Programación Lógica

Tema 3

Francisco José Correa Zabala
fcorrea@eafit.edu.co

Departamento de informática y Sistemas
Universidad Eafit

Medellín, Colombia. Febrero de 2012

Objetivos

Reconocer argumentos fundamentales de la programación lógica.

Presentar los argumentos que sustentan y definen la programación lógica.

Utilizar el lenguaje Prolog como un lenguaje de programación.

Extender los aprendizajes para ganar fortalezas para el desarrollo de software en programación orientada a objetos.

Agenda

1. Lógica de predicados.
2. Cláusulas de Horn.
3. Programación Lógica.
4. Prolog.
5. Java.

Lógica de primer Orden

1. Sintaxis: Lenguaje de primer orden $\mathcal{L} = (\mathcal{A}, \mathcal{F})$:
   Alfabeto (A): Conjunto de símbolos que permiten construir las fórmulas del lenguaje.
   Conjunto de formulas o expresiones ($\mathcal{F}$)**:** objetos y propiedades de esos objetos expresadas mediante fórmulas.

2. Semántica de los lenguajes de primer orden: **Interpretación**, **Teoría de Modelos**

3. Cálculo deductivo: **axiomas** y **reglas de inferencia**

4. El problema: Dado una fórmula G y un conjunto de fórmulas CF, se trata de probar que dicha fórmula G es un consecuencia de CF.

$$CF \models G$$

Sintaxis: Lenguaje de primer orden $\mathcal{L} = (\mathcal{A}, \mathcal{F})$

Alfabeto ($\mathcal{A}$): Conjunto de símbolos que permiten construir las fórmulas del lenguaje.

Comunes a todos los formalismos:

- ✓ Conjunto infinito numerable de *Variables* ($\mathcal{V}$): . . . x, y, z . . .
- ✓ *Conectivas lógicas*: $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$
- ✓ *Cuantificadores*. $\forall, \exists$
- ✓ *Signos de puntuación y agrupación*: , ; ( ) { }

Símbolos propios de un formalismo:

- ✓ *Constantes* ($\mathcal{C}$): . . . a, b, c . . .
- ✓ *Símbolos de función* ($\mathcal{F}$): . . . f, g, h . . .
- ✓ *Símbolos de predicados* ($\mathcal{P}$): . . . p, q, r . . .

Cada símbolo de función $\mathcal{F}$ o de predicado $\mathcal{P}$ tiene asociado un número natural, denominado aridad, que indica el número de argumentos.

Sintaxis: Lenguaje de primer orden $\mathcal{L} = (\mathcal{A}, \mathcal{F})$
Conjunto de formulas($\mathcal{F}$)

1. **Términos $\mathcal{T}$:**
   - Cada variable o constante del alfabeto $\mathcal{A}$ es un término t, es decir, si $t \in \mathcal{V} \cup \mathcal{C}$.
   - Si f es una función n-aria y $t_1, t_2, \ldots, t_n$ son términos entonces $f(t_1, t_2, \ldots, t_n)$ es un término.

2. **Formulas Bien Formadas, fbf:**
   - Si p es un símbolo de predicado n-ario y $t_1, t_2, \ldots, t_n$ son términos entonces $p(t_1, t_2, \ldots, t_n)$ es un átomo y todo átomo es una fbf.
   - Si F, F_1, F_2 son fbf's entonces: $\neg F$, $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \rightarrow F_2$, $F_1 \leftrightarrow F_2$, $\exists x F$, $\forall x F$ son fbf's.

Un término o fórmula es básica (ground) si no contiene variables.

Sintaxis: Lenguaje de primer orden $\mathcal{L} = (\mathcal{A}, \mathcal{F})$ Conjunto de formulas($\mathcal{F}$)

- ★ **Ocurrencia de una variable:** Es la aparición de una variable en una fórmula.
- ★ **Alcance de un cuantificador:** El alcance de $\forall x$ en la fórmula $\forall x F$ es F. Del mismo modo, El alcance de $\exists x$ en la fórmula $\exists x F$ es F.
- ★ **Ocurrencia libre de una variable.** Cualquier ocurrencia de variable que no sea ligada es libre.
- ★ **Fórmula cerrada.** Una fórmula es cerrada si no posee ocurrencias libres de variables.
- ★ **Fórmula abierta.** Una fórmula es abierta si posee al menos una ocurrencia libre de variable.

Lógica de primer Orden

1. Sintaxis: Lenguaje de primer orden $\mathcal{L} = (\mathcal{A}, \mathcal{F})$:
   Alfabeto (A): Conjunto de símbolos que permiten construir las fórmulas del lenguaje.
   Conjunto de formulas($\mathcal{F}$): objetos y propiedades de esos objetos expresadas mediante fórmulas.

2. Semántica de los lenguajes de primer orden: **Interpretación**, **Teoría de Modelos**

3. Cálculo deductivo: **axiomas** y **reglas de inferencia**

4. El problema: Dado una fórmula G y un conjunto de fórmulas CF, se trata de probar que dicha fórmula G es un consecuencia de CF.

Semántica de los lenguajes de primer orden:

Un lenguaje de primer orden nos permite "hablar" sobre un universo de discurso. En el:

- Las constantes denotan los **objetos** del universo de discurso.
- Los predicados denotan las **propiedades** sobre los objetos del universo de discurso.
- Las fbf son **enunciados o sentencias** sobre el universo de discurso.

La **semántica permite**:

- Dar significado a cada uno de los símbolos del alfabeto $\mathcal{A}$.
- Poder conocer el valor de verdad de cualquier fbf de $\mathcal{F}$.

Para ello se construye un contexto donde se puedan evaluar las fórmulas: Interpretación

Semántica de los lenguajes de primer orden:

Una **Interpretación** es $\mathcal{I} = (\mathcal{D}, \mathcal{T})$ donde $\mathcal{T}$ es un conjunto de funciones $\mathcal{T} = (\mathcal{K}, \mathcal{E})$:

- $\mathcal{D}$ **es un conjunto no vacío, denominado Dominio de** $\mathcal{I}$**.** En $\mathcal{D}$ las variables toman un valor y es el rango de variación de los cuantificadores.
- $\mathcal{K}$ **es un conjunto de funciones** que dan valor a los términos del lenguaje:
  Para cada **constante** a, $\mathcal{T}(a) \in \mathcal{D}$.
  funciones de aridad n se interpretan como funciones de $\mathcal{D}^n$ en $\mathcal{D}$.
- Los **predicados**/n, $\mathcal{E}$ **es el conjunto de relaciones** en $\mathcal{D}^n$.

  Si p es un predicado n-ario, entonces $E(p)$ se le denomina *extensión de* p *en la interpretación* $\mathcal{I}$ ($E(p) \subseteq D^n$).

Valor de verdad de una fórmula en una interpretación:

El valor de verdad de una fórmula cerrada en $\mathcal{I} = (\mathcal{D}, \mathcal{T})$ se define como sigue:

- Si G es una fórmula atómica cerrada, $G = p(t_1, \ldots, t_n)$ entonces G es $cierta$ en $\mathcal{I}$ si y sólo si: $< \mathcal{T}(t_1), \ldots, \mathcal{T}(t_n) > \in E(p)$ y $falsa$ en caso contrario.
- La fórmula $(\forall \mathbf{x})\mathbf{T}$ es $cierta$ si para cada valor de x en $\mathcal{D}$, la fórmula es $cierta$. Es decir, para cualquier valor $d \in \mathcal{D}$, se cumple que la fórmula cerrada que resulta al sustituir todas las ocurrencias de x por d en T –T[x/d]– es $cierta$ en I.
- La fórmula $(\exists \mathbf{x})\mathbf{T}$ es $cierta$ si para algún valor de x en $\mathcal{D}$, la fórmula es $cierta$.
- El valor de verdad de una fbf cerrada, se calcula de acuerdo con el conectivo.

Valor de verdad de una fórmula cerrada y compuesta

F_1	$\neg F_1$
T	F
F	T

F_1	F_2	$F_1 \wedge F_2$	$F_1 \vee F_2$	$F_1 \rightarrow F_2$	$F_1 \leftrightarrow F_2$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Ejemplos

1. Evaluar el valor de verdad de la fórmula $(\forall x)(\exists y)p(x,y)$ en la interpretación: $\mathcal{D} = \{1,2\}$ y la relación que interpreta a p es E(P) = $\{p(1,1), p(2,2)\}$

2. Evaluar el valor de verdad de la fórmula $(\forall x)(p(x) \rightarrow q(f(x),a))$ en la interpretación: $\mathcal{D} = \{1,2\}$ y la función de evaluación:
   * para a es $a \mapsto 1$
   * para f es $\{f(1) \mapsto 2, f(2) \mapsto 1\}$
   * para p es $E(p) = \{p(2)\}$
   * para q es $E(q) = \{q(1,1), q(1,2), q(2,2)\}$

Modelo de un Conjunto de Fórmulas

Una interpretación $\mathcal{I}$ de un lenguaje $\mathcal{L}$ **es modelo** de una fórmula G de $\mathcal{L}$ si y solo si G **se evalúa** a $cierto$ con respecto a $\mathcal{I}$ (i.e. $\models_{\mathcal{I}} G$)

Ejemplo: Sean $G_1 : (\forall x)(\exists y)p(x,y)$ y $G_2 : (\exists y)(\forall x)p(x,y)$ en $\mathcal{L}$. Sea $\mathcal{I} = (\mathcal{D}, \mathcal{T})$, donde $\mathcal{D} = \mathbb{N}$ y $p_{\mathcal{I}}$ es $\leq_{\mathbb{N}}$. Entonces $\mathcal{I}$ es modelo de G_1 pero no de G_2

Un fórmula G es **satisfacible (consistente)** si y solo si existe una interpretación $\mathcal{I}$ de $\mathcal{L}$ tal que $\mathcal{I}$ **es modelo** de G (i.e. $(\exists \mathcal{I})(\models_{\mathcal{I}} G)$)

Ejemplo: $(\forall x)(\exists y)p(x,y)$

Modelo de un Conjunto de Fórmulas

Un fórmula G es **insatisfacible (inconsistente)** si y solo si no existe una interpretación $\mathcal{I}$ que **sea modelo** de G (i.e. $(\nexists\mathcal{I})(\models_{\mathcal{I}} G)$)

Ejemplo: $(\exists x)(p(x) \wedge \neg p(x))$

Un fórmula G es **válida** si y solo si cada interpretación $\mathcal{I}$ **es modelo** de G (i.e. $\models G$)

Ejemplo: $(\exists x)(p(x) \vee \neg p(x))$

Una fórmula G es una **consecuencia lógica**[*] de un conjunto de fórmulas $G_1, G_2, \ldots, G_n$ si y sólo si todo modelo M de $G_1 \wedge G_2 \wedge \ldots \wedge G_n$ es modelo de G y escribimos $G_1, G_2, \ldots, G_n \models G$

Ejemplo: $\{p(a), (\forall x)(p(x) \rightarrow q(x))\} \models q(a)$

[*]También llamada **consecuencia semántica**.

Teorema de la Deducción

$G_1, G_2, \ldots, G_n \models G$ si y sólo si $G_1 \wedge G_2 \wedge \ldots \wedge G_n \cup \sim G$ es insatisfacible
si y sólo si $G_1 \wedge G_2 \wedge \ldots \wedge G_n \rightarrow G$ es válida

Tesis de Church

`La validez de una fórmula es, en general, indecidible`

Uno de los objetivos más importantes perseguidos por la lógica es el de demostrar que una fórmula G es consecuencia lógica de CF (i.e. probar que todo modelo de CF es también modelo de G), lo que es prácticamente imposible de abordar. La idea es buscar un procedimiento automático que lo haga.

Lógica de primer Orden

1. Sintaxis: Lenguaje de primer orden $\mathcal{L} = (\mathcal{A}, \mathcal{F})$:
   Alfabeto (A): Conjunto de símbolos que permiten construir las fórmulas del lenguaje.
   Conjunto de formulas($\mathcal{F}$): objetos y propiedades de esos objetos expresadas mediante fórmulas.

2. Semántica de los lenguajes de primer orden: **Interpretación**, **Teoría de Modelos**

3. Cálculo deductivo: **axiomas** y **reglas de inferencia**

4. El problema: Dado una fórmula G y un conjunto de fórmulas CF, se trata de probar que dicha fórmula G es un consecuencia de CF.

Sistema de Deducción en Lógica de primer Orden

Sistema formal de deducción S_F. Axiomas y reglas de inferencia.

Ejemplo:

Axiomas:

1) $\neg q \vee p$
2) $r \vee q$
3) $\neg r$

Reglas de Inferencia:

1) Si $(p \rightarrow r)$ y p entonces r
2) $\neg p \vee q$ equivale $p \rightarrow q$
3) $\neg(\neg p)$ equivale p

Sistema de Deducción en Lógica de primer Orden

Demostración. Es una sucesión $A_1, \ldots, A_n$ de fbf, en donde cada A_i es un axioma o el resultado de aplicar una regla de inferencia.

4)	$\neg(\neg \mathtt{r}) \vee \mathtt{q}$	de regla (3) y axioma (2)
5)	$\neg \mathtt{r} \rightarrow \mathtt{q}$	de regla (2) y resultado (4)
6)	$\mathtt{q}$	de (5), (3) y la regla (1)
7)	$\mathtt{q} \rightarrow \mathtt{p}$	de regla (2) y axima (1)
8)	$\mathtt{p}$	de (7), (6) y la regla (1)

Sistema de Deducción en Lógica de primer Orden

Teorema. Si G es el último miembro de una demostración. ($\vdash_{S_F} G$, G es demostrable.)

Ejemplo: En el ejemplo anterior p es un teorema. Es decir $\vdash_{S_F} \mathrm{p}$

Teoría. Es un conjunto de fbf, CF, de un lenguaje de primer orden.

Teorema en una teoría. Sea CF una teoría. $CF \vdash_{S_F} F$ sii si F es un teorema a partir de CF.

Teorema de corrección y completitud.

$$CF \models F \text{ si y solo si } CF \vdash_{S_F} F$$

Sistema de Deducción en Lógica de primer Orden

Teorema de Completitud

$$\models G \quad \Rightarrow \quad \vdash G$$

SEMÁNTICA		SINTAXIS
Validez $\models \mathbf{G}$	Teoría de la Deducción Semántica / Teoría de la Deducción Sintáctica	Demostrable $\vdash \mathbf{G}$
Consecuencia semántica $\mathbf{CF} \models \mathbf{C}$		Deducible $\mathbf{CF} \vdash \mathbf{C}$

Teorema de Corrección

$$\models G \quad \Leftarrow \quad \vdash G$$

Algunas propiedades de la lógica de predicados

Teorema. Sea G una fbf cerrada de $\mathcal{L}$ y sea $\mathcal{I}$ una interpretación. Entonces, ($\models_{\mathcal{I}} G$) o $\models_{\mathcal{I}} \neg G$.

Teorema *[insatisfacibilidad].* Un conjunto de fórmulas CF es insatisfacible si y sólo existe un fbf G tal que ($CF \vdash G \wedge \neg G$).

Teorema *[consistencia].* Un conjunto de fórmulas CF es consistente si y sólo si no existe un fbf G tal que ($CF \vdash G$) y $CF \vdash \neg G$.

Teoría de la deducción

Teorema. Sea $\mathcal{A}$ y $\mathcal{B}$ fbf cerradas y $\mathcal{T}$ un conjunto de fbf. Entonces $\mathcal{T} \cup \{\mathcal{A}\} \vdash \mathcal{B}$ si y solo si $\mathcal{T} \vdash \mathcal{A} \rightarrow \mathcal{B}$

Teorema. Sea $\mathcal{A}$ y $\mathcal{B}$ fbf cerradas. $\mathcal{A} \Leftrightarrow \mathcal{B}$ si y solo si $\{\mathcal{A}\} \models \mathcal{B}$ y $\{\mathcal{B}\} \models \mathcal{A}$.

