Apunte final - Paradigmas UBA, FCEyN.

Sebastián Andrés - 02/2025

1. Programación funcional en Haskell y esquemas de recursión.
2. Razonamiento ecuacional e inducción estructural.
3. Deducción natural para lógica proposicional y de primer orden.
4. Cálculo-λ, sistemas de tipos y semántica operacional.
5. Unificación e inferencia de tipos.
6. Programación lógica en Prolog.
7. Métodos de resolución general y SLD.
8. (Rudimentos de) programación orientada a objetos en SmallTalk.

1. Programación funcional en Haskell y esquemas de recursión.

Esquemas de recursion

```haskell
--Estructural
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

--Iterativa
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f ac [] = ac
foldl f ac (x:xs) = foldl f (f ac x) xs

--Primitiva
recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
recr f z [] = z
recr f z (x:xs) = f x xs (recr f z xs)

-- En la otra notación es esta ídea..
foldr' cEmpty cC l = case l of
    [] -> cEmpty
    x xs -> cC x (rec xs)
    where rec = foldr' cEmpty cC

recr' z f l = case l of
    [] -> z
    x xs -> f x xs (rec xs)
    where rec = recr' z f l
```

Funciones útiles de Haskell

```haskell
null :: [a] -> Bool                          -- True si la lista es vacía
head :: [a] -> a                             -- Devuelve el primer elemento de
tail :: [a] -> [a]                           -- Devuelve la lista sin el primer
length :: [a] -> Int                         -- Devuelve la longitud de la lista
init :: [a] -> [a]                           -- Devuelve la lista sin el último
last :: [a] -> a                             -- Devuelve el último elemento de
take :: Int -> [a] -> [a]                    -- Devuelve los primeros n elemento
drop :: Int -> [a] -> [a]                    -- Devuelve la lista sin los primer
(++) :: [a] -> [a] -> [a]                    -- Concatena dos listas
concat :: [[a]] -> [a]                       -- Concatena una lista de listas
(!!) :: [a] -> Int -> a                      -- Devuelve el n-ésimo elemento de
elem :: Eq a => a -> [a] -> Bool             -- True si el elemento pertenece a
zip :: [a] -> [b] -> [(a, b)]                -- Combina dos listas en una lista
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]  -- Combina dos listas con una funci
repeat :: a -> [a]                           -- Repite un elemento infinitamente
iterate :: (a -> a) -> a -> [a]              -- Aplica una función infinitamente
takeWhile :: (a -> Bool) -> [a] -> [a]       -- Devuelve los elementos que cumpl
dropWhile :: (a -> Bool) -> [a] -> [a]       -- Devuelve la lista sin los elemen
filter :: (a -> Bool) -> [a] -> [a]          -- Devuelve los elementos que cumpl
map :: (a -> b) -> [a] -> [b]                -- Aplica una función a todos los e
```

2. Razonamiento ecuacional e inducción estructural.

Demostración de igualdades con inducción

Podemos utilizar:

> Principio de reemplazo.
> Principio de extensionalidad sobre la estructura.

Demostración de isomorfismo

Para mostrar isomorfismo entre A y B ($A \simeq B$).
Armar dos funciones de la forma:
$f : A \to B$
$g : B \to A$
Y ver que sean inversas

$f.g = id : B \to B$
$g.f = id : A \to A$

3. Deducción natural para lógica proposicional y de primer orden.

$$\frac{}{\Gamma, \tau \vdash \tau}\ \text{ax} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \tau}\ \bot_e \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \wedge \sigma}\ \wedge_i \qquad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \tau}\ \wedge_{e_1} \qquad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \sigma}\ \wedge_{e_2}$$

$$\frac{\Gamma, \tau \vdash \sigma}{\Gamma \vdash \tau \Rightarrow \sigma}\ \Rightarrow_i \qquad \frac{\Gamma \vdash \tau \Rightarrow \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash \sigma}\ \Rightarrow_e$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \vee \sigma}\ \vee_{i_1} \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \tau \vee \sigma}\ \vee_{i_2} \qquad \frac{\Gamma \vdash \tau \vee \sigma \quad \Gamma, \tau \vdash \rho \quad \Gamma, \sigma \vdash \rho}{\Gamma \vdash \rho}\ \vee_e$$

$$\frac{\Gamma, \tau \vdash \bot}{\Gamma \vdash \neg\tau}\ \neg_i \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \neg\tau}{\Gamma \vdash \bot}\ \neg_e$$

$$\frac{\Gamma \vdash \sigma \quad X \notin \text{fv}(\Gamma)}{\Gamma \vdash \forall X.\sigma}\ \forall_i \qquad \frac{\Gamma \vdash \forall X.\sigma}{\Gamma \vdash \sigma\{X := t\}}\ \forall_e$$

$$\frac{\Gamma \vdash \sigma\{X := t\}}{\Gamma \vdash \exists X.\sigma}\ \exists_i \qquad \frac{\Gamma \vdash \exists X.\sigma \quad \Gamma, \sigma \vdash \tau \quad X \notin \text{fv}(\Gamma, \tau)}{\Gamma \vdash \tau}\ \exists_e$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \neg\neg\tau}\ \neg\neg_i \qquad \frac{\Gamma \vdash \tau \Rightarrow \sigma \quad \Gamma \vdash \neg\sigma}{\Gamma \vdash \neg\tau}\ \text{MT}$$

Reglas Clásicas

$$\frac{\Gamma \vdash \neg\neg\tau}{\Gamma \vdash \tau}\ \neg\neg_e \qquad \frac{\Gamma, \neg\tau \vdash \bot}{\Gamma \vdash \tau}\ \text{PBC} \qquad \frac{}{\Gamma \vdash \tau \vee \neg\tau}\ \text{LEM}$$

Validez y satisfactibilidad

Decimos que una fórmula σ es:

VÁLIDA si $\alpha \vDash_{\mathcal{M}} \sigma$ para toda $\mathcal{M}, \alpha$	SATISFACTIBLE si $\alpha \vDash_{\mathcal{M}} \sigma$ para alguna $\mathcal{M}, \alpha$
INVÁLIDA si $\alpha \nvDash_{\mathcal{M}} \sigma$ para alguna $\mathcal{M}, \alpha$	INSATISFACTIBLE si $\alpha \nvDash_{\mathcal{M}} \sigma$ para toda $\mathcal{M}, \alpha$

Observaciones

σ es VÁLIDA	sii	σ no es INVÁLIDA
σ es SATISFACTIBLE	sii	σ no es INSATISFACTIBLE
σ es VÁLIDA	sii	$\neg\sigma$ es INSATISFACTIBLE
σ es SATISFACTIBLE	sii	$\neg\sigma$ es INVÁLIDA

Modelos (M, I)

Una *sentencia* es una fórmula σ sin variables libres.
Una *teoría de primer orden* es un conjunto de sentencias.

