PLP

Resumen final

UBA (Exactas)

Profesor.: Hernan

Resumen hecho por.: Chiara Tarzia

1. Programación funcional: Haskell

- **Transparencia referencial:** El valor de una expresion depende solo de los elementos que la constituyen.

  Implica:

  - Posibilidad de demostrar propiedades usando las propiedades de las subexpresiones y metodos de deduccion logica.
  - Facilita modularidad

- **Tipado:**

  - Un tipo tiene operaciones asociadas.
  - Toda expresión bien-formada tiene un tipo.

- **Polimorfismo parametrico:** `it :: a -> a`

- **Alto orden:** las funciones pueden ser pasadas como parametros y ser el resultado de evaluar una expresion.

- **Currificacion:** Mecanismo que permite reemplazar argumentos estructurados por una secuencia de argumentos "simples". Permite evaluacion parcial.

  - **Curry:**

    ```
    curry :: ((a,b) -> c) -> (a -> (b -> c))
    curry f x y = f (x, y)
    ```

  - **Uncurry:**

    ```
    uncurry :: (a -> b -> c) -> ((a, b) -> c)
    uncurry f (x, y) = f x y
    ```

- **Evaluacion lazy:**

- **Reduccion:** Se reemplaza un redex por otra utilizando las ecuaciones orientadas. Un redex (reducible expresion) es una sub-expresión que puede reescribirse.
- **Orden normal/Lazy:** Se selecciona el redex más externo para el que se pueda conocer que ecuación del programa utilizar.

  En general: Primero las funciones más externas y luego los argumentos (sólo si se necesitan).
- La **no terminacion** no siempre es un problema porque la evaluacion es **no estricta**.

- **Map:**

```
map ::(a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs)= (f x):(map f xs)
```

- **Filter:**

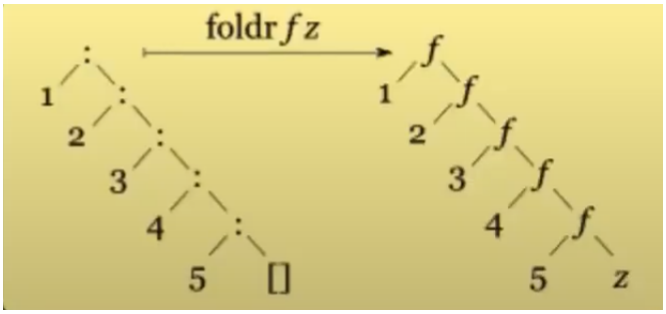
```
filter ::(a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)= if (p x) then x:(filter p xs)
                          else (filter p xs)
```

- **Recursion estructural:** permite acceder a los argumentos no recursivos de los constructores, y a los resultados de la recursión para las subestructuras.

```
g :: [a] -> b
g [] = z
g (x:xs) = f x (g xs)
```

- **foldr:** `g == foldr f z`

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



- *Ejemplos:*

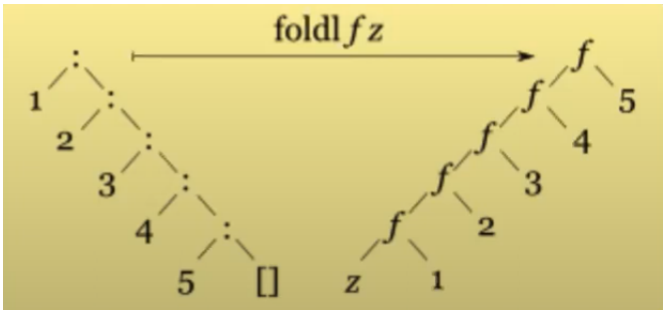
```
sumaL = foldr (+) 0
concat = foldr (++) []
reverse = foldr ((flip (++)) . (:[])) []
```

- *Map y filter:*

```
map f = foldr ((:) . f ) []
filter p = foldr (\x xs -> if p x then x:xs else xs) []
```

- **foldl:**

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z [] = z
foldl f z (x:xs)= foldl f (f z x) xs
```



- **Recursion primitiva:** como la estructural, pero además permite acceder a las subestructuras.

```
g :: [a] -> b
g [] = z
g (x:xs) = f x xs (g xs)
```

- **Recursion global:** como la primitiva, pero además permite acceder a los resultados de las recursiones anteriores. Es equivalente a `foldr`.

```
g == recr z f
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z _ [] = z
recr z f (x:xs)= f x xs (recr z f xs)
```

`dropWhile` *en terminos de* `recr`

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p = recr [] (\x xs res -> if p x then res else x:xs)
```

`foldr` *en terminos de* `recr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z = recr z (\x xs res -> f x res)
```

`recr` *en terminos de* `foldr`:

```
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z f l = snd (foldr g ([],z) l)
    where g a (tail, acum) = (a:tail, f a tail acum)
```

- **foldl vs foldr**

  Si tenemos una lista de n valores `[x1, x2,... xn ]`, una funcion f y semilla z.

foldr	**foldl**
asociacion a derecha `f x1 (f x2... (f xn z) ... )`	asociacion a izquierda `f ( ... (f (f z x1) x2) ...) xn`
recursivo hacia el argumento cada iteracion aplica f al siguiente valor y el resultado de foldear el resto de la lista	tail recursive itera la lista, produce el valor despues
forwards `foldr (:) []` devuelve la lista	backwards `foldl (flip (:)) []` revierte la lista

  - *En listas infinitas:*

    `foldr` *puede* terminar en listas infinitas, porque puede convertir una lista infinita en una estructura de datos lazy. Si toma una funcion que usa ambos argumentos inmediatamente (por ej `(+)`) no termina.

    `foldl` nunca termina con listas infinitas, porque como es backwards, cada aplicacion de f se agrega a la parte de afuera del resultado. Entonces, para computar cualquier parte del resultado Haskell tiene que iterar por la lista completa.

  - *foldl en terminos de foldr:*

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f b as =
        foldr (\a g x -> g (f x a)) id as b
```