Teorema. Sea $\mathcal{A}$ y $\mathcal{B}$ fbf cerradas. $\mathcal{A} \Leftrightarrow \mathcal{B}$ si y solo si $\{\mathcal{A}\} \vdash \mathcal{B}$ y $\{\mathcal{B}\} \vdash \mathcal{A}$.

Teorema Sea $\mathcal{A}$, $\mathcal{B}$ y $\mathcal{C}[\mathcal{A}]$ fbf cerradas y además $\mathcal{C}[\mathcal{A}]$ es un contexto en donde destacamos la presencia de $\mathcal{A}$. Si $\vdash \mathcal{A} \leftrightarrow \mathcal{B}$ entonces $\vdash \mathcal{C}[\mathcal{A}] \leftrightarrow \mathcal{C}[\mathcal{B}]$.

$$\frac{(\mathcal{A} \leftrightarrow \mathcal{B}), \mathcal{C}[\mathcal{A}]}{\mathcal{C}[\mathcal{B}]}$$

Teoría de la deducción

Teorema Sea CF un conjunto de fórmulas consistente. Si $CF \cup \neg G$ es inconsistente, entonces $(CF \cup G)$ consistente.

Modus Ponens

$$\frac{\mathcal{A}, \mathcal{A} \rightarrow \mathcal{B}}{\mathcal{B}}$$

Método de casos

$$\frac{\mathcal{A} \vee \mathcal{B}, \mathcal{A} \rightarrow \mathcal{C}, \mathcal{B} \rightarrow \mathcal{C}}{\mathcal{C}}$$

⋮

La lógica es correcta, completa y supone el principio de la contradicción pero no decidible.

Sistema formal: Propiedades

1. Completitud: si toda fórmula que "se sigue lógicamente" de un conjunto de premisas (es válida según la noción de verdad), es demostrable en el sistema a partir de dicho conjunto de premisas.

2. Corrección: si toda fórmula demostrable a partir de un conjunto de premisas, "se sigue lógicamente" de ellas.

3. Decidibilidad: si existe un procedimiento finito para comprobar si una formula es o no válida.

4. Consistencia: Cuando de él no se desprenden contradicciones.

Lógica de primer orden: formas normales.

La lógica de primer orden puede construirse a partir de dos símbolos primitivos: el símbolo $\neg$ y un símbolo del conjunto $\{\vee, \wedge, \rightarrow, \leftrightarrow\}$.

Para efectos de automatización de la lógica se han escogido tres símbolos estándares: $\neg, \vee, \wedge$. Una fórmula bien formada está en forma normal si sus símbolos conectores son: $\vee, \wedge$ y $\neg$ tiene el alcance de fórmulas atómicas.

Forma normal disyuntiva. Una fbf $\mathcal{A}$ está en forma normal disyuntiva sii $\mathcal{A} \equiv \mathcal{A}_1 \vee \mathcal{A}_2 \vee \ldots \vee \mathcal{A}_n$, con $n \geq 1$, y cada $\mathcal{A}_i$ es una conjunción de literales*.

Forma normal conjuntiva. Una fbf $\mathcal{A}$ está en forma normal disyuntiva sii $\mathcal{A} \equiv \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n$, con $n \geq 1$, y cada $\mathcal{A}_i$ es una disyunción de literales

*__Literal:__ Si G es una fórmula atómica. Entonces G es literal y $\neg G$ es también un literal.

Lógica de primer orden: formas normales.

Es posible definir un algoritmo para calcular la forma normal en la lógica proposicional. ¿Cuál es?

En el caso de la lógica de predicados, los cuantificadores se pueden colocar como primeros símbolos de la fbf. Para hacerlo se aplican las propiedades de los cuantificadores. Una forma normal que cumpla está propiedad se llama **forma normal prenexa**.

Es posible diseñar un algoritmo para transformar una fbf a su forma normal prenexa. ¿Cuál es?

Y para que esto?

Hacia la programación automática

Es posible realizar un procedimiento automático, por refutación. Es decir, para probar la fórmula $\mathcal{A}$ se prueba que la fórmula $\neg\mathcal{A}$. Para ello utilizamos el siguiente procedimiento:

1. La fórmula $\mathcal{A}$ se transforma en una forma normal prenexa de la forma $(Q_1X_1, Q_2X_2, \ldots, Q_nX_n)\mathcal{M}$

2. Transformar a $\mathcal{M}$ en una forma normal conjuntiva

3. Hallar la forma norma de Skolem

Pasos para la forma normal de Skolem: Sustituir la variable del primer cuantificador existencial por una constante o una función. Si no hay cuantificadores universales a su izquierda es una constante, en caso contrario es una función que tiene como parametros todas las variables cuantificadas a su izquierda. Borrar el cuantificador existencial y seguir.

Lógica de primer orden: formas normales.

forma normal de Skolem Una fbf $\mathcal{A}$ que esta escrita en forma normal conjuntiva es de Skolem, si todas las variables ligadas a cuantificadores existenciales han sido sustituidas por funciones de Skolem y eliminando después los cuantificadores existenciales.

La aridad de la función de Skolem depende del número de cuantificadores universales que tenga la expresión a transformar a lado izquierdo del cuantificador existencial que se quiera sustituir. Si la función tiene parámetros estos serán variables frescas.

Ejemplo. $(\exists x_1)(\forall x_2)(\forall x_3)(\exists x_4)((\neg p(x_1, x_3) \wedge q(x_3)) \vee r(x_2, x_4) \vee t(x_2))$

Existe un algoritmo para obtener la forma normal de Skolem de una fbf que está en forma normal conjuntiva.

Lógica de primer orden: formas normales.

Cláusula. Una cláusula es una disyunción finita de cero o más literales $\mathcal{A}$. Sean $L_1, L_2, \ldots, L_n$ un conjunto de literales y $x_1, x_2, \ldots, x_m$ el conjunto de todas las variables que aparecen en $L_1, L_2, \ldots, L_n$, la forma general de una cláusula es

$$(\forall x_1, x_2, \ldots, x_m)(L_1 \vee L_2 \vee \ldots \vee L_n)$$

Una fbf en forma estándar puede representarse como una conjunción de cláusulas.

Dado que los cuantificadores universales son los únicos que existen, se pueden suprimir y conservar el implícito de que todas las variables está, cuantificadas universalmente.

Hacia la Programación Lógica ...

Programación Lógica . . .

1. Lenguaje formal:
   Alfabeto: Variables, Constantes, s. predicado, s. estructuras
   Términos: Variables, Constantes y Estructuras (funciones).
   Reglas: Átomos, Literales, Hechos, cláusulas, predicados.

2. Cálculo deductivo:
   Axiomas: programas,
   Reglas de inferencia: Principio de resolución de Robinson.

3. Semántica:
   Interpretación: Herbrand
   Noción de verdad: pertenencia a la base de Herbrand.

Propiedades: Consistencia, Corrección, Completitud y decidibilidad.

Sintaxis: términos . . .

usando notación Prolog

Variables: comienzan con una letra mayúscula (o "_"), y puede incluir caracteres, números y "_"

```
X,  Im4u,  Mi_coche,  _,  _x,  _22
```

Constantes: comienzan con una letra minúscula, y puede incluir caracteres, números y "_" (entre comillas simples, cualquier cosa)

```
a,  juan,  juan_luis,  66,  'Juan Luis'
```

Sintaxis: términos . . . (2)

usando notación Prolog

Estructuras: encabezadas por un **functor** (como el nombre de una constante) seguidas de un número fijo de argumentos entre paréntesis

```
fecha(miercoles, Mes, 1996)
```

(los argumentos pueden ser a su vez variables, constantes o estructuras)

El número de argumentos de una estructura es su **aridad**

Los functores se suelen representar por *nombre/aridad*. Una constante se puede ver como un functor de aridad cero

Sintaxis: términos . . . (3)

usando notación Prolog

Las variables, constantes y estructuras se denominan en conjunto **términos** (de un lenguaje de primer orden). Son las estructuras de datos de un programa lógico

Ejemplos

Término	**Tipo**	**Functor principal**
pedro	constante	pedro/0
hora(minuto,segundo)	estructura	hora/2
par(Calvin,tigre(Hob))	estructura	par/2
Par(uno,dos)	ilegal	—
Par	variable	—

Sintaxis: formulas bien formadas fbf . . .

Atomos: un átomo es una expresión de la forma $p(t_1, t_2, \ldots, t_n)$ donde p es el **símbolo de predicado** del átomo (mismas convenciones que los functores), n es su aridad y t_1, t_2, . . . , t_n son términos. En este contexto los átomos se denominan también **Hechos:**

El símbolo de predicado de un átomo se denota por p/n

Literales: un literal es un átomo positivo o negativo.

Ejemplos

```
perro(nombre(rocky), color(negro)).
amigos('Ana', 'Juan').
```

Sintaxis: cláusulas . . .

Cláusula: es una fórmula de la forma: $\forall x_1 \ldots \forall x_n (L_1 \vee \ldots \vee L_n)$ donde cada L_i es un literal y $x_1 \ldots x_n$ son variables que aparecen en $L_1 \vee \ldots \vee L_n$.

Notación: Una cláusula $A_1 \vee \ldots \vee A_n \vee \sim B_1 \vee \ldots \vee \sim B_m$ es lógicamente equivalente a $B_1 \wedge \ldots \wedge B_m \rightarrow A_1 \vee \ldots \vee A_n$ y que escribiremos como

$$A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m$$
$$A_1; \ldots; A_n \leftarrow B_1, \ldots, B_m$$

Desde el ***punto de vista declarativo*** dice que si ocurre $B_1, \ldots, B_m$ entonces debe ocurrir $A_1; \ldots; A_n$.

Desde el ***punto de vista operacional*** dice que para resolver el problema $A_1; \ldots; A_n$ se debe resolver $B_1, \ldots, B_m$.

Sintaxis: cláusulas de Horn . . .

Cláusula de Horn (definidas): es una cláusula de la forma: $A \leftarrow B_1, \ldots, B_m$ donde $A, B_1, \ldots, B_m$ son literales positivos.

Tipos de Cláusulas:

Reglas Es una cláusula de Horn de la forma: $A \leftarrow B_1, \ldots, B_m$

Hecho Es una cláusula de Horn de la forma: $A \leftarrow$. Desde el punto de vista lógico

$$A \Leftrightarrow A \vee False \Leftrightarrow \neg True \vee A \Leftrightarrow True \rightarrow A$$

Objetivos Es una cláusula de Horn de la forma: $\leftarrow B_1, \ldots, B_m$. Desde el punto de vista lógico

$$\sim A \Leftrightarrow \sim A \vee False \Leftrightarrow A \rightarrow False$$

Sintaxis: cláusulas de Horn . . .

Las cláusulas de Horn son un subconjunto de la lógica de predicados:

La cláusula $trabaja(X) \vee desempleado(X)$ no es una cláusula de Horn.

La estructura “if - then - else” no se puede definir en Prolog. Es decir, la expresión

$$(p \Rightarrow q) \wedge (\neg p \Rightarrow r)$$

que da lugar a la cláusula, que no es una cláusula de Horn:

$$\neg(\neg p \vee q) \vee (p \vee r)$$

Ejemplos.

Dada la cláusula, el hecho y el objetivo.

♣. $(\forall \mathtt{X}, \mathtt{Y}, \mathtt{P}, \mathtt{M})(\mathtt{hermanas}(\mathtt{X}, \mathtt{Y}) \vee \neg \mathtt{mujer}(\mathtt{X}) \vee \neg \mathtt{mujer}(\mathtt{Y}) \vee \neg \mathtt{padres}(\mathtt{X}, \mathtt{P}, \mathtt{M}) \vee \neg \mathtt{padres}(\mathtt{Y}, \mathtt{P}, \mathtt{M}))$.

♣. $\mathtt{mujer}(\mathtt{marta})$.

♣. $\neg \mathtt{hermana}(\mathtt{lucia}, \mathtt{petra})$.

Se escriben, en su orden, como:

✓. $\mathtt{hermanas}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{mujer}(\mathtt{X}), \mathtt{mujer}(\mathtt{Y}), \mathtt{padres}(\mathtt{X}, \mathtt{P}, \mathtt{M}), \mathtt{padres}(\mathtt{Y}, \mathtt{P}, \mathtt{M})$.

✓. $\mathtt{mujer}(\mathtt{marta})$.

✓. $\leftarrow \mathtt{hermana}(\mathtt{lucia}, \mathtt{petra})$.

Observe que las cláusulas dadas son de Horn.

Prolog ...

Ejemplos: Hechos . . .

usando notación Prolog

Hechos: un hecho es una expresión de la forma $p(t_1, t_2, \ldots, t_n).$ donde $p(t_1, t_2, \ldots, t_n)$ es un átomo

Ejemplos

```
perro(nombre(ricky), color(negro)).
amigos('Ana', 'Juan').
```

Los átomos y términos se distinguen por el contexto:
`perro(nombre(ricky), color(negro))` es un átomo y
`color(negro)` es un término

Ejemplos: Reglas ...
usando notación Prolog

Reglas: una regla es una expresión de la forma

$$p_0(t_1, t_2, \ldots, t_{n_0}) \leftarrow p_1(t_1^1, t_2^1, \ldots, t_{n_1}^1), \ldots, p_m(t_1^m, t_2^m, \ldots, t_{n_m}^m).$$

La expresión a la izquierda de la flecha se llama la **cabeza** de la regla y las expresiones a la derecha forman el **cuerpo** de la regla. Los literales del cuerpo se llaman también **llamadas a procedimiento**

Ejemplo

```
comida(Primero, Segundo, Tercero) :-
        aperitivo(Primero),
        plato_principal(Segundo),
        postre(Tercero).
```

Ambos reglas y hechos se denominan **cláusulas**

Ejemplos: Reglas . . .

usando notación Prolog

Predicados: todas las cláusulas del programa cuyas cabezas tienen el mismo nombre y aridad forman la **definición** del predicado

Ejemplo

```
mascota(rocky).
mascota(X) :- animal(X), ladra(X).
mascota(X) :- animal(X), vuela(X).
```

El predicado `mascota/1` tiene 3 cláusulas (un hecho y dos reglas)

Ejemplos: Programa Prolog . . .
usando el Prolog

Ejemplo

```
mascota(X) :- animal(X), ladra(X).
mascota(X) :- animal(X), vuela(X).
animal(rocky).
ladra(rocky).
animal(piolin).
vuela(piolin).
animal(hob).
ruge(hob).
```

Objetivos:

?- mascota(piolin).
?- mascota(hob).
?- mascota(X).
?- animal(X).

Ejemplos: Programa Prolog . . .
usando el Prolog

Ejemplo 2

```
progenitor(libia,jaime).
progenitor(jaime,caty).
progenitor(clara,jose).
progenitor(tomas, jose).
progenitor(tomas,isabel).
progenitor(jose, ana).
progenitor(jose, patricia).
progenitor(patricia,jaime).
```

Objetivos:

?- progenitor(clara, jose).
?- progenitor(clara, jaime).
?- progenitor(X,jose).
?- progenitor(X, jose), progenitor(jose, Y).

Ejercicios: Programa Prolog . . .

usando el Prolog

Con el fin de entender como funciona el prolog vamos a consultar problemas simples.

Traer 3 ejemplos de programas prolog. Deben ser sencillos y fácil de explicar. Los programas no deben incluir listas. Apenas estamos empezando.

Por ejemplo.

Un programa que determine el mayor de dos números.

Programación Lógica . . .

1. Lenguaje formal:
   Alfabeto: Variables, Constantes, s. predicado, s. estructuras
   Términos: Variables, Constantes y Estructuras (funciones).
   Reglas: Átomos, Literales, Hechos, cláusulas, predicados.

2. Cálculo deductivo:
   Axiomas: programas,
   reglas de inferencia: Principio de resolución de Robinson.

3. Semántica:
   Interpretación: Herbrand
   Noción de verdad: pertenencia a la base de Herbrand.

Propiedades: Consistencia, Corrección, Completitud y decidibilidad.

Teoría de deducción para Lógica de Cláusula de Horn.

