

Paradigmas de Lenguajes de Programación

Tomás Agustín Hernández



Programación Funcional

Consiste en definir funciones y aplicarlas para procesar información.
Las funciones son verdaderamente funciones (parciales):

- Aplicar una función no tiene efectos secundarios.
- A una misma entrada le corresponde siempre la misma salida.
- Las estructuras de datos son inmutables.

Las funciones, además son datos como cualquier otro:

- Se pueden pasar como parámetros.
- Se pueden devolver como resultados.
- Pueden formar parte de estructuras de datos. Ej.: Un árbol binario que en sus nodos hay funciones.

Expresiones

Son secuencias de símbolos que sirven para representar datos, funciones, y funciones aplicadas a los datos.
Una expresión puede ser:

- Un constructor: `True`, `False`, `[]`, `(:)`, `0`, `1`, `2`.
 - Type Constructor: Es un constructor que se utiliza para crear un nuevo tipo.
 - Data Constructor: Se utiliza para crear valores de ese tipo.
 - Ej.: `data Color = Rojo | Verde | Azul`. `Azul` es un **data constructor** pues nos permite crear valores del tipo `Color` mientras que el Type Constructor es `Color`.
 - Ej.: `data Complejo = C Float Float`. `C` es una función que recibe dos `Float` y es un constructor de `Complejo`.
- Una variable: `longitud`, `ordenar`, `x`, `xs` `(+)`, `(*)`.
- La aplicación de una expresión a otra: `ordenar lista`, `not True`, `(+) 1`.

Función Parcialmente Aplicada

Una función parcialmente aplicada es una función a las cuales se llama otra función pero no se le proporcionan todos los argumentos.

Ahora, es importante que para que esta función parcialmente aplicada tenga sentido se le manden todos los parámetros.

Es una especie de $a \rightarrow b$ Ej.:

```
1 | add :: Int -> Int -> Int
2 | add x y = x + y
3 |
4 | add5 :: Int -> Int
5 | add5 = add 5
6 |
7 | add5 es una función que ejecuta la función parcial add pues le manda solo un parámetro y necesita dos
```

Una buena pregunta entonces es ¿pero por qué es una función parcial `add 5`? Pues hay una función `add5` que la implementa sin pasarle todos los parámetros directamente explícitamente.

```
1 | add5 3 -> Este 3 llega como "y" a add
2 |
3 | add5 = add 5
4 | add = 5 + y
5 | add = 5 + 3
6 | add = 8
```

Otro ejemplo

```
1 | const :: a -> b -> a
2 | const x y = x
3 |
4 | const (const 1) 2 -> const 1
5 | El llamado de (const 1) es una función parcialmente aplicada porque no envía el valor de y, sin embargo, const (const 1) 2 no la aplica parcialmente porque le termina mandando los dos parámetros.
```

Aplicación de Expresiones

Es asociativa hacia la izquierda:

- $f\ x\ y \equiv (f\ x)\ y$
- $((((f\ a)\ b)\ c)\ d)$: Primero calcula el resultado que devuelve la expresión f enviando el valor de a . Nótese que la idea sería que $(f\ a)$ devuelva una expresión del tipo función pues luego le pasamos otro parámetro (b) .

Importancia de la Aplicación de las Expresiones

¿Qué sucede si aplicamos Head Tail? Recordemos que la asociatividad a la izquierda haría algo así $\text{Head}(\text{Tail})$ pero Tail en ese momento no es nada, y si aplicamos head explota.

En este caso los paréntesis son importantes: $(\text{head}\ (\text{tail}\ l))$

Veamos otro ejemplo $\text{map}(\backslash x \rightarrow x\ 0)\ (\text{map}(+)\ [1,2,3])$. En este caso la asociatividad a la izquierda nos muestra algo así $\text{map}(\backslash x \rightarrow x\ 0)\ ((\text{map}(+)\ [1,2,3]))$

Esto genera $\text{map}(\backslash x \rightarrow x\ 0)\ [2,3,4]$ ahora según lo que haga la función x , hace una cosa u otra. Imaginemos que lo llamamos como $\text{map}\ (+1)\ [2,3,4]$ esto daría $[3,4,5]$

Función \$

Aplica una función a un valor

```
1 | g :: (a -> b) -> (a -> b)
2 | g x y = x y
```

Construyendo una lista paso a paso con constructores

Ej.: ¿Cómo construimos la lista $[1, 2]$ utilizando constructores?

- Lo primero que necesitamos, es un constructor de listas. Para eso tenemos la expresión $(:)$. Recordando que la aplicación de expresiones es asociativo a izquierda.
- Veamos su tipo desde GHCi $(:) :: a \rightarrow [a] \rightarrow [a]$.
- Necesitamos enviarle un valor de tipo a , una lista y como resultado, la operación $(:)$ devuelve una lista.
- Preguntemos lo siguiente: ¿Con $((:) 1)$ nos basta para agregar a una lista? No, porque no estamos cumpliendo el tipado del constructor. Si quisiéramos una lista con solamente el 1 sí bastaría, pero acá también queremos el 2.
- Entonces, comenzamos aplicando la expresión $(:)$ con el número 2, y ahí sí enviamos como segundo parámetro una lista vacía (que da como resultado) una lista vacía.
- $(((:) 2)\ []) = [2]$
- Por último, $(((:) 1)\ [2])$ también cumple el tipo pues nos quedaría $(((:) 1)\ [2]) = [1, 2]$

¿Y si quisiéramos construir 1, 2, 3, 4? Recordemos la asociatividad a la izquierda, pero a la hora de querer utilizar una expresión, hay que cumplir el tipado. Hasta que ese tipado no se cumpla, Haskell tratará de resolver la expresión más profunda para ver si reduciendo cumple el tipo.

$(((:)1)\ ((((:) 4)\ ((((:) 2)\ ((((:) 3)\ []))))))$

Esto lo podemos ver como $a \rightarrow (b \rightarrow (c \rightarrow (d)))$ pues para conocer a , necesitamos construir la lista de la derecha, para conocer b necesitamos la lista de la derecha, para conocer c necesitamos la lista de d , y d inicializa la lista vacía. Esto es importantísimo porque claramente no podríamos usar $(:) 1$ sin una lista. Entonces Haskell evalúa todo lo de la derecha hasta que obtenga una lista. Si no se obtuviera una lista, sería inválido.

Polimorfismo

Sucede en aquellas expresiones que tienen más de un tipo.

- $id :: a \rightarrow a$
- $(:) :: a \rightarrow [a] \rightarrow [a]$

- $fst :: (a, b) \rightarrow a$
- $cargarStringArray :: String \rightarrow \square \rightarrow [String]$. No es una expresión Polimórfica.

Nota: Es importante que si tenemos 2 parámetros de tipo a significa que los dos parámetros deben ser de ese tipo. Si tenemos 2 parámetros uno de tipo a y uno de b podría suceder que sean diferentes pero puede que sean el mismo. **Importante:** Recordar que si usamos operadores, colocar las clases que correspondan. Ej.: $a > 0$ a debe ser $(Num\ a, Ord\ a)$. Véase anexo para ver ejemplos de las clases de tipos.

Polimorfismo con Clases (Type Classes)

Es posible limitar el polimorfismo a clases específicas. Es decir, si nos mandan un tipo a podemos decir que ese tipo a es genérico pero de una clase específica.

Ejemplo: $func :: Num\ a \implies a \rightarrow a \rightarrow a$

En este caso, $Num\ a$ es una Type Class.

Modelo de Cómputo (cálculo de valores)

Dada una expresión, se computa su valor usando las ecuaciones siempre y cuando estén bien tipadas.

Importante: Que una expresión se cumpla el tipado, no significa que devuelva un valor.

- $sumarUno :: a \rightarrow a$: No falla nunca (función total)
- $division :: a \rightarrow b$: Falla si $b = 0$. Esto nos demuestra que aunque los parámetros estén bien tipados, podemos tener indefiniciones (función parcial).

¿Cómo está dado un programa en Funcional?

Un programa funcional está dado por un conjunto de **ecuaciones orientadas**. Se les llama de esta forma pues del **lado izquierdo está lo que define y del lado derecho la definición** (o expresión que produce el valor) Una ecuación $e1 = e2$ se interpreta desde dos puntos de vista

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

¿Cómo es el lado izquierdo de una ecuación orientada?

No es una expresión arbitraria. Debe ser una función aplicada a **patrones**.

Un patrón puede ser:

- Una variable: a, b, c .
- Un comodín: $_$
- Un constructor aplicado a patrones: Recordemos que un constructor sería algo que construye un tipo.
 - $True, False, 1$, etc.

Importante: El lado izquierdo NO debe contener variables repetidas.

Ej.: $iguales\ x\ x = True$ es una expresión mal formada. Esto pues dos valores que pueden ser diferentes, no pueden caer en la misma variable.

Ej.: $predecesor\ (n + 1) = n$ también está mal formada porque estamos haciendo un cálculo del lado izquierdo, y recordemos que del lado izquierdo solo hay definiciones. Del lado derecho se hacen los cálculos o cálculo de los valores.

¿Cómo evaluamos las expresiones?

- Buscamos la subexpresión más externa que coincida con el lado izquierdo de una ecuación.
- Reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
- Continuar evaluando la expresión resultante.

¿Cuándo se detiene la evaluación de una expresión?

- Cuando el programa estalla.
 - Loop infinito.
 - Indefinición.
- Cuando la expresión es una función **parcialmente aplicada**. ¿qué sería esto? ¿se refiere a utilizar mal la función parcialmente aplicada?
- La expresión es un constructor o un constructor aplicado. Ej.: True, (:), 1, [1, 2, 3].
 - Yo lo veo como algo más del tipo: una fórmula atómica o algo irreducible.

¿Cómo ayuda el Lazy Evaluation (Evaluación Perezosa) a Haskell?

Muchas veces nos ayuda a evitar tocar con un valor indefinido aunque esté ahí. Como no evalúa cosas que no necesita, si hay un indefinido por ahí y no necesita ni siquiera llegar, no lo toca.

```
1 |     indefinido :: Int
2 |     indefinido = indefinido
3 |     head(tail [indefinido, 1, indefinido])
```

¿Qué hace tail? Toma la cola de la lista, entonces como resultado arroja [1, indefinido] (ni siquiera evaluó el primer indefinido)

¿Qué hace head ahora entonces? [1] (ni siquiera evaluó el indefinido del final)

Importancia del orden de las Ecuaciones

El criterio más fuerte debe ir por encima del resto. Porque puede haber un caso que matchee y nunca se evalúe el siguiente caso.

Veamos un ejemplo:

```
1 |     esCorta (_:_:_) = False
2 |     esCorta _ = True
```

Considerando este orden:

- Si mando una lista que siempre tiene ≥ 3 elementos da False.
- Si mando una lista que tiene < 3 elementos es siempre True.

Cambiamos el orden y veamos qué cambia

```
1 |     esCorta _ = True
2 |     esCorta (_:_:_) = False
```

Considerando este orden:

- Si la lista tiene 0, 1, 2, 3, 4, ... n elementos siempre dará True.

En la segunda aplicación ¡nunca caemos en el caso 2! y esto es importante porque sabemos que si tiene 1 cae siempre en la primera.

Notación Infija & Notación Prefija

Infija: argumento + función + argumento

Prefija: función + argumentos

Importante: $(\leq)18 \neq (\leq 18)$ pues la opción de $(\leq)18$ sería $(18 \leq x)$ mientras que (≤ 18) sería $(x \leq 18)$. Donde claramente denota que el parámetro cuando está fuera de la notación infija es el primer argumento mientras que si lo ponemos adentro hardcodeamos su posición. Es decir $(18 \leq)$ estaríamos diciendo $(18 \leq x)$

Curricación

Una función es curricada cuando tiene la siguiente forma $ab :: Int \rightarrow Int \rightarrow Int$

Esto puede parecer que recibe 2 parámetros y devuelve uno, pero en realidad lo que hace es básicamente por cada parámetro asociar a la izquierda y hacer una función por cada parámetro.

Es decir, sería algo así $ab :: Int \rightarrow (Int \rightarrow Int)$

La función Curry **NO CAMBIA** la manera en que la función original curricada recibe los argumentos, sino que, la función curry es una especie de *punte* para que nosotros mandemos los argumentos separados y los aplique a la función no curricada.

Importante: No tiene sentido hablar de curricación cuando las funciones tienen un solo parámetro

Las funciones currificadas son realmente importantes en la Programación Funcional porque nos permite aplicarlas de forma parcial. Es una especie mas reutilizable.

Una función no currificada tiene esta pinta $\text{suma} :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}$ porque acá estoy obligando a la función suma a recibir una tupla de elementos, no puedo ir mandando de a uno parcialmente. Véase **anexo** para armar una función curry y uncurry

Funciones de Orden Superior

Son funciones que reciben como parámetro otras funciones.

Definamos la composición de funciones $(g \cdot f)$

Recordemos que en Álgebra vimos Composición de Funciones y es algo así: sean $A : B \rightarrow C$ y $D : A \rightarrow B$

La composición $g \circ f$ es $A \rightarrow B$. Es decir, va desde el dominio de D hasta la imagen de A. (la salida de una función debe ser la entrada de la otra.)

En Haskell, la podemos definir así

```
1  ent: Entrada sal: Salida
2
3  (.) :: (b -> c) -> (a -> b) -> (a -> c)
4
5  Queremos: (g . f) x => g (f x)
6
7  Desglosemos
8  (.) :: (ent. de g -> sal. de g) -> (ent. de f -> sal. de f) -> (ent. de f -> sal. de g)
9
10 El resultado de la composición nos da una nueva función que tiene como entrada un valor de tipo a, y
    la salida es un valor de tipo c.
11
12 Entonces, primero se evalúa f x, lo cual produce un resultado de tipo b.
13
14 Luego, este resultado (de tipo b) se pasa a g, produciendo un resultado de tipo c.
15
16 (g . f) x: envía el parámetro x a la función f, y el resultado de f x se manda a g. Esto da como
    resultado g (f x)
```

Otra forma de definir la composición es usando Notación Lambda.

```
1  (.) :: (b -> c) -> (a -> b) -> (a -> c)
2  g . f = \ x -> g (f x)
```

Recordemos que esto es algo recordable pues $f \circ g$ sería meter g adentro de f y para eso, la salida de g debe ser la entrada de f. Y luego se devuelve como salida el dominio de g y la salida de f. Nótese que la notación lambda es súper útil para definir funciones sin nombre, que solamente hagan algo. Esto es muy útil cuando queremos mandar una función por parámetro y que una función dada la llame a esta función.

Llamados estándar vs llamados en composición

$$\text{not}(\text{null } x) \equiv (\text{not} \cdot \text{null}) x$$

Esto quiere decir que si primero componemos todas las funciones que queremos aplicarle a un valor, es lo mismo que ir aplicando de una a una las funciones al argumento.

Importante: La composición funciona **sí y solo sí** alguna de las funciones está parcialmente aplicada. Ej.: $(== 0).mod\ n\ m$ no funciona, pero $(== 0).mod\ n$ sí.

Funciones Lambda

Son funciones anónimas, es decir, no tienen nombre.

$(\lambda x \rightarrow x + 1)$: Es una función anónima que dado un x, te devuelve una función que le aplica x+1.

Funciones Lambda Anidadas

$\lambda y \rightarrow (\lambda x \rightarrow y)$: Esto a ojo es una función constante, porque dado un valor y, se aplica la función x pero se devuelve el mismo valor y.

Los parámetros se obvían en algunos casos, este es uno. La mejor forma sería $\lambda y \rightarrow \lambda x \rightarrow y$.

Mejor notación para funciones anidadas: $\lambda y\ x \rightarrow y$

Reducción de Lambda

Preguntar $\text{foldr}(\backslash x \text{ rec} \rightarrow f x : \text{rec}) \equiv \backslash x \rightarrow (:) (f x) \text{ y } \backslash x \rightarrow e x \equiv e$

Asignar nombre a una función

$a = \backslash x \rightarrow x$: al nombre **a** le asigno la función anónima $\backslash x \rightarrow x$

¿Para qué queremos funciones de orden superior? Pt 1

```
1 | dobleL :: [Float] -> [Float]
2 | dobleL [] = []
3 | dobleL (x:xs) = x * 2 : dobleL xs
4 |
5 | esParL :: [Int] -> [Bool]
6 | esParL [] = []
7 | esParL (x:xs) = x 'mod' 2 == 0 : esParL xs
```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar.

```
1 | g [] = []
2 | g (x : xs) = f x : g xs
```

Lo único que cambia es **qué se hace** en cada paso recursivo.

Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? en este caso sí pero ¿qué operación se está haciendo? se están haciendo ciertas manipulaciones de los datos y se devuelve la información modificada en una lista nueva. Esto, en varios lenguajes se conoce como **map**. Se hace una manipulación de los datos pero se devuelve **la misma cantidad de elementos**.

Map

Recibe como parámetro una función que es aplicada a todos los elementos y devuelve una lista nueva. Se utiliza para modificar los valores de una lista dada según lo que haga la función.

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

Entonces, podemos definir **qué operación de modificación** se le realizan a los elementos de una lista dada.

```
1 | multiplicarPorDos :: [a] -> [b]
2 | multiplicarPorDos xs = map (\x -> x * 2) xs
3 |
4 | dividirPorDos :: [a] -> [b]
5 | dividirPorDos xs :: map (\x -> x / 2) xs
```

Entonces gracias a Map nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **qué hacen** con los elementos.

Véase [anexo](#) para ver ejercicios interesantes con Map.

¿Para qué queremos funciones de orden superior? Pt 2

```
1 | negativos :: [Int] -> [Int]
2 | negativos [] = []
3 | negativos (x:xs) = if x < 0
4 |                     then x : negativos xs
5 |                     else negativos xs
6 |
7 | pares :: [Int] -> [Int]
8 | pares [] = []
9 | pares (x:xs) = if x 'mod' 2 == 0
10 |                then x : pares xs
11 |                else pares xs
```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar, pero lo único que cambia es **cuando vamos a agregar a la lista los elementos**.

```
1 | g [] = []
2 | g (x:xs) = f x : g xs
```

Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? Puede que sí, puede que no. ¿Qué operación se está haciendo? se están filtrando ciertos elementos de una lista que no cumplan una condición dada. Esto, en varios lenguajes se conoce como **filter**. Se devuelve una nueva lista con los elementos que cumplan una condición dada.

Filter

Recibe como parámetro una función que es aplicada a todos los elementos. Se utiliza para "borrar" elementos de una lista, o quitar aquellos que no cumplan un criterio dado.

```
1 | filter :: (a -> Bool) -> [a] -> [b]
2 | filter p [] = []
3 | filter p (x : xs) = if p x
4 |                       then x : filter p xs
5 |                       else filter p xs
```

La función que le enviamos es para especificar **qué elementos nos queremos quedar**.

Entonces, podemos definir **qué operación de filtro** se le realizan a los elementos de una lista dada.

```
1 | eliminarImpares :: [a] -> [b]
2 | eliminarImpares xs = filter (\x -> x `mod` 2 == 0) xs
3 |
4 | borrarNegativos :: [a] -> [b]
5 | borrarNegativos xs = filter (\x -> x > 0) xs
```

Entonces gracias a Filter nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **con qué elementos nos quedamos** dada una condición

Recursión

¿Qué tienen en común los siguientes problemas? ¿En qué difieren?

```
1 | concat :: [[a]] -> [a]
2 | concat [] = []
3 | concat (x:xs) = x ++ concat xs
4 |
5 | reverso :: [a] -> [a]
6 | reverso [] = []
7 | reverso (x:xs) = reverso xs ++ [x]
8 |
9 | sum :: [int] -> int
10 | sum [] = 0
11 | sum (x:xs) = x + sum xs
12 |
13 | En común:
14 |     1. Todos tienen un caso base, pero es diferente.
15 |     2. Todos hacen un paso recursivo, pero cambia la operación que hacen.
16 |     3. El tipo de entrada puede no coincidir con el de la salida.
```

En este tipo de problemas lo mejor que podemos hacer es realizar una especie de función que nos permita modularizar lo más posible y aquello que difiere, pasarlo por parámetros.

Este tipo de problemas lo podemos solucionar con Foldr.

Recursión Estructural (foldr)

Se la conoce como Plegado a la Derecha porque va anidando el llamado de la función hacia adentro hasta que llega al caso base. Cuando llega al caso base, resuelve de derecha a izquierda. Una función f está dada por recursión estructural sí:

- El caso base devuelve un valor fijo z .
- El caso recursivo es una función de x y $(g\ xs)$



- Se trabaja solamente con la cabeza de la lista.
- Se hace recursión sobre la cola pero no se tiene acceso a ella, sino al llamado recursivo. Es decir, el caso recursivo no usa xs.
- La recursión es la clásica, va desde derecha a izquierda. La R de foldr es de Right.

Su estructura formal es la siguiente

```

1 | g [] = <caso base> -> valor
2 | g (x:xs) = <caso recursivo> -> función cabeza de la lista y resultado de la recursión.

```

Por lo tanto foldr se define como

```

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)

```

Importantísimo: El tipado de foldr es $Foldable\ t \implies (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow ta \rightarrow b$

- a = Es el elemento que tenemos actualmente en la recursión.
- b = Es cualquier tipo que nos permita acumular. Puede ser una tupla, un número, lo que sea. Pero en cada paso recursivo hay que llenar eso. El caso base tiene que cumplir este tipo b.
- Véase [anexo](#) para ver ejemplos usando foldr.

¿Qué es lo que produce foldr (:) []? Es una identidad sobre listas porque haría recursión sobre una lista vacía.

¿Qué es lo que hace la siguiente función foldr? $foldr(\backslash x\ r \rightarrow x)\ 0\ [1, 2]$

- Toma la lista y hace recursión. Empieza con el 2, ejecuta la función y devuelve 2. Hace el paso recursivo.
- Toma la lista y hace recursión. Ahora sigue con el 1, recibe 1 y la salida es 1.
- Esta función agarra el primer elemento de una lista, sería el head.

IMPORTANTÍSIMO: Se pueden recorrer dos listas, tres listas, las que quieras a la vez con Foldr. La recursion se hace sobre una sola, pero las n-1 listas las enviarías por argumento en cada paso recursivo. Véase [anexo](#) para ver ver el árbol de recursión y un ejemplo práctico. **Importante:** Foldr puede trabajar con listas infinitas.

Véase [anexo](#) para ver ejemplos de Foldr.

Recursión que no es estructural

```

1 | ssort :: Ord a => [a] -> [a]
2 | ssort [] = []
3 | ssort (x:xs) = minimo (x:xs) : ssort(sacarMinimo(x:xs))

```

No es estructural pues se esta usando xs para el llamado de minimo(x:xs) y no solamente en el llamado recursivo de ssort.

