Cláusulas de Horn.
Resolución SLD

Cláusulas de Horn

La estrategia común para aligerar las dificultades computacionales asociadas al uso del lenguaje de primer orden consiste en restringir el lenguaje,

buscando un compromiso entre expresividad y tratabilidad.

Cláusulas de Horn

La restricción del lenguaje de primer orden típica de la programación lógica consiste en considerar exclusivamente cláusulas como estas:

$$
\begin{array}{lll}
p_1 \wedge \cdots \wedge p_n \rightarrow & q & (\textit{positivas o definidas}) \\
p_1 \wedge \cdots \wedge p_n \rightarrow & \neg q & (\textit{negativas}).
\end{array}
$$

- Los símbolos p_i y q no representan aquí proposiciones únicamente, sino que pueden ser predicados con argumentos cualesquiera (*átomos*).
- Los cuantificadores se han eliminado (suponiendo implícitos los universales y recurriendo a funciones de Skolem para los existenciales).

Cláusulas de Horn

La restricción del lenguaje de primer orden típica de la programación lógica consiste en considerar exclusivamente cláusulas como estas:

$$\begin{aligned} p_1 \wedge \cdots \wedge p_n &\rightarrow q \\ \neg(p_1 \wedge \cdots \wedge p_n) &\vee q \\ \neg p_1 \vee \cdots \vee \neg p_n &\vee q \end{aligned} \qquad (\textit{positivas o definidas})$$

$$\begin{aligned} p_1 \wedge \cdots \wedge p_n &\rightarrow \neg q \\ \neg(p_1 \wedge \cdots \wedge p_n) &\vee \neg q \\ \neg p_1 \vee \cdots \vee \neg p_n &\vee \neg q \end{aligned} \qquad (\textit{negativas}).$$

- Una cláusula de Horn es una en la que todas las literales son negaciones, salvo quizás una.
- Alfred Horn [1951]

Resolución con cláusulas de Horn

El conjunto de cláusulas de Horn es cerrado para el cálculo de resolventes.

Para que dos cláusulas de Horn den lugar a una resolvente, al menos una de ellas ha de ser positiva:

$$\frac{\neg\bullet \vee \cdots \vee \neg\bullet \vee \quad q \quad +}{\neg\bullet \vee \cdots \vee \neg\bullet \vee \neg q \quad -} \qquad \frac{\neg\bullet \vee \cdots \vee \neg\bullet \vee \neg\bullet \vee \quad q \quad +}{\neg\bullet \vee \cdots \vee \neg\bullet \vee \quad p \vee \neg\bullet \quad +}$$

$$\qquad\qquad -\qquad\qquad\qquad\qquad +$$

Resolución con cláusulas de Horn

El conjunto de cláusulas de Horn es cerrado para el cálculo de resolventes.

Para que dos cláusulas de Horn den lugar a una resolvente, al menos una de ellas ha de ser positiva:

$$\frac{\neg\bullet \vee \cdots \vee \neg\bullet \vee \quad q \quad +}{\neg\bullet \vee \cdots \vee \neg\bullet \vee \neg q \quad -}{\scriptstyle\quad -}$$

$$\frac{\neg\bullet \vee \cdots \vee \neg\bullet \vee \neg\bullet \vee \quad q \quad +}{\neg\bullet \vee \cdots \vee \neg\bullet \vee \quad p \vee \neg\bullet \quad +}{\scriptstyle\quad +}$$

El tipo de la otra cláusula coincide con el de la resolvente.

Resolución con cláusulas de Horn

Supongamos que

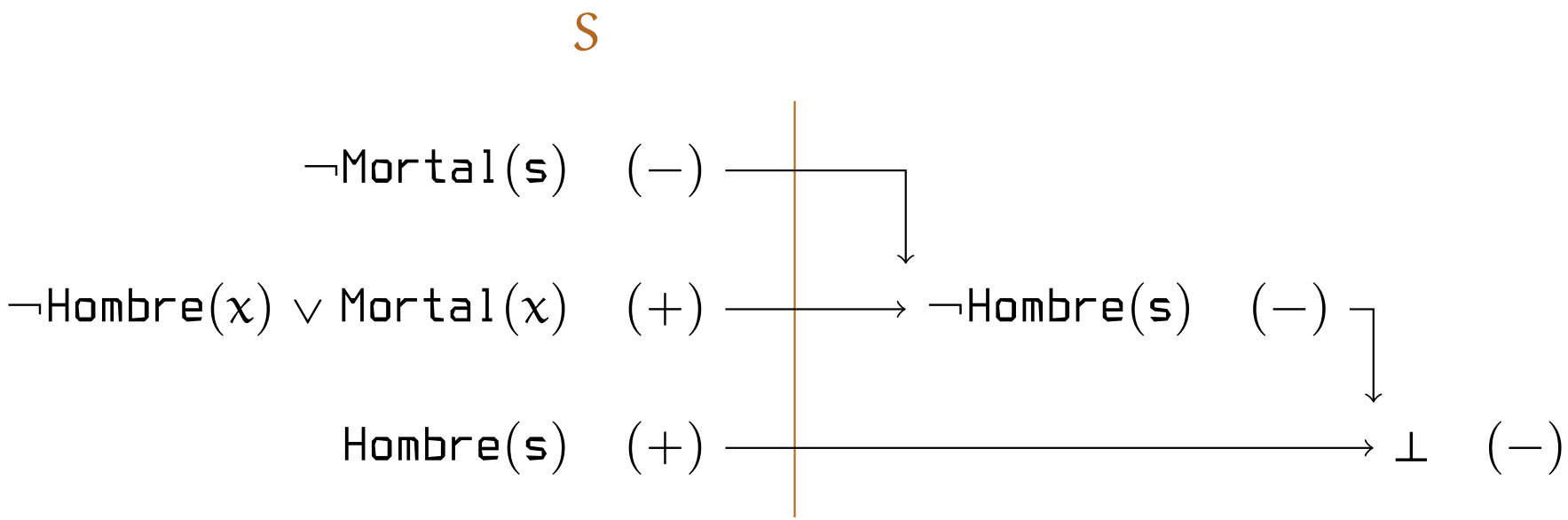
- S es un conjunto de cláusulas de Horn,
- c es una cláusula de Horn negativa (por ejemplo, $\bot$) y
- $S \vdash c$ (mediante resolución).