El sistema de deducción para HCL consta de una única regla de inferencia (la SLD-resolución) y ningún axioma.

La SLD-resolución es un refinamiento de la regla de resolución.

El principio de resolución es una regla de inferencia muy potente que incorpora varias reglas de inferencia de la lógica.

Teoría de deducción para HCL.

Proposiciones padres	**Cláusulas padres**	**Resolvente**	**Observación**
$\mathcal{C}_1 : p \vee q$ $\mathcal{C}_2 : \neg p$	$\mathcal{C}_1 : p \vee q$ $\mathcal{C}_2 : \neg p$	$\mathcal{C} : q$	*Silogismo Disjuntivo*
$\mathcal{C}_1 : p \to q$ $\mathcal{C}_2 : p$	$\mathcal{C}_1 : \neg p \vee q$ $\mathcal{C}_2 : p$	$\mathcal{C} : q$	*Modus ponens*
$\mathcal{C}_1 : p \to q$ $\mathcal{C}_2 : \neg q$	$\mathcal{C}_1 : \neg p \vee q$ $\mathcal{C}_2 : \neg q$	$\mathcal{C} : \neg p$	*Modus Tollens*
$\mathcal{C}_1 : p \to q$ $\mathcal{C}_2 : q \to r$	$\mathcal{C}_1 : \neg p \vee q$ $\mathcal{C}_2 : \neg q \vee r$	$\mathcal{C} : \neg p \vee r$	*Silogismo Hipotético*
$\mathcal{C}_1 : p \vee q$ $\mathcal{C}_2 : \neg p \vee q$	$\mathcal{C}_1 : p \vee q$ $\mathcal{C}_2 : \neg p \vee q$	$\mathcal{C} : q$	—
$\mathcal{C}_1 : p \vee q$ $\mathcal{C}_2 : \neg p \vee \neg q$	$\mathcal{C}_1 : p \vee q$ $\mathcal{C}_2 : \neg p \vee \neg q$	$\mathcal{C} : \neg p \vee p$ $\mathcal{C} : \neg q \vee q$	—
$\mathcal{C}_1 : \neg p$ $\mathcal{C}_2 : p$	$\mathcal{C}_1 : \neg p$ $\mathcal{C}_2 : p$	$\mathcal{C} : \square$	—

Principio de resolución de Robinson.

Aplicable a cláusulas. Se basa en el principio de reducción al absurdo.

$$\frac{\mathcal{A} \in \mathcal{C}_1, \neg\mathcal{A} \in \mathcal{C}_2}{(\mathcal{C}_1 - \{\mathcal{A}\}) \vee (\mathcal{C}_2 - \{\neg\mathcal{A}\})}$$

- $\mathcal{C}_1$ y $\mathcal{C}_2$ se denominan *cláusulas padre*.
- $\mathcal{A}$ y $\neg\mathcal{A}$ son los *literales resueltos*.
- A la cláusula obtenida se le llama *resolvente* de $\mathcal{C}_1$ y $\mathcal{C}_2$

De la Demostración de Teoremas a la Programación Lógica

Sea $\mathcal{P}$ un programa lógico y sea el objetivo $G : (\exists x_1, \ldots, x_n) F(t_1, \ldots, t_m)$. Entonces los procedimientos posibles son:

$$
\begin{aligned}
\mathcal{P} \models G \quad &\Leftrightarrow \quad \mathcal{P} \cup \neg G \text{ es insatisfacible} \\
&\Leftrightarrow \quad \mathcal{P} \cup \neg G \text{ es inconsistente} \\
&\Leftrightarrow \quad \models \mathcal{P} \rightarrow G \\
&\Leftrightarrow \quad \vdash \mathcal{P} \rightarrow G \\
&\Leftrightarrow \quad \mathcal{P} \cup \neg G \vdash \square
\end{aligned}
$$

Se desea probar que $\mathcal{P} \models (\exists x_1, \ldots, x_n) F(t_1, \ldots, t_m)$, para ello probamos que $\mathcal{P} \cup \neg(\exists x_1, \ldots, x_n) F(t_1, \ldots, t_m)$ es insatisfacible. Pero el objetivo $\neg G : (\forall x_1, \ldots, x_n) \neg F(t_1, \ldots, t_m)$. Asumiendo implícita la cuantificación universal queda $G :\leftarrow F(t_1, \ldots, t_m)$.

Al demostrar G interesan los valores que pueden tomar las variables.

Ejemplo.

Probar cada una de las siguientes afirmaciones

$$
\begin{aligned}
\{p \to \neg q, q\} \models \neg p \quad &\Leftrightarrow \quad \{p \to \neg q, q\} \cup \neg(\neg p) \text{ es insatisfacible} \\
&\Leftrightarrow \quad \{p \to \neg q, q\} \cup \neg(\neg p) \text{ es inconsistente} \\
&\Leftrightarrow \quad \models ((p \to \neg q) \wedge q) \to \neg p \\
&\Leftrightarrow \quad \vdash ((p \to \neg q) \wedge q) \to \neg p \\
&\Leftrightarrow \quad \{p \to \neg q, q\} \cup \neg\neg p \vdash \square
\end{aligned}
$$

Probemos que: $\{p \to \neg q, q\} \models \neg p$. Es decir, debemos probar que todo modelo para $\{p \to \neg q, q\}$ es modelo para $\neg p$.

Ejemplo.

En efecto, sea $\mathcal{I}$ una interpretación tal que $\models_{\mathcal{I}} \{p \rightarrow \neg q, q\}$

entonces $\models_{\mathcal{I}} p \rightarrow \neg q$ y $\models_{\mathcal{I}} q$.

Por lo tanto, $\not\models_{\mathcal{I}} \neg q$

y según la combinación de valores de verdad para la implicación se tiene que $\not\models_{\mathcal{I}} p$

lo que significa que $\models_{\mathcal{I}} \neg p$

Ejemplo.

Probemos que: $\{p \to \neg q, q\} \cup \neg(\neg p)$ es insatisfacible.

Es decir, probemos que $\{p \to \neg q, q, \neg(\neg p)\}$ es siempre falsa.

Utilice tablas de verdad

Ejemplo.

p	q	$p \to \neg q$	$(p \to \neg q) \wedge q$	$((p \to \neg q) \wedge q) \wedge p$
T	T	F	F	F
T	F	T	F	F
F	T	T	T	F
F	F	T	F	F

Ejemplo.

Probemos que: $\{p \rightarrow \neg q, q\} \cup \neg(\neg p)$ es inconsistente.

Es decir, probemos que el conjunto de $\{p \rightarrow \neg q, q, \neg(\neg p)\}$ genera una contradicción.

Ejemplo.

Supongamos que el conjunto $\{p \rightarrow \neg q, q, \neg(\neg p)\}$ es cierta para alguna interpretación $\mathcal{I}$

Si $\neg(\neg p)$ es cierta en $\mathcal{I}$, entonces p es cierta en $\mathcal{I}$

Dado $p \rightarrow \neg q$ es cierta en I y que p es cierta en I

se concluye $\neg q$ es cierta para $\mathcal{I}$.

Se sabe que $\neg q$ es cierto y que q es cierto en $\mathcal{I}$

y esto no es posible.

Ya que, no existe un modelo en donde $\neg q$ y q sean ciertos.

Ejemplo.

Probemos que: $\models ((p \to \neg q) \wedge q) \to \neg p$.

Es decir, probemos que $((p \to \neg q) \wedge q) \to \neg p$ es siempre cierta.

Utilice tablas de verdad

Ejemplo.

p	q	$p \to \neg q$	$(p \to \neg q) \wedge q$	$((p \to \neg q) \wedge q) \to \neg p$
T	T	F	F	T
T	F	T	F	T
F	T	T	T	T
F	F	T	F	T

Ejemplo.

Probemos que: $\vdash ((p \to \neg q) \wedge q) \to \neg p$.

Es decir, probemos que $((p \to \neg q) \wedge q) \to \neg p$ es un teorema.

Como es una implicación, suponga el antecedente y elabore una demostración para concluir $\neg p$

Ejemplo.

Supongamos $p \rightarrow \neg q$ y q.

Entonces, por definición "$\rightarrow$" se concluye $\neg p \vee \neg q$,

por la conmutatividad se obtiene $\neg q \vee \neg p$

y de nuevo por definición de "$\rightarrow$" tenemos que $q \rightarrow \neg p$.

Por modus ponens tenemos que $\neg p$.

Ejemplo.

Probemos que $\{p \to \neg q, q\} \cup \neg\neg p \vdash \square$

Para ello utilicemos como única regla de inferencia el principio de resolución de Robinson

Ejemplo.

$\mathcal{C}_1$: $\neg \mathrm{p} \vee \neg \mathrm{q}$
$\mathcal{C}_2$: q
$\mathcal{C}_3$: $\neg\neg p$

generamos los siguientes resolventes:

$\mathcal{C}_4$: $\neg q$ resolución de Robinson con $\mathcal{C}_1$ y $\mathcal{C}_3$
$\mathcal{C}_5$: $\square$ resolución de Robinson con $\mathcal{C}_2$ y $\mathcal{C}_5$

Calculemos los valores . . .

Sustituciones.

Sea $\mathcal{V}ar$ el conjunto de todas la variables y τ el conjunto de términos.

$$\begin{array}{rccc} \theta: & \mathcal{V}ar & \longrightarrow & \tau \\ & X & \longrightarrow & \theta(X) \end{array}$$

- Es habitual representar una **sustitución** como una aplicación finita de variables a términos, denotada por $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, donde:
  - las variables $X_1, \ldots, X_n$ son diferentes
  - para $i = 1, \ldots, n$, $X_i \not\equiv t_i$
- Un par X_i/t_i se llama un **enlace** (binding)
- Las sustituciones operan sobre *expresiones* (denotadas por E), i.e., un término, una secuencia de literales, o una cláusula

Sustituciones(2).

La aplicación de θ a E (denotado por $E\theta$) se obtiene reemplazando simultaneamente cada ocurrencia de X_i en E por t_i, donde $X_i/t_i \in \theta$

La expresión resultante $E\theta$ se denomina una **instancia** de E

Ejemplo

- Sea $\theta = \{X/f(Y),\ Y/h(a)\}$ y $E \equiv p(X, Y, c)$
  Entonces, $E\theta \equiv p(f(Y), h(a), c)$

- Sea $\theta = \{X/Y,\ Y/Z\}$ entonces $E\theta =$

Sustituciones(3).

Dadas $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ y $\sigma = \{Y_1/s_1, \ldots, Y_m/s_m\}$, su **composición** $\theta\sigma$ se define a partir del conjunto:

$$\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma, Y_1/s_1, \ldots, Y_m/s_m\}$$

eliminando aquellos pares $X_i/t_i\sigma$ tales que $X_i \equiv t_i\sigma$, así como aquellos pares Y_i/s_i tales que $Y_i \in \{X_1, \ldots, X_n\}$

Ejemplo: si $\theta = \{X/3,\ Y/f(X,1)\}$ y $\sigma = \{X/4\}$, entonces $\theta\sigma = \{X/3,\ Y/f(4,1)\}$

Ejercicio: si $\theta = \{X/f(Z),\ Z/W\}$ y $\sigma = \{X/a, Z/b, W/Y, R/S\}$, entonces $\theta\sigma =$

Ejercicio: si $\theta = \{X/f(Z),\ Z/a\}$ y $\sigma = \{W/Y, R/S\}$, entonces $\theta\sigma =$

Sustituciones(4).

Relación de orden: Una sustitución θ es **más general** que σ si existe algún γ tal que $\theta\gamma = \sigma$ y se escribe que $\theta \leq \sigma$

Ejemplo: $\theta = \{X/f(Y)\}$ es más general que $\sigma = \{X/f(h(Z)),\ Y/h(Z)\}$

Ejercicio: $\theta = \{X/a\}$ es más general que $\sigma = \{X/a,\ Y/b\}$

Ejercicio: $\theta = \{X/Y\}$ es más general que $\sigma =$

Calculemos los valores . . .

Sustituciones(5).

Unificador Una sustitución β es un unificador de un conjunto de expresiones $S = \{E_1, \ldots, E_n\}$ si $E_1\beta = E_2\beta = \ldots = E_n\beta$

Ejemplo: si $\beta = \{X/a,\ Y/Z\}$ es unificador para $S = \{f(X, Z, g(W)),$ $f(a, Y, g(W))\}$

Unificador Más General: Una sustitución θ es el **unificador más general** de un conjunto de expresiones S si para cualquier otro unificador σ del mismo conjunto se cumple que $\theta \leq \sigma$

Ejemplo: $E = \{p(f(Y), Z), p(X, a)\}$ $\theta = \{X/f(a), Z/a, Y/a\}$ es unificador para E, $\sigma = \{X/f(Y),\ Z/a\}$ es unificador para E y $\sigma = \{X/f(Y),\ Z/a\}$ es unificador para E, el cual es el más general.

El Unificador Más General, si existe es único y existe un algoritmo para calcularlo

Unificación(6)

Unificar dos términos A y B consiste en encontrar una sustitución para sus variables que los hace idénticos

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)

1. dos términos con diferente nombre o aridad no se pueden unificar.
2. una variable no se puede enlazar a un término que la contenga (resulta un término infinito). Esto se conoce como “occur check”

Método para calcular el Unificador Más General (UMG)

1.	$f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$		$\rightarrow$	$s_1 = t_1, \ldots s_n = t_n$
2.	$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_n),$	$n, m \geq 0, f \neq g$	$\rightarrow$	fallo
3.	$X = X$		$\rightarrow$	borrar la ecuación
4.	$t = X, t \notin \mathcal{V}ar$		$\rightarrow$	$X = t$
5.	$X = t, X \neq t$	si X ocurre en t	$\rightarrow$	fallo
		si no	$\rightarrow$	aplicar $\{X/t\}$ al resto.

Ejemplo

$$
\begin{array}{lll}
h(X, X) = h(Y, f(Y)) & \Rightarrow & \{X = Y, X = f(Y)\} \\
& \Rightarrow & \{X = Y, Y = f(Y)\} \\
& \Rightarrow & \text{fallo}
\end{array}
$$

Método para calcular el Unificador Más General (UMG)

Ejemplo $A \equiv p(X, X)$, $B \equiv p(f(A), f(B))$

$$
\begin{array}{lcl}
p(X, X) = p(f(A), f(B)) & \Rightarrow & \{X = f(A), X = f(B)\} \\
& \Rightarrow & \{X = f(A), f(A) = f(B)\} \\
& \Rightarrow & \{X = f(A), A = B\} \\
& \Rightarrow & \{X = f(B), A = B\}
\end{array}
$$

Entonces $\sigma = \{X/f(B), A/B\}$

$$
\begin{array}{lcl}
p(X, f(Y)) = p(Z, X) & \Rightarrow & \{X = Z, f(Y) = X\} \\
& \Rightarrow & \{f(Y) = Z, f(Y) = X\}
\end{array}
$$

Entonces $\theta = \{X/f(Y), Z/f(Y)\}$

Evaluación de unificación

Ejercicio Realice los ejercicios del 1 al 17 del taller que está en EAFIT interactiva “ejercicios_pl”. Haremos plenaria y examen el próximo miércoles.

Teorema del Unificador Más General (UMG)

Dadas dos expresiones E_1 y E_2 unificables, la secuencia de transformaciones a partir de la sustitución vacía

$$\langle E_1 \cup E_2, \phi \rangle \Rightarrow \langle G_1, \theta_1 \rangle \Rightarrow \langle G_2, \theta_2 \rangle \Rightarrow \ldots \Rightarrow \langle \phi, \theta \rangle$$

genera como resultado final el unificador más general θ de las expresiones E_1 y E_2. Dicho unificador es único salvo renombramientos.

Principio de resolución en la lógica de predicados

Ejemplo.