Iteración (foldl)

Se la conoce como Plegado a la Izquierda porque va resolviendo inmediatamente de izquierda a derecha hasta que termine el proceso.

Este tipo de recursión es más que recursión una iteración, en este caso empezamos yendo desde el primer valor hasta el último. En este enfoque voy modificando una solución parcial.

Por lo tanto foldl se define como

$$\text{foldl1 } f \text{ ac } \left(\begin{array}{c} (:) \\ / \quad \backslash \\ 1 \quad (:) \\ / \quad \backslash \\ 2 \quad (:) \\ / \quad \backslash \\ 3 \quad [] \end{array} \right) \rightsquigarrow^* \left(\begin{array}{c} f \\ / \quad \backslash \\ f \quad 3 \\ / \quad \backslash \\ f \quad 2 \\ / \quad \backslash \\ \text{ac} \quad 1 \end{array} \right)$$

```

bin2dec :: [Int] -> Int
bin2dec = foldl1 (\ ac b -> b + 2 * ac) 0

bin2dec [1, 0, 0]
  ~> foldl1 (\ ac b -> b + 2 * ac) 0           [1, 0, 0]
  ~> foldl1 (\ ac b -> b + 2 * ac) (1 + 0)      [0, 0]
  ~> foldl1 (\ ac b -> b + 2 * ac) (0 + 2 * (1 + 0)) [0]
  ~> foldl1 (\ ac b -> b + 2 * ac) (0 + 2 * (0 + 2 * (1 + 0))) []
  ~> 0 + 2 * (0 + 2 * (1 + 0))
  ~> 4

```

```

1 | foldl :: (b -> a -> b) -> b -> [a] -> b
2 | foldl f ac [] = ac
3 | foldl f ac (x:xs) = foldl f (f ac x) xs

```

¿Qué es lo que sucede en el siguiente ejemplo?

```

1 | foldl (\x y -> 1) 0 unos
2 | foldl f (f 0 1) unos
3 | foldl f(f(f 0 1) 1) unos

```

Esto es in-realizable con foldl. Es decir, foldl no puede manejar listas infinitas.

Importantísimo: El tipado de foldl es $Foldable\ t \implies (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow ta \rightarrow b$

- b = Es cualquier tipo que nos permita acumular. Puede ser una tupla, un número, lo que sea. Pero en cada paso recursivo hay que llenar eso. El caso base tiene que cumplir este tipo b .
- a = Es el elemento que tenemos actualmente en la recursión.

Véase [anexo](#) para ver ejemplos usando foldl.

¿Por qué foldl es peor que foldr?

foldl: Procesa la lista de izquierda a derecha, lo que requiere evaluar toda la lista antes de devolver un resultado, lo que no es posible con listas infinitas.

foldr: Procesa la lista de derecha a izquierda, permite trabajar de manera perezosa y puede manejar listas infinitas si la función y el valor inicial permiten una evaluación parcial o completa sin necesidad de procesar todos los elementos.

Recursión Primitiva (recr)

No existe en Haskell. Es una manera de nosotros podemos tener las mismas ventajas de foldr pero en este caso, la recursión primitiva nos permite utilizar la cola de la lista

- Se trabaja solamente con la cabeza de la lista y la cola.
- Se hace recursión sobre la cola.

Su estructura forma les la siguiente

```

1 | g [] = b
2 | g (x:xs) = f x xs (g xs)

```

Por lo tanto recr se define como

```

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)

```

Tipos

Existen diferentes maneras de definir tipos. Esto es según sea el objetivo. Definimos tipos con la palabra **data** + Nombre = Tipo1 — Tipo2 — Tipo 3

Si hacemos en GHCi : `tTipo1` saldrá que es de tipo Nombre.

Nota: | indica que a continuación hay otro tipo.

Tipos Comunes

Son no recursivos, ej: $data Dia = Lu|Ma|Mi|Mie|Ju|Vi|Sa|Do$

Lu, Ma, Mi, ... son constructores del tipo dia.

Los argumentos podrán ser recibidos diciendo qué tipo se espera, y usamos pattern matching para utilizarlos.

```
1 |   EsLunes :: Dia -> Bool
2 |   EsLunes Lu = True
3 |   EsLunes _ = False
4 |
5 |   EsFinDeSemana :: Dia -> Bool
6 |   EsFinDeSemana Sa = True
7 |   EsFinDeSemana Do = True
8 |   EsFinDeSemana _ = False
```

Tipos con Funciones

Los tipos también pueden tener tipos que necesiten argumentos. Difieren en la info que devuelven.

Ej.: $data Persona = LaPersona String String Int$

Importante

- Si evaluamos : $tLaPersona$ sin enviar los argumentos retornará $LaPersona :: String \rightarrow String \rightarrow Int$.
- Si evaluamos : $tLaPersona "T", "H", 23$ enviando los argumentos retornará que LaPersona es de tipo Persona

```
1 |   Edad :: Persona -> Int
2 |   Edad (LaPersona n a e) = e
3 |
4 |   Cumpleaños :: Persona -> Persona
5 |   Cumpleaños (LaPersona n a e) = LaPersona n a (e+1)
```

Importante: Nótese que estamos devolviendo **una nueva persona**. En programación funcional no existe el concepto de "modificar algo", sino crear algo nuevo con lo anterior y cambiarle algo.

Tipos Recursivos

Cuando tengo tipos recursivos, las funciones que los usen deben manejar casos bases y la recursión

Ej.: $data Nat = Zero \text{---} Succ Nat$

Tipos Polimórficos

Al igual que las funciones, podemos definir que los tipos tengan constructores de un tipo específico.

```
1 |   data List a = Vacía | Const a (List a)
```

Importante en Tipos

No se puede repetir un mismo constructor para un mismo tipo.

Es decir

```
1 |   data List = Vacía | Cons Int ListI
2 |   data List a = Vacía | Cost a (List a)
3 |
4 |   Error. No puede estar Vacía como constructor de dos tipos diferentes.
```

Listas

- Por extensión: Es dar la lista implícita escribiendo todos sus elementos. Ej. [1, 2, 3]
- Secuencias: Progresiones aritméticas en un rango particular. Ej.: [3..7] es la lista que tiene los números del 3 al 7.
- Por compresión: Se definen de la siguiente manera [expresión — selectores, condiciones]. Ej.: [(x,y) — x i- [0..5], y i- [0..3], x+y==4] es la lista de pares que tienen elementos de x e y que dan menos que 4

Estructuras Recursivas sobre Otras Estructuras de Datos

¿Cómo podemos plantear la estructura de un foldr para un árbol binario o cualquier estructura plegable que no sea una lista? Lo primero que podemos intuir es que en las listas, solo hay una recursión, sobre la lista propiamente pero en un Árbol Binario tenemos dos caminos: rama izquierda y rama derecha. Esto quiere decir que de alguna manera, tenemos que capturar 2 recursiones.

- Planteamos varios problemas acerca de esa estructura de datos y vemos qué patrones hay en común.
- Las cosas que sean diferentes, las pasamos por parámetros.

Por ejemplo, sean estas operaciones de un árbol binario

```
1 |   nodos :: AB -> Int
2 |   nodos Nil = 0
3 |   nodos (Bin i r d) = nodos i + 1 + nodos d
4 |
5 |   preorder :: AB -> [a]
6 |   preorder Nil = []
7 |   preorder (Bin i r d) = [r] ++ preorder i ++ preorder d
```

Podemos observar que cambia lo siguiente

- Tipos de salida: En el primer ejemplo devolvemos un int, en el segundo una lista de a. La lista de a sería el tipo que tenga el AB.
- Caso base: En uno es [] y en otro 0. Por lo tanto tenemos que admitir un tipo b para el caso base. El caso base es del mismo tipo que el tipo de salida **siempre**.
- Función que realiza: En uno hace sumas, en el otro hace concatenaciones.

¿Cómo podríamos escribir las funciones anteriormente mencionadas de manera anónima?

```
1 |   nodos: (\ri r rd -> ri + 1 + rd)(nodos i) r (nodos d)
2 |   preorder: (\ri r rd -> [r] ++ ri ++ rd) (preorder i) r (preorder d)
```

Recordemos la firma de foldr: $Foldable\ t \implies (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow t\ a \rightarrow b$

Donde b era el caso base / recursivo y a el elemento actual, acá tenemos dos.

Recordemos la estructura del tipo AB: $AB\ a = Nil \mid Bin\ (AB\ a)\ a\ (AB\ a)$

Entonces nuestra función foldAB va a tener que estar preparado para recibir ambos (AB a) y a. $foldAB :: (b \rightarrow a \rightarrow b \rightarrow b) \rightarrow b \rightarrow AB\ a \rightarrow b$ donde $t\ Bin: (AB\ a) \rightarrow a \rightarrow (AB)\ a \rightarrow AB\ a$ donde

- $(b \rightarrow a \rightarrow b \rightarrow b)$
 - la primera b: recursión rama izquierda
 - la a: la raíz
 - la segunda b: recursión rama derecha
 - la tercera b: el resultado del proceso.
- b: el resultado.
- AB a: la entrada
- b: el resultado.

Es súper importante entender que la firma de la función que acepta foldAB está prácticamente atada al tipo de la estructura de dato.

Validación y Verificación de Programas

- Trabajaremos con estructuras de datos **finitas**. Más técnicamente, con tipos de datos **inductivos**.
- Trabajamos con **funciones totales**
 - Las ecuaciones deben cubrir todos los casos posibles.
 - La recursión siempre termina.
- El programa **no depende del orden** de las ecuaciones.

Principio de Reemplazo

Sea $e1 = e2$ una ecuación del programa. Las siguientes operaciones preservan la igualdad de expresiones.

- Reemplazar **cualquier instancia** de $e1$ por $e2$.
- Reemplazar **cualquier instancia** de $e2$ por $e1$.

Importante: Si una igualdad se puede demostrar usando sólo el principio de reemplazo, decimos que la igualdad vale **por definición**.

PRINCIPIO DE REEMPLAZO

Sea $e1 = e2$.

Las siguientes operaciones preservan la igualdad de expresiones

- Reemplazar $e1$ por $e2$
- Reemplazar $e2$ por $e1$

{l0} LENGTH [] = 0

{l1} LENGTH (x:xs) = 1 + LENGTH xs

{s0} SUMA [] = 0

{s1} SUMA (x:xs) = x + SUMA xs

PROBAN: LENGTH ["A", "B"] = SUMA [1, 1]

LENGTH ["A", "B"]

$L1 = 1 + \text{LENGTH ["B"]}$

$L1 = 1 + (1 + \text{LENGTH []})$

$L1 = 1 + (1 + 0)$

$S0 = 1 + (1 + \text{SUMA []})$

$S1 = 1 + \text{SUMA [1]}$

$S1 = \text{SUMA [1, 1]}$

Inducción Estructural

Cada tipo de datos tiene su propio principio de inducción.

Importante: El \cdot acá cumple el rol del $()$ en el \forall es decir $\forall x :: \text{Bool}. P \equiv (\forall x :: \text{Bool})(P)$ y el $::$ cumple el rol de **es del tipo...**
Los constructores de tipo NO recursivos serán los casos base mientras que aquellos constructores recursivos serán las hipótesis inductivas.

Formas de demostrar

Existen 3 formas conocidas de demostrar cosas, por lo menos en la materia lo hacemos de maneras determinadas. Cuando usar una o la otra depende de la experiencia, y no es tan intuitivo saber cuando usar una opción u otra.

- Forma Álgebra: Arrancás desarrollando el lado izquierdo de tu TI. Aplicás la HI y tratás de llegar al lado derecho del TI.
- Forma Intermedia: Llegar de ambos lados de la TI a aplicar la HI. Solo funciona si estamos probando **igualdades**.
- Desarrollando lado derecho en base a contexto: Esta suele ser la opción más coherente en muchos casos. Consiste en desarrollar el TI del lado izquierdo, aplicar la HI y cuando llegamos a algo trivial o que no podemos seguir reduciendo, empezamos a desarrollar el lado derecho de la TI pero asumiendo las hipótesis del lado izquierdo.
 - Digamos que del lado izquierdo para aplicar la HI tuvimos que entrar a varios casos, por ejemplo: $xs = (z:zs)$, $e == z$. Si llegamos a una respuesta trivial del lado izquierdo entonces desarrollamos el lado derecho asumiendo que xs tiene la pinta $(z:zs)$ y efectivamente $e == z$. Esto nos hará todo mucho más corto.

Estas estrategias, las dejaré en el anexo para que claramente se note su diferencia y ventajas en cada caso.

Inducción sobre booleanos

Si $\mathcal{P}(\text{True})$ y $\mathcal{P}(\text{False})$ entonces $\forall x :: \text{Bool} \mathcal{P}(x)$

INDUCCIÓN CON BOOLEANOS:

{NT} $\text{NOT } \text{TRUE} = \text{FALSE}$

{NF} $\text{NOT } \text{FALSE} = \text{TRUE}$

PROBAR: $(\forall x :: \text{Bool}) (\text{NOT}(\text{NOT } x) = x)$

Se toman 2 CASOS: $x = \text{TRUE}$ y $x = \text{FALSE}$

CASO TRUE:

$\text{NOT}(\text{NOT } \text{TRUE}) = \text{TRUE}$

{NT} $\Rightarrow \text{NOT } \text{FALSE} = \text{TRUE}$

{NF} $\Rightarrow \text{TRUE} = \text{TRUE}$

$\Rightarrow \checkmark$

CASO FALSE:

$\text{NOT}(\text{NOT } \text{FALSE}) = \text{FALSE}$

{NF} $\Rightarrow \text{NOT } \text{TRUE} = \text{FALSE}$

{NT} $\Rightarrow \text{FALSE} = \text{FALSE}$

$\Rightarrow \checkmark$

Inducción sobre pares

Si $(\forall x :: a)(\forall y :: b)(\mathcal{P}(x, y))$ entonces $\forall p :: (a, b)\mathcal{P}(p)$

{FST} $\text{fst } (x, _) = x$

{SND} $\text{snd } (_, y) = y$

{SWAP} $\text{swap } (x, y) = (y, x)$

Para probar $\forall p :: (a, b). \text{fst } p = \text{snd } (\text{swap } p)$

basta probar:

$\forall x :: a. \forall y :: b. \text{fst } (x, y) = \text{snd } (\text{swap } (x, y))$

$\text{fst } (x, y) = x \underset{\text{FST}}{\overset{\uparrow}{=}} \underset{\text{SND}}{\overset{\uparrow}{\text{snd } (y, x)}} \underset{\text{SWAP}}{\overset{\uparrow}{=}} \text{snd } (\text{swap } (x, y))$

INDUCCIÓN SOBRE PARES:

{FST} $\text{FST}(x, _) = x$

{SND} $\text{SND}(_, y) = y$

{SWAP} $\text{SWAP}(x, y) = (y, x)$

PROBAR: $(\forall p :: (a, b))(\text{FST } p = \text{SND } (\text{SWAP } p))$

Necesitamos probar $(\forall a :: a)(\forall b :: b)(\text{FST } (x, y) = \text{SND } (\text{SWAP } (a, b)))$

FST (x, y)

{FST} x

{SND} $\text{SND}(y, x)$

{SWAP} $\text{SND}(\text{SWAP}(x, y))$

¿Cómo sabemos o tenemos la p? ¿No la tenemos?

Inducción sobre naturales

Si $\mathcal{P}(\text{Zero})$ y $(\forall n :: \text{Nat})(\mathcal{P}(n) \implies \mathcal{P}(\text{Suc } n))$ entonces $(\forall n :: \text{Nat})(\mathcal{P}(n))$ donde

- $\mathcal{P}(n)$: Hipótesis Inductiva.
- $\mathcal{P}(\text{Suc } n)$: Tesis Inductiva.

Ejemplo

$\{S0\}$ suma Zero $m = m$
 $\{S1\}$ suma (Suc n) $m = \text{Suc (suma } n) m$

Para probar $\forall n :: \text{Nat. suma } n \text{ Zero} = n$
 basta probar:

1. suma Zero Zero = Zero.
 Inmediato por S0.
2. $\underbrace{\text{suma } n \text{ Zero} = n}_{\text{H.I.}} \Rightarrow \underbrace{\text{suma (Suc } n) \text{ Zero} = \text{Suc } n}_{\text{T.I.}}$

$$\text{suma (Suc } n) \text{ Zero} = \text{Suc (suma } n \text{ Zero)} = \text{Suc } n$$

\uparrow S1 \uparrow H.I.

INDUCCIÓN SOBRE NATURALES:

DATA NAT = ZERO | SUC NAT

2 CONSTRUCCIONES = 2 CASOS

Si $\underbrace{P(\text{ZERO})}_{\text{CB}}$ y $\underbrace{(\forall m :: \text{Nat}) (P(m) \Rightarrow P(\text{SUC } m))}_{\text{H.I.}} \mid \text{ent } \underbrace{P(m)}_{\text{Q.V.Q.}} (\forall m :: \text{Nat})$

$\{S0\}$ SUMA ZERO $m = m$

$\{S1\}$ SUMA (SUC m) $m = \text{SUC (SUMA } m) m$

PROBAR: $(\forall m :: \text{Nat}) (\text{SUMA } m \text{ ZERO} = m)$

CB: SUMA ZERO ZERO = ZERO

PI: $\underbrace{\text{SUMA } m \text{ ZERO} = m}_{\text{H.I.}} \Rightarrow \underbrace{\text{SUMA (SUC } m) \text{ ZERO} = \text{SUC } m}_{\text{Q.V.Q.}}$

$\text{SUMA (SUC } m) \text{ ZERO} = \text{SUC (SUMA } m \text{ ZERO)} = \text{SUC } m$

\uparrow S1 \uparrow H.I.

Inducción Estructural: Caso General

Sea \mathcal{P} una propiedad acerca de las expresiones tipo T tal que

- \mathcal{P} vale sobre todos los constructores base de T ,
- \mathcal{P} vale sobre todos los constructores recursivos de T , asumiendo como hipótesis inductiva que vale para los parámetros de tipo T ,

entonces $(\forall x :: T)(\mathcal{P}(x))$

INDUCCIÓN SOBRE LISTAS:

DATA $[a] = [] \mid a : [a]$ 2 CONSTRUCCIONES = 2 CASOS

$\wedge_i \underbrace{P([])}_{CB} \wedge (\forall x::a) (\forall xs::[a]) (\underbrace{P(xs)}_{Hi} \Rightarrow \underbrace{P(x:xs)}_{QVQ}) \Rightarrow (\forall xs::[a]) (P(xs))$

{M0} $\text{MAP } f [] = []$

{M1} $\text{MAP } f (x:xs) = f x : \text{MAP } f xs$

{A0} $[] ++ ys = ys$

{A1} $(x:xs) ++ ys = x : (xs ++ ys)$

CB: $P([])$

PI: $(\forall x::a) (\forall xs::[a]) (P(xs) \Rightarrow P(x:xs))$

PROPIEDAD: $\wedge_i f :: a \rightarrow b, xs::[a], ys::[a] \Rightarrow$

$P = \text{MAP } f (xs ++ ys) = \text{MAP } f xs ++ \text{MAP } f ys$

$Hi = P(xs) : \text{MAP } f (xs ++ ys) = \text{MAP } f xs ++ \text{MAP } f ys$

$QVQ = P(x:xs) : \text{MAP } f ((x:xs) ++ ys) = \text{MAP } f (x:xs) ++ \text{MAP } f ys$

$P([]) = xs = []$

CB: $\text{MAP } f ([] ++ ys)$

{A0} $\text{MAP } f ys \rightsquigarrow (\text{Forma: } [] ++ ys \text{ donde } ys = \text{MAP } f ys)$

{A0} $[] ++ \text{MAP } f ys$

{M0} $\text{MAP } f [] ++ \text{MAP } f ys$

se cumple $P([]) \Rightarrow \text{MAP } f xs ++ \text{MAP } f ys$ para $xs = []$

PI: $\text{MAP } f ((x:xs) ++ ys)$

{A1} $\text{MAP } f (x : (xs ++ ys))$

{M1} $f x : \text{MAP } f (xs ++ ys)$

{Hi} $f x : \text{MAP } f xs ++ \text{MAP } f ys$ (Forma: $x : (xs ++ ys)$)

{A1} $(f x : \text{MAP } f xs) ++ \text{MAP } f ys$

{M1} $\text{MAP } f (x:xs) ++ \text{MAP } f ys$

QVQ

Principio de Inducción sobre Árboles Binarios

Ejemplo: principio de inducción sobre árboles binarios

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

Sea \mathcal{P} una propiedad sobre expresiones de tipo `AB a` tal que:

► $\mathcal{P}(\text{Nil})$

► $\forall i :: \text{AB } a. \forall r :: a. \forall d :: \text{AB } a.$

$$\underbrace{((\mathcal{P}(i) \wedge \mathcal{P}(d))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\text{Bin } i \ r \ d))}_{\text{T.I.}}$$

Entonces $\forall x :: \text{AB } a. \mathcal{P}(x).$

Principio de Inducción sobre Polinomios

Ejemplo: principio de inducción sobre polinomios

```
data Poli a = X
            | Cte a
            | Suma (Poli a) (Poli a)
            | Prod (Poli a) (Poli a)
```

Sea \mathcal{P} una propiedad sobre expresiones de tipo `Poli a` tal que:

► $\mathcal{P}(X)$

► $\forall k :: a. \mathcal{P}(\text{Cte } k)$

► $\forall p :: \text{Poli } a. \forall q :: \text{Poli } a.$

$$\underbrace{((\mathcal{P}(p) \wedge \mathcal{P}(q))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\text{Suma } p \ q))}_{\text{T.I.}}$$

► $\forall p :: \text{Poli } a. \forall q :: \text{Poli } a.$

$$\underbrace{((\mathcal{P}(p) \wedge \mathcal{P}(q))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\text{Prod } p \ q))}_{\text{T.I.}}$$

Entonces $\forall x :: \text{Poli } a. \mathcal{P}(x).$

Relación entre foldr y foldl

Propiedad: Si $f :: a \rightarrow b \rightarrow b, z :: b, xs :: [a]$ entonces $\text{foldr } f \ z \ xs = \text{foldl } (\text{flip } f) \ z \ (\text{reverse } xs)$

Lema: Si $g :: b \rightarrow a \rightarrow b, z :: b, x :: a, xs :: [a]$ entonces $\text{foldl } g \ z \ (xs ++ [x]) = g \ (\text{foldl } g \ z \ xs) \ x$

Puntos de vista intensional vs extensional

Sí, es con **s**. Es intensional.

¿Es equivalente mergesort = insertionSort? La realidad es que: hacen lo mismo (llegan al mismo resultado); ordenan algo pero de una manera diferente.

- Punto de vista intensional: Dos valores son iguales si están definidos de la misma manera.
- Punto de vista extensional: Dos valores son iguales si son indistinguibles al observarlos.