Aplicado a una lista con elems:

```
foldl (flip (:)) [0] [1,2] = foldr (\a g x -> g ((flip (:)) x a)) id [1,2] [0]
    = (\a g x -> g ((flip (:)) x a)) 1 (foldr (\a g x -> g ((flip (:)) x a)) id [2]) [0]
    = (\g x -> g ((flip (:)) x 1)) (foldr (\a g x -> g ((flip (:)) x a)) id [2]) [0]
    = (\g x -> g ((flip (:)) x 1)) ((\a g x -> g ((flip (:)) x a)) 2 (foldr (\a g x -> g ((flip (:)) x a)) id [])) [0]
    = (\g x -> g ((flip (:)) x 1)) ((\g x -> g ((flip (:)) x 2)) (foldr (\a g x -> g ((flip (:)) x a)) id [])) [0]
    = (\g x -> g ((flip (:)) x 1)) ((\g x -> g ((flip (:)) x 2)) id) [0]
    = (\x -> ((\g x -> g ((flip (:)) x 2)) id) ((flip (:)) x 1)) [0]
    = (((\g x -> g ((flip (:)) x 2)) id) ((flip (:)) [0] 1))
    = (\g x -> g ((flip (:)) x 2)) id [1,0]
    = (\x -> id ((flip (:)) x 2)) [1,0]
    = id ((flip (:)) [1,0] 2)
    = ((flip (:)) [1,0] 2)
    = [2,1,0]
```

- *foldr en terminos de foldl:* Solo valido para listas finitas.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b as =
        foldl (\g a x -> g (f a x)) id as b
```

Aplicado a una lista con elems:

```
foldr (:) [3] [1,2] = foldl (\g a x -> g ((:) a x)) id [1,2] [3]
    = foldl (\g a x -> g ((:) a x)) ((\g a x -> g ((:) a x)) id 1) [2] [3]
    = foldl (\g a x -> g ((:) a x)) ((\g a x -> g ((:) a x)) ((\g a x -> g ((:) a x)) id 1) 2) [] [3]
    = ((\g a x -> g ((:) a x)) ((\g a x -> g ((:) a x)) id 1) 2) [3]
    = ((\a x -> ((\g a x -> g ((:) a x)) id 1) ((:) a x)) 2) [3]
    = (\x -> ((\g a x -> g ((:) a x)) id 1) ((:) 2 x)) [3]
    = ((\g a x -> g ((:) a x)) id 1) ((:) 2 [3])
    = ((\a x -> id ((:) a x)) 1) ((:) 2 [3])
    = (\x -> id ((:) 1 x)) ((:) 2 [3])
    = (id ((:) 1 ((:) 2 [3])))
    = ((:) 1 ((:) 2 [3]))
    = ((:) 1 [2,3])
    = [1,2,3]
```

- **foldr en otros tipos de datos algebraicos:** la firma del fold depende de los constructores.

  Ejemplo arboles:

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)

Hoja :: a -> Arbol a
Nodo :: a -> Arbol a -> Arbol a -> Arbol a

foldA :: (a -> b) -> (a -> b -> b -> b) -> Arbol a -> b
foldA f g (Hoja x) = f x
foldA f g (Nodo x izq der) = g x (foldA f g izq) (foldA f g der)

idArbol == foldA Hoja Nodo
```

2. Calculo Lambda Tipado

- **Sistema de tipado:** Sistema formal de deduccion (o derivacion) que utiliza axiomas y reglas de inferencia para caracterizar un subconjunto de los terminos llamados tipados.

- **Reglas de inferencia:**
  - *Axiomas de tipado* establecen que ciertos juicios de tipado son derivables.
  - *Reglas de tipado* establecen que ciertos juicios de tipado son derivables siempre y cuando ciertos otros lo sean.
- **Juicio de tipado:** Expresion de la forma $\Gamma \rhd M : \sigma$
- **Contexto de tipado:** conjunto de pares $x_i : \sigma_i$, anotado $\{x_1 : \sigma_1, ..., x_n : \sigma_n\}$ donde los $\{x_i\}_{i \in 1..n}$ son distintos. Usamos letras $\Gamma, \Delta, ...$ para contextos de tipado.
- $\Gamma \rhd M : \sigma$ es **derivable** si puede derivarse usando axiomas y reglas de tipado.
- M es **tipable** si $\Gamma \rhd M : \sigma$ puede derivarse para algun Γ y σ.
- **Unicidad de tipos:** Si $\Gamma \rhd M : \sigma$ y $\Gamma \rhd M : \tau$ son derivables, entonces $\sigma = \tau$.
- **Weakening+Strengthening:** Si $\Gamma \rhd M : \sigma$ es derivable y $\Gamma \cap \Gamma'$ contiene todas las *variables libres* de M, entonces $\Gamma' \rhd M : \sigma$.
- **Variables libres (FV):** x si no se encuentra bajo el alcance de algun λx.

2.1. Semantica Operacional

- Consiste en:
  - interpretar **terminos como estados** de una maquina abstracta
  - definir una **funcion de transicion** que indica, dado un estado, cual es el siguiente estado.
- **Significado** de un termino M: el estado final que alcanza la maquina al comenzar con M como estado inicial.
- Formas de definirla:
  - **Small-step:** la funcion de transicion describe un paso de computacion.
  - **Big-step:** la funcion de transicion, en un paso, evalua el termino a su resultado (un valor).
- **Juicios de evaluacion ($M \to N$):**
  - *Axiomas de evaluacion:* establecen que ciertos juicios de evaluacion son derivables.
  - *Reglas de evaluacion:* establecen que ciertos juicios de evaluacion son derivables siempre y cuando ciertos otros lo sean.
- **Valores:** Los posibles resultados de evaluacion de terminos bien-tipados y cerrados.
- **Determinismo del juicio de evaluacion en un paso:** Si $M \to M'$ y $M \to M''$, entonces $M' = M''$.
- **Forma normal:** un termino que no puede evaluarse mas.

- Todo valor esta en forma normal.
- **Juicio de evaluacion en muchos pasos $\twoheadrightarrow$:** clausura reflexiva, transitiva de $\rightarrow$.
- **Unicidad de formas normales:** Si $M \twoheadrightarrow U$ y $M \twoheadrightarrow V$ con U, V formas normales, entonces $U = V$.
- **Terminacion:** Para todo M existe una forma normal N tal que $M \twoheadrightarrow N$.
- **Sustitucion ($M\{x \leftarrow N\}$):** Sustituir las ocurrencias libres de x en el termino M por el termino N. Le da semantica a la aplicacion de funciones.
- **α-equivalencia:** Terminos que difieren solo en el nombre de sus variables.
- **Estado de error:** no es un valor pero la evaluacion esta trabada. El objetivo de un sistema de tipos es garantizar la ausencia de estados de error.
- **Correccion = Progreso + Preservacion:** Todo termino bien tipado termina.

  Si un termino cerrado esta bien tipado, entonces evalua a un valor.

  - **Progreso:** Si M es cerrado y bien tipado entonces
    1. M es un valor
    2. o existe M' tal que $M \rightarrow M'$.