- Simplifique (factor de un cláusula). Encuentre el factor de la cláusula $C_1 \equiv p(X) \vee p(f(Y)) \vee r(g(Y))$.

$$p \vee p \leftrightarrow p$$

- Calcule el unificador de las siguientes cláusulas y su resolvente $C_1 \equiv r(X) \vee \neg p(X, f(Y), Z)$ y $C_2 \equiv s(x) \vee p(a, Y, Z)$

- Calcule el unificador de las siguientes cláusulas y su resolvente $C_1 \equiv p(X) \vee p(f(Y)) \vee r(g(Y))$ y $C_2 \equiv \neg p(f(g(a))) \vee q(b)$

Principio de resolución en la lógica de predicados

Ejemplo.

- Pruebe que $\{\neg p(f(X)) \vee s(Y), \neg s(X) \vee r(X,b), p(X) \vee s(X)\} \models (\exists X)(\exists Y) r(X,Y)$. Calcule los correspondientes unificadores más generales.

- Pruebe que $\{\neg s(X,Y) \vee \neg p(Y) \vee r(f(X)), \neg r(Z), s(a,b), p(b)\}$ es insatisfacible.

Resolución SLD para HCL

Sea P un conjunto de cláusulas de Horn definidas (no hay negativas). Sea $S_i : \boxed{B} \leftarrow C_1, \ldots, C_n$ una (variante de una) cláusula en P y sea $G :\leftarrow B_1, \ldots, \boxed{B_i}, \ldots, B_n$ un objetivo entonces G' **se deriva de** G **y** S **usando la regla de computación** R si se cumplen las siguientes condiciones:

1. B_i es el átomo de G seleccionado por la Regla de computación R.

2. θ es el UMG de B_i y B.

3. $G' :\leftarrow (B_1, \ldots, B_{i-1}, \boxed{C_1, \ldots, C_n}, B_{i+1}, \ldots, B_n)\theta$

a G' se le denomina resolvente SLD de G y S, mediante θ, R, S_i

escribimos $G \overset{\theta,R,S_i}{\rightsquigarrow} G'$

Derivación SLD

Sea $\mathcal{P}$ un programa Definido y G un objetivo definido. Una derivación SLD para $P \cup \{G\}$ consiste en:

1. Una secuencia de objetivos $G_0, G_1, \ldots$ donde $G_0 = G$
2. Una secuencia de variantes de sentencias de P: $S_1, S_2, \ldots$
3. Una secuencia de unificadores más generales $\theta_1, \theta_2, \ldots$

$$G_0 \overset{\theta_1, S_1}{\leadsto} G_1 \overset{\theta_2, S_2}{\leadsto} G_2 \ldots G_i \overset{\theta_{i+1}, S_{i+1}}{\leadsto} G_{i+1} \ldots$$

Tal que G_{i+1} se deriva de G_i y S_{i+1} usando θ_{i+1} (G_{i+1} es el resolvente de G_i y S_{i+1})

Resolución HSLD

Sea $\mathcal{P}$ un programa lógico y Q un objetivo

1. Inicializar el "resolvente" R con el valor de $\{Q\}$
2. Elegir un literal A de R.
3. Elegir una cláusula (renombrada) $A' \leftarrow B_1, \ldots, B_n$ de $\mathcal{P}$.
4. Unificar A y $A' \Rightarrow \theta$ (si no hay ninguna, terminar con fallo).
5. eliminar A de R y añadir $B_1, \ldots, B_n$ a R en el mismo lugar de A.
6. aplicar θ a R.
7. devolver R.

Resolución HSLD

- Los pasos 2. y 3. son indeterministas.
- **Regla de computación:** decide qué literal (paso 2.)
- **Regla de búsqueda:** decide que cláusula (paso 3)
- La elección de una cláusula distinta (paso 3.) puede llevar a una solución distinta.
- En general, hay que intentar varias alternativas antes de encontrar la solución (o las soluciones).
- Los pasos 2 hasta el 6 se repiten hasta que R sea vacío.
- **Tipos derivaciones SLD:** infinitas, fracasa (G_n no es unificable) o éxito ($G_n = \square$)
- Una **refutación SLD** es una derivación que tiene éxito.

Respuesta Computada

Sea P un programa definido y G un objetivo definido.

Una **respuesta** θ para $P \cup \{G\}$ es una sustitución de las variables de G.

Una **respuesta correcta** θ para $P \cup \{G\}$ es tal que $P \models \forall G\theta$. ($\forall$ indica la cuantificación universal de las variables libres de $G\theta$)

Una **respuesta computada** θ para $P \cup \{G\}$ es la sustitución restringida a las variables de G obtenida por la composición de las $\theta_1, \theta_2, \ldots, \theta_n$ resultantes de una refutación de $P \cup \{G\}$.

Si $G_0 \overset{\theta_1, S_1}{\rightsquigarrow} G_1 \overset{\theta_2, S_2}{\rightsquigarrow} G_2 \ldots G_{n-1} \overset{\theta_n, S_n}{\rightsquigarrow} \square$ entonces $\theta = \theta_1\theta_2 \ldots \theta_n$ es una respuesta computada

Propiedades del procedimiento SLD.

Corrección. Toda respuesta computada para $P \cup \{G\}$ es una respuesta correcta para $P \cup \{G\}$

Completitud para toda respuesta correcta θ para $P \cup \{G\}$ existe una respuesta computada σ para $P \cup \{G\}$ que es más general ($\theta = \sigma\beta$, para algún β)

Independencia de la regla de computación.

Dado un programa definido P y un objetivo definido G tal que existe una respuesta computada para $P \cup \{G\}$ usando una regla de computación C, entonces existirá también una respuesta computada para $P \cup \{G\}$ usando cualquier otra regla.

Ejemplo

$\mathcal{C}_1$: `mascota(X) :- animal(X), ladra(X).`
$\mathcal{C}_2$: `mascota(X) :- animal(X), vuela(X).`

$\mathcal{C}_3$: `animal(ricky).`
$\mathcal{C}_4$: `animal(piolin).`
$\mathcal{C}_5$: `animal(hobbes).`
$\mathcal{C}_6$: `ladra(ricky).`
$\mathcal{C}_7$: `vuela(piolin).`
$\mathcal{C}_8$: `ruge(hobbes).`

`Resolver` $\leftarrow$ `mascota(X).:`

Q	**R**	**Cláusula**	θ
mascota(X)	mascota(X)	C2	X/X1
mascota(X1)	animal(X1), vuela(X1)	C7	X1/piolin
mascota(piolin)	animal(piolin)	C4	
mascota(piolin)	---	---	---

`Resolver` $\leftarrow$ `mascota(X). (estrategia diferente):`

mascota(X)	mascota(X)	C1	X/X1
mascota(X1)	animal(X1), ladra(X1)	C5	X1/hobbes
mascota(hobbes)	ladra(hobbes)	???	fallo

Árbol SLD

El conjunto de todas las derivaciones posibles para $P \cup \{G\}$ se puede representar como un árbol, donde:

- Cada nodo es un objetivo.
- El nodo raíz es G.
- Si $G_i = B_1, \ldots, B_k, \ldots, B_n$, $n \geq 1$ es un nodo del árbol y B_k es el átomo seleccionado por alguna regla de computación entonces por cada sentencia $S = A \leftarrow A_1, \ldots, A_s$ tal que $\theta = mgu(B_k, A)$ el nodo tiene un hijo $(B_1, \ldots, B_{k-1}, A_1, \ldots, A_s, B_{k+1}, \ldots, B_n)\theta$
- Los nodos $\square$ son hojas.

Ejemplo

Construir su árbol de ejecución.

```
p(X,Z) :-   q(X,Y), p(Y,Z).
p(X,X).
q(a,b).
```

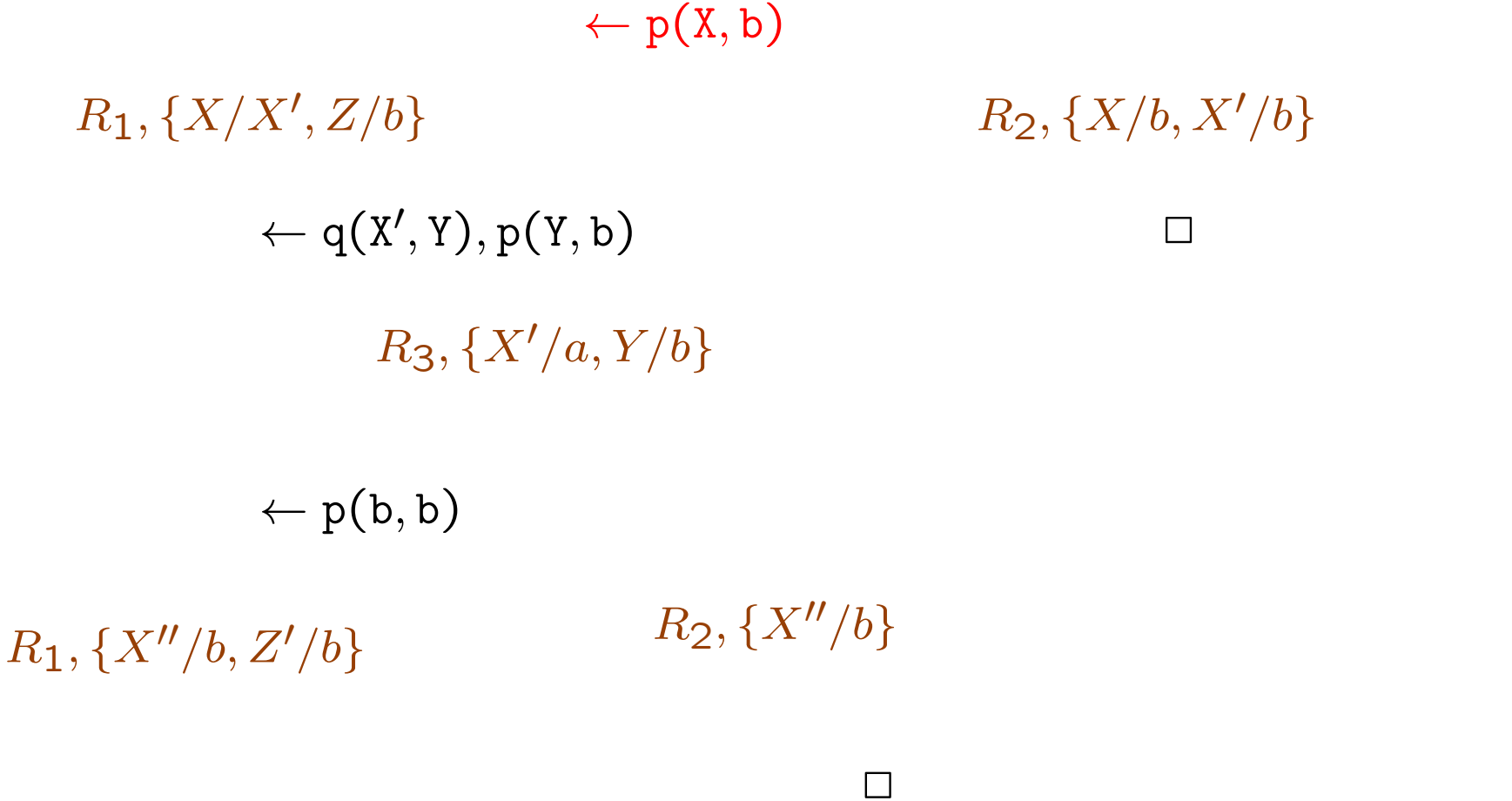
Ejecutar el objetivo `p(X,b)`.

Que pasa si en lugar de elegir el primero de la izquierda elegimos el primero de la derecha

Ejecutar este programa en Prolog.

Árbol de ejecución, $\texttt{p(X, b)}$

EL PROLOG: Primero a la izquierda y primero en profundidad.



Ejercicios

Del taller realice los ejercicios del 18 al 26.

En la WIKI de EAFIT interactiva elija un reto de programación y notifique su elección en la misma.

Usando el Prolog ...

Programa Prolog

Programación Lógica

*	Programa:	Teoría de cláusulas de Horn
*	Semántica declarativa:	Teoría de Modelos
*	Semántica Operacional:	SLD-resolución

PROLOG	=	P. Lógica	+	Control fn. de selección Est. de búsqueda	+	Facilidades extralogicas entrada/salida Corte . . .

Programa Prolog: Teoría clausal, cuyos axiomas son reglas y hechos.

Ejecución Programación Lógica: Demostrar que una conclusión se deduce de las cláusulas del programa.

Recursividad

```
progenitor(X,Y)  :-  padre(X,Y).
progenitor(X,Y)  :-  madre(X,Y).
ancestro(X,Y)    :-  progenitor(X,Y).
ancestro(X,Y)    :-  progenitor(X,Z),progenitor(Z,Y).
ancestro(X,Y)    :-  progenitor(X,Z),progenitor(Z,W),
                     progenitor(W,Y).
ancestro(X,Y)    :-  progenitor(X,Z),progenitor(Z,W),
                     progenitor(W,K),progenitor(K,Y).
...
```

Podemos dar una definición recursiva:

```
progenitor(X,Y)  :-  padre(X,Y).
progenitor(X,Y)  :-  madre(X,Y).
ancestro(X,Y)    :-  progenitor(X,Y).
ancestro(X,Y)    :-  progenitor(X,Z),ancestor(Z,Y).
```

Recursividad: Aritmética

Los términos se pueden clasificar en **tipos**, cada uno de ellos conteniendo un conjunto (posiblemente infinito) de términos.

Un programa que defina un tipo se denomina una **definición de tipo.**

Tipos recursivos: se definen mediante programas lógicos recursivos.

El tipo recursivo simple son los números naturales: $\boxed{0, s(0), s(s(0)), \ldots}$ definido por:

```
numero_natural(0).
numero_natural(s(X))  :-  numero_natural(X).
```

y se pueden ordenar por $\boxed{\text{“}\leq\text{”}}$:

```
≤(0,X)        :-  numero_natural(X).
≤(s(X),s(Y))  :-  ≤(X,Y).
```

Recursividad: Aritmética

Operación suma:

```
suma(0,X,X))          :-  numero_natural(X).
suma(s(X),Y,s(Z))     :-  suma(X,Y,Z).
```

Algunos objetivos:

`suma(s(s(0)),s(0),Z)`,
`suma(s(s(0)),Y,s(s(s(0))))`, ...
`suma(X,Y,s(s(s(0))))`, ...

Otra posible definición para la suma:

```
suma(0,X,X))          :-  numero_natural(X).
suma(X,s(Y),s(Z))     :-  suma(X,Y,Z).
```

Cuál es la mejor?. Eficiencia.

Cómo son las otras operaciones?. Pensando en la definición formal

Recursividad: Listas

Es una estructura binaria: el primer argumento es un *elemento*, el segundo argumento es el *resto* de la lista. Necesitamos:

un símbolo constante: la lista vacía, denotada por [ ]

un functor de aridad 2: tradicionalmente el punto "." Notación alternativa: $.(\mathtt{X}, \mathtt{Y}) \equiv [\mathtt{X}|\mathtt{Y}]$ ($\mathtt{X}$ es la *cabeza*, $\mathtt{Y}$ es la *cola*)

Ejemplo

Objeto formal	Notación cabeza/cola	Notación por enumeración
.(a, [])	[a|[]]	[a]
.(a, .(b, []))	[a|[b|[]]]	[a, b]
.(a, .(b, .(c, [])))	[a|[b|[c|[]]]]	[a, b, c]
.(a, X)	[a|X]	[a|X]
.(a, .(b, X))	[a|[b|X]]	[a, b|X]

Recursividad: Listas

Observe que:

[a, b] y [a|X] *unifican usando* X/[b]

[a] y [a|X] *unifican usando* X/[]

[a] y [a, b|X] *no unifican*

[] y [X] *no unifican*

Definición del tipo lista:

```
lista([]).
lista([X|Y])  :-  lista(Y).
```

Recursividad: Operaciones con Listas

El elemento `X` es miembro de la lista `Y`:

```
miembro(X,[X|_]) :- lista([X|_]).
miembro(X,[_|R]) :- miembro(X,R).
```

Concatenación de listas:

```
append([],Ys,Ys)          :- lista(Ys).
append([X|Xs],Ys,[X|Zs])  :- append(Xs,Ys,Zs).
```

- Invertir el orden de los elementos de una lista.
- .....Muchos problemas interesantes.
- Ir a la Wiki y escoger dos problemas.

