Programación Declarativa:
Lógica y Restricciones

Conceptos Básicos de la Programación en Prolog

Mari Carmen Suárez de Figueroa Baonza
mcsuarez@fi.upm.es

Contenidos

- Unificación
- Estructuras de datos
- Recursividad, *backtracking* y búsqueda
- Control de ejecución

Unificación (I)

- La **unificación** es el mecanismo que se encarga de resolver las igualdades lógicas y de dar valor a las variables lógicas
  - En la unificación no se evalúan expresiones
  - Para evaluar expresiones existe un operador especial "*is*"
    - Antes de realizar la unificación evalúa la parte derecha como si se tratase de una expresión aritmética

Unificación (II): Reglas Generales

- Si T1 y T2 son constantes, entonces T1 y T2 unifican si son idénticas
- Si T1 y T2 son variables, entonces T1 y T2 unifican siempre
- Si T1 es una variable y T2 es cualquier tipo de término, entonces T1 y T2 unifican y T1 se instancia con T2
- Si T1 y T2 son términos complejos, unifican si:
  - Tienen el mismo functor y aridad
  - Todos los argumentos unifican

Unificación (III). Ejemplos

- *¿pepe(A,rojo)=jose(B,rojo)?*
  - *pepe!=jose → No unifica*
- *¿pepe(A,rojo)=pepe(Z,rojo)?*
  - *pepe=pepe, A=Z, rojo= rojo → Si unifican*
- *¿X=a?*
- *¿X=Y?*
- *¿f(a)=f(X)?*
- *¿f(X,a)=f(b,Y)?*
- *¿f(X,a,X)=f(b,Y,c)?*
- *¿X=f(X)?*

Unificación (IV): =/2

- **=** simboliza el predicado *"unifica"*:
  - El predicado =/2 está *predefinido* en ISO-Prolog, no es necesario programar la unificación, que es una característica básica del demostrador automático
  - Sus dos argumentos son las dos expresiones a unificar
  - Este predicado es *verdadero* si sus dos argumentos unifican:
    - a = a ; f(a,b) = f(a,b) ; X = a ; f(a,b) = X ; X = Y ; f(a,X) = f(a,b) ; X = f(Y)
  - Y es *falso* si sus dos argumentos no unifican:
    - a = b ; f(a,b) = f(b,a) ; X = f(X) ; f(X,X) = f(g(Y),Y)

Unificación implícita y la variable anónima

- La unificación de una variable puede realizarse:
  - Explícitamente, empleando =/2 como un objetivo más
  - Implícitamente, dado que Resolución = Corte + Unificación

```
% “Juan es amigo de cualquiera que sea rico“
% (x)(rico(x) -> amigo(juan,x))
amigo(juan,X):- rico(X).
```

    - Es necesario unificar el segundo argumento de amigo/2, para que su valor se emplee en la prueba de rico/1

- La variable anónima:
  - Sintácticamente: ‘_’ ; ‘_Anónima’; ‘_X’
  - Semánticamente: no se unifica, no toma nunca valor

```
% "Pepe es amigo de todo el mundo“
% (x) amigo(pepe,x)
amigo(pepe,_).
```

    - Es innecesario unificar el segundo argumento de amigo/2, cualquier valor es aceptable y no se emplea en ulteriores objetivos

Papel de la Unificación en Ejecución

Role of Unification in Execution

- As mentioned before, unification used to *access data* and *give values to variables*.
  Example: Consider query `?- animal(A), named(A,Name).` with:

```
animal(dog(barry)).
named(dog(Name),Name).
```

  Execution of `animal(A)` assigns a (ground) value to `A`.
  Execution of `named(A,Name)` assigns a (ground) value to `Name` by accessing the data in the subfield of the `dog/1` structure.

- Also, unification is used to *pass parameters* in procedure calls and to *return values* upon procedure exit.

`?- animal(A), named(A,Name)`	returns a value upon exit of `animal(A)`
`?- named(dog(barry),Name)`	passes a value in first argument of call to `named/2`
`Name = barry`	returns a value in second argument on exit of `named/2`

Modos de uso

Modes

Nota: En la definición de un procedimiento no hay parámetros predefinidos de "entrada" y/o "salida". El modo de uso de cada parámetro depende de la llamada o pregunta que se haga en cada momento al procedimiento.

- In fact, argument positions are not fixed a priory to be input or output.

