y `fail` permiten la consideración de una forma de negación habitual en programación lógica. Es lo que se denomina la *negación por fallo*. Intuitivamente el principio de funcionamiento que la define es muy simple y puede resumirse en la siguiente semántica declarativa:

"La negación por fallo, `not(X)`*, de un predicado* `X` *es* `fail` *si podemos demostrar que* `X` *es cierto"*
" En otro caso, `not(X)` *es cierta."*

y aunque aparentemente tal definición se corresponde con la noción de negación lógica, esto no es exactamente así. Basta pensar que no poder demostrar la veracidad de algo, no quiere decir que sea falso, simplemente que hemos fallado en tal intento. De ahí la denominación de negación por fallo, lo que tendrá profundas implicaciones en la construcción de programas que la incluyan. Su implementación se resume en dos cláusulas:

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

lo que se corresponde exactamente con la semántica declarativa antes comentada. La combinación "`!, fail`" permite provocar el fallo, a la vez que el corte evita el retroceso sobre la segunda cláusula de la negación por fallo.

Ejemplo 43 *Para ilustrar tanto el funcionamiento de la negación, como de la combinación "*`!, fail`*", consideraremos un ejemplo entorno a la idempotencia de la negación, de hecho una de sus propiedades fundamentales. En concreto consideraremos las dos preguntas siguientes:*

1. *La primera será* `:- not(not(fail)).`*, cuya respuesta debiera ser* `fail`*.*
2. *La segunda será* `:- not(not(true)).`*, cuya respuesta debiera ser* `true`*.*

donde `true` *es un predicado habitualmente predefinido, que siempre es cierto. Para ello nos remitimos a la figura 8, donde con el fin de diferenciar los diferentes cortes implicados, nos referiremos a ellos mediante subíndices que nos permitan distinguirlos. De este modo, en los dos árboles de resolución mostrados podemos ver cortes "*$!_1$*" y "*$!_2$*". Tal como se puede comprobar, en ambos casos se alcanzan las respuestas esperadas.*

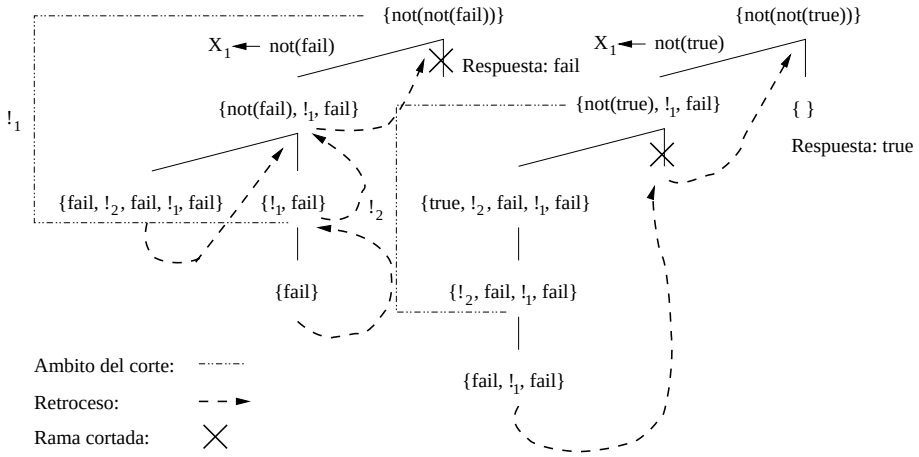


Fig. 8. Resolución para las preguntas `:- not(not(fail))` y `:- not(not(true))`.

5 Listas

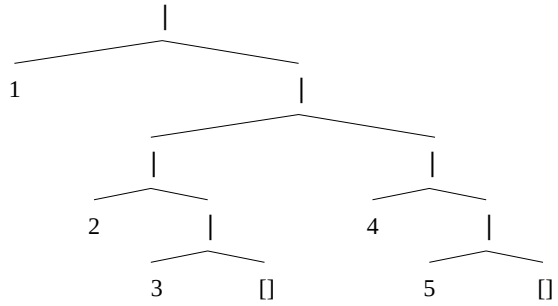
Las listas consituyen una estructura de programación básica en el paradigma lógico, y ello por su adaptabilidad a dos características fundamentales del mismo.

La primera al estilo de programación aquí requerido, netamente recursivo. La segunda a la unificación como mecanismo de gestión dinámico.

Una lista se representa por la sucesión de sus elementos separados por comas, y encerrada entre corchetes. Así, la lista cuyo primer elemento es `1`, cuyo segundo elemento es `2` y cuyo tercer elemento es `3`, se representará en la forma `[1,2,3]`.

La implementación física de la estructura de datos es un *doblete* cuyo primer elemento denominamos `car`, siendo el segundo el `cdr`. El `car` es el primer elemento de la lista y el `cdr` es lo que queda de la lista original una vez eliminado el primer elemento. Los dobletes conectan `car` y `cdr` mediante el conectivo `cons`, que habitualmente se representa mediante el símbolo "`|`".

Ejemplo 51 *Consideremos la lista de números naturales* `[1, [2, 3], 4, 5]`*, entonces su representación física se corresponde con el siguiente árbol binario:*



que podemos referir igualmente, entre otras, mediante las notaciones

$$[1 \mid [[2,3],4,5]] \qquad [1,[2,3] \mid [4,5]] \qquad [1,[2,3],4 \mid [5]]$$
$$[1,[2,3],4,5 \mid []] \qquad [1,[2 \mid [3]],4,5] \qquad [1,[2,3 \mid []],4,5]$$

donde "`[]`*" es el símbolo de fin de lista, comúnmente denominado* lista vacía*, pero que curiosamente no es una lista al carecer de* `car` *y* `cdr`*.*

Nuestra intención ahora será la de utilizar todo el potencial de la unificación como mecanismo para manejar la recursividad en programas cuya estructura básica de datos sean las listas.

Ejemplo 52 *Consideraremos la implementación de un predicado* `concat(L_1, L_2, R)` *describiendo la concatenación de las listas* `L_1` *y* `L_2` *para obtener la lista* `R`*. La implementación podría ser la siguiente:*

```
concat([],L,L).
concat([Car|Cdr],L,[Car|R]) :- concat(Cdr,L,R).
```

cuya semántica declarativa viene dada por:

> "La concatenación de la lista vacía con otra cualquiera, es esta última."
> "La concatenación de una lista `[Car | Cdr]` con otra lista `L` es el resultado de concatenar el `Car` al resultado `R` previamente obtenido de la concatenación de `Cdr` con `L`."