Introducción al análisis de programas

Idea. Hacer un parser para analizar el programa. Sugerencias: partir el problema en varios problemas.

- Definir la estructura del programa. Es necesario disponer de reglas básicas para reconocer los caracteres especiales.
- Crear el programa en un archivo. El usuario escribe un nombre del programa y las sentencias del programa por teclado. Utilice:
  - ✓ predicados dinámicos: dynamic programa/# donde # es la aridad
  - ✓ Leer sentencia de la forma $L : -L_1, ..., L_n.$
  - ✓ Graba la sentencia.
  - ✓ Muestra el archivo

Introducción al análisis de programas

- Comandos que se puedan utilizar para guardar en un predicado dinámico el archivo
  - ✓ read
  - ✓ repeat
  - ✓ retractall
  - ✓ assert, asserta, assertz
  - ✓ see, seen
  - ✓ tell, told
  - ✓ write

Introducción al análisis de programas

- Parser. Transforma la regla $L : -L_1, ..., L_n.$ a la forma:

$$regla(equa(L), [equa(L1), ..., equa(Ln)]).$$

- Predicados que podría usar.

  - $\surd$ functor
  - $\surd$ arg
  - $\surd$ append
  - $\surd$ end_of_file
  - $\surd$ ..
  - $\surd$!

- Chequear propiedad

Semántica Declarativa: Lo previo

El Universo de Herbrand $\mathcal{U}_\mathcal{L}$ para un lenguaje de primer orden $\mathcal{L}$ es el conjunto de todos los términos sin variables construidos a partir de los funtores y las constantes. $\mathcal{U}_\mathcal{L}$ contiene al menos un símbolo de constante.

La Base de Herbrand $\mathcal{B}_\mathcal{L}$ es el conjunto de todos los átomos base (sin variables) que pueden formar los símbolos de predicado de $\mathcal{L}$ y todos los términos de $\mathcal{U}_\mathcal{L}$

Una Interpretación de Herbrand $\mathcal{I}$ es un triplete $(\mathcal{U}_\mathcal{L}, K, E)$ donde K asigna a los símbolos de constantes a ellos mismos y a toda función f aplicada a $(t_1, \ldots, t_n) \in \mathcal{U}_\mathcal{L}^n$ le asigna $f(t_1, \ldots, t_n) \in \mathcal{U}_\mathcal{L}$ y E asigna a cada símbolo de predicado un conjunto n-tuplas de la forma $\mathcal{U}_\mathcal{L}^n$.

La diferencia entre interpretaciones está en las asignaciones creadas por E. Por ello, definimos una interpretación de Herbrand $\mathcal{I} \subseteq \mathcal{B}_\mathcal{L}$. Como consecuencia podemos hablar modelo de Herbrand.

Ejemplo

Dado el siguiente programa $\mathcal{R}$, calcular $\mathcal{U}_\mathcal{R}$, $\mathcal{B}_\mathcal{R}$

```
p(X,Z) :-   q(X,Y), p(Y,Z).
p(X,X).
q(a,b).
```

Determine $\mathcal{U}_\mathcal{R}$, $\mathcal{B}_\mathcal{R}$, presente varias interpretaciones y modelos.

Ejemplo

$\mathcal{U}_{\mathcal{R}} = \{\mathtt{a}, \mathtt{b}\}$

$\mathcal{B}_{\mathcal{R}} = \{\mathtt{p(a,a)}, \mathtt{p(b,b)}, \mathtt{p(a,b)}, \mathtt{p(b,a)}, \mathtt{q(a,a)}, \mathtt{q(b,b)}, \mathtt{q(a,b)}, \mathtt{q(b,a)}\}$

$\mathcal{I}_1 = \{\mathtt{p(b,a)}, \mathtt{q(a,a)}, \mathtt{q(b,b)}\}$

$\mathcal{I}_2 = \{\mathtt{p(b,b)}, \mathtt{q(a,b)}\}$

$\mathcal{I}_3 = \{\mathtt{p(b,b)}, \mathtt{q(a,b)}, \mathtt{p(a,a)}, \mathtt{p(a,b)}\}$

$\mathcal{I}_4 = \mathcal{B}_{\mathcal{R}}$

Se busca modelar la información expresada por el programa y nada más.

Modelo Mínimo de Herbrand

Sea P un programa definido y G un objetivo definido.

Si existe un modelo $\mathcal{I}$ para $P \cup \{G\}$ entonces la interpretación definida por $\{A \in \mathcal{B}_\mathcal{P} / \models_\mathcal{I} A\}$ es un modelo de Herbrand de $P \cup \{G\}$

Sea $\{M_i\}_{i \in \mathcal{I}}$ una familia no vacía de modelos de Herbrand de un programa definido P. Entonces, $\bigcap_{i \in \mathcal{I}} M_i$ es un modelo de Herbrand de P. Se denomina el Modelo Mínimo de Herbrand M_P.

$$M_P = \{A \in \mathcal{B}_\mathcal{P} / P \models A\}$$

Del ejemplo anterior podemos ver que $\mathcal{I}_3 = \{\mathtt{p(b,b)}, \mathtt{q(a,b)}, \mathtt{p(a,a)}, \mathtt{p(a,b)}\}$ es el modelo mínimo. Este no contiene información redundante.

No siempre la intersección de modelos es un modelo.

Ejercicios 27 al 37

Semántica de punto fijo

¿Cómo construir el modelo mínimo?

Operador de Consecuencias Inmediatas. Sea $ground(P)$ el conjunto de todas las instancias básicas de las cláusulas de P. Entonces $T_P : 2^{\mathcal{B}_P} \rightarrow 2^{\mathcal{B}_P}$

$$T_P(\mathcal{I}) = \{A_0 / A_0 \leftarrow A_1, \ldots, A_n \in ground(P) \wedge A_1, \ldots, A_n \subseteq \mathcal{I}\}$$

Propiedades.

- El operador T_P es monótono con respecto a la inclusión de conjuntos.

  Si $\mathcal{I} \subseteq \mathcal{J}$ entonces $T_P(\mathcal{I}) \subseteq T_P(\mathcal{J})$

- El operador T_P es monótono con respecto a la inclusión de conjuntos.

  Si $\mathcal{I}_0 \subseteq \mathcal{I}_2 \ldots$ entonces $T_P(\bigcup_{i=0}^{\infty} \mathcal{I}_i) \subseteq \bigcup_{i=0}^{\infty} T_P(\mathcal{I}_i)$

Semántica de punto fijo

Potencias ordinales de T_P

$T_P \uparrow 0 = \emptyset$
$T_P \uparrow n = T_P(T_P \uparrow (n-1))$
$T_P \uparrow \omega = \bigcup_{i=0}^{\infty} T_P \uparrow i$

El menor punto fijo de T_P se alcanza en ω iteraciones empezando en la interpretación vacia

Propiedad. Sea P un programa definido y M_P el modelo mínimo de Herbrand, entonces

- $T_P(M_P) = M_P$
- $M_P = lfp(T_P) = T_P \uparrow \omega$

Ejemplo

Dado los siguientes programas determine su Modelo Mínimo de Herbrand mediante el cálculo de las potencias del operador de consecuencias inmediatas

1.
```
p(X,Z) :-   q(X,Y), p(Y,Z).
p(X,X).
q(a,b).
```

2.
```
even(0).
even(s(s(X))) :- even(X).
```

Ejemplo

```
p(X,Z) :-   q(X,Y), p(Y,Z).
p(X,X).
q(a,b).
```

$T_P \uparrow 0 = \emptyset$
$T_P \uparrow 1 = \{q(a,b), p(a,a), p(b,b)\}$
$T_P \uparrow 2 = \{q(a,b), p(a,a), p(b,b), p(a,b)\}$
$T_P \uparrow 3 = \{q(a,b), p(a,a), p(b,b), p(a,b)\}$

Ejemplo

```
even(0).
even(s(s(X))) :- even(X).
```

$$\mathcal{U}_P = \{0, s(0), s(s(0)), s^3(0), \ldots\}$$
$$\mathcal{B}_P = \{even(0), even(s(0)), even(s(s(0))), even(s^3(0)), \ldots\}$$

$$T_P \uparrow 0 = \emptyset$$
$$T_P \uparrow 1 = \{even(0)\}$$
$$T_P \uparrow 2 = \{even(0), even(s^2(0))\}$$
$$T_P \uparrow 3 = \{even(0), even(s^2(0)), even(s^4(0)), \}$$
$$\vdots$$
$$T_P \uparrow \omega = \{even(0), even(s^2(0)), \ldots, even(s^{2*n}(0)), \ldots\}$$

Ejercicios 38 al 42

Semántica Operacional

Semántica de respuestas computadas (conjunto de éxitos básicos). Sea P un programa

$$SS(P) = \{A \in \mathcal{B}_P / P \cup \{A\} \leadsto^* \square\}$$

Ejemplo

```
p(X,Z)  :-   q(X,Y), p(Y,Z).
p(X,X).
q(a,b).
```

$$SS(P) = \{\mathtt{q(a,b)}, \mathtt{p(a,a)}, \mathtt{p(b,b)}, \mathtt{p(a,b)}\}$$

Equivalencia entre Semánticas

Sea P un programa lógico definido. Entonces se cumple que:

$$
\begin{aligned}
M_P &= \bigcap_{i \in \mathcal{I}} M_i \\
&= lfp(T_p) \\
&= \{A \in \mathcal{B}_P / P \models A\} \\
&= \{A \in \mathcal{B}_P / P \cup \{A\} \rightsquigarrow^* \square\} \\
&= SS(P)
\end{aligned}
$$

Ejercicios

Terminar el taller

Ejemplos: Programa Prolog . . .

Ejemplo

```
hermano(ana,pablo).
hermano(mimi,juana).
progenitor(pablo,jaimito).
progenitor(juana,caty).
hembra(caty).
varon(jaimito).
tia(X,Y) :- hermana(X,Z), progenitor(Z,Y).
```

Objetivos:

?- hermana(ana,pablo).
?- tia(ana,jaimito).
?- tia(X,caty).
?- tia(U,V).
?- hembra(maria).

En la ejecución . . .

?- help. Te muestra una pantalla de ayuda. Vemos cada uno de los elementos de dicha ventana:

search. Presenta una tabla y eliges.

view. Tabla de contenidos.

Go. Para ver temas

help. Otras ayudas disponibles.

Cuando se ejecuta un objetivo se presentan las siguientes opciones:

```
Actions:

; (n, r):      redo    t:                    trace & redo b:
```

En la ejecución . . . (2)

Con las teclas Control + c

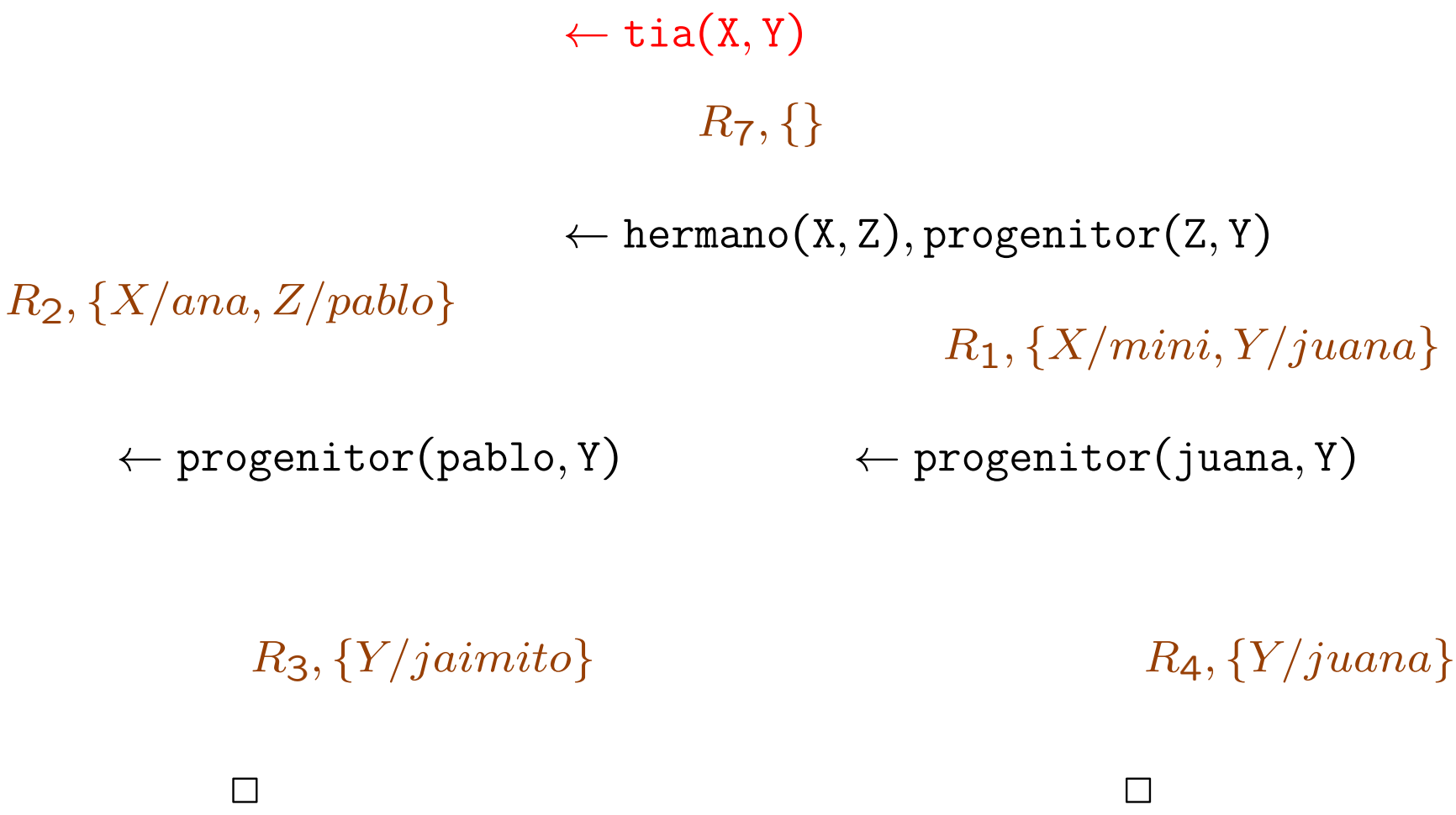
```
Action (h for help) ? Options: a:                    abort
```

Haciendo un trace de la ejecución

Control + C elegir o digitar **trace** con el objetivo `tia(X, Y)`.

