Inferencia de Tipos

- Términos **sin** anotaciones de tipos: $U ::= x \mid \lambda x.U \mid U\ U \mid True \mid False \mid if\ U\ then\ U\ else\ U$
- Términos **con** anotaciones de tipos: $U ::= x \mid \lambda x : \tau.M \mid U\ U \mid True \mid False \mid if\ U\ then\ U\ else\ U$

Erase

erase es una función que recibe un término tipado y la respuesta es el mismo término pero sin tipos.
Ej.: $erase((\lambda x : Bool.x)\ True) = (\lambda x.x)\ True$

Algoritmo de Martelli-Montanari

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

$$\{C(\tau_1, \ldots, \tau_n) \stackrel{?}{=} C(\sigma_1, \ldots, \sigma_n)\} \cup E \xrightarrow{\texttt{Decompose}} \{\tau_1 \stackrel{?}{=} \sigma_1, \ldots, \tau_n \stackrel{?}{=} \sigma_n\} \cup E$$

$$\{\tau \stackrel{?}{=} Xn\} \cup E \xrightarrow{\texttt{Swap}} \{Xn \stackrel{?}{=} \tau\} \cup E \quad \text{si } \tau \text{ no es una incógnita}$$

$$\{Xn \stackrel{?}{=} \tau\} \cup E \xrightarrow[\{Xn := \tau\}]{\texttt{Elim}} E' = \{Xn := \tau\}(E) \quad \text{si } Xn \text{ no ocurre en } \tau$$

$$\{C(\tau_1, \ldots, \tau_n) \stackrel{?}{=} C'(\sigma_1, \ldots, \sigma_m)\} \cup E \xrightarrow{\texttt{Clash}} \texttt{falla} \quad \text{si } C \neq C'$$

$$\{Xn \stackrel{?}{=} \tau\} \cup E \xrightarrow{\texttt{Occurs-Check}} \texttt{falla} \quad \text{si } Xn \neq \tau \text{ y } Xn \text{ ocurre en } \tau$$

- El Algoritmo termina para cualquier problema de unificación E.
- Si E no tiene solución, el algoritmo llega a una falla.
- Si E tiene solución llega a $\emptyset$. Además, el unificador resultante es el más general posible, es decir, mgu(E).

Algoritmo $\mathcal{W}$

- $\mathbb{W}(x) \rightsquigarrow \{x : \mathrm{X}_k\} \vdash x : \mathrm{X}_k$, $\quad \mathrm{X}_k$ incógnita fresca
- $\mathbb{W}(0) \rightsquigarrow \emptyset \vdash 0 : Nat$
- $\mathbb{W}(true) \rightsquigarrow \emptyset \vdash true : Bool$
- $\mathbb{W}(false) \rightsquigarrow \emptyset \vdash false : Bool$
- $\mathbb{W}(succ(U)) \rightsquigarrow S(\Gamma) \vdash S(succ(M)) : Nat$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
  - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(pred(U)) \rightsquigarrow S(\Gamma) \vdash S(pred(M)) : Nat$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
  - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(iszero(U)) \rightsquigarrow S(\Gamma) \vdash S(iszero(M)) : Bool$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
  - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(if\ U\ then\ V\ else\ W) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \cup S(\Gamma_3) \vdash S(if\ M\ then\ P\ else\ Q) : S(\sigma)$ donde
  - $\mathbb{W}(U) = \Gamma_1 \vdash M : \rho$
  - $\mathbb{W}(V) = \Gamma_2 \vdash P : \sigma$
  - $\mathbb{W}(W) = \Gamma_3 \vdash Q : \tau$
  - $S = MGU\{\sigma \stackrel{?}{=} \tau, \rho \stackrel{?}{=} Bool\} \cup \{\sigma_1 \stackrel{?}{=} \sigma_2 \mid x : \sigma_1 \in \Gamma_i, x : \sigma_2 \in \Gamma_j, i, j \in \{1, 2, 3\}\}$
- $\mathbb{W}(\lambda x.U) \rightsquigarrow \Gamma' \vdash \lambda x : \tau'.M : \tau' \to \rho$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \rho$
  - $\tau' = \begin{cases} \alpha \text{ si } x : \alpha \in \Gamma \\ \mathrm{X}_k \text{ con } \mathrm{X}_k \text{ variable fresca en otro caso} \end{cases}$
  - $\Gamma' = \Gamma \ominus \{x\}$
- $\mathbb{W}(U\ V) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \vdash S(M\ N) : S(\mathrm{X}_k)$ donde
  - $\mathbb{W}(U) = \Gamma_1 \vdash M : \tau$
  - $\mathbb{W}(V) = \Gamma_2 \vdash N : \rho$
  - X_k variable fresca
  - $S = MGU\{\tau \stackrel{?}{=} \rho \to \mathrm{X}_k\} \cup \{\sigma_1 \stackrel{?}{=} \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\}$

$$\begin{array}{lcl} \tau & ::= & \ldots \mid \tau + \tau \\ M & ::= & \ldots \mid \mathsf{left}_\tau(M) \mid \mathsf{right}_\tau(M) \mid \mathsf{case}\, M\, \mathsf{of}\, \mathsf{left}(x) \rightsquigarrow M \,[\!]\, \mathsf{right}(y) \rightsquigarrow M \end{array}$$

$\mathbb{W}(\mathsf{left}(U)) \stackrel{def}{=} \Gamma \vdash \mathsf{left}_X(M) : \sigma + X$

donde:

- $\mathbb{W}(U) = \Gamma \vdash M : \sigma$
- X variable fresca.

$\mathbb{W}(\mathsf{right}(U)) \stackrel{def}{=} \Gamma \vdash \mathsf{right}_X(M) : X + \tau$
donde:

- $\mathbb{W}(U) = \Gamma \vdash M : \tau$
- X variable fresca.

$\mathbb{W}(\mathsf{case}\ U_1\ \mathsf{of}\ \mathsf{left}(x) \rightsquigarrow U_2 \,[\!]\, \mathsf{right}(y) \rightsquigarrow U_3) \stackrel{def}{=}$
$S\Gamma_1 \cup S\Gamma_{2'} \cup S\Gamma_{3'} \vdash S(\mathsf{case}\ M_1\ \mathsf{of}\ \mathsf{left}(x) \rightsquigarrow M_2 \,[\!]\, \mathsf{right}(y) \rightsquigarrow M_3) : S\tau$
donde:

- $\mathbb{W}(U_1) = \Gamma_1 \vdash M_1 : \tau_1$
- $\mathbb{W}(U_2) = \Gamma_2 \vdash M_2 : \tau_2$
- $\mathbb{W}(U_3) = \Gamma_3 \vdash M_3 : \tau_3$
- $\tau_x = \begin{cases} \alpha \text{ si } x : \alpha \in \Gamma_2 \\ \text{Variable fresca en otro caso} \end{cases}$
- $\tau_y = \begin{cases} \beta \text{ si } y : \beta \in \Gamma_3 \\ \text{Variable fresca en otro caso} \end{cases}$
- $\Gamma_{2'} = \Gamma_2 \ominus \{x\}$
- $\Gamma_{3'} = \Gamma_3 \ominus \{y\}$
- $S = \mathrm{mgu}\ (\{\tau_1 \stackrel{?}{=} \tau_x + \tau_y, \tau_2 \stackrel{?}{=} \tau_3\} \cup \{\rho \stackrel{?}{=} \sigma \mid z : \rho \in \Gamma_i \wedge z : \sigma \in \Gamma_j \wedge i, j \in \{1, 2', 3'\}\})$

$$\begin{array}{lcl} \tau & ::= & \ldots \mid [\tau] \\ M & ::= & \ldots \mid [\,]_\tau \mid M :: M \mid \mathsf{foldr}\, M\, \mathsf{base} \hookrightarrow M;\ \mathsf{rec}(h, r) \hookrightarrow M \end{array}$$