Definición — consistencia

Una teoría $\mathcal{T}$ es *consistente* si $\mathcal{T} \nvdash \bot$.

Definición — modelo

Una estructura $\mathcal{M} = (M, I)$ es un *modelo* de una teoría $\mathcal{T}$ si vale $\alpha \vDash_{\mathcal{M}} \sigma$ para toda asignación $\alpha : \mathcal{X} \to M$ y toda fórmula $\sigma \in \mathcal{T}$.

Corrección y completitud

Teorema (Gödel, 1929)

Dada una teoría $\mathcal{T}$, son equivalentes:

1. $\mathcal{T}$ es consistente.
2. $\mathcal{T}$ tiene (al menos) un modelo.

Corolario

Dada una fórmula σ, son equivalentes:

1. $\vdash \sigma$ es derivable.
2. σ es válida.

Corolario

Dada una fórmula σ, son equivalentes:

1. $\vdash \neg\sigma$ es derivable.
2. σ es insatisfactible.

4. Cálculo-λ, sistemas de tipos y semántica operacional.

Sintaxis

Los tipos del cálculo lambda simplemente tipado con booleanos se definen mediante la siguiente gramática:

$$\sigma ::= \mathsf{Bool} \mid \sigma \to \sigma$$

y sus términos son los siguientes:

$$M ::= x \mid \lambda x : \sigma . M \mid MM \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M$$

donde $x \in \mathcal{X}$, el conjunto de todas las variables. Llamamos $\mathcal{T}$ al conjunto de todos los términos.

Variables libres y ligadas

Las variables libres son todas aquellas fuera del alcance de las λs. Se define la función $\mathsf{fv} : \mathcal{T} \to \mathcal{X}$, que dado un término devuelve un conjunto de las variables libres en él, de la siguiente manera:

$$\mathsf{fv}(x) = \{x\} \qquad \mathsf{fv}(\mathsf{true}) = \emptyset$$
$$\mathsf{fv}(\lambda x : \sigma . M) = \mathsf{fv}(M) \backslash \{x\} \qquad \mathsf{fv}(\mathsf{false}) = \emptyset$$
$$\mathsf{fv}(MN) = \mathsf{fv}(M) \cup \mathsf{fv}(N) \qquad \mathsf{fv}(\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ O) = \mathsf{fv}(M) \cup \mathsf{fv}(N) \cup \mathsf{fv}(O)$$

Un término se llama cerrado si no tiene variables libres, es decir, M es cerrado si y sólo si $\mathsf{fv}(M) = \emptyset$.

Asociatividad y precedencia

$\sigma \to \tau \to \rho = \sigma \to (\tau \to \rho) \neq (\sigma \to \tau) \to \rho$	Las flechas en los tipos asocian a derecha.
$MNO = (MN)O \neq M(NO)$	La aplicación asocia a izquierda.
$\lambda x : \sigma.MN = \lambda x : \sigma.(MN) \neq (\lambda x : \sigma.M)N$	El cuerpo de la lambda se extiende hasta el final del término, excepto que haya paréntesis.

Semántica operacional

Consiste en un conjunto de reglas que definen la relación de reducción $\to$ entre términos.
Cuando $M \to N$, decimos que M reduce o reescribe a N.

Formas normales

Un término es o está en forma normal cuando no existe ninguna regla que lo reduzca a otro.

Determinismo

Decimos que la semántica es determinística cuando cada término que no está en forma normal tiene sólo una forma de reducir.

Estrategias de reducción

Para implementar un lenguaje, necesitamos una relación de reducción que sea determinística. En la teórica se vieron las estrategias call-by-name y call-by-value. En la parte práctica de la materia vamos a usar la estrategia call-by-value, y en particular nos van a interesar las extensiones determinísticas del cálculo lambda.

> #### Valores
> Los valores son los resultados esperados de los programas. Se definen como los términos cerrados V producidos por la gramática de valores.

La siguiente gramática de valores y reglas de reducción definen la estrategia call-by-value.

$$V ::= \text{true} \mid \text{false} \mid \lambda x : \sigma.M$$

$$(\lambda x : \sigma.M)V \to M\{x := V\} \qquad (\beta)$$
$$\text{if true then } M \text{ else } N \to M \qquad (\text{if}_\text{t})$$
$$\text{if false then } M \text{ else } N \to N \qquad (\text{if}_\text{f})$$

Si $M \to N$, entonces

$$MO \to NO \qquad (\mu)$$
$$VM \to VN \qquad (\nu)$$
$$\text{if } M \text{ then } O \text{ else } P \to \text{if } N \text{ then } O \text{ else } P \qquad (\text{if}_c)$$

Extensión con números naturales

Semántica operacional

Se extienden los valores de la siguiente manera:

$$V ::= \ldots \mid \mathsf{zero} \mid \mathsf{succ}(V)$$

Además, usamos la notación $\underline{n}$ para $\mathsf{succ}^n(\mathsf{zero})$ con $n \geq 0$.
Se extiende la semántica operacional con las siguientes reglas:

$$\mathsf{pred}(\mathsf{succ}(V)) \to V \qquad (\mathsf{pred})$$
$$\mathsf{isZero}(\mathsf{zero}) \to \mathsf{true} \qquad (\mathsf{isZero}_0)$$
$$\mathsf{isZero}(\mathsf{succ}(V)) \to \mathsf{false} \qquad (\mathsf{isZero}_n)$$