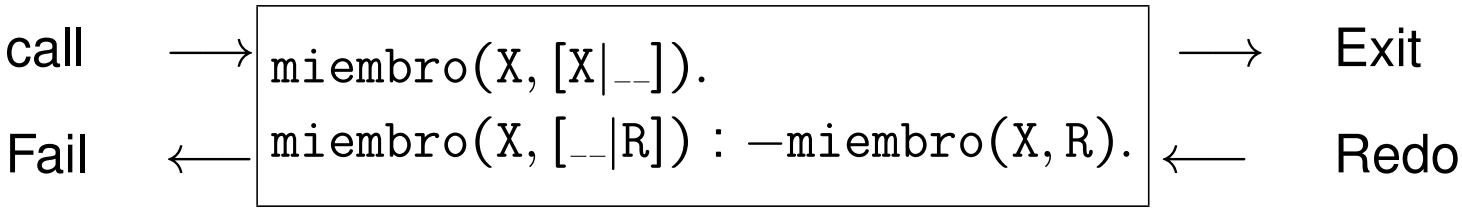
```
   Call: (6) tia(_G157, _G158) ? c
   Call: (7) hermano(_G157, _G227) ? c
   Exit: (7) hermano(ana, pablo) ? c
   Call: (7) progenitor(pablo, _G158) ? c
   Exit: (7) progenitor(pablo, jaimito) ? c
   Exit: (6) tia(ana, jaimito) ? c
X = ana Y = jaimito ;
   Fail: (7) progenitor(pablo, _G158) ? c
   Redo: (7) hermano(_G157, _G227) ? c
   Exit: (7) hermano(mimi, juana) ? c
   Call: (7) progenitor(juana, _G158) ? c
   Exit: (7) progenitor(juana, caty) ? c
   Exit: (6) tia(mimi, caty) ? c
X = mimi Y = caty ;
   Fail: (7) progenitor(juana, _G158) ? c
   Fail: (7) hermano(_G157, _G227) ? c
   Fail: (6) tia(_G157, _G158) ? c
No
```

Haciendo un trace de la ejecución, interpretación



Traza de Ejecución – Cajas de Byrd

- Los procedimientos (predicados) se pueden ver como cajas negras de la forma usual



- Los principales eventos en la ejecución de Prolog son:
  - `Call`: comienza la ejecución de un objetivo
  - `Exit`: ha producido una solución para el objetivo
  - `Redo`: intenta buscar soluciones alternativas
  - `Fail`: falla en encontrar ninguna solución (más)

Ejecutar `miembro(X, [a, b]).`

Traza de Ejecución – Cajas de Byrd

```
[trace]  ? - miembro(X,[a,b]).
         Call : (6)     miembro(_G289,[a,b])?  creep
         Exit : (6)     miembro(a,[a,b])?  creep
X = a?;
         Redo : (6)     miembro(_G289,[a,b])?  creep
         Call : (7)     miembro(_G289,[b])?  creep
         Exit : (7)     miembro(b,[b])?  creep
         Exit : (6)     miembro(b,[a,b])?  creep
X = b?;
         Redo : (7)     miembro(_G289,[b])?  creep
         Call : (8)     miembro(_G289,[])?  creep
         Fail : (8)     miembro(_G289,[])?  creep
         Fail : (7)     miembro(_G289,[b])?  creep
         Fail : (6)     miembro(_G289,[a,b])?  creep
no
```

Algunas opciones durante la traza

h	“help”: ayuda
c	“creep”: avanza hasta el siguiente call/exit/redo/fail
INTRO	(lo mismo que antes)
s	“skip”: avanzar hasta la terminación del objetivo actual
l	“leap”: avanzar hasta el siguiente “syspoint”
f	“fail”: provocar el fallo del objetivo actual
	(fuerza a reconsiderar la primera alternativa pendiente)
a	“abort”: abortar la ejecución en curso
r	“redo”: rehacer la ejecución del objetivo actual
b	“break”: suspende la traza e invoca al intérprete
	(se puede continuar con `D`)

Existen muchas más opciones en los compiladores actuales

Además, existe depuradores gráficos en algunos sistemas

Puntos espía (spypoints)

- `?- spy(hermano).`

  Pone un spypoint en el predicado `hermano` de aridad 2 (siempre traza los eventos de este predicado)

- `?- nospy(hermano).`

  Elimina el spypoint de `hermano/2`

- `?- nospyall.`

  Elimina todos los spypoints

La negación

Los programas definidos expresan conocimiento positivo.

En algunos casos es necesario introducir información negativa en el cuerpo de una cláusula y poder derivar información negativa.

Por ejemplo para

```
pertenece_complemento(X,Y) :- es_universal(U),
                              miembro (X,U), no(miembro(X,Y)).
```

Otro ejemplo

```
diferente(X,Y) :- miembro (Z,X), no(miembro(Z,Y)).
diferente(X,Y) :- no(miembro (Z,X)), miembro(Z,Y).
```

La negación

Si P es un programa definido y A un átomo de la Base de Herbrand, nunca podremos probar que $\neg A$ es consecuencia lógica de P. La razón es que $P \cup \{A\}$ siempre es satisfactible.

Ejemplo. Sea P el siguiente programa:

```
estudiante(ana).
estudiante(maria).
estudiante(isabel).
profesor(pacho).
```

Pero $\mathtt{G} = \neg(\mathtt{estudiante(pacho)})$ es objetivo, tal que $P \not\models G$ y tampoco $P \not\models \neg G$. Pero $P \cup \{\mathtt{estudiante(pacho)}\}$ es satisfacible.

La negación

Teorema
Sea P un programa definido y G un átomo de la base de Herbrand asociada al programa P. Los siguientes enunciados son equivalentes:

1. El objetivo $\neg G$ no es una consecuencia lógica del programa P. Un átomo negativo no puede pertenecer a la base de Herbrand y por tanto no puede ser consecuencia lógica de un programa.

2. El conjunto de cláusulas $P \cup \{G\}$ es satisfacible. $G \in \mathcal{B}$ por lo tanto la misma base puede ser modelo para P

3. No existe una refutación SLD para $P \cup \{G\}$.

Incorporación de la negación

Hipótesis del Mundo Cerrado (HMC): Supone que toda los hechos relevantes para el problema están explícitos. "Si un átomo base A no es consecuencia lógica del programa, entonces inferimos $\neg A$". Inferimos con base en la carencia de información positiva.

Si $P \nvDash A$ o $P \nvdash A$ entonces $\neg A$

Pero debemos probar que A no es consecuencia lógica del programa. Pero no hay algoritmo que, dado un átomo cualquiera, responda en un **tiempo finito** si A es o no, consecuencia lógica de P.

Podemos inferir como $\neg$`estudiante(pacho)` sobre la base de que `estudiante(pacho)` no es consecuencia lógica de P. Es una regla no monotónica porque al agregar axiomas disminuye la cantidad de teoremas que podemos demostrar. Si agregamos `estudiante(pacho)` ya no podemos demostrar $\neg$`estudiante(pacho)`

Incorporación de la negación

Negación como fallo: Limitamos nuestro alcance a aquellos átomos que fallan finitamente. Esta restricción se limita a un subconjunto del generado por la hipótesis del mundo cerrado.

Consiste en: "inferir $\neg A$ si y sólo si el intento de inferir A falla finitamente´´. Es decir, A pertenece al conjunto de fallos finitos SLD de P.

Sea P un programa definido P, el conjunto de fallo finito SLD de P, F_P es el conjunto de todos los átomos A de la base de Herbrand para los que existe un árbol SLD fallado finitamente para $P \cup \{A\}$, es decir un árbol finito y sin ramas de éxito.

Ejemplo

```
p(x) :-  q(x, y), p(y).
q(a, b).
q(b, a).
q(c, c).
p(b).
```

Determine la base de Herbrand, el modelo mínimo de Herbrand, el conjunto de fallo finito. Compare la solución con el conjunto de elementos resultantes de la hipotesis del mundo cerrado

Ejemplo

$$\mathcal{B}_P = \{\mathtt{q(a,a)}, \mathtt{q(b,b)}, \mathtt{q(c,c)}, \mathtt{q(a,b)}, \mathtt{q(b,a)}, \mathtt{q(a,c)}, \mathtt{q(c,a)}, \mathtt{q(c,b)}, \mathtt{q(b,c)}, \mathtt{p(a)}, \mathtt{p(b)}, \mathtt{p(c)}\}$$

$$\mathcal{M}_P = \{\mathtt{q(c,c)}, \mathtt{q(a,b)}, \mathtt{q(b,a)}, \mathtt{p(a)}, \mathtt{p(b)}\}$$

$$NF = \{\mathtt{q(a,a)}, \mathtt{q(b,b)}, \mathtt{q(a,c)}, \mathtt{q(c,a)}, \mathtt{q(c,b)}, \mathtt{q(b,c)}\}$$

$$HMC - NF = \{\mathtt{p(c)}\}$$

Pero

Tenemos objetivos con literales negativos en un programa que serían tratados por la regla de la Negación como fallo.

Pero todavía no podemos afirmar que un literal $\neg A$ es consecuencia lógica de un programa P.

El problema es que $P \cup A$ es satisfactible. Es decir, de un programa definido no se puede inferir información negativa.

La **razón de este hecho** es que los predicados están definidos en un sólo sentido. Para corregirlo consideramos que la definición de un predicado se *define de forma completa*. Toda la información que resulta de este predicado está en su definición.

Pero

Solución:

1. Considerar que el programa es una abreviación de un programa más amplio que permite derivar información negativa. Compleción del programa, $comp(P)$. Es decir, el conjunto de sentencias que tienen el mismo símbolo de predicado en la cabeza, se debe interpretar como la definición completa de dicho predicado en P.

2. Redefinir la noción de consecuencia lógica. Poder considerar algunos modelos.

Ejemplo: Un grupo de personas son estudiantes y profesores.

Hay que definir un sentencia que exprese quienes son estudiantes.

$\texttt{estudiante(X)} \rightarrow (\texttt{X} = \texttt{ana} \vee \texttt{X} = \texttt{maria} \ldots)$.

Pero

$\texttt{estudiante(X)} \rightarrow (\texttt{X} = \texttt{ana} \vee \texttt{X} = \texttt{maria} \vee \texttt{X} = \texttt{isabel})$.

Un estudiante es ana, maria, ...

Ya podemos deducir $\neg\texttt{estudiante(pacho)}$ a partir del programa aumentado

$\texttt{P} \cup \{\texttt{estudiante(X)} \rightarrow (\texttt{X} = \texttt{ana} \vee \texttt{X} = \texttt{maria} \vee \texttt{X} = \texttt{isabael})\}$.

Al programa $\texttt{P}$ se le agregan nuevas reglas que no necesariamente son cláusulas de Horn. Estas reglas definen lo que es falso.

Programa normales . . .

Cláusula normal: es una cláusula normal es de la forma: $A \leftarrow L_1, \ldots, L_m$ donde A es un átomo y $L_1, \ldots, L_m$ son literales.

Tipos de Cláusulas:

Regla. Es una cláusula normal de la forma: $A \leftarrow L_1, \ldots, L_m$

Hecho. Es una cláusula normal de la forma: $A \leftarrow$.

Objetivo. Es una cláusula normal de la forma: $\leftarrow L_1, \ldots, L_m$.

Programa normal. Es un conjunto de cláusulas normales.

Compleción de un programa normal

Paso 1. Para cada símbolo de predicado p reemplazar cada Cláusula C de la forma:

$\mathtt{p(t_1, \ldots, t_m)} \leftarrow \mathtt{L_1, \ldots L_n}$ con $n \geq 0$

por la cláusula

$\mathtt{p(X_1, \ldots, X_m)} \leftarrow \exists \mathtt{Y_1, \ldots, Y_i}(\mathtt{X_1 = t_1, \ldots, X_m = t_m, L_1, \ldots, L_n})$

donde $\mathtt{Y_1, \ldots, Y_i}$ variables de C y $\mathtt{X_1, \ldots, X_m}$ son variables frescas.

Compleción de un Programa normal

Paso 1. Ejemplo.

Sea la siguiente la definición del predicado p.

$$\begin{aligned} \mathtt{p(Y)} &\leftarrow \mathtt{q(Y)}, \neg \mathtt{r(a,Y)}. \\ \mathtt{p(f(Z))} &\leftarrow \neg \mathtt{q(Z)}. \\ \mathtt{p(b)}. & \end{aligned}$$

$$\begin{aligned} \mathtt{p(X)} &\leftarrow (\exists \mathtt{Y})(\mathtt{X} = \mathtt{Y} \wedge \mathtt{q(Y)} \wedge \neg \mathtt{r(a,Y)}). \\ \mathtt{p(X)} &\leftarrow (\exists \mathtt{Z})(\mathtt{X} = \mathtt{f(Z)} \wedge \neg \mathtt{q(Z)}). \\ \mathtt{p(X)} &\leftarrow \mathtt{X} = \mathtt{b}. \end{aligned}$$

Compleción de un Programa normal

Paso 2. Para cada símbolo de predicado p reemplazar todas las Cláusulas:

$$\begin{aligned} \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{m}) &\leftarrow \mathtt{B}_1 \\ &\vdots \\ \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{m}) &\leftarrow \mathtt{B}_\mathtt{j} \end{aligned}$$

por la fórmula

$$\begin{array}{ll} \forall \mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{m}(\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{m}) \leftrightarrow \mathtt{B}_1 \vee \ldots \vee \mathtt{B}_\mathtt{j}) & \text{if} \quad j \geq 1 \\ \forall \mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{m}(\neg \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_\mathtt{m})) & \text{if} \quad j = 0 \end{array}$$

Si $j = O$, se cumple si el símbolo de predicado p no aparece en la cabeza de alguna regla

Paso 3. Definición de la igualdad (reflexiva, simétrica, transitiva, sustitutividad para predicados y funtores, etc.

Compleción de un Programa normal

Ejemplo

Paso 2. La definición completada de $\mathtt{p}$ es:

$$(\forall \mathtt{X})(\mathtt{p(X)} \leftrightarrow (\exists \mathtt{Y})(\mathtt{X} = \mathtt{Y} \wedge \mathtt{q(Y)} \wedge \neg \mathtt{r(a, Y)}) \vee (\exists \mathtt{Z})(\mathtt{X} = \mathtt{f(Z)} \wedge \neg \mathtt{q(Z)}) \vee \mathtt{X} = \mathtt{b})$$

Como r y q no están en la cabeza de alguna regla, agregamos la nueva definición. Indica que toda información que se pueda deducir de ellos es negativa. Es decir, de q y de r no hay nada cierto.

$$(\forall \mathtt{X}, \mathtt{Y})(\neg \mathtt{r(X, Y)})$$

$$(\forall \mathtt{X})(\neg \mathtt{q(X)})$$

Ejemplo

Consideremos el “mundo´´ donde solo hay dos progenitores maria y dora

```
padre(X) :- hombre(X), progenitor(X).
progenitor(maría).
progenitor(dora).
```

Paso 1.

$$\begin{array}{lcl} \texttt{padre}(X_1) & \leftarrow & \exists X(X_1 = X, \texttt{hombre}(X), \texttt{progenitor}(X)). \\ \texttt{progenitor}(X_1) & \leftarrow & X_1 = \texttt{maria}. \\ \texttt{progenitor}(X_1) & \leftarrow & X_1 = \texttt{dora}. \end{array}$$

Paso 2.

$$\begin{array}{lcl} \forall X_1(\texttt{padre}(X_1) & \leftrightarrow & \exists X(X_1 = X, \texttt{hombre}(X), \texttt{progenitor}(X))) \\ \forall X_1(\texttt{progenitor}(X_1) & \leftrightarrow & X_1 = \texttt{maria} \vee X_1 = \texttt{dora}) \\ \forall X_1(\neg(\texttt{hombre}(X_1)) & & \end{array}$$

Propiedades

$A \in \mathcal{B}_P$ entonces $P \models A \Leftrightarrow comp(P) \models A$

Si $A \in \mathcal{B}_P$ pertenece al conjunto de fallo finito de P entonces $comp(P) \models \neg A$

Programas, sentencia y objetivo normales.

SLDNF = SLD + Negación como Fallo

Respuesta, respuesta correcta, respuesta computada.

Derivación

Programas Normales

SLDNF = SLD + Negación como Fallo

Se encuentra un literal negativo $\neg A$ y lanza un nuevo árbol cuya raíz es A. Se presentan las siguientes alternativas.

- A **contiene variables.** Regla de computación no segura. La computación se detiene. Se pierde la corrección de SLDNF. Objetivo: $\neg$p y se sabe que $comp(P) \not\models \neg$p
  p :$-\neg$q(x), r(x).
  q(a).
  r(b).

- A **no contiene variables y el árbol falla.** Entonces $\neg A$ tiene éxito.

- A **no contiene variables y es éxito.** Entonces $\neg A$ falla.

- A **no contiene variables y es infinita.** No puedo decir nada.

Programas Normales

Ejemplo

```
p(X) :- q(X) ∧ ¬r(x).
p(X) :- t(X).
r(X) :- s(X).
q(a).
q(b).
s(b).
t(c).
```

objetivo: `p(X)`

Programas Normales, problemas

Limitaciones de la Regla de Computación. En una derivación SLDNF no se puede seleccionar, en ningún objetivo, un literal negativo que contenga variables (no base).

