

Tema 4: Paradigma lógico

Lenguajes, Tecnologías y Paradigmas de Programación

Indice



- 1 Introducción a la Programación Lógica
- 2 Sintaxis de los programas lógicos
- 3 El modelo de computación de la programación lógica
- 4 Algunas cuestiones prácticas

Objetivos



- ▣ Analizar el modelo de computación lógico: **invertibilidad de definiciones, variables lógicas, indeterminismo**, etc.
- ▣ Comprender el **paso de parámetros bidireccional** y su implementación a través del mecanismo de **unificación**.
- ▣ Entender el principio de **resolución** y las diferentes reglas de computación y **estrategias de búsqueda** aplicables.
- ▣ Saber resolver sencillos problemas en el paradigma lógico.

Un ejemplo



- Los caballeros de la mesa cuadrada y sus locos seguidores (Monty Python and the Holy Grail) (1975) <http://www.youtube.com/watch?v=T6YAyNwyDG4>

Un ejemplo

□ Solución Prolog

```
bruja(X):-arde(X),mujer(X).  
arde(X):-madera(X).
```

```
madera(X):-flota(X).  
madera(puente_de_madera).  
piedra(puente_de_piedra).
```

```
flota(pan).  
flota(manzana).  
flota(salsa_verde).  
flota(ganso).
```

```
flota(X):-mismo_peso(ganso,X).
```

```
mismo_peso(ganso,la_mujer_de_la_escena). /*observacion*/  
mujer(la_mujer_de_la_escena). /*observacion*/
```



```
Last login: Mon Dec 10 16:08:00 on ttys000
millenium:~ mramirez$ cd /Users/mramirez/Documents/DOCENCIA/LTP/TEORIA/Tema\ 4/2
012-13
millenium:2012-13 mramirez$ swipl
% library(swi_hooks) compiled into pce_swi_hooks 0.00 sec, 2,284 bytes
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 5.10.4)
Copyright (c) 1990-2011 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- [bruja].
% bruja compiled 0.00 sec, 2,248 bytes
true.

?- bruja(Quien).
Quien = la_mujer_de_la_escena .

?- 
```

Algunas características distintivas



- Uso de la lógica como lenguaje de programación
- Variables lógicas
 - ▣ Extracción de respuestas
 - ▣ Inversibilidad de definiciones
 - ▣ Indeterminismo

Uso de la lógica como lenguaje de programación

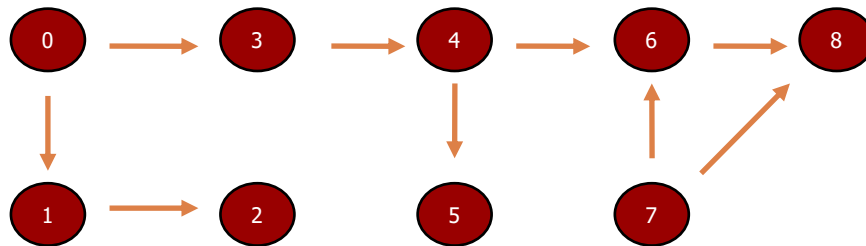
- ▣ La programación lógica surge de la idea **revolucionaria** de usar ***la lógica como lenguaje de programación.***
- ▣ Escribir un programa lógico consiste en expresar una relación (o conjunto de relaciones) utilizando una notación lógica basada en **la lógica de predicados.**
- ▣ La idea esencial del paradigma lógico es la de **COMPUTACIÓN como DEDUCCIÓN** frente a la noción más estándar de **COMPUTACIÓN como CÁLCULO.**

Uso de la lógica como lenguaje de programación

PROGRAMA

Expresar el conocimiento sobre el problema \Rightarrow

ESCRIBIR SENTENCIAS LÓGICAS



arco(0,3). arco(3,4). arco(4,6). arco(6,8).

arco(0,1). arco(1,2). arco(4,5).

arco(7,6). arco(7,8).

conectado(X,Y) :- arco(X,Y)

conectado(X,Y) :- arco(X,Z), conectado(Z,Y).

Uso de la lógica como lenguaje de programación

PROGRAMA

Expresar el conocimiento sobre el problema ⇒

ESCRIBIR SENTENCIAS LÓGICAS

EJECUCIÓN DEL PROGRAMA

*Plantear el problema a resolver ⇒ HACER DEDUCCIONES via
CONSULTAS*

```
arco(0,3).   arco(3,4).   arco(4,6).   arco(6,8).  
arco(0,1).   arco(1,2).   arco(4,5).  
arco(7,6).   arco(7,8).  
conectado(X,Y) :- arco(X,Y).  
conectado(X,Y) :- arco(X,Z), conectado(Z,Y).
```

¿están conectados 0 y 8?
¿están conectados 4 y 7?

```
?- conectado(0,8).  
yes  
?- conectado(4,7).  
no
```