    La evaluacion no puede trabarse para terminos cerrados, bien tipados que no son valores
  - **Preservacion:** Si $\Gamma \rhd M : \sigma$ y $M \rightarrow N$ entonces $\Gamma \rhd N : \sigma$

    La evaluacion preserva tipos

2.1.1. Registros

Sea $\mathcal{L}$ un conjunto de **etiquetas**

$$\lambda x : Nat.\lambda y : Bool.\{edad = x, esMujer = y\}$$

- El registro $\{l_i : M_i^{i \in 1..n}\}$ evalua a $\{l_i : V_i^{i \in 1..n}\}$ donde V_i es el valor al que evalua $M_i, i \in 1..n$
- $M.l$: evaluar M hasta que arroje $\{l_i : V_i^{i \in 1..n}\}$, luego proyectar el campo correspondiente

2.1.2. Unit

- Unit es un tipo unitario y el unico valor posible de una expresion de ese tipo es unit.
- Cumple rol similar a void
- Su utilidad principal es en lenguajes con efectos laterales
- En estos lenguajes es util poder evaluar varias expresiones en secuencia
- La evaluacion de $M_1; M_2$ consiste en primero evaluar M_1 y luego M_2.

2.1.3. Let

- $let\ x : \sigma = M\ in\ N$: evaluar M a un valor V, ligar x a V y evaluar N
- Mejora la legibilidad
- La extension con let no implica agregar nuevos tipos
- En una expresion como $let\ x :\ Nat = 2\ in\ N$
  - x es una variable declarada con valor 2.
  - El valor de x permanece inalterado a lo largo de la evaluacion de M.
  - En este sentido x es inmutable: no existe una operacion de asignacion.
- En programacion imperativa pasa todo lo contrario.
- Todas las variables son mutables.
- Vamos a extender Calculo Lambda Tipado con **variables mutables**:
  - **Alocacion:** $ref\ M$ genera una referencia fresca cuyo contenido es el valor de M.
  - **Derreferenciacion:** $!x$ sigue la referencia x y retorna su contenido
  - **Asignacion:** $x := M$ almacena en la referencia x el valor de M. Interesa por su efecto y no su valor

2.1.4. Expresiones de tipos (Ref)

- **Comando:** Expresion que se evalua para causar un efecto; definimos a unit como su valor.

  Un lenguaje funcional puro es uno en el que las expresiones son puras en el sentido de carecer de efectos
- $Ref\ \sigma$ es el tipo de las referencias a valores de tipo σ.
- **Referencia:** Es una abstraccion de una porcion de memoria que se encuentra en uso.
- Usamos **direcciones (simbolicas) o "locations"** $l, l_i \in \mathcal{L}$ para representar referencias.
- **Memoria (o "store")**: funcion parcial de direcciones a valores.
- Usamos letras μ, μ' para referirnos a stores.
- **Notacion:**
  - $\mu[l \mapsto V]$ es el store resultante de pisar $\mu(l)$ con V.
  - $\mu \oplus (l \mapsto V)$ es el store extendido resultante de ampliar μ con una nueva asociacion $l \mapsto V$ (asumimos $l \notin Dom(\mu)$).
- Los juicios de evaluacion toman la forma: $M|\mu \to M'|\mu'$

- **Nuevo juicio de tipado:** Al incluir referencias, dependemos de los valores que se almacenan en las direcciones para hacer el juicio de tipado $\Gamma \rhd I : \sigma$. Necesitamos el "contexto de tipado" para direcciones:

  Σ funcion parcial de direcciones en tipos

  $$\Gamma|\Sigma \rhd M : \sigma$$

- **Correccion = Progreso + Preservacion (reformulado):**

  - **Progreso:** Si M es cerrado y bien tipado (i.e. $\emptyset|\Sigma \rhd M : \sigma$ para algun Σ, σ) entonces
    1. M es un valor
    2. o bien para cualquier store μ tal que $\emptyset|\Sigma \rhd \mu$, existe M' y μ' tal que $M|\mu \to M'|\mu'$
  - **Preservacion:**

    Si
    - $\Gamma|\Sigma \rhd M : \sigma$
    - $M|\mu \to N|\mu'$
    - $\Gamma|\Sigma \rhd \mu$

    Implica que existe $\Sigma' \supseteq \Sigma$ tal que
    - $\Gamma|\Sigma' \rhd N : \sigma$
    - $\Gamma|\Sigma' \rhd \mu'$

2.1.5. Fix

Una manera de escribir ecuaciones recursivas.

$$\text{fix } (\lambda x : \sigma.M) \to M\{x \leftarrow \text{fix } (\lambda x : \sigma.M)\}$$

2.1.6. Letrec

Una construccion alternativa para definir funciones recursivas

$$\text{letrec } f : \sigma \to \sigma = \lambda x : \sigma.M \text{ in } N$$

Y puede escribirse en terminos de fix:

$$\text{let } f = \text{fix } (\lambda f : \sigma \to \sigma.\lambda x : \sigma.M) \text{ in } N$$

3. Inferencia de Tipos

Problema que consiste en transformar terminos sin info de tipos o info parcial en terminos tipables. Para esto se infiere la info de tipos faltante.

- **Erase:** Funcion que dado un termino de LC elimina las anotaciones de tipos de las abstracciones.

  $ERASE(.) : \wedge_\tau \to \wedge$

- **Inferencia de tipos:** Dado un termino U sin anotaciones de tipo, hallar un termino estandar M tal que

  1. $\Gamma \rhd M : \sigma$ para algun Γ, σ, y
  2. $ERASE(M) = U$

- **Chequeo de tipos:** Dado un termino estandar M determinar si existe Γ y σ tales que $\Gamma \rhd M : \sigma$ es derivable.

  - Mas facil que inferencia.
  - Dados Γ y σ, determinar si $\Gamma \rhd M : \sigma$ es derivable.

- **Sustitucion de tipos:** Funcion que mapea variables de tipo en expresiones de tipo. $S : \mathcal{V} \to \mathcal{T}$

  - Solo nos interesan las S tales que $\{t \in \mathcal{V} \mid St \neq t\}$ es finito
  - Se aplica a expresiones de tipos σ, terminos M y contextos de tipado Γ
  - *Soporte de S:* $\{t \mid St \neq t\}$ Las variables que S afecta.
  - *Sustitucion identidad (Id):* Sustitucion con soporte $\emptyset$.

- **Instancia de un juicio de tipado:** Un juicio de tipado $\Gamma' \rhd M' : \sigma'$ es una instancia de $\Gamma \rhd M : \sigma$ si existe una sustitucion de tipos S tal que

$$\Gamma' \supseteq S\Gamma,\ M' = SM \text{ y } \sigma' = S\sigma$$

- Si $\Gamma \rhd M : \sigma$ es derivable, entonces cualquier instancia del mismo tambien lo es.