Entonces mergesort = insertionSort desde el lado extensional, porque hacen lo mismo pero no son iguales desde el punto de vista intensional.

Principio de Extensionalidad Funcional

Sean $f, g :: a \rightarrow b$.

Principio de extensionalidad funcional: Si $(\forall x :: a)(f\ x = g\ x)$ entonces $f = g$

El ppio. de extensionalidad funcional nos dice que si son funciones son iguales entonces son iguales punto a punto donde las evaluemos.

Principio de Extensionalidad

Nos permite predicar sobre la forma de algo particular y nos da la ventaja de poder seguir estudiando algo particular y poder sacar ciertas conclusiones.

Importante: Es súper común tener que aplicar este tipo de estrategias durante una demostración varias veces.

Principio de Extensionalidad de Booleanos

Dada una condición booleana, difurcamos en dos casos: o es verdadero, o es falso.

Ej.: $e == x$ se separa en dos casos

- $e == x = \text{true}$
- $e == x = \text{false}$

Principio de Extensionalidad sobre Listas

Dada una lista, que no sabemos que pinta tiene, podemos predicar sobre ella diciendo que tenemos dos casos

- $xs = []$
- $\exists z :: a, z :: [a] / xs = (z : zs)$

Demostración de Desigualdades

¿Cómo demostramos que no vale una igualdad $e1 = e2 :: A$?

Entiendo que básicamente hay que encontrar hacer **ALGO** para demostrar que no son iguales. Este **ALGO** es una funcionalidad que devuelva algo con uno, y otra cosa con otro. Por lo tanto debería ser del tipo $obs\ a :: A \rightarrow Bool$.

Demostrar que **no** vale la igualdad: $id = swap :: (Int, Int) \rightarrow (Int, Int)$.

Esto es fácil de probar porque sabemos que id nos da exactamente lo mismo que envíamos, y $swap$ justamente da vuelta todo. Por lo tanto podemos comparar el primer elemento con id y el primer elemento haciendo el $swap$.

Ej.: $(1, 2) \rightarrow id(1, 2) \rightarrow (1, 2)$ pero $swap(1, 2) = (2, 1)$ y el primer elemento de $id(1, 2) \rightarrow (1, 2)$ es decir, $1 \neq 2$ que arroja el $swap$.

Por lo tanto

```
1 | Ej: (1, 2)
2 | obs :: ((Int, Int) -> (Int, Int)) -> Bool
3 | obs f = fst (f (1,2)) == 1
4 |
5 | obs id -> True
6 | obs swap False
```

Nótese que el `obs` está **exactamente armado para este caso particular**. Para demostrar que las funciones no hacen lo mismo.

Isomorfismo de Tipos

Decimos que dos tipos son isomorfos si podemos pasar de uno al otro aplicando alguna función y al componerlas el resultado arroja la función identidad.

```
1 | ("hola", (1, True)) :: (String, (Int, Bool))
2 | ((True, "hola"), 1) :: (Bool, String), Int)
3 | Estos dos tipos son isomorfos porque podemos transformar los valores de un tipo en valores del otro
4 |
5 | f :: (String, (Int, Bool)) -> (Bool, String), Int)
6 | f (s, (i, b)) = f((b, s), i)
```

```

7 |
8 | g :: ((Bool, String), Int) -> (String, (Int, Bool))
9 | g ((b, s), i) = f (s, (i, b))

```

¡Es básicamente crear una función que reciba los parámetros de otra manera y los mande a la otra función de la otra forma! Una especie de Curry/Uncurry.

Con esto se puede demostrar que $g.f = id$ y $f.g = id$

Formalmente, podemos decir que dos tipos de datos A y B son **isomorfos** si

- Hay una función $f :: A \rightarrow B$ total.
- Hay una función $g :: B \rightarrow A$ total
- Se puede demostrar que $g.f = id :: A \rightarrow A$
- Se puede demostrar que $f.g = id :: B \rightarrow B$

En criollo: Debe existir una función que me mapee de f a g y viceversa. Por último, si hago $f(g)$ o $g(f)$ tienen que dar la identidad.

Notación: $A \simeq B$ indican que A y B son isomorfos.

Sistemas Deductivos

Sirve para razonar acerca de juicios (afirmaciones que queremos probar).

Ej.: Probar que el **tipo** $(Bool \rightarrow Int)$ está sintácticamente bien formado

Estructura de los Sistemas Deductivos

Está dado por **reglas de inferencia** que tienen la siguiente estructura

$$\frac{}{\langle \text{axioma} \rangle} \quad \langle \text{nombre} \rangle \quad \frac{\langle \text{premisa}_0 \rangle \langle \text{premisa}_1 \rangle \dots \langle \text{premisa}_n \rangle}{\langle \text{conclusion} \rangle} \quad \langle \text{nombre de la regla} \rangle$$

Diferencia entre Axioma y Reglas de Inferencia

- **Axioma:** Es una afirmación que se asume como verdadera.
- **Reglas de Inferencia:** Permiten derivar afirmaciones (teoremas) a partir de axiomas y otras afirmaciones.

Cómo leer reglas de inferencia

Existen dos formas, de abajo hacia arriba o de arriba hacia abajo.

- De arriba hacia abajo: Si sabemos que valen las premisas, podemos decir que vale la conclusión.
- De abajo hacia arriba: Si queremos demostrar que vale la conclusión, tenemos que demostrar todas las premisas.

Árbol de Derivación

Es la representación gráfica de una derivación.

- Los nodos representan afirmaciones.
- La raíz es la afirmación que se quiere probar.
- Las ramas representan las reglas de inferencias que conectan a las afirmaciones.

Si llegamos a una prueba correcta, **las hojas son axiomas**.

Afirmación Derivable (Teorema)

Una afirmación es derivable si existe alguna derivación sin premisas que la tiene como conclusión.

Es decir: si podés armar el árbol de derivación y llegás a hojas que son axiomas, entonces, tu afirmación es derivable.

Lógica Proposicional

Dado un conjunto infinito de variables proposicionales: $\mathcal{P} = \{P, Q, R, \dots\}$

Fórmulas

- Cualquier variable proposicional es una fórmula.
- Si ρ es una fórmula, entonces $\neg\rho$ es una fórmula.
- Si ρ y σ son fórmulas, $(\rho \wedge \sigma)$, $(\rho \vee \sigma)$, $(\rho \implies \sigma)$, $(\rho \iff \sigma)$ son fórmulas.

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**.

Gramática de la Lógica Proposicional

Las fórmulas son las expresiones que se pueden generar a partir de la siguiente gramática

$$\tau, \sigma, \rho \dots ::= P \mid \perp \mid (\tau \wedge \sigma) \mid (\tau \implies \sigma) \mid (\tau \vee \sigma) \mid \neg\tau$$

Llamamos **contexto** a un conjunto finito de fórmulas.

Good to Know

- Los conectivos \wedge, \vee no son conmutativos ni asociativos.
- La implicación, al igual que en Haskell, asocia a la derecha.
- Ómitimos los paréntesis más externos de las fórmulas.
- \perp es siempre falso

Valuación

Una valuación es una función $v : \mathcal{V} \implies \{V, F\}$ que asigna valores de verdad a las variables proposicionales.

Una valuación **satisface** una proposición τ si $v \models \tau$ donde:

- $v \models P$ sii $v(P) = V$
- $v \models \neg\tau$ sii $v \not\models \tau$
- $v \models \tau \vee \sigma$ sii $v \models \tau$ o $v \models \sigma$
- $v \models \tau \wedge \sigma$ sii $v \models \tau$ y $v \models \sigma$
- $v \models \tau \implies \sigma$ sii $v \not\models \tau$ o $v \models \sigma$
- $v \models \tau \iff \sigma$ sii $v \models \tau$ sii $v \models \sigma$

Nota: Una valuación es una fila de la tabla de verdad. Es decir, cada variable se le asigna un valor de verdad para esa valuación. Ej.: Si tengo $\sigma \wedge \tau$ ¿qué necesito que pase para σ y τ para que sea verdadero? Bueno, necesito que la valuación de $v \models \sigma$ y $v \models \tau$. A modo de ejercicio escribamos uno de estas valuaciones con árboles de derivación a nivel de sistema deductivo:

$$\frac{v \models \tau \quad v \models \sigma}{v \models \tau \wedge \sigma} \quad \langle \wedge \rangle$$

Contexto

Es un conjunto finito de fórmulas. Es lo que tenemos como información en nuestros juicios a medida que vamos encontrando las premisas.

Ej.: $\Gamma = P \implies Q, \neg Q$

Semántica en Lógica Proposicional

Una valuación v satisface un contexto Γ ($v \models \Gamma$) sí y solo sí v satisface todas las fórmulas de Γ .

Si $\Gamma = P \wedge Q, \neg Q$ entonces para que $v \models \Gamma$ tiene que suceder que:

- $v \models P \wedge Q$ sii $v \models P$ y $v \models Q$
- $v \models \neg Q$ sii $v \not\models Q$

Good to Know: Todo v satisface al contexto vacío.

Consecuencia Lógica: Si llegase a encontrar ahora una fórmula τ del conjunto Γ ($\Gamma \models \tau$), entonces, todas las valuaciones v que valían en Γ , valen también en τ (si antes valía, y ahora agregué τ veo que siga valiendo).

Equivalencia Lógica y Tipos de Fórmulas

Dadas dos fórmulas τ y σ :

- τ es lógicamente equivalente a σ cuando $v \models \tau$ sii $v \models \sigma$ 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 $\forall \tau \in \Gamma$ se tiene $v \models \tau$
- Insatisfactible si no es satisfactible.

Teorema de la Insatisfactibilidad

Una fórmula τ es una tautología sii $\neg \tau$ es insatisfactible.

Sistema Deductivo (Deducción Natural)

El enfoque que vimos antes de la lógica proposicional a través de la semántica tiene algunas limitaciones. En nuestro caso, vemos el sistema de deducción natural aunque existen otros. Este sistema trabaja con afirmaciones (juicios) de la forma $\Gamma \vdash \tau$ donde Γ es la hipótesis y τ es la tesis.

Lógica Intuicionista (NJ) vs Lógica Clásica (NK)

Las reglas de la lógica intuicionista están contenidas en las reglas de la lógica clásica. Algunas reglas que podemos aplicar en la **lógica clásica** que no podemos usar en la lógica intuicionista

- PBC (proof by contradiction): No es derivable, es decir, no se puede deducir.
- LEM (law of excluded middle): Es un axioma, no tiene premisas y por lo tanto no se puede probar.
- $\neg\neg e$ (doble negation elimination)

Las reglas **PBC**, **LEM** y $\neg\neg e$ son equivalentes.

$\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \wedge \sigma} \wedge_i$		$\overline{\Gamma, \tau \vdash \tau} \text{ ax}$	
$\frac{\Gamma, \tau \vdash \sigma}{\Gamma \vdash \tau \Rightarrow \sigma} \Rightarrow_i$		$\frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \tau} \wedge_{e1} \quad \frac{\Gamma \vdash \tau \wedge \sigma}{\Gamma \vdash \sigma} \wedge_{e2}$	
$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \vee \sigma} \vee_{i1} \quad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \tau \vee \sigma} \vee_{i2}$		$\frac{\Gamma \vdash \tau \Rightarrow \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash \sigma} \Rightarrow_e$	
$\frac{\Gamma, \tau \vdash \perp}{\Gamma \vdash \neg \tau} \neg_i$		$\frac{\Gamma \vdash \tau \vee \sigma \quad \Gamma, \tau \vdash \rho \quad \Gamma, \sigma \vdash \rho}{\Gamma \vdash \rho} \vee_e$	
Lógica intuicionista		$\frac{\Gamma \vdash \tau \quad \Gamma \vdash \neg \tau}{\Gamma \vdash \perp} \neg_e$	
		$\frac{\Gamma \vdash \perp}{\Gamma \vdash \tau} \perp_e$	
Lógica clásica		$\frac{\Gamma \vdash \neg \neg \tau}{\Gamma \vdash \tau} \neg\neg_e$	

Reglas intuicionistas	
$\frac{\Gamma \vdash \tau}{\Gamma \vdash \neg\neg\tau} \neg\neg_i$	$\frac{\Gamma \vdash \tau \Rightarrow \sigma \quad \Gamma \vdash \neg\sigma}{\Gamma \vdash \neg\tau} \text{MT}$
Reglas clásicas	
$\frac{\Gamma, \neg\tau \vdash \perp}{\Gamma \vdash \tau} \text{PBC}$	$\frac{}{\Gamma \vdash \tau \vee \neg\tau} \text{LEM}$

Si un juicio es derivable en NJ también lo es en NK, esto se debe a que NJ es más restrictivo que NK.
¿Por qué es importante NJ? Porque es la base de un lenguaje de programación funcional.

Weakening (Debilitamiento) y Strengthening (Fortalecimiento)

Weakening

Si $\Gamma \vdash \tau$ es derivable, entonces $\Gamma, \sigma \vdash \tau$ es derivable.

Strengthening

Es lo contrario a Weakening, la idea es que podemos ir sacando hipótesis que no nos sirven y poder seguir probando que vale. Se utiliza mucho en algo llamado Cálculo Lambda que veremos luego. Se dice que es Strengthening porque estamos haciendo más fuerte la prueba, esto es porque a menor cantidad de información que tiene el contexto es más fácil de probar.

Correctitud (Soundness) y Completitud

Importante: Utilizamos \models para hablar de satisfacibilidad en el contexto de **semántica** mientras que utilizamos \vdash para indicar que una fórmula se puede derivar en el contexto de **sintaxis**.

Correctitud y Completitud responde a la siguiente pregunta: ¿cuál es la relación entre la sintaxis y la semántica?

- **Sintaxis:** Conjunto de fórmulas τ tal que $\neg\tau$ es un seciente válido.
- **Semántica:** Conjunto de fórmulas τ tal que $v \models \tau$ para toda valuación v . Ej: Tautologías.

$$\Gamma \models T \iff \Gamma \vdash T$$

Corrección

$\models \tau$ seciente válido implica que τ es tautología.

- $\sigma_1, \sigma_2, \dots, \sigma_n \vdash \tau$ seciente válido implica que $\sigma_1, \sigma_2, \dots, \sigma_n \models \tau$

Para demostrar: Hacemos inducción en la estructura de la prueba analizando por casos la última regla aplicada en la prueba. Caso base, la última regla fue un axioma.

Casos recursivos, el resto de reglas $\wedge_i, \wedge_{e1}, etc$

Completitud τ tautología implica que $\vdash \tau$ es seciente válido.

- $\sigma_1, \sigma_2, \dots, \sigma_n \models \tau$ implica que $\sigma_1, \sigma_2, \dots, \sigma_n \vdash \tau$ es un seciente válido.

Para demostrar: Usamos el contrarrecíproco: si $P \implies Q$ vale que $\neg Q \implies \neg P$
Luego probar estos dos lemas

- Si $\Gamma \not\vdash \tau$ entonces $\Gamma \cup \{\neg\tau\}$ es consistente (sale por contrarrecíproco).
- Si Γ es consistente, entonces tiene modelo. Ej.: Γ es satisfactible (es más difícil).

Consecuencia Semántica

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

$\tau_1, \tau_2, \dots, \tau_n \models \sigma \iff \forall v \text{ valuacion } ((v \models \tau_1 \wedge v \models \tau_2 \wedge \dots \wedge v \models \tau_n) \implies (v \models \sigma))$

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 $i \in 1, \dots, n \implies v \models \sigma$

Consistencia

Decimos que Γ es consistente si $\Gamma \not\vdash \perp$, es decir, Γ es consistente si no se puede derivar una contradicción a partir de él.

Cálculo Lambda (λ)

Es un lenguaje de programación. Puede computar cualquier cosa. En lenguajes funcionales hablamos de cálculo lambda mientras que en lenguajes imperativos hablamos de máquinas de Turing.

Para entrar en calor, comenzaremos hablando del cálculo lambda no tipado. Luego, introduciremos la noción de tipos.

Reglas Generales

Es probable que muchas cosas que mencione acá no se hayan definido todavía, pero ténganlo presente a medida que lean el documento. Esto, más que nada, para que las reglas más importantes estén en un único lugar.

- La aplicación es asociativa a izquierda.
 - $MNR = (MN) R$
- El constructor de tipos \rightarrow es asociativo a derecha.
 - $\tau \rightarrow \sigma \rightarrow \rho = \tau \rightarrow (\sigma \rightarrow \rho)$
- La aplicación tiene mayor precedencia (se resuelve primero) que la abstracción, y que el condicional. (APP $\hat{>}$ ABS $\hat{>}$ COND)
 - $\text{if } x \text{ then } y \text{ else } z \text{ true} \equiv \text{if } x \text{ then } y \text{ else } (z \text{ true}).$
 - $\lambda x : \tau. \text{if } x \text{ then } y \text{ else } z \equiv \lambda x : \tau. (\text{if } x \text{ then } y \text{ else } z).$
 - $\lambda x : \tau. \text{if } x \text{ then } y \text{ else } z \text{ true} \stackrel{APP}{=} \lambda x : \tau. \text{if } x \text{ then } y \text{ else } (z \text{ true}) \stackrel{ABS}{=} \lambda x : \tau. (\text{if } x \text{ then } y \text{ else } (z \text{ true})).$
- $\text{fv}(M)$: Es el conjunto de variables libres de **cada posible término M** de un sistema de tipos. Una variable está ligada si está al alcance de una abstracción.
- Todas las funciones toman un único argumento.
 - $\lambda x. \lambda y. \lambda z. (\text{if } x \text{ then } y \text{ else } z)$ genera $(\text{if } x \text{ then } y \text{ else } z)$ con los argumentos ya aplicados.

Con booleanos

Los términos válidos de cálculo lambda con booleanos los podemos describir con una gramática formal

$M ::= x$	(variable)
$\lambda x. M$	(abstracción)
$M N$	(aplicación)
true	(constante verdadera)
false	(constante falsa)
if M then N else P	(condicional)

- **Tipos:** $\tau, \sigma, \rho, \dots ::= \text{bool} \mid \tau \rightarrow \sigma.$
- **Términos:** Variables $X = \{X, Y, Z, \dots\}$

Cálculo Lambda tipado

En el Cálculo Lambda tipado damos los tipos en las operaciones, ej.: $\lambda x : \sigma. M$

Función Identidad

En cálculo lambda existe una función id **propia por cada tipo**.

Captura de variables (α equivalencias)

Decimos que dos términos M y N son α equivalentes si solamente difieren en el nombre de sus variables ligadas.

Ej.: $\lambda x. \lambda y. x = \lambda y. \lambda x. x$

Conjunto de Variables Libres $fv(M)$

El conjunto de variables libres de un término M se define para todo posible término del sistema de tipos y se denota como $fv(M)$.

Definamos, para cada término del cálculo lambda en booleanos el conjunto de variables libres

- $fv(x) \stackrel{def}{=} \{x\}$
- $fv(\lambda x. M) \stackrel{def}{=} fv(M) \setminus \{x\}$
- $fv(true), fv(false) \stackrel{def}{=} \emptyset$
- $fv(MN) \stackrel{def}{=} fv(M) \cup fv(N)$
- $fv(if\ M\ then\ N\ else\ P) \stackrel{def}{=} fv(M) \cup fv(N) \cup fv(P)$

Si $fv(M) = \emptyset$ entonces M es **cerrado** es decir, no tiene variables libres.

Sistema de Tipos

Se formaliza con un sistema deductivo. Solo tipamos términos sin variables libres. Indica **cómo se construyen los programas**.

Contexto de tipado

¿Qué sucede con el siguiente programa? $(\lambda x : Bool. (y\ x))$

Como y puede (o no) tener variables libres, necesitamos un **contexto de tipado**.

Un contexto de tipado es un conjunto finito de pares $(x_i : \tau_i) : \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ **sin variables repetidas** ($i \neq j \implies x_i \neq x_j$)

A veces notamos $dom(\Gamma) = \{x_1, \dots, x_n\}$, por lo que, si queremos saber el tipo de x_i , podemos escribir $\Gamma(x_i)$

Decimos que el contexto de tipado está vacío (\emptyset) cuando **o bien el término no tiene ninguna variable libre o bien cuando el término es una constante**

Juicios de Tipado

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

Se lee como "en el contexto Γ , M es de tipo τ "

The image shows a handwritten derivation of a type judgment on a grid background. The derivation uses several inference rules, some labeled with Ax (axioms) and \rightarrow (implication).

Top part (Axioms and initial steps):

- Ax_v : $\Gamma' \vdash x : Bool$
- Ax_T : $\Gamma' \vdash true : Bool$
- Ax_U : $\Gamma' \vdash y : Bool$

IF rule:

$\Gamma' \vdash x : Bool, \Gamma' \vdash true : Bool, \Gamma' \vdash y : Bool \vdash if\ x\ then\ true\ else\ y : Bool$

\rightarrow_i rule (Implication Introduction):

$x : Bool \vdash \lambda y : Bool. if\ x\ then\ true\ else\ y : (Bool \rightarrow Bool)$

\rightarrow_e rule (Implication Elimination):

$\Gamma \vdash (\lambda x : Bool. \lambda y : Bool. if\ x\ then\ true\ else\ y) : (Bool \rightarrow Bool) \rightarrow Bool$ (using Ax_F : $\Gamma \vdash false : Bool$)

Final result:

$\vdash (\lambda x : Bool. \lambda y : Bool. if\ x\ then\ true\ else\ y)\ false : Bool \rightarrow Bool$

Annotations:

- A blue bracket under the lambda expression is labeled M .
- A blue note says "Ax. Desc = Aislado".
- The word "VÁLIDO." is written at the bottom left.

$$\begin{array}{c}
 \frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{T-TRUE} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{T-FALSE} \\
 \\
 \frac{\Gamma \vdash M : \text{bool} \quad \Gamma \vdash N : \tau \quad \Gamma \vdash P : \tau}{\Gamma \vdash \text{if } M \text{ then } N \text{ else } P : \tau} \text{T-IF} \\
 \\
 \frac{}{\Gamma, x : \tau \vdash x : \tau} \text{T-VAR} \quad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau. M : \tau \rightarrow \sigma} \text{T-ABS} \\
 \\
 \frac{\Gamma \vdash M : \tau \rightarrow \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma} \text{T-APP}
 \end{array}$$

Unicidad de Tipos

Si $\Gamma \vdash M : \sigma$ y $\Gamma \vdash M : \rho$ son derivables ent $\sigma = \rho$

Importante: No vale en todos los sistema de tipos, pero vale en el cálculo lambda tipado y es fundamental.

Teorema (Weakening + Strengthening)

Si $\Gamma \vdash M : \tau$ es derivable y $fv(M) \subseteq \Gamma \cap \Gamma'$ entonces $\Gamma' \vdash M : \tau$ es derivable

- Weakening: Agregar más hipótesis hace que la afirmación sea más débil y difícil de demostrar.
- Strengthening: Sacar hipótesis hace que la afirmación sea más fuerte y fácil de demostrar.

