Cláusulas de Horn

Una **cláusula de Horn** es un conjunto no vacío de literales en el que **no hay más de un literal positivo**.

- Una **cláusula de Horn** es o bien
  - **Una restricción:** una cláusula **sin ningún literal positivo** (es decir, la negación de una cláusula objetivo conjuntiva)

  Ej. $\neg P \vee \neg Q \equiv \neg (P \wedge Q) \equiv (P \wedge Q) \Rightarrow F$

  - **Un "hecho"**: una cláusula unitaria formada por un solo literal positivo

  Ej. P, Q (hechos)

  - **Una "regla":** una cláusula con un solo literal positivo y con al menos un literal negativo. Una regla puede ser escrita como una **implicación**, donde la premisa es una conjunción de las negaciones de los literales negativos, y el consecuente es el literal positivo.

  Ej. $\neg P \vee \neg Q \vee \neg R \vee S \equiv (P \wedge Q \wedge R) \Rightarrow S$ (regla)

- La inferencia sobre un conjunto α de cláusulas de Horn puede ser hecha por
  - » **Encadenamiento hacia delante** (razonamiento dirigido por los datos)
  - » **Encadenamiento hacia atrás** (razonamiento dirigido por el objetivo)

» El problema **SAT** sobre un conjunto de **cláusulas de Horn es lineal** en el número de cláusulas del conjunto.

Encadenamiento hacia delante (EHD)

- **Considérese un conjunto de cláusulas de Horn α_0**

  1. $\alpha = \alpha_0$
  2. Aplicar aquellas reglas cuyas premisas están en α, e incorporar las conclusiones a α.
  3. Repetir (2) hasta que no se puedan aplicar más reglas.

- Encadenamiento hacia delante (EHD) sobre conjuntos de cláusulas de Horn

  - **EHD es correcto:** Cada inferencia es una aplicación del modus ponens, que es correcto.
  - **EHD es completo:** EHD deriva todas las sentencias atómicas que son consecuencia lógica de α_0

  Demostración:

  - Cuando EHD termina no se pueden deducir más sentencias atómicas por aplicación de EHD
  - Interpretación donde se asigna el valor *Verdadero* a todos los átomos deducidos por EHD de α_0 , y el valor *Falso* a todos los átomos que no se deducen por EHD de α_0 .
    Por contradicción: Asumamos que la regla $(P_1 \wedge P_2 \wedge ... \wedge P_n) \Rightarrow S$ de α_0 tiene valor *Falso*.
    $(P_1 \wedge P_2 \wedge ... \wedge P_n)$ debe entonces tener valor *Verdadero* y S debe tener valor *Falso.*
    Entonces $P_1, P_2, ..., P_n$ tienen valor *Verdadero* i.e. han sido deducidos por EHD,
    y la regla $(P_1 \wedge P_2 \wedge ... \wedge P_n) \Rightarrow S$ debería haber sido aplicada.
  - Asumamos que existe un átomo $P / \alpha_0 \models P$ entonces P tiene el valor *Verdadero* en todos los modelos de α_0, y P debe ser uno de los átomos deducidos por EHD de α_0 .

Encadenamiento hacia atrás (EHA)

- **Deducir el objetivo P a partir de un conjunto α de cláusulas de Horn**

  1. Si $P \in \alpha$ entonces objetivo encontrado.
  2. Demostrar por encadenamiento hacia atrás de todos los átomos en la premisa de alguna regla en α cuya conclusión es P.

- Evitar bucles:
  Chequear si el nuevo subobjetivo ya está en la pila de objetivos.

- Chequear si ya se ha demostrado que el nuevo subobjetivo:
  - puede ser deducido a partir de α.
  - no puede ser deducido a partir de α.

- Encadenamiento hacia atrás vs Encadenamiento hacia adelante

  - **EHD está dirigido por los datos**, apropiado para adquisición de conocimiento sin un objetivo específico en mente.
  - **EHA está dirigido por el objetivo**, apropiado para resolver problemas específicos. La complejidad del EHA puede ser mucho menor que lineal (en tamaño de la BC).