Entonces, existe una deducción de c en la que

- se parte de una cláusula negativa de S,
- cada cláusula nueva es negativa y
- cada cláusula nueva es la resolvente de la anterior $(-)$ de la deducción y de una cláusula $(+)$ de S.

Resolución con cláusulas de Horn

Por ejemplo, para demostrar $\texttt{Mortal}(\texttt{s})$ a partir de $\forall \texttt{x}\ (\texttt{Hombre}(\texttt{x}) \rightarrow \texttt{Mortal}(\texttt{x}))$ y $\texttt{Hombre}(\texttt{s})$:



Resolución con cláusulas de Horn

Este tipo de resolución se conoce como **SLD** (*selected literals, linear pattern, over definite clauses*).

Lo describió R. Kowalski en 1974 y K. Apt y M. van Emden lo denominaron así en 1982.

Resolución con cláusulas de Horn

Este tipo de resolución se conoce como **SLD** (*selected literals, linear pattern, over definite clauses*).

Lo describió R. Kowalski en 1974 y K. Apt y M. van Emden lo denominaron así en 1982.

Según lo anteriormente enunciado, el método de resolución puede restringirse a la resolución SLD si todas las cláusulas de partida son de Horn.

Resolución con cláusulas de Horn

Este tipo de resolución se conoce como **SLD** (*selected literals, linear pattern, over definite clauses*).

Lo describió R. Kowalski en 1974 y K. Apt y M. van Emden lo denominaron así en 1982.

Según lo anteriormente enunciado, el método de resolución puede restringirse a la resolución SLD si todas las cláusulas de partida son de Horn.

En el caso proposicional, en cada resolución se «gasta» una literal positiva del conjunto de partida.
En consecuencia, la longitud de una resolución SLD es lineal en el tamaño del conjunto de partida (en literales, que no en cláusulas).

Resolución con cláusulas de Horn

En el caso general del lenguaje de primer orden, la resolución SLD tampoco garantiza finalización.

Por ejemplo, supongamos una *base de conocimiento* formada por una única cláusula:

$$\texttt{Apellida}(\texttt{padre}(x), y) \rightarrow \texttt{Apellida}(x, y).$$

Si tratamos de averiguar si pedro se apellida pérez,

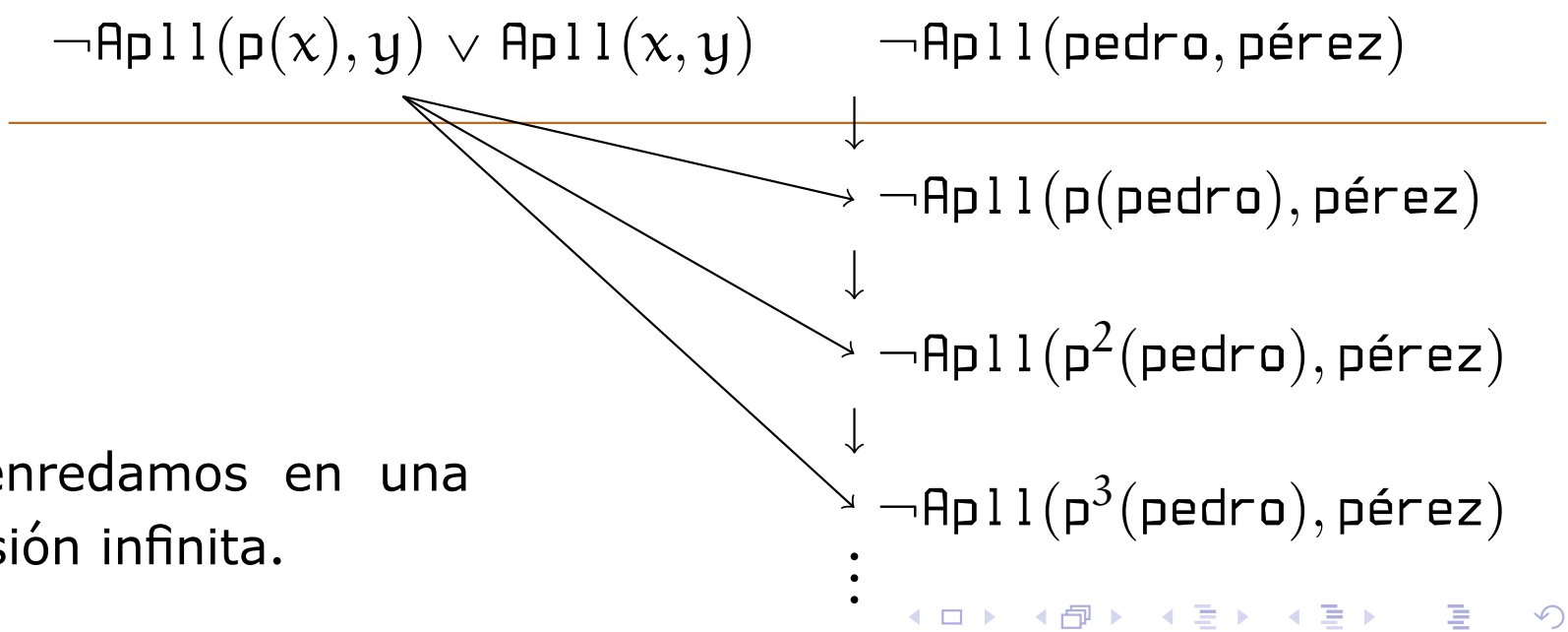
Resolución con cláusulas de Horn

En el caso general del lenguaje de primer orden, la resolución SLD tampoco garantiza finalización.

Por ejemplo, supongamos una *base de conocimiento* formada por una única cláusula:

$$\texttt{Apellida}(\texttt{padre}(x), y) \rightarrow \texttt{Apellida}(x, y).$$

Si tratamos de averiguar si pedro se apellida pérez, Índice



nos enredamos en una recursión infinita.