  Example: Consider query `?- pet(spot).` vs. `?- pet(X).`

- Upon a call to a procedure, any argument may be ground, free, or partially instantiated.

- Thus, procedures can be used in different **modes**
  (different sets of arguments are input or output in each mode).

  Example: Consider the following queries:

```
?- named(dog(barry),Name). % entrada, salida      ?- named(A,barry). % salida, entrada
?- named(dog(barry),barry). % entrada, entrada    ?- named(A,Name). % salida, salida
```

- An argument may even be both input and output.

  Example: Consider query `?- struct(f(A,b)).` with:

```
struct(f(a,B)).
```

Acceso a los datos (I)

Accessing Data

- Accesing subfields of *records*:

  Example:

```
day(date(Day,_Month,_Year),Day).
month(date(_Day,Month,_Year),Month).
year(date(_Day,_Month,Year),Year).
```

- Naming subfields:

  Example:

```
date(day,   date(Day,_Month,_Year),Day).
date(month,date(_Day,Month,_Year),Month).
date(year,  date(_Day,_Month,Year),Year).
```

Acceso a los datos (II)

Accessing Data (Contd.)

- Initializing variables:

  Example: `?- init(X), ...`

  ```
  init(date(9,6,2011)).
  ```

- Comparing values:

  Example: `?- init_1(X), init_2(Y), equal(X,Y).`

  ```
  equal(X,X).
  ```

  or simply: `?- init_1(X), init_2(X).`

Datos Estructurados (I)

Structured Data and Data Abstraction (and the '=' Predicate)

- *Data structures* are created using (complex) terms.
- Structuring data is important:

```
course(complog,wed,18,30,20,30,'F.','Bueno',new,5102).
```

course/10

- When is the Computational Logic course?

```
?- course(complog,Day,StartH,StartM,FinishH,FinishM,C,D,E,F).
```

- Structured version:

```
course(complog,Time,Lecturer, Location) :-
    Time = t(wed,18:30,20:30),
    Lecturer = lect('F.','Bueno'),
    Location = loc(new,5102).
```

course/4

Note: "X=Y" is equivalent to "=(X,Y)"
where the predicate =/2 is defined as the fact "=(X,X)." – Plain unification!

- Equivalent to:

```
course(complog,  t(wed,18:30,20:30),
       lect('F.','Bueno'),  loc(new,5102)).
```

Datos Estructurados (II)

Structured Data and Data Abstraction (and The Anonymous Variable)

- Given:

```
course(complog,Time,Lecturer, Location) :-
        Time = t(wed,18:30,20:30),
        Lecturer = lect('F.','Bueno'),
        Location = loc(new,5102).
```

- When is the Computational Logic course?

```
?- course(complog,Time, A, B).
```

has solution:

```
{Time=t(wed,18:30,20:30), A=lect('F.','Bueno'), B=loc(new,5102)}
```

- Using the *anonymous variable* ("_"):

```
?- course(complog,Time, _, _).
```

has solution:

```
{Time=t(wed,18:30,20:30)}
```

Estructuras de Datos

- *2_basics_prolog.pdf*
  - Transparencias 10-19

Listas: Ejercicios (I)

- Definir prefijo(X,Y): la lista X es un prefijo de la lista Y
  - prefijo([a,b], [a,b,c,d]).
- Definir sufijo(X,Y): la lista X es un sufijo de la lista Y
  - sufijo([c,d], [a,b,c,d]).
- Definir sublista(X,Y): la lista X es una sublista de la lista Y
  - sublista([b,c],[a,b,c,d]).
- Definir longitud(X,N): N es la longitud de la lista X

Listas: Ejercicios (II)

- Palíndromos
  - Definir el predicado palindromo/1 tal que:
    - palindromo(X) es cierto si la lista X es palíndromo, es decir, puede leerse de la misma manera al derecho y al revés
    - Ejemplos: palindromo([r,o,t,o,r]) es verdadero

      palindromo([r,o,t,a,r]) es falso

      palindromo([r,o,t|X]) es verdadero con {X = [o,r]}, o {X = [t,o,r]} o {X = [_A,t,o,r]} o ...
- Primero y Último
  - Definir el predicado primeroultimo/1 tal que:
    - primeroultimo(X) es cierto si el primer y ultimo elementos de la lista X son el mismo
    - Ejemplos: primeroultimo([a]) es verdadero