$\mathbb{W}([\,]) \stackrel{def}{=} \emptyset \vdash [\,]_X : [X]$ con X variable fresca

$\mathbb{W}(U :: V) \stackrel{def}{=} S\Gamma_1 \cup S\Gamma_2 \vdash S(M :: N) : S\tau$
donde:

- $\mathbb{W}(U) = \Gamma \vdash M : \sigma$
- $\mathbb{W}(V) = \Gamma \vdash N : \tau$
- $S = \mathrm{mgu}\ (\{\tau \stackrel{?}{=} [\sigma]\} \cup \{\rho \stackrel{?}{=} \phi \mid x : \rho \in \Gamma_1 \wedge x : \phi \in \Gamma_2\})$

$\mathbb{W}(\mathsf{foldr}\, U\, \mathsf{base} \hookrightarrow V;\ \mathsf{rec}(h, r) \hookrightarrow W) \stackrel{def}{=} S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_{3'} \vdash S(\mathsf{foldr}\, M\, \mathsf{base} \hookrightarrow N;\ \mathsf{rec}(h, r) \hookrightarrow O) : S\sigma_2$
donde:

- $\mathbb{W}(U) = \Gamma \vdash M : \sigma_1$
- $\mathbb{W}(V) = \Gamma \vdash N : \sigma_2$
- $\mathbb{W}(W) = \Gamma \vdash O : \sigma_3$
- $\Gamma_{3'} = \Gamma_3 \ominus \{h, r\}$
- $\tau_h = \begin{cases} \alpha \text{ si } h : \alpha \in \Gamma_3, \\ \text{variable fresca si no} \end{cases}$
- $\tau_r = \begin{cases} \beta \text{ si } r : \beta \in \Gamma_3, \\ \text{variable fresca si no} \end{cases}$
- $S = \mathrm{mgu}\ (\{\sigma_1 \stackrel{?}{=} [\tau_h], \sigma_2 \stackrel{?}{=} \sigma_3, \sigma_3 \stackrel{?}{=} \tau_r\} \cup \{\rho \stackrel{?}{=} \sigma \mid x : \rho \in \Gamma_i \wedge x : \sigma \in \Gamma_j \wedge i, j \in \{1, 2, 3'\}\})$

Lógica de Primer Orden

- Conjunto de Símbolos de Función $\mathcal{F} = \{f, g, h, \ldots\}$. Cada símbolo de función tiene una aridad asignada.
  - Aquellas que tienen aridad 0 son constantes.
- Conjunto de Símbolos de Predicado $\mathcal{P} = \{P, Q, R, \geq, \leq, \ldots\}$. Cada símbolo de predicado tiene una aridad asignada.
- Variables $\mathcal{X} = \{X, Y, Z, \ldots\}$
- Términos: $t ::= x(f(t_1, \ldots, t_n))$
- Fórmulas: $\sigma ::= P(t_1, \ldots, t_n) \mid \bot \mid \sigma \to \sigma \mid \sigma \wedge \sigma \mid \sigma \vee \sigma \mid \neg\sigma \mid \forall x.\sigma \mid \exists x.\sigma$
  - Si dos fórmulas son exactamente iguales pero difieren en el nombre de las variables de los cuantificadores, también se consideran iguales.

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \wedge \sigma}\ \wedge_i \qquad \frac{}{\Gamma, \tau \vdash \tau}\ \text{ax}$$

$$\frac{\Gamma, \tau \vdash \sigma}{\Gamma \vdash \tau \Rightarrow \sigma}\ \Rightarrow_i \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \tau \vee \sigma}\ \vee_{i_2} \qquad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \tau}\ \wedge_{e_1} \qquad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \sigma}\ \wedge_{e_2}$$

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

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

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

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \tau}\ \bot_e$$

Lógica intuicionista

Lógica clásica

Reglas clásicas

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

$$\frac{\Gamma \vdash \neg\neg\tau}{\Gamma \vdash \tau}\ \neg\neg_e$$

LPO

$$\frac{\Gamma \vdash \forall X.\, \sigma}{\Gamma \vdash \sigma\{X := t\}}\ \forall\text{E} \qquad \frac{\Gamma \vdash \sigma\{X := t\}}{\Gamma \vdash \exists X.\, \sigma}\ \exists\text{I}$$

$$\frac{\Gamma \vdash \sigma \quad X \notin \text{fv}(\Gamma)}{\Gamma \vdash \forall X.\, \sigma}\ \forall\text{I} \qquad \frac{\Gamma \vdash \exists X.\, \sigma \quad \Gamma, \sigma \vdash \tau \quad X \notin \text{fv}(\Gamma, \tau)}{\Gamma \vdash \tau}\ \exists\text{E}$$

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

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

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

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

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

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

Resolución

Es útil como técnica de demostración por refutación.
Importante: Un conjunto de claúsulas que tiene al menos una contradicción puede probar cualquier cosa.
Ej.: $C = \{\{\neg pago(smullyan)\}, \{pago(smullyan)\}, \{\neg espia(jefegob)\}\}$ En este caso, el objetivo $\neg\ espia(jefegob)$ ni siquiera hay que usarlo porque se cancelan las dos primeras y da vacío. Ojo, esto es solamente resolución, no resolución SLD porque sino comenzaríamos desde claúsula objetivo.

Resolución para Lógica de Primer Orden

Importante: Si un cuantificador tiene un mismo nombre de variable ligada renombrarla.

- 1. Deshacerse del conectivo $\implies$
  - $\sigma \implies \tau \to \neg\sigma \vee \tau$
- 2. Empujar el conectivo $\neg$ lo más posible hacia adentro, paso por paso afectando a cada término.
  - $\neg(\sigma \wedge \tau) \to \neg \vee \neg\tau$
  - $\neg(\sigma \vee \tau) \to \neg\sigma \wedge \neg\tau$
  - $\neg\neg\sigma \to \sigma$
  - $\neg\forall X.\sigma \to \exists X.\neg\sigma$
  - $\neg\exists X.\sigma \to \forall X.\neg\sigma$
- 3. Extraer los cuantificadores $\forall/\exists$ hacia afuera. Se asume que $X \notin fv(\tau)$
  - * $(\forall X.\sigma) \wedge \tau \to \forall X.(\sigma \wedge \tau)$
    * $(\forall X.\sigma) \vee \tau \to \forall X.(\sigma \vee \tau)$
    * $(\exists X.\sigma) \wedge \tau \to \exists X.(\sigma \wedge \tau)$
    * $(\exists X.\sigma) \vee \tau \to \exists X.(\sigma \vee \tau)$
  - * $\tau \wedge (\forall X.\sigma) \to \forall X.(\tau \wedge \sigma)$
    * $\tau \vee (\forall X.\sigma) \to \forall X.(\tau \vee \sigma)$
    * $\tau \wedge (\exists X.\sigma) \to \exists X.(\tau \wedge \sigma)$
    * $\tau \vee (\exists X.\sigma) \to \exists X.(\tau \vee \sigma)$
  - 4. Skolemización
    * En función de cuantificadores universales
      - $\forall X.\forall Y.\exists Z.P(Z) \to \forall X.\forall Y.P(f(X,Y))$
      - $\forall X.\exists Z.\forall Y.\exists D.P(D) \wedge Q(Z) \to \forall X.\forall Y.P(f(X,Y)) \wedge Q(g(X))$
    * $\exists X.\forall Y.\forall Z.P(X) \to \forall Y.\forall Z.P(c)$ c cte.
- 5. Distribuir los $\vee$
  - $\sigma \vee (\tau \wedge \rho) \to (\sigma \vee \tau) \wedge (\sigma \vee \rho)$
  - $(\sigma \wedge \tau) \vee \rho \to (\sigma \vee \rho) \wedge (\tau \vee \rho)$
- 6. Empujar los cuantificadores universales hacia adentro por cada $\wedge$.