Variables Lógicas

- ▣ Las variables del programa son incógnitas a despejar (variables matemáticas como las de una ecuación).
- ▣ Las fórmulas que constituyen el programa están cuantificadas universalmente (implícitamente).

```
conectado(X,Y) :- arco(X,Z),  
                  conectado(Z,Y).
```



```
 $\forall X,Y,Z$ (conectado(X,Y) :- arco(X,Z),  
          conectado(Z,Y))
```



```
 $\forall X,Y$ (conectado(X,Y) :-  $\exists Z$ (arco(X,Z),  
                               conectado(Z,Y)))
```

Extracción de Respuestas

- ▣ Las variables de las consultas están cuantificadas existencialmente.

```
?- conectado(X,Y).  
X=0  
Y=1
```

LECTURA: ¿Existen X Y tales que
conectado(X,Y)
es cierto con respecto al programa?



El mecanismo usado para probar el objetivo
es constructivo: si tiene éxito proporciona el valor de los
individuos X e Y desconocidos

Esto constituye la salida o respuesta a la consulta

Inversibilidad

- Los argumentos de un predicado pueden ser tanto de entrada como de salida.

`member(H,[H | L]).`

`member(H,[X | L]) :-member(H,L).`

- Verifica si un elemento está en una lista: `member(2,[1,2])`
- Devuelve todos los elementos de una lista: `member(X,[1,2])`
- Genera todas las posibles listas que contienen a un elemento dado:

`member(1,L)`

Indeterminismo

- Una consulta puede tener varias respuestas que el intérprete obtiene explorando exhaustivamente el espacio de computaciones.

?- member(X,[1,2,a]) .

Respuesta 1: X=1

Respuesta 2: X=2

Respuesta 3: X=a

2. Sintaxis de los programas lógicos:

Términos

- Globalmente, los datos de un programa lógico se denominan términos y pueden ser:
 - ▣ variables
 - Prolog: deben comenzar por mayúscula. La variable anónima se representa por “_”
 - Ej: X, Y, AreaDelCuadrado, Resultado
 - ▣ constantes
 - Prolog: numéricas y simbólicas (deben comenzar por minúscula o entre comillas simples)
 - Ej: 42, a, pedro, ‘Pedro’, ‘Hola Mundo’, minuto, segundo
 - ▣ estructuras de la forma $f(t_1, \dots, t_n)$ siendo f un nombre de función y t_1, \dots, t_n términos.
 - Prolog: f es un constructor de datos y debe comenzar por minúscula.
 - Ej: hora(H,M,S), nombre(‘Pedro’)

Listas (notación Prolog)

- Las listas son un tipo particular de datos contruidos a partir de:
 - la lista vacía: []
 - el constructor de listas [_|_]
- Ejemplos: [1|[2|[]]] (abreviado: [1,2])
[1|[2|X]] (equivalente a [1,2|X])
[1|2] ERROR
- Es similar a [] y (_:_) en Haskell

Sintaxis de los programas lógicos: Átomos

- Los átomos $p(t_1, \dots, t_n)$ están formados por
 - un símbolo de predicado p/n (aridad n)
 - Términos t_1, \dots, t_n
- Los átomos sirven para expresar propiedades o relaciones (p) sobre los datos (t_1, \dots, t_n)
- En Prolog, p es cualquier secuencia de caracteres empezando por minúscula.

Ejemplo: `arco(1,2)`

Sintaxis de los programas lógicos: programas Prolog

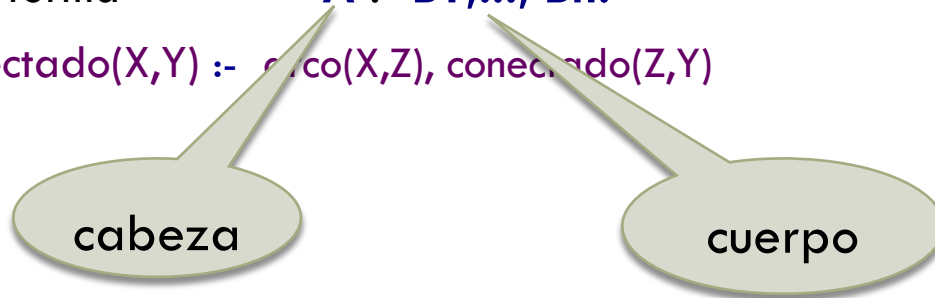
- Un programa lógico es un conjunto de sentencias/declaraciones que pueden ser de dos tipos: **hechos** o **reglas**, que se representan (siguiendo la sintaxis Prolog) como:

- HECHOS: tienen la forma **A.**

Ejemplo: arco(0,1).

- REGLAS: tienen la forma **A :- B1,..., Bn.** con $n > 0$

Ejemplo: conectado(X,Y) :- arco(X,Z), conectado(Z,Y)



donde A y cada Bi son átomos.