      primeroultimo([a,f,t]) es falso

      primeroultimo([X,f,t,a]) es verdadero con {X = a}

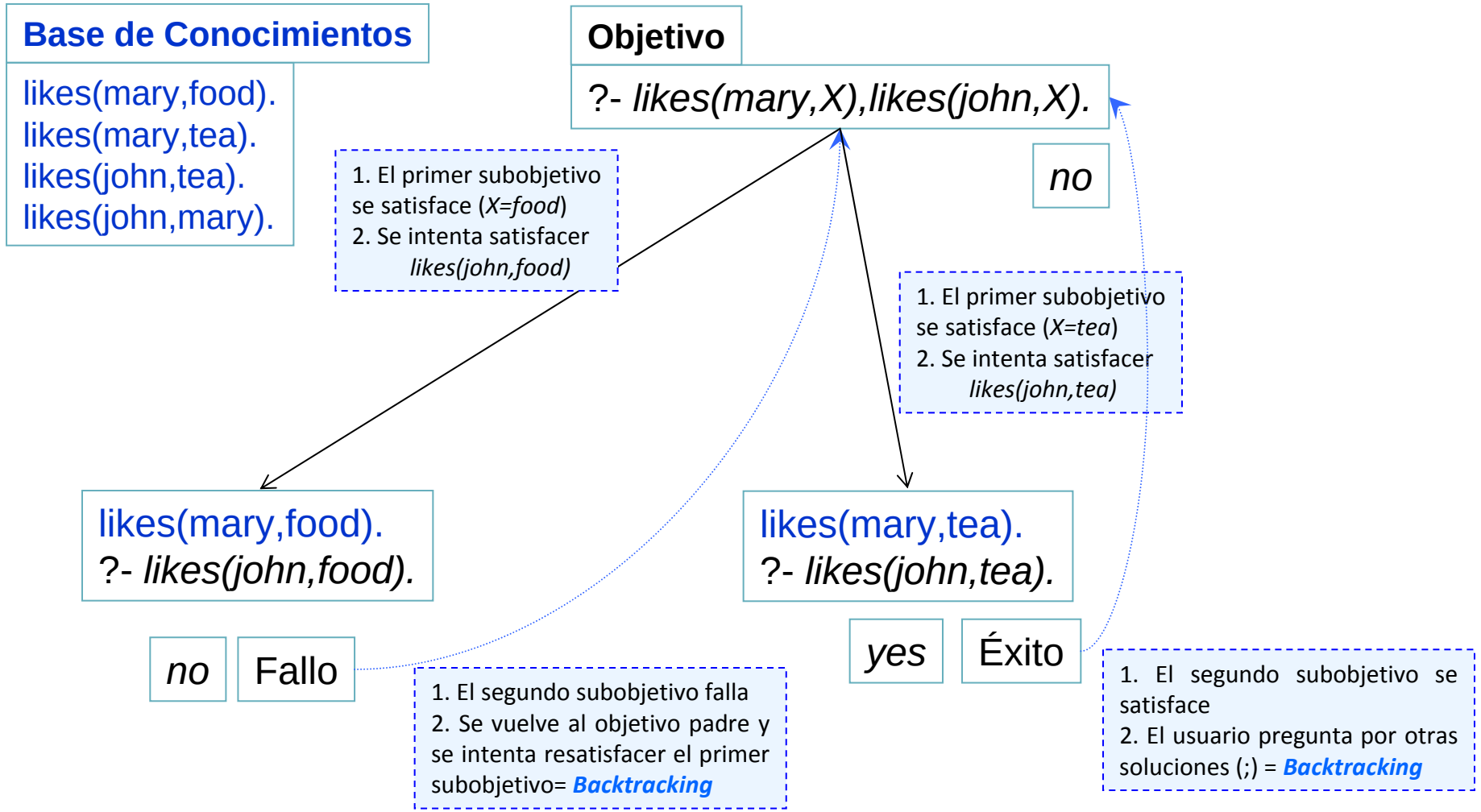
Backtracking (I)

- **Programa lógico**:
  - Base de conocimientos donde se expresan los hechos y las reglas de deducción de un dominio o problema
  - Motor de inferencia que aplica el algoritmo de resolución. Este algoritmo permite inferir nuevos datos relativos al mundo que estamos representando
    - Toma como **entrada** la base de conocimientos y el objetivo planteado y ofrece como **salida** un resultado de verdadero o falso en función de si ha podido o no demostrar el objetivo según la base de conocimientos
    - Este algoritmo se basa en el uso de la técnica de ***backtracking***, de forma que la inferencia del objetivo planteado se realiza a base de prueba y error