Lo importante es acordarse que

- Si en el contexto tenés algo que te sirve para tipar en M, entonces podés agregar cuantas cosas quieras en el contexto que va a seguir valiendo (weakening).
- Si en el contexto tenés algo que NO te sirve para tipar en M, entonces podés sacarlo que no te va a afectar en nada (strengthening).

Tipos Habitados

Un tipo está habitado si existe un término M tal que el juicio $\vdash M : \tau$ es derivable.

Por ejemplo, si nos piden algo que cumpla $\vdash M : (\tau \rightarrow \rho) \rightarrow (\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \rho)$.

Podemos notar que una función que haga esto, sería la composición: $\lambda f : \tau \rightarrow \rho. \lambda g : \sigma \rightarrow \tau. \lambda x : \sigma. f(gx)$ o mejor escrita $(\lambda f : \tau \rightarrow \rho)(\lambda g : \sigma \rightarrow \tau)(\lambda x : \sigma)(f(gx))$

Semántica Formal

Nos permite darle significado a nuestros programas.

- **Semántica Operacional:** Indica cómo se ejecuta el programa hasta llegar a un resultado. Es el tipo de semántica que usamos en la materia que trabaja en base a reglas de reducción entre términos. La idea es llegar a expresiones irreducibles.
 - small-step: ejecución paso a paso.
 - big-step: evaluación directa al resultado.
- **Semántica Denotacional:** Interpreta los programas como objetos matemáticos.
- **Semántica Axiomática:** Establece relaciones lógicas entre el estado del programa antes y después de la ejecución.

Forma Normal

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

Determinismo

Cuando cada término que **NO** está en forma normal tiene una única forma de reducir.

Estrategia de Reducción

Usaremos la estrategia de reducción **call by value**.

$M \rightarrow N$: indica que M reduce o reescribe a N.

Programa

Es un término M tipable y cerrado ($\text{fv}(M) = \emptyset$) que debe ser derivable para algún τ .

Juicios de Evaluación

La semántica operacional hace afirmaciones sobre juicios de evaluación $M \rightarrow N$ donde M y N son programas.

Valores

Los valores son los posibles resultados de evaluar programas, se definen como los términos cerrados y bien tipados V producidos por la gramática de valores.

La estrategia call-by-value define los siguientes valores: $V ::= \text{true} \mid \text{false} \mid \lambda x : \sigma. M$, estos ya tienen todos sus parámetros completamente reducidos, de lo contrario, no serían valores.

Ej.: $V ::= \dots \mid \text{Nil} \mid \text{Bin}(V, V, V)$ corresponde a los valores de trabajar con un árbol binario.

Obtener Valores

Para poder obtener valores, debemos ir reduciendo las expresiones. Depende de qué sistema de tipos y operaciones tengamos, cada una tiene su forma de reducirse.

A continuación, se demuestran las reglas de reducción para las expresiones booleanas y funciones.

Reglas de evaluación para expresiones booleanas

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

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

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

Reglas de evaluación para funciones (abstracción y aplicación)

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

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

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

- 1. Si la guarda es verdadera, entonces el término se reduce a M.
- 2. Si la guarda es falsa, entonces el término se reduce a N.
- 3. Si el término de la guarda se puede reducir, entonces la reducimos, para obtener el valor (verdadera o falsa). Luego de este paso, aplicamos la regla 1 o la regla 2.
- 4. Si el término de M se puede reducir, lo reducimos. Luego le aplicamos a N.
- 5. Si el término N se puede reducir, entonces lo reducimos.
- 6. Si el término N es un valor, sustituyo el parámetro de la lambda con el con el valor y lo aplico en el término M.

Estas reglas tienen nombres específicos, y se las puede encontrar en la siguiente imagen

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

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

$$\begin{aligned} (\lambda x : \sigma. M)V &\rightarrow M\{x := V\} && (E\text{-AppAbs o } \beta) \\ \text{if true then } M \text{ else } N &\rightarrow M && (E\text{-IfTrue}) \\ \text{if false then } M \text{ else } N &\rightarrow N && (E\text{-IfFalse}) \end{aligned}$$

Si $M \rightarrow N$, entonces:

$$\begin{aligned} MO &\rightarrow NO && (E\text{-App}_1 \text{ o } \mu) \\ VM &\rightarrow VN && (E\text{-App}_2 \text{ o } \nu) \\ \text{if } M \text{ then } O \text{ else } P &\rightarrow \text{if } N \text{ then } O \text{ else } P && (E\text{-If}) \end{aligned}$$

Sustitución

La operación de sustitución $M\{x := N\}$ nos devuelve un nuevo término tal que resulta de reemplazar todas las ocurrencias libres de x en M por N .

Nota: Se implementa por pattern matching porque es un constructor recursivo.

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

Propiedades de la Evaluación

Teorema (Determinismo)

Si $M \rightarrow N_1$ y $M \rightarrow N_2$ entonces $N_1 = N_2$

En criollo: No hay más de una forma de reducir algo.

Teorema (Preservación de Tipos o Subject Reduction)

Si $\Gamma \vdash M : \tau$ y $M \rightarrow N$ entonces $\Gamma \vdash N : \tau$

Si deducimos el tipo τ para un término sin ejecutar el programa, y luego ejecutamos el programa obteniendo N , entonces el término N tiene el mismo tipo.

Teorema (Progreso)

Si $\vdash M : \tau$ entonces:

- O bien M es un término cerrado irreducible bien tipado, o sea un valor.
- O bien existe un N tal que $M \rightarrow N$

Teorema (Terminación)

Si $\vdash M : \tau$ entonces no hay una cadena infinita de pasos: $M \rightarrow M_1 \rightarrow M_2 \rightarrow \dots$

Canonicidad

- Si $\vdash M : \text{bool}$ es derivable, entonces la evaluación de M termina y el resultado es true o false.
- Si $\vdash M : \tau \rightarrow \sigma$ es derivable, entonces la evaluación de M termina y el resultado es una abstracción.

Estado de error

Cuando el término está en forma normal pero no es un valor.

Operador fix (\perp)

Es útil cuando necesitamos devolver una respuesta en un caso inválido con un tipo específico.

Ej.: $\text{raiz}_\sigma = \lambda x : AB \ \sigma \rightarrow \text{case } x \text{ of } Nil_\sigma \rightarrow \perp_\sigma \mid \text{Bin}(i, r, d) \rightarrow r$

Anotación de Tipo (Tipo Anotado)

Un término tiene anotación de tipo cuando se indica, en el término, su tipo. Ej.: Nil_σ

Subtérmino

Un subtérmino no es más que un término dentro de otro término.

Ej.: $\text{Bin}(M, N, O)$ el término es Bin y los subtérminos son M , N y O .

Ej.: $\text{Case } M \text{ of } Nil \rightarrow \text{True} \mid \text{Bin}(i, r, d) \rightarrow \text{False}$, el término es el case y los subtérminos son True y False.

Extensiones del Cálculo Lambda

Voy a escribir un par de tips o cosas interesantes que encontré, y después voy a escribir algunos ejercicios como guía que justamente utilizan estos tips.

En las reglas de tipado tenemos contexto, en las reglas de semántica, no. Hay una **pequeña excepción**, y es, cuando necesitamos de antemano conocer como dato el tipo de salida de una función a la hora de aplicar la regla de semántica.

Valores

Los valores de la extensión de cálculo lambda particular son los valores con los que quiero operar en mis términos. Las operaciones que devuelven cosas que no reducen a algo de mi tipo, no son valores posibles.

Ej.: $\tau ::= \dots \mid Nil_\sigma \mid Bin(M, N, O)_\sigma \mid esRaiz(M) \mid esNil(M) \mid \text{case } M \text{ of } Nil \rightarrow N; Bin(i, r, d) \rightarrow O$

De acá los valores de mi nueva extensión serían: Nil_σ y $Bin(V, V, V)$. ¿Por qué case, esRaiz y esNil no? Porque no nos aportan ningún posible valor nuevo.

Si reducimos $raiz(M)$ va a terminar siendo, o bien un Nil o un $Bin(V1, V2, V3) \rightarrow V2$

Si reducimos $esNil(M)$ va a terminar siendo, o bien true o false que opera en base a un Nil o un $Bin(V, V, V)$.

Importante: ¿Qué pasa si hacemos $raiz(Nil)$? Explota (perdemos progreso, porque no puede reducirse más y no es un valor). En estos casos particulares que no está definido, no haríamos una regla de tipado para ese caso, y, si tenemos que hacer una macro, en el caso nil devolveríamos un \perp del tipo de salida del case.

Cantidad de Reglas de Tipado

Agregamos, al menos una por cada término.

Reglas de tipado con términos con ocurrencias libres

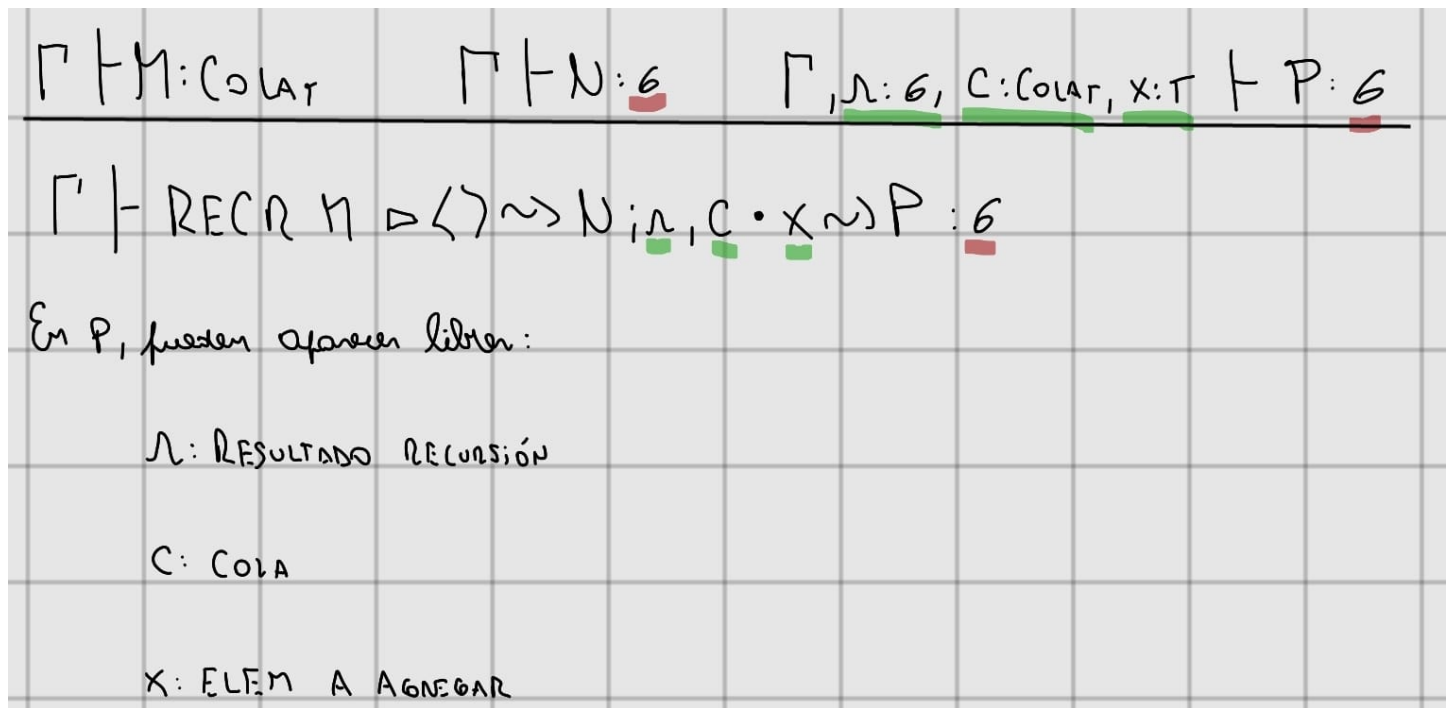
Si tenemos una estructura del tipo condicional como un case, que devuelve un término que puede tener variables libres, entonces, en el contexto de ese caso, tenemos que añadir las variables que necesite.

$\Gamma \vdash M : \sigma \vdash \tau$	$\Gamma, x : \sigma \vdash N : \tau$	$\Gamma, y : \tau \vdash O : \tau$
$\Gamma \vdash \text{CASE } M \text{ OF LEFT}(x) \leadsto N \mid \text{RIGHT}(y) \leadsto O : \tau$		

- Y PODRÍA APARECER LIBRE EN O.
- X PODRÍA APARECER LIBRE EN N.
- El tipo τ representa lo que voy a devolver las operaciones del CASE (si o si del mismo tipo)

$\Gamma \vdash M : AB_\tau$	$\Gamma \vdash N : \sigma$	$\Gamma, i : AB_\tau, r : \tau, d : AB_\tau \vdash O : \sigma$
$\Gamma \vdash \text{CASE } M \text{ OF Nil} \leadsto N; Bin(i, r, d) \leadsto O : \sigma$		

- Nil o N o ser nula NO aparece en contexto. No lo necesitamos
- i, r, d PODRÍAN aparecer libres en O.
- El tipo τ representa lo que voy a devolver las operaciones del CASE (si o si del mismo tipo)



Cantidad de Reglas de Congruencia (Semántica)

Normalmente, tenemos una por cada término. No obstante, si el término tiene más de un argumento, va a tener esa cantidad. Ej.: $Bin(M, N, O)$ es un único término pero que tiene 3 subtérminos. Por lo tanto, tendría 3 reglas de congruencia para reducirlo a un valor.

Cantidad de Reglas de Cómputo (Semántica)

Normalmente tenemos una por valor. No obstante, si es una estructura tipo if/case tenemos más de una por cada posible salida.

Importante: Si las salidas condicionales son un término que podría eventualmente tener variables libres, hay que sustituir esas variables libres por el valor correspondiente que deberán tener ligado.

Importante: Como estamos en call-by-value, **no reducimos ni casos then/else** de estructuras de control. Es decir, en un case, solo reduciríamos su guarda, y lo mismo en el if.

Cantidad de Reglas de Cómputo

Normalmente, agregamos una por cada término usando los valores del tipo reducido. No obstante, hay veces que un término tiene más de una regla de cómputo.

Ej.: Un case reduce a dos posibles cosas. Por lo tanto, tendría dos reglas de cómputo.

Inferencia de Tipos

Términos sin anotaciones de tipos: $U :: = x \mid \lambda x. U \mid U \mid U \mid \text{True} \mid \text{False} \mid \text{if } U \text{ then } U \text{ else } U$

Términos con anotaciones de tipos: $U :: = x \mid \lambda x. M \mid M \mid M \mid \text{True} \mid \text{False} \mid \text{if } U \text{ then } U \text{ else } U$

erase(M)

Es el término sin anotaciones de tipos que resulta de borrarlos del término M. El resultado es U.

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

Esto nos quiere decir que, si tenemos un término M es fácil conseguir U pero no al revés.

¿Cuándo podemos tipar un término U?

Un término U sin anotaciones de tipos es tipable sí y solo si existen:

- Un contexto de tipado Γ
- Un término con anotaciones de tipos M

- Un tipo τ tales que $\text{erase}(M) = U$ y $\Gamma \vdash M : \tau$

Entonces, el problema de **inferencia de tipos** consiste en:

- Dado un término U determinar si es tipable.
- En caso de que U sea tipable, entonces
 - Hallar un contexto Γ , un término M y un tipo τ tales que cuando hagamos $\text{erase}(M) = U$ y $\Gamma \vdash M : \tau$

En criollo: Si tengo un U , entonces buscate un M , que tenga contexto, un τ y que si le mando ese M me da U .

Ej.: $\lambda x.x$ es la función identidad. ¿Cómo podemos tiparla?. De antemano, x podría ser cualquier cosa, una función, un valor, lo que realmente quiera.

Entonces para poder tiparla, necesitamos de antemano, saber en el contexto de qué tipo es x , por lo tanto, podemos decir que $\Gamma \vdash \{x : \alpha\}$.

Ya tenemos entonces Γ y U . Nos falta conseguir τ que sería lo que nos falta para poder concluir $\Gamma \vdash M : \tau$. Como es una Lambda que recibe un x , y devuelve un x , entonces el tipo es $\tau : \alpha \rightarrow \alpha$.

Entonces, armamos el término M : $\lambda(x : \alpha).x$ y podemos chequear ambas condiciones.

- $\text{erase}(\lambda(x : \alpha).x) = \lambda x.x$ y esto es verdadero.
- $\Gamma \vdash (\lambda(x : \alpha).x) : \alpha \rightarrow \alpha$

$$\Gamma, x : \alpha \vdash x : \alpha$$

T-ABS

$$\Gamma \vdash (\lambda(x : \alpha).x) : \alpha \rightarrow \alpha$$

y es justamente lo que queríamos derivar.

Importante: Solo tipo en M si \exists una lambda. Ej.: $U = x \text{ True} \equiv M = x \text{ True}$ pues no existe ninguna lambda en U .

Incorporando incógnitas a los tipos

El algoritmo de inferencia de tipos se basa en manipular tipos **parcialmente conocidos**.

Ej.: $x \text{ True}$, ¿cómo podemos tiparlo?. Bueno, recordemos que asociamos a la izquierda, por lo tanto sería algo así: $x(\text{True})$ por lo tanto, x debería de ser una función que recibe obligatoriamente un booleano y devuelve un tipo desconocido. Ese tipo desconocido lo denotamos con una variable.

Por lo tanto, el tipo sería: $(x : \text{Bool} \rightarrow X_1)$

Ej. 2: $(X_1 \rightarrow X_1) \stackrel{?}{=} ((\text{Bool} \rightarrow \text{Bool}) \rightarrow X_2)$.

En criollo lo que nos pide es: ¿cómo tiene que ser X_1 y X_2 para que ambos lados se verifiquen? bueno, si vemos la ecuación izquierda antes de \rightarrow tiene $\text{Bool} \rightarrow \text{Bool}$ entonces no queda otra que $X_1 = \text{Bool} \rightarrow \text{Bool}$ Así mismo, entonces, del lado izquierdo nos queda $\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} = (\text{Bool} \rightarrow \text{Bool})X_2$ entonces la única forma que tiene X_2 es $\text{Bool} \rightarrow \text{Bool}$.

Por lo tanto, $X_1 : \text{Bool} \rightarrow \text{Bool}$ y $X_2 : \text{Bool} \rightarrow \text{Bool}$

Ej. 3: if xy then y else true

La estrategia en los condicionales, o lugares donde hay ramas es la siguiente: **considerar y tipar cada rama por separado, luego se hace puesta en común.**

Entonces, en este caso tenemos que tipar tres ramas, una incógnita por cada variable:

- $xy : \Gamma_c = \{x : X_1 \rightarrow X_2, y : X_1\}$
- $y : \Gamma_t = \{t : X_3\}$
- $\text{true} : \Gamma_e = \emptyset$

Entonces ahora planteamos un sistema de ecuaciones

$$\begin{cases} X_3 \stackrel{?}{=} \text{Bool} \\ X_1 \stackrel{?}{=} X_3 \\ X_2 \stackrel{?}{=} \text{Bool} \end{cases}$$

Sustituimos ahora las incógnitas por los tipos y nos queda $X_1 = \text{Bool}, X_2 = \text{Bool}, X_3 = \text{Bool}$.

Resolver ecuaciones del tipo $X_1 \stackrel{?}{=} X_2 \rightarrow X_3$ se llama UNIFICACIÓN. Reemplazando las variables deberíamos llegar a una igualdad.

Unificación

Es el problema de resolver sistemas de ecuaciones entre tipos con incógnitas. Es muy importante pues luego lo usamos para dar un algoritmo de inferencia de tipos.

Tenemos conjuntos finitos de constructores de tipos

- Tipos constantes (no tienen parámetros): $\text{Bool}, \text{Int}, \dots$
- Constructores Unarios (reciben un parámetro): $(\text{List } \bullet), (\text{Maybe } \bullet), \dots$

- Constructores Binarios (reciben dos parámetros): $(\bullet \rightarrow \bullet)$, $(\bullet \times \bullet)$, $(Either \bullet \bullet)$

Los **tipos** se forman usando incógnitas y constructores: $\tau ::= X_n \mid C(\tau_1, \dots, \tau_n)$

Antes de definir formalmente la Unificación, primero veamos la Sustitución, que es una de las herramientas que vamos a usar.

Sustitución

Es una función que a cada incógnita le asocia un tipo.

Notamos: $\{T_1 \stackrel{?}{=} \sigma_1, \dots, T_k \stackrel{?}{=} \sigma_k\}$ a la sustitución S tal que $S(T_1) = S(\sigma_1) \wedge \dots \wedge S(T_k) = S(\sigma_k)$

Si σ es un tipo, escribimos $S(\sigma)$ para que cada incógnita de σ sea reemplazada por lo que haya sido definido en S .

Ej.: Si $S = \{X_1 := Bool, X_3 := (X_2 \rightarrow X_2)\}$ entonces $S((X_1 \rightarrow Bool) \rightarrow X_3) = (Bool \rightarrow Bool) \rightarrow (X_2 \rightarrow X_2)$

Unificación

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

Un **unificador** para E es una sustitución S tal que

- $S(\tau_1) = S(\sigma_1)$
- ...
- $S(\tau_n) = S(\sigma_n)$

Ej.: $\{X_1 \stackrel{?}{=} X_2\}$ mientras que sea una igualdad, se puede decir cualquier cosa de X_1 y X_2 . Todas serán solución pero si la solución tiene menos variables para solucionar el problema, es más general.

Sustitución más general

Una sustitución S_A es **más general** que una sustitución S_B si existe una sustitución S_C tal que: $S_B = S_C \circ S_A$

En Criollo: Si S_B nace de S_A particularmente, entonces S_A es más general.

Algoritmo de unificación Martelli-Montanari (M-M) y la Corrección del Algoritmo

Consiste en aplicar secuencialmente un conjunto de reglas las cuales hay dos chances: o falla, o sigue adelante reemplazando. Este algoritmo siempre termina para cualquier problema de unificación E .

- Mientras que $E \neq \emptyset$ se aplica sucesivamente alguna de las reglas.
- La regla puede resultar en una falla (si E no tiene solución)
- De lo contrario, la regla es de la forma $E \rightarrow_s E'$ y al aplicarlo sucesivamente, si no hay falla, el algoritmo llega a \emptyset :
 - $E = E_0 \rightarrow_{S_1} E_1 \rightarrow_{S_2} E_2 \dots \rightarrow_{S_n} E_n = \emptyset$ $S = S_n \circ S_{n-1} \dots \circ S_2 \circ S_1$ es un unificador para E , es el más general posible y lo denotamos $\text{mgu}(E)$.

