

Resumen - Paradigmas de la Programación

⚠ Disclaimer

Hice este resumen basándome principalmente en las diapositivas/apuntes teóricos de **Pablo Barenbaum y Alejandro Díaz-Caro**, sumado a conclusiones que fui sacando durante la cursada. Este apunte está hecho por mí, y para mí; pensado para ser leido por mí, entendido por mí.

No es un lugar fiable de estudio ni mucho menos. Puede contener *orrorez*, pues tiene mi toque de conclusiones (y humor) que pueden estar erradas, aún así decidí ponerlo público, pues capaz a alguien le sirve.

Puede ser que no profundice en temas que considere medio elementales, y en otros profundice de más pues los consideré más relevante al momento de redactar el documento.

Con esto dicho, invito la lectura a la persona aventurera que lo deseé.

≈ @valnrm

Programación funcional, Expresiones, Tipos y polimorfismo, Modelo de cómputo (Capítulo 0)

Programación funcional 101 (Capítulo 0: parte I)

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

Solemos decir que "**las funciones son funciones** (parciales), **posta**." (como en matemáticas!!1):

- **Aplicar una función no tiene efectos secundarios.**
- **Transparencia referencial:** el **valor del output** corresponde **únicamente del input**.
- **Las estructuras de datos son inmutables.**

Las funciones son ***first-class citizen***, es decir, **son como cualquier otro tipo de dato**:

- Se pueden pasar como parámetro.
- Se pueden devolver como resultado.
- Pueden formar partes de estructuras de datos.

Un programa funcional está dado por un conjunto de ecuaciones. (*más información sobre esto en la parte IV de modelo de cómputo*)

Funciones de orden superior

Son funciones que cumplen **alguna** de estas dos condiciones:

1. **Toman** una o más **funciones como argumento**.
2. **Devuelven** una **función** como resultado.

Un ejemplo clave se relaciona con la notación lambda (λ) en Haskell.

Supongamos la siguiente función $add\ x = (\lambda\ y \rightarrow x + y)$

add es una función de orden superior. Fíjate que devuelve una función como resultado.

¡Devuelve una Lambda (λ)!

Nótese que al devolver funciones, permitimos algo muy fuerte de la programación funcional llamado **curryficación**. La curryficación nos permite realizar aplicaciones parciales a funciones. Por ejemplo,

```
1  add :: Int -> Int -> Int
2  add n m = n + m
3
4  -- Puedo hacer
5  next = add 1
6
7  -- Ahora hacer next {Int} me da el número siguiente a {Int}.
8  next 0 -- Devuelve 1
9  next 1 -- Devuelve 2
10   -- ...
```

Más ejemplos míticos: *curry*, *uncurry*, *map*, *(.)*, etc...

Por ejemplo, *map* toma una función.

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

Expresiones (Capítulo 0: parte II)

Tenemos **expresiones**. Son secuencias de **símbolos**, por ejemplo, *carlos*.

carlos, es una expresión. Así como *carlos* es una expresión, tenemos más expresiones.

Por ejemplo: *True*, *False*, *argentinaCampeónDelMundo*, *[]*, *(:)*, *plp*, *miau*, *0*, *1337*, *curry*, *x*, *y*, *funciónMágicaQueResuelveTodosLosProblemas*...

Las **expresiones** se pueden caracterizar de **diferentes maneras**:

- **Constructor:** *True*, *False*, *[]*, *(:)*, *0*, *1337*
- **Variable:** *dobleHuevo*, *plp*, *miau*, *curry*, *x*, *y*
funciónMágicaQueResuelveTodosLosProblemas
- **Aplicación:**
 - *dobleHuevo siemprePá*
 - *curry x y*
 - *funciónMágicaQueResuelveTodosLosProblemas problemaMuyMuyFeo*
 - *miau meow*

Hay más caracterizaciones pero estas son **fundamentales**.

Tipos y polimorfismo (Capítulo 0: parte III)

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

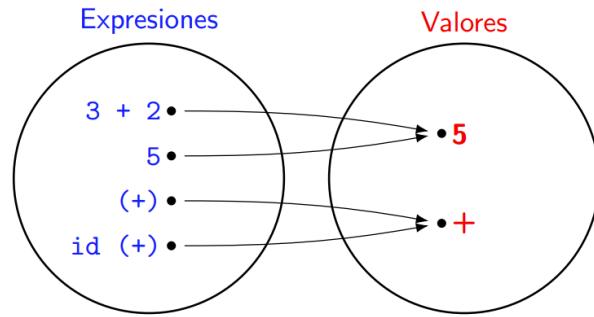
El tipo de una función expresa un **contrato**.

Si la expresión tiene más de un tipo, decimos que hay **polimorfismo**.

Ejemplitos: $id :: a \rightarrow a$, $[] :: [a]$, $(:) :: a \rightarrow [a] \rightarrow [a]$, $fst :: (a, b) \rightarrow a$, $snd :: (a, b) \rightarrow b$

Modelo de cómputo (prog. func.) (Capítulo 0: parte IV)

Dada una expresión, se computa su *valor*, usando las ecuaciones.



No toda expresión bien tipada tiene valor. Por ejemplo: $1/0$.

Decimos que este tipo de expresiones se **indefinen**, o tienden a \perp "bottom".

Un programa funcional está dado por un conjunto de ecuaciones. Más precisamente, por un conjunto de **ecuaciones orientadas**.

Se le llama *ecuaciones orientadas* pues las ecuaciones responden a una cierta orientación: del lado izquierdo está lo que se define, y del lado derecho la definición (o expresión que produce el valor).

Una ecuación $e1 = e2$ se interpreta desde dos puntos de vista:

1. **Denotacional**: Declara que $e1$ y $e2$ tienen el mismo significado.
 - "*Denotan lo mismo*".
2. **Operacional**: Computar el valor de $e1$ se reduce a computar el valor de $e2$.
 - "*Operan de la misma forma*".

El lado izquierdo de una ecuación no puede ser cualquier cosa. *ladoIzquierdo* = ...

¿Me comprás si te digo $1 + n = sumaleUno n$? , ¿no verdad?

¿Cómo que decir $sumaleUno n = n + 1$ es más natural? , ¿no?

El lado izquierdo de una ecuación no es una expresión arbitraria. Debe ser una función aplicada a patrones.

Un patrón puede ser:

1. Variable: n , x , xs
2. Comodín: $_$.
3. Constructor aplicado a patrones: $x : xs$, 0 , Nil

En nuestro ejemplo, n es un patrón pues es una variable.

Y no debe contener variables repetidas. (*Escribía divisiónEntera n n = div n n, ¿te imaginás?, ¿cuál n es cuál?, ni idea, incomprendible*)

Modelo de cómputo Evaluar una expresión consiste en: <ol style="list-style-type: none">1. Buscar la subexpresión más externa que coincide con el lado izquierdo 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 evaluando la expresión resultante.	La evaluación se detiene cuando se da uno de los siguientes casos: <ol style="list-style-type: none">1. La expresión es un constructor o un constructor aplicado. $True$ $(:) 1$ $[1, 2, 3]$2. La expresión es una función <i>parcialmente aplicada</i>. $(+)$ $(+) 5$3. Se alcanza un <i>estado de error</i>. Un estado de error es una expresión que no coincide con las ecuaciones que definen a la función aplicada.
--	--

Esquemas de recursión y Tipos de datos inductivos (Capítulo 1)

Esquemas de recursión (Capítulo 1: parte I)

Listas

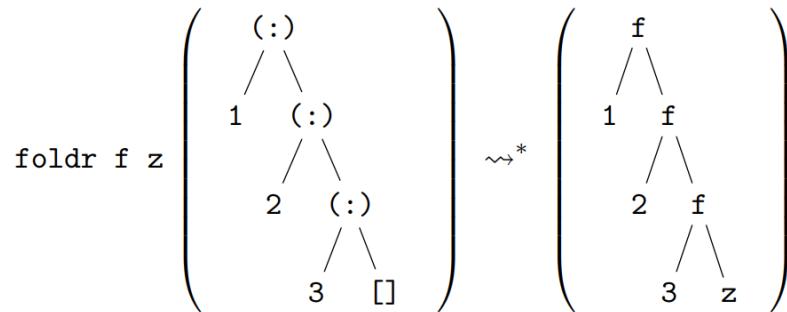
Cuando hacemos recursión, hay ciertos patrones que suelen repetirse. Es por ello que contamos con ciertas funciones que abstraen ciertos esquemas de recursión.

- **Recursión estructural (*foldr*):**

- El caso base devuelve un valor fijo *z*.
- El caso recursivo se escribe usando (cero, una, o muchas veces) *x*, y la llamada recursiva (en *foldr* es `(foldr f z xs)`, en *map* es `map f xs`). No se utiliza ni *xs*, ni otros llamados recursivos.

```
1  foldr :: (a -> b -> b) -> b -> [a] -> b
2  foldr f z [] = z
3  foldr f z (x : xs) = f x (foldr f z xs)
4
5  -- Ejemplito de map usando foldr:
6  map :: (a -> b) -> [a] -> [b]
7  map f = foldr (\x rec -> f x : rec) []
```

Ilustración gráfica del plegado a derecha



Existe también *foldr1*. Es igual a *foldr*, solo que no toma un caso base *z*. El caso base es la instancia con un solo elemento. Por ejemplo en listas sería *x* sería el caso base cuando la lista es *[x]*.

```
1  foldr1 :: (a -> a -> a) -> [a] -> a
2  foldr1 _ [x] = x
3  foldr1 f (x : xs) = f x (foldr1 f xs)
```

- **Recursión primitiva (*recr*):**

- Casi igual a la estructural; tiene de diferencia que podés usar el *xs*.
- Recursión primitiva \implies Recursión estructural (*la 2da es un caso particular de la 1ra*)

```
1  recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
2  recr f z [] = z
3  recr f z (x : xs) = f x xs (recr f z xs)
4
5  -- Ejemplito de trim usando recr:
6  trim :: String -> String
7  trim = recr (\x xs rec -> if x == ' ' then rec else x : xs) []
```

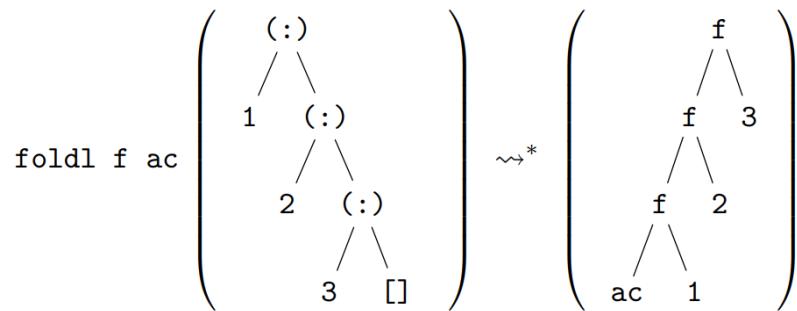
- **Recursión iterativa (foldl):**
 - El caso base devuelve el acumulador *acc*.
 - El caso recursivo invoca inmediatamente a (*f acc x*). Onda, vas trabajando con el acumulador.

```

1  foldl :: (b -> a -> b) -> b -> [a] -> b
2  foldl f acc [] = acc
3  foldl f acc (x : xs) = foldl f (f acc x) xs
4
5  -- Ejemplito de reverse usando foldl
6  reverse :: [a] -> [a]
7  reverse = foldl (\acc x -> x : acc) []

```

Ilustración gráfica del plegado a izquierda



Al igual que en *foldr*, existe *foldl1*. Sigue la misma idea refente al caso base.

Relación entre *foldr* y *foldl*:

No es casualidad los nombres. Estas funciones están estrechamente relacionadas.

En general *foldr* y *foldl* tienen comportamientos diferentes:

$$\begin{aligned} \text{foldr} (\star) z [a, b, c] &= a \star (b \star (c \star z)) \\ \text{foldl} (\star) z [a, b, c] &= ((z \star a) \star b) \star c \end{aligned}$$

Si (\star) es un operador asociativo y conmutativo, *foldr* y *foldl* definen la misma función. Por ejemplo:

$$\begin{array}{lll} \text{suma} & = \text{foldr } (+) 0 & = \text{foldl } (+) 0 \\ \text{producto} & = \text{foldr } (*) 1 & = \text{foldl } (*) 1 \\ \text{and} & = \text{foldr } (\&&) \text{ True} & = \text{foldl } (\&&) \text{ True} \\ \text{or} & = \text{foldr } (||) \text{ False} & = \text{foldl } (||) \text{ False} \end{array}$$

```

1  f :: a -> b -> b
2  z :: b
3  xs :: [a]
4
5  foldr f z xs = foldl (flip f) z (reverse xs)

```

Otras estructuras

Puede ser interesante leer la parte II antes de entrar acá.

Cuando queremos hacer recursión estructural sobre estructuras de datos con más de un constructor (*sean/n recursivo/s, o no*), es importante tener en cuenta que vamos a tener tanta cantidad de casos, como constructores tenga nuestro tipo de datos.

La recursión estructural se generaliza a tipos algebraicos en general.

Supongamos que T es un tipo algebraico.

Dada una función $g : T \rightarrow Y$ definida por ecuaciones:

$$\begin{aligned} g(\text{CBase}_1 \langle \text{parámetros} \rangle) &= \langle \text{caso base}_1 \rangle \\ \dots \\ g(\text{CBase}_n \langle \text{parámetros} \rangle) &= \langle \text{caso base}_n \rangle \\ g(\text{CRecursivo}_1 \langle \text{parámetros} \rangle) &= \langle \text{caso recursivo}_1 \rangle \\ \dots \\ g(\text{CRecursivo}_m \langle \text{parámetros} \rangle) &= \langle \text{caso recursivo}_m \rangle \end{aligned}$$

```
1 -- Ejemplo típico:
2 data AB a = Nil | Bin (AB a) a (AB a) -- Árbol binario
3
4 -- Definamos foldAB para abstraer la recursión estructural en el tipo de
5 dato de árbol binario
6 foldAB :: (b -> a -> b -> b) -> b -> AB a -> b
7 foldAB cBin cNil Nil = cNil
8 foldAB cBin cNil (Bin i r d) = cBin (foldAB cBin cNil i) r (foldAB cBin cNil
d)
9
10 -- Y ya que estamos haciendo map para el árbol binario
11 mapAB :: (a -> b) -> AB a -> AB b
12 mapAB f = foldAB (\recIzq raíz recDer -> Bin recIzq (f raíz) recDer) Nil
```

Tipos de datos inductivos (Capítulo 1: parte II)

Tenemos algunos tipos de datos primitivos: *Char*, *Int*, *Float*, $a \rightarrow b$, (a, b) , $[a]$, *String*.

Y podemos definir los nuestros propios.

```
1 -- Cómo se construye: data Type = <declaración de constructores>
2
3 -- Ejemplo
4 data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
```

Dom, *Lun*, ..., *Sab*, son **constructores (simples)**. Todos ellos son (*los únicos constructores*) del tipo *Dia*.

Ahora, un constructor también puede tener parámetros.

```
1 data Vehiculo = Auto String Int | Moto String String Float
2
3 Auto :: String -> Int -> Vehiculo
4 Moto :: String -> String -> Float -> Vehiculo
```

En este caso *Auto* y *Moto* son los únicos dos constructores del tipo de dato *Vehiculo*.

Auto toma dos parámetros, un *String* y un *Int*; mientras que *Moto* toma un *String*, un *String*, y un *Float*.

Y también tenemos **tipos de datos recursivos**.

```
1 data Nat = Zero | Succ Nat
2
3 Zero :: Nat
4 Succ :: Nat -> Nat
```

Zero es un constructor simple, mientras que *Succ* es un constructor que "toma" algo de tipo *Nat*, esto lo hace recursivo.

Las funciones sobre tipos de datos con constructores recursivos normalmente se definen por recursión.

```
1 doble :: Nat -> Nat
2 doble Zero = Zero
3 doble (Succ n) = Succ (Succ (doble n))
```

Podemos definir estructuras de datos infinitas:

```
1  infinito :: Nat
2  infinito = Succ infinito
```

Tipos de datos algebraico — caso general

En general un tipo de datos algebraico tiene la siguiente forma:

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

- ▶ Los constructores **base** no reciben parámetros de tipo T.
 - ▶ Los constructores **recursivos** reciben al menos un parámetro de tipo T.
 - ▶ Los valores de tipo T son los que se pueden construir aplicando constructores base y recursivos un número **finito** de veces y **sólo** esos.
*(Entendemos la definición de T de forma **inductiva**).*

(Entendemos la definición de T de forma **inductiva**).

```

1 -- Ya que estoy defino Polinomio también
2 data Polinomio a = X
3     | Cte a
4     | Suma (Polinomio a) (Polinomio a)
5     | Prod (Polinomio a) (Polinomio a)
6
7 foldPoli :: b -> (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -
8 > b
9 foldPoli z _ _ _ X = z
10 foldPoli z fCte _ _ (Cte m) = fCte m
11 foldPoli z fCte fSuma fProd (Suma p q) = fSuma (foldPoli z fCte fSuma fProd
12 p) (foldPoli z fCte fSuma fProd q)
13 foldPoli z fCte fSuma fProd (Prod p q) = fProd (foldPoli z fCte fSuma fProd
14 p) (foldPoli z fCte fSuma fProd q)

evaluar :: Num a => a -> Polinomio a -> a
evaluar n = foldPoli n id (+) (*)

```

Razonamiento ecuacional, Inducción estructural, Extensionalidad, Isomorfismo de tipos (Capítulo 2)

Dijimos en un primer momento que los programas en el paradigma funcional son un conjunto de ecuaciones orientadas.

Al ser ecuaciones es razonable plantearse, ¿es posible demostrar que ciertas expresiones son equivalentes?, y si se pudiese, ¿para qué nos interesaría?

Podríamos, por ejemplo, demostrar que un algoritmo es correcto, o realizar optimizaciones a un programa.

Razonamiento ecuacional (Capítulo 2: parte I)

Para razonar sobre equivalencia de expresiones vamos a asumir:

1. Que trabajamos con **estructuras de datos finitas** (*técnicamente: con tipos de datos inductivos*)
2. Que trabajamos con **funciones totales**.
 - Las ecuaciones deben cubrir todos los casos.
 - La recursión siempre debe terminar.
3. Que el programa **no depende del orden** de las ecuaciones.

Inducción estructural (Capítulo 2: parte II)

Principio de inducción estructural:

Sea P una propiedad acerca de las expresiones tipo T tal que:

- P vale sobre todos los constructores base de T ,
- P vale sobre todos los constructores recursivos de T , asumiendo como hipótesis inductiva que vale para los parámetros de tipo T , entonces $\forall x :: T. P(x)$.

En criollo: Para probar P sobre todas las instancias de un tipo T , basta probar P para cada uno de los constructores (asumiendo la H.I. para los constructores recursivos).