Una **regla de computación es segura** si y sólo si:

1. nunca selecciona un literal negativo no base; y

2. si en una derivación se produce un objetivo de la forma $\leftarrow L_1, \ldots L_m$, tal que para todo i cada L_i es un literal negativo no base entonces la computación se detiene. A esta situación se la denomina **tropiezo**.

Programas Normales, problemas

Ejemplo. Sea P el siguiente programa normal:

$$p :- \neg q(x), r(x).$$
$$q(a).$$
$$r(b).$$

y sea G el objetivo normal $\leftarrow \neg p$. Supongamos que se utiliza una regla de computación no-segura que selecciona el 1º de la izquierda.

Elabore el árbol.

y sin embargo $comp(P) \models \neg p$. O sea que utilizando una regla de computación no-segura se pierda la corrección del procedimiento de resolución SLDNF.

Propiedades del SLDNF

Correcto: sea P un programa normal y sea G un objetivo normal. Si θ es una respuesta computada para $P \cup G$, entonces θ es una respuesta correcta para $comp(P) \cup G$.

No completo: dado un programa normal P y un objetivo normal G existen respuestas correctas θ que no pueden ser computadas por el SLDNF.

Sea P el siguiente programa normal

$$r :- p.$$
$$r :- \neg p.$$
$$p :- p.$$

y sea G el objetivo normal r. Se ve $comp(P) \models r$, pero $comp(P) \not\vdash r$. Todo árbol SLDNF para $P \cup \{r\}$ es infinito.

Nuevos tipos de programas($\Diamond$)

La incompletitud del SLDNF está provocada por:

Tropiezo: se detiene la computación.

Negación + Recursión: la existencia de ramas infinitas provoca el no poder deducir consecuencias lógicas.

Vamos a definir subclases de programas normales que por su forma eviten estos problemas. En concreto presentaremos los programas permitidos, jerárquicos y estratificados.

Semántica y nuevos tipos de programas($\Diamond$)

Programas Permitidos, Jerárquicos y Estratificados

En programas normales surgen (debido a la negación) tres problemas nuevos:

1. **Inconsistencia de comp(P):** debido a la presencia de recursión en términos de negación es posible que la compleción de un programa sea inconsistente (es decir, no tenga modelo).

2. **TP es no-monotónico:** debido también a la negación el operador consecuencia inmediata ya no es monotónico, con lo cual no se puede aplicar la Teoría del Punto Fijo para localizar este modelo mínimo.

3. **Existencia de varios modelos mínimos:** al introducir la negación en el programa perdemos la unicidad de modelo mínimo.

Programas permitidos (no tropiezo)($\Diamond$)

Sentencia admisible. Una sentencia de programa normal $A \leftarrow L_1 \wedge \ldots \wedge L_n$ es admisible si cada variable de la sentencia o aparece en la cabeza A o en algún literal positivo del cuerpo.

Sentencia permitida. Una sentencia de programa normal $A \leftarrow L_1 \wedge \ldots \wedge L_n$ es permitida si cada variable de la sentencia aparece en un literal positivo del cuerpo.

Objetivo permitido. Un objetivo normal $\leftarrow L_1 \wedge \ldots \wedge L_n$ es un objetivo permitido si cada una de sus variables aparece en un literal positivo.

Programa Permitido. Un programa normal P es permitido si cada sentencia de P es permitida.

Programas y objetivo permitido($\Diamond$)

Un programa y un objetivo, $P \cup \{G\}$, es permitido si se cumplen las siguientes condiciones:

1. cada sentencia de P es admisible

2. si un símbolo de predicado p aparece en un literal positivo en G o en el cuerpo de alguna una sentencia del programa P, entonces toda sentencia del programa P en la que p aparece en su cabeza debe ser permitida.

3. G es permitido.

Programas permitidos, ejemplo(◊)

El siguiente programa es permitido?

```
p(x) :- q(x), not(r(x))
q(x) :- t(x,y)
t(a, a)
t(a, b)
```

El siguiente programa no es permitido?

```
p(x) :- q(x), not(r(x))
q(x) :- t(x, y)
s(x)
t(a, a)
t(a, b)
```

...

Programas permitidos, teorema($\Diamond$)

Sea P un programa normal y sea G un objetivo normal. Si $P \cup \{G\}$ es permitido entonces:

1. ninguna computación de $P \cup \{G\}$ tropieza; y

2. cada respuesta computada de $P \cup \{G\}$ es una sustitución sin variables (base) de todas las variables del objetivo G.

...

Programas jerárquicos y estratificados($\Diamond$)

El otro problema que puede provocar la incompletitud del procedimiento de resolución SLDNF es la presencia de Recursión en el programa en términos de Negación.

Para evitar este problema definiremos dos clases de programas:

- $\surd$ Programas Jerárquicos
- $\surd$ Estratificados

...

Programas jerárquicos($\Diamond$)

Un **grafo de dependencias** de un programa normal P tiene un nodo por cada símbolo de predicado que ocurre en en alguna cláusula de P y aristas dirigidas del nodo del predicado B al nodo del predicado C cuando B está en el cuerpo alguna cláusula y C es la cabeza de la misma cláusula del programa P. La arista que une a B con C se etiqueta como positiva si B es un átomo y será negativa si B es un literal negativo.

Ejemplo, elabore un grafo del siguiente programa

$$
\begin{aligned}
p(x) &\leftarrow q(x) \wedge \neg r(x) \\
r(x) &\leftarrow s(x) \\
s(x) &\leftarrow t(x,y) \wedge p(y) \\
t(x,y) &\leftarrow t(y,z) \wedge p(y) \\
q(a) & \\
t(a,b) &
\end{aligned}
$$

Programas jerárquicos ($\Diamond$)

Si el grafo de dependencias de un programa normal P no contiene ciclos entonces P es un **programa jerárquico**

Un predicado q de un programa P es **recursivo** si ocurre en un ciclo del grafo de dependencias. Dos predicados son **mutuamente recursivos** si ocurren en un ciclo del grafo de dependencias.

Si el grafo de dependencias de un programa normal contiene un ciclo entonces el programa es **recursivo**.

Programas estratificado($\Diamond$)

Un programa normal admite la negación en el cuerpo de la cláusula. Es importante ser cuidadoso en la semántica del programa ante la presencia de la negación.

$$\texttt{soltero(x)} \leftarrow \neg\texttt{casado(x)}$$
$$\texttt{casado(x)} \leftarrow \neg\texttt{soltero(x)}$$

Una solución es no permitir recursión vía la negación.

Un programa normal es estratificado si cada ciclo del grafo de dependencias está formado por aristas positivas únicamente.

Todo programa jerárquico puede ser considerado como un programa estratificado (no hay ciclos) y todos los programas definidos son jerárquicos (no hay negación).

Programas estratificado($\Diamond$)

El siguiente programa es estratificado, pero no es jerárquico y es definido

```
progenitor(X,Y)  ←  padre(X,Y).
progenitor(X,Y)  ←  madre(X,Y).
ancestro(X,Y)    ←  progenitor(X,Y).
ancestro(X,Y)    ←  progenitor(X,Z),ancestor(Z,Y).
padre(X,Y)       ←  hombre(X),hijo(Y).
madre(X,Y)       ←  mujer(X),hijo(Y).
```

Programas estratificado($\Diamond$)

El siguiente programa es estratificado

C_1 : p(x) $\leftarrow$ q(x) $\wedge \neg$r(x).
C_2 : r(x) $\leftarrow$ s(x).
C_3 : s(x) $\leftarrow$ t(x, y) $\wedge \neg$u(y).
C_4 : s(x) $\leftarrow$ u(x).
C_5 : q(a).
C_6 : r(a).
C_7 : t(a, b).

...

Programas estratificado($\Diamond$)

Un programa P es estratificado si existe una partición del programa

$$P = P_1 \cup P_2 \cup \ldots \cup P_n$$

tal que las siguientes condiciones son ciertas para todo P_i, $1 \leq i \leq n$

1. Si un átomo A ocurre positivamente en el cuerpo de una cláusula de P_i, entonces la *definición del predicado* incluido en A está en algún P_j, con $j \leq i$

2. Si un átomo A ocurre negativamente en el cuerpo de una cláusula de P_i, entonces la *definición del predicado* incluido en A está en algún P_j, con $j < i$

Cada P_i es un **estrato** donde i es su **nivel**.

Programas estratificado($\Diamond$)

En el ejemplo anterior tenemos dos particiones.

1. $P = P_1 \cup P_2 \cup P_3$ donde:
   $P_1 = \{C5, C7\}$
   $P_2 = \{C2, C3, C4, C6\}$
   $P_3 = \{C1\}$

2. $P = P_1 \cup P_2 \cup P_3 \cup P_4$ donde:
   $P_1 = \{C5\}$
   $P_2 = \{C3, C4, C7\}$
   $P_3 = \{C2, C6\}$
   $P_4 = \{C1\}$

...

Programas estratificado

El siguiente teorema establece la equivalencia entre las definiciones anteriores

Teorema. Un programa normal es estratificado (según la definición de partición) si y sólo si en su grafo de dependencias no hay ciclos con aristas negativas.

Una función de nivel ℓ de un programa normal es una función del conjunto de predicados al conjunto de enteros no-negativos. Si L es un literal, entonces el nivel del literal $\ell(L)$, es el nivel del símbolo de predicado de L. Si C es una cláusula, entonces el nivel de la cláusula $\ell(C)$, es el nivel máximo entre los símbolos de predicados de C.

Programas jerárquico

Teorema. Un programa normal P es jerárquico si existe una función de nivel ℓ tal que para cualquier sentencia de P de la forma:

$$\mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_m) \leftarrow \mathtt{L}_1, \ldots, \mathtt{L}_n$$

se cumple que $\forall i, 1 \leq i \leq n, \ell(p) > \ell(p_i)$, donde p_i es un símbolo de predicado que ocurre en el literal L_i.

Esto quiere decir que se prohíbe la presencia de recursión en el programa.

...

Programas jerárquico

El siguiente programa es jerárquico

C_1 : p(x) $\leftarrow$ q(x) $\wedge \neg$r(x).
C_2 : q(x) $\leftarrow$ $\neg$t(x).
C_3 : t(a).
C_4 : r(b).

...

Programas estratificado

Teorema. Un programa normal P es estratificado si existe una función de nivel ℓ, tal que para cualquier sentencia de P, de la forma:

$$\mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_m) \leftarrow \mathtt{L}_1, \ldots, \mathtt{L}_n$$

se cumple que:

1. $\ell(p) \geq \ell(p_i)$ para todo $L_i = p_i(t_1, \ldots, t_r)$

2. $\ell(p) > \ell(p_j)$ para todo $L_i = \neg p_i(t_1, \ldots, t_r)$

Esto quiere decir intuitivamente que en esta clase de programas no se puede dar recursión en términos de negación.

Programas estratificado

El siguiente programa es estratificado

C_1 : p(x) $\leftarrow$ $\neg$r(x) $\wedge$ p(x).
C_2 : r(x) $\leftarrow$ $\neg$q(x).
C_3 : q(a).
C_4 : q(b).

...

Completitud del SLDNF para programas estratificado

Sea P un programa jerárquico, sea G un objetivo normal y sea C una regla de computación segura. Si $P \cup \{G\}$ es permitido entonces:

"toda respuesta correcta θ para $comp(P) \cup \{G\}$ y sustitución base de todas las variables de G es una respuesta computada para $P \cup \{G\}$".

...

Francisco José Correa Zabala. U. EAFIT

Semántica declarativa de los programas normales

Inconsistente de la compleción del programa

Utilice el siguiente ejemplo para probar que la compleción del siguiente programa P, $comp(P)$, es inconsistente, o sea, no tiene ningún modelo.

C_1 : p(x) $\leftarrow$ $\neg$p(x).
C_2 : q(a).
C_3 : q(b).

Pero, si P es estratificado entonces $comp(P)$ es consistente.

Semántica declarativa de los programas normales

El operador de consecuencias inmediatas debe redefinirse

$T_P(I) = \{A \in B_P | A \leftarrow A_1 \wedge \ldots \wedge A_n$
es una instancia base de una sentencia del programa P
y $I \models \{A_1 \wedge \ldots \wedge A_n\}, I \subseteq B_P\}$

Aplicar las potencias sucesivas del operador para hallar la semántica por punto fijo del siguiente programa

C_1 : p(x) $\leftarrow$ ¬q(x).
C_2 : q(a).
C_3 : r(b).

...

Semántica declarativa de los programas normales

El operador de consecuencias inmediatas es no-monotónico

Las sucesivas aplicaciones del operador $\mathtt{T_P}$ son:

$$\mathtt{T_P} \uparrow 1 = \{\mathtt{p(a)},\ \mathtt{p(b)},\ \mathtt{q(a)},\ \mathtt{r(b)}\}$$
$$\mathtt{T_P} \uparrow 2 = \{\mathtt{q(a)},\ \mathtt{p(b)},\mathtt{r(b)}\}$$

Se puede observar que en el programa dado el T_P es no-monotónico

...

Semántica declarativa de los programas normales

El operador de consecuencias inmediatas es no-monotónico

Calcular el punto fijo del siguiente programa

C_1 : p(x) $\leftarrow$ q(x, y) $\wedge \neg$t(y).
C_2 : t(y) $\leftarrow$ s(y).
C_3 : q(a, b).
C_4 : q(b, a).
C_5 : s(b).

...

Semántica declarativa de los programas normales

El operador de consecuencias inmediatas es no-monotónico

Las sucesivas aplicaciones del operador T_P son:

$$T_P \uparrow 1 = \{q(a, b), q(b, a), s(b)\}$$
$$T_P \uparrow 2 = \{q(a, b), q(b, a), s(b), t(b), \mathbf{p(a)}, p(b)\}$$
$$T_P \uparrow 3 = \{q(a, b), q(b, a), s(b), t(b), p(b)\}$$

...

Semántica declarativa de los programas normales

Existencia de varios modelos mínimos

Dado siguiente programa

C_1 : p(a) $\leftarrow$ $\neg$q(a).
C_2 : q(b).

Tiene dos modelos mínimos

$$M_1 = \{p(a), q(b)\}$$

$$M_2 = \{q(a), q(b)\}$$

...

Semántica declarativa de los programas normales

Existencia de varios modelos mínimos

Según el ejemplo anterior, para programas normales no existe un modelo mínimo M_P en el sentido en que se cumple para los programas definidos, es decir, no existe un M_P que sea la intersección de todos los modelos M_i del programa P

$$M_P = \bigcap_{i \in I} M_i$$

Si la negación aparece en alguna sentencia de P entonces no existe un único modelo de Herbrand.

Semántica declarativa de los programas normales

Existencia de varios modelos mínimos

Dado el siguiente programa

C_1 : p(x) $\leftarrow$ q(x, y) $\wedge \neg$ t(y).
C_2 : q(a, b).
C_3 : q(b, a).
C_4 : t(b).

Es posible encontrar dos modelos mínimos, que son:

$$M_1 = \{q(a,b), q(b,a), t(b), p(b)\}$$

$$M_2 = \{q(a,b), q(b,a), t(b), t(a)\}$$

Analice las instancias de C_1. ¿Cuál es el modelo más razonable?

Semántica declarativa de los programas normales

Modelo estándar

Un modelo M de P es **soportado** si para cada elemento A de M existe una cláusula $H \leftarrow C$ de P y una sustitución ground θ tal que $A = H\theta$ y $C\theta$ es verdad en M

Ejemplo, dado el programa $\{\mathtt{p} \leftarrow \neg\mathtt{q}\}$ el modelo *minimal* $\{\mathtt{p}\}$ es soportado, mientras que $\{\mathtt{q}\}$

Si el programa P es estratificado, entonces existe una partición de P de tal forma que $P = P_1 \cup \ldots \cup P_n$ que vamos a utilizar para definir la semántica declarativa de los programas normales.

Semántica declarativa Modelo estándar

El modelo estándar lo definimos del siguiente modo.