Importante: 1 y 2 hacen a Forma Normal Negada. Añadiendo 3 tenemos Forma Normal Prenexa. Añadiendo 4 tenemos Forma Normal Skolem. Añadiendo 5 tenemos Forma Normal Conjuntiva y Añadiendo 6 tenemos Forma Clausal.

Fórmula Derivada

Una fórmula σ deriva de un conjunto de claúsulas sí y solo sí $\neg\sigma$ es insatisfactible ($\emptyset$) aplicando las claúsulas que tomamos como verdaderas.

- Representar en forma clausal la siguiente información referida a conjuntos, pertenencia (predicado Pert) e inclusión (predicado Inc).
  - i $\forall X.\forall Y.(\text{Inc}(X,Y) \Leftrightarrow \forall Z.(\text{Pert}(Z,X) \Rightarrow \text{Pert}(Z,Y)))$
    X está incluido en Y si y solo si cada elemento de X es un elemento de Y.
  - ii $\forall X.\neg\text{Pert}(X,\emptyset)$
    Ningún elemento pertenece al vacío.
- Usar resolución para probar que el vacío está incluido en todo conjunto.
- Indicar justificando si la prueba realizada es SLD (volveremos sobre esto más adelante).

i) $\forall X.\forall Y.(\text{Inc}(X,Y) \iff \forall Z.(\text{Pert}(Z,X) \Rightarrow \text{Pert}(Z,Y)))$

$\forall X.\forall Y((\text{Inc}(X,Y) \Rightarrow \forall Z.(\text{Pert}(Z,X) \Rightarrow \text{Pert}(Z,Y))) \wedge (\forall Z.(\text{Pert}(Z,X) \Rightarrow \text{Pert}(Z,Y)) \Rightarrow \text{Inc}(X,Y)))$

$\forall X.\forall Y.((\neg\text{Inc}(X,Y) \vee \forall Z.(\text{Pert}(Z,X) \Rightarrow \text{Pert}(Z,Y))) \wedge (\neg\forall Z.(\text{Pert}(Z,X) \Rightarrow \text{Pert}(Z,Y)) \vee \text{Inc}(X,Y)))$

$\forall X.\forall Y.((\neg\text{Inc}(X,Y) \vee \forall Z.(\neg\text{Pert}(Z,X) \vee \text{Pert}(Z,Y))) \wedge (\neg\forall Z.(\neg\text{Pert}(Z,X) \vee \text{Pert}(Z,Y)) \vee \text{Inc}(X,Y)))$

$\forall X.\forall Y.((\neg\text{Inc}(X,Y) \vee \forall Z.(\neg\text{Pert}(Z,X) \vee \text{Pert}(Z,Y))) \wedge (\exists Z.\neg(\neg\text{Pert}(Z,X) \vee \text{Pert}(Z,Y)) \vee \text{Inc}(X,Y)))$

$\forall X.\forall Y.((\neg\text{Inc}(X,Y) \vee \forall Z.(\neg\text{Pert}(Z,X) \vee \text{Pert}(Z,Y))) \wedge (\exists Z.(\text{Pert}(Z,X) \wedge \neg\text{Pert}(Z,Y)) \vee \text{Inc}(X,Y)))$

$\forall X.\forall Y.\forall Z.((\neg\text{Inc}(X,Y) \vee (\neg\text{Pert}(Z,X) \vee \text{Pert}(Z,Y))) \wedge ((\text{Pert}(f(X,Y,Z),X) \wedge \neg\text{Pert}(f(X,Y,Z),Y)) \vee \text{Inc}(X,Y)))$

$\forall X.\forall Y.\forall Z.(\neg\text{Inc}(X,Y) \vee \neg\text{Pert}(Z,X) \vee \text{Pert}(Z,Y)) \wedge$

$\forall X.\forall Y.\forall Z.(\text{Pert}(f(X,Y,Z),X) \vee \text{Inc}(X,Y)) \wedge$

$\forall X.\forall Y.\forall Z.(\neg\text{Pert}(f(X,Y,Z),Y) \vee \text{Inc}(X,Y))$

$C = \{\{\neg\text{Inc}(X_1,Y_1), \neg\text{Pert}(Z_1,X_1), \text{Pert}(Z_1,Y_1)\},$
$\{\text{Pert}(f(X_2,Y_2,Z_2),X_2), \text{Inc}(X_2,Y_2)\},$
$\{\neg\text{Pert}(f(X_3,Y_3,Z_3),Y_3), \text{Inc}(X_3,Y_3)\}\}$

ii) $\forall X(\neg\text{Pert}(X,\emptyset)) \rightsquigarrow C = \{\{\neg\text{Pert}(X_4,\emptyset)\}\}$

$$C = \{ \{ \neg INC(x_1, y_1), \neg PERT(z_1, x_1), PERT(z_1, y_1) \}$$
$$\{ PERT(f(x_2, y_2, z_2), x_2), INC(x_2, y_2) \}$$
$$\{ \neg PERT(f(x_3, y_3, z_3), y_3), INC(x_3, y_3) \}$$
$$\}$$

ii) $\forall x (\neg PERT(x, \theta)) \rightsquigarrow C = \{ \{ \neg PERT(x_4, \theta) \} \}$

6: $\forall x . (INC(\theta, x)) \rightsquigarrow$ El vacío está incluido en todo x.

7b: $\neg(\forall x . (INC(\theta, x)) = \exists x . \neg(INC(\theta, x))$
$= \exists x . (\neg INC(\theta, x))$
$= (\neg INC(\theta, c))$
$\rightsquigarrow \{ \{ \neg INC(\theta, c) \} \}$

Cláusulas:

1. $\{ \neg INC(x_1, y_1), \neg PERT(z_1, x_1), PERT(z_1, y_1) \} \rightsquigarrow$ x no está en y1 $\forall$ z que está en x1
2. $\{ PERT(f(x_2, y_2, z_2), x_2), INC(x_2, y_2) \}$
3. $\{ \neg PERT(f(x_3, y_3, z_3), y_3), INC(x_3, y_3) \}$
4. $\{ \neg PERT(x_4, \theta) \} \rightsquigarrow$ El vacío no tiene elementos.
5. $\{ \neg INC(\theta, c) \} \rightsquigarrow$ El vacío no está incluido en ningún conjunto

4 y 2:

$x_4 := f(x_2, y_2, z_2), \theta := x_2$

6: $\{ INC(\theta, y_2) \}$

6 y 5: $\{ y_2 := c \}$

7: $\emptyset$

Luego, como 7 es insat, 6 es sat.

Tips

- Cada cláusula tendrá variables llamadas diferentes.
- La Skolemización preserva la satisfactibilidad pero no la validez, es decir, no son fórmulas equivalentes.
  - $\exists X.(P(0) \implies P(x))$ es válida pero $P(0) \implies P(c)$ es inválida
- Cada cláusula está separada por un $\wedge$.
- Las claúsulas que estén skolemizadas por la misma función no son renombradas a la hora de escribir las claúsulas, solo las variables.
- La Skolemización reemplaza la variable existencial por las variables ligadas de los cuantificadores que la encapsulan.
- Al calcular el MGU, las cláusulas van en positivo (aunque si o si tomamos un negativo y un positivo). Tomamos un literal de cada claúsula, los cancelamos y nos da un nuevo resultado.
  - 3 y 6: $\{\neg esDormilon(x_9), \neg posee(x_9, y_9), \neg ruidoso(y_9)\}, \{esDormilon(pepe)\}$
  - $S9 = mgu(\{x_9 \stackrel{?}{=} pepe\}) = \{x_9 := pepe\}$
  - 9: $\{\neg posee(pepe, y_9), \neg ruidoso(y_9)\}$
  - Nótese que 9 son los literales que no cancelamos pero con la sustitución correspondiente.