Backtracking (II)

- Un hecho puede hacer que un objetivo se cumpla inmediatamente
- Una regla sólo puede reducir la tarea a la de satisfacer una conjunción de subobjetivos
- Si no se puede satisfacer un objetivo, se inicia un proceso de *backtracking*
  - Este proceso consiste en intentar satisfacer los objetivos buscando una forma alternativa de hacerlo
- El mecanismo de *backtracking* permite explorar los diferentes caminos de ejecución hasta que se encuentre una solución
  - *Backtracking* por fallo
  - *Backtracking* por acción del usuario

Backtracking (III). Ejemplo



Control de la búsqueda

- Existen 3 formas principales de controlar la ejecución de un programa lógico
  - El orden de las cláusulas en un predicado
  - El orden de los literales en el cuerpo de una cláusula
  - Los operadores de poda (ej., '*cut*')
- El orden de las cláusulas en el programa y el orden de los literales en una cláusula son importantes
  - El orden afecta tanto al correcto funcionamiento del programa, como al recorrido del árbol de llamadas, determinando, entre otras cosas, el orden en que Prolog devuelve las soluciones a una pregunta dada

Orden de las cláusulas

- El orden de las cláusulas determina el orden en que se obtienen las soluciones
  - Varía la manera en que se recorren las ramas del árbol de búsqueda de soluciones
- Si el árbol de búsqueda tiene alguna rama infinita, el orden de las sentencias puede alterar la obtención de las soluciones, e incluso llegar a la no obtención de ninguna solución
- Regla heurística: es recomendable que los hechos aparezcan antes que las reglas del mismo predicado

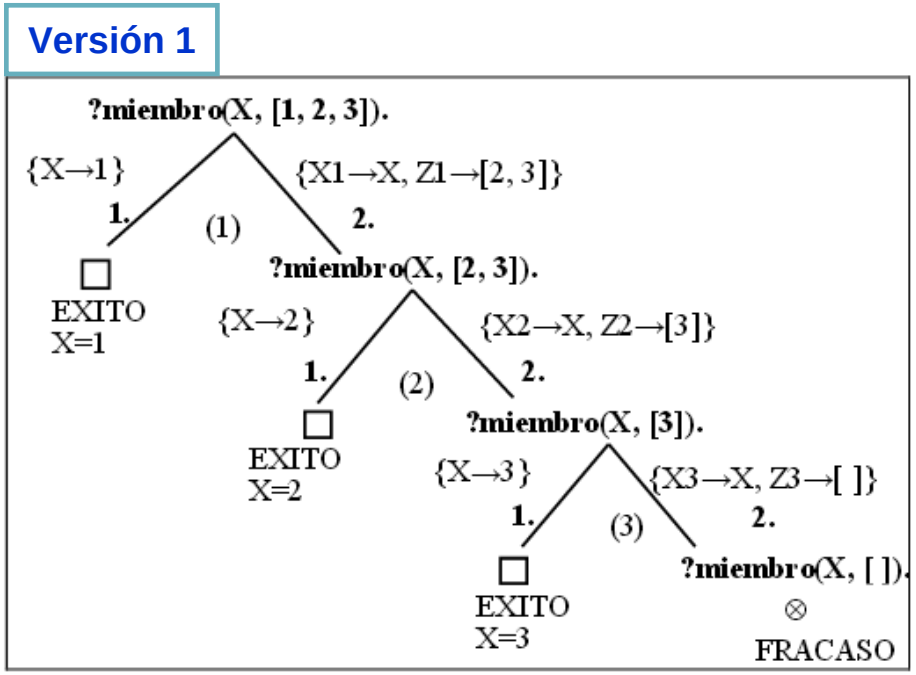
Orden de las cláusulas: Ejemplo (I)

- Dos versiones de miembro de una lista (miembro(X,L))
  - Ambas versiones tienen las mismas cláusulas pero escritas en distinto orden
- Versión 1:
  1) miembro(X,[X|_]).
  2) miembro(X,[_|Z]):- miembro(X,Z).
- Versión 2:
  1) miembro(X,[_|Z]):- miembro(X,Z).
  2) miembro(X,[X|_]).

Orden de las cláusulas: Ejemplo (II)

- A ambas versiones les hacemos la misma pregunta
  - ?- miembro (X, [1,2,3]).