- Principio de **inducción sobre booleanos**:
 - Si $P(\text{True})$ y $P(\text{False})$ entonces $\forall x :: \text{Bool}. P(x)$
 - En criollo: si es un bool podés separar en casos *True* y *False*.
 - Ejemplito:
 - Para probar $\forall x :: \text{Bool}. \text{not}(\text{not } x) = x$, basta probar $\text{not}(\text{not True}) = \text{True} \wedge \text{not}(\text{not False}) = \text{False}$.
- Principio de **inducción sobre pares**:
 - Si $\forall x :: a. \forall y :: b. P(x, y)$ entonces $\forall p :: (a, b). P(p)$.
 - En criollo: si probás que vale para una tupla cualquiera (x, y) tal que $x :: a, y :: b$, entonces vale para todo par/tupla $p :: (a, b)$.
- Principio de **inducción sobre naturales**:
 - No requiere mucha presentación. Es el mismo que el de Álgebra I.
 - Si $P(\text{Zero})$ y $\forall n :: \text{Nat}. P(n) \implies P(\text{Suc } n)$, entonces $\forall n :: \text{Nat}. P(n)$.
 - A $P(\text{Zero})$ se lo conoce como **caso base**.

- Al antecedente del $\forall n$ (*es decir*, $P(n)$), se lo conoce como **hipótesis inductiva**. Al consecuente como tesis inductiva.
- Tenés que probar el caso base, y que, asumiendo la **H.I.**, podés concluir la **T.I.**.

Extensionalidad (Capítulo 2: parte III)

Se deriva a partir del principio de inducción estructural.

- Principio de **extensionalidad para pares**:
 - Si $p :: (a, b)$, entonces $\exists x :: a. \exists y :: b. p = (x, y)$.
 - En criollo: si tenés un par/tupla $p :: (a, b)$, entonces existen $x :: a, y :: b$ que lo forman (al par/tupla).
- Principio de **extensionalidad para sumas**:
 - Parecido al de booleanos.
 - Si $e :: Either a b$, entonces:
 - o bien $\exists x :: a. e = Left x$
 - o bien $\exists y :: b. e = Right y$

Al preguntarnos sobre la igualdad de expresiones tenemos dos posturas.

- Decimos que dos valores son iguales:
 - **intensionalmente** si están definidos de la misma manera.
 - **extensionalmente** si son indistinguibles al observarlos.
- Principio de **extensionalidad funcional**:
 - Si $(\forall x :: a. f x = g x)$ entonces $f = g$.
 - En criollo: para probar que dos funciones son iguales, basta probar que son iguales punto a punto.

Algunas propiedades sacadas de la práctica:

1	$F :: a \rightarrow b$
2	$G :: a \rightarrow b$
3	$Y :: b$
4	$Z :: a$

Estas dos reglas son casos particulares del principio de extensionalidad de funciones (más o menos):

$$F = G \iff \forall x :: a. F x = G x$$

$$F = \lambda x \rightarrow Y \iff \forall x :: a. F x = Y$$

Regla β : $(\lambda x \rightarrow Y) Z = Y$ (reemplazando x por Z)

Regla η : $\lambda x \rightarrow F x = F$

Isomorfismo de tipos (Capítulo 2: parte IV)

Decimos que dos tipos de datos A y B son **isomorfos** si:

1. Hay una función $f :: A \rightarrow B$ total.
2. Hay una función $g :: B \rightarrow A$ total.
3. Se puede demostrar que $g . f = id :: A \rightarrow A$.
4. Se puede demostrar que $f . g = id :: B \rightarrow B$.

Escribimos $A \simeq B$ para indicar que A y B son isomorfos.

Sistemas deductivos (Capítulo 3)

Antes de empezar a hablar de sistemas deductivos es importante entender las **fórmulas** son un **conjunto inductivo**.

Fórmulas (gramática de las fórmulas en lógica proposicional)

1. cualquier variable proposicional es una fórmula.
2. si τ es una fórmula, $(\neg\tau)$ es una fórmula.
3. si τ y σ son fórmulas, $(\tau \wedge \sigma)$ es una fórmula.
4. si τ y σ son fórmulas, $(\tau \vee \sigma)$ es una fórmula.
5. si τ y σ son fórmulas, $(\tau \Rightarrow \sigma)$ es una fórmula.
6. si τ y σ son fórmulas, $(\tau \Leftrightarrow \sigma)$ es una fórmula.

Al ser un conjunto inductivo, viene provisto de:

- Esquema de prueba para probar propiedades sobre ellos (**inducción estructural**).
- Esquema de recursión para definir funciones sobre el conjunto (**recursión estructural**).

Valuación

- Una valuación es una función $v : \mathcal{V} \Rightarrow \{V, F\}$ que asigna valores de verdad a las variables proposicionales.
- Una valuación satisface una proposición τ si $v \models \tau$ donde:
 - $v \models P$ si $v(P) = V$
 - $v \models \neg\tau$ si $v \not\models \tau$ (i.e. no $v \models \tau$)
 - $v \models \tau \vee \sigma$ si $v \models \tau$ o $v \models \sigma$
 - $v \models \tau \wedge \sigma$ si $v \models \tau$ y $v \models \sigma$
 - $v \models \tau \Rightarrow \sigma$ si $v \not\models \tau$ o $v \models \sigma$
 - $v \models \tau \Leftrightarrow \sigma$ si $(v \models \tau \text{ si } v \models \sigma)$

Equivalencia lógica y tipos de fórmulas

Dadas fórmulas τ y σ :

- τ es lógicamente equivalente a σ cuando $v \models \tau$ si $v \models \sigma$ para toda valuación v .

Una fórmula τ es:

- una tautología si $v \models \tau$ para toda valuación v .
- satisfactible si existe una valuación v tal que $v \models \tau$.
- insatisfactible si no es satisfactible.

Un conjunto de fórmulas Γ es:

- satisfactible si existe una valuación v tal que para todo $\tau \in \Gamma$, se tiene $v \models \tau$.
- insatisfactible si no es satisfactible.

Teorema: Una fórmula τ es una tautología si $\neg\tau$ es insatisfacible.

Sistema deductivo basado en reglas de prueba

Secuente

$$\tau_1, \tau_2, \dots, \tau_n \vdash \sigma$$

Denota que a partir de asumir que el conjunto de fórmulas $\{\tau_1, \tau_2, \dots, \tau_n\}$ son tautologías, podemos obtener una prueba de la validez de σ .

Reglas de prueba

$$\frac{\Gamma_1 \vdash \tau_1 \quad \dots \quad \Gamma_n \vdash \tau_n}{\Gamma \vdash \sigma} \text{ nombre_de_la_regla}$$

Permiten deducir un secuente (conclusión) a partir de ciertos otros (premisas).

$$\begin{aligned} & \Gamma_1 \vdash \tau_1 \dots \Gamma_n \vdash \tau_n : \text{premisas} \\ & \Gamma \vdash \sigma : \text{conclusión} \end{aligned}$$

Pruebas

- Aplicando reglas de prueba a premisas y conclusiones obtenidas previamente.
- Un secuente es **válido** si podemos construir una prueba.

Importancia de la elección de las reglas

- Deben permitir construir solo pruebas que constituyan una argumentación válida.
 - Deberían impedir probar secuentes tales como $P, Q \vdash P \wedge \neg Q$.
- Deberían permitir inferir todas las fórmulas que se desprenden de las premisas.

Llamamos **teorema** a toda fórmula lógica τ tal que el secuente $\vdash \tau$ es válido.

Lógica clásica vs constructiva / intuicionista

Existe la lógica clásica y la intuicionista.

Las reglas *vistas en clase* de la **lógica clásica** son: *PBC* (proof by contradiction), *LEM* (law of excluded middle), y $\neg\neg_e$ (double negation elimination).

El resto que vimos son de lógica intuicionista.

Las tres reglas *vistas en clase* de lógica clásica son equivalentes. Se puede probar que

$$PBC \implies LEM \implies \neg\neg_e \implies PBC$$

Aclaración: cuando digo "reglas de lógica clásica", estoy pensando en $\{\text{reglas de lógica clásica}\} \setminus \{\text{reglas de lógica intuicionista}\}$, es sabido que las reglas de la lógica intuicionista están contenidas en las reglas de la lógica clásica.

Corrección y completitud

Responde a la pregunta: ¿cuál es la relación entre la sintaxis y la semántica?

Sintaxis: Conjunto de fórmulas τ tal que $\neg\tau$ es un secuente válido.

Semántica: Conjunto de fórmulas τ tal que $v \models \tau$, para toda valuación v (i.e. tautologías).

Corrección

$\vdash \tau$ secuente válido implica que τ es tautología.

- En criollo: τ tiene una prueba implica que τ es tautología
- Generalizada: $\sigma_1, \sigma_2, \dots, \sigma_n \vdash \tau$ secuente válido implica que $\sigma_1, \sigma_2, \dots, \sigma_n \models \tau$

Una idea para la demo de **corrección** es hacer inducción en la estructura de la prueba, procedemos analizando por casos la última regla aplicada en la prueba.

Caso base, la última regla fue un axioma.

Casos recursivos, el resto de reglas $\wedge_i, \wedge_{e_1}, \text{etc} \dots$

Completitud

τ tautología implica que $\vdash \tau$ es secuente válido.

- En criollo: τ tautología implica que τ tiene una prueba.
- Generalizada: $\sigma_1, \sigma_2, \dots, \sigma_n \models \tau$ implica que $\sigma_1, \sigma_2, \dots, \sigma_n \vdash \tau$ es secuente válido

Una idea para la demo de **completitud** es usar por el contrarrecíproco: si $P \Rightarrow Q$ vale que $\neg Q \Rightarrow \neg P$.

Luego, probar estos dos lemas:

1. Si $\Gamma \not\vdash \tau$, entonces $\Gamma \cup \{\neg\tau\}$ es consistente. (*Sale por el contrarrecíproco también*)
2. Si Γ es consistente, entonces tiene modelo (i.e. Γ es satisfacible). (*Más jodido*)

Con esos dos lemas sale, pues supongamos $\Gamma \not\vdash \tau$

$$\begin{aligned} &\Rightarrow \Gamma \cup \{\neg\tau\} \text{ es consistente (por Lema 1)} \\ &\Rightarrow \Gamma \cup \{\neg\tau\} \text{ tiene modelo (por Lema 2)} \\ &\Rightarrow \exists v \text{ tal que } \forall \sigma \in \Gamma \cup \{\neg\tau\}, v \models \sigma \text{ (def. de tener modelo)} \\ &\Rightarrow \exists v \text{ tal que } v \not\models \tau \text{ y } \forall \sigma \in \Gamma, v \models \sigma \text{ (def. de } \models) \\ &\Rightarrow \Gamma \not\models \tau \text{ (def. de consecuencia semántica)} \end{aligned}$$

Consecuencia semántica

Sean $\tau_1, \tau_2, \dots, \tau_n, \sigma$ fórmulas de la lógica proposicional.

$$\tau_1, \tau_2, \dots, \tau_n \models \sigma \iff \forall v \text{ valuación } ((v \models \tau_1 \wedge v \models \tau_2 \wedge \dots \wedge v \models \tau_n) \Rightarrow (v \models \sigma)).$$

En criollo: Vale que $\tau_1, \tau_2, \dots, \tau_n \models \sigma \iff$ (para toda valuación v tal que v satisface a todas las hipótesis ($v \models \tau_i$ para toda $i \in 1, \dots, n$) $\Rightarrow v \models \sigma$).

Definiciones extra

Sea $\Gamma = \tau_1, \tau_2, \dots, \tau_n$ un conjunto de hipótesis.

Se dice que Γ es **consistente** si $\Gamma \not\vdash \perp$

- En criollo: Γ es consistente si no se puede derivar una contradicción a partir de él.

Decimos que Γ tiene un **modelo** (o es *satisfacible*) si existe una valuación v tal que $v \models \tau$, para toda $\tau \in \Gamma$.

Decimos que Γ es **consistente maximal** si

- Γ es consistente.

- Si $\Gamma \subseteq \Gamma'$ y Γ' es consistente, entonces $\Gamma' = \Gamma$.
 $\equiv \Gamma \subset \Gamma'$, entonces Γ' es inconsistente.

Comentarios

Afirmación

" $\sigma_1, \sigma_2, \dots, \sigma_n \vdash \tau$ es secuente válido" significa que hay una prueba de τ bajo las hipótesis $\sigma_1, \sigma_2, \dots, \sigma_n$.

$\sigma_1, \sigma_2, \dots, \sigma_n \models \tau$ significa que para toda valuación v tal que $v \models \sigma_i$, para toda $i \in 1 \dots n$, vale que $v \models \tau$.

Negación

" $\sigma_1, \sigma_2, \dots, \sigma_n \vdash \tau$ no es secuente válido" significa que no hay una prueba de τ bajo las hipótesis $\sigma_1, \sigma_2, \dots, \sigma_n$.

$\sigma_1, \sigma_2, \dots, \sigma_n \not\models \tau$ significa que existe valuación v tal que $v \models \sigma_i$, para toda $i \in 1 \dots n$, pero $v \not\models \tau$.

Cálculo- λ (Capítulos 4, 5, 6)

- Para esta sección recomiendo ir directamente [al apunte de cálculo- \$\lambda\$ de Jano](#).

Voy a resolver algunos ejercicios del apunte de cálculo- λ de Jano.

Ejercicio 1.3

Faltarán dos reglas de congruencia.

$$\begin{array}{lcl} \text{if } M \text{ then } O \text{ else } P & \rightarrow & \text{if } M \text{ then } O' \text{ else } P & (\text{if}_{c_t}) \\ \text{if } M \text{ then } O \text{ else } P & \rightarrow & \text{if } M \text{ then } O \text{ else } P' & (\text{if}_{c_f}) \end{array}$$

Servirían capaz para optimizar programas, pero es un tema, suponete que la rama *True* reduce a una función λ y la rama *False* a un \mathbb{Bool} , te podés meter en quilombo al pedo.

A su vez, un *if* jamás va a ser un valor, por lo que si tenés que ejecutarlo (*estoy pensando en evaluación lazy*), sí o sí vas a tener que hacer (if_c).

Conclusión, no te aportan nada.

Ejercicio 1.4

Aclaración: la regla α equivalencia la usamos siempre en clase, pero no la llamamos como tal nunca (salvo una vez que la mencionó Gabi).

$$\begin{array}{ccc} (\lambda x. \lambda x. x) \mathbf{ttff} & \xrightarrow{\alpha \text{ equivalencia}} & (\lambda x. \lambda y. y) \mathbf{ttff} \\ (\lambda x. \lambda y. y) \mathbf{ttff} & \xrightarrow{\beta \text{ reducción}} & (\lambda y. y) \{x := \mathbf{tt}\} \mathbf{ff} \equiv (\lambda y. y) \mathbf{ff} & \xrightarrow{\beta \text{ reducción}} & y \{y := \mathbf{ff}\} \equiv \mathbf{ff} \end{array}$$

Dejo el resto a ustedes, este ejercicio no es muy interesante.

Ejercicio 1.5

Recordemos la gramática de M .

$$M ::= x \mid \lambda x. M \mid MM \mid tt \mid ff \mid \text{if } M \text{ then } M \text{ else } M$$

Casos base:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(tt) &= FV(ff) = \emptyset \end{aligned}$$

Casos recursivos:

$$\begin{aligned} FV(\lambda x. M) &= FV(M) \setminus \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\text{if } M_1 \text{ then } M_2 \text{ else } M_3) &= FV(M_1) \cup FV(M_2) \cup FV(M_3) \end{aligned}$$

Ejercicio 1.11

1. Idea de la demo para el Lema 1:

Sé que \rightarrow_R satisface la *propiedad del diamante*. Quiero ver que \rightarrow_R^* lo satisface.

Pensemos un poco. Sé que si $M \rightarrow_R N_1$ y $M \rightarrow_R N_2$, entonces existe O tal que $N_1 \rightarrow_R O$ y $N_2 \rightarrow_R O$ (definición de satisfacer la *propiedad del diamante*).

Ahora, esta relación \rightarrow_R es en un paso. Mientras que \rightarrow_R^* puede ser en $0, 1, \dots, n$ pasos (*finitos*).

Nada, sabés que uno lo hace en un paso, querés ver que el otro lo haga en una cierta cantidad finita de pasos entre 0 y n . Asumo que ya ves por donde voy, tomá cantidad de pasos = 1 y listo.

Luego como \rightarrow_R^* cumple la *propiedad del diamante*, entonces \rightarrow_R es *Church-Rosser*.

2. Idea de la demo para el Lema 2:

Sé que \rightarrow_R es *Church-Rosser*. Es decir, que \rightarrow_R^* satisface la *propiedad del diamante*.

Quiero ver que \rightarrow_R tiene *formas normales únicas*, es decir que si $M \rightarrow_R^* N_1$ y $M \rightarrow_R^* N_2$, entonces $N_1 = N_2$. Bueno, sé que si $M \rightarrow_R^* N_1$ y $M \rightarrow_R^* N_2$, entonces $N_1 \rightarrow_R^* O$ y $N_2 \rightarrow_R^* O$ (pues \rightarrow_R^* cumple la propiedad del diamante), luego $O = O$.

Ejercicio 1.20

Reglas de reducción (**CBN**):

$$\begin{array}{lll} (\lambda x. M)N & \rightarrow & M\{x := N\} & (\beta \text{ reducción}) \\ \text{if tt then } M \text{ else } N & \rightarrow & M & (\text{if}_t) \\ \text{if ff then } M \text{ else } M & \rightarrow & N & (\text{if}_f) \end{array}$$

Reglas de congruencia (**CBN**):

- Si $M \rightarrow M'$, entonces

$$\begin{array}{lll} MN & \rightarrow & M'N & (\text{app}) \\ \text{if } M \text{ then } O \text{ else } P & \rightarrow & \text{if } M' \text{ then } O \text{ else } P & (\text{if}_c) \end{array}$$

Ejercicio 1.23

Sea V un valor.

Reglas de reducción (**CBV**):

$$\begin{array}{lll} (\lambda x. M)V & \rightarrow & M\{x := V\} & (\beta \text{ reducción}) \\ \text{if tt then } M \text{ else } N & \rightarrow & M & (\text{if}_t) \\ \text{if ff then } M \text{ else } M & \rightarrow & N & (\text{if}_f) \end{array}$$

Reglas de congruencia (**CBV**):

- Si $M \rightarrow M'$, entonces

$$\begin{array}{llll}
 MN & \rightarrow & M'N & (\text{app}_l) \\
 VM & \rightarrow & VM' & (\text{app}_r) \\
 \text{if } M \text{ then } O \text{ else } P & \rightarrow & \text{if } M' \text{ then } O \text{ else } P & (\text{if}_c)
 \end{array}$$

Ejercicio 1.27

No es muy interesante, me lo salteo.