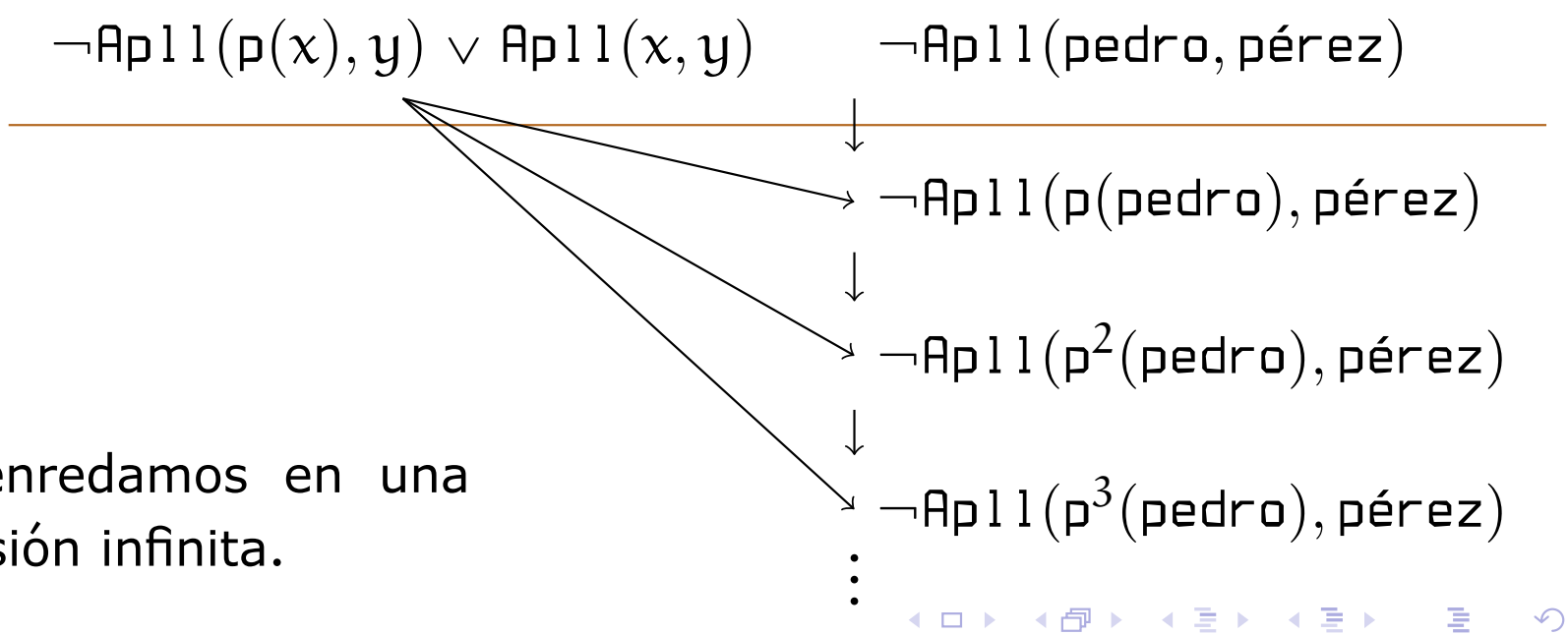
Resolución con cláusulas de Horn

La restricción del lenguaje de primer orden a las cláusulas de Horn sigue siendo **indecidible** (semidecidible).

Por ejemplo, supongamos una *base de conocimiento* formada por una única cláusula:

$$\texttt{Apellida}(\texttt{padre}(x), y) \rightarrow \texttt{Apellida}(x, y).$$

Si tratamos de averiguar si pedro se apellida pérez, Índice



nos enredamos en una recursión infinita.

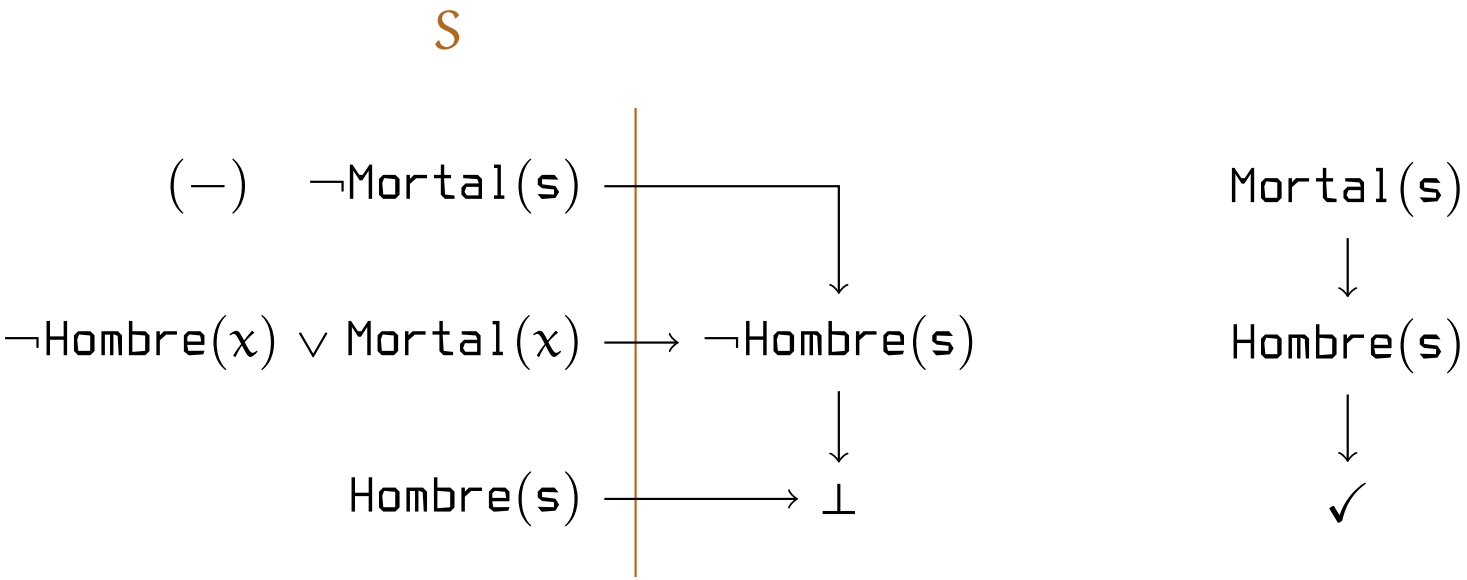
Cláusulas de Horn

A la vista de los ejemplos, podemos apuntar esta clasificación de las cláusulas de Horn.