NOTA: los hechos pueden verse como reglas con el cuerpo vacío y escribirse como

A :- true.

Sintaxis de los programas lógicos: objetivos

- La llamada a ejecución de un programa lógico se llama objetivo y se escribe como una cláusula sin cabeza, es decir **?- B1,..., Bn.** con $n > 0$

Ejemplo: ?- conectado(X,Y).

Observa que, a diferencia de la PF los términos NO se evalúan ya que los objetivos se componen de átomos.

- Una cláusula sin cabeza ni cuerpo se denomina cláusula vacía y se representa como “?-”

Se usa como “centinela” para indicar que una computación ha terminado con éxito.

De Haskell a Prolog

- Tanto Haskell como Prolog son lenguajes basados en reglas. Sintácticamente, las principales diferencias son que en Prolog:
 - ▣ no hay funciones (solo procedimientos)
 - ▣ no se pueden anidar las llamadas a procedimiento

- Ejemplo:

`fibonacci(0) = 0`

`/* Haskell */`

`fibonacci(1) = 1`

`fibonacci(n) | n > 1 = fibonacci(n-1) + fibonacci(n-2)`

`fibonacci(0,0).`

`/* Prolog */`

`fibonacci(1,1).`

`fibonacci(N,M) :- N > 1, N1 is N-1, N2 is N-2, fibonacci(N1,F1), fibonacci(N2,F2), M is F1+F2.`

Las funciones se convierten en procedimientos
con un parámetro extra para devolver el
resultado

De Haskell a Prolog

- Tanto Haskell como Prolog son lenguajes basados en reglas. Sintácticamente, las principales diferencias son:
 - ▣ no hay funciones (solo procedimientos)
 - ▣ no se pueden anidar las llamadas a procedimiento

- Ejemplo:

`fibonacci(0) = 0`

`/* Haskell */`

`fibonacci(1) = 1`

`fibonacci(n) | n > 1 = fibonacci(n-1) + fibonacci(n-2)`

`fibonacci(0,0).`

`/* Prolog */`

`fibonacci(1,1).`

`fibonacci(N,M) :- N > 1, N1 is N-1, N2 is N-2, fibonacci(N1,F1), fibonacci(N2,F2), M is F1+F2.`

La guarda es una
relación más

De Haskell a Prolog

- Tanto Haskell como Prolog son lenguajes basados en reglas. Sintácticamente, las principales diferencias son:
 - ▣ no hay funciones (solo procedimientos)
 - ▣ no se pueden anidar las llamadas a procedimiento

- Ejemplo:

```
fibonacci(0) = 0                                /* Haskell */  
fibonacci(1) = 1  
fibonacci(n) | n>1 = fibonacci(n-1) + fibonacci(n-2)
```

```
fibonacci(0,0).                                /* Prolog */  
fibonacci(1,1).  
fibonacci(N,M) :- N>1, N1 is N-1, N2 is N-2, fibonacci(N1,F1), fibonacci(N2,F2), M is F1+F2.
```

No podemos anidar la resta y la llamada a fibonacci!
(básicamente, “is” evalúa la expresión de la derecha y se la “asigna” a la variable de la izquierda)

Ejemplos

Longitud de una lista:

□ En Haskell: $\text{length } [] = 0$

$\text{length } (x:xs) = \text{length } xs + 1$

□ En Prolog:

$\text{length } ([], 0).$

$\text{length } ([_ | T], N) :- \text{length}(T, N1),$
 $N \text{ is } N1 + 1.$

Ejemplos

Concatenación de listas

□ En Haskell: $[] ++ y = y$

$(x:xs) ++ y = x : (xs ++ y)$

□ En Prolog:

`append([], Y, Y).`

`append([X | R], Y, Z) :- append(R, Y, RY), Z = [X | RY].`

Ejemplos

Concatenación de listas

□ En Haskell: $[] ++ y = y$

$(x:xs) ++ y = x : (xs ++ y)$

□ En Prolog:

`append([], Y, Y).`

`append([X | R], Y, Z) :- append(R, Y, RY), Z = [X | RY].`

■ mejor:

`append([], Y, Y).`

`append([X | R], Y, [X | RY]) :- append(R, Y, RY)`

reemplazar el parámetro que
representa el resultado de la función
por la salida

Ejemplos

Último elemento de una lista

- En Haskell: $\text{last } [x] = x$

$$\text{last } (x:y:xs) = \text{last } (y:xs)$$

- En Prolog:

$\text{last}([X], X).$

$\text{last}([X,Y | XS], Z) :- \text{last}([Y | XS], Z).$

- pero también podemos usar append:

$\text{last}(XS, Z) :- \text{append}(YS, [Z], XS).$

Ejercicio

- Expresa mediante un programa Prolog la relación “antepasado”