Ejercicio 1.28

El término $\lambda x : \tau. xx$ es tipable únicamente en **Sistema F** (*al menos dentro del foco de la materia*).

Ejercicio 1.30

Quiero probar que, si $\Gamma \vdash M : \tau$ y $M \rightarrow N$, entonces $\Gamma \vdash N : \tau$.

Jano propone demostrar el **lema de sustitución**, y luego usarlo para demostrar el teorema por inducción en la definición de \rightarrow .

Probemos primero el lema de sustitución.

Lema: Si $\Gamma, x : \tau \vdash M : \sigma$ y $\Gamma \vdash N : \tau$, entonces $\Gamma \vdash M\{x := N\} : \sigma$.

Jano propone demostrar este lema por inducción en M . Hagámoslo por inducción en M .

Recordemos nuevamente la gramática de M .

$$M ::= x \mid \lambda x. M \mid MM \mid tt \mid ff \mid \text{if } M \text{ then } M \text{ else } M$$

Casos base:

Caso $M = y$ con $y \neq x$:

Tenemos $\Gamma, x : \tau \vdash y : \sigma$ y $\Gamma \vdash N : \tau$

Luego, por la propiedad de *fortalecimiento* o *strengthening* (Guía 4: Ej. 11, item 2), vale que $\Gamma \vdash y\{x := N\} : \sigma$ (o lo que es lo mismo, $\Gamma \vdash y : \sigma$).

Caso $M = x$ con $\sigma = \tau$:

Tenemos $\Gamma, x : \tau \vdash x : \tau$ y $\Gamma \vdash N : \tau$

Luego, vale que $\Gamma \vdash x\{x := N\} : \tau$ (o lo que es lo mismo, $\Gamma \vdash N : \tau$).

También los casos tt y ff son casos base, pero son triviales.

Pasos inductivos:

Ahora nos toca probarlo para todos los otros constructores de M -términos. Siendo nuestra hipótesis inductiva, que vale para cualquier subtérmino.

Caso $M = \lambda x. M'$:

Sé que $\Gamma \vdash N : \tau$, y que $\Gamma, x : \tau \vdash \lambda y. M' : \rho \rightarrow \sigma$,
y quiero concluir que $\Gamma \vdash \lambda y. M'\{x := N\} : \rho \rightarrow \sigma$.

Usando \rightarrow_i en $\Gamma, x : \tau \vdash \lambda y. M' : \rho \rightarrow \sigma$

$$\frac{\Gamma, x : \tau, y : \rho \vdash M' : \sigma}{\Gamma, x : \tau \vdash \lambda y. M' : \rho \rightarrow \sigma} \rightarrow_i$$

Obtengo $\Gamma, x : \tau, y : \rho \vdash M' : \sigma$

Y sé que, por **HI** aplicada al subtérmino M' que la derivación $\Gamma, y : \rho \vdash M'\{x := N\} : \sigma$ es cierta.

Entonces, usando nuevamente \rightarrow_i pero en $\Gamma, y : \rho \vdash M'\{x := N\} : \sigma$ obtengo

$$\frac{\Gamma, y : \rho \vdash M' \{x := N\} : \sigma}{\Gamma \vdash \lambda y. M' \{x := N\} : \rho \rightarrow \sigma} \rightarrow_i$$

como se quería probar.

Ahora sería necesario realizar lo mismo, pero para todo el resto de constructores de la gramática de M .

Bueno, pero esto era para el lema de sustitución, todavía no demostramos el teorema.

Jano dijo de hacer inducción sobre la definición de \rightarrow .

Pensemos cómo sería eso. Sabemos que hice $M \rightarrow N$, es porque utilicé alguna regla.

Podemos hacer inducción en las reglas utilizadas :).

Repasemos las reglas.

Reglas de resolución:

$$\begin{array}{lll} (\lambda x. M)N & \rightarrow & M\{x := N\} & (\beta \text{ reducción}) \\ \text{if tt then } M \text{ else } N & \rightarrow & M & (\text{if}_t) \\ \text{if ff then } M \text{ else } N & \rightarrow & N & (\text{if}_f) \end{array}$$

Reglas de congruencia (si $M \rightarrow N$ entonces):

$$\begin{array}{lll} MO & \rightarrow & NO & (\text{app}_l) \\ OM & \rightarrow & ON & (\text{app}_r) \\ \lambda x. M & \rightarrow & (\lambda x. N) & (\text{fun}) \\ \text{if } M \text{ then } O \text{ else } P & \rightarrow & \text{if } N \text{ then } O \text{ else } P & (\text{if}_c) \end{array}$$

Supongo $M \rightarrow N$ usando la regla β reducción.

Entonces $M = (\lambda x. M')N'$ y $N = M'\{x := N'\}$

Sé que $\Gamma \vdash (\lambda x. M')N' : \tau$, quiero probar que $\Gamma \vdash M'\{x := N'\} : \tau$.

De $\Gamma \vdash (\lambda x. M')N' : \tau$ puedo sacar que:

$$\frac{\Gamma \vdash (\lambda x. M') : \sigma \rightarrow \tau \quad \Gamma \vdash N' : \sigma}{\Gamma \vdash (\lambda x. M')N' : \tau} \rightarrow_e$$

Y de $\Gamma \vdash (\lambda x. M') : \sigma \rightarrow \tau$ (arriba a la izquierda en el árbol) puedo sacar que:

$$\frac{\Gamma, x : \sigma \vdash M' : \tau}{\Gamma \vdash (\lambda x. M') : \sigma \rightarrow \tau} \rightarrow_i$$

Finalmente, tengo $\Gamma, x : \sigma \vdash M' : \tau$ y $\Gamma \vdash N' : \sigma$.

Luego, por el **lema de sustitución**, sé que vale que $\Gamma \vdash M'\{x := N'\} : \tau$, que es lo que quería probar.

Igual que antes, me faltaría hacer el resto de casos para todas las reglas de reducción \rightarrow :).

Ejercicio 1.33

Nos piden extender la gramática, semántica operacional, y reglas de tipado, del cálculo lambda para pares.

Gramática:

$$M ::= \dots \mid (M, M) \mid fst(M) \mid snd(M)$$

Podemos definir también los valores si es que nos interesa.

$$V ::= \dots \mid (V, V)$$

Me voy a meter con la semántica al final. Vayamos antes con las **reglas de tipado**.

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash (M, N) : \sigma \times \tau} \text{par}$$

$$\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash fst(M) : \sigma} \text{par}_{fst} \quad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash snd(M) : \tau} \text{par}_{snd}$$

Para la **semántica operacional** propongo:

$$\overline{fst((V, W)) \rightarrow V} \quad \overline{snd((V, W)) \rightarrow W}$$

$$\frac{M \rightarrow M'}{(M, N) \rightarrow (M', N)} \quad \frac{M \rightarrow M'}{(V, M) \rightarrow (V, M')}$$

$$\frac{M \rightarrow M'}{fst(M) \rightarrow fst(M')} \quad \frac{M \rightarrow M'}{snd(M) \rightarrow snd(M')}$$

Nótese que agregar este conjunto de reglas no quita el **determinismo** (esto igual habría que demostrarlo formalmente) en la semántica operacional (para un término M cualquiera, hay una única regla que lo reduce). Seguro sirven para **CBV**, yo justifico (creo) que sirven también para **CBN**, pues formalmente la única diferencia es radica en que en **CBN** pasamos los argumentos a las λ s sin que sean necesariamente valores; fíjate como está definida la semántica operacional en el ejercicio 1.20 sino.

A su vez, me faltaría ponerles nombre a las reglas, pero soy medio queso con eso.

Ejercicio 2.2

Item 1: no se puede tipar.

Item 2:

Sea $I_1 = \lambda x : \sigma_1. x$ y $I_2 = \lambda x : \sigma_2. x$.

Determino $\sigma_1 = \sigma_2 \rightarrow \sigma_2$.

$$\frac{\frac{x : \sigma_2 \rightarrow \sigma_2 \vdash x : \sigma_2 \rightarrow \sigma_2}{\vdash \lambda x : \sigma_2 \rightarrow \sigma_2. x : (\sigma_2 \rightarrow \sigma_2) \rightarrow \sigma_2 \rightarrow \sigma_2} ax_v \rightarrow_i \quad \frac{x : \sigma_2 \vdash x : \sigma_2}{\vdash \lambda x : \sigma_2. x : \sigma_2 \rightarrow \sigma_2} ax_v \rightarrow_i}{\vdash I_1 I_2 : \sigma_2 \rightarrow \sigma_2} \rightarrow_e$$

Ejercicio 2.5

Lo defino de manera recursiva. Los primeros dos son los casos base, el resto son los casos recursivos.

$$FV(x) = \{x\}$$

$$FV(\text{Bool}) = \emptyset$$

$$FV(\tau_1 \rightarrow \tau_2) = FV(\tau_1) \cup FV(\tau_2)$$

$$FV(\forall X. \tau) = FV(\tau) \setminus \{X\}$$

Ejercicio 2.6

Este es un re chocio de escribí acá lesuento como es la idea.

Querés mostar que esta regla:

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

se puede obtener desde la derivación de $(\lambda x. M)N$.

Planteate $(\lambda x. M)N$ y andá aplicando reglas hasta llegar que necesitas las mismas premisas que para la regla **let**.

Nos estamos basando fuertemente en que $\text{let } x = N \text{ in } M$ reduce exáctamente igual que $(\lambda x. M)N$.

Ejercicio 2.10

El primer item es una rama del árbol del 2do así que lo salteo, deberías tomar $e = [X]$ y el tipo de la λ es $\forall X. [X \rightarrow X]$.

Segundo item:

$$\frac{\frac{\frac{x : [X] \vdash x : [X]}{x : [X] . x : [X \rightarrow X]} \text{ax}_v}{\vdash \lambda x : [X] . x : [X \rightarrow X]} \rightarrow_i}{\vdash \lambda x : [X] . x : \forall X. [X \rightarrow X]} \forall_i \quad \frac{\frac{\frac{\Gamma \vdash i : \forall X. [X \rightarrow X]}{\Gamma \vdash i : [(X \rightarrow X) \rightarrow X \rightarrow X]} \text{ax}_v}{\vdash i : \forall X. [X \rightarrow X]} \forall_e}{\vdash i : \forall X. [X \rightarrow X]} \forall_e \quad \frac{\frac{\Gamma \vdash i : \forall X. [X \rightarrow X]}{\Gamma \vdash i : [X \rightarrow X]} \text{ax}_v}{\vdash i : [X \rightarrow X]} \forall_e$$

$$\frac{\vdash \lambda x : [X] . x : \forall X. [X \rightarrow X]}{\vdash \text{let } i = \lambda x : [X] . x \text{ in } ii : [X \rightarrow X]} \text{let} \quad \frac{\vdash i : \forall X. [X \rightarrow X]}{\vdash i : [X \rightarrow X]} \text{let}$$

$$\frac{\vdash \text{let } i = \lambda x : [X] . x \text{ in } ii : [X \rightarrow X]}{\vdash \text{let } i = \lambda x : [X] . x \text{ in } ii : \forall X. [X \rightarrow X]} \forall_i$$

Notar que en la rama de la derecha, cuando aplicamos los dos \forall_e , se podría pensar en el reemplazo $[(X \rightarrow X) \rightarrow X \rightarrow X]\{Y := (X \rightarrow X)\}$, y luego te queda $\Gamma \vdash i : \forall Y. [Y \rightarrow Y]$, le dejo X por gusto, pero si la variable está ligada a un cuantificador, me da igual si se llama X o *PEPITO*.

En mi caso yo tomo $[(X \rightarrow X) \rightarrow X \rightarrow X]\{X := (X \rightarrow X)\}$, pero es medio raro de leer.

Los otros dos items tienen truquito.

El tercer item nos pide tipar $(\lambda f : e. ff)(\lambda x : e'. x)$, para algún e, e' .

Si sacamos los tipos y vemos como reduce, vemos que me termina quedando la identidad, por lo que el tipo debería ser $\forall X. [X \rightarrow X]$.

Por lo que vimos antes (*ej anterior*), $e' = [X]$ es una buena opción. El tema es e . Voy a necesitar que sea un esquema cuantificado, sino esto no va a salir. El problema es que si digo que $e = \forall X. [X \rightarrow X]$, me voy a trabar más adelante en una rama del árbol de derivación. Por lo que así como está, no es tipable.

Puedo pensar un término *análogo*, $\text{let } f = \lambda x : [X]. x \text{ in } ff : \forall X. [X \rightarrow X]$, que es literalmente el item 2.

Ojo, $(\lambda f : e. ff)(\lambda x : e'. x)$ no es el mismo término que $\text{let } f = \lambda x : [X]. x \text{ in } ff : \forall X. [X \rightarrow X]$, digo que es *análogo* en cierto grado pues sé que $\text{let } x = N \text{ in } M$ reduce exáctamente igual que $(\lambda x. M)N$.

El item 4 directamente no un término tipable de cálculo lambda, al igual que $(\lambda x. xx)$. No existe término análogo let que lo salve.

Ejercicio 2.19

$$(\mu f. \lambda x. fx)0 \rightarrow (\lambda x. (\mu f. \lambda x. fx)x)0 \rightarrow (\mu f. \lambda x. fx)0$$

No.

Ejercicio 2.20

Recuerdo el factorial original.

$$\text{Fact}_\mu = \mu f. \lambda n. \text{if } \text{isZero}(n) \text{ then } 1 \text{ else } n \times (f \text{ pred}(n))$$

Recuerdo lo que dice el apunte sobre **FIX** sin μ .

$$Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$$

Sea F una función cualquiera, $YF \rightarrow (\lambda x. F(xx))(\lambda x. F(xx)) \rightarrow F(YF)$. Ergo, YF es un punto fijo de F .

Ahora vamos con lo que pide el ejercicio.

$$\text{Tomo } F = \lambda f. \lambda n. \text{if } \text{isZero}(x) \text{ then } 1 \text{ else } n \times (f \text{ pred}(n)).$$

$$\text{Luego, } \text{Fact}_\mu = YF = Y(\lambda f. \lambda n. \text{if } \text{isZero}(n) \text{ then } 1 \text{ else } n \times (f \text{ pred}(n))).$$

Ejercicio 2.21

No es muy interesante.

Ejercicio 3.10

Bien, gracias. Sale haciendo inducción en estructural en los términos, o sea, inducción estructural en M . Planteate la gramática (está escrita más arriba) y resolvelo. Tu **HI** es que vale que para todos los subterminos que pueda contener M , si estos son cerrados irreducibles y bien tipados, entonces es un valor.

Ejercicio 3.11

Fíjate que Jano en la hoja siguiente lo hizo para *pred*. Es la misma idea pero para el resto de términos.

El *error* se propaga, al igual que el \perp .

Ejercicio 3.15

Lo mismo que el ejercicio de arriba pero con μ .

Y hasta acá llegué, los últimos ejercicios me quedé medio ¿?, si me sobra tiempo los hago.

Extras

Quiero hablar un poquito del punto fijo μ en semántica denotacional.

Tengo $[[\mu x : \tau. M]]_v = \text{FIX}(V^{[[\tau]]} \mapsto [[M]]_{v,x=V})$. Que para mí es chino básico.

El punto fijo del factorial es: (es un galerazo, ahora lo explico)

$$\text{FIX}(f \in \mathbb{N} \cup \{\perp\} \rightarrow \mathbb{N} \cup \{\perp\} \mapsto n \in \mathbb{N} \cup \{\perp\} \mapsto \begin{cases} 1 & \text{si } n = 0 \\ n \times f(n-1) & \text{sino} \end{cases})$$

¿Cómo razonamos esto?

Recordemos que el punto fijo de $\lambda f : \tau. M$ es $\mu f : \tau. M$. Esto es, por definición.

"¿Y ESTO QUÉ QUIERE DECIR 🤔🤔?"

Veamos primero la definición matemática.

Def: x es punto fijo de una función f si. $f(x) = x$.

Por ejemplo, la función identidad $\text{id}(x) = x$ tiene infinitos puntos fijos.

La función $f(x) = x + x$ tiene uno solo, con $x = 0$ pues $f(0) = 0$ y no hay otro x que lo cumpla.

Y $g(x) = x + 1$ no tiene ningún x que sea punto fijo.

Volviendo al cálculo- λ

Quiere decir que, si le paso $\mu f : \tau. M$ a $\lambda f : \tau. M$, obtengo nuevamente $\mu f : \tau. M$.

O sea, $M\{f := \mu f : \tau. M\} = \mu f : \tau. M$ análogo en matemáticas, $M(\mu \dots M) = \mu \dots M$

Ahora, esta es una definición, nada más.

Cuestión, suponete que hago $[[\text{Fact}_\mu]]$ con

$\text{Fact}_\mu = \mu f. \lambda n. \text{if } \text{isZero}(n) \text{ then } \underline{1} \text{ else } n \times (\text{f pred}(n))$.

Tenemos que pensar qué es $V^{[[\tau]]}$, es un elemento del tipo de f , por lo que tenemos

$f \in \mathbb{N} \cup \{\perp\} \rightarrow \mathbb{N} \cup \{\perp\}$.

Ahora nos queda $[[M]]_{v,x=V}$. Chequeate que es buscar la semántica denotacional de una lambda. Siguiendo la idea de arriba nos queda $n \in \mathbb{N} \cup \{\perp\} \mapsto \begin{cases} 1 & \text{si } n = 0 \\ n \times f(n-1) & \text{sino} \end{cases}$.

El $n \in \mathbb{N} \cup \{\perp\}$ sale de $V^{[[\tau]]}$, y lo otro de la recursión del subtermino.

Luego buscamos un punto fijo, agarramos el más chico que sería \perp , esto nos va a dar $FIX(n \in \mathbb{N} \cup \{\perp\}) \mapsto \begin{cases} 1 & \text{si } n = 0 \\ \perp & \text{sino} \end{cases}$ que no es la función constantemente \perp pues si $n = 0$, nos da 1.

En cambio si le pasamos el factorial como f , es decir, $Fact_\mu$, nos va a dar el punto fijo (es decir, $Fact_\mu$ para cualquier n). Nos quedaría:

$$FIX(n \in \mathbb{N} \cup \{\perp\}) \mapsto \begin{cases} 1 & \text{si } n = 0 \\ n \times Fact_\mu(n - 1) & \text{sino} \end{cases}$$

Que si lo pensás un cacho, es literalmente la definición matemática de factorial. Antes f era genérica, ahora es el factorial.

Compilación, inferencia de tipos, y máquinas abstractas (Capítulo 7)

En la sección anterior vimos cálculo- λ , la base de la programación funcional. Algo interesante que se menciona allí es el cálculo- λ **tipado**, a su vez, vimos como inferir el tipo de un programa mediante árboles de deducción.