Relación entre Claúsulas y LPO

- 1 claúsula: $\{\neg menor(X,Y), menor(c,Y)\} \equiv \forall X.\forall Y.(\neg menor(X,Y) \vee menor(c,Y))$
- 2 claúsulas: $\{\neg menor(X,Y), menor(c,Y)\}$ y $\{impar(Z), mayor(Z,w)\} \equiv \forall X.\forall Y.(\neg menor(X,Y) \vee menor(c,Y)) \wedge \forall Z.(impar(Z) \vee mayor(Z,w))$

Importante: Las letras en minúscula son constantes, son un valor fijo. Es por eso que quedan igual.

Resolución SLD

- Solo con claúsulas de Horn
- Resolución binaria (1 literal por cada cláusula)
- Resolución lineal (por cada resultado, usamos el resultado + otra claúsula para resolver)
- Empezamos a resolver con claúsula objetivo.

Cláusulas de Horn

- Cláusula Objetivo: 0 positivas n negativas
- Claúsulas de Definición
  - Hecho: 1 positiva 0 negativa
  - Consulta: 1 positiva n negativas

Importante: Las Claúsulas de Horn no pueden tener en una cláusula más de un positivo, ej.: $\{libro(k), radio(k)\}$ no es claúsula de Horn.

Ejercicios Útiles

Ejercicio 13 ★

Alan es un robot japonés. Cualquier robot que puede resolver un problema lógico es inteligente. Todos los robots japoneses pueden resolver todos los problemas de esta práctica. Todos los problemas de esta práctica son lógicos. Existe al menos un problema en esta práctica. ¿Quién es inteligente? Encontrarlo utilizando resolución SLD y composición de sustituciones.

Utilizar los siguientes predicados y constantes: $R(X)$ para expresar que X es un robot, $Res(X,Y)$ para X puede resolver Y, $PL(X)$ para X es un problema lógico, $Pr(X)$ para X es un problema de esta práctica, $I(X)$ para X es inteligente, $J(X)$ para X es japonés y la constante alan para Alan.

Ejercicio 13 ★

Alan es un robot japonés. Cualquier robot que puede resolver un problema lógico es inteligente. Todos los robots japoneses pueden resolver todos los problemas de esta práctica. Todos los problemas de esta práctica son lógicos. Existe al menos un problema en esta práctica. ¿Quién es inteligente? Encontrarlo utilizando resolución SLD y composición de sustituciones.

Utilizar los siguientes predicados y constantes: $R(X)$ para expresar que X es un robot, $Res(X, Y)$ para X puede resolver Y, $PL(X)$ para X es un problema lógico, $Pr(X)$ para X es un problema de esta práctica, $I(X)$ para X es inteligente, $J(X)$ para X es japonés y la constante alan para Alan.

ROBOT(x) ~> x es ROBOT

RES(x,y) ~> x PUEDE RESOLVER y

PL(x) ~> x es PROBLEMA LÓGICO

PR(x) ~> x es un PROBLEMA DE ESTA PRÁCTICA

I(x) ~> x es INTELIGENTE

J(x) ~> x es JAPONÉS

alan

ORDEN:

1) Existe al menos un problema en esta práctica
2) Todos los problemas de esta práctica son lógicos.
3) Cualquier robot que puede resolver un problema lógico es INTELIGENTE
4) Todos los ROBOTS JAPONESES pueden resolver los problemas de esta práctica
5) Alan es ROBOT JAPONÉS

=> ALAN es INTELIGENTE.

1) $\exists x \cdot (PR(x))$
2) $\forall x \cdot (PR(x) \Rightarrow PL(x))$
3) $\forall R \cdot (Robot(R) \wedge \exists y \cdot (PL(y) \wedge RES(R,y)) \Rightarrow I(R))$
4) $\forall R \cdot \forall Y ((Robot(R) \wedge J(R) \wedge PR(Y)) \Rightarrow RES(R,Y))$
5) $Robot(a),\ J(a)$ ~> HIPÓTESIS

6: $\exists x \cdot I(x)$ ~> QUQ

$\neg 6: \neg(\exists x \cdot I(x)) \rightsquigarrow \forall x \cdot \neg I(x) \rightsquigarrow \{\neg I(x)\}$

1) $\exists x \cdot (PR(x)) \rightsquigarrow \{PR(c)\}$
2) $\forall x \cdot (PR(x) \Rightarrow PL(x)) \rightsquigarrow \forall x \cdot (\neg PR(x) \vee PL(x)) \rightsquigarrow \{\neg PR(x_2), PL(x_2)\}$
3) $\forall R \cdot (\neg(Robot(R) \wedge \exists y (PL(y) \wedge RES(R,y))) \vee I(R))$

3) $\forall R.(\neg(Robot(R) \wedge \exists y(PL(y) \wedge RES(R,y)) \vee I(R))$

$\forall R.(\neg Robot(R) \vee \neg\exists y(PL(y) \wedge RES(R,y)) \vee I(R))$

$\forall R.(\neg Robot(R) \vee \forall y.\neg(PL(y) \wedge RES(R,y)) \vee I(R))$

$\forall R.(\neg Robot(R) \vee \forall y(\neg PL(y) \vee \neg RES(R,y)) \vee I(R))$

$\forall R.\forall y(\neg Robot(R) \vee \neg PL(y) \vee \neg RES(R,y) \vee I(R))$

$\rightsquigarrow \{\neg Robot(R_3), \neg PL(y_3), \neg RES(R_3,y_3), I(R_3)\}$

4) $\forall R.\forall y((Robot(R) \wedge S(R) \wedge PL(y)) \Rightarrow RES(R,y))$

$\forall R.\forall y(\neg Robot(R) \vee \neg S(R) \vee \neg PL(y) \vee RES(R,y))$

$\rightsquigarrow \{\neg Robot(R_4), \neg S(R_4), \neg PL(y_4), RES(R_4,y_4)\}$

1) $\{PR(c)\}$

2) $\{\neg PR(x_2), PL(x_2)\}$

3) $\{\neg Robot(R_3), \neg PL(y_3), \neg RES(R_3,y_3), I(R_3)\}$ Dio: $R \equiv Robot$

4) $\{\neg Robot(R_4), \neg S(R_4), \neg PL(y_4), RES(R_4,y_4)\}$

5) $\{Robot(a)\}$

6) $\{S(a)\}$

7) $\{\neg I(x_7)\}$

7 ⨆ 3: $\{x_7 := R_3\}$

8: $\{\neg Robot(R_3), \neg PL(y_3), \neg RES(R_3,y_3)\}$

8 ⨆ 2: $\{y_3 := x_2\}$

9: $\{\neg Robot(R_3), \neg RES(R_3,x_2), \neg PR(x_2)\}$

9 ⨆ 1: $\{x_2 := c\}$

10: $\{\neg Robot(R_3), \neg RES(R_3,c)\}$

10 ⨆ 5: $\{R_3 := a\}$

11: $\{\neg RES(a,c)\}$

11 ⨆ 4: $\{R_4 := a, y_4 := c\}$

12: $\{\neg Robot(a), \neg S(a), \neg PR(c)\}$

12 ⨆ 5: $\{a := a\}$

13: $\{\neg S(a), \neg PR(c)\}$

13 ⨆ 6: $\{a := a\}$

14: $\{\neg PR(c)\}$

14 ⨆ 1: $\emptyset \rightsquigarrow \exists x.I(x)$ soddisfacibile

El ejercicio concluye haciendo la composición de lo queremos saber, es decir, necesitamos saber quién era inteligente. Como X7 era quien queríamos saber, por transitividad con R3, R3 termina siendo a donde a es alan. Por lo tanto El Robot inteligente es Alan.