X es un antepasado de Y si

X es el padre de Y

X es la madre de Y

X es el padre de Z y Z es un antepasado de Y

X es la madre de Z y Z es un antepasado de Y

Interpretación Operacional

CLÁUSULA DE PROGRAMA \equiv *DEFINICION DE UN MÉTODO O PROCEDIMIENTO*

$m(t_1, \dots, t_n) :- A_1, \dots, A_n.$

$m(t_1, \dots, t_n) \{$
 call A_1
 ...
 call A_n
 $\}$

ÁTOMOS DE UN OBJETIVO \equiv *LLAMADAS A MÉTODOS*

$?- C_1, \dots, C_k$

call C_1
...
call C_k

UN PASO DE RESOLUCIÓN \equiv *UN PASO DE EJECUCIÓN*

UNIFICACIÓN

\equiv

MECANISMO PARA:

Paso Parámetros

Selección y Construcción de datos

3. El modelo de computación de la Programación Lógica

- El modelo de computación de la PL se basa en la regla de inferencia conocida como **Resolución**.
- La idea básica: para ejecutar la llamada **A** (un átomo):
 - ▣ Si el programa contiene el hecho A_0 y la llamada **A** **unifica** con A_0 , entonces la llamada **A** tiene éxito (y concluimos que es cierto).
 - ▣ Si el programa contiene una regla $A_0 :- A_1, \dots, A_n$ y **A** **unifica** con la cabeza A_0 , entonces debemos proceder a chequear de la misma forma A_1 hasta A_n .

Sustituciones. Composición de sustituciones

- **Notación:** Una sustitución $\{x_1 \rightarrow t_1, \dots, x_n \rightarrow t_n\}$ la denotaremos ahora como $\{x_1 / t_1, \dots, x_n / t_n\}$
- La aplicación de una sustitución σ a un objeto s (un término o un átomo) se denota en notación postfija: $s\sigma$ (en vez de $\sigma(s)$).
- **Composición de sustituciones:** Dadas dos sustituciones $\Theta = \{X_1/t_1, \dots, X_n/t_n\}$ y $\sigma = \{Y_1/s_1, \dots, Y_m/s_m\}$, su composición $\Theta\sigma$ es una sustitución que puede calcularse siguiendo el siguiente procedimiento:
 1. se aplica σ a las segundas componentes de Θ , es decir, $\{X_1/t_1\sigma, \dots, X_n/t_n\sigma\}$
 2. se añaden los enlaces Y_i/s_i de σ tales que Y_i sea una variable para la cual no haya ya un enlace.
 3. se eliminan los enlaces triviales (de la forma X/X siendo X una variable).

EJEMPLO: $\Theta = \{X/f(Y), Y/Z\}$ $\sigma = \{X/a, Y/b, Z/Y\}$

1. $\{X/f(Y)\sigma, Y/Z\sigma\} = \{X/f(b), Y/Y\}$
2. $\{X/f(b), Y/Y, Z/Y\}$
3. $\{X/f(b), \cancel{Y/Y}, Z/Y\}$

$\Theta\sigma = \{X/f(b), Z/Y\}$

¿Cómo tratar las variables de las consultas?

Unificación (paso de parámetros bidireccional)

- **Unificar** dos expresiones A y B consiste en encontrar una **sustitución** para sus variables que los hace idénticas.
- Informalmente:

	X	c	$f(t_1, \dots, t_n)$
X'	Sí, $\{X/X'\}$	Sí, $\{X'/c\}$	Sí, $\{X'/f(t_1, \dots, t_n)\}$
c'	Sí, $\{X/c'\}$	Sólo si $c=c'$	No
$f'(t'_1, \dots, t'_m)$	Sí, $\{X/f'(t'_1, \dots, t'_m)\}$	No	Sólo si $f=f'$, $n=m$ y cada t_i unifica con t'_i

1. dos expresiones con diferente símbolo principal o número de argumentos no se pueden unificar
2. una variable no se puede enlazar a un término que contenga dicha variable (crearíamos un término infinito). Esto se conoce como “occur check”.

Unificación (paso de parámetros bidireccional)

□ Ejemplo:

A unifica	con B	usando θ
$vuela(piolin)$	$vuela(piolin)$	$\{ \}$
X	Y	$\{X/Y\}$
X	a	$\{X/a\}$
$f(X, g(t))$	$f(m(h), g(M))$	$\{X/m(h), M/t\}$
$f(X, g(t))$	$f(m(h), t(M))$	imposible (1)
$f(X, X)$	$f(Y, h(Y))$	imposible (2)

Unificación de listas

□ Ejemplos:

$[a,b]$ unifica con $[X \mid R]$ usando $\{X/a, R/[b]\}$

$[a]$ unifica con $[X \mid R]$ usando $\{X/a, R/[]\}$

$[a \mid X]$ unifica con $[Y,b,c]$ usando $\{Y/a, X/[b,c]\}$

$[a]$ y $[X,Y \mid R]$ no unifican

$[]$ y $[X]$ no unifican

MGU (unificador más general)