Ahora, uno podría decir "muy lindo todo, pero yo quiero hacer esto en la compu..."; para eso existe esta sección. Veremos por encima que es un compilador, otra forma de inferencia de tipos, y dónde corre todo esto, una máquina abstracta.

Cuando termines con esta sección debería poder agarrar, por ejemplo, Haskell y crear tu propio "lenguaje" con base en cálculo lambda que corre en tu propia máquina abstracta hecha en Haskell.

Si estás leyendo esto en mi [GitLab](#), en la sección *Útil*, hay una [implementación de un compilador](#) de cálculo- λ hecho en Haskell que traduce a máquina abstracta SECD :).

Compilación (Capítulo 7: parte I)

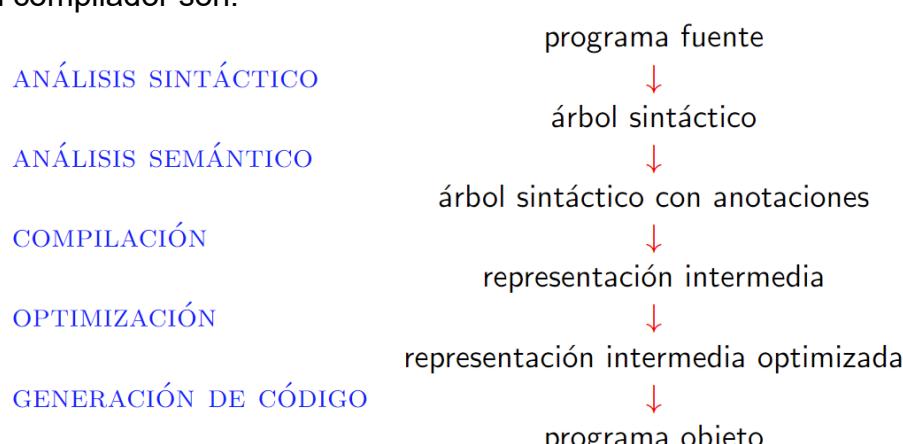
Def. (compilador): Un compilador es un programa que traduce programas.

- **Entrada:** programa escrito en un **lenguaje fuente**.
- **Salida:** programa escrito en un **lenguaje objeto**.

Este proceso de traducción debe **preservar la semántica**. Es decir, mantener el **significado** (*que nos interesa*).

Como motivación, queremos **traducir de alto nivel** (*C, Haskell, Python, Lisp, Erlang...*) a un lenguaje de **bajo nivel** como *Assembly*; aún así uno podría tener un compilador de C a Python y sería perfectamente válido.

Las faces de un compilador son:



Inferencia de tipos (Capítulo 7: parte II)

Entrada en calor

Clarifiquemos un poco la notación.

Tenemos términos **sin anotaciones**

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

y términos **con anotaciones**

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

Notamos $\text{erase}(M)$ al término sin anotaciones de tipos que resulta de borrar las anotaciones de tipos de M .

Ejemplo: $\text{erase}((\lambda x : \text{Bool}. x) \text{ True}) = (\lambda x. x) \text{ True}$

Decimos que un término U sin anotaciones de tipos es **tipable** si existen:

- un contexto de tipado Γ
- un término con anotaciones de tipos M
- un tipo τ

tales que $\text{erase}(M) = U$ y $\Gamma |- M : \tau$.

Queremos un **algoritmo de inferencia de tipos** que dado un término U me determine si es tipable, y si lo es, que me halle un contexto Γ , un término M , y un tipo τ tales que $\text{erase}(M) = U$ y $\Gamma |- M : \tau$.

Sin embargo, antes de adentrarnos en ese lugar, tenemos que aprender sobre el **algoritmo de unificación...**

Y antes de aprender sobre el algoritmo de unificación, tenemos que aprender sobre **unificación**, es decir, qué es **unificar**.

Unificación

Introduzcamos una noción, la de incógnita, tal que una incógnita se denota como $?n$ para algún $n \in \mathbb{N}^+$.

Supongamos que queremos determinar el tipo de f , y sabemos que f devuelve obligatoriamente un Bool . Podemos decir que $f : ?1 \rightarrow \text{Bool}$.

Y a su vez sabemos que f debe tomar un \mathbb{N} , entonces podemos decir que $f : \text{Nat} \rightarrow ?2$.
Guion de notación a semántica denotacional (pues usé Bool y \mathbb{N})

Estas dos **ecuaciones** son independientes. Razonablemente uno podría pensar ahora que f debe ser de tipo $f : \text{Nat} \rightarrow \text{Bool}$.

La idea de la unificación se basa en **manipular tipos parcialmente conocidos**. Para esto, se incorporan incógnitas (*representadas como $?1, ?2, ?3, etc.$*) a los tipos.

En esta sección trabajaremos únicamente con **dos** ecuaciones de tipos (*salvo por las definiciones*), después se generalizará.

Para resolver estas incógnitas, es necesario resolver ecuaciones entre tipos. Por ejemplo:

$$1. (?1 \rightarrow \text{Bool}) = ((\text{Bool} \rightarrow \text{Bool}) \rightarrow ?2)$$

Tiene solución: $?1 := (\text{Bool} \rightarrow \text{Bool})$ y $?2 := \text{Bool}$.

$$2. (?1 \rightarrow ?1) \stackrel{?}{=} ((Bool \rightarrow Bool) \rightarrow ?2)$$

Tiene solución: $?1 := (Bool \rightarrow Bool)$ y $?2 := (Bool \rightarrow Bool)$.

$$3. (?1 \rightarrow Bool) = ?1$$

No tiene solución.

Este proceso de resolución de ecuaciones entre tipos con incógnitas va ser fundamental para el **algoritmo de unificación** y **algoritmo de inferencia de tipos** (*ambos los veremos más adelante, pero en este capítulo, no te emociones tanto*).

Pensemos un poco en la gramática de los tipos ahora que agregamos las incógnitas...

Supongamos fijado un conjunto finito de **constructores de tipos**:

Constantes: $Bool, Nat, \dots$

Unarios: $(Lista \bullet), (Maybe \bullet)$

Binarios: $(\bullet \rightarrow \bullet), (\bullet \times \bullet), (Either \bullet \bullet)$

Y la lista sigue...

Su gramática queremos que se defina usando incógnitas $?n$ y constructores, es decir:

$$\tau ::= ?n \mid C(\tau_1, \dots, \tau_n)$$

Llamamos **sustitución** a una función que a cada incógnita le asocia un tipo.

Notamos: $S = \{?k_1 := \tau_1, \dots, ?k_n := \tau_n\}$ a la sustitución S .

$S(k_i) = \tau_i$ para cada $1 \leq i \leq n$, y $S(w) = w$ para el resto.

En criollo, si k_i pertenece a S , te devuelve τ_i ; si no pertenece te devuelve lo mismo que le diste.

Un **problema de unificación** es un conjunto finito E de ecuaciones entre tipos que pueden involucrar incógnitas, $E = \{\tau_1 = \sigma_1, \tau_2 = \sigma_2, \dots, \tau_n = \sigma_n\}$.

O sea, es el conjunto de ecuaciones que queremos despejar; en el ejemplo de f sería $E = \{(?1 \rightarrow Bool) = ?(Nat \rightarrow ?2)\}$

Y un **unificador** para E es una sustitución S tal que: $S(\tau_1) = S(\sigma_1), \dots, S(\tau_n) = S(\sigma_n)$.

En el ejemplo de f sería $S(?1 \rightarrow Bool) = S(Nat \rightarrow ?2)$.

Notar que acá funcaría $S = \{?1 := Nat, ?2 := Bool\}$ como sustitución. Ya veremos cómo encuentra ese S , que no panda el cúnico.

Supongamos que contamos con el siguiente problema de unificación: $E = \{(?1 \rightarrow Bool) = ?2\}$

Uno acá podría tirar varios $S = \{algo\}$, es por ello que metemos una noción más. La noción del **unificador más general (MGU)**

Formalmente se dice que: una susitución S_A es más general 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 .

Pero yo sé que a vos no te gusta la formalidad, así que te lo hago tipo [r/ExplainLikeImFive](#).

Vos querés encontrar una sustitución para el unificador E , pero querés la más general, ¿a qué te suena?, a que no querés pedir de más, querés la sustitución "más débil" por así decirlo.

Ponele, si yo le pido a las incógnitas que: $?1 := Int$ y que $?2 := (Int \rightarrow Bool)$ es medio un montón, ¿de dónde saqué que $?1$ tenga que ser de tipo Int ? es un contra galerazo.

Vamos a querer unificar de forma tal que valga la igualdad $S(\tau_i) = S(\sigma_i)$, pero pidiendo lo menos posible.

"Bueno, bueno, pero decime cómo lo hago che..."

Más adelante vamos a ver (literalmente en la siguiente sección), por ahora quedate con que

vine yo y te dije que el **MGU** es $S = \{?2 := (?1 \rightarrow \text{Bool})\}$, podés intentar pensar el motivo, pero no te quemes porque en 5 minutos vas a tener todo el material para poder deducirlo.

Uno podría preguntarse si $S = \{?1 := ?3, ?2 := (?3 \rightarrow \text{Bool})\}$ es también un **MGU** válido. La respuesta es que no, por la definición de **MGU**.

Tomá:

- $S_A = \{?2 := (?1 \rightarrow \text{Bool})\}$
- $S_B = \{?1 := ?3, ?2 := (?3 \rightarrow \text{Bool})\}$
- $S_C = \{?1 := ?3\}$

Te va a quedar que S_A es más general que S_B pues $S_B = S_C \circ S_A$.

Algoritmo de unificación (de Martelli–Montanari)

Ahora que estás familiarizado con el concepto de unificación de términos (o tipos, porque los tipos son términos). Planteamos el siguiente algoritmo dado un **problema de unificación** E (conjunto de ecuaciones a despejar):

- Mientras $E \neq \emptyset$, se aplica sucesivamente alguna de las reglas que veremos a continuación.
 - La regla nos puede devolver un nuevo problema de unificación E' , o una *falla*.
- Si la regla nos devuelve una *falla*, el problema de unificación no tiene solución.
- De lo contrario la regla es de la forma $E \rightarrow_S E'$.
La resolución del problema E se reduce a resolver otro problema E' , aplicando la sustitución S .

Cuestión, que hay dos posibilidades.

$$1. E = E_0 \rightarrow_{S_1} E_1 \rightarrow_{S_2} E_2 \rightarrow_{S_3} \dots \rightarrow_{S_n} E_n \rightarrow_{S_{n+1}} \text{falla}$$

El problema de unificación E no tiene solución

$$2. E = E_0 \rightarrow_{S_1} E_1 \rightarrow_{S_2} E_2 \rightarrow_{S_3} \dots \rightarrow_{S_n} E_n = \emptyset$$

El problema de unificación E tiene solución.

Este algoritmo está demostrado que es **correcto**. Es decir:

- Termina para cualquier E .
- Si E tiene solución, la encuentra.
 - Se puede reconstruir la solución con $S = S_n \circ S_{n-1} \circ \dots \circ S_2 \circ S_1$.
- Si E no tiene solución, llega a una *falla*.

Y se denota $\text{mgu}(E)$ al unificador más general de E , si existe.

Reglas:

$$\begin{array}{ccc} \{x \stackrel{?}{=} x\} \cup E & \xrightarrow{\text{Delete}} & E \\ \{C(\tau_1, \dots, \tau_n) \stackrel{?}{=} C(\sigma_1, \dots, \sigma_n)\} \cup E & \xrightarrow{\text{Decompose}} & \{\tau_1 \stackrel{?}{=} \sigma_1, \dots, \tau_n \stackrel{?}{=} \sigma_n\} \cup E \\ \{\tau \stackrel{?}{=} ?n\} \cup E & \xrightarrow{\text{Swap}} & \{?n \stackrel{?}{=} \tau\} \cup E \\ & & \text{si } \tau \text{ no es una incógnita} \\ \{?n \stackrel{?}{=} \tau\} \cup E & \xrightarrow{\text{Elim}}_{\{?n := \tau\}} & E' = \{?n := \tau\}(E) \\ & & \text{si } ?n \text{ no ocurre en } \tau \\ \{C(\tau_1, \dots, \tau_n) \stackrel{?}{=} C'(\sigma_1, \dots, \sigma_m)\} \cup E & \xrightarrow{\text{Clash}} & \begin{array}{l} \text{falla} \\ \text{si } C \neq C' \end{array} \\ \{?n \stackrel{?}{=} \tau\} \cup E & \xrightarrow{\text{Occurs-Check}} & \begin{array}{l} \text{falla} \\ \text{si } ?n \neq \tau \\ \text{y } ?n \text{ ocurre en } \tau \end{array} \end{array}$$

Algoritmo \mathbb{W} de inferencia de tipos (finalmente)

El algoritmo \mathbb{W} recibe un término U sin anotaciones de tipos y procede recursivamente sobre la estructura de U :

- Puede fallar, indicando que U no es tipable.
- Si tiene éxito, devuelve una tripla (Γ, M, τ) , donde $\text{erase}(M) = U$ y $\Gamma \vdash M : \tau$ es válido.

Escribimos $\mathbb{W}(U) \rightsquigarrow \Gamma \vdash M : \tau$ para indicar que el algoritmo de inferencia tiene éxito cuando se le pasa U como entrada y devuelve una tripla (Γ, M, τ) .

Las reglas del algoritmo \mathbb{W} son:

1. Para constantes booleanas:

$$\overline{\mathbb{W}(\text{True}) \rightsquigarrow \emptyset \vdash \text{True} : \text{Bool}}$$

$$\overline{\mathbb{W}(\text{False}) \rightsquigarrow \emptyset \vdash \text{False} : \text{Bool}}$$

2. Para variables:

$$\overline{\mathbb{W}(x) \rightsquigarrow x : ?k \vdash x : ?k}$$

donde $?k$ es una incógnita fresca.

3. Para condicionales:

$$\frac{\mathbb{W}(U_1) \rightsquigarrow \Gamma_1 \vdash M_1 : \tau_1 \quad \mathbb{W}(U_2) \rightsquigarrow \Gamma_2 \vdash M_2 : \tau_2 \quad \mathbb{W}(U_3) \rightsquigarrow \Gamma_3 \vdash M_3 : \tau_3}{\mathbb{W}(\text{if } U_1 \text{ then } U_2 \text{ else } U_3) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \cup S(\Gamma_3) \vdash S(\text{if } M_1 \text{ then } M_2 \text{ else } M_3) : S(\tau_2)}$$

donde $S = \text{mgu}(\{\tau_1 = ?\text{Bool}, \tau_2 = ?\tau_3\} \cup \{\Gamma_i(x) = ?\Gamma_j(x) \mid i, j \in \{1, 2, 3\}, x \in \Gamma_i \cap \Gamma_j\})$.

4. Para aplicaciones:

$$\frac{\mathbb{W}(U) \rightsquigarrow \Gamma_1 \vdash M : \tau \quad \mathbb{W}(V) \rightsquigarrow \Gamma_2 \vdash N : \sigma}{\mathbb{W}(UV) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \vdash S(MN) : S(?k)}$$

donde $?k$ es una incógnita fresca y

$$S = \text{mgu}(\{\tau = ?\sigma \rightarrow ?k\} \cup \{\Gamma_1(x) = ?\Gamma_2(x) : x \in \Gamma_1 \cap \Gamma_2\}).$$

5. Para abstracciones:

$$\frac{\mathbb{W}(U) \rightsquigarrow \Gamma \vdash M : \tau}{\mathbb{W}(\lambda x. U) \rightsquigarrow \Gamma \ominus \{x\} \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau}$$

donde $\sigma = \begin{cases} \Gamma(x) & \text{si } x \in \Gamma \\ \text{una incógnita fresca } ?k & \text{si no} \end{cases}$.

6. Para naturales:

$$\frac{\mathbb{W}(U) \rightsquigarrow \Gamma \vdash M : \sigma}{\mathbb{W}(\text{succ}(U)) \rightsquigarrow S(\Gamma) \vdash S(\text{succ}(M)) : \text{Nat}}$$

$$\frac{\mathbb{W}(U) \rightsquigarrow \Gamma \vdash M : \sigma}{\mathbb{W}(\text{pred}(U)) \rightsquigarrow S(\Gamma) \vdash S(\text{pred}(M)) : \text{Nat}}$$

$$\frac{\mathbb{W}(U) \rightsquigarrow \Gamma \vdash M : \sigma}{\mathbb{W}(\text{izZero}(U)) \rightsquigarrow S(\Gamma) \vdash S(\text{isZero}(M)) : \text{Bool}}$$

donde $S = \text{mgu}(\{\sigma = ?\text{Nat}\})$ en todos los casos.

7. Para μ (recursión):

$$\frac{\mathbb{W}(U) \rightsquigarrow \Gamma \vdash M : \sigma}{\mathbb{W}(\mu x. U) \rightsquigarrow S(\Gamma) \ominus \{x\} \vdash S(\mu x. M) : S(\sigma)}$$

donde $S = mgu(\{\sigma \stackrel{?}{=} \tau\})$ y $\tau = \begin{cases} \Gamma(x) & \text{si } x \in \Gamma \\ \text{una incógnita fresca } ?k & \text{si no} \end{cases}$

El algoritmo \mathbb{W} es correcto:

1. Si U no es tipable, $\mathbb{W}(U)$ falla al resolver alguna unificación.
2. Si U es tipable, $\mathbb{W}(U) \rightsquigarrow \Gamma \vdash M : \tau$, donde $erase(M) = U$ y $\Gamma \vdash M : \tau$ es un juicio válido.

Además, $\Gamma \vdash M : \tau$ es el juicio de tipado más general posible. Es decir, si $\Gamma' \vdash M' : \tau'$ es un juicio válido y $erase(M') = U$, existe una sustitución S tal que:

- $\Gamma' \supseteq S(\Gamma)$
- $M' = S(M)$
- $\tau' = S(\tau)$

Tip práctico: generalmente para un ejercicio de inferencia de tipos con el algoritmo \mathbb{W} está bueno armarte el árbol de subtérminos, después agarrás y vas resolviendo $\mathbb{W}(\text{hoja})$ para toda las hojas, y así para arriba hasta llegar a la raíz.

Así vas reconstruyendo para arriba la solución evitás confundirte y marearte.

Máquinas abstractas (Capítulo 7: parte III)

Bueno, ya tenemos una base sólida para crear nuestro propio lenguaje. ¡Pero no tenemos dónde correrlo!, de eso se trata esta parte del capítulo :).

Si cursaste alguna materia de Sistemas (por ejemplo, Sistemas Digitales), seguramente este capítulo te sea más ameno.

