# Apunte Final PLP

Mateo Ziffer

15 de julio de 2025

# Índice

# 1. Introducción

En esta sección escribiré algunas definiciones generales y el objetivo de la materia.
Def: la *programación* es el proceso de escribir instrucciones que una computadora puede ejecutar para resolver algún problema.
Def: un *programa* es una serie de instrucciones/definiciones que una computadora sigue para realizar una tarea específica.
Def: un *lenguaje de programación* es un formalismo artificial en el que se pueden describir compataciones.
Def: un *paradigma* es una marco filosófico y teórico de una escuela científica o disciplina en la que se formulan teorías, leyes y generalizaciones y se llevan a cabo experimentos que les dan sustento.
Def: un *paradigma de programación* es una marco filosófico y teórico en el que se formulan soluciones a problemas de naturaliza algorítmica.
Lo entendemos como un estilo de programación en el que se escriben soluciones a problemas en términos de algoritmos.

Estudiamos la gramática, semántica, pragmática e implementación de los lenguajes de programación.
Def: la *gramática* responde a ¿Qué frases son correctas?, establece el alfabeto, las palabras (o tokens), es decir, la secuencia válida de símbolos y la sintaxis, es decir, las secuencias de palabras que son frases legales.
Def: la *semántica* responde a ¿Qué significa una frase correcta?, estableciendole un significado a cada frase correcta.
Def: la *semántica* responde a ¿Qué significa una frase correcta?, estableciendole un significado a cada frase correcta.
Def: la *pragmática* responde a ¿Cómo usamos una frase significativa?. Las frases con el mismo significado pueden usarse de diferentes maneras, diferentes contextos pueden requerir frases más elegantes, eficientes, dialectales, etc.
Def: la *implementación* responde a ¿Cómo ejecutar una frase correcta, de manera que respetemos la semántica?. Es fundamentas para los diseñadores e implementadores del lenguaje, no necesariamente para el usuario (programador).

Hay tres apectos importantes de los lenguajes de programación:
Motivación de la *programación*: los lenguajes de programación tienen distintas características que permiten abordar un mismo problema de distintas ma los lenguajes de programación tienen distintas características que permiten abordar un mismo problema de distintas maneras.
Motivación de la *semántica*: probar teoremas sobre el comportamiento de los programas, para darles significado matemático y poder confiar en que hace lo que queremos, en AED vimos como hacerlo con triplas de Hoare, pero en PLP veremos otras maneras de dar semántica.
Motivación: la *implementación*: una computadora física ejecuta programas escritos en un lenguaje, el código máquina, pero necesita poder ejecutar programas escritos en otros lenguajes, a través de la interpretación, el chequeo e inferencia de tipos y la compilación.

# 2. Deducción natural, lógica proposicional y lógica de primer orden

## 2.1. Sistema deductivo

Queremos podes hacer afirmaciones matemáticamente precisas sobre programas en distintos lenguajes de programación.
Def: sistema deductivo: sirve para razonar acerca de juicios. Dado un conjunto de axiomas y reglas de inferencia que tienen la siguiente estructura:

$$\frac{}{\langle\text{axioma}\rangle}\langle\text{nombre del axioma}\rangle \quad \frac{\langle\text{premisa}_0\rangle\ \langle\text{premisa}_1\rangle\ ...\ \langle\text{premisa}_n\rangle}{\langle\text{conclusión}\rangle}\langle\text{nombre de la regla}\rangle$$

Def: axioma. afirmacion basica que se asuma como verdadera
Def: regla de inferencia. permite derivar afirmaciones (teoremas) a partir de axiomas y otras afirmaciones
Obs: un axioma es una regla de inferencia sin premisas

Las premisas son condiciones suficientes para la conclusión.
Def: derivación. Procedimiento sistemático que permite construir una demostración, mostrando cómo una afirmación se deduce a partir de un conjunto de axiomas y reglas de inferencia.
Def: arbol de derivacion. Representación gráfica de una derivación. Un árbol finito donde los nodos representan afirmaciones, la raóz es la afirmación que se quiere probar y las ramas representan las reglas de inferencias que conectan a las afirmaciones. Parte de ciertas premisas (hojas) y llega a una conclusión (raíz).
Def: una afirmación es derivable si existe alguna derivación sin premisas que la tiene como conclusión.

## 2.2. Lógica proposicional

### 2.2.1. Sintaxis

Suponesos un conjunto infinito de variables proposicionales $\mathcal{P} = \{P, Q, R, ...\}$. Las fórmulas bien formadas (fbf) de la lógica proposicional se construyen inductivamente según las siguientes reglas:

- cualquier variable proposicional es una fórmula.
- $\perp$ es una fórmula.
- si $\tau$ es una fórmula, entonces $\neg\tau$ es una fórmula.
- si $\tau$ y $\sigma$ son fórmulas, entonces $(\tau \wedge \sigma)$, $(\tau \vee \sigma)$ y $(\tau \Rightarrow \sigma)$ son fórmulas.

Como sistema deductivo, la afirmación X FORM denota que X es una fórmula de la lógica proposicional.

$$\frac{P \in \mathcal{P}}{P\ \text{FORM}}\text{FP} \quad \frac{}{\perp\ \text{FORM}}\text{F}\perp \quad \frac{\tau\ \text{FORM}}{\neg\tau\ \text{FORM}}\text{F}\neg$$

$$\frac{\tau\ \text{FORM} \quad \sigma\ \text{FORM}}{\tau \wedge \sigma\ \text{FORM}}\text{F}\wedge \quad \frac{\tau\ \text{FORM} \quad \sigma\ \text{FORM}}{\tau \vee \sigma\ \text{FORM}}\text{F}\vee \quad \frac{\tau\ \text{FORM} \quad \sigma\ \text{FORM}}{\tau \Rightarrow \sigma\ \text{FORM}}\text{F}\Rightarrow$$

Usualmente no vamos a definir la sintaxis de lenguajes a través de sistemas deductivos, vamos a escribirlos de maneras abreviadas, usando gramáticas.
Def: las fórmulas son las expresiones que se pueden generar a partir de la siguiente gramática:

$$\tau, \sigma, \rho ::= P|\bot|(\tau \wedge \sigma)|(\tau \vee \sigma)|(\tau \Rightarrow \sigma)|\neg\tau$$

Obs: las gramáticas definen sistemas deductivos de manera abreviada. Una expresión $\tau$ se puede generar a partir de la gramática de arriba sii el juicio $\tau FORM$ es derivable en el sistema anterior.

Por convenciones de notación, omitimos paréntesis más externos y la implicación es asociativa a derecha, pero cuidado con los otros conectivos que no son conmutativos ni asociativos.

### 2.2.2. Semántica

Recordemos, la *semántica* responde a ¿Qué significa una frase correcta?, estableciendole un significado a cada frase correcta.
Def: una valuación es una función $\mathcal{P} \to \{V, F\}$ que asigna valores de verdad a las variables proposicionales.
Def: una valuación $v$ satisface una fórmula $\tau$ si $v \models \tau$, donde

$$\begin{gathered}
v \models P \text{ sii } v(P) = V \\
v \models \tau \wedge \sigma \text{ sii } v \models \tau \text{ y } v \models \sigma \\
v \models \tau \vee \sigma \text{ sii } v \models \tau \text{ o } v \models \sigma \\
v \models \tau \Rightarrow \sigma \text{ sii } v \not\models \tau \text{ o } v \models \sigma \\
v \models \neg\tau \text{ sii } v \not\models \tau
\end{gathered}$$

Obs: $v \models \bot$ nunca vale.
Def: un contexto $\Gamma$ es un conjunto finito de fórmulas.
Def: una valuación $v$ satisface un contexto $\Gamma$ ($v \models \Gamma$) sii $v$ satisface a todas las fórmulas de $\Gamma$. Nota: toda valuación satisface al contexto vacío.
Def: una fórmula $\tau$ es consecuencia lógica (o consecuencia semántica) de un conjunto $\Gamma$ ($\Gamma \models \tau$) sii cualquier valuación $v$ que satisface a $\Gamma$ también satisface a $\tau$.

Hay varios problemas con un enfoque puramente semántica, por eso vamos a definir un sistema deductivo.

## 2.3. Deducción natural

Es un sistema deductivo, pero existe otros. Trabaja con afirmaciones de la forma.

$$\underbrace{\Gamma}_{\text{hipótesis}} \vdash \underbrace{\tau}_{\text{tesis}}$$

A estas afirmaciones las denominamos juicios. Informalmente, un juicio afirma que a partir de las hipótesis en el contexto $\Gamma$ es posible deducir la fórmula de la tesis.

### 2.3.1. Deducción natural intuicionista (NJ)

$$\frac{}{\Gamma, \tau \vdash \tau}\text{ax} \quad \frac{\Gamma \vdash \bot}{\Gamma \vdash \tau}\bot_e \quad \frac{\Gamma, \tau \vdash \bot}{\Gamma \vdash \neg\tau}\neg_i \quad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \neg\tau}{\Gamma \vdash \bot}\neg_e$$

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \wedge \sigma}\wedge_i \quad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \tau}\wedge_{e_1} \quad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \sigma}\wedge_{e_2}$$

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

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

<ins>Teo:</ins> (debilitamento) Si $\Gamma \vdash \tau$ es derivable, entonces $\Gamma, \sigma \vdash \tau$ es derivable.

$$\frac{\Gamma \vdash \tau}{\Gamma, \sigma \vdash \tau}\text{W}$$

Se puede demostrar por inducción estructural en la derivación.

*Reglas derivadas*

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

### 2.3.2. Deducción natural clásica (NK)

NK extiende a NJ con principios de razonamiento clásicos. Si un juicio es derivable en NJ, también es derivable en NK. NJ es más restrictiva. Para hacer matemática, comúnmente usamos lógica clásica. Las derivaciones NJ se pueden entender como programas. NJ es la base de un lenguaje de programación funcional.

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

Con agregar una de las tres reglas pues de cualquiera se pueden deducir las otras.

<ins>Teo:</ins> (Corrección y completitud). Son equivalentes $\Gamma \vdash \tau$ es derivable en NK y $\Gamma \models \tau$.

## 2.4. Unificación

Suponemos un conjunto finito de constructores de tipos. Los tipos se forman usando incógnitas y constructores. $\tau ::= X_n | C(\tau_1, ..., \tau_n)$. La unificación es el problema de resolver sistemas de ecuaciones entre tipos con incógnitas.