- Durante la ejecución de un programa, necesitamos calcular el **MGU** entre los átomos del objetivo y las cabezas de las cláusulas

¿Cómo se calcula el mgu? (I)

- Dadas dos expresiones, t_1 y t_2 , si una de ellas es una variable, por ejemplo, t_1 es X :
 - devolvemos como mgu $\{X/t_2\}$
 - excepción 1: si $t_1 = t_2 = X$, el mgu es $\{ \}$ (sustitución vacía)
 - excepción 2: si t_2 no es una variable, y X aparece en t_2 , **fallo!** (no existe el mgu)

Nota: si se trata de variables diferentes, por ejemplo X e Y , el mgu puede ser tanto $\{X/Y\}$ como $\{Y/X\}$, ambos son válidos.

MGU (unificador más general)

- Durante la ejecución de un programa, necesitamos calcular el **MGU** entre los átomos del objetivo y las cabezas de las cláusulas

¿Cómo se calcula el mgu? (II)

- Si las expresiones tienen la forma $p(t_1, \dots, t_n)$ y $q(s_1, \dots, s_m)$
 - comprobamos que $p=q$ y $n=m$ (si no, **fallo**)
 - recorreremos los términos de izquierda a derecha, realizando la unificación de t_i y s_i con este mismo algoritmo para $i=1, \dots, n$
 - cada unificador θ_i calculado para t_i y s_i , se debe aplicar a **todos** los $t_1, \dots, t_n, s_1, \dots, s_m$ antes de seguir con la unificación de t_{i+1} y s_{i+1} , así como a los **términos** de los mgu's calculados anteriormente
 - si alguna unificación falla, terminamos con **fallo**
 - si llegamos al final **sin fallo** (las 2 expresiones serán ahora iguales), la unión de todos los θ_i es el MGU de las expresiones

MGU (unificador más general): Ejemplo

□ ¿Cuál es el MGU de $p([X, c], X)$ y $p([f(Y) | R], f(a))$?

1. Ponemos las listas en el mismo formato:

$p([X | [c]], X)$ y $p([f(Y) | R], f(a))$

2. El predicado y su aridad (núm de argumentos) coinciden, así que comenzamos a calcular unificadores de izquierda a derecha:

$p([X | [c]], X)$

$p([f(Y) | R], f(a))$

1er argumento: ¿unifican $[X | [c]]$ y $[f(Y) | R]$? Sí, con $\{X/f(Y), R/[c]\}$

MGU (unificador más general): Ejemplo

□ ¿Cuál es el MGU de $p([X,c], X)$ y $p([f(Y) \mid R], f(a))$?

1. Ponemos las listas en el mismo formato:

$p([X \mid [c]], X)$ y $p([f(Y) \mid R], f(a))$

2. El predicado y su aridad (núm de argumentos) coinciden, así que comenzamos a calcular unificadores de izquierda a derecha:

$p([X \mid [c]], X) \Rightarrow p([f(Y) \mid [c]], f(Y))$

$p([f(Y) \mid R], f(a)) \Rightarrow p([f(Y) \mid [c]], f(a))$

$\{X/f(Y), R/[c]\}$

Ahora aplicamos $\{X/f(Y), R/[c]\}$ a todos los términos

MGU (unificador más general): Ejemplo

□ ¿Cuál es el MGU de $p([X,c], X)$ y $p([f(Y) | R], f(a))$?

1. Ponemos las listas en el mismo formato:

$$p([X | [c]], X) \text{ y } p([f(Y) | R], f(a))$$

2. El predicado y su aridad (núm de argumentos) coinciden, así que comenzamos a calcular unificadores de izquierda a derecha:

$$p([X | [c]], X) \Rightarrow p([f(Y) | [c]], f(Y))$$

$$p([f(Y) | R], f(a)) \Rightarrow p([f(Y) | [c]], f(a))$$

$$\{X/f(Y), R/[c]\}$$

2° argumento: ¿unifican $f(Y)$ y $f(a)$? Sí, con $\{Y/a\}$

MGU (unificador más general): Ejemplo

□ ¿Cuál es el MGU de $p([X,c], X)$ y $p([f(Y) | R], f(a))$?

1. Ponemos las listas en el mismo formato:

$$p([X | [c]], X) \text{ y } p([f(Y) | R], f(a))$$

2. El predicado y su aridad (núm de argumentos) coinciden, así que comenzamos a calcular unificadores de izquierda a derecha:

$$\begin{array}{lcl} p([X | [c]], X) & \Rightarrow & p([f(Y) | [c]], f(Y)) \Rightarrow p([f(a) | [c]], f(a)) \\ p([f(Y) | R], f(a)) & \Rightarrow & p([f(Y) | [c]], f(a)) \Rightarrow p([f(a) | [c]], f(a)) \\ & \{X/f(a), R/[c]\} & \{Y/a\} \end{array}$$

Ahora aplicamos $\{Y/a\}$ a todos los términos
(incluyendo el mgu calculado anteriormente)

MGU (unificador más general): Ejemplo

□ ¿Cuál es el MGU de $p([X,c], X)$ y $p([f(Y) \mid R], f(a))$?