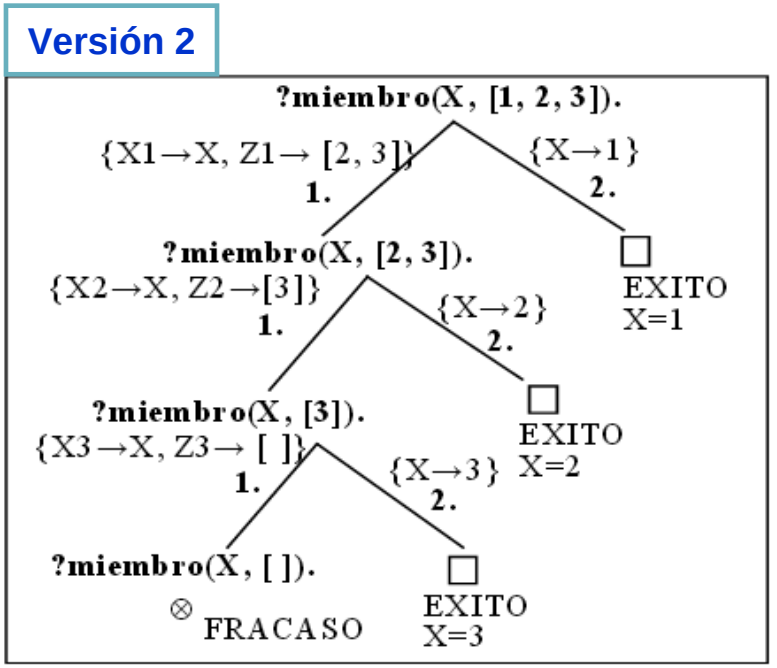
- Versión 1:
  1) miembro(X,[X|_]).
  2) miembro(X,[_|Z]):- miembro(X,Z).



Orden de las cláusulas: Ejemplo (III)

- A ambas versiones les hacemos la misma pregunta
  - ?- miembro (X, [1,2,3]).

- Versión 2:
  1) miembro(X,[_|Z]):- miembro(X,Z).
  2) miembro(X,[X|_]).



Orden de los literales (I)

- El orden de los literales dentro de una cláusula afecta al espacio de búsqueda y a la complejidad de los cómputos lógicos
- Distintas opciones en el orden de los literales pueden ser preferibles para distintos modos de uso
  - hijo(X, Y) :- hombre(X), padre(Y, X).
    - Para modo (in, out): Se comprueba primero que el X es hombre y después se busca a su padre Y
  - hijo(X, Y) :- padre(Y, X), hombre(X).
    - Para modo (out, in): Se buscan los hijos de Y y después se seleccionan si son hombres

Orden de los literales (II)

- El orden de los literales en el cuerpo de una regla influye también en la terminación
  - inversa([ ], [ ]).
  - inversa([C|R], Z) :- inversa(R, Y), concatenar(Y, [C], Z).
    - Para preguntas en modo (in, out) termina
    - Para preguntas en modo (out, in) el árbol de búsqueda tiene una rama infinita, por lo que tras dar la respuesta correcta se queda en un bucle
  - ¿Qué sucede si intercambiamos los literales en la regla?

ordenLiterales.pl

Control con poda: *cut* (I)

- Prolog proporciona un predicado predefinido llamado cut (!/0) que influye en el comportamiento procedural de los programas
- Su principal función es reducir el espacio de búsqueda podando dinámicamente el árbol de búsqueda
- El corte puede usarse:
  - Para aumentar la eficiencia
    - Se eliminan puntos de *backtracking* que se sabe que no pueden producir ninguna solución
  - Para modificar el comportamiento del programa
    - Se eliminan puntos de *backtracking* que pueden producir soluciones válidas. Se implementa de este modo una forma débil de negación
    - Este tipo de corte debe utilizarse lo menos posible