Una sustitución es una función que a cada incógnita le asocia un tipo. Notamos $\{X_(k_1) := \tau_1, ..., X_(k_n) := \tau_n\}$ a la sustitución $S$ tq $S(X_(k_i)) = \tau_i$ para cada $1 \leq i \leq n$ y $S(X_k) = X_i$ para cualquier otra incógnita. Si $\tau$ es un tipos, escribimos $S(\tau)$ para el resultado de reemplazar cada incógnita de $\tau$ que le otorga $S$.

El problema de unificación es un conjunto finito $E$ de ecuaciones entre tipos que pueden involucrar incógnitas.

$$E = \{\tau_1 \stackrel{?}{=} \sigma_1, ..., \tau_n \stackrel{?}{=} \sigma_n\}$$

El unificador para $E$ es una sustitución $S$ tq $S(\tau_1) = S(\sigma_1)$, ..., $S(\tau_n) = S(\sigma_n)$.

En general la solución al problema de unificación no es única (pueden haber infinitas). Una sustitución $S_A$ es más general que una sustitución $S_B$ si existe una sustitución $S_C$ tal que $S_B = S_C \circ S_A$. Es decir, $S_B$ se obtiene instanciando variables de $S_A$.

*Algoritmo de unificación de Martelli-Montanari* Dado un problema de unificación $E$, mientras $E \neq$, se aplica sucesivamente alguna de las seis reglas. Esta puede resultar en una falla, de lo contrario, la regla es de la forma $E \to_S E'$. La resolución del problema $E$ se reduce a resolver otro problema $E'$ aplicando la sustitución $S$.

$$\begin{aligned}
\{X_n \stackrel{?}{=} X_n\} \cup E &\stackrel{\text{Delete}}{\longrightarrow} E \\
\{C(\tau_1, ..., \tau_n) \stackrel{?}{=} C(\sigma_1, ..., \sigma_n)\} \cup E &\stackrel{\text{Decompose}}{\longrightarrow} \{\tau_1 \stackrel{?}{=} \sigma_1, ..., \tau_n \stackrel{?}{=} \sigma_n\} \cup E \\
\{\tau \stackrel{?}{=} X_n\} \cup E &\stackrel{\text{Swap}}{\longrightarrow} \{X_n \stackrel{?}{=} \tau\} \cup E \\
\{X_n \stackrel{?}{=} \tau\} \cup E &\stackrel{\text{Elim}}{\longrightarrow}_{\{X_n := \tau\}} E' = \{X_n := \tau\}(E) \text{si } X_n \text{ no ocurre en } \tau \\
\{C(\tau_1, ..., \tau_n) \stackrel{?}{=} C'(\sigma_1, ..., \sigma_m)\} \cup E &\stackrel{\text{Clash}}{\longrightarrow} \text{falla si } C \neq C' \\
\{X_n \stackrel{?}{=} \tau\} \cup E &\stackrel{\text{Occurs-Check}}{\longrightarrow} \text{falla si } X_n \neq \tau \text{ y } X_n \text{ ocurre en } \tau
\end{aligned}$$

Teo: (Corrección) El algoritmo termina para cualquier problema de unificación $E$. Si $E$ no tiene solución, el algoritmo llega a una falla. Si tiene solución, el algoritmo llega a : $E = E_0 \to_{s_1} E_1 \to_{s_2} E_2 \to ... \to_{s_n} E_n =$. Además, $S = S_n \circ ... \circ S_2 \circ S_1$ es un unificador para $E$. Además, dicho unificador es el más general posible (salvo renombre de incógnitas).

Notamos $mgu(E)$ al unificador más general de $E$, si existe.

## 2.5. Lógica de primer orden

La lógica proposicional permite razonar acerca de proposiciones, mientras que la lógica de primer orden permite razonar acerca de elementos sobre los que se predica, extendiendola con términos y cuantificadores.
Def: un lenguaje de primer orden $\mathcal{L}$ está dado por un conjunto de símbolos de función $\mathcal{F} = \{f, g, h, ...\}$ y un conjunto de símbolos de predicado $\mathcal{P} = P, Q, R, ...$, cada símbolo y predicado con una aridad asociada $\geq 0$.

Suponemos fijado un lenguaje de primer orden $\mathcal{L}$ y un conjunto infinito numerable de variable $\mathcal{X} = \{X, Y, Z, ...\}$.
Def: el conjunto $\mathcal{T}$ de términos se define por la siguiente gramática: $t ::= X | f(t_1, ..., t_n)$.

Extendemos la gramática de las fórmulas en lógica proposicional y la extendemos a lógica de primer orden.
$\sigma ::= \mathcal{P}(t_1, ..., t_n) | \bot | \sigma \Rightarrow \sigma | \sigma \wedge \sigma | \sigma \vee \sigma | \neg\sigma | \forall X.\sigma | \exists X.\sigma$

Una ocurrencia de una variable $X$ en una fórmula está ligada si está bajo el alcance de un cuantificador $\forall X / \exists X$ y libre si no. Dos fórmulas que sólo difieren en los nombres de las variables ligadas se consideran iguales.

Notamos $\sigma\{X := t\}$ a la sustitución de las ocurrencias libres de $X$ en la fórmula $\sigma$ por el término $t$, evitando la captura de variables.

### 2.5.1. Deducción natural en LPO

Se agregan reglas de introducción y eliminación para $\forall$ y $\exists$.

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

### 2.5.2. Semántica

Suponemos fijado un lenguaje de primer orden $\mathcal{L}$.
Def: una estructura de primer orden es un par $\mathcal{M} = (M, I)$ donde $\mathcal{M}$ es un conjunto no vacío, llamado universo, $I$ es una función que le da una interpretación a cada símbolo, para cada símbolo de función $f$ de aridad $n$, $I(f) : M^n \to M$, y para cada símbolo de predicado $P$ de aridad $n$, $I(P) \subseteq M^n$.

Suponemos fijada una estructura de primer orden $\mathcal{M} = (M, I)$.
Def: una asignación es una funci on que a cada variables le asigna un elemento del universo $a : \mathcal{X} \to M$.
Def: cada término $t \in \mathcal{T}$ se interpreta como un elemento $a(t) \in M$.

Definimos una relación de satisfacción $a \models_{\mathcal{M}} \sigma$, representando que la asignación $a$ bajo la estructura $\mathcal{M}$ satisface la fórmula $\sigma$.

$$\begin{aligned}
&a \models_{\mathcal{M}} P(t_1, ..., t_n) \text{ sii } (a(t_1), ..., a(t_n)) \in I(P)\\
&a \models_{\mathcal{M}} \sigma \land \tau \text{ sii } a \models_{\mathcal{M}} \sigma \text{ y } a \models_{\mathcal{M}} \tau\\
&a \models_{\mathcal{M}} \sigma \lor \tau \text{ sii } a \models_{\mathcal{M}} \sigma \text{ o } a \models_{\mathcal{M}} \tau\\
&a \models_{\mathcal{M}} \sigma \Rightarrow \tau \text{ sii } a \not\models_{\mathcal{M}} \sigma \text{ o } a \models_{\mathcal{M}} \tau\\
&a \models_{\mathcal{M}} \neg\sigma \text{ sii } a \not\models_{\mathcal{M}} \sigma\\
&a \models_{\mathcal{M}} \forall X.\sigma \text{ sii } a[X \mapsto m] \models_{\mathcal{M}} \sigma \text{ para todo } m \in M\\
&a \models_{\mathcal{M}} \forall X.\sigma \text{ sii } a[X \mapsto m] \models_{\mathcal{M}} \sigma \text{ para algún } m \in M
\end{aligned}$$

Def: decimos que una fórmula $\sigma$ es

$$\begin{aligned}
&\text{Válida si } a \models_{\mathcal{M}} \sigma \text{ para toda } \mathcal{M}, a\\
&\text{Inválida si } a \not\models_{\mathcal{M}} \sigma \text{ para alguna } \mathcal{M}, a\\
&\text{Satisfactible si } a \models_{\mathcal{M}} \sigma \text{ para alguna } \mathcal{M}, a\\
&\text{Insatisfactible si } a \models_{\mathcal{M}} \sigma \text{ para toda } \mathcal{M}, a
\end{aligned}$$

Obs: sea $\sigma$ una fórmula,

$$\begin{aligned}
&\sigma \text{ es Válida sii } \sigma \text{ no es Inválida}\\
&\sigma \text{ es Satisfactible sii } \sigma \text{ no es Insatisfactible}\\
&\sigma \text{ es Válida sii } \neg\sigma \text{ es Insatisfactible}\\
&\sigma \text{ es Satisfactible sii } \neg\sigma \text{ es Inválida}
\end{aligned}$$

Una sentencia es una fórmula $\sigma$ sin variables libres.
Una teoría de primer orden es un conjunto de sentencias.
Def: (Consistencia) Una teoría $\mathcal{T}$ es consistente si $\mathcal{T} \nvdash \bot$.
Def: (Modelo) Una estructura $\mathcal{M} = (M, I)$ es un modelo de una teoría $\mathcal{T}$ si vale $\models_{\mathcal{M}} \sigma$ para toda fórmula $\sigma \in \mathcal{T}$ (la asignación es irrelevante pues $\sigma$ es cerrada). Teo: Dada una teoría $\mathcal{T}$, son equivalentes: $\mathcal{T}$ y $\mathcal{T}$ tiene al menos un modelo. Coro: Dada una fórmula $\sigma$, son equivalentes: $\vdash \sigma$ es derivable y $\sigma$ es válida. Coro: Dada una fórmula $\sigma$, son equivalentes: $\vdash \sigma$ es derivable y $\sigma$ es insatisfactible.

*El problema de la decisión.* Dada una fórmula $\sigma$, retornar un booleano que indica si $\sigma$ es válida. No es posible dar una lgoritmo que cumpla dicha especificación.