Si $M \to N$, entonces

$$\mathsf{succ}(M) \to \mathsf{succ}(N) \qquad (\mathsf{succ}_c)$$
$$\mathsf{pred}(M) \to \mathsf{pred}(N) \qquad (\mathsf{pred}_c)$$
$$\mathsf{isZero}(M) \to \mathsf{isZero}(N) \qquad (\mathsf{isZero}_c)$$

Tipado

Tipos: La gramática que define los tipos del cálculo lambda simplemente tipado con booleanos es:

$$\sigma ::= \mathsf{Bool} \mid \sigma \to \sigma$$

Los contextos son conjuntos finitos de asociaciones entre tipos y variables. Por ejemplo:

$$\Gamma_1 = y{:}\mathsf{Bool} \to \mathsf{Bool} \qquad \Gamma_2 = y{:}\mathsf{Bool} \to \mathsf{Bool}, x{:}\mathsf{Bool}$$

son contextos válidos, pero

$$\Gamma_3 = y{:}\mathsf{Bool} \to \mathsf{Bool}, y{:}\mathsf{Bool}$$

no lo es.
Juicios de tipado: Un juicio de tipado es la relación $\Gamma \vdash M : \tau$ y se lee "en el contexto Γ, M es de tipo τ ". Por ejemplo:

$$x{:}\mathsf{Bool} \to \mathsf{Bool} \vdash x : \mathsf{Bool} \to \mathsf{Bool}$$
$$\vdash \mathsf{true} : \mathsf{Bool}$$
$$f{:}\mathsf{Bool} \to \mathsf{Bool}, x{:}\mathsf{Bool} \vdash fx : \mathsf{Bool}$$

son juicios de tipado válidos.

Sistema de tipado

Los juicios de tipado $\Gamma \vdash M : \tau$ válidos se pueden derivar mediante el siguiente sistema de reglas de deducción:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ ax_v \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x{:}\tau.M : \tau \to \sigma}\ \to_i \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}\ \to_e$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ ax_t \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ ax_f \qquad \frac{\Gamma \vdash M : \mathsf{Bool} \quad \Gamma \vdash N_1 : \tau \quad \Gamma \vdash N_2 : \tau}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \tau}\ \texttt{if}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}}\ \mathsf{zero} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}(M) : \mathsf{Nat}}\ \mathsf{succ}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}(M) : \mathsf{Nat}}\ \mathsf{pred} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{isZero}(M) : \mathsf{Bool}}\ \mathsf{isZero}$$

Sistemas polimórficos de tipos

El texto menciona dos sistemas polimórficos: el Sistema F y el polimorfismo let.

Sistema F

Los tipos en el Sistema F son los mismos que los de tipos simples, pero se agregan variables y un cuantificador universal para ligar esas variables. El Sistema F no es dirigido por sintaxis, lo que significa que, dado un término, no siempre es trivial saber qué regla de tipado aplicar. De hecho, no existe un algoritmo de inferencia que pueda hacerlo, es indecidible.

2.1.2. Sistema F

Los tipos en Sistema F son los mismos de tipos simples (ver Definición 1.24) , a los que se agregan variables y para-todo ligando esas variables:

$$\tau ::= X \mid \texttt{Bool} \mid \tau \Rightarrow \tau \mid \forall X.\tau$$

A continuación damos las reglas de Sistema F. Como se puede apreciar, las reglas son exactamente las mismas que las de tipos simples (ver Definición 1.25), a las que se agregan las reglas de introducción y eliminación del para-todo.

Definición 2.4 (Sistema F)

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ ax_v \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x{:}\tau.M : \tau \to \sigma} \to_i \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma} \to_e$$

$$\frac{}{\Gamma \vdash \mathbf{tt} : \texttt{Bool}}\ ax_t \qquad \frac{}{\Gamma \vdash \mathbf{ff} : \texttt{Bool}}\ ax_f \qquad \frac{\Gamma \vdash M : \texttt{Bool} \quad \Gamma \vdash N_1 : \tau \quad \Gamma \vdash N_2 : \tau}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \tau}\ \texttt{if}$$

$$\frac{\Gamma \vdash M : \tau \quad X \notin \mathsf{FV}(\Gamma)}{\Gamma \vdash M : \forall X.\tau}\ \forall_i \qquad \frac{\Gamma \vdash M : \forall X.\tau}{\Gamma \vdash M : \tau\{X := \sigma\}}\ \forall_e$$

Donde si $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, entonces $\mathsf{FV}(\Gamma) = \mathsf{FV}(\tau_1) \cup \cdots \cup \mathsf{FV}(\tau_n)$.
¿Quién es $\mathsf{FV}(\tau)$?

Polimorfismo LET

Este sistema es menos general que el Sistema F, pero es dirigido por sintaxis y decidible. En el polimorfismo let, solo se permite el cuantificador universal (∀) en la variable ligada por el let. Se distinguen los tipos sin cuantificadores (llamados "tipos") de los tipos cuantificados (llamados "esquemas de tipos"). Este sistema permite un buen compromiso entre la reusabilidad de código y la inferencia de tipos. El sistema de tipos de Hindley-Milner es un sistema de tipos para el polimorfismo let en el que cada término tiene una sola regla para derivar su tipo, lo que lo hace dirigido por sintaxis.

La nueva gramática de términos es:

$$M ::= x \mid \lambda x{:}\tau.M \mid MM \mid \mathbf{tt} \mid \mathbf{ff} \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid \mathsf{let}\ x = M\ \mathsf{in}\ M$$

y la semántica operacional se extiende con la siguiente regla:

$$\mathsf{let}\ x = N\ \mathsf{in}\ M \to M\{x := N\}$$

Notar que $\mathsf{let}\ x = N\ \mathsf{in}\ M$ reduce exactamente igual que $(\lambda x.M)N$.
Finalmente, las reglas de tipado simple de la Definición 1.25 se extienden con la siguiente regla:

$$\frac{\Gamma \vdash M : \tau \quad \Gamma, x : \tau \vdash N : \sigma}{\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M : \sigma}\ \mathsf{let}$$

Primera aproximación: Damos un primer sistema de tipos para polimorfismo let, que no es dirigido por sintaxis, pero cumple con que sólo se puede cuantificar la variable del let.