$M_1 = \bigcap\{M/M$ es un modelo soportado de $P_1\}$
$M_2 = \bigcap\{M/M$ es un modelo soportado de $P_1 \cup P_2\}$
$M_i = \bigcap\{M/M$ es un modelo soportado de $P_1 \cup P_2 \ldots \cup P_i\}$

Ejemplo, dado el siguiente programa:

C_1 : p(x) $\leftarrow$ q(x) $\wedge \neg$r(x).
C_2 : r(x) $\leftarrow$ s(x).
C_3 : s(x) $\leftarrow$ t(x, y) $\wedge \neg$u(y).
C_4 : s(x) $\leftarrow$ u(x).
C_5 : q(a).
C_6 : r(a).
C_7 : t(a, b).

...

Semántica declarativa Modelo estándar

Una partición para el programa es

$$P_1 = \{C5\}$$
$$P_2 = \{C3, C4, C7\}$$
$$P_3 = \{C2, C6\}$$
$$P_4 = \{C1\}$$

Entonces,

$$M_1 = \{q(a)\}$$
$$M_2 = \{q(a), t(a,b), s(a)\}$$
$$M_3 = \{q(a), t(a,b), s(a), r(a)\}$$
$$M_4 = \{q(a), t(a,b), s(a), r(a)\}$$

¿Cómo determinar el modelo estándar?

...

Punto fijo por Niveles

En programas estartificado se puede demostrar que si el operador T_P se aplica sucesivamente a los niveles de $P = P_1 \cup \ldots \cup P_n$, entonces T_P es monotónico y, por tanto, tiene punto fijo.

Consideremos el programa

C_1 : p(x) $\leftarrow$ q(x, y) $\wedge \neg$t(y).
C_2 : t(y) $\leftarrow$ s(y).
C_3 : q(a, b).
C_4 : q(b, a).
C_5 : s(b).

Donde

$$P_1 = \{C3, C4, C5\}$$
$$P_2 = \{C2\}$$
$$P_3 = \{C1\}$$

Punto fijo por niveles

$$T_{P_1} \uparrow \omega(\phi) = lfp(T_{P_1}) = \{q(a,b), q(b,a), s(b)\}$$
$$T_{P_2} \uparrow \omega(lfp(T_{P_1})) = lfp(T_{P_2}) = \{q(a,b), q(b,a), s(b)\} \cup \{t(b)\}$$
$$T_{P_3} \uparrow \omega(lfp(T_{P_2})) = lfp(T_{P_3}) = \{q(a,b), q(b,a), s(b), t(b)\} \cup \{p(b)\}$$

El modelo estándar del programa coincide con el punto fijo del nivel máximo del programa. Más aún, el modelo de P es independiente de la estratificación de P.

Teorema. Sea P un programa estratificado. Entonces se cumple que:

$$MS_P = \{A \in B_P / comp(P) \models A\}$$

Punto fijo por niveles

Ejercicio. Sea P el programa estratificado siguiente:

C_1 : p(x) $\leftarrow$ q(x, y) $\wedge \neg$ t(y).
C_2 : t(y) $\leftarrow$ q(y, z) $\wedge$ t(z).
C_3 : t(y) $\leftarrow$ s(y).
C_4 : q(a, d).
C_5 : q(d, a).
C_6 : q(d, e).
C_7 : s(e).

$$MS_P = lfp(T_{P_n}) = \{q(d,e), q(a,d), q(d,a), s(e), t(e), t(d), t(a), p(b)\}$$

...

Modelo estable y modelo bien fundado

Estudiar del libro los temas Modelo estable y modelo bien fundado.

...

Francisco José Correa Zabala. U. EAFIT

Algunos teoremas

Teorema. Sea P un programa normal estratificado. Entonces $comp(P)$ es consistente.

Teorema. Sea P un programa normal. Entonces I es un modelo de $comp(P)$ si y solo si $T_p(I) = I$.

...

Programas Generales

Sentencia General. Una sentencia general tiene la forma siguiente:

$$A \leftarrow F$$

donde A es un átomo y F es cualquier fbf.

Un **programa general** es un programa cuyas sentencias son generales.

Un **objetivo general** tiene la forma $\leftarrow F$ donde F es cualquier fórmula bien formada.

Como la semántica procedural para los programas generales la da el procedimiento de resolución SLDNF (aplicable a programa normales), es necesario realizar una transformación del programa :

$$Prog.General \cup Obj.General \Rightarrow Prog.Normal \cup Obj.Normal$$

Consultar el algoritmo de LLoyd y Topor, 1984

Símbolos Predefinidos en Prolog

Algunos functores y símbolos de predicado están predefinidos como infijos (o postfijos), aparte de la notación estándar. Util para hacer los programas más legibles

Notación estándar	Notación Prolog
$+(a, /(b, c))$	$a + b/c$
$is(X, mod(34, 7))$	$Xis34mod7$
$<(+(3, 4), 8)$	$3 + 4 < 8$
$=(X, f(Y))$	$X = f(Y)$
$-(3)$	-3
$spy(/foo(, 3))$	$spy(foo)$
$:-(p(X), q(Y))$	$p(X) :- q(Y)$
$:-(p(X), ',', (q(Y), r(Z)))$	$p(X) :- q(Y), r(Z)$

Note que con esta notación, las cláusulas en Prolog se puede ver también como términos Prolog

Aritmética

El tipo de las expresiones aritméticas es:

$\surd$ Un número es una expresión aritmética

$\surd$ Si `f` es un operador aritmético de aridad n y `X1`, ..., `Xn` son expresiones aritméticas, entonces `f(X1, ..., Xn)` es una expresión aritmética

Operadores aritméticos: $+, -, *, /, //, \texttt{mod}, \ldots$ ($//$ es la división entera)

Ejemplos

♠ $\boxed{(3 * \texttt{X} + \texttt{Y})/\texttt{Z}}$ será correcta siempre que las variables `X`, `Y`, `Z` se evalúen a expresiones aritméticas (en otro caso, error).

♠ $\boxed{\texttt{a} + 3 * \texttt{X}}$ siempre será incorrecta (error)

Predicados del sistema (predefinidos):

♠ $=:=$ (igualdad), $= \backslash =$ (desigualdad), $<, >, =<, =>$

♠ `Z is X`: evalúa la expresión aritmética `X` y el resultado se unifica con `Z`

Aritmética

Sean `X`, `Y` variables enlazadas a `3` y `4`, respectivamente, y `Z` una variable libre:

- Las expresiones $\boxed{\texttt{Y} < \texttt{X} + \texttt{1}}$, $\boxed{\texttt{X is Y} + \texttt{1}}$, $\boxed{\texttt{X} =:= \texttt{Y}}$ **fallan** (se producirá backtracking)

- Las expresiones $\boxed{\texttt{Y} < \texttt{a} + \texttt{1}}$, $\boxed{\texttt{X is Z} + \texttt{1}}$ producen un **error** (se aborta la ejecución)

La suma:

```
suma(X,Y,Z) :- Z is X+Y.
```

Sólo funciona en una dirección (`X` e `Y` instanciadas a expresiones aritméticas)

Predicados Tipo

Relaciones unarias referentes al **tipo** de los términos:

- `integer(X)`
- `float(X)`
- `number(X)`
- `atom(X)` (constante)
- `atomic(X)` (constante o número)

Versión bi-direccional de `suma`:

```
suma(X,Y,Z) :- number(X),number(Y),Z is X+Y.
suma(X,Y,Z) :- number(X),number(Z),Z is Z-X.
suma(X,Y,Z) :- number(Y),number(Z),Z is Z-Y.
```

Igualdad, Identidad y Unificación

Unificación $\boxed{\texttt{Term}_1 = \texttt{Term}_2}$ Es éxito si el término 1 unifica con el término 2.

Igualdad Aritmética $\boxed{\texttt{Expr}_1 =:= \texttt{Expr}_2}$ Es éxito cuando la expresión 1 y la expresión 2 son evaluadas a un número igual. Primero evalúa ambas expresiones y después las compara.

Identidad $\boxed{\texttt{Term}_1 == \texttt{Term}_2}$ es éxito si el término 1 es equivalente al término 2. Se supone que los términos ya están asignados. $\texttt{X}_1 == \texttt{X}_2$ fracasa si no están asignadas

Identidad estructural $\boxed{\texttt{Term}_1 = @ = \texttt{Term}_2}$ es éxito si el término 1 es estructuralmente igual al término 2.

asignación $\boxed{\texttt{X is Exp}_2}$ evalúa la expresión 2 y el resultado lo unifica con la variable `X`.

Entrada/salida

Predicado	Explicación
`write(X)`	escribe el término `X` en el COS
`nl`	nueva línea en el COS
`read(X)`	lee un término (terminado en punto) del CIS y lo unifica con `X`
`put(X)`	escribe el carácter cuyo código ASCII es `N`
`get(X)`	lee el código ASCII del siguiente carácter y lo unifica con `X`
`see(File)`	`File` se convierte en el CIS
`seeing(File)`	`File` se unifica con el CIS
`seen`	Cierra el CIS
`tell(File)`	`File` se convierte en el COS
`telling(File)`	`File` se unifica con el COS
`told`	Cierra el COS

(CIS: current input stream, COS: current output stream)

⋈ Existen predicados de entrada/salida mucho más refinados

⋈Todos los predicados de entrada/salida pueden tener efectos laterales

Relación datos/solución

Member hace muchas cosas.

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

1. Verificar si un elemento está en una lista: `member(b, [a, b, c])`.

2. Generar miembros de una lista: `member(X, [a, b, c])`.

3. Generar listas: `member(a, L)`.

Relación datos/solución

append hace muchas cosas.

```
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

- Verifica si es concatenación: `append([1, 2], [a, b], [1, 2, a, b])`.
- Concatena: `append([1, 2], [a, b], X)`.
- Elimina un prefijo: `append([1, 2], X, [1, 2, a, b])`.
- Elimina un sufijo: `append(X, [a, b], [1, 2, a, b])`.
- Genera todas las posibilidades de dividir una lista: `append(X, Y, [1, 2, a, b])`.

Ejecución: el backtracking

```
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).
```

Metas u objetivos: de izquierda a derecha (regla de computación).

Cláusulas: en el orden en el que están escritas (regla de búsqueda)

Construir el árbol de `puede_nadar(Y)`.

El operador de corte

- Se denota como: !
- Restringe el backtracking:
  - Siempre tiene éxito la primera vez que se ejecuta.
  - Si por backtracking se vuelve a seleccionar, hace que se descarten las soluciones alternativas:
    - Fallan los subobjetivos del cuerpo de la cláusula a su derecha
    - Falla el predicado a la cabeza de la cláusula en la que aparece el corte

```
case(...) :- condicion1, !, accion1.
case(...) :- condicion2, !, accion2.
...
case(...) :- accionN.
```

El operador de corte

La ejecución de un **corte** obliga a Prolog a descartar todas las alternativas pendientes desde el comienzo de la ejecución del objetivo que invocó a la cláusula que contiene el corte.

Por tanto, se descartan:

- todas las cláusulas por debajo de la cláusula que contiene el corte
- todas las soluciones alternativas para los objetivos que aparecen a la izquierda del corte

pero no afecta al árbol de búsqueda de los átomos que aparecen a la derecha del corte

Operador de Corte

Cuando el sistema se encuentra un corte, el sistema es forzado a mantener todas las opciones ya satisfechas desde que se llamo la cabeza de la regla. **Objetivo padre** del corte. Cualquier intento de resatisfacer culaquier objetivo entre objetivo padre y el corte fracasará.

Se entiende como: corta opciones o congela lo hecho o descarte alternativas.

Tipos de cortes Cortes **blancos:** no eliminan soluciones

```
max(X,Y,X) :- X > Y, !.
max(X,Y,Y) :- X =< Y.
```

No afectan ni a la corrección ni a la completitud $\Rightarrow$. Es seguro.

Tipos de cortes

Cortes **verdes:** eliminan soluciones correctas que no son necesarias

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

Afectan a la completitud, pero no a la corrección $\Rightarrow$ es necesario en muchos casos, tener precaución en su uso.

Cortes **rojos:** elimina soluciones que no son correctas de acuerdo con el significado pretendido.

```
max(X,Y,X) :- X > Y, !.
max(X,Y,Y).
```

Pueden afectar a la completitud y a la corrección
(e.g., `max(6,3,3)`) $\Rightarrow$ evitar siempre que sea posible

Tipos de cortes

Los usos del corte se pueden ver desde tres áreas:

- El sistema ha encontrado la regla correcta para resolver un objetivo concreto.
- Fracasar el objetivo sin intentar encontrar soluciones alternativas. Si llegaste hasta aquí debes dejar de intentar satisfacer este objetivo.
- No generar soluciones alternativas mediante reevaluaciones.

Negación como fallo

Se implementa mediante `call`, el corte y un predicado predefinido `fail` (que siempre produce un fallo cuando se invoca)

```
not(Goal) :- call(Goal), !, fail.
not(Goal).
```

La terminación de `not(Goal)` depende de la terminación de `Goal`. Si en la ejecución de `Goal` aparece al menos una solución a la izquierda de la primera rama infinita, `not(Goal)` terminará con éxito.

`not(Goal)` nunca instancia variables de `Goal`

Negación como fallo

Resulta muy útil, pero peligroso:

```
estudiante_soltero(X) :- not(casado(X)),
                         estudiante(X).
estudiante(pedro).
casado(juan).
```

dado el objetivo `estudiante_soltero(X)` el sistema dice que **no** (cuando sí hay una solución)

La negación como fallo funciona correctamente cuando `Goal` está completamente instanciado (es misión del programador asegurar que esto ocurre)

Ejercicio

Dado el siguiente programa Prolog:

```
p(a) :- !.
p(b) :- !.
q(X) :- p(X).
```

y el objetivo `?- q(X).` ¿Cuál de las siguientes afirmaciones es falsa?

A el conjunto de respuestas computadas por Prolog no cambia si se elimina el corte de la segunda cláusula

B el conjunto de respuestas computadas por Prolog no cambia si se eliminan todos los cortes del programa, pero se lanza como objetivo `?- q(X),!.`

C el conjunto de respuestas computadas por Prolog no cambia si se eliminan todos los cortes del programa

D el conjunto de respuestas computadas por Prolog no cambia si se elimina el corte de la primera cláusula, pero se lanza como objetivo `?- q(X),!.`

Ejercicio

Dado el siguiente programa Prolog:

```
nat(0).
nat(X) :- nat(Y), X is Y+1.
p(X) :- nat(Y), d(Y,X).
d(Y,X) :- X is Y+Y.
```

y el objetivo $?-$ `p(Z).`, ¿qué valores computa el programa para la variable `Z`?

A números enteros positivos en orden ascendente

B números impares

C potencias de dos

D números pares

Ejercicio

Dado el siguiente programa Prolog:

```
pc(a,1).
pc(b,1).
acc(b,2).
p(X,Y) :- pc(X,Y), fail; acc(X,Y).
```

indica cuál es la primera respuesta computada por Prolog para el objetivo

$?-\texttt{p(Z,W).}$

A `Z = a, W = 1`

B `Z = b, W = 1`

C `Z = b, W = 2`

D el objetivo no tiene éxito

Ejercicio

Dado el siguiente programa Prolog:

```
p(X,[X]).
p(X,[_|Y]) :- p(X,Y).
```

la respuesta computada para el objetivo `?- p(X,[1,2,3,1,5]).` es:

A `X = 3`

B `X = 5`

C `X = 2`

D `X = 1`

Ejercicio

Dado el siguiente programa Prolog:

```
nose([],0).
nose([C|L],S) :- not(member(C,L)), nose(L,S1), S is C+S1.
```

en el que el predicado `member(X,L)` tiene éxito cuando `X` es un elemento de la lista `L`. Dado el objetivo $\boxed{?-\texttt{nose}([7,1,3,7],\texttt{X}).}$ ¿cuál de las siguientes afirmaciones es cierta?

A el objetivo tiene éxito y la respuesta computada es: `X = 18`

B el objetivo tiene éxito y la respuesta computada es: `X = 11`

C el objetivo falla

D el objetivo produce un error de ejecución