### 2.5.3. Unifiacción

$$\begin{aligned}
\{X \stackrel{?}{=} X\} \cup E &\xrightarrow{\text{Delete}} E \\
\{f(t_1, ..., t_n) \stackrel{?}{=} f(s_1, ..., s_n)\} \cup E &\xrightarrow{\text{Decompose}} \{t_1 \stackrel{?}{=} s_1, ..., t_n \stackrel{?}{=} s_n\} \cup E \\
\{t \stackrel{?}{=} X\} \cup E &\xrightarrow{\text{Swap}} \{X \stackrel{?}{=} t\} \cup E \text{ si } t \text{ no es una variable} \\
\{X \stackrel{?}{=} t\} \cup E &\xrightarrow{\text{Elim}}_{\{X:=t\}} E\{X := t\} \text{si } X \text{ no ocurre en fv}(t) \\
\{f(t_1, ..., t_n) \stackrel{?}{=} g(s_1, ..., s_m)\} \cup E &\xrightarrow{\text{Clash}} \text{falla si } g \neq g \\
\{X \stackrel{?}{=} t\} \cup E &\xrightarrow{\text{Occurs-Check}} \text{falla si } X \neq t \text{ y } X \in \text{fv}(t)
\end{aligned}$$

## 2.6. Resolución

### 2.6.1. Resolución para lógica proposicional

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

*Método de resolución.* Escribir $\neg\sigma$ como un conjunto $\mathcal{C}$ de cláusulas y luego buscar una refutación de $\mathcal{C}$, es decir, 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, $\sigma$ es válida es válida/tautología. Si no se encuentra una refutación de $\mathcal{C}$ no vale $\neg\sigma \vdash \bot$, es decir, $\neg\sigma$ es satisfactible. Luego no vale $\vdash \sigma$, es decir, $\sigma$ no es válida/tautología.

#### 2.6.1.1. Pasaje a forma clausal

**Paso 1.** deshacerse del "$\Rightarrow$", usando $\sigma \Rightarrow \tau \to \neg\sigma \vee \tau$.
**Paso 2.** empujar el conectivo "$\neg$" hacia adentro, con $\neg(\sigma \wedge \tau) \to \neg\sigma\neg\tau, \neg(\sigma \vee \tau) \to \neg\sigma \wedge \tau$ y $\neg\neg\sigma \to \sigma$.
La fórmula resultante está en forma normal negada (NNF)

$$\sigma_{nnf} ::= P | \neg P | \sigma_{nnf} \wedge \sigma_{nnf} | \sigma_{nnf} \vee \sigma_{nnf}$$

**Paso 3.** distribuir $\vee$ sobre $\wedge$ con $\sigma \vee (\tau \wedge \rho) \to (\sigma \vee \tau) \wedge (\sigma \vee \rho)$ y $\sigma \wedge \tau) \vee \rho \to (\sigma \vee \rho) \vee (\tau \vee \rho)$.
La fórmula resultante está en forma normal conjuntiva (CNF), es una conjunción de

disyunciones de literales.

$$\text{Fórmulas en CNF } \sigma_{\text{cnf}} ::= (k_1 \wedge k_2 \wedge ... \wedge k_n)$$
$$\text{Cláusulas } k ::= (l_1 \wedge l_2 \wedge ... \wedge l_n)$$
$$\text{Literales } l ::= P | \neg P$$

Usando que la disyunción es asociativa, conmutativa e idempotente, notamos una cláusula (disyunción de literales) como un conjunto $\{l_1, l_2, ..., l_n\}$.
Análogamente con la conjunción, notamos una conjunción de cláusulas como un conjunto $\{k_1, k_2...k_n\}$.

**Refutación**. Una vez obtenido un conjunto de cláusulas $\mathcal{C} = \{k_1, ..., k_n\}$, se busca una refutación (demostración de $\mathcal{C} \vdash \bot$).
Regla de resolución

$$\frac{P \vee l_1 \vee ... \vee l_n \quad \neg P \vee l'_1 \vee ... \vee l'_n}{l_1 \vee ... \vee l_n \vee l'_1 \vee ... \vee l'_n}$$

Escrita con notación de cláusulas

$$\frac{\{P, l_1, ..., l_n\} \quad \{\neg P, l'_1, ..., l'_n\}}{\{l_1, ..., l_n, l'_1, ..., l'_n\}}$$

Entrada: conjunto de cláusulas $\mathcal{C}_0 = \{k_1, ..., k_n\}$. Salida: SAT/INSAT indicando si $\mathcal{C}_0$ es insatisfactible ($\mathcal{C}_0 \vdash \bot$).

*Algoritmo de refutación*
Sea $\mathcal{C} := \mathcal{C}_0$, repetir mientras sea posible:

- Si $\{\} \in \mathcal{C}$, devolver INSAT.
- Elegir dos cláusulas $k, k' \in \mathcal{C}$ tales que $k = \{P, l_1, ..., l_n\}$ y $k' = \{\neg P, l'_1, ..., l'_n\}$ y la resolvente $\rho = \{l_1, ..., l_n, l'_1, ..., l'_n\}$ no está en $\mathcal{C}$.
  Si no es posible, devolver SAT.
- Tomar $\mathcal{C} := \mathcal{C} \cup \{\rho\}$ y volver al paso 1.

### 2.6.2. Resolución para lógica de primer orden

Si $\sigma$ es válida, el método siempre termina. Si $\sigma$ no es válida, el método puede no terminar, pues usamos un procedimiento de semi-decisión.

#### 2.6.2.1. Pasaje a forma clausal

**Paso 1.** igual.
**Paso 2.** ahora también usamos $\forall X.\sigma \rightarrow \exists X.\neg\sigma$ y $\neg\exists.X \rightarrow .\neg\sigma$. La fórmula resultante está en forma normal negada (NNF)

$$\sigma_{nnf} ::= P(t_1, ..., t_n) | \neg P(t_1, ..., t_n) | \sigma_{nnf} \wedge \sigma_{nnf} | \sigma_{nnf} \vee \sigma_{nnf} | \forall X.\sigma_{nnf} | \exists X.\sigma_{nnf}$$

**Paso 3.** extraer los cuantificadores hacia afuera, se asume siempre que $X \notin \text{fv}(\tau)$

$$(\forall X.\sigma) \wedge \tau \rightarrow \forall X.(\sigma \wedge \tau) \qquad \tau \wedge (\forall X.\sigma) \rightarrow \forall X.(\tau \wedge \sigma)$$

$$(\forall X.\sigma) \vee \tau \to \forall X.(\sigma \vee \tau) \qquad \tau \vee (\forall X.\sigma) \to \forall X.(\tau \vee \sigma)$$
$$(\exists X.\sigma) \wedge \tau \to \exists X.(\sigma \wedge \tau) \qquad \tau \wedge (\exists X.\sigma) \to \exists X.(\tau \wedge \sigma)$$
$$(\exists X.\sigma) \vee \tau \to \exists X.(\sigma \vee \tau) \qquad \tau \vee (\exists X.\sigma) \to \exists X.(\tau \vee \sigma)$$

Todas las reglas transforman la fórmula en otra equivalente. La fórmula resultante está en forma normal prenexa.

$$\sigma_{\text{pre}} ::= \mathcal{Q}_1 X_1.\mathcal{Q}_2 X_2....\mathcal{Q}_n X_n.\tau$$

donde cada $\mathcal{Q}_i$ es un cuantificador y $\tau$ representa una fórmula en NNF libre de cuantificadores.

**Paso 4.** deshacerse de los cuantificadores existenciales con la técnica de Herbrand y Skolem. Introducimos testigos para los $\exists$ para eliminiarlos sin cambiar la satisfactibilidad. Es decir, todo cuantificador existencial se instancia en una constante o función de skolem. Estas funciones y constantes se suelen conocer como parámetros.
Cada ocurrencia de una subfórmula $\exists x.B$ en A se reemplaza por $B\{x := f(x_1, ..., x_n)\}$. Si $\exists x.B$ forma parte de una fórmula mayor, $x$ solo depende de las variables libres de $B$.
Sea $A$ una sentencia rectificada en forma normal negada, reemplazar sucesivamente cada ocurrencia de una subfórmula de la forma $\exists X.B$ en $A$ por $B\{X := f_X(y_1, ..., y_m)\}$, donde $\text{fv}(B) = \{x, y_1, ..., y_m\}$, como $A$ está rectificada, cada $f_x$ es única. Caso especial $m = 0$, se usa una constante $c_x$. $\exists X.B$ se reemplaza por $B\{X := c_x\}$.
La skolemización no es determinística, es mejor skolemizar de afuera hacia adentro. La skolemización preserva la satisfactibilidad, pero po preserva la validez.
En resumen, se aplica la regla

$$\forall X_1....\forall X_n.\exists Y.\sigma \to \forall X_1....\forall X_n.\sigma\{Y := f(X_1, ..., X_n)\}$$

Forma normal de Skolem:

$$\sigma_{\text{sk}} ::= \forall X_1 X_2...X_n.\tau$$

donde $\tau$ representa una fórmula en NNF libre de cuantificadores.

**Paso 5.** dada una fórmula en forma normal de Skolem $\forall X_1 X_2...X_n.\phi$, con $\phi$ libre de cuantificadores, se pasa $\phi$ a FNC.
**Paso 6.** se empujan los cuantificadores universales hacia adentro, por último se escribe la forma clausal.

Cada paso produce una fórmula equivalente, excepto la Skolemizaci'on que s'olo preserva satisfactibilidad.

*Refutación en lógica de primer orden.* Una vez obtenido un conjunto de cláusulas se busca una refutación.

$$\frac{\{\sigma_1, ..., \sigma_p, l_1, ...l_n\} \quad \{\neg\tau_1, ..., \neg\tau_q, l'_1, ..., l'_m\}}{S(\{l_1, ..., l_n, l'_1, ..., l'_n\})}$$

con $p > 0$, $q > 0$ y $S = \text{mgu}(\{\sigma_1 \overset{?}{=} \sigma_2 \overset{?}{=} ... \overset{?}{=} \sigma_p \overset{?}{=} \tau_1 \overset{?}{=} \tau_2 \overset{?}{=} ... \overset{?}{=} \tau_q\})$.

*Resolución binaria.*

$$\frac{\{\sigma, l_1, ...l_n\} \quad \{\neg\tau, l'_1, ..., l'_m\} \quad S = \text{mgu}(\{\sigma \overset{?}{=} \tau\})}{S(\{l_1, ..., l_n, l'_1, ..., l'_n\})}$$

con $p > 0$ y $q > 0$.

No es completa, por ejemplo $\{\{P(X), P(Y)\}, \{\neg P(Z), \neg P(W)\}\}$ es insatisfactible pero no es posible alcanzar la cláusula vacía con resolución binaria.

Teo: (corrección del pasaje a forma clausal) Dada una fórmula $\sigma$, el pasaje a forma clausal termina y el conjunto de cláusulas obtenido es equisatisfactible a $\sigma$. Es decir, $\sigma$ es sat sii $\mathcal{C}$ es sat.

Teo: (corrección del algoritmo de refutación) Dado un conjunto de cláusulas $\mathcal{C}_0$, si $\mathcal{C}_0 \vdash \bot$, existe una manera de elegir las cláusulas tal que el algoritmo de refutación termina y el algoritmo retorna INSAT sii $\mathcal{C}_0 \vdash \bot$.