Definición 2.9 (Polimorfismo let) El sistema de tipos asocia contextos y términos con esquemas de tipos, $\Gamma \vdash M : e$, y viene dado por

$$\frac{}{\Gamma, x : e \vdash x : e}\ ax_v \qquad \frac{\Gamma, x : [\tau] \vdash M : [\sigma]}{\Gamma \vdash \lambda x{:}[\tau].M : [\tau \to \sigma]} \to_i \qquad \frac{\Gamma \vdash M : [\tau \to \sigma] \quad \Gamma \vdash N : [\tau]}{\Gamma \vdash MN : [\sigma]} \to_e$$

$$\frac{}{\Gamma \vdash \mathbf{tt} : [\mathtt{Bool}]}\ ax_t \quad \frac{}{\Gamma \vdash \mathbf{ff} : [\mathtt{Bool}]}\ ax_f \quad \frac{\Gamma \vdash M : [\mathtt{Bool}] \quad \Gamma \vdash N_1 : [\tau] \quad \Gamma \vdash N_2 : [\tau]}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : [\tau]}\ \mathtt{if}$$

$$\frac{\Gamma \vdash N : e \quad \Gamma, x : e \vdash M : [\tau]}{\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M : [\tau]}\ \mathsf{let} \qquad \frac{\Gamma \vdash M : e \quad X \notin \mathsf{FV}(\Gamma)}{\Gamma \vdash M : \forall X.e}\ \forall_i \qquad \frac{\Gamma \vdash M : \forall X.e}{\Gamma \vdash M : e\{X := \tau\}}\ \forall_e$$

En la definición anterior se atribuye un esquema de tipo a cada término, en particular a las variables. La regla $\to_i$ pide $x : [\tau]$, es decir, un esquema sin cuantificar. Sólo let permite darle a la variable cualquier esquema.

Sistema dirigido por sintaxis de Hindley-Milner: La idea es que la inferencia de el tipos sea decidible, para ello se toma un sistema de tipos en el que cada término tiene una sola regla para derivar su tipo (eso es, dirigido por sintaxis, como teníamos en tipos simples).

Definición 2.11 (Relación de orden entre esquemas de tipos) La relación $\preceq$ es una relación de orden entre esquemas de tipos definida por las siguientes reglas.

$$e \preceq \forall X.e, \text{ si } X \notin \mathsf{FV}(e) \qquad \text{y} \qquad \forall X.e \preceq e\{X := \tau\}$$

Definición 2.12 (Cierre de un esquema de tipo respecto a un contexto) Sea Γ un contexto, τ un tipo, y $\vec{X} = \mathsf{FV}(\tau) \setminus \mathsf{FV}(\Gamma)$. Definimos el cierre de τ respecto a Γ como $\vec{\Gamma}(\tau) = \forall \vec{X}.[\tau]$.

Ejemplo 2.13

- El cierre de `Bool` respecto a cualquier Γ es $\vec{\Gamma}(\texttt{Bool}) = [\texttt{Bool}]$.

- El cierre de $X \Rightarrow Y \Rightarrow Z$ respecto a $\Gamma = x : X, w : W$ es

$$\{x : X, \vec{w} : W\}(X \Rightarrow Y \Rightarrow Z) = \forall Y.\forall Z.[X \Rightarrow Y \Rightarrow Z]$$

Definición 2.14 (Sistema de tipos de Hindley-Milner)

$$\frac{e \preceq e'}{\Gamma, x : e \vdash x : e'}\ ax_v \qquad \frac{\Gamma, x : [\tau] \vdash M : [\sigma]}{\Gamma \vdash \lambda x{:}[\tau].M : [\tau \to \sigma]}\ \to_i \qquad \frac{\Gamma \vdash M : [\tau \to \sigma] \quad \Gamma \vdash N : [\tau]}{\Gamma \vdash MN : [\sigma]}\ \to_e$$

$$\frac{}{\Gamma \vdash \mathtt{tt} : [\texttt{Bool}]}\ ax_t \qquad \frac{}{\Gamma \vdash \mathtt{ff} : [\texttt{Bool}]}\ ax_f \qquad \frac{\Gamma \vdash M : [\texttt{Bool}] \quad \Gamma \vdash N_1 : [\tau] \quad \Gamma \vdash N_2 : [\tau]}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : [\tau]}\ \mathtt{if}$$

$$\frac{\Gamma \vdash N : [\tau] \quad \Gamma, x : \vec{\Gamma}(\tau) \vdash M : [\sigma]}{\Gamma \vdash \mathsf{let}\ x = N\ \mathsf{in}\ M : [\sigma]}\ \mathsf{let}$$

Observación. Si permitimos $\forall$ en λ obtenemos el Sistema F de la Definición 2.4, pero la tipabilidad es indecidible (teorema de Wells), es decir, está demostrado que no existe algoritmo de inferencia.
Por eso, dar polimorfismo sólo a let es un buen compromiso que permite reusabilidad de código e inferencia de tipos.

5. Unificación e inferencia de tipos.

El resto esta en las hojas del machete inferencia de tipos:

Algoritmo de unificación

El algoritmo de unificación que conocíamos se adapta a términos de primer orden sólo cambiando la notación:

$$\{X \stackrel{?}{=} X\} \cup E \xrightarrow{\texttt{Delete}} E$$

$$\{\mathtt{f}(t_1, \ldots, t_n) \stackrel{?}{=} \mathtt{f}(s_1, \ldots, s_n)\} \cup E \xrightarrow{\texttt{Decompose}} \{t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n\} \cup E$$

$$\{t \stackrel{?}{=} X\} \cup E \xrightarrow{\texttt{Swap}} \{X \stackrel{?}{=} t\} \cup E \quad \text{si } t \text{ no es una variable}$$

$$\{X \stackrel{?}{=} t\} \cup E \xrightarrow{\texttt{Elim}}_{\{X:=t\}} E\{X{:=}t\} \quad \text{si } X \notin \mathrm{fv}(t)$$

$$\{\mathtt{f}(t_1, \ldots, t_n) \stackrel{?}{=} \mathtt{g}(s_1, \ldots, s_m)\} \cup E \xrightarrow{\texttt{Clash}} \texttt{falla} \quad \text{si } \mathtt{f} \neq \mathtt{g}$$

$$\{X \stackrel{?}{=} t\} \cup E \xrightarrow{\texttt{Occurs-Check}} \texttt{falla} \quad \text{si } X \neq t \text{ y } X \in \mathrm{fv}(t)$$

6. Programación lógica en Prolog.

Un programa en Prolog es un conjunto de cláusulas de definición.
Una consulta en Prolog es una cláusula objetivo.