Para ilustrar la potencia de cálculo en este caso, consideraremos la pregunta:

```
:- concat(X, Y, Z).
```

cuyas respuestas con obviamente infinitas. En este sentido, la figura 9 muestra el árbol de resolución correspondiente, así como algunas de las respuestas que airosamente provee en este caso la resolución SLD.

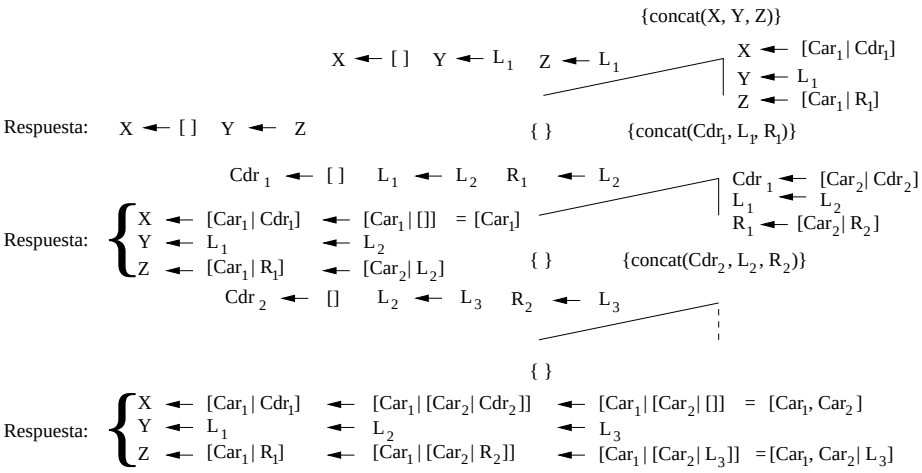


Fig. 9. Resolución para la pregunta `:- concat(X, Y, Z)`.

Observar en particular la técnica utilizada para introducir un elemento en cabeza de una lista en la segunda cláusula del predicado `concat/3`. *En efecto, se ha utilizado la unificación para insertar el* `Car` *al principio de la lista* `R`.

Ejemplo 53 *Nuestro objetivo ahora es la implementación del conocido algoritmo de ordenación de listas numéricas habitualmente conocido como* quicksort. *Técnicamente el método se sustancia en los siguientes pasos:*

1. *Elegir un elemento de la lista, al que denominaremos* pivote, *y que servirá como referencia comparativa para los demás. En nuestro caso, y para simplificar la exposición, el pivote será siempre el primer elemento de la lista a ordenar.*
2. *Elegido el pivote, dividimos la lista original en dos partes. Una, que situaremos a su izquierda, que contendrá a los elementos en la lista cuyo valor sea menor o igual al pivote. La otra, que situaremos a la derecha del pivote, contendrá a los elementos en la lista cuyo valor sea mayor al referido elemento.*
3. *El proceso se aplica ahora recursivamente sobre cada elemento, izquierdo y derecho, de la partición de la lista original hasta que las particiones se agoten. En ese momento la lista original estará totalmente ordenada.*

que implementaremos usando tres predicados diferentes:

- *El primero será nuestro predicado principal* `quicksort(L,R)`*, cierto cuando* `R` *sea el resultado de ordenar la lista numérica* `L` *mediante el algoritmo del* quicksort*.*
- *El segundo será el predicado* `partir(Pivot,L,MenIg,May)`*, cierto cuando* `MenIg` *y* `May` *sean respectivamente la lista de elementos menores o iguales y mayores que el número* `Pivot` *en la lista* `L`*.*
- *El tercero será el predicado* `concat/3`*, ya comentado en el ejemplo 52.*

a partir de los cuales construimos el siguiente conjunto de cláusulas que reflejan el algoritmo declarativo antes descrito:

```
quicksort([],[]).
quicksort([Car|Cdr],R) :- partir(Car,Cdr,Izq,Der),
                          quicksort(Izq,Izq_ordenada),
                          quicksort(Der,Der_ordenada),
                          concat(Izq_ordenada,[Car|Der_ordenada],R).

partir(Pivot,[],[],[]).
partir(Pivot,[Car|Cdr],[Car|Izq],Der) :- Car =< Pivot,
                                         !,
                                         partir(Pivot,Cdr,Izq,Der).
partir(Pivot,[Car|Cdr],Izq,[Car|Der]) :- partir(Pivot,Cdr,Izq,Der).

concat([],L,L).
concat([Car|Cdr],L,[Car|R]) :- concat(Cdr,L,R).
```

Observar el uso del corte en la segunda cláusula del predicado `partir/4`*. Ello asegura que la tercera cláusula del referido predicado sólo se ejecutará cuando el test* `Car > Pivot` *sea cierto. De prescindirse del uso del corte, sería necesario incluir explítamente dicho test como primer elemento de la cola esa tercera cláusula en* `partir/4`*.*

6 Evaluación perezosa

Tal y como ya hemos comentado, las incongruencias declarativo/procedurales habitualmente asociables a PROLOG, tienen su origen en la alteración de las directrices de evaluación de la resolución SLD original y, en particular, en el tratamiento de términos no instanciados. A este respecto, los intérpretes lógicos introducen el concepto de *evaluación perezosa* como mecanismo de control de la evaluación, sujeta a condiciones fijadas por el propio usuario.

6.1 El predicado `freeze/2`

La función del predicado `freeze(Variable, Objetivo)` es demorar la evaluación de `Objetivo` hasta que `Variable` esté instanciada. De esta manera,

podemos considerar dos casos en la evaluación de este predicado. Cuando `Variable` está instanciada, `Objetivo` se evaluará normalmente. En caso contrario el objetivo de la resolvente actual correspondiente a la instanciación del término `freeze(Variable, Objetivo)` se retira y su evaluación queda pendiente. De este modo, el intérprete continúa con la evaluación del resto de objetivos, hasta que `Variable` haya sido por fin instanciada. Es entonces cuando `Objetivo` vuelve a la cabeza de la resolvente para ser evaluado.

Esta forma de control en la ejecución de objetivos en función de la disponibilidad de sus instanciaciones permite al programador asegurar ésta última, evitando las incongruencias antes referidas.