# 3. Haskell

## 3.1. Programación funcional

La *programación funcional* consiste en definir funciones y aplicarlas para procesar información.

Las *funciones* son verdaderamente funciones (parciales):

- Aplicarlas no tiene efectos secundarios.
- A una misma entrada corresponde siempre la misma salida (determinismo).
- Las estructuras de datos son inmutables.

Las funciones son datos como cualquier otro, se pueden pasar como parámetros, devolver como resultado y formar parte de estructuras de datos.

Un *programa funcional* está dado por un conjunto de ecuaciones.

### 3.1.1. Gramática

Las *expresiones* son secuencias de símbolos que sirven para representar datod, funciones y funciones aplicadas a los datos. Una expresión puede ser:

1. Un contructor: True False [] (:) 0 1 2 ...
2. Una variable: longitud ordenar x xs (+) (*) ...
3. La aplicación: de una expresión a otra: ordenar lista, not True, ((+) 1) (alCuadrado 5)
4. ...

La aplicación es asociativa a izquierda

$$\begin{aligned} \text{f x y} &\equiv \text{(f x) y} \not\equiv \text{f (x y)} \\ \text{f a b c d} &\equiv \text{(((f a) b) c) d} \end{aligned} \tag{1}$$

### 3.1.2. Tipos

Hay secuencias de símbolos que no son expresiones bien formadas como 1,,2 ó )f x(, y hay expresiones que están bien pormadas pero no tienen sentido, como True + 1, 0 1 y [[], (+)].

Un *tipo* es una especificación del invariante de un dato o de una función. El tipo de una función expresa un *contrato*.

```
99 :: Int
not :: Bool -> Bool
not True :: Bool
((+) 1) 2 :: Int
```

$\rightarrow$es asociativo a derecha.

$$\begin{gathered} \mathrm{a} \rightarrow \mathrm{b} \rightarrow \mathrm{c} \equiv \mathrm{a} \rightarrow (\mathrm{b} \rightarrow \mathrm{c}) \not\equiv (\mathrm{a} \rightarrow \mathrm{b}) \rightarrow \mathrm{c} \\ \mathrm{a} \rightarrow \mathrm{b} \rightarrow \mathrm{c} \rightarrow \mathrm{d} \equiv \mathrm{a} \rightarrow (\mathrm{b} \rightarrow (\mathrm{c} \rightarrow \mathrm{d})) \end{gathered} \tag{2}$$

### 3.1.3. Condiciones de tipado

Para que un programa esté *bien tipado*:

1. todas las expresiones deben tener tipo.
2. cada variable se debe usar siempre con un mismo tipo.
3. los dos lados de una ecuación deben tener el mismo tipo.
4. el argumento de una función debe tener el tipo del dominio.
5. el resultado de una función debe tener el tipo del codominio.

$$\frac{\mathrm{f} :: \mathrm{a} \rightarrow \mathrm{b} \qquad \mathrm{x} :: \mathrm{a}}{\mathrm{f\ x} :: \mathrm{b}} \tag{3}$$

Sólo tienen sentido los programas bien tipados. No es necesario escribir explícitamente los tipos (Inferencia).

### 3.1.4. Polimorfismo

Hay expresiones que tienen más de un tipo, usamos variables de tipo a, b, c para denotar tipos desconocidos:

```
id :: a -> a
[] :: [a]
(:) :: a -> [a] -> [a]
fst :: (a,b) -> a
flip :: (a -> b -> c) -> b -> a -> c
```

### 3.1.5. Modelo de cómputo

Dada una expresión, se computa su valor usando las ecuaciones.
Hay expresiones bien tipadas que no tienen valor como 1 / 0. Decimos que se indefinen o tienen el valor $\perp$.
Un programa funcional está dado por un conjunto de *ecuaciones orientadas*. Ena ecuación e1 = e2 se interpreta desde dos puntos de vista:
Def: Punto de vista denotacional: declara que e1 y e2 tienen el mismo significado.
Def: Punto de vista operacional: computar el valor de e1 se reduce a computar el valor de e2.
El lado izquiedo de una ecuación no es una expresión arbitraria, debe ser una función

aplicada a *patrones*. Un patrón puede ser una variable, un comodín ó un consstructor aplicado a patrones. Este no debe contener variables repetidas.

Evaluar una expresión consiste en:

1. buscar la subexpresión más externa que coincida con el lado izquerdo de una ecuación.
2. reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
3. continuar evaluendo la expresión resultante.

La evaluación se detiene cuando se da uno de los siguientes casos:

1. la expresión en un constructor o un constructor aplicado. True, (:) 1, [1,2,3].
2. la expresión es una función parcialmente aplicada. (+), (+) 5.
3. se alcanza un estado de error, es decir, una expresión que no coincide con las ecuaciones que definen a la función aplicada.

Ejemplos de evaluaciones y resultados

1. constructor: tail (tail [1,2,3]) $\rightsquigarrow$tail [2,3] $\rightsquigarrow$[3]
2. función parcialmente aplicada: const (const 1) 2 $\rightsquigarrow$const 1
3. error: head (head [[], [1], [1,1]]) $\rightsquigarrow$head [] $\rightsquigarrow\perp$
4. no terminación: loop n = loop (n + 1), loop 0 $\rightsquigarrow$loop (1 + 0) $\rightsquigarrow$loop(1 + (1 + 0)) $\rightsquigarrow$loop (1 + (1 + (1 + 0))) $\rightsquigarrow$...
5. evalucación no estricta: indefinido = indefinido, head (tail [indefinido, 1, indefinido]) $\rightsquigarrow$head [1, indefinido] $\rightsquigarrow$1
6. listas infinitas: desde n = n : desde (n + 1), desde 0 $\rightsquigarrow$0 : desde 1 $\rightsquigarrow$0 : (1 : desde 2) $\rightsquigarrow$0 : (1 : (2 : desde 3)) $\rightsquigarrow$...
7. listas infinitas 2: head (tail (desde 0)) $\rightsquigarrow$head (tail (0 : desde 1)) $\rightsquigarrow$head (desde 1) $\rightsquigarrow$head (1 : desde 2) $\rightsquigarrow$1

<u>Obs:</u> en Haskell, el orden de las ecuaciones es relevante, si hay varias ecuaciones que coinciden siempre se usa la primera.

### 3.1.6. Listas

El tipo [a] denota listas de elementos de tipo a. Por ejemplo, [1,2,3] abrevia a 1:2:3:[], (:) :: a ->[a] ->[a] es el operador cons, que es el constructor de listas, asocia a derecha y no tiene una definición asociadia ya que es un constructor, luego la expresión no se puede simplificar más.
Las listas toman una de las siguientes formas. Una lista indefinida, undefined :: [a], una lista vacía [] :: [a] y una lista de la forma x:xs donde x :: a y xs :: [a].
Luego hay tres tipos de listas. Las listas finitas como 1:2:3:[], las listas parciales como 1:2:3:undefined ó filter (<4) [1..] $\rightsquigarrow$1:2:3:undefined y las listas infinitas como [1..].

```haskell
iterate :: (a -> a) -> a -> [a]
iterate f x = x:iterate f (f x)
-- ghci> iterate (+1) 1
```

```haskell
-- ghci> divisors n = filter (\m -> n `mod` m == 0) [1..(n-1)]
-- ghci> divisors 6
-- [1,2,3]
-- ghci> head (filter (\n -> n == sum (divisors n)) [1..])
-- 6

until p f = head . filter p . iterate f
-- ghci> until (\n -> n == sum (divisors n)) (+1) 1
-- 6
```

Las listas pueden ser enumeradas por la clase Enum, por ejemplo:

```haskell
[m..n] ⇝ [m,m+1,..,n]
[m..] ⇝ [m,m+1,..]
[m,n..p] ⇝ [m,m+(n-m),m+2(n-m),..,p]
[m,n..] ⇝ [m,m+(n-m),m+2(n-m),..]
['a'..'z'] ⇝ "abcdefghijklmnopqrstuvwxyz"
```

Las listas también pueden ser definidas por comprensión:

```haskell
[x*x | x <- [1..5]] ⇝ [1,4,9,16,25]
[(i,j) | i <- [1..5], even i, j <- [i..5]]
  ⇝ [(2,2),(2,3),(2,4),(2,5),(4,4),(4,5)]
[x | xs <- [[(3,4)],[(5,4),(3,2)]], (3,x) <- xs] ⇝ [4,2]

divisors x = [d | d <- [2..x-1], x `mod` d == 0]

disjoint [] _ = True
disjoint (x:xs) ys = not (x `elem` ys) && disjoint xs ys

coprime x y = disjojint (divisors x) (divisors y)

triads n = [(x,y,z) | x <- [1..m], y <- [x+1..n],
                      coprime x y,
                      z <- [y+1..n], x*x+y*y==z*z]
           where m = floor (fromIntegral n / sqrt 2) -- optimizacion aprovechando
                                                     -- que 2x^2 < x^2 + y^2 = z^2 ≤ n^2, luego x < ⌊n/√2⌋

map f xs = [f x | x <- xs]
filter p xs = [x | x <- xs, p x]
concat xss = [x | xs <- xss, x <- xs]
```

En realidad Haskell hace lo contrario, evalua las listas por comprensión en términos de map y concat, sus reglas de traducción son:

```haskell
[e | True]        = [e]
[q | q]           = [e | q, True]
[e | b, Q]        = if b then [e | Q] else []
[e | p <- xs, Q]  = let ok p = [e | Q]
                        ok _ = []
                    in concat (map ok xs)
```

```
[e | Q1, Q2] = concat [[e | Q2] | Q1]
```

La definición de ok usa un patrón don't care, o un wild card. Dice que la lista vacia se retorna para cualquier elemento que no une con el patrón p.