Reglas del Algoritmo (M-M)

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

¿Por qué los condicionales en las reglas?

- Swap: τ no debe ser una variable. Debe ser un valor porque sino entramos en un loop infinito de hacer referencia a sí mismo. Ej.: $4 = x \equiv x = 4$ pero $x = y \neq y = x$ NO es lo mismo porque NO son valores.
- Elim: No podemos reemplazar un término que depende de sí mismo y reducirlo. Ej.: $X_1 = X_1 \rightarrow Bool$ no tiene sentido sustituir en este caso porque vamos a estar en prácticamente un loop infinito.
- Clash: Si no tienen la misma cantidad de elementos falla.
- Occurs-Check: Solo podemos unificar si la incógnita no aparece en τ . Ej.: $X_1 = X_1 \rightarrow Bool$ falla pero $X_1 = Int$ funciona.

La regla Decompose es muy útil para separar funciones. La Delete la hacemos cuando nos queda una igualdad trivial. La Elim la usamos dejando todo menos la igualdad que queremos quitar.

Handwritten notes showing the unification process:

$$\{ \overset{\tau_1}{(x_2 \rightarrow (x_1 \rightarrow x_1))} \stackrel{?}{=} \{ \overset{\tau_2}{(Bool \rightarrow Bool) \rightarrow (x_1 \rightarrow x_2)} \}$$

DECOMPOSE $\rightarrow \{ x_2 \stackrel{?}{=} (Bool \rightarrow Bool); x_1 \rightarrow x_1 \stackrel{?}{=} x_1 \rightarrow x_2 \}$

DECOMPOSE $\rightarrow \{ x_2 \stackrel{?}{=} (Bool \rightarrow Bool); x_1 \stackrel{?}{=} x_1; x_1 \stackrel{?}{=} x_2 \}$

DELETE $\rightarrow \{ x_2 \stackrel{?}{=} (Bool \rightarrow Bool); x_1 \stackrel{?}{=} x_2 \}$

ELIM $\rightarrow \{ x_1 \stackrel{?}{=} (Bool \rightarrow Bool) \}$

ELIM $\rightarrow \emptyset \equiv Bool \rightarrow Bool$

Nota: DECOMPOSE ahora lo \rightarrow

$$\{ \overset{1}{x_1 \stackrel{?}{=} (x_2 \rightarrow x_2)}; \overset{2}{x_2 \stackrel{?}{=} (x_1 \rightarrow x_1)} \} \text{ mepa se re culpa.}$$

Reemp. 1 en 2 $x_2 = (x_2 \rightarrow x_2 \rightarrow x_2 \rightarrow x_2)$

\hookrightarrow El OCCURS-CHECK falla.

Nota: El MGU del primer ejercicio sería lo mínimo indispensable para cumplir la igualdad. En este caso sería $X_1 : Bool \rightarrow Bool$, $X_2 : Bool \rightarrow Bool$

Importante. El algoritmo de simplificación **no siempre** devuelve el mismo resultado si existe una solución.

Algoritmo W (Inferencia de Tipos)

La idea es siempre empezar haciendo las reglas de tipado. La regla de tipado deriva el algoritmo de inferencia. Al igual que el Algoritmo M-M hay dos opciones

- Puede fallar si U no es tipable.
- Puede tener éxito y si lo tiene devuelve una tripla (Γ, M, τ) donde $erase(M) = U$ y $\Gamma \vdash M : \tau$ ¡sí! es lo mismo que vimos al principio sin ningún algoritmo

Se denota $W(U) \rightsquigarrow \Gamma \vdash M : \tau$ para indicar que el algoritmo de inferencia tiene éxito cuando se le pasa U como entrada y devuelve la tripla.

- $\mathbb{W}(x) \rightsquigarrow \{x : X_k\} \vdash x : X_k, \quad X_k \text{ inc6gnita fresca}$
- $\mathbb{W}(\emptyset) \rightsquigarrow \emptyset \vdash \emptyset : Nat$
- $\mathbb{W}(true) \rightsquigarrow \emptyset \vdash true : Bool$
- $\mathbb{W}(false) \rightsquigarrow \emptyset \vdash false : Bool$
- $\mathbb{W}(succ(U)) \rightsquigarrow S(\Gamma) \vdash S(succ(M)) : Nat$ donde
 - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
 - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(pred(U)) \rightsquigarrow S(\Gamma) \vdash S(pred(M)) : Nat$ donde
 - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
 - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(iszero(U)) \rightsquigarrow S(\Gamma) \vdash S(iszero(M)) : Bool$ donde
 - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
 - $S = MGU\{\tau \stackrel{?}{=} Bool\}$
- $\mathbb{W}(if U then V else W) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \cup S(\Gamma_3) \vdash S(if M then P else Q) : S(\sigma)$ donde
 - $\mathbb{W}(U) = \Gamma_1 \vdash M : \rho$
 - $\mathbb{W}(V) = \Gamma_2 \vdash P : \sigma$
 - $\mathbb{W}(W) = \Gamma_3 \vdash Q : \tau$
 - $S = MGU\{\sigma \stackrel{?}{=} \tau, \rho \stackrel{?}{=} Bool\} \cup \{\sigma_1 \stackrel{?}{=} \sigma_2 \mid x : \sigma_1 \in \Gamma_i, x : \sigma_2 \in \Gamma_j, i, j \in \{1, 2, 3\}\}$
- $\mathbb{W}(\lambda x.U) \rightsquigarrow \Gamma' \vdash \lambda x : \tau'.M : \tau' \rightarrow \rho$ donde
 - $\mathbb{W}(U) = \Gamma \vdash M : \rho$
 - $\tau' = \begin{cases} \alpha \text{ si } x : \alpha \in \Gamma \\ X_k \text{ con } X_k \text{ variable fresca en otro caso} \end{cases}$
 - $\Gamma' = \Gamma \ominus \{x\}$
- $\mathbb{W}(UV) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \vdash S(MN) : S(X_k)$ donde
 - $\mathbb{W}(U) = \Gamma_1 \vdash M : \tau$
 - $\mathbb{W}(V) = \Gamma_2 \vdash N : \rho$
 - X_k variable fresca
 - $S = MGU\{\tau \stackrel{?}{=} \rho \rightarrow X_k\} \cup \{\sigma_1 \stackrel{?}{=} \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\}$

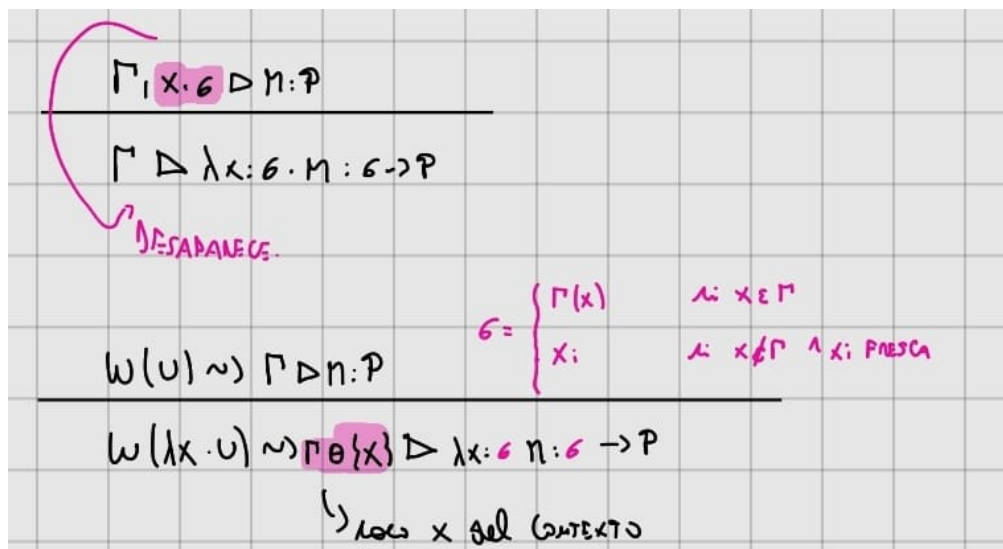
Realmente Importante: Recordemos que en los condicionales o cosas donde hay varios casos, cada tipo debe ser diferente aunque parezca obvio. Las restricciones se colocan luego. N6tese que en la segunda imagen, U_2 y U_3 son de tipos distintos $\mathbb{W}(U_2) = \Gamma_2 \vdash M_2 : \tau_2$ y $\mathbb{W}(U_3) = \Gamma_3 \vdash M_3 : \tau_3$ pero la restricci6n de como son se le da a la hora de hacer la sustituci6n $\{t_2 = t_3\}$

Realmente Importante: Como en los casos condicionales puede ser que una variable aparezca en m6s de una rama, tenemos que verificar que esa variable tenga el mismo tipo en todas las ramas. Esto se puede ver en la segunda imagen marcado en rojo.

Imaginemos que en U_1 , U_2 , y U_3 tengamos apariciones de una misma variable pero con diferente tipo, esto no ser6a v6lido.

Realmente Importante: \emptyset en el Algoritmo de \mathbb{W} representa quitar una variable espec6fica del contexto. Esto se ve f6cil si de

antemano vemos como es el juicio de tipado de la regla.



Incógnitas Frescas

Las Incógnitas Frescas son variables de tipo que se introducen durante el proceso de inferencia para representar tipos que aún no se conocen. Que sea fresca garantiza que es única y no ha sido utilizada antes en el contexto de inferencia actual.

Lógica de Primer Orden (LPO)

Es prácticamente igual a la Lógica Proposicional pero acá se introducen los cuantificadores \forall y \exists . En cuanto a tema de materia, es parecido a deducción natural pero con cuantificadores.

Necesitamos entender la Lógica de Primer Orden porque es en lo que se basa Prolog (Programación Lógica)

Lenguaje de Primer Orden \mathcal{L}

Está formado por

- Conjunto de Símbolos de Función $\mathcal{F} = \{f, g, h, \dots\}$. Cada Símbolo de función tiene asociada una aridad (≥ 0)
 - Los Símbolos de Función con aridad 0 se llaman **constantes**
- Conjunto de Símbolos de Predicado $\mathcal{P} = \{P, Q, R, \dots\}$. Cada Símbolo de predicado tiene asociada una aridad (≥ 0)
- Conjunto Infinito de Numerable de Variables $\mathcal{X} = \{X, Y, Z, \dots\}$

Importante: Los predicados solamente definen una relación entre dos elementos de mi dominio (devuelven un valor de verdad). La idea es enviar como argumento cosas que ya son irreducibles o pueden reducirse hasta poder compararse.

Importante: Las funciones solamente reciben como argumentos elementos de mi dominio y devuelven un elemento de mi dominio. Si se quisiese usar para una fórmula lógica ese valor, debería colocarse un predicado que nos arroje un valor de verdad para ese valor del dominio.

Términos de Primer Orden

Definimos el conjunto \mathcal{T} de términos mediante la siguiente gramática $t ::= X \mid f(t_1, \dots, t_n)$ donde X denota una variable y f un símbolo de función de aridad n .

Importante: Si un símbolo de función, llamémosle g tiene aridad 3, se debe usar enviando los 3 parámetros. Acá no existe la opción de algo ser opcional.

Véase anexo para poder ver con más profundidad la justificación de qué es o no un término.

Diferencia entre Símbolo de Función y Símbolo de Predicado

Un Símbolo de Función me devuelve un elemento de mi Dominio o universo. Los predicados no.

Ej.: n , $succ(n)$ son símbolos de función mientras que \geq es un símbolo de predicado.

Notación Infija, Prefija en LPO

Ej.: $+(0, succ(X)) \equiv 0 + succ(X)$. Nosotros usaremos la notación infija.

$\sigma ::= \mathbf{P}(t_1, \dots, t_n)$	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

Al igual que en Cálculo Lambda, los cuantificadores ligan variables siempre y cuando esté definida en ese ámbito. En este caso, **X** está ligada; **P** denota un símbolo de predicado de aridad n.

Importante: Dos fórmulas que solo difieren en los nombres de las **variables ligadas** se consideran iguales. Esto es por el isomorfismo de las variables ligadas, da igual qué nombre tengan.

Sustitución

Hay que tener cuidado en sustituir en LPO, si vamos a sustituir una variable que depende de otra que está ligada a un cuantificador, antes hay que renombrar la variable ligada en cuantificador.

Ej.: $\sigma := succ(X) = Y \Rightarrow \exists Z. X + Z = Y$

¿Qué es lo que sucede si hacemos $\sigma\{X := Z * Z\}$? Cuando querramos reemplazar en el cuantificador del existe, el existe **ya tiene una Z** pero esta Z que está en el cuantificador es diferente a la que está en nuestro $\{X := Z * Z\}$ por lo tanto hay que renombrar la Z del cuantificador porque de lo contrario, parecerá que son la misma.

El resultado de hacer esta operación quedaría como $\sigma := succ(Z * Z) = Y \Rightarrow \exists Z'. (Z * Z) + Z' = Y$

Importante: Recordar que solamente reemplazamos las ocurrencias libres de una variable, no la que está ligada a un cuantificador.

Deducción Natural extendida a LPO

Todas las reglas de deducción natural proposicional siguen vigentes pero acá se agregan las reglas de introducción y eliminación para el \forall y \exists .

Igual que antes:

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

Axioma	AX		
Conjunción	$\wedge I$	$\wedge E_1$	$\wedge E_2$
Disyunción	$\vee I_1$	$\vee I_2$	$\vee E$
Implicación	$\Rightarrow I$	$\Rightarrow E$	
Negación	$\neg I$	$\neg E$	
Contradicción	$\perp E$		
Lógica clásica	$\neg \neg E$		
Cuantificación universal	$\forall I$	$\forall E$	
Cuantificación existencial	$\exists I$	$\exists E$	

Regla de eliminación

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

Regla de introducción

$$\frac{\Gamma \vdash \sigma \quad X \notin \text{fv}(\Gamma)}{\Gamma \vdash \forall X. \sigma} \forall I$$

Regla de introducción

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

Regla de eliminación

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

Estructuras de Primer Orden

Nos ayuda a decir **cómo interpretamos** los elementos del universo (M, I) . Definimos como **estructura de primer orden** como el par $\mathcal{M} = (M, I)$

- \mathcal{M} : Conjunto **no vacío**, llamado universo.
- \mathcal{I} : Es una función que le da una interpretación a cada símbolo.
- Para cada símbolo de función **f** de aridad n : $I(f) : M^n \rightarrow M$
- Para cada símbolo de predicado **P** de aridad n : $I(P) \subseteq M^n$

¿Por qué es necesario esto? Hasta este momento $\forall X. X$ no nos dice nada. Es solo sintaxis.

Cuando hablamos de 0 podría significar falso en el ámbito o universo de los booleanos, podría significar 0 en los naturales.

Para poder darle todo este significado a cada símbolo tenemos que usar la función de interpretación. La función de interpretación

es re importante, porque algo que está escrito de la misma manera puede ser verdadero o falso según en donde habitemos.

$M := \mathbb{N}$ (los elementos son números naturales)

$$\begin{aligned} I(0) &= 0 \\ I(\text{succ})(n) &= n + 1 \\ I(+)(n, m) &= n + m \\ I(*) &= n \cdot m \end{aligned} \quad \begin{aligned} (n, m) \in I(=) &\iff n = m \\ (n, m) \in I(<) &\iff n < m \end{aligned}$$

Bajo esta estructura, la fórmula $\forall X. X = X + X$ es falsa.

Asignación

Una asignación es una función que a cada variable le asigna un elemento del universo: $a : \mathcal{X} \rightarrow M$

Interpretación de Términos

Cada término $t \in \mathcal{T}$ se interpreta como un elemento, extendiendo la definición de \mathbf{a} a términos: $a(t) \in M$: $a(f(t_1, \dots, t_n)) = I(f)(a(t_1), \dots, a(t_n))$

Interpretación de Fórmulas

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

Definimos una relación de **satisfacción** $\mathbf{a} \models_{\mathcal{M}} \sigma$ "La asignación \mathbf{a} (bajo la estructura \mathcal{M} satisface la fórmula σ)"

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

$$\mathbf{a} \models_{\mathcal{M}} \sigma \clubsuit \tau \quad \text{ sii } \mathbf{a} \models_{\mathcal{M}} \sigma \text{ brócoli } \mathbf{a} \models_{\mathcal{M}} \tau$$

Importante: $a[x \mapsto m]$ representa el mapsTo y está definido de la siguiente forma:

$$a : V \rightarrow M$$

$$a[x \mapsto m] : V \rightarrow M$$

$$a[x \mapsto m](y) = \begin{cases} m & \text{si } x = y \\ a & \text{si } x \neq y \end{cases}$$

Véase **anexo** para ver ejemplos de fórmulas válidas, aplicaciones de predicados y símbolos de función

Decimos que una fórmula σ es:

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

Observaciones

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

Pre-Prolog

Términos de Primer Orden

Prolog opera con ellos. Son de la pinta $X \ Y \ succ(succ(zero)) \ bin(I, R, D) \dots$

Fórmulas Atómicas

En Prolog son de la forma $pred(t_1, \dots, t_n)$

Ej.: $padre(zeus, atenea)$

Programa en Prolog

Es un conjunto de reglas: $\sigma : \neg\tau_1, \dots, \tau_n$.

Cada σ es una regla.

Ej.: $abuelo(X, Y) : \neg padre(X, Z), \text{ padre}(Z, Y)$. En este caso $\sigma := abuelo(X, Y)$, $\tau_1 := padre(X, Z)$ y $\tau_2 := padre(Z, Y)$.

Hechos

Son aquellas reglas en las cuales $n = 0$. Es lo que se considera verdadero o tautológico.

¿A qué nos referimos con $n = 0$? Bueno, que no dependen de nada.

Ej.: $\sigma : \neg padre(zeus, ares)$

Interpretación Lógica de las Reglas

Cada uno de los τ en conjunción deben implicar σ

Es decir, tienen la siguiente interpretación lógica $\forall X_1, \dots, \forall X_k ((\tau \wedge \dots \wedge \tau_n) \implies \sigma)$ donde X_1, \dots, X_k son todas las variables libres de las fórmulas.

Ej.: $\forall X. \forall Y. \forall Z. ((padre(X, Z) \wedge padre(Z, Y)) \implies abuelo(X, Y))$

Consultas

Hablamos de Consultas en Prolog cuando aparece una incógnita X .

Una consulta es de la forma: $? - \sigma_1, \dots, \sigma_n$

Ej.: $? - abuelo(X, ares)$

Interpretación Lógica de las Consultas

El X hay que predicarlo en base a existenciales.

Ej.: $\exists X_1, \dots, \exists X_k. (\sigma_1 \wedge \dots \wedge \sigma_n)$ donde X_1, \dots, X_k son todas las variables libres de las fórmulas.

Claúsula y Literales

Claúsula: $(P \wedge Q)$.

Literal: P

Lógica Proposicional

Pasaje de Logica Proposicional a Forma Clausal

Es un algoritmo.

La entrada es una **fórmula** σ de la **lógica proposicional** y la salida es un booleano que indica si σ es válida.

- Reescribir los \implies : $a \implies b \equiv \neg a \vee b$
- Pasar a Forma Normal (f.n) Negada: Empujar los \neq hacia adentro (si hay).
- Pasar a Forma Normal (f.n) Conjuntiva: Distribuir \vee sobre \wedge (si hay).

Luego, la forma Clausal está formada por conjunciones de disyunciones.

Ej.: $(p \vee q) \wedge (q \vee r)$ en Forma Clausal es $\mathcal{C} = \{\{p, q\}, \{q, r\}\}$. En este caso, esta Forma Clausal está dado por 2 Cláusulas, donde cada Cláusula tiene 2 literales.

Prioridad de Formas Normales en Logica Proposicional

Hay tantas que puede ser un quilombo pero es así: $negada \subseteq conjuntiva$

Refutación (Método de Resolución) en Lógica Proposicional

Llamemos σ a una fórmula de la lógica proposicional cualquiera.

Refutación de \mathcal{C} : Derivación de $\mathcal{C} \vdash \perp$

Si encontramos una refutación de \mathcal{C} :

- Vale $\neg\sigma \vdash \perp$: Es decir, $\neg\sigma$ es insatisfactible o contradicción.
- Luego, vale $\vdash \sigma$. Es decir, σ es tautología.

Si NO encontramos una refutación de \mathcal{C} :

- No vale $\neg\sigma \vdash \perp$. Es decir, σ es satisfactible.
- Luego, no vale $\vdash \sigma$. Es decir, σ no es válida.

Es un algoritmo y tiene varios pasos pero veamos un ejemplo.

Ej.: $(P \wedge Q) \implies P$ ¿Cuáles son tautologías? Si nos está pidiendo cuales son tautologías tenemos que ver que $\neg\sigma$ sea insatisfactible o contradicción, si esto sucede entonces σ es tautología.

Lo primero que hacemos es pasar la fórmula $\neg\sigma$ a Forma Clausal, por el algoritmo descripto anteriormente nos queda

- $\neg((P \wedge Q) \implies P)$
- Eliminación del \implies : $\neg(\neg(P \wedge Q) \vee P)$
- Empujar \neg interno: $\neg(\neg P \vee \neg Q \vee P)$
- Empujar \neg externo: $P \wedge Q \wedge \neg P$

Luego, la forma Clausal es $\mathcal{C} = \{\{P\}, \{Q\}, \{\neg P\}\}$

Ahora básicamente tenemos que ir eligiendo cláusulas k y k' hasta que lleguemos al vacío.

Eligiendo $k = \{P\}$ y $k' = \{\neg P\}$ nos da \emptyset . Por lo tanto, se agrega \emptyset a \mathcal{C} pero en el paso recursivo, como \emptyset pertenece a \mathcal{C} llegamos a que $\neg\sigma$ es insatisfactible. Luego, σ es tautología.

Importante: Es útil ver que para llegar a vacío no siempre es necesario usar todas las cláusulas. Inclusive, va a haber veces que jamás se llegue a insatisfactible. Entonces eso indicaría que si vale $\neg\sigma$ entonces σ es inválida.

Importante: Este proceso siempre termina.

Lógica de Primer Orden (LPO)

Pasaje de LPO a Forma Clausal

Es prácticamente un algoritmo. Es parecido al anterior en los primeros 2 pasos. La diferencia es que acá si σ no es válida, el método puede no terminar.

En la teórica les ponen los pasos de una, pero a mí no me gustó como lo ordenaron. Primero lean lo de abajo, y va a haber una sección que se llama igual que esta pero con un (2).

Resolución para Lógica de Primer Orden

Hay que pasar por varias formas hasta llegar a la forma Clausal. **Importante:** Lo anoto en todos lados porque siempre me olvido. No por algo nos hacen detectar siempre las variables ligadas y libres. Si tenés más de un cuantificador con la misma variable, entonces renombrá algún cuantificador y reemplazá su ocurrencia por la nueva variable, porque después es un quilombo.