Ejemplo 61 *Un primer ejemplo ilustrativo de la situación nos lo proporciona el predicado extralógico* `is`*, que activa la evaluación de los operadores aritméticos clásicos en* PROLOG*. En particular, su uso exige la instanciación previa de todas y cada una de las variables implicadas en la expresión aritmética. Por tanto, por ejemplo, cuando queremos calcular el doble de un número mediante la cláusula:*

```
doble(X,Y) :- Y is X*2.
```

el valor de `X` *debe ser conocido previamente, lo que podemos asegurar mediante* `freeze/2` *en la forma:*

```
evaluar(X,Y,Z):- ..., freeze(X, doble(X,Y)), ...
```

En este contexto, dos son los ejemplos paradigmáticos en el control de la evaluación, justamente referidos a la resolución de las incongruencias declarativo/procedurales típicas de PROLOG. Se trata del tema de la recursividad izquierda y de los problemas planteados por el uso de la negación por fallo. Ilustraremos ambas situaciones mediante ejemplos.

Ejemplo 62 *Vamos a retomar el ejemplo 37 en el que se describía un posible programa para definir a un individuo como humano y, en concreto la primera versión propuesta:*

```
humano(X) :- humano(Y), madre(X,Y).
humano("Elena").
madre("Juan","Elena").
```

donde la existencia de recursividad izquierda frustraba el cálculo de cualquier respuesta a la pregunta `:- humano(X).`*, llevando al programa a una derivación sin fin de resolventes cada vez mayores. En aquella ocasión la solución pasaba por la eliminación directa de dicha recursividad. Sin embargo, podemos considerar otra alternativa el uso de* `freeze/2` *para asegurar que los objetivos de tipo* `humano(X)` *se evalúen sobre variables ya instanciadas,en la forma:*

```
humano(X) :- freeze(Y, humano(Y)), madre(X,Y).
humano("Elena").
madre("Juan","Elena").
```

lo que evita la entrada en el lazo recursivo, al obligar a evaluar primero `madre(X,Y)`.

Ejemplo 63 *Supongamos que queremos diferenciar un conjunto de comidas respecto a la temperatura a la que se consumen, esto es, en calientes o frías. Para ello, construímos el siguiente programa:*

```
caliente("sopa").
caliente("asado").
fria(Comida):- not(caliente(Comida)).
```

que declarativamente pueden interpretarse como:

> "La sopa es una comida caliente"
> "El asado es una comida caliente"
> "Una comida está fría, si no está caliente".

aunque sabemos que la última cláusula, operacionalmente, tiene una interpretación sensiblemente diferente. A saber:

> "Una comida está fría, si no podemos probar que está caliente".

Para poner en evidencia las incongruencias declarativo/procedurales a las que puede dar lugar una mala gestión de la negación, introduciremos el siguiente predicado `igual/2`*, que especifica la igualdad de dos términos cuando éstos se pueden unificar:*

```
igual(X,X).
```

En este contexto, podemos preguntar si una ensalada es una comida fría, entre otras, de las siguientes dos maneras:

```
:- fria("ensalada").
:- fria(X), igual("ensalada",X).
```

Ambas preguntas tienen la misma semántica declarativa, y, a pesar de ello, las respuestas que obtendremos, serán distintas: `true` *y* `fail`*, respectivamente. La diferencia fundamental entre ambos casos es que, en el primero, la variable afectada por la negación,* $\texttt{Comida}_1$ *está instanciada al valor* `ensalada` *en el momento de la negación. Esto es, conocemos perfectamente lo que estamos negando, situación en la que la negación por fallo garantiza su equivalencia con la negación lógica. Ello permite deducir si, en efecto, la ensalada es una comida caliente según los hechos del programa, tal y como se muestra en la figura 10.*

Sin embargo, en el caso de la segunda pregunta, tal y como se puede ver en la figura 11, ocurre exactamente lo contrario. Esto es, la variable `Comida` *no está instanciada cuando evaluamos la negación. Como resultado, obtenemos una respuesta negativa, totalmente incongruente.*

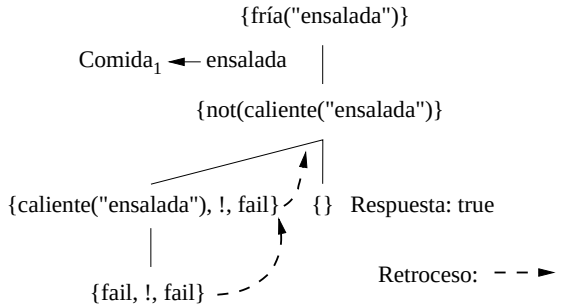


Fig. 10. Resolución para la pregunta `:- fria("ensalada")`.

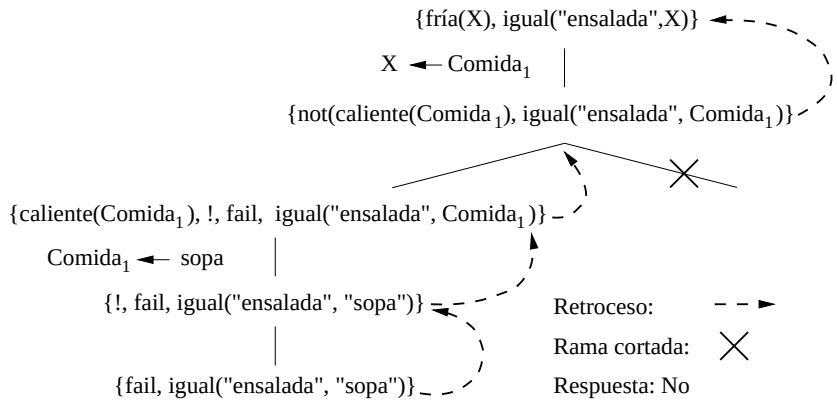


Fig. 11. Resolución para la pregunta `:- fria(X), igual("ensalada",X)`.

Para solventar el problema lo que haremos será retrasar la evaluación de la negación hasta el momento en el que todas las variables implicadas en ésta puedan ser conocidas. Esto es, hasta el momento en el que la negación por fallo garantice un comportamiento lógico congruente, tarea para la que echaremso mano del predicado `freeze/2` *en la forma:*

```
caliente("sopa").
caliente("asado").
fria(Comida):- freeze(Comida, not(caliente(Comida))).
```

obteniendo como resultado la respuesta correcta, tal y como se muestra en la figura 12.