- **Funcion de inferencia:** $\mathbb{W}(.)$ dado un termino U sin anotaciones verifica

  - **Correccion:** $\mathbb{W}(U) = \Gamma \rhd M : \sigma$ implica
    - $ERASE(M) = U$ y
    - $\Gamma \rhd M : \sigma$ es derivable

    Cosas que caracterizan a una solucion correcta.

  - **Completitud:** Si $\Gamma \rhd M : \sigma$ es derivable y $ERASE(M) = U$, entonces
    - $\mathbb{W}(U)$ tiene exito y
    - Produce un juicio $\Gamma' \rhd M' : \sigma'$ tal que $\Gamma \rhd M : \sigma$ es instancia del mismo. ($\mathbb{W}(.)$ computa un *tipo principal*).

    Si existe el M, el algoritmo tiene que devolverlo o una solucion mas general.

- **Unificacion:** Determinar si existe una sustitucion S tal que dos expresiones de tipos σ, τ son unificables (ie. $S\sigma = S\tau$)

- **Composicion de sustituciones:** $(S \circ T)(\sigma) = S(T\sigma)$

- **Preorden:** Una sustitucion S es **mas general** que T si existe U tal que $T = U \circ S$

- **Ecuacion de unificacion:** $\sigma_1 \doteq \sigma_2$.
- Una sustitucion S es una **solucion** de un conjunto de ecuaciones de unificacion $\{\sigma_1 \doteq \sigma'_1, ..., \sigma_n \doteq \sigma'_n\}$ si $S\sigma_1 = S\sigma'_1, ..., S\sigma_n = S\sigma'_n$.
- **Unificador Mas General (MGU):** una sustitucion S es un MGU de $\{\sigma_1 \doteq \sigma'_1, ..., \sigma_n \doteq \sigma'_n\}$ si
  1. es solucion de $\{\sigma_1 \doteq \sigma'_1, ..., \sigma_n \doteq \sigma'_n\}$
  2. es mas general que cualquier otra solucion de $\{\sigma_1 \doteq \sigma'_1, ..., \sigma_n \doteq \sigma'_n\}$
- Si $\{\sigma_1 \doteq \sigma'_1, ..., \sigma_n \doteq \sigma'_n\}$ tiene solucion, existe un MGU y ademas es unico salvo renombre de variables.

3.1. Algoritmo de Martinelli-Montanari

Algoritmo no deterministico. Consiste en **reglas de simplificacion** que simplifican conjuntos de pares de tipos a unificar (goals).

$$G_0 \mapsto G_1 \mapsto ... \mapsto G_n$$

Si termina en goal vacio, fue exitosa. Si termina en falla, fue fallida. Tambien puede llevar a una sustitucion que es una solucion parcial.

$$G_0 \mapsto G_1 \mapsto_{S_1} G_2 \mapsto ... \mapsto_{S_k} G_n$$

Si fue exitosa, el MGU es $S_k \circ ... \circ S_1$.
Reglas:

1. **Descomposicion**
   $\{\sigma_1 \to \sigma_2 \doteq \tau_1 \to \tau_2\} \cup G \mapsto \{\sigma_1 \doteq \tau_1, \sigma_2 \doteq \tau_2\} \cup G$
2. **Eliminacion de par trivial**
   $\{s \doteq s\} \cup G \mapsto G$
3. **Swap:** Si σ no es una variable.
   $\{\sigma \doteq s\} \cup G \mapsto \{s \doteq \sigma\} \cup G$
4. **Eliminacion de una variable:** si $s \notin FV(\sigma)$
   $\{s \doteq \sigma\} \cup G \mapsto_{\{\sigma/s\}} \{\sigma/s\}G$
5. **Colision**
   $\{\sigma \doteq \tau\} \cup G \mapsto$ `falla`, con $(\sigma, \tau) \in T \cup T^{-1}$ y
   $T = \{(Bool, Nat), (Nat, \sigma_1 \to \sigma_2), (Bool, \sigma_1 \to \sigma_2)\}$
6. **Occur Check:** si $s \neq \sigma$ y $s \in FV(\sigma)$
   $\{s \doteq \sigma\} \cup F \mapsto$ `falla`

Propiedades:

- Siempre termina
- Sea G un conjunto de pares:
  - si G tiene un unificador, termina exitosamente y retorna un MGU.
  - si G no tiene unificador, termina con `falla`.

3.2. Subtipado

- **Principio de sustitutividad ($\sigma <: \tau$):** En todo contexto donde se espera una expresion de tipo τ , puede utilizarse una de tipo σ en su lugar sin que ello genere un error.

  Esto se refleja en subsumption:

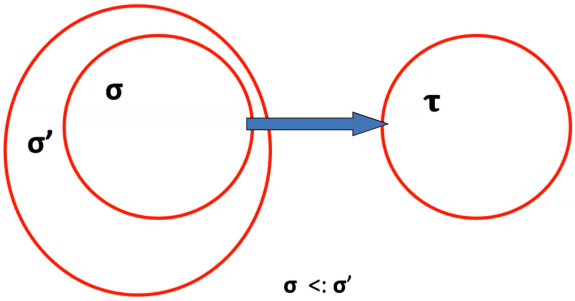
$$\frac{\Gamma \rhd M : \sigma \quad \sigma <: \tau}{\Gamma \rhd M : \tau} \text{ (T-Subs)}$$

  Ejemplos: $Nat <: Float$, $Int <: Float$, $Bool <: Nat$

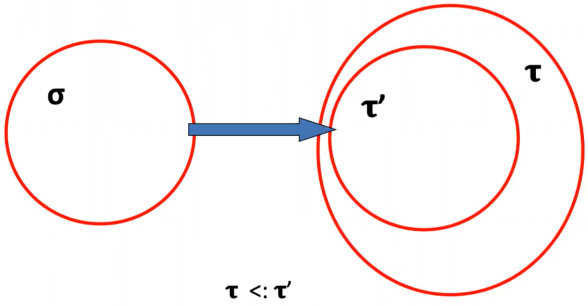
- **Subtipado a lo ancho:**
  - `{nombre: String, edad: Int}` $<:$ `{nombre: String}`
  - $\sigma <:$ `{}`
- **Subtipado de funciones:** $<:$ se da vuelta para el tipo del argumento de la funcion (**contravariante**) pero **no** para el tipo del resultado (**variante**).

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{ (S-Func)}$$

  Una funcion puede reemplazar a otra si requiere algo menos particular.



  Una funcion puede reemplazar a otra si devuelve algo mas particular.