Pasaje a forma clausal en Lógica de Primer Orden

Skolemización

La Skolemización es deshacerse de los cuantificadores existenciales. La idea es introducir testigos para los \exists sin cambiar la satisfactibilidad.

- Todo cuantificador existencial se instancia en una constante o función de skolem.
 - Utilizamos una función cuando el cuantificador existencial está dentro al alcance de un cuantificador para todo (el más cercano).
 - Utilizamos simplemente c cuando el cuantificador existencial no está al alcance de otro cuantificador.

Importante: La Skolemización **no es determinística**.

Importante 2: Skolemizar de afuera hacia adentro.

Importante 3: La Skolemización **preserva la satisfactibilidad** pero **no produce fórmulas equivalentes**, esto quiere decir que no preserva la validez. Ej.: $\exists X.(P(0) \implies P(X))$ es válida mientras que $P(0) \implies P(c)$ es inválida.

Ejemplo de Skolemización con existenciales sin alcance de universales

$$\begin{aligned} & \exists x. \exists y. x < y \\ & \quad \exists y. c < y \\ & \quad \quad c < d \\ & C = \{ \{ c, d \} \} \end{aligned}$$

Ejemplo de Skolemización con existenciales al alcance de universales

DEPENDIE DE X, $\forall x \exists y$ ENCAPSULO X.

$$\begin{aligned} & \forall x. \exists y. x < y \\ & \forall x. x < f(x) \\ & C = \{ \{ x < f(x) \} \} \\ & f \text{ espera } x \text{ y devuelve los posibles } y \text{ de quien} \end{aligned}$$

Forma Normal de Skolem (NNF)

Sea A una sentencia rectificadora en FNN. Una fórmula está rectificadora si todos sus cuantificadores ligan variables distintas entre sí, y a la vez distintas de todas las variables libres.

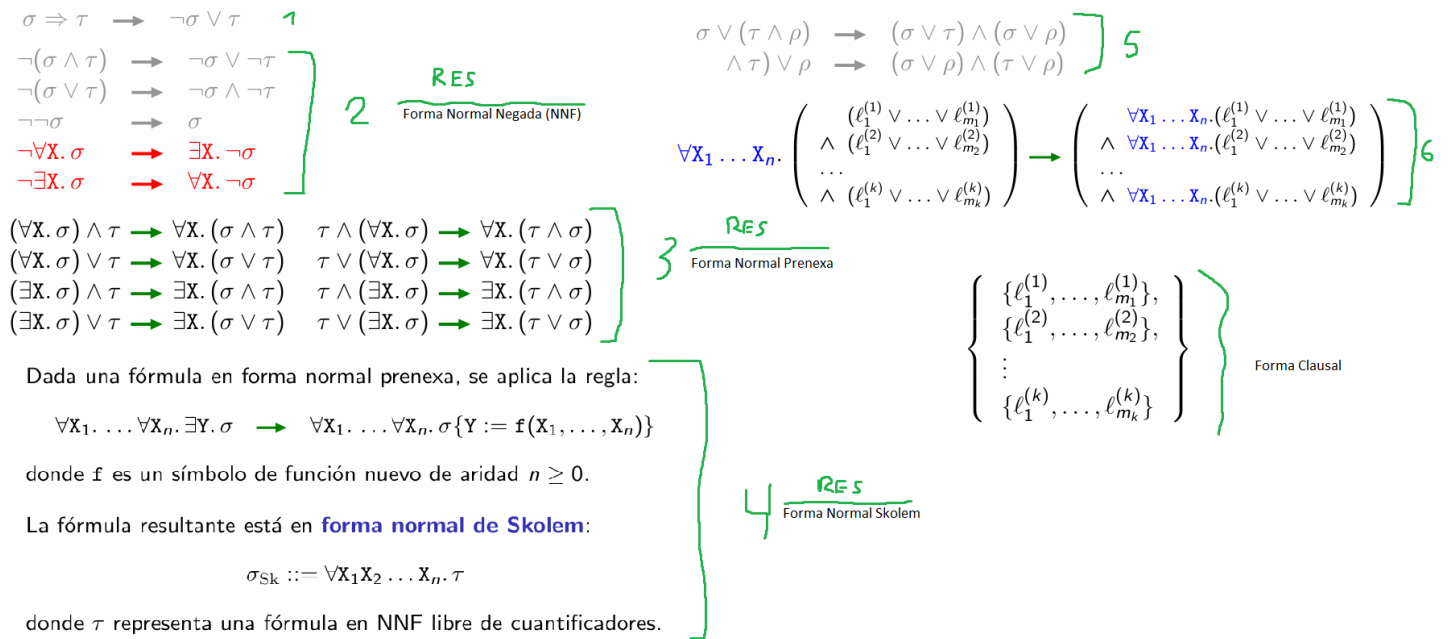
En criollo: Los cuantificadores tienen variables con nombres únicos y no se pisan con variables libres.

Una fórmula en forma normal de Skolem tiene la siguiente pinta: $\sigma_{SK} ::= \forall X_1 X_2 \dots X_n * \tau$

Pasaje de LPO a Forma Clausal (2)

- Reescribimos \Rightarrow usando \neg y \vee .
- Pasar a f.n negada, empujando \neg hacia adentro.
- Pasar a f.n extrayendo \forall y \exists hacia afuera.
- Pasar a f.n de Skolem, Skolemizando los existenciales.
- Pasar a f.n. conjuntiva, distribuyendo \vee sobre \wedge .
- Empujar los cuantificadores hacia adentro de las conjunciones.

Acá dejo una imagen que hice por si es de utilidad, con las propiedades y todo (res: es lo que te devuelve al aplicar ese paso)



Véase [anexo](#) para dos ejemplos

Prioridad de Formas Normales en LPO

Hay tantas que puede ser un quilombo pero es así: $negada \subseteq prenexa \subseteq skolem \subseteq conjuntiva$

Refutación (Método de Resolución) en LPO

Es exactamente igual que en Lógica Proposicional **excepto** que ahora como existen funciones P que tienen argumentos, hay que ver si los argumentos son unificables.

Hasta ahora, esto vale $\forall X. P(X) \wedge \neg P(0)$, esto es porque las cláusulas que tenemos son $\mathcal{C} = \{\{P(X)\}, \{\neg P(0)\}\}$ y la regla propuesta no aplica porque las P son iguales pero sus argumentos no.

Ej.: Sea $\neg \sigma$ con sus cláusulas, $\mathcal{C} = \{\{P(X, Y)\}, \{\neg P(f(z), Z)\}\}$

- Para cancelar los términos opuestos P y $\neg P$, unificamos.
- $mgu(P(X, Y) \stackrel{?}{=} P(f(Z), Z))$. Para que estos sean iguales en argumento vemos que unificamos como $\{X := f(z), Y := Z\}$.
- Luego, $k = P(f(z), Z)$ y $k' = \neg P(f(z), Z)$ nos da como resultado $\{\}$ o \emptyset .
- Por lo tanto como $\neg \sigma$ es insatisfactible, σ es tautología.

Importante: Primer paso de todo ejercicio: si comenzamos teniendo implicaciones, o cuantificadores universales existenciales con las mismas variables, recordar renombrarlas.

Importante2: (A nivel MGU) Si hay un cuantificador existencial que ahora depende un universal, primero hay que chequear en qué términos se usa la variable **ligada al cuantificador universal**. Esto es, porque cuando eliminemos el **cuantificador existencial**, la existencial se elimina y será del tipo **f(variableCuantUniversal)** pero como variableCuantUniversal estaba en otro término (no ligado al existencial) al intentar unificar nos dará occurs-check pues variableCuantUniversal NO unifica con

$f(\text{variableCuantUniversal})$. La idea justamente acá, es que como el existencial asegura que existe (puede no existir), entonces no podemos confundir con la ligada al para todo porque esa sí vale siempre. Igual, dejo un ejemplo acá abajo.

Handwritten logical derivation on a grid background:

$$\neg [(\exists x. \forall y. R(x, y)) \Rightarrow \forall y. \exists x. R(x, y)]$$

$$\neg [\neg (\exists x. \forall y. R(x, y)) \vee \forall y. \exists x. R(x, y)]$$

$$\neg \neg (\exists x. \forall y. R(x, y)) \wedge \neg (\forall y. \exists x. R(x, y))$$

$$\exists x. \forall y. R(x, y) \wedge \exists y. \neg \exists x. R(x, y)$$

$$[(\exists x. \forall y. R(x, y)) \wedge (\exists y. \forall x. \neg R(x, y))] \rightarrow \text{Removiendo nombres repetidos}$$

$$\exists x. ((\forall y. R(x, y)) \wedge (\exists w. \forall t. \neg R(t, w)))$$

$$\exists x. \forall y (R(x, y) \wedge (\exists w. \forall t. \neg R(t, w)))$$

$$\exists x. \forall y. \exists w. \forall t (\neg R(t, w) \wedge R(x, y))$$

$$\exists x. \forall y. \forall t (\neg R(t, f(y)) \wedge R(x, y))$$

$$\forall y. \forall t (\neg R(t, f(y)) \wedge R(c, y))$$

$$\forall y. \forall t. \neg R(t, f(y)) \wedge \forall y. \forall t. R(c, y)$$

Annotations on the right side:

- * IMPORTANTE: POR SKOLEM, TODAS LAS OCURRENCIAS DE W SEAN LAS MISMAS POR $f(y)$ PERO OJO PORQUE •
- SIGUE DEPENDIENDO DE y ($\forall y$ ORIGINAL)
- CON ESTO DIGO QUE $f(y) = y_2$ y (NOTA $\exists w$ en $\forall y$)
- NO ES CIERTO QUE $f(y) = y$ $\forall y$.
- ACHAHO XQ $f(y) = y \xrightarrow{\text{SUMA}} y = f(y)$ en OCCURS-CHECK

OJO. Esto es SOLO a nivel MGU. Con esto quiero decir que si vas a unificar, le cambies el nombre a las variables.

Prolog

El algoritmo que vimos antes para ver si una fórmula de primer orden σ es válida funciona, pero es costosa. Esto es porque a nivel computacional implica hacer

- Búsqueda: Elegir dos cláusulas
- Selección: Elegir un subconjunto de literales de cada cláusula

Además en cada paso se agrega una cláusula, resolverse ecuaciones de unificación y se usa BFS.

¿Cómo soluciona este problema Prolog? Usando **resolución SLD**.

Resolución SLD

La resolución SLD se aplica solamente sobre cláusulas de Horn.

Cláusulas de Horn

Recordemos que una cláusula es un conjunto de literales

$$\{l_1, \dots, l_n\}$$

Los literales son de la forma

$$l ::= P(t_1, \dots, t_n) \mid \neg P(t_1, \dots, t_n)$$

En SLD cada cláusula tiene como máximo un literal positivo.

Por ejemplo, en SLD no es posible resolver esto: $\mathcal{C} = \{ \{P(X), Q(Y)\}, \{\neg Q(Y)\} \}$ porque la primera cláusula tiene dos positivos.

Grupos en Cláusulas de Horn

- Las que tienen un literal positivo y ningún negativo se llaman hechos o axiomas.
 - En LPO: $\{P(X)\}$
 - padre(m, t): Es un axioma que dice que m es padre de t.
 - madre(k, t): Es un axioma que dice que k es madre de t.
- Las que tienen un literal positivo y una/varias negativas se llaman reglas.
 - En LPO: $\{P(X), \neg Q(X, Y)\}$
 - En Prolog: padres(x) :- madre(m, x), padre(p, x).
 - Esta regla define que x tiene 'padres' cuando existe una madre m de x y un padre p de x.
- Las que NO tienen literal positivo y una/varias negativas se llaman cláusulas objetivo.
 - En LPO: $\{\neg P(X)\}$
 - padres(t).
 - Cuando esto le llega a Prolog, se conforma una cláusula objetivo que se compone de literales negativos.

Nota: Los axiomas + reglas en Prolog se llaman **cláusulas de definición (PGM)** y estas nunca llegan a ser insatisfactibles.

Derivación SLD

En cada paso, Prolog siempre toma una **cláusula de definición (PGM)** y una **cláusula objetivo (Goal)**

Programación Orientada a Objetos (POO)

No subieron la teórica así que voy a tener que escribir en base a lo que anoté yo y me pareció interesante. Vamos a usar SmallTalk donde todo es prácticamente un objeto.

En SmallTalk los objetos se comunican entre sí a través de mensajes, donde cada mensaje es una invocación a un método de un objeto.

Como en el Paradigma Orientado a Objetos podemos mutar instancias de objetos determinados a través de métodos (getters/setters), necesitamos tener un estado. Al principio de la materia vimos que el estado no es más que el conjunto o estado de variables en un momento dado.

Mensajes

Siempre están asociados a un objeto.

Métodos de Instancia y Métodos de Clase

Los métodos no son más que la implementación de la respuesta a un mensaje dado.

```
1 | Object subclass: Persona [  
2 |     Persona class >> hola [  
3 |         ^'hola, ¿como estás'  
4 |     ]  
5 |  
6 |     Persona class >> crearNuevaPersona [  
7 |         ^Persona new.  
8 |     ]  
9 | ]  
10 |  
11 | obj := Persona new.  
12 | obj hola.
```

En este caso el mensaje es hola (invocar al método de instancia hola), y la respuesta 'hola, como estás' la devuelve el método hola de la clase persona.

La diferencia entre un método de instancia y método de clase es la siguiente

- Método de Instancia: Son métodos que se pueden ejecutar **a través de la instancia de una clase**.
 - Ej.: obj hola.
- Método de Clase: Se ejecutan en la clase misma, y nos pueden ayudar a la creación de nuevas instancias.
 - Ej.: obj := Persona crearNuevaPersona.

Objeto Receptor

Es el objeto que recibe el mensaje. Puede recibirlo tanto una clase a través de un método de clase, o una instancia de la clase a través de un método de instancia.

Objeto Colaborador

Es un objeto con el que el receptor interactúa o colabora para lograr una tarea o propósito. Es un parámetro. Ej.: 10 mcm: 4, 4 actúa como colaborador.

Return

En el código, el return se puede notar como \wedge *variable*

Uso de Variables en Clases

No nos referimos a ella como this, como si hacemos en muchos lenguajes, sino que acá, en SmallTalk simplemente mencionamos la variable.

POO y Clean Code

Jamás, bajo ningún termino agregamos campos a una clase por facilidad de cómputo. Las variables que representan a una clase deben ser la menor cantidad, y deben estar pensadas para el objetivo que tenga ese objeto de resolver. Esto es más que nada porque, un objeto debería ser lo más chico posible además de que si agregamos campos extra, hay que mantenerlos y eso da un costo mayor.

Tipos de Expresiones en SmallTalk

- Variables Locales: Se escriben en minúscula
 - $x:=3$. $y:=2$.
- Variables Globales: Se escriben en mayúscula (no obligatorio). Se definen a nivel de clase, y son accesibles desde cualquier parte del programa a través de la clase a la que pertenecen.
 - classVariableNames: 'contador'
- Mensajes Unarios: Toman un único parámetro. Se usan de manera prefija.
 - 10 print. Imprime el valor 10.
 - 5 isOdd. Devuelve true.
 - -1. Devuelve el número -1, el (-) es un mensaje unario.
- Mensajes Binarios: Toman dos parámetros. Se usan de manera infija.
 - $3 + 5$.
 - $10 * 2$.
 - true and: false.
- Mensajes Keyword: Pueden tener múltiples argumentos, con cada argumento despues de su palabra clave correspondiente.
 - Rectangulo new ancho: 5 alto:10. Los keywords son ancho y alto.
- Asignación
 - persona := Persona new

La prioridad, dentro de una expresión es la siguiente: *unarios* > *binarios* > *keyword*

Veamos un ejemplo donde tiene importancia esto: Sea una clase de un entero, implemente un método de instancia que permita saber cual es el mcm del número instanciado y otro que viene por parámetro.

```
1 | mcm: numberTwo
2 | res |
3 | res := self * b // self gcd: b
4 | ^res
```

Como está implementado este código va a fallar, porque primero va a hacer `self * b`, e inmediatamente va a querer dividir por algo que todavía no se resolvió.

```
1 | mcm: numberTwo
2 | | res |
3 | res := self * b // (self gcd: b)
4 | ^res
```

Importante: Nótese que todos los métodos los estamos haciendo dentro de una clase. Esto quiere decir que siempre tenemos que asumir que estamos dentro de un objeto.

Si tuviésemos que notar qué mensajes, receptores, colaboradores, resultados hay para la siguiente entrada: `6 mcm 10`

- Mensaje: `mcm` — Receptor: `6` — Colaboradores: `10` — Resultado: `30`
- Mensaje: `*` — Receptor: `Self (6)` — Colaboradores: `10` — Resultado: `60`
- Mensaje: `gcd` — Receptor: `Self (6)` — Colaboradores: `10` — Resultado: `2`
- Mensaje: `//` — Receptor: `60` — Colaboradores: `2` — Resultado: `30`

Nótese que básicamente fuimos paso a paso viendo qué instrucciones estaban y quienes estaban involucrados.

Palabras Reservadas en SmallTalk

- `true`
- `false`
- `self`
- `super`
- `nil`
- `thisContext`

Algunos ejemplos de instancias de objetos

- `nil`: Es la instancia de `UndefinedObject`
- `true`: Es la instancia de la clase `True`
- `false`: Es la instancia de la clase `False`
- Constantes Numéricas: `1`, `12`, `125`... son números, instancias de clases como `SmallInteger` o `LargeInteger`.
- Strings: `'hola'` es la instancia de la clase `String`
- Símbolos: `#Rectángulo`. Instancia de la clase `Symbol`. Un Símbolo es un objeto que representa un identificador único e inmutable.
- Caracteres: `$a` es la instancia de la clase `Character`. Representa un único caracter.

Herencia

Importante: En la materia no se lo menciona, pero siempre es mejor priorizar la composición antes que la herencia.

La Herencia trae muchos problemas a nivel de objetos porque se vuelve inmantenible si la clase padre tiene muchos atributos, y esto es importante, porque una clase no debería estar abierta a eliminar cosas núcleo pero sí abierta a extenderla.

Esto es porque si refactorizamos la clase padre, si hay mas de una clase que heredaban esta, vamos a tener que considerar modificar andá a saber cuantas más clases.

El concepto de Herencia es importante, porque cuando hablamos de una relación padre-hijo, si desde hijo queremos llenar campos que están en el padre, usamos la palabra `super`, mientras que si queremos hablar de nuestro ámbito local de objeto usamos `self`.

Voy a mostrar un ejemplo en un lenguaje que conozca bien (TypeScript) para que después puedan relacionarlo en cualquier lenguaje.

```
1 | class ContactoBasico {
2 |     msg: string;
3 |
4 |     constructor(data){
5 |         this.msg = data.msg;
6 |     }
7 | }
```



```

8
9 class ContactoEmail extends ContactoBasico {
10     email: string;
11
12     constructor(data){
13         super(data);
14         this.email = data.email;
15     }
16
17     hacerAlgo(){
18         console.log('Email: ${this.email}, Mensaje: ${this.msg}');
19     }
20 }

```

En el ejemplo anterior, podemos ver que ContactoEmail puede almacenar un valor en msg como si fuese una variable de clase de sí misma, pero en realidad, esta puede ser utilizada a nivel ContactoEmail porque la estamos llenando vía super.

Self

Cuando hablamos de Self, estamos hablando de utilizar las variables o métodos de la instancia del objeto que estamos manipulando.

El Self siempre busca hacia abajo.

Super

Cuando hablamos de Super, estamos hablando de herencia. Esto quiere decir que si una clase habla de super, está extendiendo de otra. Ojo, super habla de la clase inmediatamente que está por encima.

Esto es súper importante, porque hay muchos lenguajes (y gracias a dios) que no aceptan herencia múltiple.

```

1 class ContactoBasico {
2     msg: string;
3
4     constructor(data){
5         this.msg = data.msg;
6     }
7
8     hacerAlgo(){
9         console.log("Hola");
10    }
11 }
12
13 class ContactoEmail extends ContactoBasico {
14     email: string;
15
16     constructor(data){
17         super(data);
18         this.email = data.email;
19     }
20
21     hacerAlgo(){
22         console.log('Email: ${this.email}, Mensaje: ${this.msg}');
23     }
24 }
25
26 class ContactoEmailMasComplicado extends ContactoEmail {
27     emailSecundario: string;
28
29     constructor(data){
30         super(data);
31         this.emailSecundario = data.emailSecundario;
32     }
33
34     hacerAlgo(){
35         return super.hacerAlgo();

```

```

36 |         }
37 |
38 |     }

```

Ojo. En este ejemplo, si queremos instanciar `ContactoEmailMasComplicado` tendremos que cumplir la precondition de `super`, es decir, el constructor de `ContactoEmail`, y a su vez el de `ContactoBásico`. Esto es porque es una cadena. Ya se dan cuenta del por qué es malo laburar con herencia.

¿Qué es lo que sucede, si una vez instanciada `ContactoEmailMasComplicado` en `obj`, llamamos `obj.hacerAlgo?` ¿Qué `hacerAlgo` ejecuta?, bueno, el padre inmediato. Ejecutaría `console.log('Email: this.email, Mensaje :this.msg')`, es decir, el `hacerAlgo()` que está en `ContactoEmail`, que es padre de `ContactoEmailMasComplicado`.

Importancia del Return

En los métodos de una clase, si no tenemos un `return`, entonces el método retornará **self**. Es decir, la instancia.

Métodos de Clase

Los métodos de clase nos permiten crear instancias de una clase sin utilizar desde fuera de la clase la palabra reservada **new**. Esto nos permite que la clase por sí misma, diga qué es lo que podemos usar para crear instancias de ella.

```

1 |     Object subclass: #Rectangulo
2 |         instanceVariableNames: 'ancho_alto'
3 |         classVariableNames: ''
4 |
5 | Rectangulo class >> ancho: ancho alto: alto [
6 |     ^self new
7 |         ancho: ancho;
8 |         alto: alto;
9 |         yourself.
10 | ]
11 |
12 | Rectangulo >> ancho: valor [
13 |     ancho := valor
14 | ]
15 |
16 | Rectangulo >> alto: valor [
17 |     alto := valor
18 | ]
19 |
20 | Rectangulo >> descripcion [
21 |     ^'Rectángulo de ancho', ancho printString, 'y alto', alto printString
22 | ]
23 |
24 | Uso:
25 | rect := Rectangulo ancho: 30 alto: 20.
26 | rect descripcion.

```

Importante notar, que cuando hacemos `^self new ancho : ancho`; estamos creando una nueva instancia, mandando un mensaje al método `ancho` con el parámetro `ancho`.