Imaginemos una máquina con tres instrucciones (*como las que brinda la arquitectura de un procesador, por ejemplo*):

$$LDI(n) \mid ADD \mid MUL$$

Un programa ℓ (*lista de instrucciones*) opera sobre una pila π (*ponele la memoria*):

$LDI(n) : \ell$	π	\longrightarrow	ℓ	$n : \pi$
$ADD : \ell$	$m : n : \pi$	\longrightarrow	ℓ	$(n + m) : \pi$
$MUL : \ell$	$m : n : \pi$	\longrightarrow	ℓ	$(n * m) : \pi$

Consideremos la siguiente gramática de expresiones aritméticas:

$$E ::= n \mid E + E \mid E * E$$

Una expresión se puede **compilar** a una lista de instrucciones:

$$\begin{aligned} \mathcal{C}\{n\} &= LDI(n) \\ \mathcal{C}\{E_1 + E_2\} &= \mathcal{C}\{E_1\}; \mathcal{C}\{E_2\}; ADD \\ \mathcal{C}\{E_1 * E_2\} &= \mathcal{C}\{E_1\}; \mathcal{C}\{E_2\}; MUL \end{aligned}$$

Ejemplito de compilación

$$\begin{aligned}
 & \textcolor{red}{C}\{(2 * 3) + (4 * 5)\} \\
 &= \text{LDI}(2) : \text{LDI}(3) : \text{MUL} : \text{LDI}(4) : \text{LDI}(5) : \text{MUL} : \text{ADD}
 \end{aligned}$$

código	pila
LDI(2) : LDI(3) : MUL : LDI(4) : LDI(5) : MUL : ADD	[]
→ LDI(3) : MUL : LDI(4) : LDI(5) : MUL : ADD	2
→ MUL : LDI(4) : LDI(5) : MUL : ADD	3 : 2
→ LDI(4) : LDI(5) : MUL : ADD	6
→ LDI(5) : MUL : ADD	4 : 6
→ MUL : ADD	5 : 4 : 6
→ ADD	20 : 6
→ []	26

Bueno pero este fue un ejemplo medio de juguete. Hagamos algo con cálculo- λ .

Queremos definir un compilador tal que:

- Lenguaje fuente: términos del cálculo- λ con booleanos.
- Lenguaje objeto: código para la [máquina abstracta SECD](#).

El compilador implementa la estrategia **call-by-value**.

La evaluación de una aplicación MN es de izquierda a derecha.

Una **máquina abstracta** es una abstracción de la arquitectura real.

Es sencillo traducir su código a un lenguaje de bajo nivel.

La máquina SECD

La máquina SECD trabaja con los siguientes tipos de datos:

CÓDIGO	$\ell ::= [] i : \ell$	lista de instrucciones
VALORES	$v ::= tt ff \langle \ell, e \rangle$	booleanos y clausuras
ENTORNOS	$e ::= [] v : e$	lista de valores
PILAS	$\pi ::= [] v : \pi$	lista de valores
DUMPS	$d ::= [] \langle \ell, \pi, d \rangle : d$	

Las instrucciones de la máquina son:

LDB(b)	$(b \in \{tt, ff\})$	carga un booleano en la pila
MKCL0(ℓ)		crea una clausura en la pila
LD(n)	$(n \in \mathbb{N})$	carga un valor del entorno en la pila
AP		invoca a una función
RET		retorna de una función
TEST(ℓ_1, ℓ_2)		ejecuta condicionalmente ℓ_1 ó ℓ_2

Un estado de la máquina es una 4-upla (ℓ, π, e, d) . Lo notamos así:

código	pila	entorno	dump
--------	------	---------	------

Ejemplo de ejecución en la máquina abstracta SECD

LD(1) : LD(0) : AP	[]	tt : $\langle LD(0) : RET, e \rangle$	[]
→	LD(0) : AP	$\langle LD(0) : RET, e \rangle$	tt : $\langle LD(0) : RET, e \rangle$ []
→	AP	tt : $\langle LD(0) : RET, e \rangle$	tt : $\langle LD(0) : RET, e \rangle$ []
→	LD(0) : RET	[]	tt : $\langle [], [], tt : \langle LD(0) : RET, e \rangle \rangle$ []
→	RET	tt : $\langle [], [], tt : \langle LD(0) : RET, e \rangle \rangle$	[]
→	[]	tt : $\langle LD(0) : RET, e \rangle$	[]

Compilación del cálculo- λ a la máquina SECD

Definición del compilador

Dada una lista de variables $\omega = [z_1, \dots, z_n]$ y un término M del cálculo- λ , definimos una lista de instrucciones $C_\omega\{M\}$ por recursión estructural sobre M :

$$\begin{aligned} C_\omega\{\text{True}\} &= \text{LDB(tt)} \\ C_\omega\{\text{False}\} &= \text{LDB(ff)} \\ C_\omega\{\text{if } M \text{ then } N \text{ else } P\} &= C_\omega\{M\}; \text{TEST}(C_\omega\{N\}, C_\omega\{P\}) \\ C_\omega\{x\} &= \text{LD}(i) \\ &\quad \text{si } i \text{ es el menor índice t.q. } \omega[i] = x \\ C_\omega\{\lambda x. M\} &= \text{MKCLO}(C_{x:\omega}\{M\}; \text{RET}) \\ C_\omega\{M\ N\} &= C_\omega\{M\}; C_\omega\{N\}; \text{AP} \end{aligned}$$

Ejemplo de compilación

$$\begin{aligned} C_{[]}(\lambda x. \lambda y. x) \text{ True False} &= C_{[]}(\lambda x. \lambda y. x) \text{ True}; C_{[]} \text{ False}; \text{AP} \\ &= C_{[]}(\lambda x. \lambda y. x); C_{[]} \text{ True}; \text{AP}; C_{[]} \text{ False}; \text{AP} \\ &= \text{MKCLO}(C_{[]}(\lambda y. x); \text{RET}); C_{[]} \text{ True}; \text{AP}; C_{[]} \text{ False}; \text{AP} \\ &= \text{MKCLO}(\text{MKCLO}(C_{[v,x]}\{x\}; \text{RET}); \text{RET}); C_{[]} \text{ True}; \text{AP}; C_{[]} \text{ False}; \text{AP} \\ &= \text{MKCLO}(\text{MKCLO}(\text{MKCLO}(\text{LD}(1); \text{RET}); \text{RET}); C_{[]} \text{ True}); \text{AP}; C_{[]} \text{ False}; \text{AP} \\ &= \text{MKCLO}(\text{MKCLO}(\text{MKCLO}(\text{LD}(1); \text{RET}); \text{RET}); \text{LDB(tt)}; \text{AP}; \text{LDB(ff)}; \text{AP} \end{aligned}$$

Corrección del compilador (teorema)

Sea M un término cerrado del cálculo- λ y v un valor booleano.

$M \rightarrow^* v$ es equivalente a

$$\boxed{C_{[]} \{ M \} \quad \boxed{\quad} \quad \boxed{\quad} \quad \boxed{\quad}} \longrightarrow^* \boxed{\quad} \quad v : \pi \quad e \quad d$$

para ciertos π, e, d .

Lógica de primer orden (Capítulo 8)

Bueno, ya vimos mucho del paradigma funcional. Este capítulo es el 101 que nos va a servir para poder pasar al paradigma lógico. Cubriremos la noción de lógica de primer orden, tema que, va más allá de la programación lógica.

Veníamos trabajando con **lógica proposicional** (*ahora vas y recordás el capítulo 3*), esta lógica nos permite razonar sobre **proposiciones**, ejemplo: $P \vee \neg P$ con P proposición.

Ahora vamos a trabajar con **lógica de primer orden**, permite **razonar acerca de elementos** sobre los que se **predica**, ejemplo: $\forall X. (P(X) \implies \neg P(\text{término}(X)))$.

Extiende a la lógica proposicional con **términos y cuantificadores**.

"Daaa Valen, ¿qué es un término?..."

Ya vamos con la gramática, tranqui palanqui. Dejame explayar un poco mejor el motivo de estudiar **LPO** en este curso. Aunque actualmente hay banda de motivos, voy a restringirme a lo que nos interesa, en **AED** tuviste que haber visto especificación, la idea es que queremos que los programas sean similares a ese mundo, onda copiar y pegar la especificación en código y que la compu lo resuelva.

Venimos laburando con **programación declarativa** hace rato "**define el ¿qué? y no el ¿cómo?**", seguro lo escuchaste 80 veces. Bueno, la idea de la programación declarativa, es que **los programas se asemejen lo más posible a las especificaciones**. A su vez, dentro de la programación declarativa, encuentra el **paradigma funcional** y el **paradigma lógico**. El funcional ya lo vimos, ahora **vamos con el lógico**.

Psst..., si te gusta todo eso de programación declarativa, chequeate [NixOS](#), un S.O. GNU/Linux que es declarativo en lugar de caótico e imperativo :).

Volviendo a lo nuestro. Estamos parados en la programación lógica, queremos que el usuario (*programador*) escriba una fórmula como por ejemplo $\exists X. P(X)$, y el sistema busca satisfacer o refutar la fórmula.

En caso de lograr satisfacerla, el sistema produce una salida que verifica la propiedad P buscada.

Sintaxis de la lógica de primer orden

Un **lenguaje de primer orden** \mathcal{L} está dado por:

1. Un conjunto de **símbolos de función** $\mathcal{F} = \{f, g, h, \dots\}$
Cada símbolo de función tiene asociada una aridad (≥ 0).
2. Un conjunto de **símbolos de predicado** $\mathcal{P} = \{P, Q, R, \dots\}$.
Cada símbolo de predicado tiene asociada una aridad (≥ 0).

Suponemos fijado un lenguaje de primer orden \mathcal{L} , y un conjunto infinito numerable de **variables** $\mathcal{X} = \{X, Y, Z, \dots\}$.

El conjunto \mathcal{T} de **términos** se define por la siguiente gramática:

$$t ::= X \mid f(t_1, \dots, t_n)$$

donde:

- X denota una variable.
- f denota un símbolo de función de aridad n .

Es decir, los términos son o bien variables, o bien funciones de aridad $0, 1, 2, \dots, n$. Y adentro de las funciones van otros términos (o sea: o variables, o funciones).

Ejemplo — el lenguaje $\mathcal{L}_{\text{aritmética}}$

$$\underbrace{\begin{array}{cccc} 0^0 & \text{succ}^1 & +^2 & *^2 \\ & \text{símbolos de función} & & \end{array}}_{\text{símbolos de función}} \quad \underbrace{\begin{array}{cc} =^2 & <^2 \\ & \text{símbolos de predicado} \end{array}}_{\text{símbolos de predicado}}$$

Ejemplo — términos sobre el lenguaje $\mathcal{L}_{\text{aritmética}}$

$$+(0, \text{succ}(X)) \quad *(+(X, Y), Z)$$

Los símbolos de función de aridad 0 se llaman constantes.

Nota. Usamos notación infija como conveniencia.

Bueno, yo dije que esto era una extensión de la lógica proposicional (prefiero decir que la lógica proposicional es un caso particular de LPO, pero bueno). Cuestión, en el capítulo 3 vimos la **gramática de las fórmulas** en lógica proposicional.

Extendámosla para lógica de primer orden.

$\sigma ::=$	$\mathbf{P}(t_1, \dots, t_n)$	fórmula atómica
	\perp	contradicción
	$\sigma \Rightarrow \sigma$	implicación
	$\sigma \wedge \sigma$	conjunción
	$\sigma \vee \sigma$	disyunción
	$\neg \sigma$	negación
	$\forall X. \sigma$	cuantificación universal
	$\exists X. \sigma$	cuantificación existencial

\mathbf{P} denota un símbolo de predicado de aridad n .

Los cuantificadores ligan una variable X .

Ejemplos de fórmulas

$$\forall X. \exists Y. = (+ (X, Y), 0)$$

$$\forall X. \forall Y. (\text{succ}(X) = \text{succ}(Y) \Rightarrow X = Y)$$

$$\forall X. (X < 0 \vee X = 0 \vee 0 < X)$$

Sean:

$$\sigma : \equiv \text{succ}(X) = Y \implies \exists Z. X + Z = Y$$

entonces:

$$\sigma\{X := Z * Z\} \equiv \text{succ}(Z * Z) = Y \implies \exists Z'. (Z * Z) + Z' = Y$$

Notar el renombramiento de Z a Z' para evitar la captura de variables.

Deducción natural

Sí, otra vez; **it's back**, it's back in full force.

Vamos a extender lo que vimos de lógica proposicional a lógica de primer orden, y una de las cosas que vimos fue deducción natural.

Se mantiene que

- Un **contexto** Γ es un conjunto finito de fórmulas.
- Un **secuente** es de la forma $\Gamma \vdash \sigma$.

Todas las reglas de deducción natural proposicional siguen vigentes, se agregan reglas: la introducción y eliminación del \forall y \exists .

Introducción del \forall

$$\frac{\Gamma \vdash \sigma \quad X \notin \text{fv}(\Gamma)}{\Gamma \vdash \sigma\{X := t\}} \forall_I$$

Eliminación del \forall

$$\frac{\Gamma \vdash \forall X. \sigma}{\Gamma \vdash \sigma\{X := t\}} \forall_E$$

Introducción del \exists

$$\frac{\Gamma \vdash \sigma\{X := t\}}{\Gamma \vdash \exists X. \sigma} \exists_I$$

Eliminación del \exists

$$\frac{\Gamma \vdash \exists X. \sigma \quad \Gamma, \sigma \vdash \tau \quad X \notin \text{fv}(\Gamma, \tau)}{\Gamma \vdash \sigma} \exists_E$$

Semántica de la lógica de primer orden

Bueno, hasta acá era todo muy color de rosas. Esta es la parte más densa del capítulo; vamos a meterle semántica a la cosa.

Definamos primero qué es una **estructura de primer orden**.

Una **estructura de primer orden** es un par $\mathcal{M} = (M, I)$ donde:

- M es un conjunto **no vacío**, llamado *universo*.
- I es una **función** que a le da una *interpretación a cada símbolo*.
- Para cada símbolo de **función f** de aridad n :

$$I(f) : M^n \rightarrow M$$

- Para cada símbolo de **predicado P** de aridad n :

$$I(P) \subseteq M^n$$

<p>Ejemplo — una estructura sobre $\mathcal{L}_{\text{aritmética}}$ $M := \mathbb{N}$ (los elementos son números naturales)</p> $\begin{array}{lll} I(0) & = 0 & (n, m) \in I(=) \iff n = m \\ I(\text{succ})(n) & = n + 1 & \\ I(+)(n, m) & = n + m & (n, m) \in I(<) \iff n < m \\ I(*)(n, m) & = n \cdot m & \end{array}$ <p>Bajo esta estructura, la fórmula $\forall X. X = X + X$ es falsa.</p>	<p>Ejemplo — otra estructura sobre $\mathcal{L}_{\text{aritmética}}$ $M := \mathcal{P}(\mathbb{R})$ (los elementos son conjuntos de números reales)</p> $\begin{array}{lll} I(0) & = \emptyset & (A, B) \in I(=) \iff A = B \\ I(\text{succ})(A) & = \{1 + x \mid x \in A\} & \\ I(+)(A, B) & = A \cup B & (A, B) \in I(<) \iff A \subseteq B \\ I(*)(A, B) & = A \cap B & \end{array}$ <p>Bajo esta estructura, la fórmula $\forall X. X = X + X$ es verdadera.</p>
--	---

Interpretación de términos

Suponemos fijada una estructura de primer orden $\mathcal{M} = (M, I)$.

Definición

Una **asignación** es una función que a cada variable le asigna un elemento del universo:

$$\alpha : \mathcal{X} \rightarrow M$$

Definición – interpretación de términos

Cada término $t \in \mathcal{T}$ se interpreta como un elemento $\alpha(t) \in M$, extendiendo la definición de α a términos:

$$\alpha(f(t_1, \dots, t_n)) = I(f)(\alpha(t_1), \dots, \alpha(t_n))$$

Interpretación de fórmulas

Suponemos fijada una estructura de primer orden $\mathcal{M} = (M, I)$.

Definimos una relación de **satisfacción** $\alpha \models_{\mathcal{M}} \sigma$.

“La asignación α (bajo la estructura \mathcal{M}) satisface la fórmula σ ”.

$\alpha \models_{\mathcal{M}} \mathbf{P}(t_1, \dots, t_n)$	sii	$(\alpha(t_1), \dots, \alpha(t_n)) \in I(\mathbf{P})$
$\alpha \models_{\mathcal{M}} \sigma \wedge \tau$	sii	$\alpha \models_{\mathcal{M}} \sigma$ y $\alpha \models_{\mathcal{M}} \tau$
$\alpha \models_{\mathcal{M}} \sigma \vee \tau$	sii	$\alpha \models_{\mathcal{M}} \sigma$ o $\alpha \models_{\mathcal{M}} \tau$
$\alpha \models_{\mathcal{M}} \sigma \Rightarrow \tau$	sii	$\alpha \not\models_{\mathcal{M}} \sigma$ o $\alpha \models_{\mathcal{M}} \tau$
$\alpha \not\models_{\mathcal{M}} \perp$		
$\alpha \models_{\mathcal{M}} \forall X. \sigma$	sii	$\alpha[X \mapsto m] \models_{\mathcal{M}} \sigma$ para todo $m \in M$
$\alpha \models_{\mathcal{M}} \exists X. \sigma$	sii	$\alpha[X \mapsto m] \models_{\mathcal{M}} \sigma$ para algún $m \in M$

Validez y satisfactibilidad

Decimos que una fórmula σ es:

VÁLIDA si $\alpha \models_{\mathcal{M}} \sigma$ para toda \mathcal{M}, α	SATISFACTIBLE si $\alpha \models_{\mathcal{M}} \sigma$ para alguna \mathcal{M}, α
INVÁLIDA si $\alpha \not\models_{\mathcal{M}} \sigma$ para alguna \mathcal{M}, α	INSATISFACTIBLE si $\alpha \not\models_{\mathcal{M}} \sigma$ para toda \mathcal{M}, α

$$\begin{array}{lll} \sigma \text{ es VÁLIDA} & \text{sii} & \sigma \text{ no es INVÁLIDA} \\ \sigma \text{ es SATISFACTIBLE} & \text{sii} & \sigma \text{ no es INSATISFACTIBLE} \\ \sigma \text{ es VÁLIDA} & \text{sii} & \neg\sigma \text{ es INSATISFACTIBLE} \\ \sigma \text{ es SATISFACTIBLE} & \text{sii} & \neg\sigma \text{ es INVÁLIDA} \end{array}$$

Modelos

Una *sentencia* es una fórmula σ sin variables libres.

Una *teoría de primer orden* es un conjunto de sentencias.

Definición — consistencia

Una teoría \mathcal{T} es *consistente* si $\mathcal{T} \not\vdash \perp$.