1. Ponemos las listas en el mismo formato:

$$p([X \mid [c]], X) \text{ y } p([f(Y) \mid R], f(a))$$

2. El predicado y su aridad (núm de argumentos) coinciden, así que comenzamos a calcular unificadores de izquierda a derecha:

$$\begin{array}{lcl} p([X \mid [c]], X) & \Rightarrow & p([f(Y) \mid [c]], f(Y)) \Rightarrow p([f(a) \mid [c]], f(a)) \\ p([f(Y) \mid R], f(a)) & \Rightarrow & p([f(Y) \mid [c]], f(a)) \Rightarrow p([f(a) \mid [c]], f(a)) \\ & \{X/f(a), R/[c]\} & \{Y/a\} \end{array}$$

$$\text{El MGU es } \{X/f(a), R/[c]\} \cup \{Y/a\} = \{X/f(a), R/[c], Y/a\}$$

Cálculo del MGU: método alternativo

Algoritmo de Unificación:

Para unificar un par de expresiones $p(t_1, \dots, t_n)$ y $q(s_1, \dots, s_m)$

1. Se comprueba que son compatibles, es decir $p=q \wedge n=m$
(si no, **parar con fallo**)
2. Se forma el conjunto de ecuaciones $\{t_1=s_1, \dots, t_n=s_n\}$
3. Se transforma usando las reglas de unificación hasta obtener un fallo o un conjunto $\{X_1=s_1, \dots, X_k=s_k\}$ tal que aplicándole cualquier regla no cambia.
4. En caso de éxito el unificador más general (mgu) es la sustitución: $\{X_1/s_1, \dots, X_k/s_k\}$

MGU: reglas de unificación

Reglas de la unificación: Se selecciona una ecuación cualquiera del conjunto y se aplica la regla que corresponda a la forma de la ecuación seleccionada:

- (i) $f(s_1, \dots, s_n) = f(t_1, \dots, t_n), n \geq 0 \Rightarrow$ **reemplazar por $s_1 = t_1, \dots, s_n = t_n$**
- (ii) $f(s_1, \dots, s_n) = g(t_1, \dots, t_m), f \neq g, n, m \geq 0 \Rightarrow$ **parar con fallo**
- (iii) $(X = X) \Rightarrow$ **borrar la ecuación**
- (iv) $(t = X), t \notin V \Rightarrow$ **reemplazar por $(X = t)$**
- (v) $(X = t), X \neq t$ *si X ocurre en $t \Rightarrow$ **parar con fallo** %occur-check
* si no \Rightarrow **aplicar $\{X/t\}$ al resto de ecuaciones
pero sin eliminar el $(X = t)$**

MGU: ejemplo

Ejemplo: encontrar el mgu de

$\text{is_tree}(\text{tree}(Y,2,\text{empty}))$ e $\text{is_tree}(\text{tree}(Z,X,Z))$

1. Son compatibles
2. Conjunto de ecuaciones inicial: $\{\text{tree}(Y,2,\text{empty})=\text{tree}(Z,X,Z)\}$
3. Secuencia de transformación

$\{\text{tree}(Y,2,\text{empty})=\text{tree}(Z,X,Z)\}$

$\Downarrow (i)$

$\{Y=Z, 2=X, \text{empty}=Z\}$

$\Downarrow (iv)$

$\{Y=Z, X=2, \text{empty}=Z\}$

$\Downarrow (iv)$

$\{Y=Z, X=2, Z=\text{empty}\}$

$\Downarrow (v)$

$\{Y=\text{empty}, X=2, Z=\text{empty}\}$

4. El mgu es $\{Y/\text{empty}, X/2, Z/\text{empty}\}$

Ejercicios MGU

- ¿Cuál es el MGU de

$$p(f(X, b), Z) \text{ y } p(f(a, Y), g(c)) \quad ?$$

- ¿Cuál es el MGU de

$$p([a, X], Y) \text{ y } p([H | R], b) \quad ?$$

- ¿Cuál es el MGU de

$$p(f(X), b, X) \text{ y } p(f(a), Y, b) \quad ?$$

3. El modelo de computación de la programación lógica: Resolución

Dado un programa lógico P y un objetivo $?-A_1, \dots, A_m$,

- si el programa contiene una cláusula $A :- B_1, \dots, B_n$ (cuyas variables han sido renombradas para evitar conflictos de unificación) y la cabeza A unifica con A_1 , con mgu σ **entonces** la regla de resolución genera el siguiente nuevo objetivo

$$\frac{\begin{array}{l} A :- B_1, \dots, B_n \\ ?- A_1, A_2, \dots, A_m \end{array}}{?- (B_1, \dots, B_n, A_2, \dots, A_m)\sigma}$$