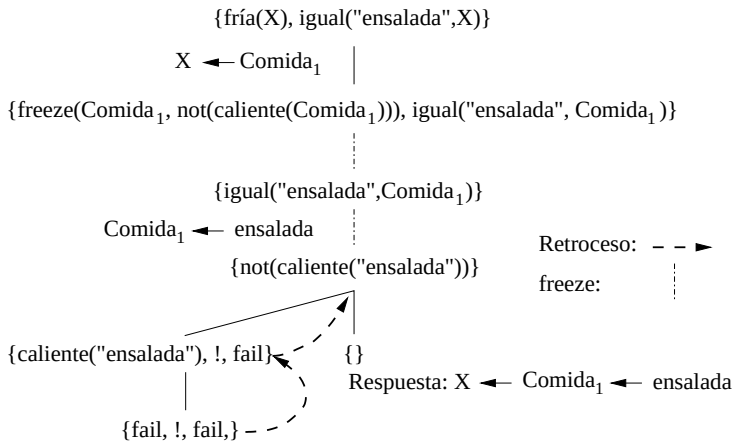


Fig. 12. Resolución para `:- fria(X), igual("ensalada",X)` con `freeze/2`.

6.2 El predicado `when/2`

Existen dos posibilidades al evaluar `when(Condición, Objetivo)`. Si `Condición` se verifica, se ejecuta `Objetivo`. En caso contrario, el átomo se retira de la resolvente y `Objetivo` sólo se ejecutará cuando al fin se verifique `Condición`. A este respecto, `Condición` suele construirse a partir de alguno de los siguientes predicados, algunos extralógicos, habituales en PROLOG:

- `?=(X,Y)`: se verifica si `X` e `Y` son idénticos o no pueden unificar.
- `nonvar(X)`: se cumple si `X` está instanciada.
- `ground(X)`: se verifica si `X` está instanciada a un término sin variables no instanciadas.
- `(Cond1, Cond2)`: conjunción (AND) de las condiciones `Cond1` y `Cond2`.
- `(Cond1; Cond2)`: disyunción (OR) de las condiciones `Cond1` y `Cond2`.

En esencia, al igual que `freeze/2`, el predicado `when/2` demora la ejecución de un objetivo, aunque en este caso en función de condiciones más complejas. De hecho, `freeze/2` puede definirse a partir de `when/2`, en la forma:

```
freeze(Variable,Objetivo) :- when(nonvar(Variable),Objetivo).
```

de modo que podemos utiliar `when/2` para solucionar el mismo tipo de anomalías de la negación que `freeze/2`.

Ejemplo 64 *El problema que nos ocupa ahora es decidir qué coche, de una determinada marca y modelo, queremos comprar. La única restricción que se nos impone es que la marca no sea europea. Consideremos, por ejemplo, el siguiente programa, que pretende resolver esta situación:*

```
coche("Dodge","Caliber").
coche("Opel","Corsa").
coche("Toyota","Prius").
europea("Opel").
comprar(Marca, Modelo) :- not(europea(Marca)), coche(Marca,Modelo).
```

cuya sem ántica declarativa viene dada por:

> "El Dodge Caliber es un coche"
> "El Opel Corsa es un coche"
> "El Toyota Prius es un coche"
> "Compra un modelo de coche si su marca no es europea".

Si ahora interrogamos a este programa con la pregunta `:- comprar(X.Y).`*, nos encontramos con que la respuesta es* `fail`*, a pesar de que modelos de marcas no europeas están listados como hechos.*

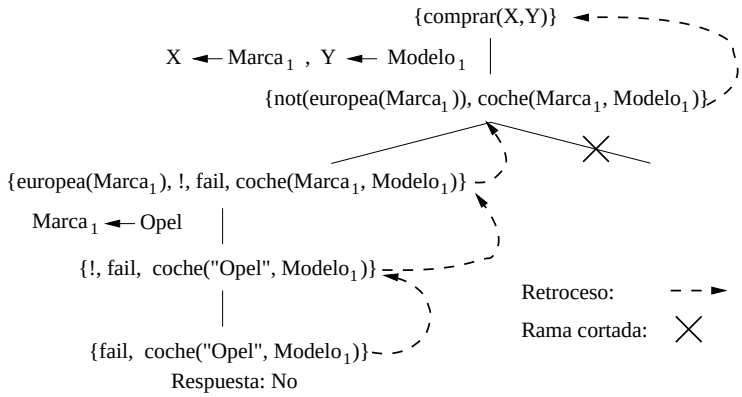


Fig. 13. Resolución para la pregunta `:- comprar(X,Y).`

La incongruencia tiene su origen en la no instanciación de la variable `Marca` *antes de evaluar la negación, como se puede ver en la figura 13. Para subsanar ese problema debemos asegurarnos que que* `Marca` *esté instanciada en el momento de la negación, lo que podemos conseguir a través de* `when/2`*, en la forma:*

```
comprar(Marca, Modelo) :- when(nonvar(Marca), not(europea(Marca))),
                          coche(Marca,Modelo).
```

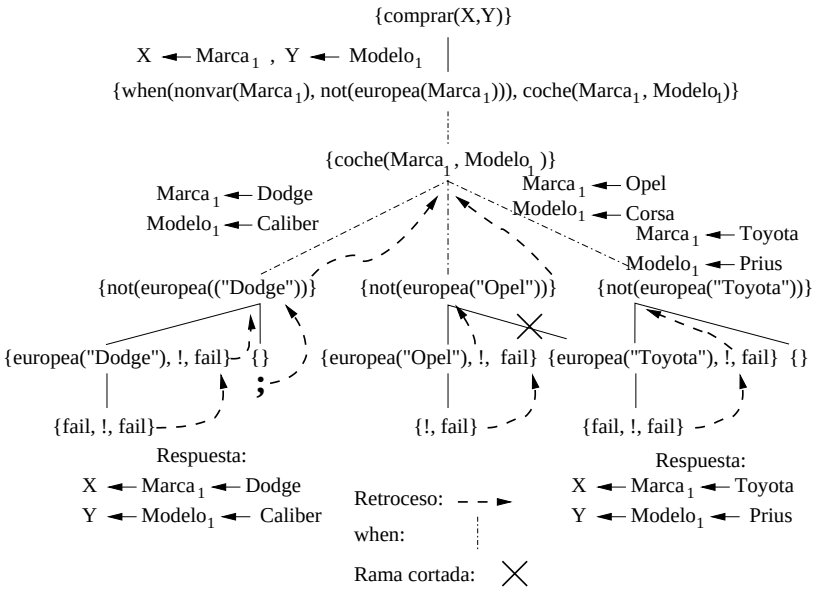


Fig. 14. Resolución para la pregunta `:- comprar(X,Y)` con `when`.