Podemos definir funciones sobre listas con patterrn matching y sabemos que [] y x:xs son disjuntos y exhaustivos. También podemos usar el don't care pattern.

```
null :: [a] -> Bool
null [] = True
null (x:xs) = False

null2 :: [a] -> Bool
null2 [] = True
null2 _ = False

head :: [a] -> a
head (x:xs) = x

head :: [a] -> [a]
head (x:xs) = xs

-- en last el orden importa!
last :: [a] -> a
last [x] = x -- [x] ≡ (x:_)
last (_:xs) = last xs
```

### 3.1.7. Funciones de orden superior

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(g . f) x = g (f x)
-- o bien (g . f) = \x -> g (f x), con notación lambda

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

dobleL = map (*2)
longitudL = map length

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

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

uncurry :: (a -> b -> c) -> (a, b) -> c
```

```haskell
uncurry f (x, y) = f x y
```

Prop: Sean `suma x y = x + y`, `suma' (x, y) = x + y`,

- `suma = curry suma'`
- `suma' = uncurry suma`

Obs: Por notación lambda, una expresión de la forma `\ x -> e` representa una función que recibe un parámetro x y devuelve e.

`(\ x1 x2 ... xn -> e)` ≡ `(\ x1 -> (\ x2 -> ... (\ xn -> e)))`

### 3.1.8. Más funciones y propiedades

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

*Ejemplo* de secuencia de evaluación

```haskell
[1,2] ++ [3,4,5]
= (1:(2:[])) ++ (3:(4:(5:[])))
= 1:((2:[]) ++ (3:(4:(5:[]))))
= 1:(2:([] ++ (3:(4:(5:[])))))
= 1:(2:(3:(4:(5:[]))))
= [1,2,3,4,5]
```

La concatenación es asociativa, ¿Pero esta implementación la hace conmutativa?

```haskell
undefined ++ [1,2] = undefined
[1,2] ++ undefined = 1:2:undefined
```

Listaré propiedades que pueden ser probadas con razonamineto ecuacional:

```haskell
filter p = concat . map (\x -> if p x then [x] else [])

-- estas dos se llamas functor laws of map, nombre prestado de Teoría de Categorías
map id = id
map (f . g) = map f . map g

-- Haskell provee una clase de tipos Function cuya definición es
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- ahora que lo generalizé puedo aplicarlo a otras estructuras
data Tree a = Tip a | Fork (Tree a) (Tree a)
instance Functor Tree whee
  fmap f (Tip x) = Tip (f x)
  fmap f (Fork u v) = Fork (fmap f u) (fmap f v)

-- En realidad map es un sinónimo para la instancia fmap para listas
map(+1) [2,3,4] ≡ fmap (+1) [2,3,4]

-- Para operaciones que no dependen de la naturaleza de los elementos de la lista.
```

```haskell
-- Son funciones que mezclan, descartan o extraen elementos de listas.
-- funciones con tipos polimorficos satisfacen alguna.
-- ley que se pueden cambiar valores antes de aplicar la función.
-- En matemáticas se llaman transformaciones naturales y
-- las leyes asociadas leyes de naturalidad.
f . head = head . map f -- sólo si f es estricta (mirar caso [])
map f . tail = tail . map f
map f . concat = concat . map (map f)

-- otro ejemplo
map f . reverse = reverse . map f

-- mas propiedades
concat . map concat = concat . concat
filter p . map f = map f . filter (p . f)
```

Las siguientes son las definiciones de zip y zipWith

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y): zip xs ys
zip _ _ = []

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _ = []

-- zip se puede definir con zipWith y el constructor de pares
zip ≡ zipWith (,)

nondec :: (Ord a) => [a] -> Bool
nondec xs = and (zipWith (<=) xs (tail xs))

position :: (Eq a) => a -> [a] -> Int
position x xs = head ([j | (j,y) <- zip [0..] xs, y==x] ++ [-1])

sort :: (Ord a) => [a] -> [a]
sort [] = []
sort [x] = [x]
sort xs = merge (sort ys) (sort zs)
  where (ys,zs) = halve xs

halve xs = (take n xs, drop n xs)
  where n = length xs `div` 2

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y = x:merge xs (y:ys)
```

```haskell
  | otherwise = y:merge (x:xs) ys

-- esta ultima linea se podria escribir asi
merge xs'(x:xs) ys'(y:ys)
  | x <= y = x:merge xs ys'
  | otherwise = y:merge xs' ys

enumerarPares :: (Num a) -> [(a,a)]
enumerarPares = [(x,y) | n <- [0..], x <- [0..n], y <- [n-x]]
```

Obs: Haskell define la comparación <= en pares como

```haskell
(x1,y1) <= (x2,y2) = (x1,x2) || (x1 == x2 && y1 <= y2)
```

### 3.1.9. Tipos de datos algebraicos

Conocemos algunos tipos de datos "primitivos", como Char, Int, Float, (a ->b), (a, b), [a], String (sinónimo de [Char]).

Se pueden definir nuevos tipos de datos con la cláusula data `data Tipo =` <declaración de los constructores>

Por ejemplo, `data Dia = Dom | Lun | ... | Sab` declara que existen constructores `Dom :: Dia`, `Lun :: Dia`, ..., `Sab :: Dia` y que son los únicos constructores del tipo Dia.

```haskell
esFinDeSemana :: Dia -> Bool
esFinDeSemana Sab = True
esFinDeSemana Dom = True
esFinDeSemana _ = False
```

Un constructor puede tener muchos parámetros, por ejemplo, `data Persona = LaPersona String Stri` declara que el único constructor de persona es `LaPersona :: String -> String -> Int -> Persona`.

```haskell
nombre :: Persona -> String
nombre (LaPersona n _ _) = n
```

Un tipo puede tener muchos constructores,

```haskell
data Forma = Rectangulo Float Float
           | Circulo Float
```

declara que el tipo Forma tiene dos constructores y sólo esos.

Los constructores pueden ser recursivos,

```haskell
data Nat = Zero
         | Succ Nat

-- valores de la forma
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
```

```haskell
...

doble :: Nat -> Nat
doble Zero = Zero
doble (Succ n) = Succ (Succ (doble n))

-- Haskell permite trabajar con estructurAas infinitas
-- porque las definiciones recursivas se interpretan de manera
-- coinductiva en lugar de inductiva
infinito :: Nat
infinito = Succ infinito
```

Forma general de tipos algebráicos:

```
data T = CBase₁ <parámetros>
           ...
       | CBaseₙ <parámetros>
       | CRecursivo₁ <parámetros>
           ...
       | CRecursivoₙ <parámetros>
```

Los constructores base no reciben parámetros de tipo T y los recursivos reciben al menos uno. Los valores de tipo T son los que se pueden construir aplicando constructores base y recursivos un número finito de veces y sólo esos, es decir, entendemos la definición de T de forma inductiva.

*Ejemplo* `data Lista a = Vacia | Cons a (Lista a)`

*Ejemplo* `data AB a = Nil | Bin (AB a) a (AB a)`. Definir preorder, postorder e inorder.

```haskell
data AB a = Nil | Bin (AB a) a (AB a)

preorder :: AB a -> [a]
preorder Nil = []
preorder (Bin i v d) = v : (preorder i ++ preorder d)

inorder :: AB a -> [a]
inorder Nil = []
inorder (Bin i v d) = inorder i ++ (v : inorder d)

postorder :: AB a -> [a]
postorder Nil = []
postorder (Bin i v d) = postorder i ++ postorder d ++ [v]
```

## 3.2. Esquemas de Recursion

### 3.2.1. Sobre Listas

#### 3.2.1.1. Recursión estructural

Sea `g :: [a] -> b` definida por dos ecuaciones:

```haskell
g []     = <caso base>
g (x:xs) = < caso recursivo>
```

g está dada por recursión estructural si:

- El caso base devuelve un valor fijo z.
- El caso recursivo se escribe usando (cero, una o muchas veces) x y (g xs), pero sin usar el valor de xs ni de otros llamados recursivos.

*Ejemplo* suma, (++), isort están dadas por recursión estructural, pero ssort no.

```haskell
-- dada por recursion estructural
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insertar x (isort xs)

-- no dada por recursion estructural
ssort :: Ord a => [a] -> [a]
ssort [] = []
ssort (x:xs) = minimo (x:xs)
             : ssort (sacarMinimo (x:xs))
```

foldr abstrae el esquema de recursión estructural

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

toda recursión estructural es una instancia de foldr.

*Ejemplo*

```haskell
suma = foldr (+) 0

suma [1, 2]
  ⇝ foldr (+) 0 [1,2]
  ⇝ (+) 1 (foldr (+) 0 [2])
  ⇝ (+) 1 ((+) 2 (foldr (+) 0 []))
  ⇝ (+) 1 ((+) 2 0)
  ⇝* 3
```

*Ejemplo*

```haskell
reverse = foldr (\x rec -> rec ++ [x]) []
  ≡ foldr (\x rec -> (++) rec [x]) []
  ≡ foldr (\x rec -> flip (++) [x] rec) []
  ≡ foldr (\x -> flip (++) [x]) []
  ≡ foldr (\x -> flip ((++) (:[]) x)) []
  ≡ foldr (flip (++) . (:[])) []
```

Prop:

- `foldr (:) [] = id`
- `foldr ((:) . f) [] = map f`
- `foldr (const (+ 1)) = [] = length`

*Ejemplo* Casos degenerados

```
length [] = 0
length (_:xs) = 1 + length xs -- no usa la cabeza

head [] = error "No tiene cabeza."
head (x:_) = x -- no usa el llamado recursivo sobre la cola
```

#### 3.2.1.2. Propiedad universal de fold

Al igual que el operador fold encapsula un patrón simple de recursión para procesar listas, la propiedad universal de fold encapsula un patrón simple de prueba inductiva para listas.

Prop: universalidad de fold

```
g [] = v                    ⇔ g = fold f v
g (x:xs) = f x (g xs)
```

*Ejemplo* universalidad como principio de prueba
Queremos ver que `(+1) . sum = fold (+) 1`. Primero notamos que satisface el lado derecho de la propiedad universal de fold, con `g = (+1) . sum`, `f = (X)` y `v = 1`. Luego, por la propiedad universal, concluimos que la ecuación a ser probada es equivalente a las dos siguientes ecuaciones:

```
((+1) . sum) [] = 1
((+1) . sum) (x:xs) = (+) x (((+1) . sum) xs)
```

Luego simplificando

```
sum [] + 1 = 1
sum (x:xs) + 1 = x + (sum xs + 1)
```

Ahora, se puede calcularlo sólo con definiciones

```
sum [] + 1
  = 0 + 1       {Definicion de sum}
  = 1           {Aritmética}
sum (x:xs) + 1
  = (x + sum xs) + 1 {Definicion de sum}
  = x + (sum xs + 1) {Asociatividad de la suma}
```

Y queda probado sin uso explícito de inducción. □

Con la propiedad de universalidad de fold se pueden probar propiedades más específicas, como la propiedad de fusión de fold. Esta puede ser un poco más simple en los casos que se puede aplicar, pero siempre que se puede hacer una prueba for fusión se puede hacer por universalidad.