Definición — modelo

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

Corrección y completitud (sí, como en el capítulo 3)

Teorema (Gödel, 1929)	Corolario	Corolario
Dada una teoría \mathcal{T} , son equivalentes: 1. \mathcal{T} es consistente. 2. \mathcal{T} tiene (al menos) un modelo.	Corolario Dada una fórmula σ , son equivalentes: 1. $\vdash \sigma$ es derivable. 2. σ es válida.	Corolario Dada una fórmula σ , son equivalentes: 1. $\vdash \neg\sigma$ es derivable. 2. σ es insatisfacible.

Ejemplo

Determinar si son (in)válidas/(in)satisfactibles:

1. $\forall X. X = X$ satisfacible e inválida
2. $\forall X. P(X) \Rightarrow \forall X. P(f(X))$ válida (\therefore satisfacible)
3. $\forall X. \neg P(X) \wedge \exists X. P(X)$ insatisfacible (\therefore inválida)
4. $\forall X. \exists Y. P(X, Y) \Rightarrow \exists Y. \forall X. P(X, Y)$ satisfacible e inválida
5. $\forall X. (P(X) \Rightarrow \sigma) \Rightarrow (\exists X. P(X)) \Rightarrow \sigma$ con $X \notin \text{fv}(\sigma)$ válida

El problema de la decisión

Querríamos un algoritmo que resuelva el siguiente problema:

- Entrada: una fórmula σ .
- Salida: un booleano que indica si σ es válida.

No es posible dar un algoritmo que cumpla dicha especificación.

Unificación de términos de lógica de primer orden

Anteriormente dije que los tipos no eran más que términos. El algoritmo de unificación para tipos es literalmente el mismo que para términos. Con un leve cambio de notación nos quedan las reglas:

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

Resolución lógica (Capítulo 9)

En el capítulo anterior... (*flashbacks* de LPO)...

Ahora vamos a ver **resolución lógica**. ¿Cuál es la motivación para aprenderla?, bueno, de **Prolog**, el lenguaje de **paradigma lógico** que vamos a utilizar.

Veamos para motivarnos cómo funciona **Prolog**. A este programa:

```

1  padre(cronos, zeus).
2  padre(zeus, atenea).
3  padre(zeus, hefesto).
4
5  padre(zeus, ares). abuelo(X, Y) :- padre(X, Z), padre(Z, Y).

```

Se le pueden realizar, por ejemplo, las siguientes **consultas**:

```

1  ?- padre(zeus, atenea).
2  >> true.

3

4  ?- padre(zeus, cronos).
5  >> false.

6

7  ?- abuelo(X, atenea).
8  >> X = cronos.

9

10 ?- abuelo(X, zeus).
11 >> false.

12

13 ?- abuelo(cronos, X).
14 >> X = atenea ;
15 >> X = hefesto ;
16 >> X = ares.

17

18 ?- abuelo(X, Y).
19 >> X = cronos, Y = atenea ;

```

```
20  >> X = cronos, Y = hefesto ;
21  >> X = cronos, Y = ares.
```

Prolog opera con **términos de primer orden**:

$X \ Y \ succ(succ(zero)) \ bin(I, R, D) \ \dots$

Las **fórmulas atómicas** son de la forma $pred(t_1, \dots, t_n)$

```
1  padre(zeus, atenea)
2  suma(zero, X, X)
```

Un programa es un conjunto de **reglas**. Cada regla es de la forma:

$\sigma \ :- \ \tau_1, \dots, \tau_n$

donde $\sigma, \tau_1, \dots, \tau_n$ son fórmulas atómicas.

```
1  abuelo(X, Y) :- padre(X, Z), padre(Z, Y).
```

Las reglas en las que $n = 0$ se llaman **hechos** y se escriben: σ .

```
1  padre(zeus, ares).
```

Las reglas tienen la siguiente interpretación lógica:

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

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

```
1  \(\forall X. \forall Y. \forall Z. ((padre(X, Z) \wedge padre(Z, Y)) \Rightarrow abuelo(X, Y))
```

Una **consulta** es de la forma:

?- $\sigma_1, \dots, \sigma_n$

```
1  ?- abuelo(X, ares).
```

Las consultas tienen la siguiente interpretación lógica:

$$\exists_1 \dots \exists_k. (\sigma_1 \wedge \dots \wedge \sigma_n)$$

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

El **entorno de Prolog** busca demostrar la fórmula τ de la consulta.

En realidad **busca refutar** $\neg\tau$.

La **búsqueda de la refutación se basa en el método de resolución**.

(¡Chan!, ahora sabés por qué vemos esto)

Véámoslo primero para *lógica proposicional* para entender los conceptos (más fácil), después para *lógica de primer orden* (más complicado).

Método de resolución (para lógica proposicional)

Pensemos **qué queremos** primero. Queremos un algoritmo tal que la

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

Idea

1. Escribir $\neg\sigma$ como un conjunto \mathcal{C} de **cláusulas**. (Pasar a *forma clausal*).
2. Buscar una **refutación** de \mathcal{C} . Una refutación de \mathcal{C} es una derivación de $\mathcal{C} \vdash \perp$.
 - Si se encuentra una refutación de \mathcal{C} :
 - Vale $\neg\sigma \vdash \perp$. Es decir, $\neg\sigma$ es insatisfacible/contradicción.
 - Luego vale $\vdash \sigma$. Es decir, σ es válida/tautología.
 - Si no se encuentra una refutación de \mathcal{C} :
 - No vale $\neg\sigma \vdash \perp$. Es decir, σ es satisfacible.
 - Luego no vale $\vdash \sigma$. Es decir, σ no es válida.

Veamos como hacer lo primero (obtener el conjunto de **cláusulas** \mathcal{C} , i.e. *pasar a forma clausal*)

Pasaje a forma clausal

Una fórmula se pasa a forma clausal aplicando las siguientes reglas. Todas las reglas transforman la fórmula en otra equivalente.

Paso 1. Deshacrese del conectivo " \Rightarrow ":

$$\sigma \Rightarrow \tau \rightarrow \neg\sigma \vee \tau$$

La fórmula resultante solo usa los conectivos $\{\neg, \vee, \wedge\}$.

Paso 2. Empujar el conectivo " \neg " hacia adentro:

$$\begin{aligned}\neg(\sigma \wedge \tau) &\rightarrow \neg\sigma \vee \neg\tau \\ \neg(\sigma \vee \tau) &\rightarrow \neg\sigma \wedge \neg\tau \\ \neg\neg\sigma &\rightarrow \sigma\end{aligned}$$

La fórmula resultante esta en **forma normal negada (NNF)**:

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

Paso 3. Distribuir \vee sobre \wedge :

$$\begin{aligned}\sigma \vee (\tau \wedge \rho) &\rightarrow (\sigma \vee \tau) \wedge (\sigma \vee \rho) \\ (\sigma \wedge \tau) \vee \rho &\rightarrow (\sigma \vee \rho) \wedge (\tau \vee \rho)\end{aligned}$$

La fórmula resultante esta en **forma normal conjuntiva (CNF)**. Una fórmula en **CNF** es conjunción de disyunciones de literales (asumiendo que permitimos asociar libremente \wedge y \vee):

$$\begin{array}{lll}\text{Fórmulas en CNF} & \sigma_{\text{cnf}} & ::= (\kappa_1 \wedge \kappa_2 \wedge \dots \wedge \kappa_n) \\ \text{Cláusulas} & \kappa & ::= (\ell_1 \vee \ell_2 \vee \dots \vee \ell_m) \\ \text{Literales} & \ell & ::= P \mid \neg P\end{array}$$

Paso 4. Final:

Por último, usando el hecho de que la disyunción (\vee) es:

$$\begin{array}{lll}\text{Asociativa} & \sigma \vee (\tau \vee \rho) & \iff (\sigma \vee \tau) \vee \rho \\ \text{Comutativa} & \sigma \vee \tau & \iff \tau \vee \sigma \\ \text{Idempotente} & \sigma \vee \sigma & \iff \sigma\end{array}$$

notamos una cláusula (disyunción de literales) como un conjunto:

$$(\ell_1 \vee \ell_2 \vee \dots \vee \ell_n) \quad \text{se nota} \quad \{\ell_1, \ell_2, \dots, \ell_n\}$$

Análogamente, usando el hecho de que la conjunción (\wedge) es asociativa, comutativa, e idempotente, notamos una conjunción de cláusulas como un conjunto:

$$(\kappa_1 \wedge \kappa_2 \wedge \dots \wedge \kappa_n) \quad \text{se nota} \quad \{\kappa_1, \kappa_2, \dots, \kappa_n\}$$

Completamos el primer paso. Veamos ahora como buscamos una **refutación**.

Una vez obtenido un conjunto de cláusulas $\mathcal{C} = \{\kappa_1, \dots, \kappa_n\}$, se busca una **refutación**, es decir, una demostración de $\mathcal{C} \vdash \perp$.

El método de refutación se basa en la siguiente regla de deducción:

$$\frac{\textcolor{red}{P} \vee \ell_1 \vee \dots \vee \ell_n \quad \neg \textcolor{red}{P} \vee \ell'_1 \vee \dots \vee \ell'_m}{\ell_1 \vee \dots \vee \ell_n \vee \ell'_1 \vee \dots \vee \ell'_m} \text{ Regla de resolución}$$

Escrita con notación de clásulas:

$$\frac{\{\textcolor{red}{P}, \ell_1, \dots, \ell_n\} \quad \{\neg \textcolor{red}{P}, \ell'_1, \dots, \ell'_m\}}{\{\ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_m\}} \text{ Regla de resolución}$$

La conclusión se llama la **resolvente** de las premisas.

Algoritmo de refutación

- Entrada: un conjunto de cláusulas $\mathcal{C}_0 = \{\kappa_1, \dots, \kappa_n\}$.
- Salida: **SAT/INSAT** indicando si \mathcal{C}_0 es insatisfacible ($\mathcal{C}_0 \vdash \perp$).

Sea $\mathcal{C} := \mathcal{C}_0$. Repetir mientras sea posible:

1. Si $\{\} \in \mathcal{C}$, devolver **INSAT**.
2. Elegir dos cláusulas $\kappa, \kappa' \in \mathcal{C}$, tales que:
 - $\kappa = \{P, \ell_1, \dots, \ell_n\}$
 - $\kappa' = \{\neg P, \ell'_1, \dots, \ell'_m\}$
 La resolvente $\rho = \{\ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_m\}$ no está en \mathcal{C} .
 Si no es posible, devolver **SAT**.
3. Tomar $\mathcal{C} := \mathcal{C} \cup \{\rho\}$ y volver al paso 1.

Corrección del pasaje a forma clausal (teorema)

Dada una fórmula σ :

1. El pasaje a forma clausal termina.
2. El conjunto de cláusulas \mathcal{C} obtenido es equivalente a σ .
 Es decir, $\vdash \sigma \iff \mathcal{C}$.

Corrección del algoritmo de refutación (teorema)

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

1. El algoritmo de refutación termina.
2. El algoritmo retorna **INSAT** si y solo si $\mathcal{C}_0 \vdash \perp$.

Método de resolución (para lógica de primer orden)

Bueno, ahora que sabemos sobre cómo funciona esto en lógica proposicional, podemos encarar lo mismo, pero en lógica de primer orden, veamos...

Acá la milanesa tiene más vueltas, principalmente en algunos casos, **nuestro algoritmo se puede colgar** 😊.

Seguimos queriendo casi lo mismo, un algoritmo tal que la

- Entrada sea: una formula σ de la lógica de primer orden.
- Salida sea: un booleano que indica si σ es válida.

Si σ es válida, el método siempre termina.

Si σ no es válida, el método puede no terminar.

Idea (procedimiento de semi-decisión)

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

Veamos cómo pasar una fórmula a forma clausal, pero en **LPO**. Los pasos son parecidos, pero con cambios importantes.

Paso 1. Deshacerse del conectivo " \Rightarrow ":

$$\sigma \Rightarrow \tau \rightarrow \neg\sigma \vee \tau$$

La fórmula resultante solo usa los conectivos $\{\neg, \vee, \wedge, \forall, \exists\}$.

Paso 2. Empujar el conectivo " \neg " hacia adentro:

$$\begin{aligned} \neg(\sigma \wedge \tau) &\rightarrow \neg\sigma \vee \neg\tau \\ \neg(\sigma \vee \tau) &\rightarrow \neg\sigma \wedge \neg\tau \\ \neg\neg\sigma &\rightarrow \sigma \\ \neg\forall X. \sigma &\rightarrow \exists X. \neg\sigma \\ \neg\exists X. \sigma &\rightarrow \forall X. \neg\sigma \end{aligned}$$

La fórmula resultante está en **forma normal negada (NNF)**.

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

Paso 3. Extraer los cuantificadores (\forall, \exists) hacia afuera.

Se asume siempre que $X \notin \text{fv}(\tau)$:

$$\begin{aligned} (\forall X. \sigma) \wedge \tau &\longrightarrow \forall X. (\sigma \wedge \tau) \quad \tau \wedge (\forall X. \sigma) \longrightarrow \forall X. (\tau \wedge \sigma) \\ (\forall X. \sigma) \vee \tau &\longrightarrow \forall X. (\sigma \vee \tau) \quad \tau \vee (\forall X. \sigma) \longrightarrow \forall X. (\tau \vee \sigma) \\ (\exists X. \sigma) \wedge \tau &\longrightarrow \exists X. (\sigma \wedge \tau) \quad \tau \wedge (\exists X. \sigma) \longrightarrow \exists X. (\tau \wedge \sigma) \\ (\exists X. \sigma) \vee \tau &\longrightarrow \exists X. (\sigma \vee \tau) \quad \tau \vee (\exists X. \sigma) \longrightarrow \exists X. (\tau \vee \sigma) \end{aligned}$$

Todas las reglas transforman la fórmula en otra equivalente.

La fórmula resultante está en **forma normal prenexa**:

$$\sigma_{\text{pre}} ::= Q_1 X_1. Q_2 X_2. \dots. Q_n X_n. \tau$$

donde cada Q_i es un cuantificador $\{\forall, \exists\}$ y τ representa una fórmula en **forma normal negada (NNF)** libre de cuantificadores.

(!) Paso 4. Deshacerse de los \exists , a este paso se lo conoce como **Skolemización** (*técnica de Herbrand y Skolem*):

$\forall X. \exists Y. P(X, Y)$	es sat.	<i>sii</i>	$\forall X. P(X, f(X))$	es sat.
$\forall X_1 X_2. \exists Y. P(X_1, X_2, Y)$	es sat.	<i>sii</i>	$\forall X_1 X_2. P(X_1, X_2, f(X_1, X_2))$	es sat.
.
$\forall \vec{X}. \exists Y. P(\vec{X}, Y)$	es sat.	<i>sii</i>	$\forall \vec{X}. P(\vec{X}, f(\vec{X}))$	es sat.

El lado izquierdo es una fórmula en el lenguaje \mathcal{L} .

El lado derecho es una fórmula en el lenguaje $\mathcal{L} \cup \{f\}$.

Caso particular cuando $|\vec{X}| = 0$

$$\exists Y. P(Y) \text{ es sat. } \text{i.e. } P(c) \text{ es sat.}$$

El lenguaje se extiende con una nueva constante c .

(!) La Skolemización **preserva la satisfactibilidad**.

Pero **no siempre produce fórmulas equivalentes**.

Es decir, **no preserva la validez**.

Ejemplo — la Skolemización no preserva la validez

$$\underbrace{\exists X. (P(0) \Rightarrow P(X))}_{\text{válida}} \quad \underbrace{P(0) \Rightarrow P(c)}_{\text{inválida}}$$

"Entonces, ¿cómo hacemos?"

Dada una fórmula en **forma normal prenexa**, se aplica la regla:

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

donde f es un símbolo de función nuevo de aridad $n \geq 0$.

La fórmula resultante está en **forma normal de Skolem**:

$$\sigma_{Sk} ::= \forall X_1 X_2 \dots X_n. \tau$$

donde τ representa una fórmula en **forma normal negada (NNF)** libre de cuantificadores.

(!) **Paso 5.** Dada una fórmula en **forma normal de Skolem**:

$$\forall X_1, X_2, \dots, X_n. \tau \quad (\tau \text{ libre de cuantificadores})$$

se pasa τ a forma normal conjuntiva usando las reglas ya vistas:

$$\begin{aligned} \sigma \vee (\tau \wedge \rho) &\rightarrow (\sigma \vee \tau) \wedge (\sigma \vee \rho) \\ (\sigma \wedge \tau) \vee \rho &\rightarrow (\sigma \vee \rho) \wedge (\tau \vee \rho) \end{aligned}$$

El resultado es una fórmula de la forma:

$$\forall X_1 \dots X_n. \left(\begin{array}{l} (\ell_1^{(1)} \vee \dots \vee \ell_{m_1}^{(1)}) \\ \wedge (\ell_1^{(2)} \vee \dots \vee \ell_{m_2}^{(2)}) \\ \dots \\ \wedge (\ell_1^{(k)} \vee \dots \vee \ell_{m_k}^{(k)}) \end{array} \right)$$

(!) **Paso 6.** Empujar los cuantificadores universales hacia adentro:

$$\forall X_1 \dots X_n. \left(\begin{array}{l} (\ell_1^{(1)} \vee \dots \vee \ell_{m_1}^{(1)}) \\ \wedge (\ell_1^{(2)} \vee \dots \vee \ell_{m_2}^{(2)}) \\ \dots \\ \wedge (\ell_1^{(k)} \vee \dots \vee \ell_{m_k}^{(k)}) \end{array} \right) \rightarrow \left(\begin{array}{l} \forall X_1 \dots X_n. (\ell_1^{(1)} \vee \dots \vee \ell_{m_1}^{(1)}) \\ \wedge \forall X_1 \dots X_n. (\ell_1^{(2)} \vee \dots \vee \ell_{m_2}^{(2)}) \\ \dots \\ \wedge \forall X_1 \dots X_n. (\ell_1^{(k)} \vee \dots \vee \ell_{m_k}^{(k)}) \end{array} \right)$$

Por último, la **forma clausal** es:

$$\left\{ \begin{array}{l} \{l_1^{(1)}, \dots, l_{m_1}^{(1)}\}, \\ \{l_1^{(2)}, \dots, l_{m_2}^{(2)}\}, \\ \vdots \\ \{l_1^{(k)}, \dots, l_{m_k}^{(k)}\} \end{array} \right\}$$

Resumen (de pasaje a forma clausal para LPO):

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

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

Ejemplo — pasaje a forma clausal

Queremos ver que $\sigma \equiv \exists X. (\forall Y. P(X, Y) \Rightarrow \forall Y. P(Y, X))$ es válida.

Primero la negamos: $\neg\sigma \equiv \neg\exists X. (\forall Y. P(X, Y) \Rightarrow \forall Y. P(Y, X))$.