La ejecución se basa en la regla de resolución SLD.
Escrita en notación de Prolog:

$$\frac{?\text{-}\ \mathrm{p}(t_1, \ldots, t_k), \sigma_1, \ldots, \sigma_n. \qquad \mathrm{p}(s_1, \ldots, s_k)\ \text{:-}\ \tau_1, \ldots, \tau_m. \qquad \mathbf{S} = \mathrm{mgu}(\mathrm{p}(t_1, \ldots, t_k) \stackrel{?}{=} \mathrm{p}(s_1, \ldots, s_k))}{\mathbf{S}(?\text{-}\ \tau_1, \ldots, \tau_m, \sigma_1, \ldots, \sigma_n.)}$$

Prolog busca sucesivamente todas las refutaciones haciendo DFS.

Criterio de busqueda: las reglas se usan en orden de aparicion.

Al evaluar un goal, los resultados posibles son los siguientes:

1. True: la resolucion termino en la clausula vacıa.
2. False: la resolucion termino en una clausula que no unifica con ninguna regla del programa.
3. El proceso de aplicacion de la regla de resolucion no termina

La exploración *depth-first* (DFS) es **incompleta**.
Puede provocar que Prolog nunca encuentre refutaciones posibles.

Ejemplo — incompletitud de DFS

```
esMaravilloso(X) :- esMaravilloso(suc(X)).
esMaravilloso(cero).
```

```
?- esMaravilloso(cero).
└─?- esMaravilloso(suc(cero)).
  └─?- esMaravilloso(suc(suc(cero))).
    └─?- esMaravilloso(suc(suc(suc(cero)))).
      └─...
```

El orden de las reglas se torna relevante.

Tradeoff: mayor eficiencia a cambio de menor declaratividad.
La exploración *breadth-first* (BFS) es completa pero muy costosa.

Al unificar, Prolog **no** usa la regla *occurs-check*.
Por ejemplo, `X` unifica con `f(X)`. Esto es **incorrecto**.
Puede provocar que Prolog encuentre una "refutación" incorrecta.

Ejemplo — refutación incorrecta por omisión de *occurs check*

```
esElSucesor(X, suc(X)).
```

```
?- esElSucesor(Y, Y).
└─✓ .....{Y := X, X := suc(X)}
```

Tradeoff: mayor eficiencia a cambio de incorrección lógica.
En muchos contextos la regla *occurs-check* es innecesaria.
La carga de probar corrección recae en lxs programadorxs.

Árboles de búsqueda

```
% Ejemplo

S(X,Y)
|-> p(X), not(r(X,Y))
|    |-> not(r(a,Y)) ............................. {X = a}
|    |    |-> r(a,Y), !, Fail
|    |    |    |-> q(a), !, p(Y), !, Fail
|    |    |        |-> !, p(Y), !, Fail
|    |    |           |-> p(Y), !, Fail
|    |    |           |   |-> !, Fail, {Y = a}
|    |    |           |        |-> !, Fail
|    |    |           |            |-> Fail.
|    |    |           |               |-> X
|    |    |           |-> cut
|    |    |-> cut
|    |-> not(r(b,Y)) ----------------------- {X = b}
|        |-> r(b, Y), !, Fail
|            |-> q(b), !, p(Y)
|                |-> X
|-> q(c), r(c, Y).
    |-> r(c, Y)
        |-> q(c), !, p(Y)
            |-> !, p(Y)
                |-> p(Y)
                    |-> V --------------- {Y = a}
                    |-> V --------------- {Y = b}
```

Esquema Generate & Test (G&T)

Una técnica que usaremos muy a menudo es:

- i. Generar todas las posibles soluciones de un problema (los candidatos a solucion, segun cierto criterio general)
- ii. Testear cada una de las soluciones generadas (hacer que fallen los candidatos que no cumplan cierto criterio particular)

La idea se basa fuertemente en el orden en que se procesan las reglas.

```
pred(X1,...,Xn) :- generate(X1, ...,Xm), test(X1, ...,Xm).
% generate(...) debera instanciar ciertas variables.
% test(...) debera verificar si los valores intanciados pertenecen a la solucion,
%           pudiendo para ello asumir que ya esta instanciada.

% Por ejemplo:
listasGemelas([], []).
listasGemelas(A, B) :-
    desde(0, N), % genero la longitud de las listas capicuas
    generarListaTam(N, A), % genero una lista de tamaño N
    generarListaTam(N, B), % genero otra lista de tamaño N
    sonEspejo(A, B).       % testeo la condicion
```

Reversibilidad

Un predicado define una relacion entre elementos. No hay parametros de “entrada” ni de “salida”.

Conceptualmente, cualquier argumento podrıa cumplir ambos roles dependiendo de como se consulte.

Un predicado podrıa estar implementado asumiendo que ciertas variables ya estan instanciadas, por diversas cuestiones practicas.

Cuando un argumento puede venir instanciado o instanciarse se lo nota $?X$.

Aritmetica

Aritmética

Algunos operadores aritméticos:

- `E1 < E2, E1 =< E2, E1 =:= E2, E1 =\= E2`: evalúa ambas expresiones aritméticas y realiza la comparación indicada.
- `X is E`: tiene éxito sí y sólo si `X` unifica con el resultado de evaluar la expresión aritmética `E`.

Algunos operadores no aritméticos:

- `X = Y`: tiene éxito si y sólo si `X` unifica con `Y`.
- `X \= Y`: `X` no unifica con `Y`. Ambos términos deben estar instanciados.

Metapredicados

not

not(p(X1, ..., Xn)) tiene exito si no existe instanciacion posible para las variables no instanciadas en $X_1 ... X_n$ que haga que P tenga exito.
El not no deja instanciadas las variables libres luego de su ejecucion.