*Ejemplo* universalidad como principio de definición.
Sea la función sum sum definida por recursión estructural de la siguiente forma:

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

Queremos definir sum con fold. Luego queremos resolver la ecuación `sum = fold f v` para una función f y un valor v. Observamos que la ecuación une con el lado derecho de la propiedad universal, así que concluimos que la ecuación es equivalente a las siguientes ecuaciones:

```
sum [] = v
sum (x:xs) = f x (sum xs)
```

De la primera ecuación y la def de sum, v = 0. De la segunda ecuación calculamos f

```
sum (x:xs) = f x (sum xs)
⇔ x + sum xs = f x (sum xs) {Definición de sum}
⇐ x + y = f x y             {Generalizando (sum xs) a y}
⇔ f = (+)                   {Funciones}
```

Luego, usando la propiedad universal obtuvimos que `sum = fold (+) 0`. □

#### 3.2.1.3. Recursión primitiva

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g []     = <caso base>
g (x:xs) = <caso recursivo>
```

g está dada por recursión primitiva si:

- El caso base devuelve un valor fijo z.
- El caso recursivo se escribe usando (cero, una o muchas veces) x, (g xs) y también xs, pero sin hacer otros llamados recursivos.

Es decir, es similar a la recursión estructural, pero permite referirse a xs.

Obs: Todas las definiciones dadas por recursión estructural también están dadas por recursión primitiva.
Obs: Hay definiciones dadas por recursión primitiva que no están dadas por recursión estructural.
Es decir, estructural $\Rightarrow$ primitiva, pero primitiva $\not\Rightarrow$ estructural.

*Ejemplo* trim se puede definir con recursión primitiva, pero no con recursión estructural.

recr abstrae el esquema de recursión 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)
```

toda recursión primitiva es una instancia de recr.

*Ejemplo*

```
trim = recr (\ x xs rec -> if x == ' ' then rec else x:xs) []
```

#### 3.2.1.4. Recursión iterativa

Sea `g :: [a] -> b` definida por dos ecuaciones:

```
g ac []     = <caso base>
g ac (x:xs) = < caso recursivo>
```

g está dada por recursión iterativa si:

- El caso base devuelve el acumulador ac.
- El caso recursivo invoca inmediatamente a (g ac' xs) donde ac' es el acumulador acutualizado en función de su valor anterior y el valor de x.

Es decir, es similar a la recursión estructural, pero permite referirse a xs.

*Ejemplo*

```
reverse :: [a] -> [a] -> [a]
reverse ac [] = ac
reverse ac (x:xs) = reverse (x:ac) xs
```

foldl abstrae el esquema de recursión iterativa

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

toda recursión iterativa es una instancia de foldl.

En general foldr y foldl tienen comportamiento diferentes:

```
foldr (⊕) z [a,b,c] = a ⊕ (b ⊕ (c ⊕ z))
foldl (⊕) z [a,b,c] = ((z ⊕ a) ⊕ b) ⊕ c
```

si $\oplus$ es un operador asociativo y conmutativo, foldr y foldl definen la misma función.

*Ejemplo*

```
suma = foldr (+) 0 = foldl (+) 0
producto = foldr (*) 1 = foldl (*) 1
and = foldr (&&) True = foldl (&&) True
or = foldr (||) False = foldl (||) False
```

*Ejemplo*

```
tern2dec :: [Int] -> Int
tern2dec = foldl (\ ac b -> b + 3 * ac) 0

tern2dec [2,1,1]
  ⇝ foldl (\ ac b -> b + 3 * ac) 0 [2,1,1]
  ⇝ foldl (\ ac b -> b + 3 * ac) (2 + 3 * 0) [2,1]
  ⇝ foldl (\ ac b -> b + 3 * ac) (1 + 3 * (2 + 3 * 0)) [1]
  ⇝ foldl (\ ac b -> b + 3 * ac) (1 + 3 * (1 + 3 * (2 + 3 * 0))) []
  ⇝ 1 + 3 * (1 + 3 * (2 + 3 * 0))
  ⇝* 22
```

La función foldl es un operador de iteración. Pseudocódigo imperativo:

```
función foldl f ac xs
  mientras xs no es vacía
    ac := f ac (head xs)
```

```
    xs := tail xs
  devolver ac
```

Se puede demostrar que `foldl (flip (:)) [] = reverse`.

#### 3.2.1.5. Ejercicios

*Ejercicio* Definir la siguiente función usando foldr.

```
zip :: [a] -> [b] -> [(a,b)]
zip [] [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Primero mostraré que zip está definida sii los largos de las dos listas son iguales.

Demo: Sean $a, b$ tipos, $A :: [a], B :: [b]$ listas, $n, m \in \mathbb{Z}$ las longitudes de las listas $A$ y $B$ respectivamente. Quiero ver que `zip A B` está definida si y sólo si $n = m$ y no hay elementos indefinidos en las listas.

Si hay algun elemento indefinido en alguna de las listas claramente en algún momento de la recursión ese elemento será $x$ ó $y$ si $n \neq m$, luego la expresión se indefinirá ó ocurrirá que $n = m$. Luego me reduzco a ver que zip está definida sii $n = m$.

Para la demostración usaré como invariante que a través de las llamadas recursivas de zip, la igualdad/desigualdad de $n$ y $m$ se preserva. En el caso base se ve que no hay llamadas recursivas. En el caso recursivo sólo se llama a `zip xs ys`, es decir, los nuevos valores de $n$ y $m$ son $n - 1$ e $y - 1$, luego como $n = m \Leftrightarrow n - 1 = m - 1$ se mantiene el invariante.

Si $n = m$. Por inducción en n,

Caso base. Si $n = m = 0$ luego $A$ y $B$ son listas con cero elementos y por lo tanto `A = []` y `B = []`. Luego `zip A B = zip [] [] = []` por la primera regla y entonces está definida.

Caso inductivo. Si $n = m > 0$, luego $A$ y $B$ son listas con $n > 0$ elementos, luego se usa la segunda ecuación de zip, que está definida si y sólo si `(x, y) : zip xs ys` está definida. Claramente `(x, y)` está definido, pues $x$ e $y$ están definidos. Además `zip xs ys` está definido por hipótesis inductiva. Luego la expresión entera está definida.

Luego por inducción en $n$, $\forall k$, si ambas listas tienen tamaño $k$ la función está definida.

Si $n \neq m$, por invariante se mantiene la desigualdad de las llamadas recursivas. Además el tamaño de ambas listas se reduce en uno en cada llamado recursivo. Luego eventualmente alguna de las listas se intentará llamar con tamaño 0 y la otra con tamaño mayor a 0, por lo cual no habrá definición de la función.

Queda probado que `zip A B` está definida si y sólo si $n = m$ y los elementos de ambas listas están definidos.

$\square$

Quisiera escribir algo de la forma `zip xs ys = foldr f z xs ys`, con f siendo la función que hay que definir y z el caso base, para poder reescribirlo como `zip = foldr f z`. Si hago esto, estoy evaluando el foldr con xs, y esto debería devolver una función que al evaluarla con ys me de el resultado de zip. Es decir, `foldr f z xs :: [b] -> [(a, b)]`. Entonces `z :: [b] -> [(a,b)]`, así que defino z como `(\ _ -> [])` definiendo el caso base de la función que retornaré. Ahora quiero en cada paso de la recursión agregar un caso a la función. Luego defino a f como `(\ x rec (y:ys) -> (x, y) : rec ys)`.

Luego tengo
`zip xs ys = foldr (\ x rec -> (y:ys) -> (x, y) : rec ys) (_ -> []) xs ys`

Que es equivalente a
`zip = foldr (\ x rec (y:ys) -> (x, y) : rec ys) (_ -> [])`

*Ejercicio* Definir foldr en términos de recr.

Recordemos que todas las definiciones dadas por recursión estructural también están dadas por recursión primitiva. La única diferencia entre recr y foldr es que en la función que se pasa se recibe el parámetro xs, solo debemos ignorarlo.

Ahora, `foldr f z xs = recr (\ x xs rec -> f x rec) z xs`.
Esto es equivalente a `foldr f = recr (\ x _ -> f x)`.

*Ejercicio* Definir recr en términos de recr.

Recordemos que hay definiciones dadas por recursión primitiva que no están dadas por recursión estructural. Esto no significa que el ejercicio sea imposible, sino que tenemos que modificarla definición para que siempre sea posible. Esto lo haremos devolviendo una tupla con una copia de la lista original en el caso base de la recursión, así toda la lista es accesible en todas las ejecuciones de la función a través de la tupla rec que renombramos (xs,rec) ya que contiene la lista original ademas el llamado recursivo que antes devolvíamos.

```
recr f z xs = foldr (\ x (xs,rec) -> ??) (z,xs) xs
```

Ahora nos falta devolver el valor recursivo, es decir, `f x xs rec`, pero no hay que olvidarse de preservar la tupla (xs,rec).

```
recr f z xs = foldr (\ x (xs,rec) -> (xs, f x xs rec)) (z,xs) xs
```

El problema que tenemos ahora es que la función recr devolvería la tupla (xs,resultado), y solo queremos el resultado, así que usamos la función snd para obtener el segundo valor de la tupla.

```
recr f z xs = snd (foldr (\ x (xs,rec) -> (xs, f x xs rec)) (z,xs) xs)
```

El último problema que tenemos es que en xs tenemos siempre toda la lista, pero en realidad queremos que tenga los elementos siguientes, no todos. Entonces lo que hacemos es en vez de pasar toda la lista en el caso base, pasamos una lista vacía, y en cada paso de la recursión agregamos el elemento actual, así en el backtracking iremos teniendo una cada vez un elemento más.

```
recr f z xs = snd (foldr (\x((x:xs),rec) -> (xs, f x xs rec)) ([], z) xs)
```

*Ejercicio* definir foldl en términos de foldr.

Lo único que cambia entre foldl y foldr es el orden de los términos.

```
foldl f z xs = foldr (flip f) z (reverse xs)
```

Y reverse sabemos que se puede escribir en términos de foldr.

*Ejercicio* definir foldr en términos de foldl.