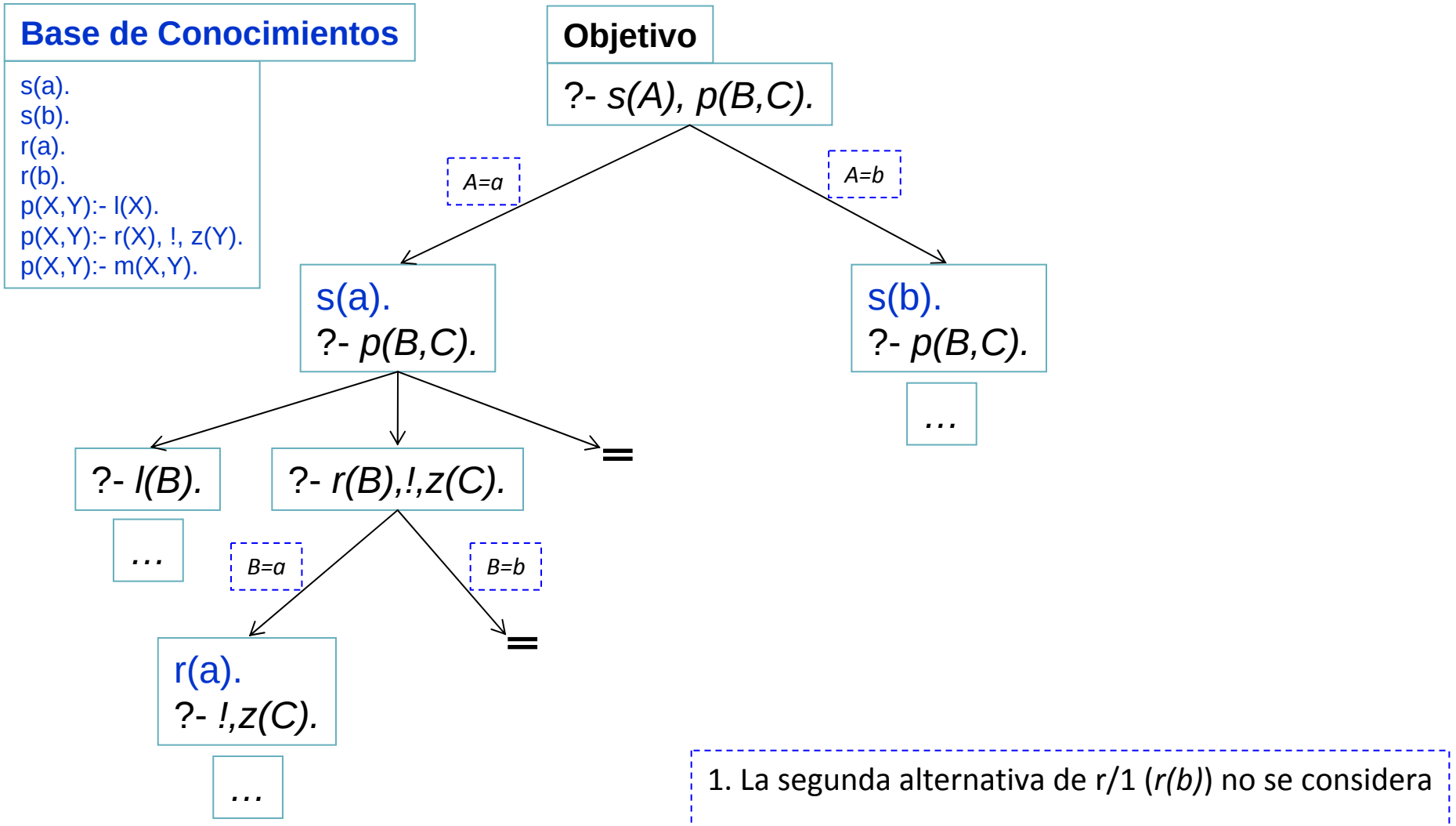
Control con poda: *cut* (II)

- El predicado cut como objetivo se satisface siempre y no puede re-satisfacerse
- Su uso permite podar ramas del árbol de búsqueda de soluciones
- Como consecuencia, un programa que use el corte será generalmente más rápido y ocupará menos espacio en memoria (no tiene que recordar los puntos de *backtracking* para una posible reevaluación)
  - El operador de corte '!' limita el *backtracking*
  - Cuando se ejecuta el operador de corte elimina todos los puntos de *backtracking* anteriores dentro del predicado donde está definido (incluido el propio predicado)

Control con poda: *cut* (III)

- Su funcionamiento implica que:
  - Un corte poda todas las alternativas correspondientes a cláusulas por debajo de él
  - Un corte poda todas las soluciones alternativas de la conjunción de objetivos que aparezcan a su izquierda en la cláusula
    - Esto es, una conjunción de objetivos seguida por un corte producirá como máximo una solución
  - Un corte no afecta a los objetivos que estén a su derecha en la cláusula
    - Estos objetivos pueden producir más de una solución, en caso de *backtracking*
    - Sin embargo, una vez que esta conjunción fracasa, la búsqueda continuará a partir de la última alternativa que había por encima de la elección de la sentencia que contiene el corte