- Cuando tenemos que plantear nosotros en base a lenguaje natural fórmulas lógicas, tenemos que literalmente usar los predicados que nos dan. No hace falta considerar ninguna precondición.
  - Ej.: Si nos dicen que todos los Robots pueden resolver un problema lógico.
    - El **un** es un existe Y, ese existe Y habla de un problema lógico (PL(Y)) pero en ningun momento nos fijamos si ese Y es un problema (ya se asume).
    - Siempre tenemos hipótesis, que son las cosas que nos dicen como verdadero.
    - Siempre tenemos conclusión, que suele ser una pregunta. Esa pregunta es existencial.

Pruebas por Contradicción

Suponemos que X cosa no vale, como X cosa no vale, vemos que implica y terminamos viendo por qué no fue así.
Ej.: Si el pronóstico anunciaba lluvia, se juntaban en lo de Ana, sino en lo de Carlos. El pronóstico no anunció lluvia y se juntaron igual en lo de Ana.
Suponemos que el pronóstico no anunciaba lluvia, como no anunció lluvia se juntan en lo de Carlos pero terminaron yendo a lo de Ana aunque no llovió.

Programación Lógica

Tips

- Casos base al comienzo. El caso base permite unificar cuando está vacío.
- Recordar siempre los llamados recursivos.
- Recordar para matchear cosas que sean iguales decirlo directamente en la definición.
  - notasEstudiante(E, [(E, M, N) — T], [(E, M, N) — Res]) :- notasEstudiante(E, T, Res). Almacena las notas del estudiante si las E coinciden.
  - notasEstudiante(E, [(E2, M, N) — T], Res) :- E $\setminus =$ E2 $\rightarrow$ importante el E $\setminus =$ E2 porque sino, aunque sea diferente va a armar diferentes ramas donde seguro aparezca el estudiante de vuelta. Nosotros solo queremos un resultado.
- Recordar siempre si un predicado es reversible o no para saber cómo usarlo.

Funciones Útiles

```
member(?elemento, ?lista): Devuelve verdadero si el elemento está en la lista.
Eso sí, solo devuelve verdadero si el elemento está completo.
Ej.: member((tomas, plp, N), [(tomas, plp, 10), (angel, plp, 3)]) devuelve N = 10 porque es lo
que le falta para unificar.
Ej.: member((tomas, plp, 10), [(tomas, plp, 10)]) devuelve true.
Obs: usar member con cuidado, uno de los parámetros enviarlo seguro.

length(?lista, ?longitud): Devuelve la longitud de la lista si no se proporciona la
longitud, caso contrario, devuelve true o false.
Ej.: length([1, 2], A): devuelve A = 2
Ej.: length([1, 2], 2): devuelve true
Ej.: length(A, 2): devuelve [_, _]
Ej.: length(A, B): devuelve todas las posibles listas, es decir: A = [_], B = 0, A=[_, _], B = 1...

sum_list(+Lista, ?Res): Devuelve la suma de los elementos de una lista si no se especifica Res.
Si se especifica, devuelve true o false.
Ej.: sum_list([1, 2], 3): true
Ej.: sum_list([1, 2], A): A = 3

reverse(?L1, ?L2): Tiene varias funciones.
Ej.: reverse([1, 2], [2, 1]) -> false.
Ej.: reverse([1, 2], B) -> B = [2, 1]
Ej.: reverse(A, [2, 1]) -> B = [1, 2]

between(+Low, +High, ?N): Devuelve como respuesta
los elementos que están entre Low y High inclusive. Abre n ramas donde n es la distancia de 1 a 2.
```

Ej.: between(1, 2, N): N = 1 y N = 2.

nonvar(@Term): Devuelve true si es un valor.
Ej.: nonvar(2): true
Ej.: nonvar(A): false

var(@Term): Devuelve true si es una variable, es decir, no tiene un valor.
Ej.: var(A): true
Ej.: var(2): false

append(?List1, ?List2, ?List1And2): Tiene varias funciones.
Si se envían los tres argumentos devuelve true si la concatenación de List1 y List2 es la tercera.
    Ej.: append([1], B, C): C = [1 | B]
    Ej.: append(A, B, [1, 2, 3]): Devuelve todas las posibles instancias de A y B
    que resultan en [1, 2, 3].
    A = [1] B = [2, 3], A = [1, 2] B = [3], ...
    Ej.: inorder(Izq, Res2), inorder(Der, Res3), append(Res2, [R | Res3], Res).
    Agrega el llamado recursivo siempre a la izquierda, luego la raíz y la recursión del lado derecho.

not(:Goal): Devuelve verdadero si el cuerpo es falso.
    Ej.: not(esEstudiante(tomas)) será verdadero sí y solo sí tomas NO es estudiante.
    Ej.: not(conoceLenguaje(tomas, lisp)) es verdadero sí y solo sí tomás no sabe lisp.
    Ej.: not((conoceLenguaje(tomas, lisp), esEstudiante(tomas))) será verdadero sí y solo sí
    tomás no sabe lisp ni tampoco es estudiante.

last(?List, ?Last): Devuelve el último elemento de una lista, o devuelve
verdadero sí y solo sí Last es el último elemento de List.
Ej.: last([1, 2], 1): False
Ej.: last([1, 2], A): A = 2

flatten(+List, -FlatList): Elimina las listas anidadas pasando todo a un solo nivel.
Ej.: flatten([1, [2, 3, [4, 5, [6,7]]]], B) -> B = [1, 2, 3, 4, 5, 6, 7]

sort(+List, ?SortedList): Ordena la lista de manera ascendente.
Si se proporcionan ambos elementos, es verdadero sí y solo sí SortedList es List ordenada de
manera ascendente.
Ej.: sort([3, 2, 1], B): B = [1, 2, 3]
Ej.: sort([1, 2, 3], [3, 1, 2]): False
Ej.: sort([3, 2, 1], [1, 2, 3]): True

is(?Expr, +Expr2): Resuelve la expresión de la derecha y la unifica con la expresión de la izquierda.
is se convierte a = cuando se resuelve la expresión.
Ej.: 1 is 0+1 -> Evalua a true pues 1 = 1

Operadores Aritméticos: Requieren de tener ambos argumentos instanciados.
<(+A, +B): Es verdadero sí y solo sí A es menor a B.
Ej.: 2 < 2: False
<=(+A, +B): Es verdadero sí y solo sí A es menor o igual que B.
Ej.: 2 =< 2: True
=:=(+A, +B): Es verdadero sí y solo sí A y B son iguales.
Ej.: 1 =\= 1: True
=\=(+A, +B): Es verdadero sí y solo sí A y B tienen un valor diferente.
Ej.: 1 =\= 2: True

Operadores no Aritméticos
= (?T, ?V): Realiza la unificación de términos.
Si ambos términos son proporcionados, es verdadero sí y solo sí unifican.
Ej.: A = B -> A = B.
Ej.: A = 1 -> A = 1. Asigna el valor de 1 a la variable A.
Ej.: 1 = 2 -> False.