cuyo árbol de resolución para la pregunta referida se muestra en la figura 14. Podemos observar que, al forzar la evaluación de `coche/2` *antes de la negación, se prueba cada modelo de coche por separado. Como resultado, el* `Opel Corsa` *queda descartado por la negación, al constar en los hechos del programa que* `Opel` *es una marca europea.*

7 Operadores y capacidad expresiva

La posibilidad de definir dinámicamente nuevos operadores en programación lógica dota a este paradigma de una potencia expresiva que no encontraremos en otros lenguajes y que, de hecho, permitirá acercar la utilización y comprensión de programas a los usuarios no expertos de una forma efectiva, sobre la base de un acercamiento real al lenguaje natural que sirve de vínculo de comunicación entre humanos.

Formalmente, distinguimos tres tipos de operadores en relación a la posición que guardan respecto a sus argumentos: *infijos*, *prefijos* y *sufijos*; según aparezcan entre aquellos, delante de ellos o después. También podemos clasificarlos en relación al número de argumentos que manejan. Así podemos, por ejemplo, hablar de operadores *binarios* cuando poseen dos argumentos; y de *unarios* si sólo tienen uno.

Desde un punto de vista analítico, un operador necesita fijar tres parámetros para que su uso en programación no conlleve la introducción de ambigüedades en la interpretación del código:

1. La *notación* que lo representará.
2. La *prioridad* de evaluación en relación a otros operadores. Ello eliminará cualquier posibilidad de ambigüedad en la interpretación de expresiones que incluyan diferentes operadores.
3. La *asociatividad* en su evaluación. Ello eliminará cualquier posibilidad de ambigüedad en la interpretación de expresiones que incluyan al mismo operador repetido varias veces. Distinguiremos tres tipos de asociatividad: izquierda, derecha e inexistente.

En PROLOG, estos tres parámetros se definen por parte del usuario a través del predicado `op/3`, cuya sintaxis es la siguiente:

```
op(Prioridad, Asociatividad, Lista_de_notaciones)
```

y que ahora pasamos a describir detalladamente. En cuanto a la prioridad, ésta se designa mediante un número en el intervalo $[1, 1200]$, siendo más alta cuanto más pequeño es el número. Así, por ejemplo, podríamos asumir perfectamente que la suma tuviera una prioridad de 500, y la multiplicación una de 400.

En relación a la asociatividad, el lenguaje considera tres tipos distintos de operador binario: `xfx`, `xfy` y `yfx`. También permite la consideración de operadores unarios, cuya asociatividad se expresa de forma diferente según sean prefijos o sufijos. En el caso de los prefijos consideraremos asociatividades del tipo `fx` y `fy`, en el de los sufijos serán del tipo `xf` e `yf`. En cualquier caso, la interpretación de los valores `y`, `x` y `f` es común a todos ellos:

- `f` representa al operador binario, respecto al cual intentamos definir la asociatividad.
- `x` indica que dicha subexpresión debe tener una prioridad estrictamente menor[12] que la del funtor `f`.
- `y` indica que dicha subexpresión puede tener una prioridad menor o igual que la del funtor `f`.

considerando que la prioridad en la evaluación de una expresión viene dada por la prioridad de su funtor principal, esto es, del funtor que aparece en la raíz de su árbol de análisis sintáctico. En definitiva, lo que estamos diciendo es que

[12] es decir, un indicativo de prioridad mayor.

un operador del tipo `xfy` tendrá una asociatividad por la derecha, mientras que un operador de tipo `yfx` tendrá un asociatividad por la izquierda. El valor `xfx` indicará la ausencia de asociatividad. Así, por ejemplo, `op(500,yfx,[+])` declararía al operador "`+`" como asociativo por la izquierda y con una prioridad de 500. Además, dado que se trata de un simple predicado de interfaz con el sistema, no cabe esperar respuestas congruentes a preguntas del tipo:

```
:- op(P,yfx,[+]).    :- op(500,A,[+]).    :- op(500,yfx,N).
```

Ya en el caso unario, todo operador `op` declarado de tipo `fy` o `yf` tiene carácter asociativo, por lo que es válido escribir[13] `op op ... op operando` puesto que la expresión `op operando` tiene igual prioridad que `op` y en consecuencia puede ser utilizada como operando de este último operador. En cambio, un operador `op` definido como `fx` o `xf` no puede ser utilizado asociativamente, puesto que una expresión como `op op ... op operando` no será válida al no ser `op operando` de menor prioridad que `op`, lo cual implica que no puede ser utilizada como operando de este último operador.

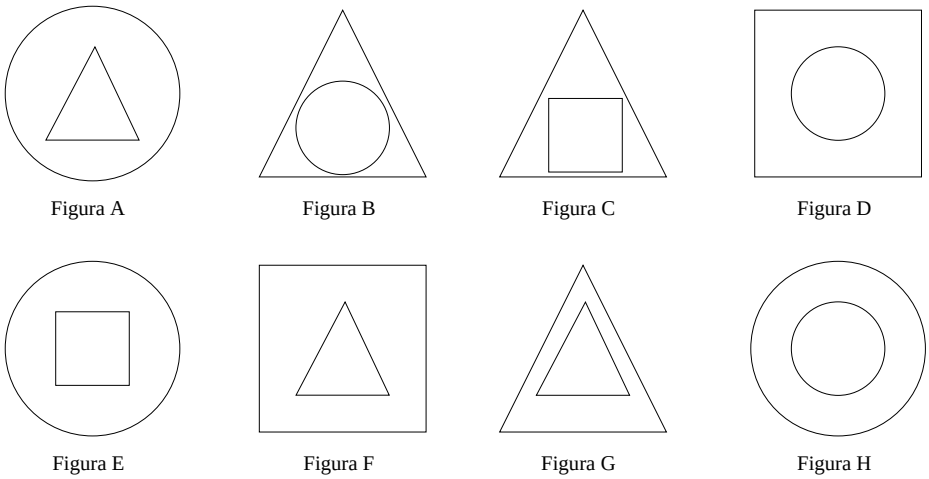


Fig. 15. Un conjunto de figuras para el problema de la analogía

Finalmente, es posible definir más de un operador con el mismo nombre, siempre que sean de diferente tipo. Es lo que se conoce como *sobrecarga* de un operador. El intérprete PROLOG identifica el operador concreto que se está utilizando mediante el examen de los operandos en análisis sintáctico. La utilización de operadores sobrecargados debe restringirse en lo posible.