```
foldr f z xs = foldl (flip f) z (reverse xs)
```

Y reverse sabemos que se puede escribir en términos de foldl.

### 3.2.2. Sobre Otras Estructuras

La recursión estructural se generaliza a tipos algebraicos en general. Supongamos que T es un tipo algebráico.
Dada una función `g :: T -> Y` definida por ecuaciones:

```
g (CBase₁ <parámetros>) = <caso base₁>
...
g (CBaseₙ <parámetros>) = <caso baseₙ>
g (CRecursivo₁ <parámetros>) = <caso recursivo₁>
...
g (CRecursivoₙ <parámetros>) = <caso recursivoₙ>
```

Decimos que g está dada por recursión estructural si:

- Cada caso base se escribe combinando los parámetros.
- Cada caso recursivo se escribe combinando los parámetros del constructor que no son de tipo T y el llamado recursivo sobre cada parámetro de tipo T, pero sin unar los parámetros del constructor que son de tipo T y sin hacer otros llamados recursivos.

*Ejemplo* `data AB a = Nil | Bin (AB a) a (AB a)`

```
foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
foldAB cNil cBin Nil = cNil
foldAB cNil cBin (Bin i r d) =
  cBin (foldAB cNil cBin i) r (foldAB cNil cBin d)

idAB :: AB a -> AB a
idAB = foldAB Nil Bin

mapAB :: (a -> b) -> AB a -> AB b
mapAB f = foldAB Nil (\i r d -> Bin i (f r) d)

maximoAB :: AB a -> Maybe a
maximoAB = foldAB Nothing (\i r d -> maxMaybe i (maxMaybe r d))
  where maxMaybe Nothing Nothing = Nothing
        maxMaybe (Just x) Nothing = x
        maxMaybe Nothing (Just y) = y
        maxMaybe (Just x) (Just y) = max x y

altura :: AB a -> Int
altura = foldAB (const 0) (\i _ d -> 1 + max i d)

-- Se puede definir ordenado :: AB a -> Bool retornado
-- en cada paso recursivo un booleano ordenado, el valor
-- minimo y el valor maximo. Luego en ordenado retorno la
-- primera componente de la tripla.
```

```
-- Se puede definir directamente caminoMasLargo :: AB a -> [a]
-- con una idea similar retornando tanto la rama mas larga
-- como el camino mas largo, y eligiendo la rama como la mejor
-- de las dos y el camino como el mejor de los dos caminos y unir las ramas.
```

## 3.3. Razonamiento Ecuacional

Para razonar sobre equivalencia de expresiones vamos a asumir:

- Que trabajamos con estructuras de datos finitas. Específicamente con tipos de datos inductivos.
- Que trabajamos con funciones totales, es decir, las ecuaciones deben cumplir todos los casos y la recursión siempre debe terminar.
- El programa no depende del orden de las ecuaciones.

### 3.3.1. Reemplazo

*Principio de reemplazo.* Sea e1 = e2 una ecuación incluida en el programa. Las siguientes operaciones preservan la igualdad de expresiones:

- Reemplazar cualquier instaincia de r1 por e2.
- Reemplazar cualquier instancia de e2 por e1.

Si podemos demostrar una igualdad usando sólo el principio de reemplazo decimos que la igualdad vale *por definición.*

*Ejemplo* `length ["a", "b"] = suma [1, 1]`

$\{L0\}$ `length [] = 0`
$\{L1\}$ `length (_:xs) = 1 + length xs`
$\{S0\}$ `suma [] = 0`
$\{S1\}$ `suma (x:xs) = x + suma xs`

```
length ["a", "b"]
= 1 + length ["b"]     por L1
= 1 + (1 + length []) por L1
= 1 + (1 + 0)          por L0
= 1 + (1 + suma [])    por S0
= 1 + suma [1]         por S1
= suma [1, 1]          por S1
```

### 3.3.2. Inducción sobre booleanos

*Principio de indución sobre booleanos.*
Si $P(True)$ y $P(False)$ entonces $\forall x :: Bool.P(x)$.

*Ejemplo* probar que ∀ `x :: Bool. not (not x) = x`

$\{NT\}$ `not True = False`
$\{NF\}$ `not False = True`

Por el principio de inducción sobre booleanos, basta probar que `not (not True) = True` y que `not (not False) = False`, los cuales pueden ser probados por definición.

### 3.3.3. Inducción sobre pares

*Principio de indución sobre pares.*
Si ∀`x :: a`. ∀`y :: b`. Si `P((x, y))` entonces ∀`p :: (a, b). P(p)`.

### 3.3.4. Inducción sobre naturales

*Principio de indución sobre naturales.*
`data Nat = Zero | Suc Nat`
Si `P(Zero)` y ∀`n :: Nat`. `(P(n)` ⇒ `P (suc n))`, entonces ∀`n :: Nat`. `P(n)`.

### 3.3.5. Inducción estructural

*Principio de indución estructural.*
Sea P una propiedad acerca de las expresiones tipo T tal que vale sobre todos los constructores base de T y vale sobre todos los constructores recursivos de T asumiendo como hipótesis inductiva que vale para los parámetros de tipo T, entonces ∀`x :: T`. `P(x)`.

*Ejemplo* `data [a] = [] | a : [a]`
Sea P una propiedad sobre expresiones de tipo `[a]` tal que `P([])` y ∀`x :: a`. ∀`xs :: [a]`. `(P(xs)` ⇒ `P(x:xs))`, entonces ∀`xs :: [a]. P(xs)`.

#### 3.3.5.1. Lemas de generación

*Lema de generación para pares.*
Si `p :: (a, b)`, entonces ∃ `x :: a`. ∃ `y :: b. p = (x, y)`.

*Principio de indución sumas.*
`data Either a b = Left a | Right b`
Si `e :: Either a b`, entonces o bien ∃ `x :: a. e = Left x` ó ∃ `y :: b. e = Right y`.

### 3.3.6. Extensionalidad

Punto de vista intensional. Dos valores son iguales si están construidos de la misma manera.

Punto de vista extensional. Dos valores son iguales si son indistinguibles al observarlos.

*Principio de extensionalidad funcional.*
Si (∀`x :: a`. `f x = g x`) entonces `f = g`.

*Corrección con respecto a observaciones* Si demostramos eq = e2 :: A, entonces: obs e1 ⇝True sii obs e2 ⇝True para toda posible observacion obs :: A ->Bool.

*Desigualdades* Hay que buscar una observación que las distinga.
*Ejemplo* Demostrar que no vale `id = swap`