\=(+Expr, +Expr2): Su uso tiene sentido cuando ambas expresiones están instanciadas.
Devuelve verdadero si no unifican.

```
Ej.: f(g) \= h(f) -> Verdadero porque no unifica por clash.
```

Ejercicios Útiles

```
desdeReversible(+Low, ?High): Devuelve tantos elementos haya de distancia de X a Y. Uno por uno.
Abre n ramas siendo n la distancia entre X e Y.

desdeReversible(X, Y) :- var(Y), Y = X.
desdeReversible(X, Y) :- nonvar(Y), X =< Y.
desdeReversible(X, Y) :- var(Y), X1 is X + 1, desdeReversible(X1, Y).
--
parteQueSuma(+L,+S,-P): Es verdadero cuando P es una lista con elementos de L que suman S.

parteQueSuma(_, 0, []).
parteQueSuma([X|XS], S, [X|P]) :- S1 is S - X, S1 >= 0, parteQueSuma(XS, S1, P).
parteQueSuma([_|XS], S, P) :- S > 0, parteQueSuma(XS, S, P).
--
borrar(+L, +E, ?Res): La idea es usarlo con L y E instanciadas.

borrar([], _, []).
borrar([H | T], H, XS) :- borrar(T, H, XS).
borrar([H | T], E, [H | XS]) :- H \= E, borrar(T, E, XS).
--
sacarDuplicados(+L, ?Res): La idea es usarlo con L instanciada.

sacarDuplicados([], []).
sacarDuplicados([H | XS], L2) :- member(H, XS), sacarDuplicados(XS, L2).
sacarDuplicados([H | XS], [H | L2]) :- not(member(H, XS)), sacarDuplicados(XS, L2).
--
permutacion(?L, ?Res): La idea es usarlo con L instanciada.

permutacion([], []).
permutacion(L1, [H|T]) :- append(L, [H|R], L1), append(L, R, Resto), permutacion(Resto, T).
--
reparto(+L, +N, -LListas): Es verdadero sí y solo sí LListas es una
lista de N listas (N mayor a 1) de cualquier longitud (inc. vacias) tales que al concatenarlas se obtiene

reparto([], 0, []). % Cuando N=0 solo podemos unificar si ya repartimos todo L.
reparto(L, N, [X|Xs]) :-
    N > 0, % Hay sublistas por generar.
    append(X, L2, L), % Generamos todas las posibles sublistas X.
    N2 is N-1, % L2 es lo que queda de L para repartir en N-1 sublistas.
    reparto(L2, N2, Xs). % Generamos el resto de las sublistas.
--
repartoSinVacias(+L, -LListas) similar al anterior, pero ninguna de las listas de LListas puede ser vacía
Como no pueden haber sublistas vacías, a lo sumo hay N sublistas siendo length(L, N).

repartoSinVacias(L, Xs) :-
    length(L, N),
    between(1, N, K), % Generamos todas los posibles K = cantidades de sublistas.
    reparto(L, K, Xs), % Repartimos en K sublistas.
    not((member(X, Xs), length(X, 0))). % No pueden haber sublistas vacías.
```

Generación Infinita

- Nunca usar más de un generador infinito.
- Los generadores infinitos siempre van a la izquierda de cualquier otro generador.
- Los generadores infinitos deben usarse únicamente para generar infinitas soluciones.
- Las soluciones generadas en cada paso deben ser finitas, y que entre todas cubran todo el espacio de soluciones que se busca generar.
  - Si hay que generar todas las listas finitas de enteros positivos no nos sirve que el generador infinito nos vaya dando la longitud de la lista porque para cada longitud hay infinitas listas posibles.

- La idea es acotar y generar las listas que sumen una cantidad determinada. Esto es útil porque además las soluciones en cada paso son disjuntas y no hay repetidos (véase parteQueSuma).

Ejemplos de Generación Infinita

```
generarCapicua(L) :- desde(1, N), listaQueSuma(N, L), esCapicua(L).
esCapicua(L) :- reverse(L,L).
Acá el handler de la generación infinita es listaQueSuma, mientras que el filter lo hace esCapicua.

coprimos(X,Y) :- nonvar(X), nonvar(Y), 1 is gcd(X,Y).
coprimos(X, Y) :- desdeReversible(1, S), between(1, S, X), Y is S-X, 1 is gcd(X, Y).
```

Paradigma Orientado a Objetos

Básico

- Los programas están conformados por objetos que interactúan entre sí con mensajes.
- Todos los mensajes tienen respuesta.
- Un mensaje es simplemente una solicitud al objeto receptor donde este mensaje tiene respuesta sí y solo sí el objeto receptor entiende ese mensaje. Esta respuesta es ofrecida por un método del mismo objeto.
- Colaboradores Internos: Son los atributos o variables de instancia de un objeto.
- Colaboradores Externos: Son los parámetros o argumentos que tiene un mensaje particular.
  - 1@1 insideTriangle with: 0@0 with: 0@0 with: 0@0. Se realiza una instancia de Point 1@1 y se envía el mensaje insideTriangle con tres keywords o también llamados parámetros.

Encapsulamiento

Una clase debería estar abierta a extensión pero cerrada a modificaciones.
Solo es posible interactuar con un objeto a través de sus métodos los cuales ofrece pues, el estado interno de un objeto es inaccesible desde el exterior.

Objetos, objetos y objetos

Todo objeto es instancia de alguna clase, y a su vez, estas son objetos.

- Una clase es un objeto que abstrae el comportamiento de todas sus instancias.
- Todas las instancias de una clase tienen los mismos atributos
  - Cada instancia puede modificar a gusto sus valores sin afectar a las demás. Cada instancia es única.
- Todas las instancias de una clase usan el mismo método para responder un mismo mensaje.
  - Todas las instancias responden de **la misma manera**. Esto es porque el receptor no conoce al emisor, salvo que el emisor se envíe como colaborador.

Palabras Reservadas

nil, true, false, self, super, thisContext

Literales

- Caracteres: $
- Strings: 'hola'
- Símbolos (inmutables): #hola
- Constantes numéricas: 29, -1, 5

Herencia

- Cada clase es subclase de alguna otra clase. Si no se especifica, las clases heredan de Object por defecto.
- Una clase hereda todos los métodos de su superclase.
- Una clase puede hacer un overrida a un método definido en la superclase por otro más específico.
- self: Hace referencia al objeto de instancia.
- super: Hace referencia a la super-clase de la instancia.

Nota: self==super porque refieren al mismo objeto pero difieren en que, si se usa super la búsqueda del método que implementa el mensaje m debe comenzar desde la superclase.

super vs self

- super busca siempre hacia arriba, es decir, las super-clases de la clase a la que mandamos el mensaje. Si el mensaje no está en el super, arroja un error NotUnderstand.
- self busca siempre hacia abajo. Si el mensaje no está, arroja un error NotUnderstand.

Importante: si hacemos instanciaObj mensaje pero el mensaje no es respondido por instanciaObj mandará el mensaje a su padre, y si su padre no lo tiene, fallará.

Clase Abstracta

Llamamos clase abstracta a una clase que está destinada a abstraer el comportamiento de sus subclases pero no tienen instancias.

Tipos de Mensajes

- Mensajes Unarios: Reciben un solo parámetro.
  - 1 class
  - Mensaje: class — Receptor: 1
- Mensajes Binarios: Reciben dos parámetros.
  - $1 + 2$
  - Mensaje: $+$ — Receptor: 1 — Colaborador: 2
- Mensajes Keyword: Reciben parámetros que se pueden distinguir con nombre. No importa el orden en cual se envían porque están dados por una key.
  - a at: 1 put: 'hola'
  - Mensaje: at:put — Receptor: a — Colaborador/es: 1, 'hola'

La prioridad de los mensajes es la siguiente: $unario > binario > keyword$
Nota: Los paréntesis **()** definen la prioridad máxima.