$+$	$-$
$\bullet$ *fact*	$\neg\bullet$ *query*
$\neg\star \vee \cdots \vee \neg\star \vee \bullet$ *rule*	$\neg\bullet \vee \cdots \vee \neg\bullet$ *multiple query*
$(\star \wedge \cdots \wedge \star) \rightarrow \bullet$	$\neg(\bullet \wedge \cdots \wedge \bullet)$

Goal trees

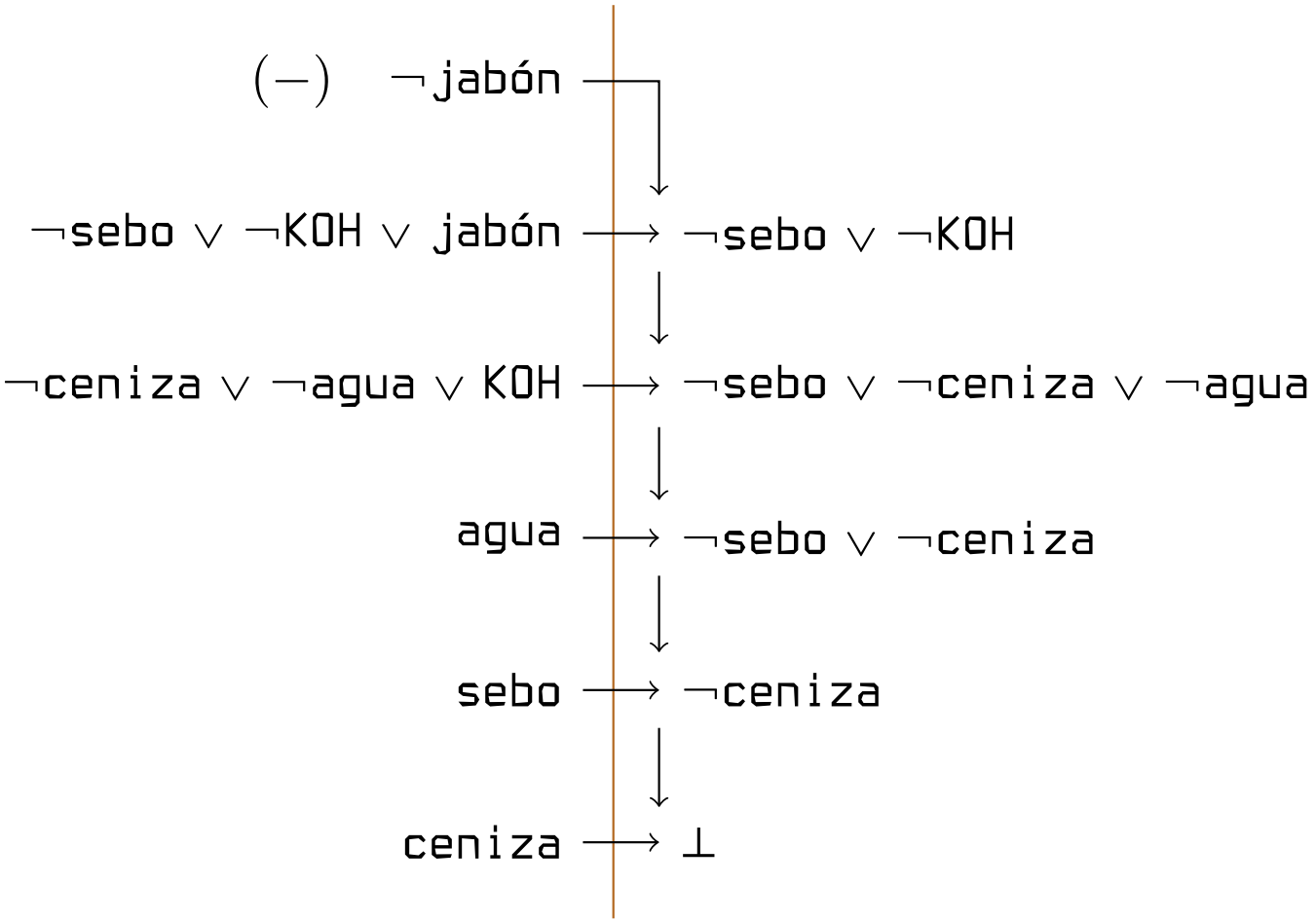
Además de la notación para las cláusulas que venimos utilizando, donde una consulta se traduce en una literal negativa, el mecanismo de resolución SLD puede representarse como un árbol de objetivos:

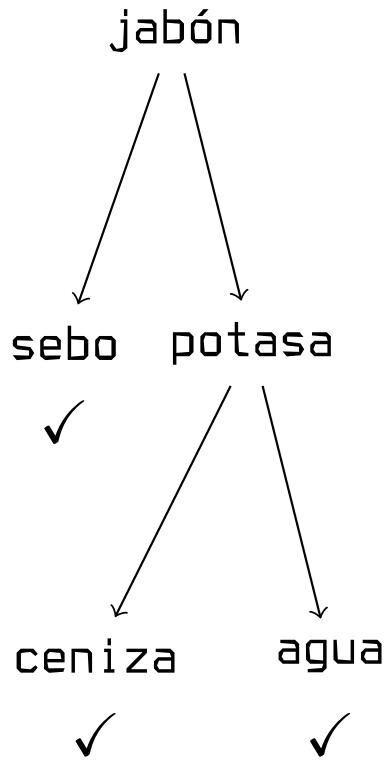


◂ Índice

Goal trees

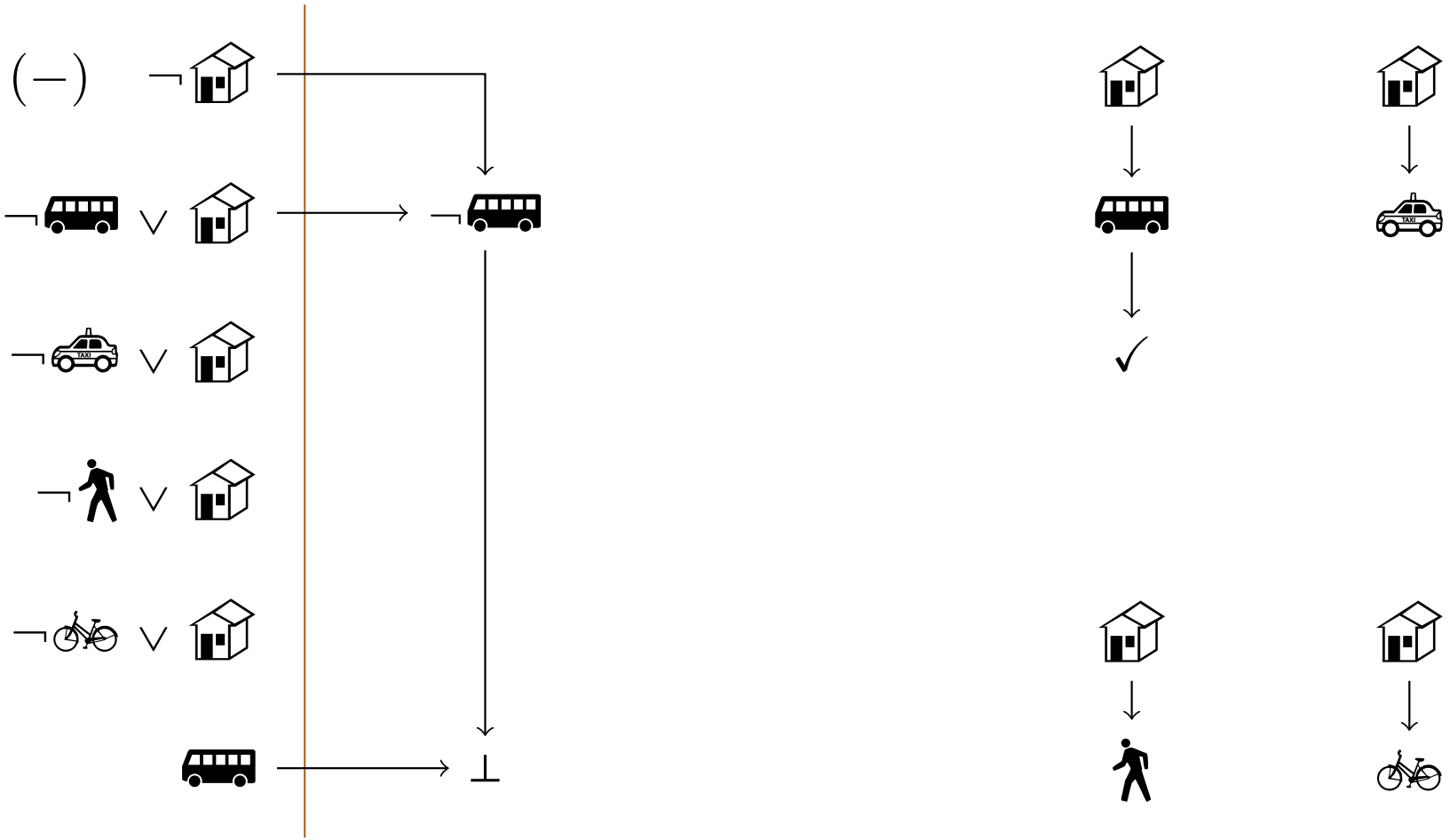
Además de la notación para las cláusulas que venimos utilizando, donde una consulta se traduce en una literal negativa, el mecanismo de resolución SLD puede representarse como un árbol de objetivos:





Goal trees

Además de la notación para las cláusulas que venimos utilizando, donde una consulta se traduce en una literal negativa, el mecanismo de resolución SLD puede representarse como un árbol de objetivos:



Restricción

Al restringir la sentencias lógicas que manejamos, perdemos la capacidad de tratar deducciones sencillas como esta:

$$\begin{array}{rcl} p & \rightarrow & q \\ \neg p & \rightarrow & q \\ \hline & & q \end{array}$$

La segunda implicación no es una cláusula de Horn.

Implicaciones direccionales

Las tres sentencias siguientes son semánticamente equivalentes: dicen lo mismo con formas distintas.

$$\begin{array}{rcl} p & \rightarrow & q \\ \neg p & \vee & q \\ \neg q & \rightarrow & \neg p \end{array}$$

Implicaciones direccionales

Las tres sentencias siguientes son semánticamente equivalentes: dicen lo mismo con formas distintas.

$$\begin{array}{rcl} p & \to & q \\ \neg p & \vee & q \\ \neg q & \to & \neg p \end{array}$$

Una estrategia para enfocar el uso computacional de una implicación consiste en «direccionarla»,

admitiendo una sola de sus dos vías de utilización.

Así, por ejemplo, la implicación $\texttt{Hombre}(x) \to \texttt{Mortal}(x)$ se utilizaría para deducir $\texttt{Mortal}(a)$ a partir de $\texttt{Hombre}(a)$,

dejando de lado su potencial aplicación para deducir que alguien no es hombre si se observa que sobrevive a una caída de doce pisos.

Implicaciones direccionales

En el contexto de los sistemas expertos, se utiliza el símbolo $\Rightarrow$ para indicar esa direccionalidad:

$$\texttt{Hombre}(x) \Rightarrow \texttt{Mortal}(x).$$

Así, por ejemplo, la implicación $\texttt{Hombre}(x) \rightarrow \texttt{Mortal}(x)$ se utilizaría para deducir $\texttt{Mortal}(a)$ a partir de $\texttt{Hombre}(a)$,

> dejando de lado su potencial aplicación para deducir que alguien no es hombre si se observa que sobrevive a una caída de doce pisos.

Backward chaining

En los últimos ejemplos,

- Se parte de una consulta (*query*) u objetivo:

$$\texttt{Mortal}(s), \quad \texttt{Apellida}(\texttt{pedro}, \texttt{pérez}).$$

- Esta consulta se traduce en una cláusula negativa de la que arranca la resolución SLD.

Backward chaining

En los últimos ejemplos,

- Se parte de una consulta (*query*) u objetivo:

$$\texttt{Mortal}(s), \quad \texttt{Apellida}(\texttt{pedro}, \texttt{pérez}).$$

- Esta consulta se traduce en una cláusula negativa de la que arranca la resolución SLD.
- Las cláusulas positivas de la base de conocimiento se emplean como implicaciones direccionales (**reglas**):

$$\neg\bullet \vee \cdots \vee \neg\bullet \vee \mathrm{q} \qquad\qquad \left(\bullet \wedge \cdots \wedge \bullet\right) \Rightarrow \mathrm{q}.$$

  Para demostrar el objetivo (*goal*) q, se intentan resolver todos los objetivos $\bullet, \ldots, \bullet$.

Backward chaining

En los últimos ejemplos,

- Se parte de una consulta (*query*) u objetivo:

$$\texttt{Mortal}(s), \quad \texttt{Apellida}(\texttt{pedro}, \texttt{pérez}).$$

- Esta consulta se traduce en una cláusula negativa de la que arranca la resolución SLD.
- Las cláusulas positivas de la base de conocimiento se emplean como implicaciones direccionales (**reglas**):

$$\neg\bullet \vee \cdots \vee \neg\bullet \vee \mathrm{q} \qquad\qquad (\bullet \wedge \cdots \wedge \bullet) \Rightarrow \mathrm{q}.$$

  Para demostrar el objetivo (*goal*) q, se intentan resolver todos los objetivos $\bullet, \ldots, \bullet$.
- Un objetivo queda resuelto si viene afirmado por un ***fact*** de la base de conocimiento ($\bullet$, que es $\top \to \bullet$).