Control con poda: *cut* (IV). Ejemplo



1. La segunda alternativa de r/1 (*r(b)*) no se considera
2. La tercera cláusula de p/2 no se considera

Control con poda: *cut* (V)

- Resumiendo, de forma general, el efecto de un corte en una regla C de la forma A :- B1, ..., Bk, !, Bk+2, ..., Bn, es el siguiente:
  - Si el objetivo actual G se unifica con A y los objetivos B1, ..., Bk se satisfacen, entonces el programa fija la elección de esta regla para deducir G; cualquier otra regla alternativa para A que pueda unificarse con G se ignora
  - Además, si los Bi con i>k fracasan, la vuelta atrás sólo puede hacerse hasta el corte. Las demás elecciones que quedaran para computar los Bi con i≤ k se han cortado del árbol de búsqueda
  - Si el *backtracking* llega de hecho al corte entonces éste fracasa y la búsqueda continúa desde la última elección hecha antes de que G eligiera la regla C

Control con poda: *cut*. Ejemplo (I)

- mezcla (L1, L2, L), en modo (in,in,out), mezcla dos listas ordenadas de números L1 y L2 en la lista ordenada L
  1) mezcla ([X| Xs], [Y| Ys], [X| Zs]) :- X<Y, mezcla (Xs, [Y| Ys], Zs).
  2) mezcla ([X| Xs], [Y| Ys], [X,Y | Zs]) :- X=Y, mezcla (Xs, Ys, Zs).
  3) mezcla ([X| Xs], [Y| Ys], [Y| Zs]) :- X>Y, mezcla ([X| Xs], Ys, Zs).
  4) mezcla (Xs, [ ], Xs).
  5) mezcla ([ ], Ys, Ys).
- La mezcla de dos listas ordenadas es una operación determinista
  - Sólo una de las cinco cláusulas se aplica para cada objetivo (no trivial) en una computación dada
  - En concreto, cuando comparamos dos números X e Y, sólo una de las tres comprobaciones X<Y, X=Y ó X>Y es cierta
  - Una vez una comprobación se satisface, no existe posibilidad de que alguna otra comprobación se satisfaga

Control con poda: *cut*. Ejemplo (II)

- El corte puede usarse para expresar la naturaleza mutuamente exclusiva de las comprobaciones de las tres primeras cláusulas
  1) mezcla ([X | Xs], [Y | Ys], [X | Zs]) :- X<Y, !,mezcla (Xs, [Y | Ys], Zs).
  2) mezcla ([X| Xs], [Y| Ys], [X, Y| Zs]) :- X=Y, !,mezcla (Xs, Ys, Zs).
  3) mezcla ([X| Xs], [Y| Ys], [Y| Zs]) :- X>Y, !, mezcla ([X| Xs], Ys, Zs).
- Por otra parte, los dos casos básicos del programa (cláusulas 4 y 5) son también deterministas
  4) mezcla (Xs, [ ], Xs):- !.
  5) mezcla ([ ], Ys, Ys).
  - La cláusula correcta a utilizar se elige por unificación con la cabeza, por eso el corte aparece como el primer objetivo (en este caso el único) en el cuerpo de la cláusula 4
    - Dicho corte elimina la solución redundante (que se volvería a obtener con la cláusula 5) dado el objetivo "?-mezcla([ ], [ ], X)"

Tipos de Corte

- Verdes: descartan soluciones correctas que no son necesarias
  - No afectan al sentido declarativo del programa
  - Sólo afectan a la eficiencia del programa
  - Afectan a la completitud pero no a la corrección
  - Podan ramas inútiles, redundantes o infinitas
- Rojos: descartan soluciones que no son correctas
  - Afectan a la semántica declarativa del programa
  - Modifican el significado lógico del programa
  - Al eliminar el operador de corte se obtiene un programa incorrecto
  - Evitan soluciones erróneas podando ramas que conducen a éxitos no deseados

Cortes Verdes

- No alteran el significado declarativo del programa
- En un programa semánticamente correcto se añade el corte para obtener un programa más eficiente
- Generalmente se usan para expresar determinismo
  - la parte del cuerpo que precede al corte (o a veces el patrón de la cabeza) comprueba un caso que excluye a todos los demás

Ejemplos:

- address(X,Add):- home_address(X,Add), !.
- address(X,Add):- business_address(X,Add).