Bloques / Closures

- Permiten reutilizar código. Recuerdan el estado cuando fueron definidos y qué variables estaban presente. Es una secuencia de envíos de mensajes.
- No usar return dentro de bloques. Corta todo tipo de ejecución.
- Los argumentos obligatorios tienen prioridad sobre los locales.
- Los parámetros se envían como value: param
- Cuando se almacenan en una variable **no se ejecutan**.
- Para llamar a un bloque hay que enviar todos sus parámetros.
- Devuelven como resultado la última expresión.

```
hacerAlgo
|bloque val|
bloque := [:x :y | |z| z:=10. x+y+z].
val := bloque value: 1 value: 2. // retorna 13

hacerAlgo
|bloque val val2|
bloque := [:x | [:y | |z| z:=10. x+y+z]].
val := bloque value: 1. // retorna el bloque [:y | |z| z:=10. x+y+z] que
recuerda el valor de x cuando se definió, es decir, 1.
val2 := val value: 2. //retorna el resultado del bloque más interno, es decir,
1+2+10 = 13.
```

Return

- Se indica con $\wedge$.
- Corta todo tipo de ejecución, es decir: $[|x||x := 0. \wedge 0]\ value.$ devuelve 0.
- No usar return en bloques. Porque como el bloque vive en un universo aparte, el return es algo peligroso.
- Si el return no se indica dentro del método, devuelve self. Es decir, la instancia del objeto que recibió el mensaje.

Colecciones

Existen varias: Bag (Multiconjunto), Set (Conjunto), Array (Arreglo), OrderedCollection (Lista), SortedCollection (Lista Ordenada) y Dictionary Hash (Hash).

- Bag with: 1 with: 2 with: 4
- #(1 2 4) = (Array with: 1 with: 2 with: 4)
- Bag withAll: #(1 2 4)

Mensajes más comunes

- add: agrega un elemento
- at: devuelve el elemento en una posición (indexa desde 1).
- at:put: agrega un elemento a una posición.
- includes: responde si un elemento pertenece o no.
- includesKey: responde si una clave pertenece o no.
- do: evalúa un bloque con cada elemento de la colección. No muta ni devuelve un resultado, solo sirve para efectos secundarios.
- select: Devuelve los elementos de una colección que cumplen un predicado (filter de funcional).
- reject: la negación del select
- collect: Es el map de funcional.
- detect: devuelve el primer elemento que cumple un predicado.
- detect:ifNone: permite ejecutar un bloque si no se encuentra ningun elemento
- reduce: toma un bloque de dos o mas parámetros de entrada y hace fold de los elementos de izquierda a derecha.

Booleanos

- ifFalse:, ifTrue:ifFalse, &, —, and:, or:, not, $=$, $\leq$, $\geq$
- \\: te da el resto

Machete

```
Sintáxis
    | var1 var2... |
    [:arg1 :arg2 | | var1 var 2 | expresion1. expresion2...]
    expresion1. expresion2. expresion3
    objeto mensaje
    objeto msg1; msg2.
    var := expresion
    ^expresion
Palabras Reservadas
    self, super, thisContext, false, true, nil
Literales
    123, 123.4, $c, 'texto', #simbolo, #(123 123.3 $a) array
```

Clases Padres de Mensajes

- new: Object Class a menos que haya sido específicado con ese nombre en método de clase.

super eval value: Block Closure.

self eval value: Block Closure.

- +: Implementado en SmallInteger si son números.

Ejercicios Útiles

Jerarquía

```
20 + 3 * 5
Mensaje: + | Obj Receptor: 20 | Colaboradores: 3 | Res = 23
Mensaje: * | Obj Receptor: 23 | Colaboradores 5 | Res = 115

20 + (3*5)
Mensaje: * | Obj Receptor: 3 | Colaboradores: 5 | Res = 15
Mensaje: + | Obj Receptor: 20 | Colaboradores 15 | Res = 35

1 = 2 ifTrue: ['what!?'].
Mensaje: = | Obj Receptor: 1 | Colaboradores: 2 | Res = instanciaFalse
Mensaje: ifTrue | Obj Receptor: instanciaFalse | Colaboradores: ['what!?'].
| Res = False.

1@1 insideTriangle: 0@0 with: 2@0 with: 0@2.
Mensaje: insideTriangle:with:with: | Obj. Receptor: 1@1
(instancia de point) | Colaboradores: 0@0 with: 2@0 with: 0@2

Object subclass: #SnakesAndLadders
    instanceVariablesNames: 'players squares turn die over'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SnakesAndLadders'

Hay varios mensajes acá.
Mensaje: subclass | Receptor: Object Class |Colaborador: #SnakesAndLadders
Mensaje: instanceVariableNames | Receptor: SnakesAndLadders |
Colaboradores: 'players squares turn die over' cada uno por separado
Mensaje: classVariableNames | Receptor: SnakesAndLadders |Colaborador: ''
Mensaje: poolDictionaries | Receptor: SnakesAndLadders | Colaborador: ''
Mensaje: category | Receptor: SnakesAndLadders |Colaborador: 'SnakesAndLadders'
```

Bloques

```
[ :x :y | |z| z:=x+y ] value: 1 value: 2. Bloque bien definido, dos parámetros
y una variable local.
[ :x :y | x+1 ] value: 1. Arroja error, falta un parámetro.
[:x | [:y | x+1]] value: 2. Bloque bien definido, devuelve un nuevo bloque [:y | x+1]
[:x :y :z | x + y + z] valueWithArguments: #(1 2 3). Bloque bien definido,
envía tres argumentos en orden.
[ |x y z| x + 1] Arroja error. x es UndefinedObject.
[ :x :y :z | x + 1] Bloque bien definido, espera 3 argumentos
obligatorios pero termina usando uno.

Class: BlockClosure
curry
    ^[:x | [:y | self value: x value: y]].

flip
    ^[:x :y | self value: y value: x].

Class: Integer
Iterativa
    timesDo: aBlock
    | count |
    count := 1.
    [count <= self]
        whileTrue:
            [aBlock value. count := count + 1]

Recursiva
    timesDo2: aBlock
    self > 0 ifFalse: ^self.
    aBlock value.
```

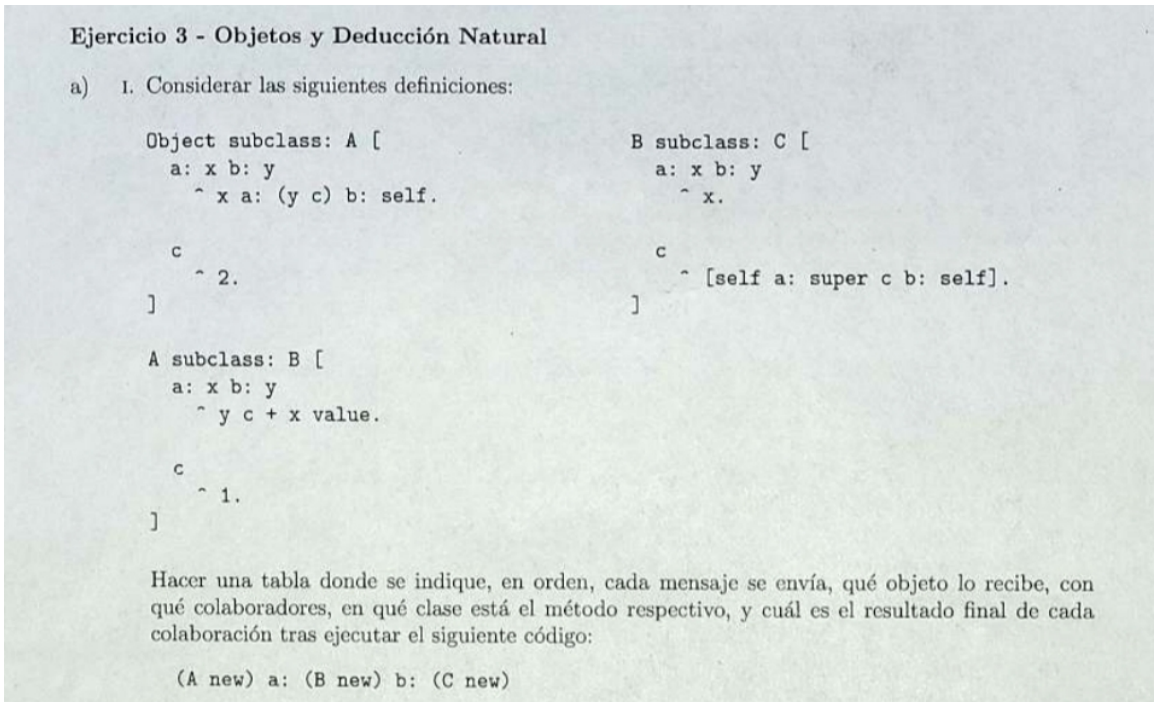
```
    self - 1 timesDo: aBlock.

Class: Collection
map: aBlock
    |col2|
    col2:=(self class) new.
    self do: [ :elem | col2 add: (aBlock value: elem)].

    ^col2.

minimo: aBlock
"Una implementación poco elegante de la obtención del valor original que genera un mínimo
luego de aplicar un bloque."
    | minElement minValue |

    self do: [:each | | val |
        minValue ifNotNil: [
            (val := aBlock value: each) < minValue ifTrue: [
                minElement := each.
                minValue := val]]
        ifNil: ["Caso del primer elemento que se lee"
            minElement := each.
            minValue := aBlock value: each].
        ].

    ^minElement
```