Backward chaining

Este esquema avanza desde la consecuencia de una implicación direccional hacia sus antecedentes,

retrotrayéndose hasta conseguir (o no) demostrarla.

Se denomina **backward chaining.**

Backward chaining

Este esquema avanza desde la consecuencia de una implicación direccional hacia sus antecedentes,

retrotrayéndose hasta conseguir (o no) demostrarla.

Se denomina **backward chaining.**

Típicamente, se corresponde con un sistema que toma un objetivo como entrada (*goal-directed*), tratando de encontrar en la base de conocimiento datos (*facts*) que lo sostengan.

Prolog

Prolog es un lenguaje de programación basado en la resolución SLD y *backward chaining*:

- Creado por A. Colmerauer y Ph. Roussel en 1972.
- Utiliza la **programación lógica** (un tipo de programación declarativa).

  No se trata de un lenguaje declarativo *puro*: es necesario tener en cuenta su comportamiento procedimental.

- La especificación de la tarea determina el mecanismo para llevarla a cabo.

  Aunque distintas maneras de escribir la misma especificación lógica dan lugar a resultados radicalmente distintos.

Prolog

Un programa consiste en una serie de **reglas**:

$$\texttt{Head} \Leftarrow \texttt{body}.$$

La cabeza se interpreta como un objetivo que se alcanzará si se pueden verificar los requisitos del otro término.
Además, en el programa se pueden establecer ***facts***:

La última línea equivale a

Prolog

Algunos ejemplos:

- Sócrates es mortal
- Saponificación
- Patronímico
- Volver a casa

$\longrightarrow$ orden de las líneas del código

Backward chaining

Opera de esta manera el sistema **GNU make**, que se emplea para compilar programas con una estructura compleja.

Los ficheros `Makefile` están formados por *reglas* que indican cómo generar un fichero objetivo (*target*) y de qué ficheros (*prerequisites*) depende.

Así, cuando se le pide construir o reconstruir cierto *target*, *make* comprueba sus prerrequisitos. Cada uno de ellos se construye, recursivamente, si falta o su fecha de modificación es anterior a la de alguno de los ficheros de los que depende.

Forward chaining

Se conoce como *forward chaining* la estrategia opuesta al *backward chaining*.

Se procede desde los *asertos* conocidos hacia sus consecuencias, siguiendo el sentido de las implicaciones direccionales: ***data-directed*** en vez de *goal-directed*.

Así, en vez de obtener objetivos o hipótesis intermedias, se trabaja con conclusiones intermedias.

Forward chaining

Se conoce como *forward chaining* la estrategia opuesta al *backward chaining*.

Se procede desde los *asertos* conocidos hacia sus consecuencias, siguiendo el sentido de las implicaciones direccionales: ***data-directed*** en vez de *goal-directed*.

Así, en vez de obtener objetivos o hipótesis intermedias, se trabaja con conclusiones intermedias.

Este esquema no se corresponde con el método de resolución.

En particular, no persigue un objetivo concreto, sino que va calculando todo lo que se deriva de los datos de partida.

Sistemas de producción

Sistemas de producción

Sistema experto: trata de emular a un experto humano.

Se ha tratado [A. Newell y H. Simon, 1972] de modelizar el proceso de una persona para obtener conclusiones como

> la aplicación de múltiples reglas sencillas, que forman su **base de conocimiento**.
>
> se oye un ladrido $\Rightarrow$ debe de ser un perro.

Sistemas de producción

Sistema experto: trata de emular a un experto humano.

Se ha tratado [A. Newell y H. Simon, 1972] de modelizar el proceso de una persona para obtener conclusiones como

> la aplicación de múltiples reglas sencillas, que forman su **base de conocimiento**.
>
> se oye un ladrido $\Rightarrow$ debe de ser un perro.

Un **sistema de producción** es un sistema experto basado en reglas de producción.

Sistemas de producción

Al estudiar la derivación SLD y Prolog,

> hemos reunido con un tratamiento uniforme los conceptos de *facts* y *rules*.

Ahora hacemos una distinción marcada entre

- la **base de conocimiento** compuesta por las reglas con las que contamos y
- la ***working memory*** (volátil) con los *facts* que manejamos en un determinado momento.

Sistemas de producción

Al estudiar la derivación SLD y Prolog,

hemos reunido con un tratamiento uniforme los conceptos de *facts* y *rules*.

Ahora hacemos una distinción marcada entre

- la **base de conocimiento** compuesta por las reglas con las que contamos y
- la ***working memory*** (volátil) con los *facts* que manejamos en un determinado momento.

Un sistema de producción opera mediante *forward chaining*,

aplicando las reglas que disparen los *facts*,

hasta que ya no pueda aplicarse ninguna.

Sistemas de producción

Esta estructura se ajusta al modelo cognitivo de A. Newell y H. Simon, utilizando reglas de producción para representar el conocimiento.

memoria a largo plazo	base de conocimiento (reglas)
memoria a corto plazo	*working memory* (datos)
procesador cognitivo	motor de inferencia

Ejemplo I

CLIPS

El ejemplo anterior está escrito en **CLIPS** (*C Language Integrated Production System*),

una herramienta de desarrollo de sistemas expertos.

Su sintaxis se basa en la de **LISP** (lenguaje de programación funcional por antonomasia), que tiene gran ascendiente sobre la inteligencia artificial.

Ejemplo I

La regla se ejecuta una sola vez, aunque la condición que la dispara sigue presente:

para evitar bucles, las reglas son *refractarias* (*refractory*).

Ejemplo II

La *refracción* no impide que una regla se dispare varias veces, en base a distintos *facts*.

Ejemplo II

La *refracción* no impide que una regla se dispare varias veces, en base a distintos *facts*.

Para comprobar si una pareja concreta de nodos están conectados, es más conveniente el método de *backward chaining* que hemos visto.

Ejemplo III

Este sistema de *forward chaining*, en cambio, resulta muy conveniente para calcular todos los vértices de una componente conexa.

Ejemplo III

Si las dos condiciones del antecedente estuvieran en el orden inverso, el proceso sería menos eficiente.