En cualquier caso, la definición de nuevos operadores es una tarea delicada que debe tener muy en cuenta el comportamiento deseado de la nueva estructura

[13] en este caso suponemos que `op` es de tipo `fy`, es decir, un operador unario prefijo.

en relación al comportamiento de los operadores ya existentes. Exije, por tanto, un conocimiento profundo del problema cuya solución intentamos determinar.

Ejemplo 71 *Para ilustrar las ventajas e la definición y uso de operadores en programación lógica, consideraremos un problema clásico en inteligencia artificial, el* Problema de la Analogía.

Esencialmente se trata de establecer analogías entre formas geométricas. Como ejemplo concreto, a partir del conjunto de formas representadas en la figura 15, podemos considerar la existencia de algún tipo de relación entre ellas.

Figura	**Descripción**	**Figura**	**Descripción**
A	`triángulo dentro_de círculo`	B	`círculo dentro_de triángulo`
C	`cuadrado dentro_de triángulo`	D	`círculo dentro_de cuadrado`
E	`cuadrado dentro_de círculo`	F	`triángulo dentro_de cuadrado`
G	`triángulo dentro_de triángulo`	H	`círculo dentro_de círculo`

Table 2. Tabla de nombres para el problema de la analogía

Nuestro objetivo será, a partir de una relación entre dos objetos y un tercero, el de encontrar el análogo a este tercer objeto según la relación establecida entre los dos primeros. Declarativamente, la semántica del problema podría expresarse como sigue:

> "La figura `A` `es_a` la figura `G`, `como` la figura `D` `es_a` la figura `H`, `mediante` una relación de tipo `inversión`."

donde hemos subrayado los elementos que permiten ligar las relaciones y que, en nuestro código, estarán al origen de la definición de diversos operadores que introduciremos más tarde.

Para resolver el problema, la forma de proceder es sistemática. Primero escogeremos una notación adecuada para referirnos a las figuras, una notación que no queremos sea meramente nominativa sino también descriptiva. Ello nos permitirá extraer información relativa a las relaciones entre los componentes de las figuras mediante unificación para, luego, determinar las posibles relaciones con otras. De este modo, asociaremos a cada una de los elementos considerados en la figura 15, las denominaciones que mostramos en la tabla 2, lo que justifica la introducción del siguiente operador que,para nosotros, tendrá la mayor prioridad de los definidos en el código:

```
op(200,xfy,[dentro_de])
```

y consideraremos además los conectivos antes introducidos en la expresión declarativa del problema:

```
op(300,xfy,[es_a])   op(400,xfy,[como])   op(500,xfy,[mediante])
```

lo que nos permitirá ligar las relaciones de analogía existentes entre los elementos de la figura 15, y que denominaremos `igualdad`, `inversión`, `interior` *y* `contorno`. *De este modo podremos trasladar casi textualmente nuestro código al lenguaje natural utilizado por los humanos.*

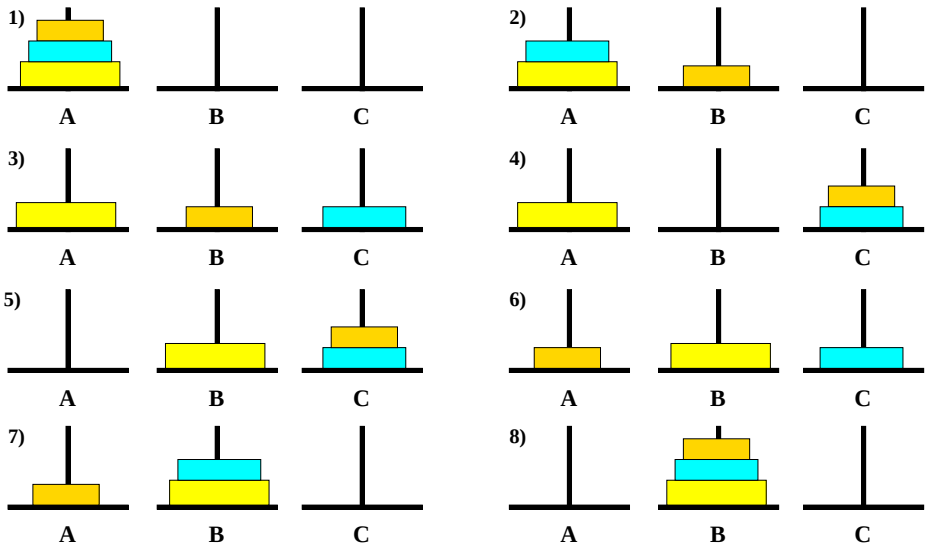


Fig. 16. Resolución de las Torres de Hanoi con tres discos.

Es importante observar que hemos definido `mediante` *como un operador con menor prioridad que* `como`, *éste a su vez menos prioritario que* `es_a`, *y éste a su vez como un operador con menor prioridad que* `dentro_de`. *Ello es imprescindible para la buena marcha del programa, puesto que queremos que* `mediante` *se evalúe más tarde que* `como`, *éste más tarde que* `es_a`, *y éste a su vez más tarde que* `dentro_de` *en las expresiones en las que los operadores aparezcan juntos. En este caso las asociatividades son irrelevantes ya que estos operadores nunca aparecen repetidos en secuencia*[14]. *Ahora ya podemos escribir nuestro programa:*

```
:- op(200,xfy,[dentro_de]).
:- op(300,xfy,[es_a]).
:- op(400,xfy,[como]).
:- op(500,xfy,[mediante]).

X es_a Y como Z es_a W mediante Relación :-
          figura(X), figura(Y),
```