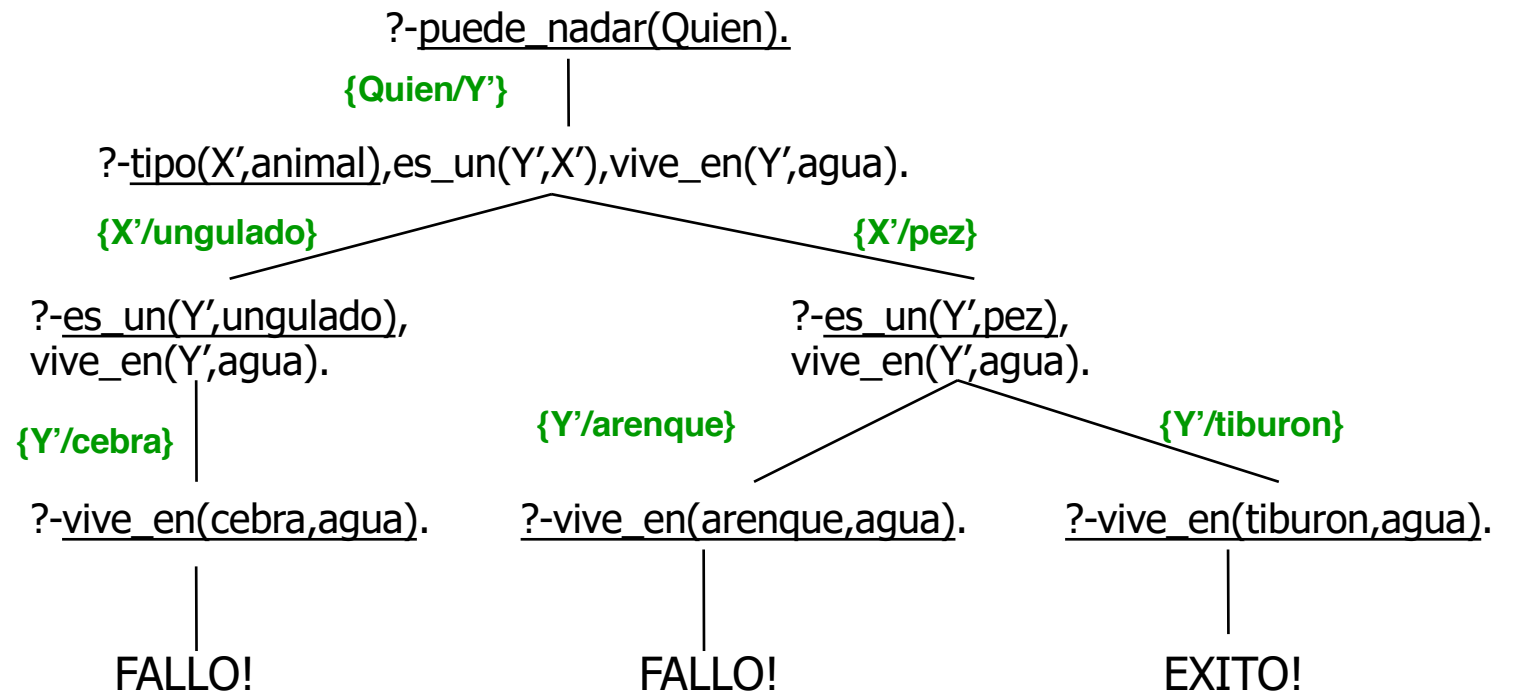
- La aplicación sucesiva de esta regla genera un **árbol de búsqueda**
- Una computación o **derivación** es una secuencia encadenada de pasos de resolución y se corresponde con cada una de las ramas del árbol.

Árbol de Búsqueda

tipo(ungulado,animal).
tipo(pez,animal).
es_un(cebra,ungulado).
es_un(arenque,pez).
es_un(tiburon,pez).

vive_en(cebra,tierra).
vive_en(rana,tierra).
vive_en(rana,agua).
vive_en(tiburon,agua).

puede_nadar(Y):-
 tipo(X,animal),
 es_un(Y,X),
 vive_en(Y,agua).



El modelo de computación de la programación lógica

Tipos de computación

- ▣ **Finita**: la computación termina en un número finito de pasos.

Distinguimos dos tipos de computaciones finitas:

- **De fallo**: ninguna cláusula unifica con el átomo seleccionado A_1
- **De éxito**: termina en la cláusula vacía (?-).

También se les llama **refutación**.

Cada rama de éxito produce una respuesta computada que se obtiene **componiendo la secuencia de los mgu's calculados a lo largo de la rama** (restringiéndolo después a las variables del objetivo inicial).