  Un agente debería usar EHD para generar hechos que probablemente serán relevantes para preguntas gestionadas por EHA

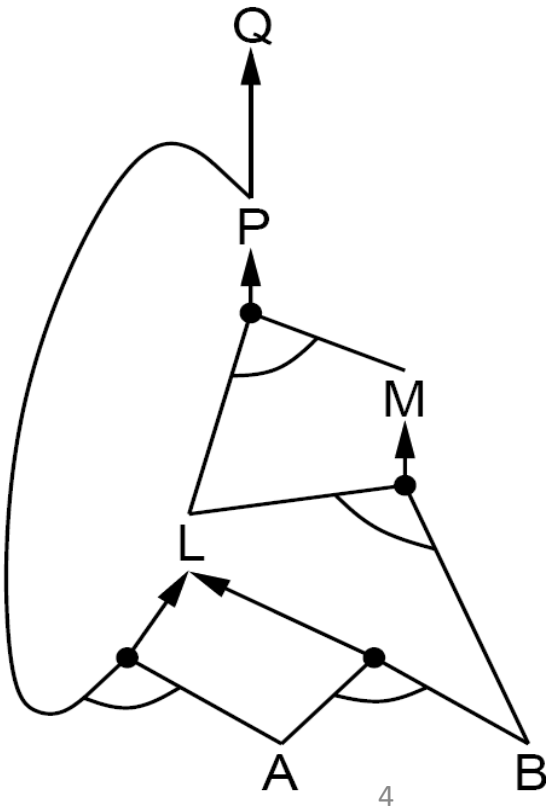
EHD/EHA: diagrama AND-OR

- Ejemplo (de Russel & Norvig)

  Considérese un agente cuya base de conocimiento es

  Hechos: A, B

  Reglas: $P \Rightarrow Q$, $L \wedge M \Rightarrow P$, $B \wedge L \Rightarrow M$, $A \wedge P \Rightarrow L$, $A \wedge B \Rightarrow L$



Consecuencia lógica por EHD/EHA

Considérese $\Delta = \{ P, N, P \Rightarrow Q, Q \Rightarrow R, R \wedge S \Rightarrow M, N \Rightarrow S \}$, $w = M \wedge R$
¿Se cumple $\Delta \models w$?

- Construir $\alpha = \Delta \cup \neg w$
- Nótese que $\neg w$ es una cláusula de Horn
  $\neg (M \wedge R) \equiv \neg M \vee \neg R \equiv M \wedge R \Rightarrow F$
- Mostrar que α is insatisfacible por o bien:

  - Encadenamiento hacia delante:
    Hechos: $\{P, N\}$
    $\{P, N, Q, S\}$ [por las reglas $P \Rightarrow Q, N \Rightarrow S$]
    $\{P, N, Q, S, R\}$ [por la regla $Q \Rightarrow R$]
    $\{P, N, Q, S, R, M\}$ [por la regla $R \wedge S \Rightarrow M$]
    $\{P, N, Q, S, R, M, F\}$ [por la regla $M \wedge R \Rightarrow F$]

  - Encadenamiento hacia atrás:
    F puede ser inferido de $M[1], R[2]$
    (1) M puede ser inferido de $R[1.1]=[2], S[1.2]$
    (1.2) S puede ser inferido de $N[1.2.1]$
    (1.2.1) N está en la base de conocimiento.
    (2) R puede ser inferido de Q [2.1]
    (2.1) Q puede ser inferido de P [2.1.1]
    (2.1.1) P está en la base de conocimiento.

PROGRAMACIÓN LÓGICA

Consideremos la siguiente definición del concepto “lista ordenada”, que asume que los conceptos “lista vacía”, “resto de la lista”, “primero de la lista” y “menor o igual” están ya definidos:

$(\forall\, l)$ $\quad (\text{Vacia}(l) \Rightarrow \text{Ordenada}(l))$ (a)

$(\forall\, l, r)$ $\quad (\text{Resto}(l, r) \wedge \text{Vacia}(r) \Rightarrow \text{Ordenada}(l))$ (b)

$(\forall\, l, r, x, y)$ $\quad (\text{Resto}(l, r) \wedge \text{Primero}(l, x) \wedge \text{Primero}(r, y) \wedge$ (c)

$\wedge\ \text{Menor\_o\_igual}(x, y) \wedge \text{Ordenada}(r)) \Rightarrow \text{Ordenada}(l))$

Una lista l está ordenada si cumple una de las siguientes condiciones:

(a) no tiene elementos

(b) tiene un elemento

(c) tiene más de un elemento, en cuyo caso el primer elemento debe ser menor o igual que el segundo, y además la lista resto de l (lista que se obtiene quitando el primer elemento) está ordenada