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

% O tambien
not(P(X)) = P(X), !, Fail.
```

! (cut)

El predicado ! tiene exito inmediatamente.

Al momento de hacer backtracking:

- Se vuelve atrás hasta el punto en el que se eligió usar la regla que hizo aparecer el operador de corte.
- Se descartan todas las elecciones alternativas.
- Se continúa buscando hacia atrás.

Ejemplo — cortes "benignos" (green cuts)

En algunos casos, ! no altera la semántica del programa.
Puede servir para construir programas equivalentes más eficien

```
add(N, zero, N) :- !.
add(zero, N, N).
add(suc(N), M, suc(P)) :- add(N, M, P).

?- add(suc(suc(suc(suc(...)))), zero, P).
```

Ejemplo — cortes "riesgosos" (red cuts)

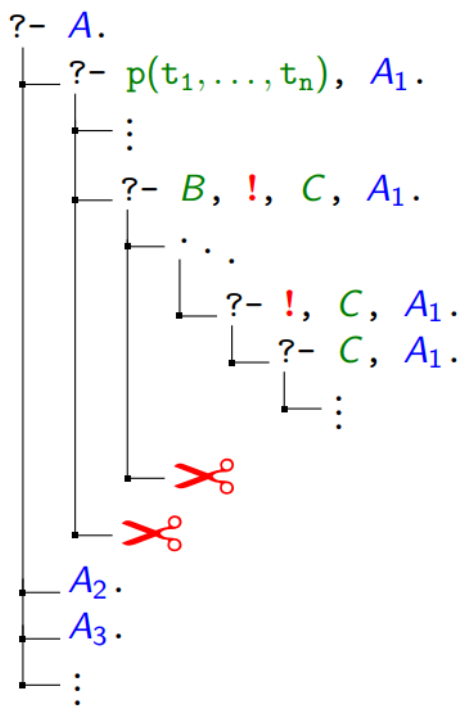
En otros casos, la semántica puede verse alterada.

```
maximo(A, B, A) :- A >= B, !.
maximo(A, B, B).

?- maximo(2, 1, B).
>> B = 2

?- maximo(2, 1, 1).
>> true.
```

Gráficamente:



setof

```
% setof(-Var, +Goal, -Set)
% unifica Set con la lista sin repetidos de Var que satisfacen Goal.

% Un ejemplo:
% ?- setof((X,Y), (between(2,3,X), Y is X + 2), L).
% L = [(2, 4), (3, 5)].
```

Predicados útiles

```
% var(A) tiene exito si A es una variable libre.
% nonvar(A) tiene exito si A no es una variable libre.
% ground(A) tiene exito si A no contiene variables libres.
% between(+A, +B, -C)