Reglas de producción

Una **regla de producción** (o simplemente **producción**) consta de

- Antecedente (*left-hand side (LHS)*, terminología de CLIPS)

  Sucesión de elementos condicionales,

  que se combinan mediante **conjunción.**

  El elemento condicional típico es

  un patrón (***pattern***) al que debe amoldarse algún *fact.*

- Consecuente (*right-hand side (RHS)*, terminología de CLIPS)

  Sucesión de *acciones.*

Reglas de producción

Las acciones del consecuente se ejecutan sucesivamente, como en programación procedimental.

De acuerdo con la naturaleza dinámica de la *working memory*, las siguientes son acciones típicas:

- añadir un *fact*,

```
(assert (ejemplo))
```

- eliminarlo,

```
(retract (ejemplo))
```

- o modificarlo.

```
(modify <n.º de fact> (campo nuevo_valor))
```

Sistemas de producción

Cada una de las reglas de producción funciona independientemente de las demás,

sobre unos datos (*working memory*) comunes a todas.

Se puede establecer una analogía con un **programa paralelo**,

aunque las reglas se disparan de una en una.

Sistemas de producción

El ciclo de funcionamiento de un sistema de producción consta de tres pasos:

- **identificar** las reglas aplicables,
- escoger una de ellas y
- **ejecutarla** (*fire it*).

Sistemas de producción

El ciclo de funcionamiento de un sistema de producción consta de tres pasos:

- **identificar** las reglas aplicables,

  Se **activan** las reglas cuyos antecedentes se satisfagan (según los *facts* de la *working memory*).

  CLIPS coloca estas activaciones en la *agenda*.

- escoger una de ellas y

- **ejecutarla** (*fire it*).

Sistemas de producción

El ciclo de funcionamiento de un sistema de producción consta de tres pasos:

- **identificar** las reglas aplicables,

  Se **activan** las reglas cuyos antecedentes se satisfagan (según los *facts* de la *working memory*).

  CLIPS coloca estas activaciones en la *agenda*.

- escoger una de ellas y

  Si hay varias activaciones, se recurre a un mecanismo de resolución de conflictos.

- **ejecutarla** (*fire it*).

Sistemas de producción

El ciclo de funcionamiento de un sistema de producción consta de tres pasos:

- **identificar** las reglas aplicables,

  Se **activan** las reglas cuyos antecedentes se satisfagan (según los *facts* de la *working memory*).

  CLIPS coloca estas activaciones en la *agenda*.

- escoger una de ellas y

  Si hay varias activaciones, se recurre a un mecanismo de resolución de conflictos.

- **ejecutarla** (*fire it*).

  Esto produce cambios en la *working memory*.

  Si se retira o modifica algún *fact* que hubiera activado alguna de regla aún no «disparada»,

  se elimina de la agenda esa activación.

Pattern matching

El cuello de botella de un sistema experto se localiza en la identificación de las reglas que procede activar.

Estos sistemas consideran muchos *facts* y muchas reglas a la vez.

En cada paso del ciclo, debe comprobarse si alguna combinación nueva de *facts* casa con el antecedente de cada una de las reglas.

Pattern matching

El cuello de botella de un sistema experto se localiza en la identificación de las reglas que procede activar.

Estos sistemas consideran muchos *facts* y muchas reglas a la vez.

En cada paso del ciclo, debe comprobarse si alguna combinación nueva de *facts* casa con el antecedente de cada una de las reglas.

La invención del algoritmo **Rete** [Ch. Forgy, 1982] para el sistema OPS5 marcó un hito decisivo para los sistemas de producción.

Este algoritmo aprovecha

- que son pocos los cambios en la *working memory* de una etapa a la siguiente y
- que reglas distintas pueden compartir condiciones en sus antecedentes.

Rete

La idea básica consiste en llevar un registro, en todo momento, del «punto» de satisfacción de los antecedentes de las reglas,

> en vez de comprobar todas las condiciones cada vez que se modifica la *working memory*.

Rete

La idea básica consiste en llevar un registro, en todo momento, del «punto» de satisfacción de los antecedentes de las reglas,

en vez de comprobar todas las condiciones cada vez que se modifica la *working memory*.

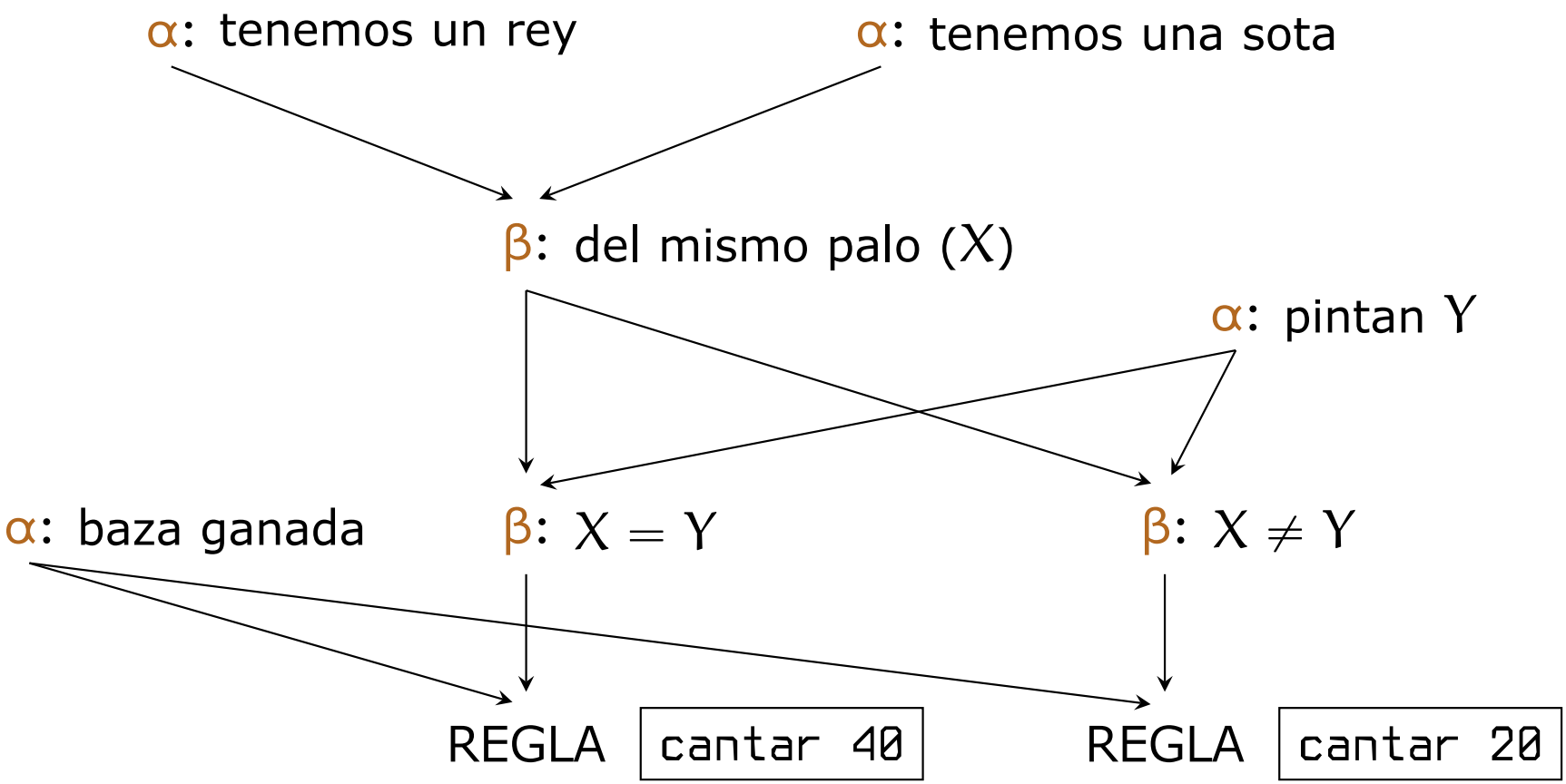
Esas condiciones se organizan en una red.