Pasamos $\neg\sigma$ a forma clausal:

$$\begin{aligned} & \neg\exists X. (\forall Y. P(X, Y) \Rightarrow \forall Y. P(Y, X)) \\ \rightarrow & \neg\exists X. (\neg\forall Y. P(X, Y) \vee \forall Y. P(Y, X)) \\ \rightarrow & \forall X. \neg(\neg\forall Y. P(X, Y) \vee \forall Y. P(Y, X)) \\ \rightarrow & \forall X. (\neg\neg\forall Y. P(X, Y) \wedge \neg\forall Y. P(Y, X)) \\ \rightarrow & \forall X. (\forall Y. P(X, Y) \wedge \exists Y. \neg P(Y, X)) \\ \rightarrow & \forall X. \exists Y. (\forall Y. P(X, Y) \wedge \neg P(Y, X)) \\ \rightarrow & \forall X. \exists Y. \forall Z. (P(X, Z) \wedge \neg P(Y, X)) \\ \rightarrow & \forall X. \forall Z. (P(X, Z) \wedge \neg P(f(X), X)) \\ \rightarrow & \forall X. \forall Z. P(X, Z) \wedge \forall X. \forall Z. \neg P(f(X), X) \end{aligned}$$

La forma clausal es:

$$\{\{P(x, z)\}, \{\neg P(f(x), x)\}\} \equiv \{\{P(x, y)\}, \{\neg P(f(z), z)\}\}$$

Refutación en lógica de primer orden

Bueno, habíamos dicho que la estrategia era la misma. Teníamos nuestra fórmula σ , la negamos y nos quedamos con $\neg\sigma$, después, la pasamos a un conjunto \mathcal{C} de cláusulas (*forma clausal*).

Veamos ahora cómo funciona el método de refutación (*general*).

Uno en un primer momento podría pensar que podríamos la regla que vimos para lógica proposicional...

Una vez obtenido un conjunto de cláusulas $\mathcal{C} = \{\kappa_1, \dots, \kappa_n\}$, se busca una **refutación**, es decir, una demostración de $\mathcal{C} \vdash \perp$.

Recordemos la regla de resolución proposicional:

$$\frac{\{\mathbf{P}, \ell_1, \dots, \ell_n\} \quad \{\neg\mathbf{P}, \ell'_1, \dots, \ell'_m\}}{\{\ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_m\}}$$

Queremos adaptarla a lógica de primer orden.

En lugar de una variable proposicional \mathbf{P} vamos a tener una fórmula atómica $\mathbf{P}(t_1, \dots, t_n)$.

¿Podemos escribir la regla así?:

$$\frac{\{\mathbf{P}(t_1, \dots, t_n), \ell_1, \dots, \ell_n\} \quad \{\neg\mathbf{P}(t_1, \dots, t_n), \ell'_1, \dots, \ell'_m\}}{\{\ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_m\}}$$

Sin embargo, esa idea no funciona pues:

Consideremos la fórmula:

$$\forall x. \mathbf{P}(x) \wedge \neg\mathbf{P}(0)$$

Debería ser refutable, pues es insatisfacible.

Su forma clausal consta de dos cláusulas:

$$\{\mathbf{P}(x)\} \quad \{\neg\mathbf{P}(0)\}$$

La regla de resolución propuesta no aplica pues $\mathbf{P}(x) \neq \mathbf{P}(0)$.

Los términos no necesariamente tienen que ser iguales.

Relajamos la regla para permitir que sean **unificables**.

Ahora sí, veamos la regla **correcta**.

La regla de resolución de primer orden es:

$$\frac{\begin{array}{c} \{\sigma_1, \dots, \sigma_p, \ell_1, \dots, \ell_n\} \quad \{\neg\tau_1, \dots, \neg\tau_q, \ell'_1, \dots, \ell'_m\} \\ \mathbf{S} = \text{mgu}(\{\sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{?}{=} \dots \stackrel{?}{=} \sigma_p \stackrel{?}{=} \tau_1 \stackrel{?}{=} \tau_2 \stackrel{?}{=} \dots \stackrel{?}{=} \tau_q\}) \end{array}}{\mathbf{S}(\{\ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_m\})}$$

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

Se asume implícitamente que las cláusulas están renombradas de tal modo que $\{\sigma_1, \dots, \sigma_p, \ell_1, \dots, \ell_n\}$ y $\{\neg\tau_1, \dots, \neg\tau_q, \ell'_1, \dots, \ell'_m\}$ no tienen variables en común.

Y referente al algoritmo de refutación, este se adapta sin mayores cambios. Se usa la nueva regla de resolución para calcular la resolvente.

Ejemplo — método de resolución

Queremos demostrar $\sigma \equiv \exists x. (\forall y. \mathbf{P}(x, y) \Rightarrow \forall y. \mathbf{P}(y, x))$.

Equivalentemente, veamos que $\neg\sigma \vdash \perp$.

La forma clausal de $\neg\sigma$ era:

$$\mathcal{C} = \underbrace{\{\{\mathbf{P}(x, y)\}\}}_{1} \cup \underbrace{\{\neg\mathbf{P}(f(z), z)\}}_{2}$$

- De 1 y 2 calculamos
 $\text{mgu}(\mathbf{P}(x, y) \stackrel{?}{=} \mathbf{P}(f(z), z)) = \{x := f(z), y := z\}$
y se obtiene la resolvente $\{\}$.

Podríamos también considerar si se puede aplicar una regla de resolución que sea binaria. Sin embargo,

Resolución binaria

Considerar la siguiente variante de la regla de resolución:

$$\frac{\{\sigma, \ell_1, \dots, \ell_n\} \quad \{\neg\tau, \ell'_1, \dots, \ell'_m\} \quad S = \text{mgu}(\{\sigma \stackrel{?}{=} \tau\})}{S(\{\ell_1, \dots, \ell_n, \ell'_1, \dots, \ell'_m\})}$$

No es completa.

Ejemplo

$\{\{P(X), P(Y)\}, \{\neg P(Z), \neg P(W)\}\}$ es insatisfacible.

No es posible alcanzar la cláusula vacía $\{\}$ con resolución binaria.

Corrección del método de resolución de primer orden (teoremas)

Teorema (corrección del pasaje a forma clausal)

Dada una fórmula σ :

1. El pasaje a forma clausal termina.
2. El conjunto de cláusulas C obtenido es **equisatisfacible** a σ . Es decir, σ es sat. si y sólo si C es sat..

Teorema (corrección del algoritmo de refutación)

Dado un conjunto de cláusulas C_0 :

1. Si $C_0 \vdash \perp$, entonces el algoritmo de refutación termina.
2. El algoritmo retorna **INSAT** si y sólo si $C_0 \vdash \perp$.

Si $C_0 \not\vdash \perp$, no hay garantía de terminación.

Y antes dijimos que este método podría no terminar. Acá hay un ejemplo donde no termina.

Ejemplo — no terminación

La siguiente fórmula σ no es válida:

$$\forall X. (P(\text{succ}(X)) \Rightarrow P(X)) \Rightarrow P(0)$$

Tratemos de probar que es válida usando el método de resolución.

Para ello pasamos $\neg\sigma$ a forma clausal:

$$\underbrace{\{\{\neg P(\text{succ}(X)), P(X)\}, \{\neg P(0)\}\}}_{\begin{array}{c} 1 \\ 2 \end{array}}$$

- ▶ De $\boxed{1}$ y $\boxed{2}$ obtenemos $\boxed{3} = \{\neg P(\text{succ}(0))\}$.
- ▶ De $\boxed{1}$ y $\boxed{3}$ obtenemos $\boxed{4} = \{\neg P(\text{succ}(\text{succ}(0)))\}$.
- ▶ De $\boxed{1}$ y $\boxed{4}$ obtenemos $\boxed{5} = \{\neg P(\text{succ}(\text{succ}(\text{succ}(0))))\}$.

Resolución SLD y Prolog (Capítulo 10)

En el capítulo anterior vimos el método de resolución general, sin embargo, este tiene varios problemas. Uno de ellos es la complejidad, más allá del cálculo, puede uno darse cuenta que se va bastante para arriba.

Requiere de un criterio de búsqueda (*elegir dos cláusulas*)

Y de un criterio de selección (*elegir un subconjunto de literales de cada cláusula*)

La cantidad de opciones es exponencial en el tamaño del problema.

Además,

- cada paso agrega una nueva cláusula.
- en cada paso se deben resolver ecuaciones de unificación.
- el método requiere usar BFS (para evitar irse por una rama que no termine / para que sea completo).

Y peor aún, no es determinístico, ¿cómo elegimos esas cláusulas, y ese subconjunto de literales?...

Es por ello que planteamos una alternativa a la resolución general, llamada resolución SLD. Esta resolución tiene ventajas y desventajas, de eso se trata este capítulo, a su vez, es el

método de resolución que usa Prolog con ligeras modificaciones (de base SLD no es determinístico).

Resolución SLD (Capítulo 10: parte I)

La resolución SLD es un *tradeoff*: menor generalidad a cambio de mayor eficiencia.

Menor generalidad

No se puede aplicar sobre fórmulas de primer orden arbitrarias. Solo se puede aplicar sobre cláusulas de Horn.

Mayor eficiencia

Se reducen las opciones de búsqueda/selección.

Cláusulas de Horn

Recordemos que una **cláusula** es un **conjunto de literales**:

$$\{\ell_1, \dots, \ell_n\}$$

donde cada literal es una fórmula atómica posiblemente negada:

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

Definición (Cláusulas de Horn 🐐)

Las cláusulas son de los siguientes tipos, **dependiendo del número de literales positivos** que contengan:

	#positivos	#negativos
Cláusula objetivo	0	*
Cláusula de definición	1	*
Cláusula de Horn	≤ 1	*

Ejemplo

$$\underbrace{\{P(x)\} \quad \{P(x), \neg Q(Y, X), \neg R(Y)\}}_{\text{Cláusulas de definición}} \quad \underbrace{\{\} \quad \{\neg P(X), \neg Q(Y, X)\}}_{\text{Cláusulas objetivo}} \quad \underbrace{\quad}_{\text{Cláusulas de Horn}}$$

Regla de resolución SLD

La regla de **resolución SLD** involucra siempre a una cláusula de **definición** y una cláusula **objetivo**.

$$\frac{\begin{array}{c} \{P(t_1, \dots, t_k), \neg \sigma_1, \dots, \neg \sigma_n\} \quad \{\neg P(s_1, \dots, s_k), \neg \tau_1, \dots, \neg \tau_m\} \\ S = \text{mgu}(\{P(t_1, \dots, t_k) \stackrel{?}{=} P(s_1, \dots, s_k)\}) \end{array}}{S(\{\neg \sigma_1, \dots, \neg \sigma_n, \neg \tau_1, \dots, \neg \tau_m\})}$$

Es un caso particular de la regla de resolución general.

La **selección es binaria** (*elige un literal de cada cláusula*).

La resolvente es una nueva cláusula **objetivo**.

Dato: hay fórmulas que no se pueden escribir como **cláusulas de Horn**, por ejemplo $P \vee Q$.

Derivaciones SLD

Una derivación SLD comienza con $n \geq 0$ cláusulas de **definición** y una cláusula **objetivo**.

$$D_1 \quad \dots \quad D_n \quad G_1$$

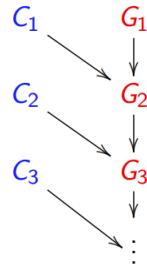
En cada paso:

- Se elige una cláusula **definición** D_j con $1 \leq j \leq n$.
- Se aplica la regla de resolución SLD sobre D_j y G_i .
- La resolvente es una nueva cláusula **objetivo** G_{i+1} .

Es decir, dado un conjunto de cláusulas:

$$D_1 \quad \dots \quad D_n \quad G_1$$

una derivación SLD es de la forma:



Cada C_i debe ser alguna de las cláusulas originales $\{D_1, \dots, D_n\}$. La cláusula G_{i+1} se obtiene aplicando resolución SLD sobre C_i y G_i .

Conclusión, se nos **simplifica la búsqueda y selección**.

Búsqueda:

- Se limita a elegir C_i como una de las n cláusulas D_1, \dots, D_n .
- La cláusula objetivo G_i está fijada, no hay alternativas.

Selección:

- Se limita a elegir uno de los literales **negativos** de G_i .
- La cláusula de definición C_i tiene un único literal **positivo**.

Definición (Sustitución respuesta)

Dada una refutación SLD, con pasos:

$$G_1 \xrightarrow{S_1} G_2 \xrightarrow{S_2} \dots \xrightarrow{S_{n-1}} G_n$$

$$C_1 \nearrow \quad C_2 \nearrow \quad \quad \quad C_{n-1} \nearrow$$

la **sustitución respuesta** es la composición $S_{n-1} \circ \dots \circ S_1$.

Compleitud del método de resolución SLD

El método de resolución **es completo para cláusulas de Horn**. Más precisamente, si D_1, \dots, D_n son cláusulas de **definición** y G una cláusula **objetivo**:

Si $\{D_1, \dots, D_n, G\}$ es insatisfacible, existe una refutación SLD. (**Teorema**)

Prolog (Capítulo 10: parte II)

Un programa en Prolog es un conjunto de cláusulas de definición.

Una consulta en Prolog es una cláusula objetivo.

La notación cambia ligeramente.

Cláusulas	Prolog
{a(0, X, X)} {a(s(X), Y, s(Z)), \neg a(X, Y, Z)}	a(0, X, X). a(s(X), Y, s(Z)) :- a(X, Y, Z).
{ \neg a(s(0), X, s(s(s(0))))}	?- a(s(0), X, s(s(s(0)))).

La ejecución se basa en la regla de resolución SLD.

El orden de los literales de cláusula objetivo es relevante, pues, para hacerlo determinístico, Prolog siempre elige el **primer literal** de la cláusula (*criterio de selección*).
"De izquierda a derecha".

Prolog busca sucesivamente todas las refutaciones haciendo DFS.

Toma las reglas en **orden de aparición** (*criterio de búsqueda*).

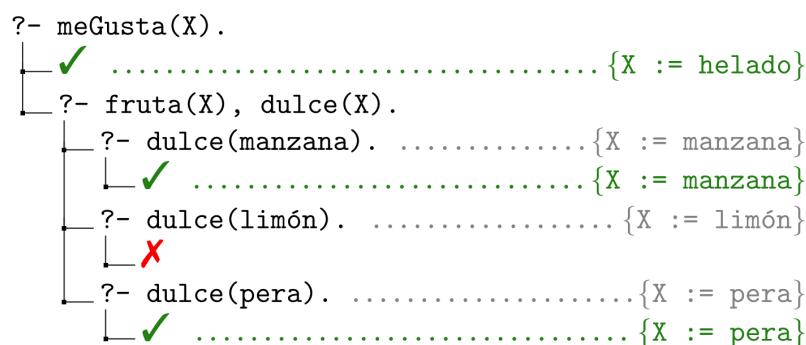
"De arriba para abajo".

Ejemplo de árbol de refutación en Prolog con

```

1  fruta(manzana).
2  dulce(manzana).
3  fruta(limón).
4  dulce(pera).
5  fruta(pera).
6  meGusta(helado).
7
8  meGusta(X) :- fruta(X), dulce(X).

```



La exploración DFS es **incompleta**.

Puede provocar que Prolog nunca encuentre refutaciones posibles.

```

1  esMaravilloso(X) :- esMaravilloso(suc(X)).
2  esMaravilloso(cero).

```

Es por ello que el orden de las reglas se torna relevante. La exploración BFS es completa pero **muy** costosa.

A su vez, Prolog al unificar, no usa la regla *occurs-check*.

$$\{X = t\} \cup E \xrightarrow{\text{Occurs-Check}} \begin{array}{l} \text{falla} \\ \text{si } X \neq t \text{ y } X \in fv(t) \end{array}$$

Esto puede provocar que Prolog encuentre una "refutación" incorrecta.

Por ejemplo, X unifica con $f(X)$, lo que es **incorrecto**.

```

1  esElSucesor(X, suc(X)).
2
3  % Encuentra una refutación {Y := X, X := (X)}
4  esElSucesor(Y, Y).

```

En muchos contextos, la regla *occurs-check* es innecesaria.

Lo que sí, perdemos corrección.

Aspectos extralógicos de Prolog

Más allá del motor lógico que fue de lo que estuvimos hablando arriba, Prolog cuenta con un motor extralógico. Tenés el operador de corte **!**, **not**, etc...

Dejo unas imágenes de la teórica para no dejar esto vacío.

Operador de corte (!) "cut"

<p>Consideremos el siguiente programa:</p> <pre>padre(zeus, atenea). % ...base de conocimiento... ancestro(X, Y) :- padre(X, Y). ancestro(X, Y) :- padre(X, Z), ancestro(Z, Y). ?- ancestro(zeus, atenea). - ?- padre(zeus, atenea). \ - ✓ \ - ?- padre(zeus, Z), ancestro(Z, atenea). \ - ... </pre> <p>Nos gustaría tener una manera de podar el árbol de búsqueda.</p>	<p>Reescribimos el programa agregando un <i>corte</i> ("!":)</p> <pre>padre(zeus, atenea). % ...base de conocimiento... ancestro(X, Y) :- padre(X, Y), !. ancestro(X, Y) :- padre(X, Z), ancestro(Z, Y).</pre> <p>El operador ! no tiene una interpretación declarativa/lógica. Es un operador extra-lógico. Su comportamiento se explica desde el punto de vista operacional. El operador de corte indica que, si se lo alcanza, no se deben explorar alternativas a la regla en la que aparece.</p>
<p>Semántica del operador de corte El predicado ! tiene éxito inmediatamente.</p> <p>Al momento de hacer <i>backtracking</i>:</p> <ul style="list-style-type: none"> ▶ Se vuelve atrás hasta el punto en el que se eligió usar la regla que hizo aparecer el operador de corte. ▶ Se descartan todas las elecciones alternativas. ▶ Se continúa buscando hacia atrás. 	<p>Operador de corte (cut)</p> <p>Ejemplo — cortes "benignos" (green cuts) En algunos casos, ! no altera la semántica del programa. Puede servir para construir programas equivalentes más eficientes.</p> <pre>add(N, zero, N) :- !. add(zero, N, N). add(suc(N), M, suc(P)) :- add(N, M, P). ?- add(suc(suc(suc(suc(...)))), zero, P).</pre> <p>Ejemplo — cortes "riesgosos" (red cuts) En otros casos, la semántica puede verse alterada.</p> <pre>maximo(A, B, A) :- A >= B, !. maximo(A, B, B).</pre> <pre>?- maximo(2, 1, B). ?- maximo(2, 1, 1). >> B = 2 >> true.</pre>

Negación por falla (not)

<p>Definición (operador de negación) Si fail es un predicado que falla siempre, se puede definir la negación en Prolog así:</p> <pre>not(P) :- P, !, fail. not(P).</pre> <p>Observación. not(P) tiene éxito si y sólo si P falla.</p> <p>Ejemplo — negación por falla</p> <pre>fruta(pera).</pre> <pre>?- not(fruta(papa)). ?- not(fruta(pera)). \ - fruta(papa), !, fail. \ - fruta(pera), !, fail. \ - ✓ \ - X</pre> <pre>?- not(fruta(pera)). ?- not(fruta(pera)). \ - fruta(pera), !, fail. \ - fruta(pera), !, fail. \ - !, fail. \ - X</pre>	<p>La negación por falla no coincide con la negación lógica.</p> <p>Ejemplo — negación por falla ≠ negación lógica</p> <pre>fruta(pera). verdura(papa).</pre> <pre>?- verdura(X), not(fruta(X)). \ - not(fruta(papa)). {X := papa} \ - ✓</pre> <pre>?- not(fruta(X)), verdura(X). \ - fruta(X), !, fail, verdura(X). \ - !, fail, verdura(papa). {X := pera} \ - fail, verdura(papa). {X := pera} \ - X</pre> <p>El orden de los literales en la consulta se vuelve relevante. Esto atenta contra la declaratividad.</p>
---	---

Programación orientada a objetos (Capítulo 11)

Bueno, finalmente el último capítulo. Sobre este no hay teórica, así que voy a escribir todo lo que me acuerde. Seguramente lo complemento con partes prácticas.