PROGRAMACIÓN LÓGICA

En LPO:

$(\forall\, l)$ $(\text{Vacia}(l) \Rightarrow \text{Ordenada}(l))$ (a)

$(\forall\, l, r)$ $(\text{Resto}(l, r) \wedge \text{Vacia}(r) \Rightarrow \text{Ordenada}(l))$ (b)

$(\forall\, l, r, x, y)$ $((\text{Resto}(l, r) \wedge \text{Primero}(l, x) \wedge \text{Primero}(r, y) \wedge$ (c)

$\wedge\, \text{Menor\_o\_igual}(x, \text{y}) \wedge \text{Ordenada}(r)) \Rightarrow \text{Ordenada}(l))$

Esta definición se puede implementar en Prolog de manera directa.

En Prolog (:

```
ordenada([]).
ordenada([_]).
ordenada([X|[Y|Zs]]) :- X =< Y, ordenada([Y|Zs]).
ordenada([1,2,3]).
```

Sintaxis y terminología de Prolog

- Los **predicados** y **constantes** empiezan con minúscula
- Las **variables** empiezan con mayúscula
- Son variables lógicas, no celdas de memoria que pueden cambiar su contenido
- Si una unificación *vincula* una variable con un valor, eso impide vínculos con otro valor:

  `p(X,X)` no unifica con `p(3,2)`
- La variable anónima es `_` (subrayado)

  (la única que unifica con cualquier caso y no recuerda su vínculo)
- **Términos:** no admiten funciones como argumentos:

  `relacionados(2,3,5)` se admite

  `relacionados(2,3,f(2,3))` no se admite
- **Expresiones:** se pueden formar con operadores comunes (`+`, `-`, `*`, …) pero son muy especiales. Prolog no es un lenguaje de expresiones sino de predicados.

Sintaxis y terminología de Prolog: Reglas

- Cada línea de un programa Prolog es la definición de una regla:

```
consecuente(X,Y,…) :- antecedente1(X,Y,…), antecedente2(…).
```

- El :- es la implicación hacia la izquierda.
- El "," que separa los antecedentes se interpreta como ^
- La regla termina siempre con un punto.
- En el consecuente y antecedentes pueden aparecer variables.
- Se sobreentiende que todas las variables están cuantificadas universalmente en cada regla.
- Puede haber reglas sin antecedentes:

```
es_divisor(X,X).
```

- Las reglas admisibles en Prolog son, por tanto, cláusulas de Horn:

$$((a_1 \wedge a_2 \wedge \ldots \wedge a_n) \Rightarrow c) \equiv (\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n \vee c)$$

Sintaxis y terminología de Prolog: LISTAS

- Listas: se forman con corchetes y comas, o también con `|`
- `[a,b]` es equivalente a `[a|[b]]`
- Todas estas listas son equivalentes:

  `[a,b,c]` `[a|[b,c]]` `[a|[b|[c]]]` `[a|[b|[c|[]]]]`

- Pero:

  `[a,b,c]` ≠ `[a,[b,c]]` (lista de dos elementos, uno de ellos otra lista)

  `[a,b,c]` ≠ `[a|[b|c]]` (lista impropia)

- Las listas se pueden anidar: `[[a,1],[b,2],[c,3]]` (lista de tres listas)

- Las listas se pueden usar como argumentos de predicados:

  `ordenada([3,5,9])`

Sintaxis y terminología de Prolog (cont.) LISTAS

- Correspondencia Prolog ↔ Lisp:

Prolog	Lisp
[a,b]	(a b)
[a | b]	(a . b)
[a,[b,c]]	(a (b c))
[[a,b] | c]	((a b) . c)
[]	( ) ≡ NIL
[c | []] ≡ [c]	(c)
[b | [c]] ≡ [b, c]	(b c)
[a | [b c]] ≡ [a, b, c]	(a b c)

Unificación en Prolog

- Variables para unificación: `simetrica([X,_,Y])`
- **`p([A|B])`** unifica con **`p([1,2,3])`** `si A=1 B=[2,3]` (“vínculos”)
- **`p([A|B])`** NO unifica con **`q([1,2,3])`**
- **`p([A|_])`** unifica con **`p([1,2,3])`** si `A=1`
- **`p([_|B])`** unifica con **`p([1,2,3])`** si `B=[2,3]`
- **`p([_])`** **NO** unifica con **`p([1,2,3])`**
- **`p([_|_])`** unifica con **`p([1,2,3])`**
- **`p([_|_])`** **SÍ** unifica con **`p([1])`**