- **Subtipado de Ref:** es **invariante**, solo se comparan referencias de tipos equivalentes.

$$\frac{\sigma <: \tau \qquad \tau <: \sigma}{Ref\sigma <: Ref\tau}$$

- **Chequeo de tipos con subtipado:**
  - Las reglas de tipado sin subtipado son dirigidas por sintaxis, por lo que implementar un algoritmo es inmediato.
  - Con subsumption ya no son dirigidas por sintaxis, y no es evidente como implementar un algoritmo.
  - Podemos cablear subsumption a las demas reglas (solo afecta T-App) e implementar $\sigma <: \tau$.
  - Al agregar las reglas de subtipado solo nos causa problemas S-Refl y S-Trans.
  - Reemplazamos S-Refl con los axiomas triviales de tipos.
  - S-Trans se puede demostrar, no hace falta tener una regla explicita

4. Programacion Orientada a Objetos

- **Method dispatch:** Asociacion entre el mensaje y el metodo a ejecutar. El tipo se define segun cuando se hace:
  - *Method dispatch estatico:* en tiempo de compilacion. Uno de los pilares de POO.
  - *Method dispatch dinamico:* en tiempo de ejecucion. Se usa por eficiencia y porque es requerido en algunas situaciones, como `super`.
- **Corrientes:**
  - **Clasificacion:**
    - Clases modelan *conceptos abstractos* del domminio del problema a resolver.
    - Definen el comportamiento y la forma de un conjunto de objetos (sus instancias).
    - Todo objeto es instancia de alguna clase
    - *Componentes:* Nombre, definicion de variables de instancia, metodos de instancia (nombre, parametros formales, cuerpo).
    - *Self:* Pseudo variable que, durante la evaluacion de un metodo, referencia al receptor del mensaje que activo dicha evaluacion.
    - *Jerarquia de clases:* Una clase puede heredar de o extender una clase preexistente (superclase). La herencia puede ser simple o multiple (complica method dispatch).
    - **Super:** Pseudovariable que referencia al objeto que recibe el mensaje.
  - **Prototipado:**
    - Construye instancias concretas que se interpretan como representantes canonicos de instancias (**prototipos**)

- Otras instancias se generan por clonacion (copia shallow)
- Los clones se pueden cambiar
- Herencia a traves de prototipos

4.1. Javascript

- Lenguaje OO basado en objetos (**prototipos**) imperativo.
- **Tipado:** Dinamico (tiempo de ejecucion), debil (conversion implicita de tipos)
- **Objetos:**
  - *Literal*

```
let o = { a : 1 , b : function ( n ) { return 1 + n }}
// o -> Object { a : 1 , b : b () }
```

  - *Notacion punto*

```
o.a // 1
o.b (1) // 2
```

  - *Parametro self implicito (`this`)*

```
let o = { a : 1 ,
        b : function ( n ) { return this . a + n }}
```

  - *Redefinicion (agregado)*

```
o.b = function () { return this . a }
o.c = true
// o -> Object { a : 1 , b : b () , c : true }
```

  - *Eliminacion de propiedades*

```
delete o.a
// o -> Object { b : b () , c : true }
```

  - *Asignacion por referencia*

```
let o = { b : b () , c : true };
let p = o ; // p y o referencian al mismo objeto
p.d = 1; // o.d == 1
```

  - *Extraccion de metodos*

```
let o = { a : function ( n ) { return n +1;}};
let f = o . a ;
f (0) ; // 1
```

- **Prototipos y herencias:**
  - Los objetos tienen una propiedad privada (llamada `[[Prototype]]`), cuyo valor es `null` u otro objeto, que es su prototipo.
  - *Herencia de prototipo:* Al intentar acceder (en lectura) a un atributo o metodo inexistente en un objeto, el mismo se busca en su prototipo.

4.2. Calculo de Objetos No Tipado (ς calculo)

- Objetos como unica estructura computacional.
- Los **objetos** son una coleccion de **atributos** nombrados (**registros**).
- Todos los **atributos** son metodos que no utilizan el parametro self.
- Cada metodo tiene una unica variable ligada que representa a self (`this`) y un cuerpo que produce un resultado.
- Los objetos proveen dos **operaciones**:
  - envio de mensaje (invocacion de un metodo)
  - redefinicion de un metodo
- **Sintaxis:**

$$
\begin{array}{rcll}
o, b & :== & x & \text{variable} \\
 & | & [l_i = \varsigma(x_i) b_i^{i \in 1..n}] & \text{objeto} \\
 & | & o.l & \text{seleccion/envio de mensaje} \\
 & | & o.l \Leftarrow \varsigma(x) b & \text{redefinicion de metodo}
\end{array}
$$

- **Semantica operacional:** Reduccion big step
- ς es un ligador para el parametro `self` x_i en el cuerpo b_i de la expresion $\varsigma(x_i)b_i$
- Se puede codificar los terminos de LC (no tipado) en ς calculo.
- **Stateless traits:**
  - Un trait es una coleccion de ciertos metodos
  - (Stateless) Traits no especifican variables de estado ni acceden al estado.
  - Son una coleccion de *pre-metodos*: $\lambda(y)b$ (no usan self)
  - `t`$= [l_i = \lambda(y_i) b_i^{i \in 1..n}]$ es un trait.

- a partir de un trait podemos definir un constructor de objetos (cuando `t` es completo).

$$new \stackrel{\text{def}}{=} \lambda(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}]$$

$$\begin{array}{rcl} o & \stackrel{\text{def}}{=} & new(\mathtt{t}) \\ & \approx & [l_i = \varsigma(s)\mathtt{t}.l_i(s)^{i\in 1..n}] \\ & \approx & [l_i = \varsigma(s)b_i^{i\in 1..n}] \end{array}$$

- **Clases:** Una clase es un trait (completo) que ademas provee un metodo new.

$$\begin{array}{c} c \stackrel{\text{def}}{=} [new = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}], \\ l_i = \lambda(s)b_i^{i\in 1..n}] \end{array}$$

$$\begin{array}{rcl} o & \stackrel{\text{def}}{=} & \mathtt{c}.new \\ & \longrightarrow & [l_i = \varsigma(s)\mathtt{c}.l_i(s)^{i\in 1..n}] \\ & \approx & [l_i = \varsigma(s)b_i^{i\in 1..n}] \end{array}$$

5. Resolucion en Logica Proposicional

- **Paradigma Logico:**
  - Logica como lenguaje de programacion
  - Se especifican: *hechos/reglas de inferencia* y un *objetivo* (goal) a probar.
  - Un motor de inferencia trata de probar que el objetivo es consecuencia de los hechos y reglas.
  - **Declarativo:** Se especifican hechos, reglas y objetivo sin indicar como se obtiene este ultimo a partir de los primeros.
- **Prolog:**
  - Los programas se escriben en un subconjunto de la logica de primer orden.
  - Se basa en el **metodo de resolucion**
- **Demostracion por refutacion:** Probar que A es *valido* mostrando que $\neg A$ es *insatisfacible*.