- membercheck(X,[X|Xs]):- !.
- membercheck(X,[Y|Xs]):- membercheck(X,Xs).

Cortes Verdes. Ejemplo (I)

- Corte verde para evitar soluciones redundantes
  - es_padre(X):-padre(X,Y),!.
  - padre(antonio,juan).
  - padre(antonio,maria).
  - padre(antonio,jose).

ejemploCorte.pl

Cortes Verdes. Ejemplo (II)

- Corte verde para evitar búsquedas inútiles
  - El predicado ordenar(L,R) indica que R es el resultado de ordenar la lista L por medio de intercambios sucesivos. Este predicado usará ordenada(L) que nos dice si la lista L está ordenada.

    1) ordenar (L, R) :- concatenar(P, [X,Y | S], L), X>Y, !, concatenar(P, [Y,X| S], NL), ordenar(NL, R).
    2) ordenar (L, L) :- ordenada(L).

  - Se sabe que sólo hay una lista ordenada. Por tanto, no tiene sentido buscar otras alternativas una vez se ha encontrado la lista ordenada

Cortes Rojos

- Afectan a la semántica declarativa del programa
- Deben emplearse con cuidado

- Ejemplos:
  - max(X,Y,X) :- X>Y, !.
  - max(X,Y,Y).
    - ?- max(5,2,2).
  - progenitores(adan,0):-!.
  - progenitores(eva,0):-!.
  - progenitores(P,2).

Ejercicios (I): Uso del Corte

- Borrar todas las apariciones de un cierto elemento en una lista dada
  - borrar(X, L1, L2)
    - L2 es la lista obtenida al borrar todas las apariciones de X en la lista L1
    - Modo de uso (in, in, out), es decir, primer y segundo parámetros de entrada y tercer parámetro de salida

Ejercicios (II): Uso del Corte

- Definir un predicado agregacion/3
  - agregacion (X,L,L1): L1 es la lista obtenida añadiendo el elemento X a la lista L (si X no pertenece a L), y es L en caso contrario
    - ?- agregacion(a,[b,c[,L).
    - L=[a,b,c])
    - ?-agregacion(b,[b,c],L).
    - L=[b,c]

Ejercicios (III): Uso del Corte

- Comprobar si una lista es sublista de otra
  - sublista(X, Y)
    - X es sublista de Y
    - Modo de uso (in, in)

Programas *Generate & Test*

- Son básicamente programas que generan soluciones candidatas que se evalúan para comprobar si son o no correctas
  - En algunas ocasiones es más sencillo comprobar si algo es una solución a un problema que crear la solución a dicho problema
- Consisten en dividir la resolución de problemas en dos partes:
  - Generar soluciones candidatas
  - Testear que las soluciones sean correctas
- Son programas con la siguiente estructura:
  - Una serie de objetivos generan posibles soluciones vía *backtracking*
  - Otros objetivos comprueban si dichas soluciones son las apropiadas

Programas *Generate & Test*: Ejemplo

- Ordenación de listas
  - ordenacion(X,Y)
    - Y es la lista resultante de ordenar la lista X de forma ascendente
    - La lista Y contiene, en orden ascendente, los mismos elementos que la lista X
    - La lista Y es una permutación de la lista X con los elementos en orden ascendente
  - ?- ordenacion([2,1,2,3], L).
    - L = [1,2,2,3]
  - ordenacion(X,Y) :-

    permutacion(X,Y),

    ordenada_ascendente(Y).

    (1) Generador: se obtiene una permutación de X en Y que pasa al objetivo (2) para comprobar si Y está ordenada

    (2) Prueba: comprueba si la lista está ordenada. Si no lo está, el *backtracking* se encarga de re-satisfacer el objetivo (1) buscando una nueva permutación

ejemploGenerateTest.pl

Programas *Generate & Test*: Ejercicio

- Definir el predicado numeroParMenor/2 que es verdadero cuando X es un numero par menor que N
  - ?- numeroParMenor(X,5).
    - X=0; X=2; X=4
  - ?- numeroParMenor(2,4).
    - Yes
  - ?- numeroParMenor(3,5).
    - No
  - ?- numeroParMenor(10,7).
    - No

Programación Declarativa:
Lógica y Restricciones

Conceptos Básicos de la Programación en Prolog

Mari Carmen Suárez de Figueroa Baonza
mcsuarez@fi.upm.es