Ejemplo Familia: Más Hechos (datos) y Más Consultas

```
padre(luis, maria).
madre(rosa, maria).
madre(rosa, pedro).
? padre(luis, X)
```

Contesta `X=maria`

```
? madre(rosa, X)
```

Contesta `X=maria` y con *;* le podemos pedir otra respuesta, a la que contesta `X=pedro`

Si volvemos a pedir otra *;* contesta `no`

Programas Prolog con Hechos y Reglas para Ejemplo Familia

```
padre(luis, juan).
padre(luis, maria).
madre(rosa, maria).
madre(rosa, pedro).
madre(maria, oscar).
abuelo(X,Y) :- padre(X,Z), padre(Z,Y).
abuelo(X,Y) :- padre(X,Z), madre(Z,Y).
herederoDe(X,Y) :- padre(Y,X).
herederoDe(X,Y) :- madre(Y,X).
```

¿Qué resultados darían las siguientes consultas?

```
? herederoDe(luis, juan)
? herederoDe(maria, Y)
? herederoDe(X, maria)
? abuelo(luis, N)
? abuelo(A, oscar)
? abuelo(A, N)
```

```
ordenada([]).                                          Regla 1
ordenada([_]).                                         Regla 2
ordenada([X|[Y|Zs]]) :- X =< Y, ordenada([Y|Zs]).      Regla 3
```

- Una vez definido el predicado “ordenada”, podemos utilizarlo para chequear si una lista concreta está ordenada (“realizamos una consulta”):

```
?ordenada([])  yes
?ordenada([1]) yes
?ordenada([1,2])  yes
?ordenada([2,2])  yes
?ordenada([2,1])  no
?ordenada([1,2,3]) yes
?ordenada([1,3,2]) no
```

```
ordenada([]).                                          Regla 1
ordenada([_]).                                         Regla 2
ordenada([X|[Y|Zs]]) :- X <= Y, ordenada([Y|Zs]).      Regla 3
```

- ¿Cómo hace esto Prolog? Busca una demostración de la consulta con las reglas definidas, usando encadenamiento hacia atrás.

- De manera informal:

  - ordenada([4,7]) se demostraría por la Regla 3, en caso de que los dos antecedentes, 4<=7 y ordenada([7]), se cumplan.

  El primero se cumple, el segundo hay que demostrarlo:

  - ordenada([7]) se demuestra por la Regla 2.

  Por tanto, los dos antecedentes de R3 se cumplen, con lo que ordenada([4,7]) es **cierto**.

```
ordenada([]).                                          Regla 1
ordenada([_]).                                         Regla 2
ordenada([X|[Y|Zs]]) :- X <= Y, ordenada([Y|Zs]).      Regla 3
```

A nivel detallado, esto es lo que haría Prolog si realizamos la consulta ?ordenada([1,2,3])

- `ordenada([1,2,3])` no se puede unificar con el consecuente de R1
- Tampoco se puede unificar con el consecuente de R2
- En cambio sí se puede unificar con el consecuente de R3:

  `{X:=1, Y:=2, Zs:=[3]}`

  Lanzamos dos nuevas consultas:

  `1<=2`, que se cumple

  `ordenada([2,3])`, que hay que demostrar. Lanzamos entonces esta nueva consulta.

  - `ordenada([2,3])` no se puede unificar con el consecuente de R1 ni con el de R2.
  - Sí con el de R3:

    `{X':=2, Y':=3, Zs':=[]}` (se renombran las variables ya que conceptualmente son variables diferentes a las del paso anterior).