  Hay varias tecnicas, por ejemplo, resolucion.

5.1. Resolucion

- Una manera de implementar demostracion automatica de teoremas.
- Tiene una unica regla de inferencia: **regla de resolucion**

$$\frac{\{A_1, ..., A_m, Q\} \quad \{B_1, ..., B_n, \neg Q\}}{\{A_1, ..., A_m, B_1, ..., B_n\}}$$

- Es conveniente asumir que las proposiciones estan en FNC.
- **Forma Normal Conjuntiva (FNC):**
  - **Literal:** variable proposicional P o $\neg P$.
  - Una prop A esta en **FNC** si es una conjuncion

    $$C_1 \wedge ... \wedge C_n$$

    donde cada **clausula** C_i es una disyuncion

    $$B_{i1} \vee ... \vee B_{in_i}$$

    y cada B_{ij} es un literal.
  - Para toda proposicion A puede hallarse una proposicion A' en FNC que es logicamente equivalente a A.
  - **Notacion conjuntista:** $\{\{P, Q\}, \{P, \neg Q\}\}$
- La clausula $\{A, B\}$ se llama **resolvente** de las clausulas $\{A, P\}$ y $\{B, \neg P\}$
- El resolvente de $\{P\}$ y $\{\neg P\}$ es la **clausula vacia** ($\square$).
- **Paso de resolucion:** Agregar a un conjunto S el resolvente C de dos clausulas C_1, C_2 que pertenecen a S (i.e. aplicar la regla de resolucion a S)
  - Asumimos que $C \notin S$
  - Preserva insatisfactibilidad.
- **Refutacion:** Conjunto de clausulas que contiene la clausula vacia ($\square$). El algoritmo de buscar una refutacion **siempre termina**.
- **Metodo de resolucion:** El metodo de resolucion trata de construir una secuencia de conjuntos de clausulas, obtenidas usando pasos de resolucion hasta llegar a una refutacion.
- **Terminacion:** Siempre termina porque refutacion siempre termina.
- **Correccion y Completitud:** Dado un conjunto finito S de clausulas,

  $$S \text{ es insatisfactible sii tiene una refutacion}$$

  Si es insatisfacible va a haber una refutacion, y si hay una refutacion la formula es insatisfacible.
- **Resolucion en logica proposicional:** Para probar que A es una tautologia:
  1. Calcular FNC B de $\neg A$.
  2. Pasamos B a forma clausal.
  3. Aplicamos metodo de resolucion.
  4. *Si hallamos refutacion:* $\neg A$ es insatisfacible $\Rightarrow$ A tautologia.
  5. *Si no hallamos refutacion:* $\neg A$ es satisfacible $\Rightarrow$ A no es tautologia.

  Teorema de Herbrand-Skolem-Godel: Cada paso de resolucion preserva satisfactibilidad.

6. Logica de Primer Orden

- **Teorema de Church:** No existe un algoritmo que pueda determinar si una formula de primer orden es valida.
- Resolucion para logica de primer orden es un procedimiento de **semi-decision**: si es insatisfactible encuentra una refutacion, si es satisfactible puede no detenerse.
- **Forma clausal:** Es una FNC, en notacion de conjuntos.

  Pasos:

  1. Eliminar implicacion (solo $\wedge, \vee, \neg, \forall, \exists$)
  2. Pasar a *forma normal negada*
  3. Pasar a *forma normal prenexa* (opcional)
  4. Pasar a *forma normal de Skolem*
  5. Pasar matriz a *forma normal conjuntiva*
  6. *Distribuir* cuantificadores universales
- **Forma normal negada (FNN):** Definicion inductiva:

  1. Para cada formula atomica A, A y $\neg A$ estan en FNN.
  2. Si $A, B \in FNN$, entonces $(A \vee B), (A \wedge B) \in FNN$.
  3. Si $A \in FNN$, entonces $\forall x.A, \exists x.A \in FNN$.

  Toda formula es logicamente equivalente a otra en FNN.
- **Forma normal prenexa:** Formula de la forma $Q_1 x_1 ... Q_n x_n.B$, $n > 0$, donde
  - B sin cuantificadores (llamada **matriz**)
  - $x_1, ..., x_n$ variables
  - $Q_i \in \{\forall, \exists\}$

  Toda formula rectificada A es logicamente equivalente a una formula B en forma prenexa.
- **Forma normal de Skolem:** El objetivo de la skolemizacion es eliminar los cuantificadores universales sin alterar la satisfactibilidad.

  Sea A una sentencia rectificada en FNN. La **forma normal de Skolem de A (SK(A))** es una formula sin existenciales ($\exists$) que se obtiene recursivamente asi:

  Sea A' cualquier subformula de A.
  - A' formula atomica o su negacion $\Rightarrow SK(A') = A'$
  - $A' = (B \star C)$ con $\star \in \{\vee, \wedge\} \Rightarrow SK(A') = (SK(B) \star SK(C))$
  - $A' = \forall x.B \Rightarrow SK(A') = \forall x.SK(B)$
  - $A' = \exists x.B$ y $\{x, y_1, ..., y_m\}$ las variables libres de B (ligadas en sentencia A)

1. $m > 0 \Rightarrow$ crear un nuevo *simbolo de funcion de Skolem*, f_x de aridad m y definir
$$SK(A') = SK(B\{x \leftarrow f_x(y_1, ..., y_m)\})$$
2. $m = 0 \Rightarrow$ crear una nueva *constante de Skolem* c_x y
$$SK(A') = SK(B\{x \leftarrow c_x\})$$

Nota: como A esta rectificada, cada f_x, c_x es unica.

Nota: Skolemizacion no es deterministica, es mejor hacerlo de afuera hacia adentro.

- **Forma normal conjuntiva (FNC):** Hasta ahora tenemos algo de la forma $\forall x_1 ... \forall x_n . B'$. Lo pasamos a FNC como si fuera una formula proposicional, obteniendo $\forall x_1 ... \forall x_n . B'$.

- **Distribuir cuantificadores universales:** Obtenemos una conjuncion de **clausulas**

$$\forall x_1 ... \forall x_n . C_1 \wedge ... \wedge \forall x_1 ... \forall x_n . C_m$$

donde cada C_i es una disyuncion de literales.