Importante notar también, que cuando hacemos `self new`, estamos enviando un mensaje de `new` con varios argumentos. Estos argumentos los separamos por ; porque son parte del mismo mensaje.

Sobrecarga

Decimos que un método de clase está sobrecargado cuando existe más de un método con el mismo nombre pero que toman diferentes parámetros (ya sea en cantidad o en tipos, lo importante es que no sean la misma cantidad de parámetros en el mismo orden y mismo tipo). Es importante notar que los métodos de clase más restrictivos deberían poder estar encapsulados en el más general.

```

1 | object subclass #robot
2 |     instance variablenames: 'posicion'
3 |
4 |     inicializar: unaposicion
5 |         posicion := unaposicion
6 |

```

```

7 |         posicion:
8 |             ^posicion
9 |
10 |         inicializar:
11 |             self inicializar: (0 @ 0)

```

Sintaxis de SmallTalk

- Usá . para indicar donde termina una instrucción. Es similar al ; de C++ pero acá, si hay solamente una línea el . no es necesario.
 - Ej.: x := 10. y := 20.
- Usá ; cuando tengas que encadenar mensajes a un mismo objeto, sin finalizar la ejecución de la expresión anterior.
 - Ej.: Transcript show: 'Hola'; cr; show: 'Mundo'

Bloques (Block-Closures)

Permite encapsular lógica dentro de un código independiente y reutilizable. Está compuesto por una secuencia de instrucciones que se agrupan y pueden ser pasadas como argumentos a métodos o invocadas como acción dentro de un método.

La sintaxis está compuesta por :variable + expresión

Si solo hay que definirlo, lo terminamos con .

```

1 | [:x | x * 2]

```

Si hay que pasarle argumentos, lo invocamos con :

```

1 | resultado := (1 to: 3) collect: [:x | x * 2].
2 | resultado tiene [2, 4, 6]

```

Si hay que obtener el resultado de un bloque, colocamos value antes del ..

El value invoca al bloque. Un value vacío es llamar al método sin argumentos.

```

1 | resultado := [1 + 2 + 3] value.
2 | resultado tiene el valor de 6

```

Si hay que usarlo en condicionales, cada rama podría ser un bloque

```

1 | resultado := 10 > 5
2 |     ifTrue: ['es_mayor']
3 |     ifFalse: ['es_menor'].

```

Si hay que enviarle argumentos a los bloques, los enviamos a través de value, es decir, invocamos al bloque con tantos argumentos instanciados.

```

1 | resultado := [:x :y | x+y ] value: 3 value: 4.

```

Si querés usar variables locales en los bloques, la sintaxis es diferente

```

1 | resultado := [ | x y | x:= 3. y:=2. x+y. ]

```

Importante: Si un bloque está definido para n argumentos, solo se podrá invocar ese bloque enviando los n argumentos.

```

1 | resultado := [:x :y | x+1] value: 1. falla
2 | resultado := [:x :y | x+1] value: 1 value: 3. funciona

```

Para invocar bloques anidados, simplemente usamos value reiteradas veces pero indicando la cantidad de argumentos.

```

1 | [ | z | z:=10 . [:x | x+z]] value value: 10.
2 | Invoca al bloque padre, y como el segundo bloque espera un argumento obligatorio le enviamos el valor de 10.

```

Importante: **NUNCA** usar un return dentro de un bloque. Es algo bastante oscuro.

Importante: Los bloques devuelven como resultado la última expresión del bloque.

Bloques = Block Closures

¿Qué relación tienen los bloques con los closures? Cuando guardamos un método que contiene un bloque, y a su vez, el bloque utiliza una variable de ese método, estamos encapsulando a este bloque en un universo donde las variables que se usen van a ser inmutables, a menos que ejecutemos este bloque reiteradas veces.

Veamos un ejemplo

```
1 | A m1
2 |   | x y |
3 |   y := 0
4 |   x := [ y := y+1 ].
5 |   ^x
6 |
7 | B m2
8 |   | a aBlock anotherBlock |
9 |   a := A new.
10 |  aBlock := a m1.
11 |  aBlock value.
12 |  aBlock value.
13 |  anotherBlock := a m1.
14 |  anotherBlock value.
15 |  ^aBlock value.
```

Prestemos atención a A por un momento, A es un método que básicamente define dos variables locales **x**, **y** donde **y** es inicializada con 0, y **x** es inicializada con un bloque que depende de **y**. Al finalizar el método, devuelve el bloque.

Ahora vayamos a B m2. La variable **a** almacenará la instancia de A creada (para poder usar el método de instancia m1). Luego que hace esto hay dos partes, una variable llamada aBlock, donde aBlock almacenará el block-closure [y := y+1].

Luego, la instrucción aBlock value ejecuta el closure esperando un valor, y hasta ahora, el valor que arroja es 1 (Nótese que arroja 1 porque es 0 := 0+1)

Luego, la instrucción aBlock value, ejecuta el closure esperando un valor, y hasta ahora, el valor que arroja es 2 (Nótese que arroja 2 porque en la llamada anterior, y valía 1, entonces 1 := 1+1 = 2)

Lo interesante viene ahora, anotherBlock := a m1. almacena el bloque de m1 pero como una instancia NUEVA. Es decir, es otro bloque INDEPENDIENTE.

Por lo tanto cuando hagamos anotherBlock value. arrojará y = 1.

Finalmente, aBlock value sumará 1 al primer bloque, y retornará su valor, es decir, y = 3. **Conclusión:** Ninguno de los bloques sabe que están instanciados para variables diferentes, y cada bloque recuerda qué valor tomaba la variable y en cada instrucción.

¿Cuál es el mensaje, cual el receptor y cual sus colaboradores? bueno, el mensaje sería m1, el receptor sería el propio objeto a y no hay ningún colaborador.

Excepciones

Si mandamos un mensaje a un método que no existe en la clase, se tratará de buscar en cada uno de los super recursivamente. El último intento lo hará en el padre de todos los objetos, es decir, la clase Object. Si no existe ahí, se ejecutará una excepción (does not understand).

Colecciones en SmallTalk

Se indexan desde el 1 en adelante. Les querían caer mal a los que indexamos desde 0.

Algunas de las colecciones mas conocidas son

- Bag (Multiconjunto)
- Set (Conjunto)
- Array (Arreglo): La cantidad de elementos es fija.
- OrderedCollection (Lista)
- SortedCollection (Lista ordenada)
- Dictionary (Hash)

El mensaje que hay que utilizar para crear estas colecciones es **with**.

Hay diferentes maneras (equivalentes) de crear una colección y agregar sus elementos

- Bag with: 1 with: 2 with: 4
- # (1 2 4) = (Array with: 1 with: 2 with: 4) →. Ojo, el # es un símbolo, por lo tanto sería una lista inmutable.

- Bag withAll: #(1 2 4)

Algunos mensajes que aceptan las 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.
- reject: la negación del select.
- collect: devuelve una colección que es resultado de aplicarle un bloque a cada elemento de la colección original.
- detect: devuelve el primer elemento que cumple un predicado.
- reduce: toma un bloque de dos o más parametros de entrada y hace fold de los elementos de izquierda a derecha.

Anexo

Prolog

Un programa en Prolog está compuesto por

- Hechos: no poseen un cuerpo. Son verdaderos siempre.
- Reglas: tambien llamados predicados. El lado izquierdo de una regla es conocido como HEAD mientras que el lado derecho, o las condiciones que deben de cumplirse para que valga el HEAD se llama BODY.
- Consultas

Los Hechos y las Reglas conforman una database o cláusulas.

La idea de Prolog consiste en realizar consultas, es decir, preguntar cosas acerca de la información que tenemos almacenada.

Hechos

Comienzan con minúscula siempre, y están en lowercase.

```
1 | woman(mia).
2 | woman(judy).
3 | playsAirGuitar(judy).
4 | party.
```

En este caso tenemos 4 hechos, que sin importar la situación que estemos, siempre serán verdad.

Si hacemos la siguiente consulta: ?- **woman(mia)** la respuesta será true.

Si hacemos la siguiente consulta ?- **woman(guada)** la respuesta será false, pues la database no tiene información acerca de que guada sea una mujer. Esto es súper importante en la Programación Lógica. Que sea falso significa que estamos en un mundo cerrado, por lo tanto, lo que no son hechos ni reglas, **siempre es falso**.

Reglas

Las reglas están conformadas por hechos en su cuerpo. El lado izquierdo de una regla es llamado HEAD mientras que el lado derecho de una regla es llamado BODY.

```
1 | happy(yolanda).
2 | listen2Music(mia).
3 | listen2Music(yolanda).
4 | playsAirGuitar(mia) :- listens2Music(mia).
```

En esta database tenemos 3 hechos y 1 regla.

Las reglas se leen como: **playsAirGuitar(mia) es verdadero si listens2Music(mia) es verdadero**. Esto es importante, porque lo que quiere decir es que cada una de las condiciones del BODY implican el HEAD.

¿Cómo es que Prolog deduce el resultado de **playsAirGuitar(mia)**? Lo primero que hace es buscar esta ecuación como un hecho. Como NO es un hecho, entonces se fija qué condiciones deben cumplirse para que esto sea verdadero (por defecto, es falso). Como observa que depende de listens2Music(mia), observa si esto es un hecho. Como efectivamente es un hecho, concluye que esto es verdadero.

Reglas ∨

Las Reglas ∨ se caracterizan por ser verdaderas si las condiciones no necesariamente se deben cumplir al mismo tiempo. Para poder definir un ∨ en Prolog, usamos ;.

```
1 | playsAirGuitar(butch) :- happy(butch); listens2Music(butch).
```

playsAirGuitar(butch) será verdadero sí y solo si happy(butch) es verdadero o listens2Music(butch) es verdadero.

Variables

Las variables en Prolog no son como en un lenguaje imperativo que significan un nombre para un valor dado. Acá las variables se unifican a valores y estas comienzan con una letra mayúscula.

Esto quiere decir que, Prolog por cada vez que trata de devolver una respuesta, en realidad está devolviendo una unificación para la cual una condición es cierta (hecho)

```
1 | woman(mia).
2 | woman(judy).
3 | woman(yolanda).
4 | loves(vincent, mia).
5 | loves(marsellus, mia).
6 | loves(pumpkin, honey_bunny).
7 | loves(honey_bunny, pumpkin).
```

De la siguiente database, podemos hacer consultas como: **woman(X)**, ¿qué es lo que estamos preguntando? **dame todas las posibilidades para X donde X cumpla woman**.

Esto se resuelve de la misma manera en que nosotros hacíamos el Algoritmo de Martinelli-Montanari, quiero decir que:

- $\{woman(X) \stackrel{?}{=} woman(mia)\}$
 - Dec $\rightarrow \{X \stackrel{?}{=} mia\}$
 - Del $\rightarrow \emptyset$
 - Luego, X=mia.

Esto mismo sucede para woman(judy) y woman(yolanda).

¿Qué pasa con loves?, bueno, falla por clash porque los predicados no son el mismo, ni tampoco necesitan la misma cantidad de argumentos.

- $\{woman(X) \stackrel{?}{=} loves(vincent, mia)\}$
 - \rightarrow clash.

Importante: Recordar que una variable unifica con cualquier cosa.

Veamos ahora un último ejemplo, ¿qué pasa con loves(marsellus, X)? Bueno, buscará en la database los hechos o reglas que hagan match con esta consulta, viendo qué posibles valores puede tomar X para ser verdadero.

Desde ya, no va a unificar nunca con woman porque falla por clash.

Tampoco unifica con loves(vincent, mia) por el mismo motivo

- $\{loves(marsellus, X) \stackrel{?}{=} loves(vincent, mia)\}$
 - Dec $\rightarrow \{marsellus \stackrel{?}{=} vincent\}, \{X \stackrel{?}{=} mia\}$
 - \rightarrow clash entre marsellus y vincent, son dos funciones de aridad 0 y son diferentes.

Términos

En Prolog, los términos están conformados por: átomos, números, variables y términos complejos.

- Los átomos son aquellos que comienzan con minúscula, están conformados por letras minúsculas, letras mayúsculas, dígitos y el símbolo de `_`. Ej.: `listens2Music`, `playsAirGuitar`, `'vincent'`.
- Los números son como conocemos en todos los lenguajes de programación, números.
- Las variables empiezan con mayúscula, están conformadas por mayúsculas, minúsculas, números y `_` y usamos `_` para hablar de una variable anónima.
- Los términos complejos o también conocidos como estructuras, están conformados por una palabra y una secuencia de argumentos. Ya vimos anteriormente algunos, por ejemplo `playsAirGuitar(jody)` y se nota `playsAirGuitar/1` pues requiere de un argumento.

Importante: Cuando decimos que `playsAirGuitar` requiere de un solo argumento, decimos que es de aridad 1 o función unaria. Esto de la aridad es útil entenderlo porque Prolog nos permite tener un mismo nombre de predicado, pero con diferente cantidad de argumentos. Esto, a bajo nivel lo trata como si fuesen dos predicados totalmente diferentes.

De ahí viene que esto es válido:

```
1 | loves(tom, guada).
2 | loves(tom, kari).
3 | loves(kari, tom, andy).
```

¿Cuándo usar variables anónimas, y cuando no?

Veamos el siguiente ejemplo

```
1 | gives_footmassage(X, mia).
2 | kills(marsellus, Y) :- gives_footmassage(Y, mia).
3 |
4 | gives_footmassage2(_, mia).
5 | kills2(marsellus, Y) :- gives_footmassage2(Y, mia).
```

¿Cual es la diferencia entre `gives_footmassage` y `gives_footmassage2`? que `gives_footmassage2` será verdadero siempre y cuando el segundo argumento sea `mia`, pero jamás sabremos con certeza quién le da un footmassage a `mia`. Entonces, `marsellus` no sabe. En el `gives_footmassage`, `marsellus` sabe que tendrá que matar al `X` que le haga masajes en los pies a `mia`.

Entonces: NO usar variables anónimas en definiciones de predicados a menos que nunca se quiera unificar o devolver una respuesta concreta de ese argumento.

Unificación

Es **fundamental** entender qué es la unificación. Decimos que dos términos unifican si son el mismo término o ellos contienen variables que pueden ser instanciadas como términos que pueden terminar siendo términos iguales.

Unifican: `42=42` unifica porque son el mismo número, `mia=mia` unifica porque son el mismo átomo, `X=X` unifican porque son la misma variable y `woman(mia) = woman(mia)` unifican porque son el mismo término complejo.

No unifican: `42=41`, ni tampoco `woman(mia) = woman(vincent)`, ni tampoco `loves(a, b) = loves(mia, vincent)`

¿Qué sucede con `woman(mia) = woman(X)`? esto si unifica, porque existe una unificación para `X` de tal manera que `woman(mia) = woman(X)` y esto sucede si y solo si `X = mia`, es decir, `X` debe instanciarse con el valor de `mia` para unificar.

¿Qué sucede con `loves(vincent,X) = loves(X, mia)`?, esta no unifica. ¿por qué?

- Dec \rightarrow : $\{vincent \stackrel{?}{=} X\}, \{X \stackrel{?}{=} mia\}$
- Swap \rightarrow : $\{X \stackrel{?}{=} vincent\}, \{X \stackrel{?}{=} mia\}$
- Delete $\{X := vincent\}$: $\{vincent \stackrel{?}{=} mia\}$ y esto es falso, pues los términos `vincent` y `mia` no unifican.

Usualmente, no estamos solamente interesados en el hecho de que dos términos unifiquen, sino que tambien queremos ver cómo las variables tienen que ser instanciadas para que sean iguales. Prolog nos da esta información pues cuando unifica dos términos realiza todas las instanciaciones necesarias.

```
1 | 2 = 2 unifica
2 | '2' = 2 no unifica
3 | mia = X unifica, X = mia
4 | X = Y unifica
5 | loves(X,X) = loves(marsellus, mia) no unifica, porque no hay forma de que X tenga dos posibles
   |   unificaciones a la vez.
```

Importante: Por defecto, Prolog no tiene activo el occurs-check. Esto quiere decir, que podríamos tener referencias circulares infinitas si no tenemos cuidado. Esto quiere decir que esto: $X_1 \stackrel{?}{=} X_3 \rightarrow X_1$ es posible, aunque no debería de serlo.

Occurs-Check

El Occurs-Check es la ocurrencia de una variable que estamos definiendo, que dependa de sí misma. Esto produce referencias circulares infinitas.

Ej.: $father(X) := X$ es una regla que tiene Occurs-Check, es decir, se cuelga. Esto viene a que si $X = father(X)$ entonces esto sería $father(father(X)) = father(X)$ pero a su vez $X = father(X)$ entonces $X = father(father(father(father(...))))$. Esto sucede porque Prolog es un lenguaje optimista. Este asume que vos no le vas a enviar nada peligroso.

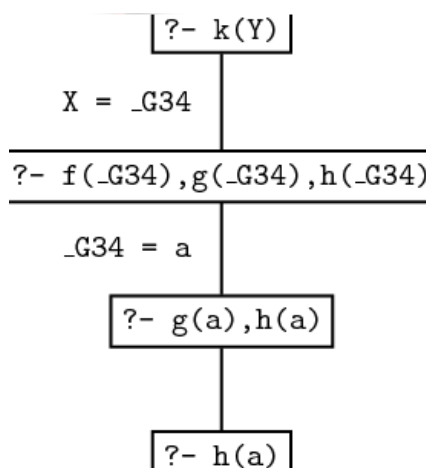
Búsqueda de Soluciones (Proof Search)

Evalúa cada una de los hechos/reglas que hagan match, luego, si la regla posee un body, va tomando cada una de las condiciones de izquierda a derecha. Busca las posibles soluciones y hace backtracking para ver cuales cumplen la segunda, y así sucesivamente. Cuando hacemos una consulta, las cosas que deben cumplirse para que sea verdadera, se llaman **goals** o cláusulas objetivo.

```
1 | 1. f(a).
2 | 2. f(b).
3 | 3. g(a).
4 | 4. g(b).
5 | 5. h(b).
6 |
7 | 6. k(X) :- f(X), g(X), h(X).
```

Desarrollemos, en forma de árbol qué es lo que hace Prolog para encontrar la solución a $?-k(Y)$

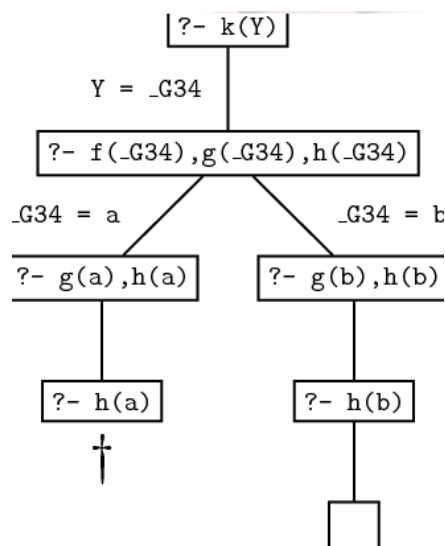
- Teniendo $k(Y)$ lo primero que hará, es buscar la primera ecuación con la que pueda unificar. como $k(Y)$ falla tratando de unificar con los hechos 1 al 5 por Clash, entonces llegamos a que $k(Y)$ debe unificar al HEAD de la regla $k(X) :- f(X), g(X), h(X)$. Cuando Prolog unifica a una variable en un hecho o una regla, genera una nueva variable del estilo **k_G34** , entonces ahora Prolog sabe que $k_G34 :- f_G34, g_G34, h_G34$.
- El siguiente paso es que Prolog reemplaza la consulta original k_G34 por la lista de **goals**: f_G34, g_G34, h_G34
- El siguiente paso es que para resolver un goal: **toma la regla que está más a la izquierda**. Para resolver el goal de f_G34 se busca en la database, con qué hecho o regla unifica.
 - La primera unificación posible de f_G34 es con $f(a)$, por lo tanto ahora nos quedan dos **goals** restantes. Entonces, unificamos f_G34 a $f(a)$ y ahora, todas las ocurrencias de $_G34$ se instancian como a .
 - Por lo tanto, los goals quedaron así: **$g(a), h(a)$**
 - Como $g(a)$ es un hecho en nuestra database, nos queda solo una regla para probar, pero notamos que $h(a)$ no existe como hecho, por lo tanto, Prolog decide que produjo un error, por lo tanto da un paso atrás y se fija qué caminos alternativos tiene para tomar antes de fallar (en este caso ninguno). Este proceso de vuelta hacia atrás es conocido como **backtracking**.



- En este momento, como Prolog se equivocó instanciando $_G34$ con el valor de a hace un **REDO**.
- Ahora f_G34 toma el valor de $f(b)$ (siguiente ecuación en la lista de hechos)
 - Todas las ocurrencias de $_G34$ se instancian como b .
 - Por lo tanto, los goals quedaron así: **$g(b), h(b)$**
 - Como $g(b)$ es un hecho, entonces el último goal por ver es $h(b)$.
 - Luego $h(b)$ también es un hecho.

- Por lo tanto, el conjunto de Goals quedó vacío, por lo tanto, la consulta original es satisfactible, y Prolog encontró una forma para que lo sea (instanciar Y por b). Si escribimos `;`, Prolog hará backtrack y tratará nuevamente de buscar más posibles soluciones, pero en este caso, como no hay más devolverá False.

El árbol de búsqueda quedó de la siguiente manera:



Recursión en Prolog

Los predicados pueden ser definidos recursivamente. Hablando de una manera más formal, un predicado es recursivo sí y solo si está definido de una o más reglas que refieren a sí mismo.

Los predicados recursivos están conformados por un caso base, y un caso recursivo.

```

1  is_digesting(X,Y) :- just_ate(X,Y).
2  is_digesting(X,Y) :-
3      just_ate(X,Z),
4      is_digesting(Z,Y).
5
6  just_ate(mosquito,blood(john)).
7  just_ate(frog,mosquito).
8  just_ate(stork,frog).

```

Ejercicio.: ¿Qué salida arroja Prolog si la consulta es `is_digesting(stork,mosquito)`?

- `is_digesting(stork,mosquito)` unifica con la primera regla, esto quiere decir que buscará `just_ate(stork,mosquito)`. Como esto es falso, entonces sigue evaluando las demás posibilidades.
- `is_digesting(X,Y)` unifica con la segunda regla. El goal por el cual Prolog reemplaza a `is_digesting(X,Y)` es por `just_ate(stork,Z)` y `is_digesting(Z,mosquito)`.
 - `just_ate(stork,Z)`: No hace match con `just_ate(mosquito,blood(john))` por falla en clash.
 - `just_ate(stork,Z)`: no hace match con `just_ate(frog,mosquito)` por falla en clash.
 - `just_ate(stork,Z)`: hace match con `just_ate(stork,frog)`, por lo tanto el posible valor de Z es frog.
 - Por lo tanto, ahora el goal se reduce a `is_digesting(frog,mosquito)`, y como esto es un hecho y el goal ahora está vacío, Prolog demuestra que la consulta es satisfactible y la unificación para que lo sea es `Z = frog`.