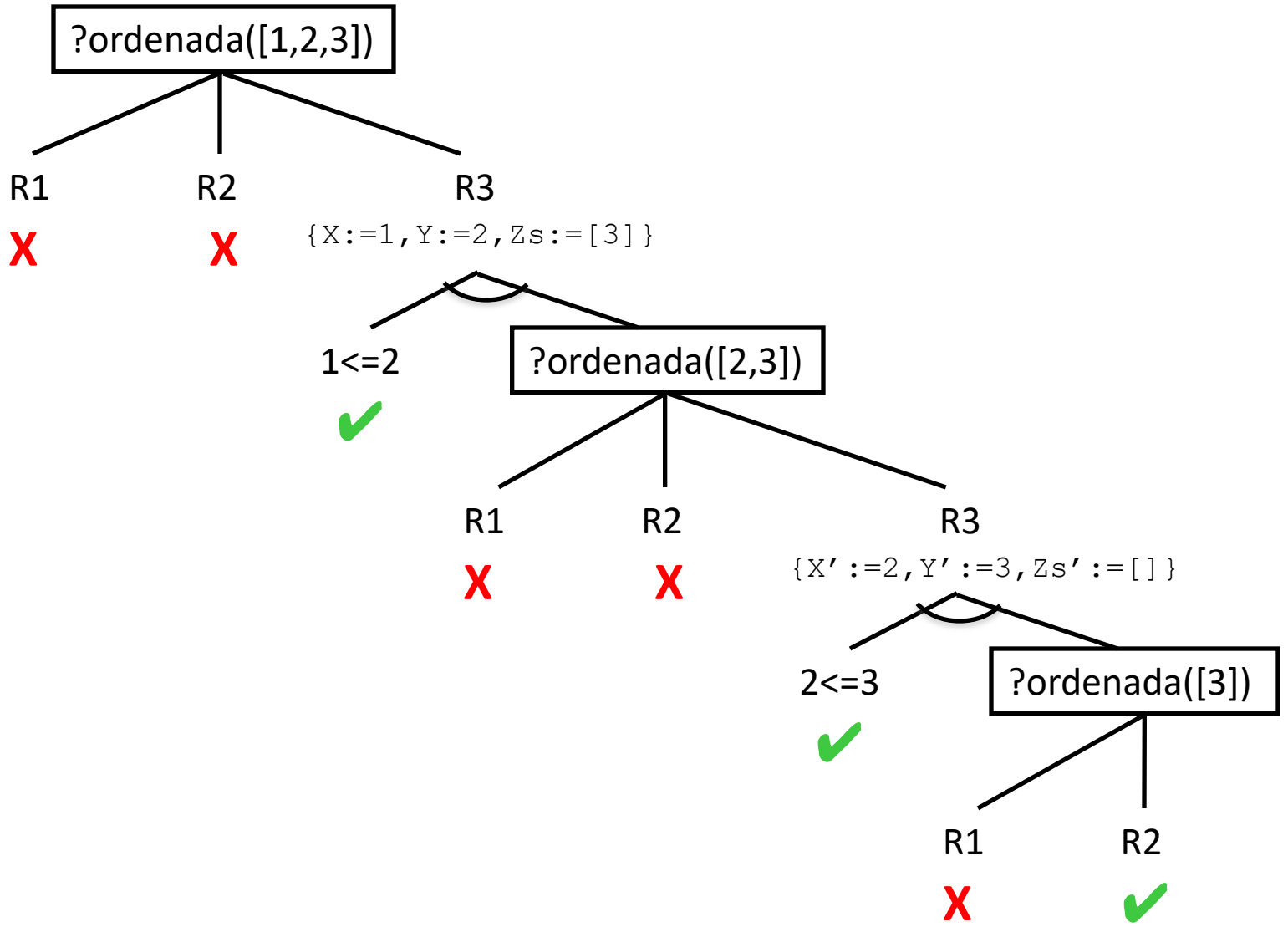
    Lanzamos dos nuevas consultas:

    `2<=3`, que se cumple.