Se simplifica escribiendo $\{C_1, ..., C_m\}$

- **Regla de resolucion:**

$$\frac{\{B_1, ..., B_k, A_1, ..., A_m\} \quad \{\neg D_1, ..., \neg D_j, C_1, ..., C_n\}}{\sigma(\{A_1, ..., A_m, C_1, ..., C_n\})}$$

donde σ es el MGU de $\{B_1 \doteq B_2, ..., B_{k-1} \doteq B_k, B_k \doteq D_1, ..., D_{j-1} \doteq D_j\}$

  - Asumimos que las clausulas $\{B_1, ..., B_k, A_1, ..., A_m\}$ y $\{\neg D_1, ..., \neg D_j, C_1, ..., C_n\}$ no tienen variables en comun; caso contrario se renombran las variables.
  - Observar que $\sigma(B_1) = ... = \sigma(B_k) = \sigma(D_1) = ... = \sigma(D_j)$.
  - La clausula $\sigma(\{A_1, ..., A_m, C_1, ..., C_n\})$ se llama **resolvente** (de $\{B_1, ..., B_k, A_1, ..., A_m\}$ y $\{\neg D_1, ..., \neg D_j, C_1, ..., C_n\}$).

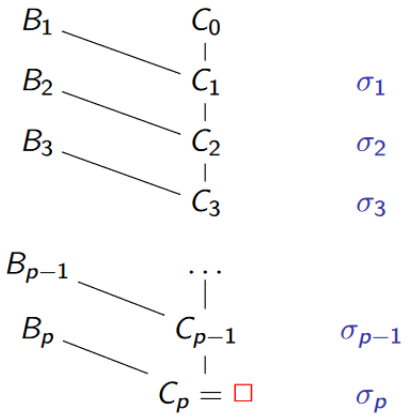
- **Resolucion en logica de primer orden:**

  - *Las siguientes nociones son analogas al caso proposicional:*
    - Clausula vacia
    - Paso de resolucion
    - Refutacion
    - Teorema de Herbrand-Skolem-Godel
  - *Diferencias con proposicional*
    - Regla de **resolucion binaria** es incompleta en LPO:
$$\frac{\{B, A_1, ..., A_m\} \quad \{\neg D, C_1, ..., C_n\}}{\sigma(\{A_1, ..., A_m, C_1, ..., C_n\})}$$
Se recupera la completitud agregando **factorizacion**:
$$\frac{\{B_1, ..., B_k, A_1, ..., A_m\}}{\sigma(\{B_1, A_1, ..., A_m\})}$$

7. Resolucion SLD

El metodo de resolucion general tiene un alto grado de no determinismo.
Se puede mejorar con **reglas de busqueda** (eleccion de clausulas) y **reglas de seleccion** (literales eliminados) para reducir el espacio de busqueda sin renunciar a la **completitud** del metodo.

- **Resolucion lineal:** Una secuencia de pasos de resolucion a partir de S es lineal si es de la forma:

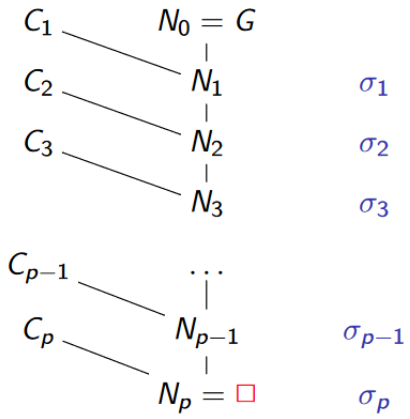


  donde C_0 y cada B_i es un elemento de S (o algun C_j con $j < i$).

  - Reduce el **espacio de busqueda** considerablemente
  - Preserva **completitud**
  - Es altamente **no-deterministico**

- **Clausulas de Horn:** Disyuncion de literales que tiene *a lo sumo* un literal positivo.
- **Clausula de definicion:** Disyuncion de literales que tiene *exactamente* un literal positivo.
- **Resolucion SLD:**
  - *Clausulas de entrada:* $S = P \cup \{G\}$
    - P: conjunto de clausulas de definicion, *programa o base de conocimientos*
    - G: clausula negativa, *goal, meta o clausula objetivo*
  - *Pasos de resolucion:* Secuencia $< N_0, N_1, ..., N_p >$ de clausulas negativas que satisfacen:
    1. N_0 es el goal G.
    2. Para todo N_i en la secuencia, $0 < i < p$, si N_i es
    $$\{\neg A_1, ..., \neg A_{k-1}, \neg A_k, \neg A_{k+1}, ..., \neg A_n\}$$
    entonces hay alguna *clausula de definicion* C_i de la forma $\{A, \neg B_1, ..., \neg B_m\}$ en P tal que A_k y A son unificables con MGU σ, y si
    - $m = 0$, entonces N_{i+1} es
    $\{\sigma(\neg A_1, ..., \neg A_{k-1}, \neg A_{k+1}, ..., \neg A_n)\}$
    - $m > 0$, entonces N_{i+1} es
    $\{\sigma(\neg A_1, ..., \neg A_{k-1}, \neg B_1, ..., \neg B_m, \neg A_{k+1}, ..., \neg A_n)\}$

- **Refutacion SLD:** Secuencia de pasos de resolucion SLD $< N_0, ..., N_p >$ tal que $N_p = \square$.



  - En cada paso, las clausulas $\{\neg A_1, ..., \neg A_{k-1}, \neg A_k, \neg A_{k+1}, ..., \neg A_n\}$ y $\{A, \neg B_1, ..., \neg B_m\}$ son resueltas.
  - Los atomos A_k y A son unificados con MGU σ_i
  - El literal A_k se llama **atomo seleccionado** de N_i.
  - **Sustitucion respuesta:** $\sigma_p \circ ... \circ \sigma_1$
  - **Correccion:** Si un conjunto de clausulas de Horn tiene una refutacion SLD, entonces es insatisfacible.
  - **Completitud:** Dado un conjunto de clausulas de Horn $P \cup \{G\}$, si $P \cup \{G\}$ es insatisfactible, existe una refutacion SLD cuya primera clausula es G.