% desde(+M, -N)
desde(0,0).
desde(0, N) :- M is N+1, desde(0, M).
```

7. Métodos de resolución general y SLD.

Algoritmo de resolución

La idea del algoritmo es que para demostrar A válida vale mostrar que ~A es insatisfactible.

$$\text{A valida} \iff \neg\text{A insatisfactible}$$

Para demostrar que A se deduce de $H_1, ..., H_n$, demostramos que $H_1, ..., H_n, \neg A$ es insatisfactible.

$$\text{QvQ: } (H_1 \wedge \cdots \wedge H_n) \rightarrow A$$

$$\neg((H_1 \wedge \ldots H_n) \rightarrow A) \equiv H_1 \wedge \cdots \wedge H_n \wedge \neg A$$

Resolución para lógica proposicional

Entrada: una fórmula σ de la lógica proposicional.
Salida: un booleano que indica si σ es válida.

Método de resolución

1. Escribir $\neg\sigma$ como un conjunto $\mathcal{C}$ de **cláusulas**.
   (Pasar a *forma clausal*).
2. Buscar una **refutación** de $\mathcal{C}$.
   Una refutación de $\mathcal{C}$ es una derivación de $\mathcal{C} \vdash \bot$.

Si se encuentra una refutación de $\mathcal{C}$:
Vale $\neg\sigma \vdash \bot$. Es decir, $\neg\sigma$ es insatisfactible/contradicción.
Luego vale $\vdash \sigma$. Es decir, σ es válida/tautología.
Si no se encuentra una refutación de $\mathcal{C}$:
No vale $\neg\sigma \vdash \bot$. Es decir, σ es satisfactible.
Luego no vale $\vdash \sigma$. Es decir, σ no es válida.

Resolución para lógica de primer orden

Entrada: una fórmula σ de la lógica de primer orden.
Salida: un booleano indicando si σ es válida.
Si σ es válida, el método siempre termina.
Si σ no es válida, el método puede no terminar.

Método de resolución de primer orden (Procedimiento de semi-decisión)

1. Escribir $\neg\sigma$ como un conjunto $\mathcal{C}$ de **cláusulas**.
2. Buscar una **refutación** de $\mathcal{C}$.
   Si existe alguna refutación, el método encuentra alguna.
   Si no existe una refutación, el método puede "colgarse".

Teoremas LPO

Teorema (corrección del pasaje a forma clausal)

Dada una fórmula σ:

1. El pasaje a forma clausal termina.
2. El conjunto de cláusulas $\mathcal{C}$ obtenido es equisatisfactible a σ.
   Es decir, σ es sat. si y sólo si $\mathcal{C}$ es sat..

Teorema (corrección del algoritmo de refutación)

Dado un conjunto de cláusulas $\mathcal{C}_0$:

1. Si $\mathcal{C}_0 \vdash \bot$, entonces el algoritmo de refutación termina.
2. El algoritmo retorna `INSAT` si y sólo si $\mathcal{C}_0 \vdash \bot$.

Si $\mathcal{C}_0 \nvdash \bot$, no hay garantía de terminación.

Reglas

La regla de resolución en el marco proposicional

$$\frac{\mathcal{A}_i = \{A_1, \ldots, A_m, Q\} \qquad \mathcal{A}_j = \{B_1, \ldots, B_n, \neg Q\}}{\mathcal{B} = \{A_1, \ldots, A_m, B_1, \ldots, B_n\}}$$

- A $\mathcal{B}$ se lo llama **resolvente** (de $\mathcal{A}_i$ y $\mathcal{A}_j$)
- La regla se apoya en el hecho de que la siguiente proposición es una tautología:

$$(A \vee P) \wedge (B \vee \neg P) \Leftrightarrow (A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)$$

- El conjunto de cláusulas $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ es lógicamente equivalente a $\{\mathcal{A}_1, \ldots, \mathcal{A}_k, \mathcal{B}\}$

La regla de resolución en primer orden

$$\frac{\mathcal{A}_i = \{A_1, \ldots, A_m, P_1, \ldots, P_k\} \quad \mathcal{A}_j = \{B_1, \ldots, B_n, \neg Q_1, \ldots, \neg Q_l\}}{\mathcal{B} = S(\{A_1, \ldots, A_m, B_1, \ldots, B_n\})}$$

- S es el MGU de $\{P_1 \stackrel{?}{=} \ldots \stackrel{?}{=} P_k \stackrel{?}{=} Q_1 \stackrel{?}{=} \ldots \stackrel{?}{=} Q_l\}$ es decir, $S(P_1) = \ldots = S(P_k) = S(Q_1) = \ldots = S(Q_l)$.
- A $\mathcal{B}$ se la llama **resolvente** (de $\mathcal{A}_i$ y $\mathcal{A}_j$).
- Cada paso de resolución **preserva satisfactibilidad** (Teorema de Herbrand-Skolem-Gödel).

¿Qué representa la cláusula vacía?

En el cálculo de resolución, cada cláusula es una disyunción de literales.
La resolución busca derivar nuevas cláusulas hasta que:
Se obtiene una cláusula vacía (∅), lo que significa que hemos encontrado una contradicción.
No hay más resolventes posibles, lo que indica que no podemos derivar una contradicción.
La cláusula vacía representa una disyunción sin literales, es decir, una proposición que siempre es falsa. Como hemos partido de una hipótesis asumida como verdadera, obtener algo falso implica que la hipótesis inicial no podía ser cierta, lo que demuestra su inconsistencia.

Sobre las clausulas

Recordemos que una cláusula es un conjunto de literales:

$$\{\ell_1, \ldots, \ell_n\}$$

donde cada literal es una fórmula atómica posiblemente negada:

$$\ell ::= \underbrace{\mathbf{P}(t_1, \ldots, t_n)}_{\text{literal positivo}} \mid \underbrace{\neg \mathbf{P}(t_1, \ldots, t_n)}_{\text{literal negativo}}$$

Definición (Cláusulas de Horn)

Las cláusulas son de los siguientes tipos, dependiendo del número de literales positivos/negativos que contienen:

	#positivos	#negativos
cláusula **objetivo**	0	$*$
cláusula de **definición**	1	$*$
cláusula de **Horn**	≤ 1	$*$

Forma Normal Conjuntiva

El método trabaja con fórmulas en FNC (Forma Normal Conjuntiva).
Esto es una conjunción de clausulas que son disyunción de literales (fórmulas atómicas o su

negacion).

Por ejemplo:

- $\{\neg menor(X,Y), menor(c,Y)\}$ representa la clausula $\forall X.\forall Y.(\neg menor(X,Y) \vee menor(c,Y))$
- $\{\{\neg menor(X,Y), menor(c,Y)\}, \{impar(Z), mayor(Z,w)\}\}$ son dos clausulas que representan a $\forall X.\forall Y.(\neg menor(X,Y) \vee menor(c,Y)) \wedge \forall Z.(impar(Z) \vee mayor(Z,w))$

Clausulas Horn

- Cláusula de Horn
  - Cláusula de la forma $\forall x_1 \ldots \forall x_m.C$ tal que la disyunción de literales C tiene a lo sumo un literal positivo.