[14] observar que, por ejemplo, si las figuras consideradas tuvieran al menos tres formas imbricadas, al menos el operador `dentro_de` si estaría en situación de definir una asociatividad real.

```
          verifican(X, Y, Relación),
          figura(Z), figura(W),
          verifican(Z, W, Relación).

verifican(Figura_1 dentro_de Figura_2,
          Figura_1 dentro_de Figura_2,igualdad).
verifican(Figura_1 dentro_de Figura_2,
          Figura_2 dentro_de Figura_1,inversión).
verifican(Figura_1 dentro_de Figura_2,
          Figura_1 dentro_de Figura_3,interior).
verifican(Figura_1 dentro_de Figura_2,
          Figura_3 dentro_de Figura_2,contorno).

figura(triángulo dentro_de círculo).
figura(círculo dentro_de triángulo).
figura(cuadrado dentro_de triángulo).
figura(círculo dentro_de cuadrado).
figura(cuadrado dentro_de círculo).
figura(triángulo dentro_de cuadrado).
figura(triángulo dentro_de triángulo).
figura(círculo dentro_de círculo).
```

que, por ejemplo, ante la pregunta:

```
:- X es_a triángulo dentro_de círculo
   como
   cuadrado dentro_de círculo es_a Y
   mediante
   Relación.
```

proporciona nada menos que 17 respuestas diferentes.

8 Aprendizaje automático

Tal y como ya hemos comentado, una característica propia de la programación lógica es su capacidad natural para la manipulación simbólica mediante la unificación. Si a ello sumamos el hecho de que en un programa lógico no existe una diferencia real entre datos y objetos, siendo todos gestionados de idéntica forma en razón de su estructura arborescente, tendremos la herramienta perfecta para abordar uno de los problemas recurrentes en inteligencia artificial, la interpretación de programas que se modifican a si mismos.

En nuestro contexto, modificar un programa supone introducir o eliminar cláusulas de su base de datos. A este respecto, los intérpretes de programación lógica proveen un conjunto de primitivas que permiten su puesta en marcha:

– El predicado `asserta/1` se verifica siempre, siendo su efecto colateral el de introducir al principio de nuestro programa la cláusula que le sirve de argumento. El predicado `assertz/1` realiza, sin embargo, la introducción de la nueva cláusula al final del programa.
– En cuanto al predicado `retract/1`, éste se verifica también siempre, siendo su efecto colateral en esta ocasión el de retirar de la base de datos de nuestro programa la cláusula que le sirve de argumento. La búsqueda se realiza comenzando por el principio del programa.

Este tipo de predicados resulta de utilidad en programas que ofrezcan al usuario la posibilidad de considerar dinámicamente la introducción de restricciones, o en aquellos en los que la repetición de esquemas recursivos múltiples aconsejen la generación de nuevas cláusulas que eviten la multiplicación de cálculos que previamente hemos realizado.

Ejemplo 81 *Las* Torres de Hanoi *son un conocido juego, en el que se parte de un escenario con los siguientes elementos:*

– *Tres palos en posición vertical, que denominamos* `A`*,* `B` *y* `C`*.*
– *Un conjunto de discos de diferente diámetro, que pueden ser insertados en cualquiera de los palos.*
– *Inicialmente todos los discos están insertados en el palo* `A`*, ordenados de mayor a menor diámetro, comenzando por la posición más baja.*

tal y como se muestra en la figura 16. Definido el escenario, el juego consiste en mover los discos desde el palo inicial `A`*, a un palo de destino* `B`*, sirviendo el tercer palo* `C` *de paso temporal de los discos en su movimiento. En todo momento debe satisfacerse, cualquiera que sea el palo considerado, la condición siguiente:*

> "Sobre un disco cualquiera solamente pueden colocarse discos de menor diámetro."

La complejidad del problema es de 2^N *movimientos para* N *discos. Por tanto, en el caso* $N = 3$ *mostrado en la figura 16, el número de movimientos sería del orden de* $2^3 = 8$*, lo que refiere a una complejidad exponencial. En cuanto al algoritmo, éste se resume en el siguiente conjunto de cláusulas:*

1. *Mover* `N-1` *discos de* `A` *hacia* `C`*, utilizando* `B` *como palo intermedio. Con ello dejamos en* `A` *un único disco: el que estaba inicialmente en la base.*
2. *Mover el disco situado en* `A` *a* `B`*.*
3. *Mover los* `N-1` *discos situados en* `C` *a* `B`*, usando* `A` *como palo intermedio.*

lo que claramente establece una estrategia doblemente recursiva en la que los pasos 1. y 3. son análogos, tal y como se reflejará en la implementación, para la cual consideraremos el predicado `hanoi/5`*, cuya semántica declarativa viene expresada por:*

> "`hanoi(N, A, B, C, Movs)` es cierto sii `Movs` es la serie de movimientos a realizar para trasladar `N` discos desde el palo `A` al `B`, tomando el palo `C` como paso intermedio"

obteniendo como resultado posible, el programa siguiente:

```
:- op(600,yfx,a).

concat([],L,L) :- !.
concat([Car|Cdr],L,[Car|R]) :- concat(Cdr,L,R).

hanoi(1,A,B,_,[A a B]).
hanoi(N,A,B,C,Movs) :- N > 1, N1 is N - 1,
                       hanoi(N1,A,C,B,Movs_1),
                       hanoi(N1,C,B,A,Movs_2),
                       concat(Movs_1,[A a B|Movs_2],Movs).
```

en el que hacemos uso de la concatenación de listas a través del predicado auxiliar `concat/3`*, y del operador infijo "*`a`*" para facilitar la lectura de la salida. Evidentemente, la primera cláusula de* `hanoi/5` *describe el caso trivial, cuando en la columna* `A` *de partida sólo hay un disco.*

La idea ahora no es otra que la de aprovechar la analogía ya descrita entre las dos llamadas recursivas de `hanoi/5`*, de manera a evitar* de facto *los cálculos que corresponderían a la segunda de esas llamada. El nuevo código sería entonces:*

```
:- op(600,yfx,a).

concat([],L,L) :- !.
concat([Car|Cdr],L,[Car|R]) :- concat(Cdr,L,R).

hanoi(1,A,B,_,[A a B]).
      hanoi(N,A,B,C,Movs):- N>1, N1 is N-1,
                            hanoi(N1,A,C,B,Movs_1),
                            asserta((hanoi(N1,A,C,B,Movs_1):-!)),
                            hanoi(N1,C,B,A,Movs_2),
                            retract((hanoi(N1,A,C,B,Movs_1):-!)),
                            concat(Movs_1,[A a B|Movs_2],Movs).
```

de esta forma, no sólo evitamos la multiplicación de cálculos mediante el uso de `asserta/1`*, sino que una vez estamos seguros de que éstos han cumplido su función, las cláusulas introducidas dinámicamente a tal efecto son eliminadas.*

8.1 Ejercicios propuestos

1. Implementar un predicado `fib(X,Y)` que verifique:
   (a) Que `Y` es el valor de la función de Fibonacci sobre `X`.
   (b) Que evita el cálculo reiterado de valores `fib(X',Y')` donde `X'< X`.
   (c) Que deje intacto el programa inicial.