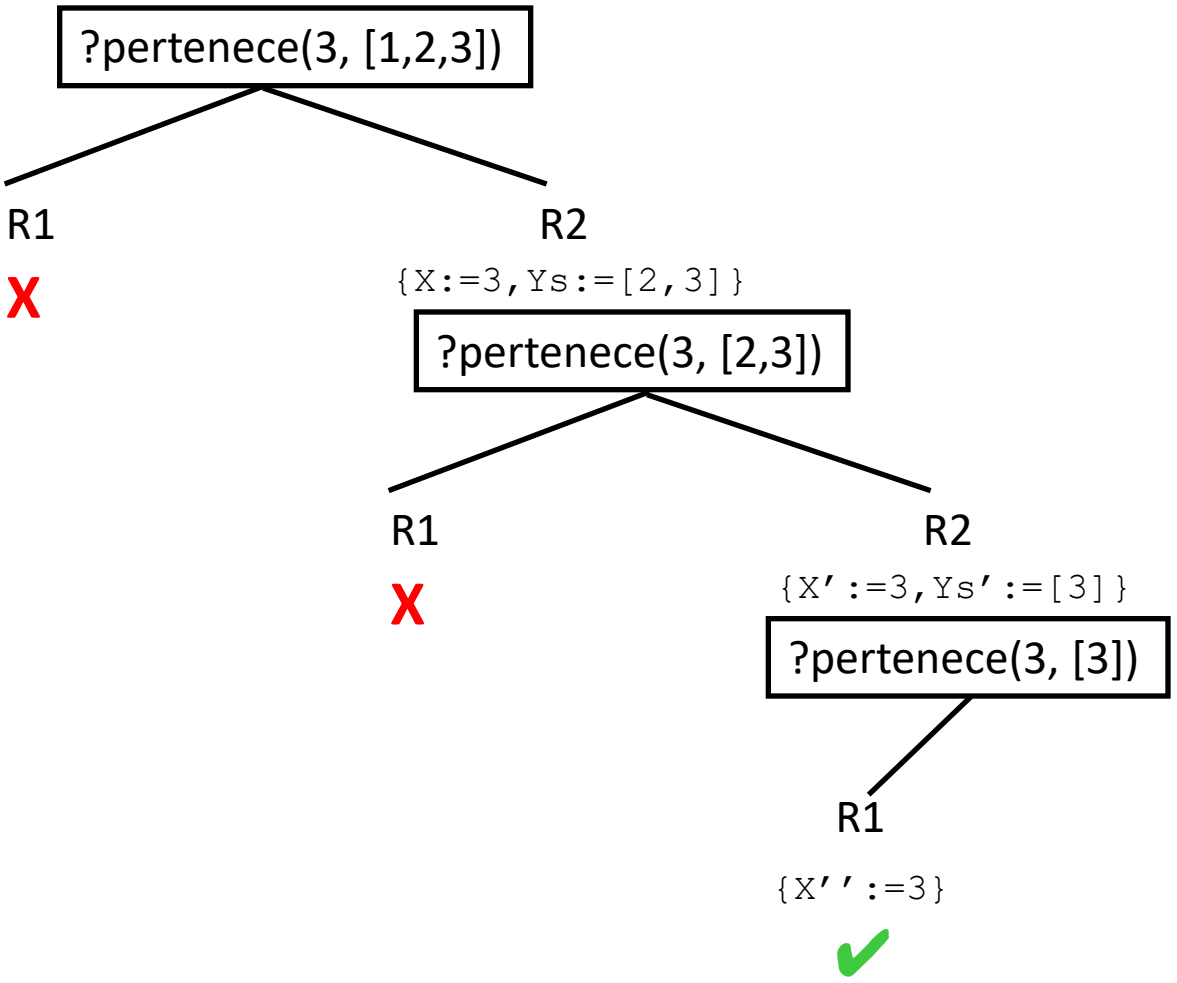
    `ordenada([3])`, que hay que demostrar. Lanzamos esta nueva consulta.

    No se puede unificar con el consecuente de R1 pero sí con el de R2. Como no hay antecedentes, queda entonces demostrada la consulta, y por tanto ordenada([2,3]), y por tanto ordenada([1,2,3])