- Cláusula de definición ("Definite Clause")
  - Cláusula de la forma $\forall x_1 \ldots \forall x_m.C$ tal que la disyunción de literales C tiene exactamente un literal positivo.
- Sea $H = P \cup \{G\}$ un conjunto de cláusulas de Horn (con nombre de variables disjuntos) tal que
  - P conjunto de cláusulas de definición y
  - G una cláusula sin literales positivos.
- $H = P \cup \{G\}$ son las cláusulas de entrada.
  - P se conoce como el programa o base de conocimientos y
  - G el goal, meta o cláusula objetivo.

Pasajes a forma clausal

Lógica proposicional

Resumen — pasaje a forma clausal

1. Reescribir $\Rightarrow$ usando $\neg$ y $\vee$.
2. Pasar a f.n. negada, empujando $\neg$ hacia adentro.
3. Pasar a f.n. conjuntiva, distribuyendo $\vee$ sobre $\wedge$.

Lógica de primer orden

Resumen — pasaje a forma clausal en lógica de primer orden

1. Reescribir $\Rightarrow$ usando $\neg$ y $\vee$.
2. Pasar a f.n. negada, empujando $\neg$ hacia adentro.
3. Pasar a f.n. prenexa, extrayendo $\forall, \exists$ hacia afuera.
4. Pasar a f.n. de Skolem, Skolemizando los existenciales.
5. Pasar a f.n. conjuntiva, distribuyendo $\vee$ sobre $\wedge$.
6. Empujar los cuantificadores hacia adentro de las conjunciones.

Cada paso produce una fórmula equivalente,
excepto la Skolemización que sólo preserva satisfactibilidad.

1. Reemplazar los $(a \to b)$ con $(\neg a \vee b)$.
2. Empujar el $\neg$ hacia adentro (tras esto obtenemos la forma normal negada).
   - $\neg(\forall X.\sigma) \equiv \exists X.\neg\sigma$
   - $\neg(\exists X.\sigma) \equiv \forall X.\neg\sigma$
3. Extraer los cuantificadores $(\forall, \exists)$ hacia afuera asumiendo $X \notin fv(\tau)$ (tras esto obtenemos la forma normal prenexa).
   - i.e. $(\forall X.\sigma) \wedge \tau \equiv \forall X.(\sigma \wedge \tau)$
4. Skolemnizacion :: Deshacerse de los cuantificadores existenciales usando la tecnica de Herbrand-Skolem (Y en funcion de las variables ligadas antes por cuantificadores, si no hay, constante). Preserva la satisfactibilidad no la validez.

$$\forall X_1..X_n \exists Y P(X,Y) \equiv \forall X_1..X_n P(X, f(X_1...X_n))$$

$$\exists Y P(Y) \equiv P(c)$$

5. Pasaje a forma normal conjuntiva (la disjuncion de uniones). Por ejemplo:
   - $\sigma \vee (\tau \wedge \rho) \rightsquigarrow (\sigma \vee \tau) \wedge (\sigma \vee \rho)$
6. Empujar los cuantificadores universales hacia el centro.

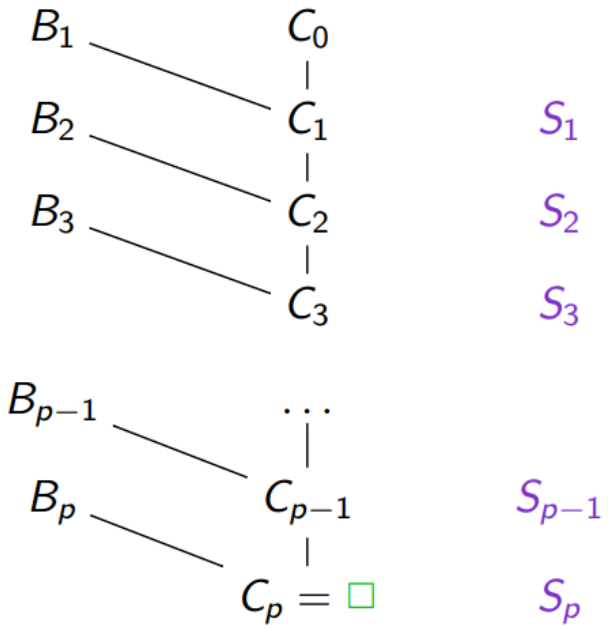
$$\forall \mathtt{X}_1 \ldots \mathtt{X}_n. \left(\begin{array}{l} (\ell_1^{(1)} \vee \ldots \vee \ell_{m_1}^{(1)}) \\ \wedge\ (\ell_1^{(2)} \vee \ldots \vee \ell_{m_2}^{(2)}) \\ \ldots \\ \wedge\ (\ell_1^{(k)} \vee \ldots \vee \ell_{m_k}^{(k)}) \end{array} \right) \rightarrow \left(\begin{array}{l} \forall \mathtt{X}_1 \ldots \mathtt{X}_n.(\ell_1^{(1)} \vee \ldots \vee \ell_{m_1}^{(1)}) \\ \wedge\ \forall \mathtt{X}_1 \ldots \mathtt{X}_n.(\ell_1^{(2)} \vee \ldots \vee \ell_{m_2}^{(2)}) \\ \ldots \\ \wedge\ \forall \mathtt{X}_1 \ldots \mathtt{X}_n.(\ell_1^{(k)} \vee \ldots \vee \ell_{m_k}^{(k)}) \end{array} \right)$$

Por último la **forma clausal** es:

$$\left\{ \begin{array}{l} \{\ell_1^{(1)}, \ldots, \ell_{m_1}^{(1)}\}, \\ \{\ell_1^{(2)}, \ldots, \ell_{m_2}^{(2)}\}, \\ \vdots \\ \{\ell_1^{(k)}, \ldots, \ell_{m_k}^{(k)}\} \end{array} \right\}$$

Resolucion SLD

Si un conjunto de cláusulas $\mathcal{C}$ es insatisfactible, existe una secuencia de pasos de resolución *lineal* que lo refuta (prueba su insatisfactibilidad). Es decir, una secuencia de la forma:



donde C_0 y cada B_i es un elemento de $\mathcal{C}$ o algún C_j con $j < i$.

En el algoritmo visto de resolución, el espacio de búsqueda - inicialmente cuadrático - crece en cada paso.

• Resolución lineal reduce el espacio de búsqueda.

• Resolución SLD es lineal y (un poco) más eficiente, preservando completitud, pero solo se aplica con Clausulas Horn.

Correctitud

Para clausulas de Horn es correcto, esto es, si $D_1, ..., D_n$ son cláusulas de definición y G una cláusula objetivo:

Teorema: Si $D_1, ..., D_n, G$ es insatisfactible, existe una refutación SLD.

Un secuencia de pasos de resolución SLD para un conjunto de cláusulas de Horn H es una secuencia

$$< N_0, N_1, \ldots, N_p >$$

de cláusulas objetivo que satisfacen las siguientes dos condiciones:

1. $N_0 \in H$ (N_0 es la cláusula objetivo de H).
2. *sigue en transparencia siguiente*.

2. para todo N_i en la secuencia, $0 < i < p$, si N_i es

$$\{\neg A_1, \ldots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \ldots, \neg A_n\}$$

entonces hay alguna cláusula de definición C_i de la forma $\{A, \neg B_1, \ldots, \neg B_m\}$ en H, tal que A_k y A son unificables con MGU S, y N_{i+1} es $\{S(\neg A_1, \ldots, \neg A_{k-1}, \neg B_1, \ldots, \neg B_m, \neg A_{k+1}, \ldots, \neg A_n)\}$.

Resolución SLD

Un caso particular de la resolución general.

- Cláusulas de Horn con exactamente una cláusula objetivo.
- Resolvemos la cláusula objetivo con una cláusula de definición.
- Eso nos da otra cláusula objetivo.
- Repetimos el proceso con esta nueva cláusula...
- Hasta llegar a la cláusula vacía.
- Si se busca un resultado, computamos la sustitución respuesta componiendo todas las sustituciones que fuimos realizando.

$$\frac{\overbrace{\{R, \neg B_1, \ldots, \neg B_n\}}^{\text{definición}} \qquad \overbrace{\{\neg A_1, \ldots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \ldots, \neg A_m\}}^{\text{objetivo}}}{\underbrace{S(\{\neg A_1, \ldots, \neg A_{k-1}, \neg B_1, \ldots, \neg B_n, \neg A_{k+1}, \ldots, \neg A_m\})}_{\text{nuevo objetivo}}}$$

donde S es el MGU de $\{R \stackrel{?}{=} A_k\}$.