En particular, usamos **Smalltalk** para trabajar con el paradigma de objetos. A su vez, usamos **Pharo** como entorno de ejecución.

¿Qué es un programa en la programación orientada a objetos?

- Es un software como modelo computable de la realidad para comprender y resolver problemas.
- Cómputo basado en **objetos** enviándose **mensajes**.
- Evitar soluciones procedurales **delegando responsabilidades** en objetos.

El conjunto de **mensajes** a los que un objeto sabe responder, define su comportamiento (qué hacen) y sus responsabilidades (qué deben saber hacer).

Es decir, en el paradigma de objetos, **todo es un objeto**.

Los **objetos se comunican** entre si enviándose **mensajes**.

Si un objeto no sabe responder al mensaje, le pregunta al de más arriba (estructura de clases).

Veamos más a fondo eso de los mensajes.

```
1  2 pesos.  
2 December first, 1985.  
3 'hola mundo' indexOf: $o startingAt: 6.
```

El primer mensaje es **unario**, se le está enviando el mensaje `pesos` al objeto `2`.

El segundo mensaje es **binario**. ¿Cómo funciona te preguntarás?

Bueno, primero tenemos que entender cómo se maneja el orden de la resolución de mensajes. Primero se resuelven los mensajes **unarios**, luego los **binarios**, y finalmente los **keywords**, aún no vimos qué son los keywords, pero tenía que mencionarlo.

Cuestión, en `December first, 1985.` tenemos dos mensajes, el primero, que es unario, es `December first`, el segundo es `,`, este mensaje se le envía al objeto `December first` con `1985` como colaborador/argumento (que también es un objeto!!).

El tercer mensaje es para desglosar. Acá apareció el concepto de **keywords**.

Vayamos primero a identificar el **objeto receptor** del mensaje

el objeto `'hola mundo'` del mensaje `indexOf: $o startingAt: 6`, este mensaje cuenta con dos **keywords**: `indexOf:` y `startingAt:`, y los colaboradores/argumentos son `$o` y `6`.

También, los mensajes se leen de izquierda a derecha, y de arriba para abajo; priorizando paréntesis. La precedencia de paréntesis es distinta a la que solemos usar, así que tratar las cosas con cuidado.

Por ejemplo, `20 + 3 * 5 = (20 + 3) * 5 != 20 + (3 * 5)` en **Smalltalk**. Para nosotros debería ser `20 + 3 * 5 = 20 + (3 * 5)` en lugar de lo de arriba.

¿Qué saben hacer los objetos?

Colaborar entre sí mediante el envío de mensajes. A esta secuencia de colaboraciones la llamamos **método**, y definen el **CÓMO** (antes definimos el **qué**).

- Los métodos son objetos
- La ejecución de un método también es un objeto.
- Ambos se pueden inspeccionar dentro del entorno de programación.

Comunicación entre objetos en Smalltalk

- Es **dirigida**: hay un emisor y un receptor.
- Es **sincrónica**: se espera a la respuesta del mensaje.
- **Siempre** hay respuesta: si no se explicita, se retorna `self`.
- **El receptor no conoce al emisor**: siempre responde igual sin importar el emisor (salvo que el emisor se envíe como colaborador del mensaje)

¿Cómo es la **sintaxis** de Smalltalk?

```
1 "Comentarios".
2 | var1 var2 ...| . "inicializar variables"
3 [:arg1 :arg2 | | var1 var2 | expresión]. "closures, como 'lambdas'"
4 expresión1. expresión2. expresión3. "'ejecutar' expresiones"
5 objeto mensaje. "mandar mensaje a objeto"
6 objeto msj1; msj2. "mandar msj1 y msj2 al mismo objeto (objeto)"
7 var := expresión. "declarar una variable"
8 ^ expresión. "devolver (return) una expresión"
```

Literales

```
1 123.
2 123.4.
3 $c.
4 'texto'.
5 #símbolo.
6 #(123 123.3 $a ábc#abc).
```

Palabras reservadas

```
1 self.
2 super.
3 nil.
4 true.
5 false.
6 thisContext.
```

Closures

Permiten **representar un conjunto de colaboraciones**. En definitiva, es **segmento de código** al cual no me importa ponerle un nombre. Su sintaxis la vimos más arriba, veamos un ejemplo.

```
1 | result |
2 result := [ :x :y |
3   | v |
4   v := x.
5   v * x + y
6 ] value: 3 value: 5.
```

Toma dos argumentos, `x` e `y` e inicializa una variable local `v`, luego la declara con el valor de `x` y ejecuta la siguiente expresión `v * x + y`; al no haber `^`, retorna la expresión de la última línea, en este caso el objeto que resulta de la expresión `v * x + y`.

Los closures se ligan al contexto de ejecución donde son creados, tanto las variables como el `return`.

Colecciones

Hay un montón, algunas conocidas son:

- Bag (multiconjunto)
- Set (conjunto)
- Array (arreglo)
- OrderedCollection (Lista)
- SortedCollection (Lista ordenada)
- Dictionary (hash)

Los mensajes `#with: with: ...` son una forma de crear colecciones. Por ejemplo

```

1 "Se le puede agregar elementos."
2 Bag with: 1 with: 2 with: 4.
3 Bag withAll: #(1 2 4).
4
5 "No se le puede agregar elementos."
6 (Array with: 1 with: 2 with: 4).
7 #(1 2 4).

```

Dependiendo de la colección va a reponderte una cosa u otra a los mensajes que le mandes. Por ejemplo `add:` no es lo mismo para lo de arriba que para lo de abajo. Lo de abajo es un arreglo, al igual que `#{1 2 4}`.

Mensajes comunes a colecciones

- `add:` agrega un elemento.
- `at:` devuelve el elemento en una posición.
- `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.
- `keysAndValuesDo:` evalúa un bloque con cada par clave-valor.
- `keysDo:` evalúa un bloque con cada clave.
- `select:` devuelve los elementos de una colección que cumplen un predicado (filter de funcional).
- `reject:` la negación del select:
- `collect:` devuelve una colección que es resultado de aplicarle un bloque a cada elemento de la colección original (map de funcional).
- `detect:` devuelve el primer elemento que cumple un predicado.
- `detect:ifNone:` como `detect:`, pero permite ejecutar un bloque si no se encuentra ningún elemento.
- `reduce:` toma un bloque de dos o más parámetros de entrada y hace fold de los elementos de izquierda a derecha (foldl de funcional).

Bueno, vayamos a resolver ejercicios.

Ejercicio 1 ★

En las siguientes expresiones, identificar mensajes. Indicar el objeto receptor y los colaboradores en cada caso.

- a) `10 numberOfDigitsInBase: 2.`
- b) `10 factorial.`
- c) `20 + 3 * 5.`
- d) `20 + (3 * 5).`
- e) `December first, 1985.`
- f) `1 = 2 ifTrue: ['what!?'].`
- g) `101 insideTriangle: 000 with: 200 with: 002.`
- h) `'Hello World', indexOf: $o startingAt: 6.`
- i) `(OrderedCollection with: 1) add: 25; add: 35; yourself.`
- j) `Object subclass: #SnakesAndLadders instanceVariableNames: 'players squares turn die over' classVariableNames: '' poolDictionaries: '' category: 'SnakesAndLadders'.`

Creo que se entiende con los colores.

Ejercicio 4 ★

Para cada una de las siguientes expresiones, indicar qué valor devuelve o explicar por qué se produce un error al ejecutarlas. Para este ejercicio recomendamos pensar qué resultado debería obtenerse y luego corroborarlo en el Workspace de Pharo.

- a) [:x | x + 1] value: 2
- b) [|x| x := 10. x + 12] value
- c) [:x :y | |z| z := x + y] value: 1 value: 2
- d) [:x :y | x + 1] value: 1
- e) [:x | [:y | x + 1]] value: 2
- f) [[:x | x + 1]] value
- g) [:x :y :z | x + y + z] valueWithArguments: #(1 2 3)
- h) [|z| z := 10. [:x | x + z]] value value: 10

¿Cuál es la diferencia entre [|x y z| x + 1] y [:x :y :z| x + 1]?

```
1  [:x | x + 1] value: 2 "3"
2  [|x| x := 10. x + 12] value "22"
3  [:x :y | |z| z := x + y] value: 1 value: 2 "3"
4  [:x :y | x + 1] value: 1 "error - falta un colaborador"
5  [:x | [:y | x + 1]] value: 2 "un closure que toma un argumento y devuelve
   constantemente 3"
6  [[:x | x + 1]] value "un closure [:x | x + 1]"
7  [:x :y :z | x + y + z] valueWithArguments: #(1 2 3) "6"
8  [|z| z := 10. [:x | x + z]] value value: 10 "20 pues el primer value te da
   [:x | x + 10] (en realidad no reemplaza z por 10 sino que queda en el
   contexto)"
```

Ejercicio 6 ★

Mostrar un ejemplo por cada uno de los siguientes mensajes que pueden enviarse a las colecciones en el lenguaje Smalltalk. Indicar a qué evalúan dichos ejemplos.

- a) #collect:
- b) #select:
- c) #inject: into:
- d) #reduce: (o #fold:)
- e) #reduceRight:
- f) #do:

Resolución de este ejercicio sacada de [honi](#).

```
1  #(1 2 3 4) collect: [:x | x * 2]. "devuelve #(2 4 6 8). - Es un `map` de
   funcional."
2
3  #(1 2 3 4) select: [:x | (x mod: 2) = 0 ]. "devuelve #(2 4). - Es un
   `filter` de funcional."
4
5  #(1 2 3 4) inject: 0 into: [:x :y | x + y ]. "devuelve 10. - Es un `foldl` de
   funcional. El argumento `inject:` es el valor inicial, `into:` es el
   bloque que se ejecuta por cada par de elementos."
6
7  #(1 2 3 4) reduce: [:x :y | x + y ]. "devuelve 10. - Es un `foldl1` de
   funcional. Falla si la colección está vacía."
8
9  #(1 2 3 4) reduceRight: [:x :y | x + y ]. "devuelve 10. - Es un `foldr1` de
   funcional. Falla si la colección está vacía."
10
11 | L1 L2 |
12 L1 := OrderedCollection withAll: #(1 2 3 4).
13 L2 := OrderedCollection new.
14 L1 do: [:x | L2 add: x + 1].
15 L2.
16 "devuelve un OrderedCollection(2 3 4 5). - Agrega todos los elementos de
   `L1` en `L2` sumándoles `1`."
```

Ejercicio 7 ★

Suponiendo que tenemos un objeto `obj` que tiene el siguiente método definido en su clase

```
SomeClass << foo: x
| aBlock z|
z := 10.
aBlock := [x > 5 ifTrue: [z := z + x. ^0] ifFalse: [z := z - x. 5]].
y := aBlock value.
y := y + z.
^y.
```

¿Cuál es el resultado de evaluar las siguientes expresiones?

- a) `obj foo: 4.`
- b) `Message selector: #foo: argument: 5.`
- c) `obj foo: 10.` (Ayuda: el resultado no es 20).

Créditos nuevamente a [honi](#).

a) Dentro del método vale `x = 4`. Cuando evaluamos el valor de `aBlock`, `x > 5` es falso entonces se evalúa el valor del argumento `ifFalse:`, puntualmente el bloque `[z := z - x. 5]`. Como `z = 10`, este bloque hace `z := 10 - 4` y luego retorna el número `5`, que se asigna a la variable `y`. La variable `z` ahora vale `6`. Finalmente se ejecuta `y := 5 + 6`, y luego se retorna el valor de `y` que es `11`.

b) Crea un objeto de la clase `Message` que codifica el mensaje para el selector `foo` con argumento `5`.

Podemos enviar este mensaje de la siguiente forma:

```
1 (Message selector: #foo: argument: 5) sendTo: obj.
```

c) Dentro del método `value` `x = 10`. Cuando evaluamos el valor de `aBlock`, `x > 5` es verdadero entonces se evalúa el valor del argumento `ifTrue:`, puntualmente el bloque `[z := z + x. ^0]`. Este bloque es interesante porque más allá de la cuentita que hace con `z`, después retorna: `^0`. El símbolo `^` es un return explícito del método, **no** del bloque. Entonces el resultado que obtenemos es `0` ya que no se ejecuta el código que viene después del bloque.

Ejercicio 8 ★

Implementar métodos para los siguientes mensajes:

a) `#curry`, cuyo objeto receptor es un bloque de dos parámetros, y su resultado es un bloque similar al original pero “currificado”.

Por ejemplo, la siguiente ejecución evalúa a 12.

```
| curried new|
curried := [ :x :res | x + res ] curry.
new := curried value: 10.
new value: 2.
```

b) `#flip`, que al enviarse a un bloque de dos parámetros, devuelve un bloque similar al original, pero con los parámetros en el orden inverso.

c) `#timesRepeat:`, cuyo objeto receptor es un número natural y recibe como colaborador un bloque, el cual se evaluará tantas veces como el número lo indique.

Por ejemplo, luego de la siguiente ejecución, `count` vale 20 y `copy` 18.

```
| count copy |
count := 0.
10 timesRepeat: [copy := count. count := count + 2].
```

Método curry:

```
1 curry
2 "Currifica."
3 ^ [:x | [:y | self value: x value: y]]
```

Método flip:

```
1 flip
2 "Flipea."
3 ^ [:x :y | self value: y value: x]
```

Método timesRepeat:

```

1  timesRepeat: aBlock
2      "Ejecuta un bloque self veces."
3      1 to: self do: [ aBlock ]

```

Ejercicio 9 ★

Agregar a la clase `BlockClosure` el método de clase `generarBloqueInfinito` que devuelve un bloque `b1` tal que:

`b1 value` devuelve un arreglo de 2 elementos #(1 b2)
`b2 value` devuelve un arreglo de 2 elementos #(2 b3)
`:`
`bi value` devuelve un arreglo de 2 elementos #(i bi+1)

```

1  generarBloqueInfinito: aBlock
2      | bloque | "inicializa la variable bloque"
3      bloque := [:i | { i . [:j | bloque value: j + 1]} ]. "define bloque
tal que, toma un i y te devuelve un arreglo {i (bloque i+1)} por así
decirlo"
4      ^ bloque value: 1. "caso base, i vale 1"

```

Ejercicio 11 ★

Suponiendo que `anObject` es una instancia de la clase `OneClass` que tiene definido el método de instancia `aMessage`. Al ejecutar la siguiente expresión: `anObject aMessage`

- ¿A qué objeto queda ligada (hace referencia) la pseudo-variable `self` en el contexto de ejecución del método que es invocado?
- ¿A qué objeto queda ligada la pseudo-variable `super` en el contexto de ejecución del método que es invocado?
- ¿Es cierto que `super == self`? ¿es cierto en cualquier contexto de ejecución?

I. Queda ligada al objeto **anObject**, es decir, a la instancia `anObject` de la clase `OneClass`.

II. Al objeto **anObject**, la diferencia es que en lugar de ir a buscar el método de instancia que implementa el mensaje **aMessage** en `OneClass`, va a ir buscar en los métodos de instancia de la clase que está arriba de `OneClass`.

III. Decir `super == self` es medio un montón. La idea es que es "análogo" usar uno u otro si. el método de instancia que implementa el mensaje (en este caso `aMessage`) no está implementado en la clase que le corresponde a esa instancia (en este caso `OneClass`).

Ergo, lo va a ir a buscar siempre al menos una clase más arriba, por lo que skiparte buscando de entrada con `super` o mandarte a buscarlo con `self` para terminar subiendo una clase y hacer lo mismo que `super`, da lo mismo.

Resuelvo el ejercicio de objetos del segundo parcial, del primer cuatrimestre de 2024.

Ejercicio 3 - Objetos y Deducción Natural

- a) 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.
]

```

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)
```

- ii. Implementar un método para el mensaje `#divisores`, cuyo objeto receptor es un número entero, que devuelve una colección con sus divisores.
Sugerencia: utilizar el mensaje binario `'\N'` que devuelve el resto de la división entera entre dos números.

a)

Objeto	Mensaje	Colaboradores	Ubicación	Respuesta	¿Qué estoy resolviendo? ✘ *
A	new		Behavior	anA	A new
B	new		Behavior	aB	B new
C	new		Behavior	aC	C new

Objeto	Mensaje	Colaboradores	Ubicación	Respuesta	¿Qué estoy resolviendo? × _*
anA	a: , b:	aB , aC	A	3 por × ₂	anA a: ab b: aC
aC	c		C	[self a: super c b: self]	La parte de (y c) en x a: (y c) b: self
aB	a: , b:	[self a: super c b: self] , anA	B	3 por × ₂	x a: (y c) b: self donde (y c) es [self a: super c b: self] y self es anA
anA	c		A	2	y c en y c + x value donde y es anA y x es [self a: super c b: self]
[self a: super c b: self]	value		C	1 por × ₁	[self a: super c b: self] value donde self y super están ligadas al objeto aC y super va a buscar el mensaje que le mandes a la clase B
aC	c		B	1	super c en [self a: super c b: self]
aC	a: , b:	1 , aC	C	1 × ₁	[self a: super c b: self] que "se puede leer" como [aC a: 1 b: aC] ponele
2	+	1	SmallInteger	3 × ₂	y c + x value

×_{*} no es necesario, pero lo pongo por claridad.

b)

```

1  Integer << divisores
2
3  | res |
4  res := OrderedCollection new.
5  1 to: self do: [ :d |
6      self \\\ d = 0 ifTrue: [res add: d]
7  ].
8
9  ^res

```