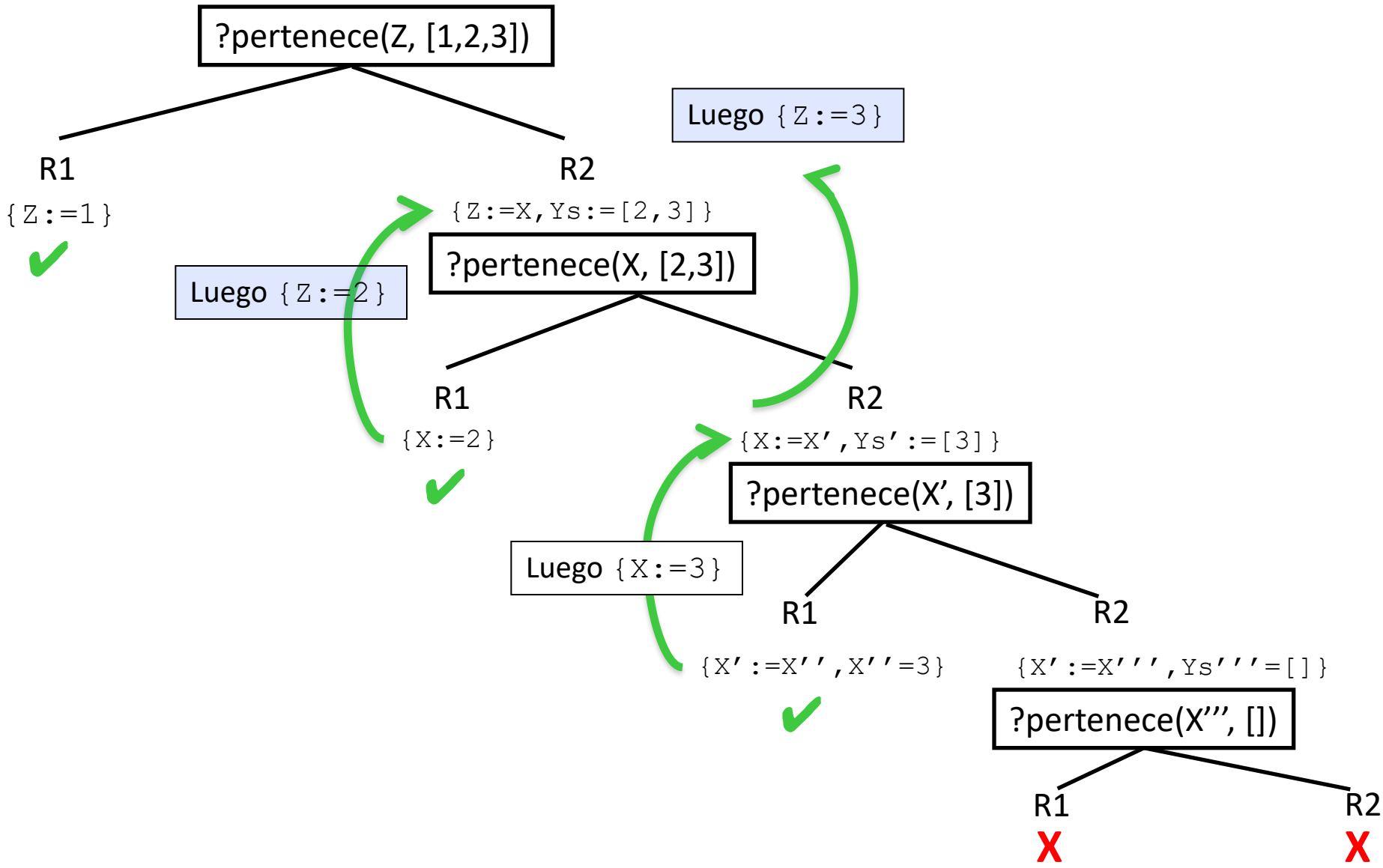
```
ordenada([]).                                   Regla 1
ordenada([_]).                                  Regla 2
ordenada([X|[Y|Zs]]) :- X <= Y, ordenada([Y|Zs]).   Regla 3
```



```
pertenece(X,[X|_]).                          Regla 1
pertenece(X,[_|Ys]) :- pertenece(X,Ys).      Regla 2
```



```
pertenece(X,[X|_]).                    Regla 1
pertenece(X,[_|Ys]) :- pertenece(X,Ys).    Regla 2
```



Operadores

- El operador “=“ unifica la expresión a su izquierda con la de la derecha, introduciendo vínculos nuevos. Si es imposible la unificación, devuelve fallo.

```
es_primer_elemento(E,[E|_]).
```

es equivalente a

```
es_primer_elemento(E,[A|_]) :- E=A.
```

- Los operadores “> “, “<“, “>=“, “<=“, “\=“ (“diferente de”) dan fallo si no se cumplen. No introducen vínculos nuevos.

```
diferentes(X,Y,Z) :- X\=Y, Y\=Z.
```

- El operador “is” realiza dos cosas: 1) evalúa la expresión de la derecha, y 2) unifica el resultado de dicha evaluación con lo que hay a su izquierda.

```
consecutivos(N,M) :- N is M+1.
```
[comparar con: `consecutivos(N,M) :- N = M+1`]

Si la expresión de la izquierda del “is” contiene alguna variable no vinculada, dará error de ejecución. Ejemplos de operaciones a la derecha del “is”:

M+N, X-Y, A*B, C/D, E mod F

Programas Prolog propuestos

- Predicado pertenencia de elemento en lista
- Encontrar el camino en un grafo
- Predicado “sin duplicados” de una lista
- Predicado suma de los elementos de una lista