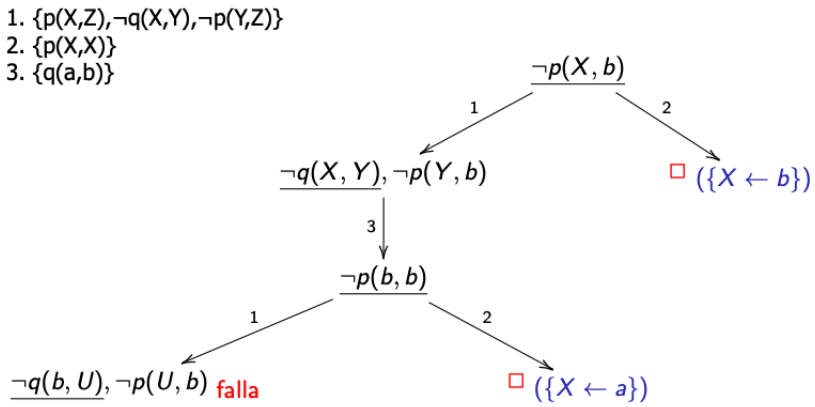
7.1. Resolucion SLD en Prolog

Prolog usa resolucion SLD con la siguiente **estrategia**:

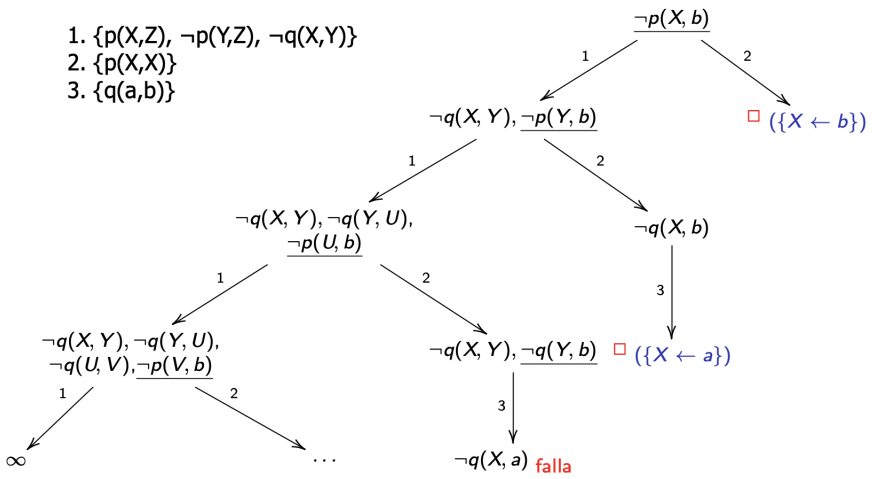
1. *Regla de busqueda:* Se seleccionan las clausulas de programa de arriba hacia abajo, en el orden en que fueron introducidas.

2. *Regla de seleccion:* seleccionar el atomo de mas a izquierda.

Cada estrategia determina un arbol de busqueda o **arbol SLD**.
Ejemplo arbol SLD con goal $\{\neg p(X, b)\}$



Ejemplo arbol SLD con goal $\{\neg p(X, b)\}$ pero regla de seleccion atomo mas de la derecha.



- **Notacion Prolog para programas logicos:** La resolucion SLD parte de un conjunto de clausulas $S = P \cup \{G\}$

  - P: clausulas de definicion: $\{B \vee \neg A_1 \vee ... \vee \neg A_n\}$, $\{B\}$
  - G es un goal: $\{\neg A_1, ..., \neg A_n\}$

  Entonces tenemos:

  $$B \vee \neg A_1 \vee ... \vee \neg A_n \iff \neg(A_1 \wedge ... \wedge A_n) \vee B$$
  $$\iff (A_1 \wedge ... \wedge A_n) \supset B$$

  Como consecuencia, las clausulas en P se escriben

  - $B :- A_1, ..., A_n.$ para $\{B \vee \neg A_1 \vee ... \vee \neg A_n\}$ (**reglas**)
  - $B.$ para B (**hechos**)
  - $? - C.$ (**goal**)

- **Correccion:** Si existe una refutacion SLD $P \cup \{G\}$ con la estrategia antes mencionada entonces es insatisfactible.

- **Completitud:** Si $P \cup \{G\}$ es insatisfactible, entonces una refutacion SLD con la estrategia antes mencionada a partir del mismo.

- **Busqueda de refutaciones SLD en Prolog:** Recorre el arbol SLD en profundidad ("depth-first search").

  Ventaja: Muy eficiente

  - *Pila* representa atomos del goal
  - *Push* del resolvente del atomo del tope de la pila con la clausula de definicion
  - *Pop* cuando el atomo del tope de la pila no unifica con ninguna clausula de definicion mas (luego el atomo que queda en el tope se unifica con la siguiente clausula de definicion).

  Desventaja: Puede que no encuentre una refutacion SLD aun si existe.

7.2. Sobre Prolog

7.2.1. Cut

- Predicado 0-ario, notado **!**.
- Solo tiene exito la primera vez que se lo invoca.
- Brinda un mecanismo de control que permite podar el arbol SLD
- Es de caracter **extra-logico** (i.e. no se corresponde con un predicado estandar de la logica)
- Se encuentra presente por cuestiones de eficiencia
- Debe usarse con cuidado, dado que puede podarse una rama de exito deseada
- Cuando se selecciona un cut, tiene exito inmediatamente
- Si, debido a backtracking, se vuelve a este cut, su efecto es el de hacer fallar el goal que le dio origen
- El goal que unifico con la cabeza de la clausula que contiene al corte y que hizo que esa clausula se "activara"
- El efecto obtenido es el de descartar soluciones (i.e. no dar mas soluciones) de
  1. otras clausulas del goal padre
  2. cualquier goal que ocurre a la izquierda del corte en la clausula que contiene el corte
  3. todos los objetivos intermedios que se ejecutaron durante la ejecucion de los goals precedentes

7.2.2. Negacion por falla/Negation as failure

- Se dice que un arbol SLD **falla finitamente** si es finito y no tiene ramas de exito
- Dado un programa P el **conjunto de falla finita** de P es

  $\{B \mid B$ atomo cerrado ('ground') y existe un arbol SLD que falla finitamente con B como raiz$\}$
- **Negation as failure:**

$$\frac{B \text{ atomo cerrado } B \text{ en conjunto de falla finita de } P}{\neg B}$$

- **Negacion por falla no es negacion logica**

```
animal(perro).                  not(G) :- G, !, fail.
animal(gato).                   not(G).
vegetal(X) :- not(animal(X)).
```

  - El goal `not(G)` **nunca** instancia variables de G.

- So G tiene exito, `fail` falla y descarta la sustitucion.
- Caso contrario, `not(G)` tiene exito inmediatamente (sin afectar G)

- En consecuencia `not(not(animal(X)))` **no** es equivalente a `animal(X)`
- Necesito que este instanciado lo que quiero negar.
- `not(algo(X))` si no esta instanciado `X` da false. Si algo lo liga primero puede dar otro resultado. Ejemplo:

```
firefighterCandidate(X) :-
        not( pyromaniac(X) ),
        punctual(X).
pyromaniac(attila).
punctual(jeanne_d_arc).
```

  - `firefighterCandidate(W)` da `false`.
  - Si invierto las clausulas `firefighterCandidate(W)` da `true`