La función de Fibonacci se define por:

$$\texttt{Fibonacci(X)} = \begin{cases} 0 & \textbf{si } \texttt{X=0} \\ 1 & \textbf{si } \texttt{X=1} \\ \texttt{Fibonacci(X-1) + Fibonacci(X-2)} & \textbf{en otro caso} \end{cases}$$

2. Implementar un predicado `mult(F_1,F_2,R)`, tal que se verifique cuando `R` sea el resultado del producto de los factores `F_1` y `F_2`.
   Ejemplo: La respuesta, expresada usando la notación `suc/1` para representar los naturales, a `:- mult(suc(suc(0)),suc(suc(suc(0))),X).`, es `X=suc(suc(suc(suc(suc(suc(0))))))`.
3. Implementar un predicado `invertir(L,R)` que sea cierto cuando `R` es la lista resultante de invertir el orden de los elementos en la lista `L`.
   Ejemplo: La respuesta a la pregunta `:- invertir([1,a],X).`, es `X=[a,1]`.
4. Implementar un predicado `longitud(L,R)` que sea cierto cuando `R` sea la longitud de la lista `L`.
   Ejemplo: La respuesta a la pregunta `:- longitud([1,a],X).`, es `X=2`.
5. Implementar un predicado `elimina(E,L,R)` que se verifique cuando `R` es la lista resultante de eliminar todas las apariciones del elemento `E` en la lista `L`.
   Ejemplo: La respuesta a la pregunta `:- elimina(1,[1,a,1,2],X).`, es `X=[a,2]`.
6. Implementar un predicado `assertb` con la funcionalidad de `asserta`, pero que en caso de retroceso elimine del universo del discurso la cláusula antes introducida.
7. Implementar un predicado `extrae(L,P,E)` que se verifique cuando `E` es el elemento de la lista `L` en la posición `P`.
   Ejemplo: La respuesta a la pregunta `:- extrae([1,a,1,2],2,X).`, es `X=a`.
8. Implementar un predicado `exp(X,Y,Z)` que se verifique cuando $\texttt{X}^{\texttt{Y}} = \texttt{Z}$.
9. Implementar un predicado `inserta(L,E,P,R)` que se verifique sii `R` es la lista resultante de insertar el elemento `E` en la posición `P` de la lista `L`.
   Ejemplo: La respuesta a la pregunta `:- inserta([1,2,3,4],a,2,X).`, es `X=[1,a,2,3,4]`.
10. Implementar un predicado `ordenar(L,R)`, que ordene por inserción la lista numérica `L`, para calcular la lista ordenada `R`.
    Ejemplo: La respuesta a la pregunta `:- ordenar([4,1,2,3],X).`, es `X=[1,2,3,4]`.
11. Implementar un predicado `aplanar(L,R)` que aplane la lista `L`, eliminando imbricaciones sobre sus elementos e instanciando la respuesta en `R`.
    Ejemplo: La respuesta para `:- aplanar([4,1,[2],[3,[4,5]],X).`, es `X=[4,1,2,3,4,5]`.
12. Resolver el problema de cripto-aritmética definido por la operación numérica siguiente:

```
  SEND
+ MORE
------
 MONEY
```

Esto es, implementar un programa que asigne los valores posibles a las letras en juego de manera que la suma sea posible. Suponer que los dígitos asignados a letras diferentes son asimismo diferentes.

13. Implementar un predicado `comprimir(L,C)` tal que `C` es la lista resultado de comprimir la lista `L`, al eliminar repeticiones consecutivas de un elemento dado.
    Ejemplo: La respuesta a `:- comprimir([a,a,b,c,c,a,a,d,e,e,e],X)` sería `X = [a,b,c,a,d,e]`.
14. Implementar un predicado `codificar(L,R)` tal que `R` es la lista resultante de codificar la lista `L` de tal forma que una sucesión de elementos repetidos consecutivos en `Lista` se sustituye por un par de la forma `[Longitud, Elemento]` con `Longitud` la longitud de la serie de repeticiones consecutivas de `Elemento`.
    Ejemplo: La respuesta a `:- codificar([a,a,b,c,c,a,a,d,e,e,e,],X).`, sería `X = [[2,a],[1,b],[2,c],[2,a],[1,d][3,e]]`.
15. Implementar un predicado `multiplicar_n(L,N,R)`, tal que `R` es la lista obtenida a partir de la lista `L` al repetir `N` veces cada elemento en `L`.
    Ejemplo: La respuesta a `:- multiplicar_n([a,b,c],3,X).`, sería `X = [a,a,a,b,b,b,c,c,c]`.
16. Implementar un predicado `extraer_n(L,P_1,P_2,R)`, tal que `R` es la lista resultante de extraer de la lista `L` la sublista formada por los elementos situados entre las posiciones `P_1` y `P_2`.
    Ejemplo: La respuesta a `:- extraer_n([a,b,c,d,e,f,g,h,i,k],3,7,X).`, es `X= [c,d,e,f,g]`.
17. Supongamos que un conjunto se representa mediante una lista en la que no aparezcan elementos repetidos. En estas condiciones, implementar los siguientes predicados:
    (a) `conjunto(X)`, que sea cierto si `X` es un conjunto, esto es, una lista sin elementos repetidos.
    (b) `intersección(C_1,C_2,R)`, tal que `R` es el resultado de la intersección de los conjuntos `C_1` y `C_2`.
    (c) `unión(C_1,C_2,R)`, tal que `R` es el resultado de la unión de los conjuntos `C_1` y `C_2`.
    (d) `cartesiano(C_1,C_2,R)`, tal que `R` es el resultado del producto cartesiano de los conjuntos `C_1` y `C_2`.
    (e) `dif_simétrica(C_1,C_2,R)`, tal que `R` es el resultado de la diferencia simétrica de los conjuntos `C_1` y `C_2`. La diferencia simétrica de dos conjuntos es su unión, menos su intersección.