- Está compuesta por nodos de dos tipos:
  - α) Admiten los «facts» que cumplan cierta condición.
  - β) Imponen condiciones que involucran a varios «facts».
- Cada *fact* que se añade a la *working memory* recorre la red hasta donde alcanza.
- Si se modifica alguno, puede que avance

  (o que tenga que retroceder por dejar de cumplir alguna condición).

Rete

Valgan como ejemplo estas instrucciones para los cantes del guiñote:



En cualquier caso, es importante el orden de los antecedentes de una regla,

> como muestra el ejemplo de *Expert Systems. Principles and Programming*, de J. Giarratano y G. Riley [sec. 11.5].

La segunda regla es mucho más costosa computacionalmente.

Pattern matching

En cualquier caso, es importante el orden de los antecedentes de una regla.

En general, se pueden sugerir los siguientes criterios:

- Anteponer los patrones más específicos,

  como ilustra el ejemplo anterior.

Pattern matching

En cualquier caso, es importante el orden de los antecedentes de una regla.

En general, se pueden sugerir los siguientes criterios:

- Anteponer los patrones más específicos,

  como ilustra el ejemplo anterior.

- Posponer los patrones de *facts* volátiles.

  Con esto, se consigue reducir los cambios frecuentes en las concordancias parciales con los antecedentes de las reglas.

Pattern matching

En cualquier caso, es importante el orden de los antecedentes de una regla.

En general, se pueden sugerir los siguientes criterios:

- Anteponer los patrones más específicos,

  como ilustra el ejemplo anterior.

- Posponer los patrones de *facts* volátiles.

  Con esto, se consigue reducir los cambios frecuentes en las concordancias parciales con los antecedentes de las reglas.

- Anteponer los patrones que se ajustan a pocos *facts*.

Se trata solo de indicaciones generales: estos criterios pueden ser contradictorios.

Resolución de conflictos

En la guía del usuario de CLIPS, de J. Giarratano, leemos

> *Now you might say, "Well, I'll just design my expert system so that only one rule can possibly be activated at one time. Then there is no need for conflict resolution". [...]*
>
> *The bad news is that this success proves that your application can be well represented by a sequential program.*
>
> *So you should have coded it in C, Java, or Ada in the first place and not bothered writing it as an expert system.*

Resolución de conflictos

Una **estructura de control rígida** en un sistema experto

(su diseño establece muchas relaciones de prioridad de unas reglas con respecto a otras),

puede revelar un programa secuencial subyacente

(y la conveniencia de recurrir a la programación convencional).

Resolución de conflictos

- El término «producción» se encuentra en el sistema de reescritura de cadenas formalizado por E. Post [1943].

  Las reglas de este sistema no se organizaban bajo ninguna estructura de control o priorización.

- A. Markov Jr. [1954] propuso un sistema de producción con reglas jerarquizadas linealmente,

  > del mismo modo que prioriza Prolog las distintas alternativas para satisfacer un objetivo.

Salience

Una manera de decidir qué regla de la agenda ejecutar es

recurrir a una prioridad explícita dada por el programador:

En CLIPS, las reglas tienen *salience* `0` por defecto.

En el ejemplo del cálculo numérico, hemos priorizado la regla que termina el proceso.

Salience

Leemos en el libro *Expert Systems. Principles and Programming*, J. Giarratano y G. Riley [p. 453]:

> *People who are just learning rule-based programming tend to overuse salience because it gives them explicit control of execution. It is more like the procedural programming they are used to [...]*
>
> *Overuse of salience results in a poorly coded program. The main advantage of a rule-based system is that the programmer does not have to worry about controlling execution.*

Salience

Leemos en el libro *Expert Systems. Principles and Programming*, J. Giarratano y G. Riley [p. 453]:

> *Salience should primarily be used as a mechanism for determining the order in which rules fire. This means that in general a rule that is placed on the agenda is eventually fired. Salience should not be used as a method for selecting a single rule from a group of rules when patterns can be used to express the criteria for selection, nor should it be used as a "quick fix" to get rules to fire in the proper order.*

Resolución de conflictos

Si la agenda contiene varias reglas activadas con la misma *salience*, CLIPS se decanta por una según el criterio seleccionado:

- `depth`
  Opción por defecto. El sistema «se concentra» en lo que está haciendo.
- `breadth`
  Todas las reglas tienen ocasión de intervenir.
- `simplicity`
- `complexity`
  Se priorizan las reglas con antecedente más específico.
- ...
- `random`
  Puede ser útil para detectar problemas en el sistema.

Otro criterio de elección atiende al **orden** en que están definidas las reglas (como hace Prolog).

Resolución de conflictos

breadth

Sistemas de producción

Esta manera de programar presenta se caracteriza por

- la sencillez de sus estructuras de control y
- la transparencia de su funcionamiento

  (comparándola, por ejemplo, con el de las redes neuronales).

Facilita, además del resultado, una **explicación** de la cadena de deducciones que ha conducido hasta él.