```
obs :: ((Int, Int) -> (Int, Int)) -> Bool
obs f = fst (f (1, 2)) == 1
```

```
obs id ⇝ True
obs swap ⇝ False
```

### 3.3.7. Isomorfismos de tipos

Dos tipos son isomorfo si representan la misma información, pero escrita de otra manera. Entonces podemos transformar los valores de un tipo en valores de otro, como una biyección.
Def: decimos que dos tipos de datos A y B son isomorfos si:

1. Hay una funcion `f :: A -> B` total.
2. Hay una funcion `g :: B -> A` total.
3. Se puede domostrar que `g . f = id :: A -> A`.
4. Se puede domostrar que `f . g = id :: B -> B`.

Escribimos `A` $\simeq$ `B`.

## 3.4. Calculo lambda

Lenguaje de programación definido de manera rigurosa. Se basa en las operaciones de construir funciones y aplicarlas.

### 3.4.1. Cálculo-$\lambda^b$

*Sintaxis de los tipos*

$$\begin{aligned}\tau, \sigma, \rho ::= &\ \text{bool}\\ &\mid \tau \to \sigma\end{aligned}$$

*Sintaxis de los términos* Suponemos un conjunto infinito numerable de variables $\mathcal{X} = \{x, y, z, ...\}$

$$\begin{aligned}M, N, P, ... ::= &\ x && \text{variable}\\ &\mid \lambda x : \tau.M && \text{abstracción}\\ &\mid MN && \text{aplicación}\\ &\mid \text{true} && \text{verdadero}\\ &\mid \text{false} && \text{falso}\\ &\mid \text{if } M \text{ then } N \text{ else } P && \text{condicional}\end{aligned}$$

Una ocurrencia de $x$ esta ligada si aparece adentro de una abstracción $\lambda x$. Una ocurrencia está libre si no está ligada.

Los términos que sólo difieren en el nombre de variables ligadas se consideran iguales por $\alpha$-equivalencia.

La noción de tipabilidad se formaliza con un sistema deductivo.

Un contexto de tipado es un conjunto finito de pares $(x_i : \tau_i)$: $\{x_1 : \tau_1, ..., x_n : \tau_n\}$ sin variables repetidas, se puede notar como como $\Gamma, \Delta, ..., \text{dom}(\Gamma)$.

El sistema de tipos predica sobre juicios de tipado de la forma $\Gamma \vdash M : \tau$.

*Reglas de tipado*

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\text{T-TRUE} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\text{T-FALSE}$$

$$\frac{\Gamma \vdash M : \mathsf{bool} \quad \Gamma \vdash N : \tau \quad \Gamma \vdash P : \tau}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : \tau}\text{T-IF}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\text{T-VAR} \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau . M : \tau \to \sigma}\text{T-ABS}$$

$$\frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\, N : \sigma}\text{T-APP}$$

<u>Teo:</u> (Unicidad de tipos)
Si $\Gamma \vdash M : \tau$ y $\Gamma \vdash M : \sigma$ son derivable, entonces $\tau = \sigma$.

<u>Demo:</u> supongo que $\Gamma \vdash M : \tau$ y $\Gamma \vdash M : \sigma$ pero $\tau \neq \sigma$.

Luego, $\exists T, S$ árboles de derivaciones de $\Gamma \vdash M : \tau$ y $\Gamma \vdash M : \sigma$ respectivamente.

Por inducción en cualquier árbol de derivación, veo que la elección de reglas es determinística porque hay exactamente un caso por constructor.

Luego, como en cada paso de la derivación se utilizan las mismas reglas y hay una única forma de aplicarlas veo que ambos árboles tienen la misma estructura.

Si en ambos árboles se llega a alguna aplicación de $T - VAR$ con una variable que deba tipar como $\tau$ en un árbol y como $\sigma$ en el otro, o algún tipo que los contenga, para que derive ambos casos debe pasar que para algún $x_i$, $(x_i : \tau) \in \Gamma$ y que $(x_i : \sigma) \in \Gamma$, pero como en $\Gamma$ no pueden haber variables repetidas, no puede ser el caso que se deriven ambos árboles, absurdo.

Si no, igualmente se puede ver por inducción en las reglas que los tipos que se deben satisfacer en cada paso son iguales menos el reemplazo $\tau$ por $\sigma$ y eventualmente se llega a $T - true$ ó $T - false$, en ambos casos terminé pidendo que en el caso base se satisfaga el tipo *bool*, osea el mismo tipo, con el caso base $T - var$ ya vimos que deben ser iguales, ahora mirando el árbol de arriba para abajo desde las hojas en cada paso los tipos son iguales y esta igualad de se mantiene hasta la raíz, absurdo por la supoción de que los tipos no coincidían. □

<u>Teo:</u> (Weakening + Strenthening) Si $\Gamma \vdash M : \tau$ es derivable y $\text{fv}(M) \subset \text{dom}(\Gamma \cap \Gamma')$ entonces $\Gamma' \vdash M : \tau$ es derivable.

<u>Demo:</u> construyo un árbol de derivación con la misma estructura pero reemplazando $\Gamma$ por $\Gamma'$ y por inducción en las reglas de derivación veo que todas siguen funcionando pues como pertenecen a $\text{fv}(M)$ no afectan ninguna regla.

*Semántica operacional* Indica cómo se ejecuta el programa hasta llegar a un resultado. Está la semántica small-step, ejecición paso a paso y la big-step, evaluación directa al resultado.

Hay otros tipos de dar semántica formal como la semántica denotacional que interpreta los programas como objetos matemáticos y la semántica axiomática que establece relaciones lógicas entre el estado del programa antes y después de la ejecución. Vamos a trabajar con la semántica small-step.
Def: un programa es un término $M$ tipable y cerrado ($\text{fv}(M) =$). El juecio de tipado $\vdash M : \tau$ debe ser derivable para algún $\tau$.
Def: juicios de evaluación. La semántica operacional predica sobre juicios de evaluación $M \to N$ donde $M$ y $N$ son programas.
Def: los valores son los posibles resultados de evaluar programas:

$$V ::= true|false|\lambda x : \tau.M$$

*Reglas de evaluación*

$$\frac{}{\text{if true then } M \text{ else } N \to M}\ \text{E-IFTRUE}$$

$$\frac{}{\text{if false then } M \text{ else } N \to N}\ \text{E-IFFALSE}$$

$$\frac{M \to M'}{\text{if } M \text{ then } N \text{ else } P \to \text{if } M' \text{ then } N \text{ else } P}\ \text{E-IF}$$

$$\frac{M \to M'}{M\, N \to M'\, N}\ \text{E-APP1}$$

$$\frac{N \to N'}{(\lambda x : \tau.\, M)\, N \to (\lambda x : \tau.\, M)\, N'}\ \text{E-APP2}$$

$$\frac{}{(\lambda x : \tau.\, M)\, V \to M\{x := V\}}\ \text{E-APPABS}$$

La operación de sustitución $M\{x := N\}$ denota el término que resulta de reemplazar todas las ocurrencias libres de $x$ en $M$ por $N$.

$$\begin{array}{rcl}
x\{x := N\} & \stackrel{\text{def}}{=} & N \\
a\{x := N\} & \stackrel{\text{def}}{=} & a \text{ si } a \in \{\text{true}, \text{false}\} \cup \mathcal{X} \setminus \{x\} \\
(\text{if } M \text{ then } P \text{ else } Q)\{x := N\} & \stackrel{\text{def}}{=} & \text{if } M\{x := N\} \\
 & & \ \text{then } P\{x := N\} \\
 & & \ \text{else } Q\{x := N\} \\
(M_1\, M_2)\{x := N\} & \stackrel{\text{def}}{=} & M_1\{x := N\}\, M_2\{x := N\} \\
(\lambda y : \tau.\, M)\{x := N\} & \stackrel{\text{def}}{=} &
\end{array}$$

$$\begin{cases}
\lambda y : \tau.\, M & \text{si } x = y \\
\lambda y : \tau.\, M\{x := N\} & \text{si } x \neq y,\ y \notin \text{fv}(N) \\
\lambda z : \tau.\, M\{y := z\}\{x := N\} & \text{si } x \neq y,\ y \in \text{fv}(N), \\
 & \ z \notin \{x, y\} \cup \text{fv}(M) \cup \text{fv}(N)
\end{cases}$$

Definimos un juicio de evaluacion en muchos pasos $M \to^* N$ donde $M$ y $N$ son programas.

$$\frac{}{M \twoheadrightarrow M} \qquad \frac{M \to N \quad N \twoheadrightarrow P}{M \twoheadrightarrow N}$$

### 3.4.2. Propiedades de la evaluación

<u>Teo:</u> (Determinismo) Si $M \to N_1$ y $M \to N_2$ entonces $N_1 = N_2$.

<u>Teo:</u> (Preservación te tipos) Si $\vdash M : \tau$ y $M \to N$ entonces $\vdash N : \tau$.

<u>Teo:</u> (Progreso) Si $\vdash M : \tau$ entonces o bien $M$ es un valor o bien existe $N$ tal que $M \to N$.

<u>Teo:</u> (Terminación) Si $\vdash M : \tau$, entonces no hay una cadena infinita de pasos $M \to M_1 \to M_2 \to \ldots$

<u>Coro:</u> (Canonicidad)

1. Si $\vdash M : \mathit{bool}$ es derivable, entonces la evaluación de $M$ termina y el resultado es true o false.
2. Si $\vdash M : \tau \to \sigma$ es derivable, entonces la evaluación de $M$ termina y el resultado es una abstracdción.

### 3.4.3. Cálculo-$\lambda^b n$

Es una extensión del cálcula lambda, mirar las diapos Un programa $M$ está en forma normal (f.n.) si no existe $M'$ tal que $M \to M'$.

Que una fórmula sea cerrada y tipable no implica que sea un valor a diferencia del cálculo-$\lambda^b$

Las f.n.'s que no son valores se llamas términos de error.

## 3.5. Inferencia

*Términos sin anotaciones de tipos*

$$U ::= x|\lambda x.U|UU|\text{True}|\text{False}|\text{if } U \text{ then } U \text{ else } U$$

*Términos con anotaciones de tipos*

$$M ::= x|\lambda x : \tau.U|UU|\text{True}|\text{False}|\text{if } U \text{ then } U \text{ else } U$$

<u>Def:</u> notamos erase$(M)$ al término sin anotaciones de tipos que resulta de borrar las anotaciones de tipos de $M$
<u>Def:</u> un término $U$ sin anotaciones de tipos es tipable sii existe un contexto tipado $\Gamma$, un término con anotaciones de tipos $M$ y un tipo $\tau$ tales que erase$(M) = U$ y $\Gamma \vdash M : \tau$.

*Problema de inferencia de tipos.* Consiste en dado un término $U$, determinar si es tipable. En caso de que lo sea, hallar $\Gamma, M, \tau$ tales que cumplan la definición de tipado.

*Algoritmo $I$* El algoritmo $I$ recibe un término $U$ sin anotaciones de tipos. Conta de cuatro pasos.

### 3.5.1. Rectificación

Decimos que un término está rectificado si no hay dos variables ligadas con el mismo nombre y no hay una variables ligada con el mismo nombre que una libre. Siempre se puede rectificar un término $\alpha$-renombrándolo.

### 3.5.2. Anotación

Tenemos un término $U$, que suponemos ya rectificado. Producimos un contexto $\Gamma_0$ que le da tipo a todas las variables libres de $U$ con una incógnita fresca y un término $M_0$ que está anotado de tal modo que $\text{erase}(M_0) = U$ con incógnitas frescas.

### 3.5.3. Generación de restricciones

Tenemos un contexto $\Gamma$ y un término $M$ con anotaciones de tipos. Recursivamente calculamos un tipo $\tau$ que corresponde al tipo de $M$ y un conjunto de ecuaciones $E$ que representan restricciones para que $M$ esté bien tipado.

Definimos un algoritmo recursivo $I(\Gamma|M) = (\tau|E)$ con la precondición de que $\Gamma$ le da tipo a todas las variables libre de $M$.

$I(\Gamma|\text{True}) = (\text{Bool}|\emptyset)$
$I(\Gamma|\text{False}) = (\text{Bool}|\emptyset)$
$I(\Gamma|x) = (\tau|\emptyset)$ si $(x : \tau) \in \Gamma$
$I(\Gamma|\text{if } M_1 \text{ then } M_2 \text{ else } M_3 = (\tau_2|\{\tau_1 \stackrel{?}{=} \text{Bool}, \tau_2 \stackrel{?}{=} \tau_3\} \cup E_1 \cup E_2 \cup E_3)$
donde $I(\Gamma|M_1) = (\tau_1|E_1)$, $I(\Gamma|M_2) = (\tau_2|E_2)$, $I(\Gamma|M_3) = (\tau_3|E_3)$
$I(\Gamma|M_1 M_2) = (X_k|\{\tau_1 \stackrel{?}{=} (\tau_2 \to X_k)\} \cup E_1 \cup E_2$
donde $I(\Gamma|M_1) = (\tau_1|E_1)$, $I(\Gamma|M_2) = (\tau_2|E_2)$
$I(\Gamma|\lambda x : \tau.M) = (\tau \to \sigma|E)$
donde $I(\Gamma, x : \tau|M) = (\sigma|E)$

### 3.5.4. Unificación

Una vez calculado $I(\Gamma_0|M_0) = (\tau|E)$, calculamos $S = \text{mgu}(E)$, si no existe, el término $U$ no es tipable, si existe, el término $U$ es tipable y vale $S(\Gamma_0) \vdash S(M_0) : S(\tau)$.

# 4. Prolog

Prolog opera con términos de primer orden.

```
X    X    succ(succ(zero))    bin(I, R, D), ...
```

Las fórmulas atómicas son de la forma `pred(t1,...tn)`.
Def: un programa es un conjunto de reglas. Cada regla es de la forma $\sigma : -\tau_1, ..., \tau_n$, donde $\sigma, \tau_i$ son fórmulas atómicas.
Def: las reglas en las que $n = 0$ se llaman hechos.
Las reglas tienen la siguiente interpretación lógica:

$$\forall X_1 ... \forall X_k.((\tau_1 \wedge ... \wedge \tau_n) \Rightarrow \sigma)$$

donde $X_1, ..., X_k$ son todas las variables libres de las fórmulas.
<u>Def:</u> una consulta es de la forma `?-`$\sigma_1, ..., \sigma_n$, y tiene la siguiente interpretación lógica:

$$\exists X_1 ... \exists X_k.(\sigma_1 \wedge ... \wedge \sigma_n)$$

donde $X_1, ..., X_k$ son todas las variables libres de las fórmulas.

El entornno Prolog busca demostrar la fórmula $\tau$ de la consulta. En realidad busca refutar $\neg\tau$, o sea, demostrar $\neg\tau \Rightarrow \perp$. La búsqueda de la refutación se basa en el método de resolución.