- **Infinita**: en cualquier objetivo de la secuencia, el átomo seleccionado A_1 unifica con (una variante de) una cláusula del programa

Tipos de derivaciones

INFINITA

$\{p(f(X)) :- p(X).\}$

$?-p(X)$

$\Downarrow \{X/f(X')\}$

$?-p(X')$

$\Downarrow \{X'/f(X'')\}$

$?-p(X'')$

$\Downarrow \{X''/f(X''')\}$

$?-p(X''')$

$\Downarrow (\infty)$

DE FALLO

$\{p(0) :- q(X).\}$

$?-p(Z)$

$\Downarrow \{Z/0\}$

$?-q(X')$

$\Downarrow \text{fallo}$

DE ÉXITO

$\{p(0) :- q(X). \\ q(1).\}$

$?-p(Z)$

$\Downarrow \{Z/0\}$

$?-q(X')$

$\Downarrow \{X'/1\}$

'?- ' **éxito!**

La importancia del renombramiento (1 / 3)

Ejemplo :

$p(f(Z)) \text{ :- } q(Z).$

$q(Y) \text{ :- } r(X).$

$r(a).$

$?- p(X)$

$\Downarrow \quad \{X/f(Z)\}$

$?- q(Z)$

$\Downarrow \quad \{Z/Y\}$

$?- r(X)$

$\Downarrow \quad \{X/a\}$

$?-$

incorrecto pues X queda enlazada en la misma derivación a dos términos distintos

La importancia del renombramiento (2/3)

Ejemplo :

$p(f(X)) \text{ :- } q(X).$

$q(a).$

$?- p(X)$

\Downarrow

$\{X/f(X)\}$ occur-check!!

$?- q(X)$

\Downarrow

...

Aquí podríamos concluir que la derivación falla (ya que el MGU falla debido al occur-check), lo que no es correcto.

Si renombramos las variables de “ $p(f(X)) \text{ :- } q(X).$ ” sí existe una derivación de éxito que computa la respuesta $\{X/f(a)\}$.

La importancia del renombramiento (3/3)

Ejercicio: ¿Cuál es la respuesta al objetivo

$?-p(X).$

con respecto al siguiente programa?

$r(0).$

$p(Y) :- q.$

$q :- r(Y).$

- A. $\{X/0, Y'/0\}$
- B. $\{X/Y'\}$
- C. $\{X/0\}$
- D. $\{Y'/0\}$

Búsqueda Predefinida

La regla de búsqueda determina:

- 1) El orden en que se ensayan las cláusulas del programa y,
- 2) La estrategia con que se recorre el árbol resultante.

Hay dos estrategias fundamentales:

- * **Profundidad**: se pierde la completitud del procedimiento de resolución SLD.
- * **Anchura**: se recorre el árbol por niveles. Se mantiene la completitud, aunque es muy costosa.

PROLOG: búsqueda predefinida automática

- 1) top-down,
- 2) búsqueda en profundidad con vuelta atrás (backtracking).

Ejercicio

- Calcula el árbol de búsqueda del objetivo

?- pair(Person1,Person2).

con respecto al siguiente programa:

editor(zenspider, emacs).

editor(drbrain, vim).

editor(phiggins, vim).

editor(tenderlove, vim).

```
pair(Person1,Person2) :- editor(Person1, Editor),  
                           editor(Person2, Editor),  
                           Person1 \==Person2.
```

Ejercicio

- Calcula el árbol de búsqueda del objetivo

?- length([1,2],L).

con respecto a este programa lógico

length ([], 0).

length ([_ | T], N) :- length(T, N1),
N is N1 + 1.

4. Algunas cuestiones prácticas

Aplicaciones de la PL

- Verificación de software y hardware
- Certificación de programas
- Prototipado automático
- Ingeniería del software automática (depuración automática, síntesis de programas a partir de especificaciones, transformación de programas,...)
- Modelización en Sistemas de Información y Bases de Datos
- Problemas de Aprendizaje
- Robótica y Planificación
- Sistemas Expertos
- Tratamiento de Lenguaje Natural

LTP: Bibliografía

BÁSICA

- ▣ Pascual Julián Iranzo, María Alpuente Frasnado. *Programación lógica: teoría y práctica*. Prentice-Hall International (Pearson Educación), 2009.
- ▣ W.F. Clocksin, C.S. Mellish. *Programación en PROLOG* (tr. por: Juan Alberto Alonso Martín). Editorial Gustavo Gili, S.A.2. ed.(10/1993). 5ª Edición en inglés 2003.

ADICIONAL

- ▣ Krzysztof R. Apt, *From logic programming to Prolog*. Prentice Hall, 1997.
- ▣ Leon Sterling. *The art of Prolog : Advanced programming techniques*. MIT Press, 1997.