Listas

```
#collect: es el map.
| res |
res := #(1 2 4) collect: [:numero | numero * 2].

El resultado sería multiplicar por dos todos los
elementos de la lista, es decir, [2, 4, 8].

#select: es el filter
| res |
res := #(1 2 3) select: [:numero | numero = 1 ].

El resultado sería [1]

sabeResponder: L
    |res|
    res := #(1 2 3) select:[:each | each respondsTo: #ptff]. -> Los true no hace falta colocarle =.
    ^res

sabeResponder (solo closure)
^[:L | L select: [:each | each respondsTo: #ptff]].

#inject: into: El primer argumento es el resultado de la llamada anterior y el segundo el
elemento actual.

listaNumeros := OrderedCollection with: 1 with: 2 with: 3.
listaNumerosSuma := listaNumeros
inject: 0
into: [ :result :elem | result + elem ].

El resultado seria 6.

#reduce: (o #fold): Es el foldl que conocemos, hace algo de izquierda a derecha.
#(10 20 5 30 15) reduce: [:max :each | max max: each].
El resultado en este ejemplo sería el 30.
```

```
#(1 2 3 4 5) reduce: [:product :each | product * each].
El resultado en este ejemplo sería 120.

#reduceRight: es un foldr convencional. Resuelve de derecha a izquierda.
#(1 2 3 6) reduceRight: [:acc :each | each-acc].
El resultado en este ejemplo sería 0.

#do

listaNumeros := OrderedCollection with: 1 with: 2 with: 3.
listaNumeros2 := OrderedCollection new.
listaNumeros do: [ :each | listaNumeros2 add: each + 1 ].

En este caso, listaNumeros2 termina teniendo los valores de [2, 3, 4].

Almacenar en una lista todos los divisores de un numero
Op1:
SmallInteger << divisores
    | lista |
    lista := OrderedCollection new.
    1 to: self do: [:each | (self \\ each) = 0 ifTrue: [lista add:each]].
    ^lista.


Op2:
SmallInteger << divisores
    |count lista|
    lista := OrderedCollection new.
    [count <= self]
        whileTrue: [
            (self \\ count) = 0 ifTrue: [lista add: count].
            count := count+1.
        ].
    ^lista.
```

Paso a paso POO



Ejercicio 3 - Objetos y Deducción Natural

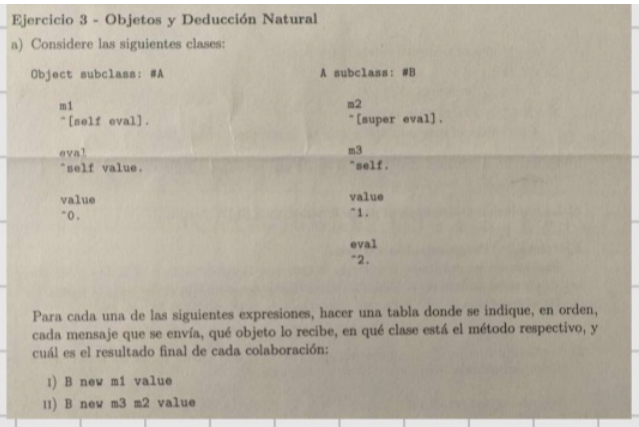
a) 1. Considerar las siguientes definiciones:

```
Object subclass: A [
  a: x b: y
    ^ x a: (y c) b: self.

  c
    ^ 2.
]

A subclass: B [
  a: x b: y
    ^ y c + x value.

  c
    ^ 1.
]

B subclass: C [
  a: x b: y
    ^ x.

  c
    ^ [self a: super c b: self].
]
```

Hacer una tabla donde se indique, en orden, cada mensaje se envía, qué objeto lo recibe, con qué colaboradores, en qué clase está el método respectivo, y cuál es el resultado final de cada colaboración tras ejecutar el siguiente código:

```
(A new) a: (B new) b: (C new)
```

Mensaje	Receptor	Colaboradores	Implementado en	Resultado
new	A		Object	unA
new	B		object	unB
new	C		Object	unC
a:b:	unA	unB, unC	A	3
c	unC		C	bloque
a:b:	unB	bloque, unA	B	3
value	bloque		BlockClosure	1
c	unC		B	1
a:b:	unC	1, unC	C	1
c	unA		A	2
+	2	1	SmallInteger	3



Ejercicio 3 - Objetos y Deducción Natural

a) Considere las siguientes clases:

```
Object subclass: #A                    A subclass: #B

    m1                                     m2
    ^[self eval].                          ^[super eval].

    eval                                   m3
    ^self value.                           ^self.

    value                                  value
    ^0.                                    ^1.

                                           eval
                                           ^2.
```

Para cada una de las siguientes expresiones, hacer una tabla donde se indique, en orden, cada mensaje que se envía, qué objeto lo recibe, en qué clase está el método respectivo, y cuál es el resultado final de cada colaboración:

i) B new m1 value
ii) B new m3 m2 value

a) B HEREDA DE A. SI MÉTODO NO ESTÁ EN B VA A A. DEVUELVEN BLOQUE NO EVALÚA.

i) B NEW ⇝ OBJECT CLASS ⇝ UNB (HABLA DE UNO.)

UNB M1 ⇝ BUSCO M1 NO ESTÁ EN B, SIGO A A. M1: ^[SELF EVAL]

[SELF EVAL] ⇝ VALUE ⇝ BLOCK CLOSURE

SELF EVAL ⇝ RETORNA 2.

MENSAJE	RECEPTOR	OBJ QUE EJECUTA	COLAB	RES
NEW:	B	OBJECT CLASS	-	UNB
M1	UNB	A	-	[SELF EVAL] ✓
VALUE	[SELF EVAL]	BLOCK CLOSURE	-	2
EVAL	UNB	B	-	2
NEW	B	OBJECT CLASS	-	UNB
M3	UNB	B	-	UNB
M2	UNB	B	-	[SUPER EVAL]
VALUE	[SUPER EVAL]	BLOCK CLOSURE	-	1
EVAL	UNB	A		1
VALUE	UNB	B		1

B NEW ⇝ UNB

UNB M3 ⇝ ^SELF ⇝ UNB.

UNB M2 ⇝ ^[SUPER EVAL].

VALUE ⇝ [SUPER EVAL]

[SUPER EVAL] ⇝ [SELF VALUE]

[SELF VALUE] ⇝ 1