Haskell

Para ejecutar un archivo hay que instalar GHCi. Una vez instalado, nos paramos en la terminal en el directorio donde está el archivo que queremos ejecutar.

- Cargar archivo: `:l nombreArchivo`
- Ver tipo: `:type tipo`
- Ejecutar funcion: `funcion parametro1 parametro2...`
- Recargar archivo: `:r`
- Si necesitamos hacer cálculos para mandar un parámetro, usar paréntesis: Ej.: `otherwise = n * factorial(n-1)`

Maybe

El Maybe se utiliza en Haskell para recibir/devolver respuestas condicionales que pueden ser de un tipo u otro.

Se define como $\text{data Maybe } a = \text{Nothing} \mid \text{Just } a$

Ej.: $\text{devolverFalsoSiVerdadero} : \text{Bool} \rightarrow \text{Prelude.Maybe Bool}$

El Maybe deja la puerta abierta a un valor posible "Nothing". Entonces tenemos dos casos: Si me envían un True devuelvo False (tipo bool), caso contrario, devuelvo Nothing.

Either

El Either se utiliza en Haskell para poder recibir/devolver un parámetro que podría ser de un tipo u otro.

Se define como $\text{data Either } a \ b = \text{Left } a \mid \text{Right } b$

Para poder saber qué operación hacer según el tipo literalmente en código usamos (Left valor) o (Right valor).

Ej.: $\text{devolverRepresentacionIntBool} :: \text{Either Int Bool} \rightarrow \text{Int}$

Si es un entero, devuelvo ese mismo entero porque no hago nada. Eso lo hacemos con $\text{Left}(a) = a$, ahora, si el tipo es booleano tengo que decir explícitamente la respuesta según su valor. Es decir, $\text{Right}(\text{False}) = 0$ sino, $\text{Right}(\text{True}) = 1$.

Declaración de tipos en Haskell

Se utiliza $\text{data nombretipo tipo} = \text{Tipo 1} \mid \text{Tipo 2 El} \mid$ se interpreta como **o bien**

Árboles Binarios

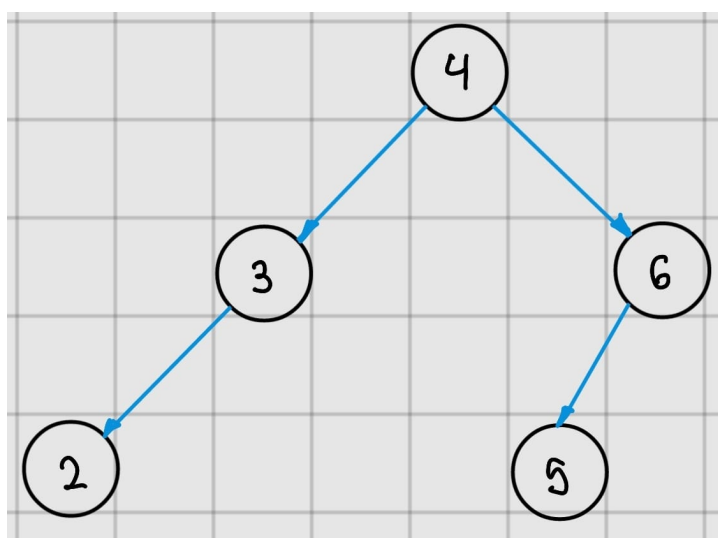
Es un tipo (para mi parecer) meramente recursivo.

$\text{data AB } a = \text{Nil} \mid \text{Bin } (\text{AB } a) \ a \ (\text{AB } a)$ Nótese que es algo re contra recursivo, porque para definir el tipo de AB a decimos que es un Bin que a su vez es de AB a y a su vez AB a es otro árbol binario. Veamos unos ejemplos de esto

- $\text{Bin } (\text{Nil}) \ \text{Nil} \ (\text{Nil})$: es el árbol que no tiene ni siquiera raíz. Y nótese que en cada paréntesis es importante indicar el Nil pues es la forma de que el tipado de Haskell nos lo acepte.
- $\text{Bin } (\text{Bin Nil } 3 \ \text{Nil}) \ 4 \ (\text{Bin Nil } 6 \ \text{Nil})$: Es el árbol que comienza con un Nodo raíz que tiene el valor de 4. El hijo izquierdo del Nodo con valor 4 es otro árbol binario que tiene como valor 3 en su nodo y no tiene hijos. El hijo derecho del Nodo con valor 4 es otro árbol binario que tiene como valor 6 en su Nodo y no tiene hijos.

Y así sucesivamente, veamos un dibujo para tener algo más visual.

El siguiente árbol binario: $\text{Bin } (\text{Bin } (\text{Bin Nil } 2 \ \text{Nil}) \ 3 \ \text{Nil}) \ 4 \ (\text{Bin } (\text{Bin Nil } 5 \ \text{Nil}) \ 6 \ \text{Nil})$ representa el siguiente:



Curry & Uncurry

Digamos que necesitamos currificar una función que recibe una tupla de elementos. Es decir, algo así: $\text{suma} :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$. Por la definición de curry necesitamos que por cada argumento, haya una función que lo devuelva, por lo tanto el resultado sería

algo así $\text{suma} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$.

Veamos el tipo de función que queremos currificar: $((a, b) \rightarrow c)$, esto lo queremos llevar a $a \rightarrow b \rightarrow c$.

Por lo tanto nuestra función curry sería algo así:

```
1 |   curryOwn :: ((a, b) -> c) -> a -> b -> c
2 |   curryOwn f a b = f (a, b)
```

Entonces, digamos que queremos hacer la suma currificada.

```
1 |   sumTuple :: (Float, Float) -> Float
2 |   sumTuple (x, y) = x + y
3 |
4 |   sumarCurry :: Float -> Float -> Float
5 |   sumarCurry = curryOwn sumTuple
```

Lo que hace `sumarCurry` es llamar a `curry(sumTuple)` es decir, a `curry` le manda la función `sumTuple`. Los parámetros que le mandamos a `sumarCurry` como a `-¿b`, los convierte en `(a, b)` para poder aceptar el tipo de la función `sumTuple`.

¿Cómo sería entonces la función `uncurry`? Si recibimos los argumentos en forma de $a \rightarrow b \rightarrow c$ debo llevarlo a $(a, b) \rightarrow c$

```
1 |   uncurryOwn :: a -> b -> c -> ((a, b) -> c)
2 |   uncurryOwn f (a, b) = f a b
3 |
4 |   sumarUncurry :: (a, b) -> c
5 |   sumarUncurry = uncurryOwn sumarCurry
```

Esto quiere decir que vamos a llamar a `sumarUncurry` que recibe la tupla, ahora `sumarCurry` está currificada, por lo que la tenemos que convertir nuevamente a la función no currificada, para luego llamar a `sumTuple` de la manera original.

Clases de Tipos

- Num a: Indica que el parámetro a es numérico
- Ord a: Indica que el parámetro a es ordenable bajo algun criterio, es decir, podemos aplicar $> < =$ etc.
- Eq a: Indica que el parámetro a se puede igualar, es decir, podemos aplicar $=$

Foldr

```
1 |   // Solo recorre listas de tipo a. Es decir, devuelve la suma de los elementos.
2 |   sumFoldrlist :: Num a => [a] -> a
3 |   sumFoldrlist = foldr (\x ac -> x + ac) 0
4 |
5 |   //Recorre tipos plegables. Acá no nos limitamos solo a listas, porque véase que usamos t a en vez de
6 |   [a]
7 |   sumFoldr :: (Foldable t, Num a) => t a -> a
8 |   sumFoldr = foldr (\x ac -> x + ac) 0
```

Foldr, el árbol de recursión y más de una lista

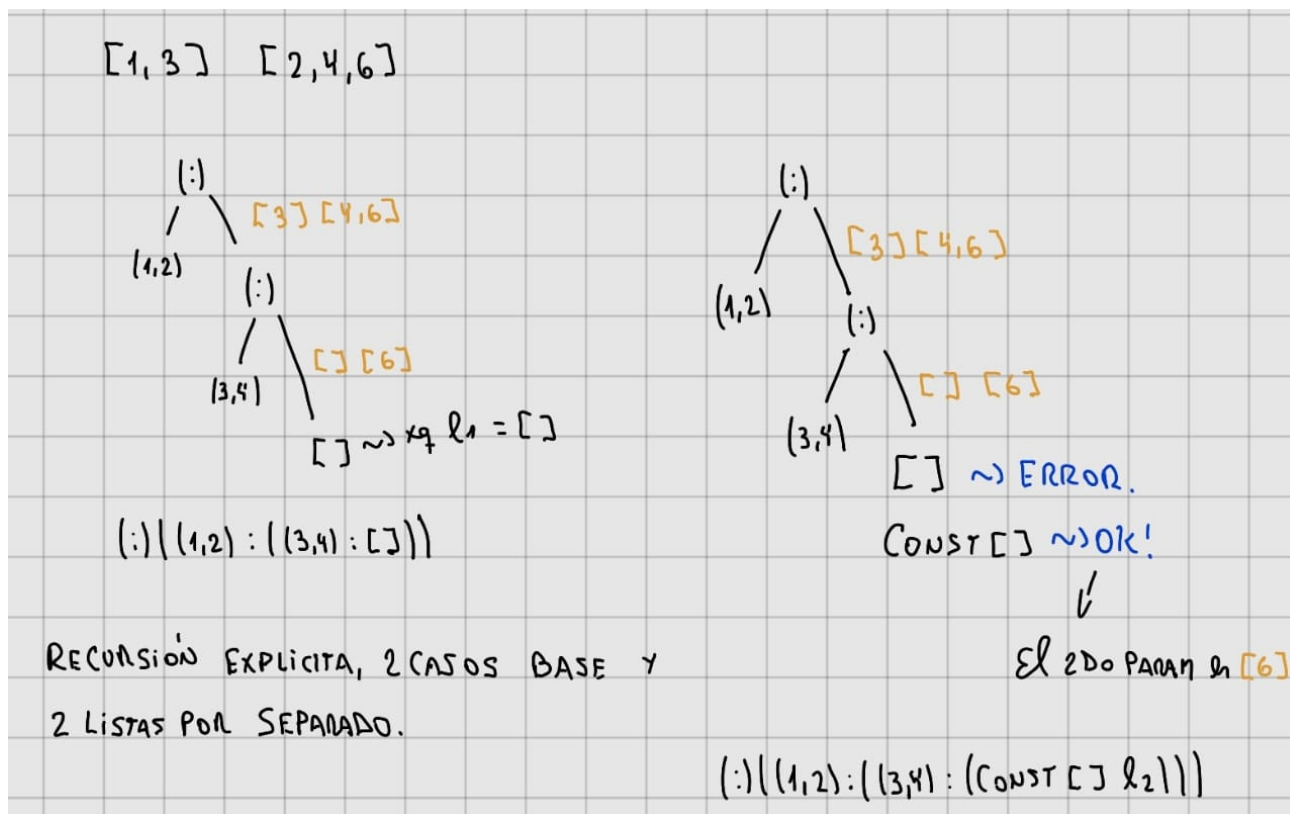
Uno de los problemas más normales es tener que enviar más de una lista a procesar a una función dada en Haskell y realizar recursión estructural.

Veamos el siguiente ejercicio: Arme pares de la forma $[(a, b)]$ usando recursión estructural.

Esto es súper simple si lo hacemos sin recursión estructural porque nos queda algo así

```
1 |   armarPares :: [a] -> [b] -> [(a, b)]
2 |   armarPares [] _ = []
3 |   armarPares _ [] = []
4 |   armarPares (x:xs) (y:ys) = (x, y) : armarPares xs ys
```

¿Cómo hacemos esto con foldr? Veamos el árbol recursivo.



El error en la recursión estructural vendría del lado de que como estamos recorriendo dos listas a la vez, y mi llamado recursivo es del tipo $[b] \rightarrow [(a, b)]$ no puedo devolver $[]$ entonces lo que hago es devolver `const []` y se aplica parcialmente al argumento que sería la segunda lista `const [] l2` y como la primera lista está vacía entonces devuelve $[]$.

Esto es súper importante a tener en cuenta, porque si mandamos más de un argumento en la recursión, recordar el concepto de curry.

Flip

Toma dos parámetros y devuelve una función que los devuelve en el orden inverso.

Es decir: $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$

Luego: $\text{flip } f \ a \ b = f \ b \ a$

Función identidad

Devuelve el mismo valor aplicado a la función.

Es decir: $(a \rightarrow b) \rightarrow a \rightarrow b$

Luego: $\$f \ a = f \ a$

Función constante

Devuelve un valor enviado sin aplicarle ninguna función.

Es decir: $a \rightarrow b \rightarrow a$

Luego: $\text{const } a \ b = a$

Reduciendo expresiones elegantemente

▪ $\text{filter}(\backslash x \rightarrow \text{length } x > 3)$

- ¿Puede hacerse algo mejor? No. Porque a x si o sí necesitamos aplicarle una función.

▪ $\text{filter}(\backslash x \rightarrow x > n)$

- ¿Puede hacerse algo mejor? Sí. $\text{filter}(> n)$

▪ $\text{filter}(\backslash x \rightarrow \text{mod } x 2 / = 0)$

- ¿Puede hacerse algo mejor? No. Porque a x le tenemos que calcular su módulo con 2.

- $\text{map}(\backslash x \rightarrow \text{map}(\backslash y \rightarrow \text{toUpper } y) \ x)$
 - ¿Puede hacerse algo mejor? Primero entendamos que hace, recorre una lista de palabras, luego en cada palabra toma cada letra y la pasa a mayúscula. Esto es un doble map, uno por palabra otro por letra. Entonces sí $\text{map}(\text{map toUpper})$
- $\text{doblarElementos.filtrarPares}$
 - ¿Puede hacerse algo mejor? No. Esto es el equivalente a un lenguaje imperativo hacer $\text{doblarElementos}(\text{filtrarPares}(\text{lista}))$

¿Qué hacen las siguientes funciones compuestas?

```

1 flip($) 0 id
2
3 (==0) . (flip mod 2)
4
5 Primero veamos que hace flip mod 2.
6 mod 2 es notación infija (Integral a => 2 -> a -> a), entonces lo que está diciendo es que si le paso
   cualquier número va a hacer mod 2 x, y nosotros por lo que yo entiendo es que queremos ver si es
   par.
7 Por lo tanto, lo primero que haríamos es invertir los argumentos de mod 2 con flip (Integral a => a
   -> 2 -> a), entonces quedaría algo como mod x 2 donde el x lo tenemos que enviar nosotros.
8 Luego, se compone la función de mod x 2 == 0 esperando solo un argumento donde verifica si
   efectivamente un número dado es par.
9 Entonces, (Integral a => x -> Bool)
10
11 map f = ((:) . f)
12 Lo que hace esta función es básicamente aplicar una función f a todos los elementos y agregarlos a
   una lista particular.
13
14 Dado ["hola", "abc"] quiero devolver ["cba", "aloh"]. Es decir, dar vuelta cada caracter de cada
   palabra y ademas dar vuelta las palabras.
15
16 reverseAnidado :: ["String"] -> ["String"]
17 reverseAnidado = reverse . (map . reverse)
18 Lo primero que hacemos es hacer un map haciendo reverse por cada caracter de la lista. Luego,
   reordenamos las palabras en sí.
19 El tipo de reverse es: [a] -> [a] pero con los elementos al revés. Entonces, por cada palabra (map)
   hacemos un reverse y las guardamos.
20 Finalmente, nos queda algo así ["aloh", "cba"], nos queda dar vuelta eso, entonces hacemos nuevamente
   un reverse de toda la lista. ["cba", "aloh"].
21
22 listacomp f xs p = [f x | x <- xs, p x]
23 listacomp f xs p = map f (filter p xs)

```

Ejercicios Foldl

1. Definir la función `sumasParciales` que dada una lista de números devuelve otra de la misma longitud que tiene en cada posición la suma parcial de los elementos de la lista original desde la cabeza hasta la posición actual.

Entendamos el enunciado:

- Vamos a usar `foldl` para ir sumando de izquierda a derecha.
- El tipado de `foldl` es $b \rightarrow a \rightarrow b$ donde b es nuestro primer argumento acumulador y a el elemento.
- Necesito de alguna manera tener el valor inmediato anterior. Podemos hacer algo como empezar enviando una lista vacía como caso base, y a medida que vamos haciendo la recursion tomar la cabeza de la lista.
- Si la lista esta vacía entonces solo agrego x a la lista de la recursion (primer elemento), si no esta vacía sumo con el elemento de la lista (cabeza)
- Porque `foldl` labura así $\rightarrow [1, 2, 3]$
 - $1:[] = [1]$
 - $2 + (\text{head } [1]) : [1] = [3, 1]$
 - $3 + (\text{head } [3, 1]) : [3, 1] = [6, 3, 1]$

- Ahora podemos usar reverse y el resultado es [1, 3, 6]

Entonces la solución sería algo así:

```
1 | sumasParciales :: Num a => [a] -> [a]
2 | sumasParciales = reverse . foldl(\acc x -> if(length acc > 0) then x+(head acc):acc else x:acc) []
```

Ejercicios Map

1. Realice una función mapDoble que toma una función currificada de dos argumentos y dos listas de igual longitud y devuelve una lista de aplicaciones de la función a cada elemento correspondiente de las dos listas.

Básicamente $f = x + y$ $l1 = [1, 2]$ $l2 = [3, 4]$ da como resultado $[4, 6]$

Desglosemos el ejercicio en partes

- 1. Lo primero que necesito hacer básicamente es recorrer ambas listas de alguna manera a la vez, y obtener algo como [(1, 3), (2, 4)] y luego aplicar a esa lista de pares la función f. Si nos ponemos a pensar, basta con hacer una función que reciba dos listas de tipo [a] y [b] y devuelva una lista de [(a, b)].
- 2. Una vez que tenemos esta lista de pares, sabemos que la función que nos va a enviar tiene que utilizar ambos elementos a la vez, pero la función es de la forma $a \rightarrow b \rightarrow c$ y esto quiere decir que está currificada pero nuestra lista de pares es de tipo [(a, b)] por lo tanto antes de aplicar f debemos aplicar *uncurry f lista* para que cuando mandemos f y la lista, f se convierta en una función que espere (a, b).
- 3. Por último, para aplicar a todos los elementos de la lista podemos usar map de la siguiente forma: *map (uncurry f)(lista)*

```
1 | mapDobleCorta :: (a -> b -> c) -> [a] -> [b] -> [c]
2 | mapDobleCorta f l r = map (uncurry f) (armarPares l r)
```

donde la función armarPares tiene la siguiente pinta

```
1 | armarPares :: [a] -> [b] -> [(a, b)]
2 | armarPares _ [] = []
3 | armarPares [] _ = []
4 | armarPares (x:xs) (y:ys) = (x, y) : armarPares xs ys
```

2. Realice una suma de matrices.

- Idea: Necesitamos de alguna manera recorrer la fila 1 de la matriz 1 y la fila 1 de la matriz 2. Esto lo podemos hacer facilmente reutilizando el ejercicio anterior (armarPares), es decir, enviamos armarPares con la fila1 matriz1 y fila1 matriz2. Esto nos armaría los pares de esa fila.
- Tenemos que generalizar este proceso para cada fila. Por lo tanto podemos recibir una matriz, y podemos hacer recursion sobre cada lista de la matriz.
- A su vez, vamos a necesitar que una vez que tenemos los pares armados, sobre esos pares se aplique un map haciendo uncurry sobre f. Porque si tenemos $F1 \ M1 + F1 \ M2 = [(1, 2), (3, 4), (5, 6)]$ esto indica que la fila de la M1 es [1, 3, 5] y la fila de la M2 es [2, 4, 6]. Por lo tanto, lo que necesito hacer es convertirlo en [[9, 12]] y agregarlo a la lista resultante. El Uncurry acá es ultra importante porque mi función pide $Int \rightarrow Int \rightarrow Int$ y yo voy a mandar $(Int, Int) \rightarrow Int$.

```
1 | sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
2 | sumaMat _ [] _ = []
3 | sumaMat _ _ [] = []
4 | sumaMat f (x:xs) (y:ys) = map (uncurry f) (armarPares x y) : sumaMat f xs ys
```

¿Qué es lo que podríamos cambiar? Estamos haciendo un laburo exactamente igual mapDoble.

```
1 | sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
2 | sumaMat f = mapDoble(mapDoble(f))
```

Armando funciones que permitan hacer recursión sobre un tipo dado

1. **foldNat**: Necesitamos hacer recursión sobre los números enteros. Una excelente pregunta es ¿recursión sobre números naturales?. Sí.

Un número natural se define de la siguiente manera *data Nat = Zero | Succ Nat*. Es decir, tiene dos chances: O es cero, o es un sucesor de algún número.

¿Cuántos casos tendríamos que probar si quisieramos verificar la correctitud del tipo? 2. Que sea Zero o que sea algún sucesor.

Así, de esta manera, podemos definir al número 4 como *Succ(Succ(Succ(Succ Zero)))*.

La recursión nos sirve justamente para esto, para poder hacer operaciones con números naturales.

Ej.: Necesitamos multiplicar un número n m veces ¿Cómo hacemos esto? Sumamos el mismo número m veces. Es decir, si quiero hacer $n * n$ equivale a decir $n+n+n+n+n$. Entonces ¿Cómo podríamos hacer esto?

Para empezar, pensemos en qué tipo de operaciones queremos hacer con foldNat. Podríamos hacer multiplicación, potencia, etc. Pensemos por un momento ¿qué pasaría si el caso base fuese 0 si estamos sumando? Nada, porque justamente para la multiplicación la queremos hacer como $n+n+n+n+n$ y si llegamos a Zero quiero que devuelva 0.

Ahora ¿qué sucede si queremos hacer la potencia? Recordemos que la potencia se define como la multiplicación de un número m veces. Si nos abstraemos a nuestro esquema $n^m \equiv n * n * n * n \dots m \equiv (n + n) + (n + n) + (n + n) + (n + n) \dots m \text{ veces}$ si quisieramos aplicar el caso base de 0 para la multiplicación se nos haría 0. Por lo tanto tenemos otro caso base, sería 1.

Por lo tanto, definamos el foldNat pero utilizando el tipo de Integer (como pidió la cátedra)

```
1 | foldNat :: Integer -> (Integer -> Integer) -> Integer -> Integer
2 | foldNat base _ Zero = base
3 | foldNat base f n = f (foldNat base f (n-1))
```

Entonces ahora podemos definir la multiplicación como

```
1 | multiplicacion :: Integer -> Integer -> Integer
2 | multiplicacion n m = foldNat 0 (+n) m
```

¡Nótese que acá el caso base es 0 porque estamos sumando!

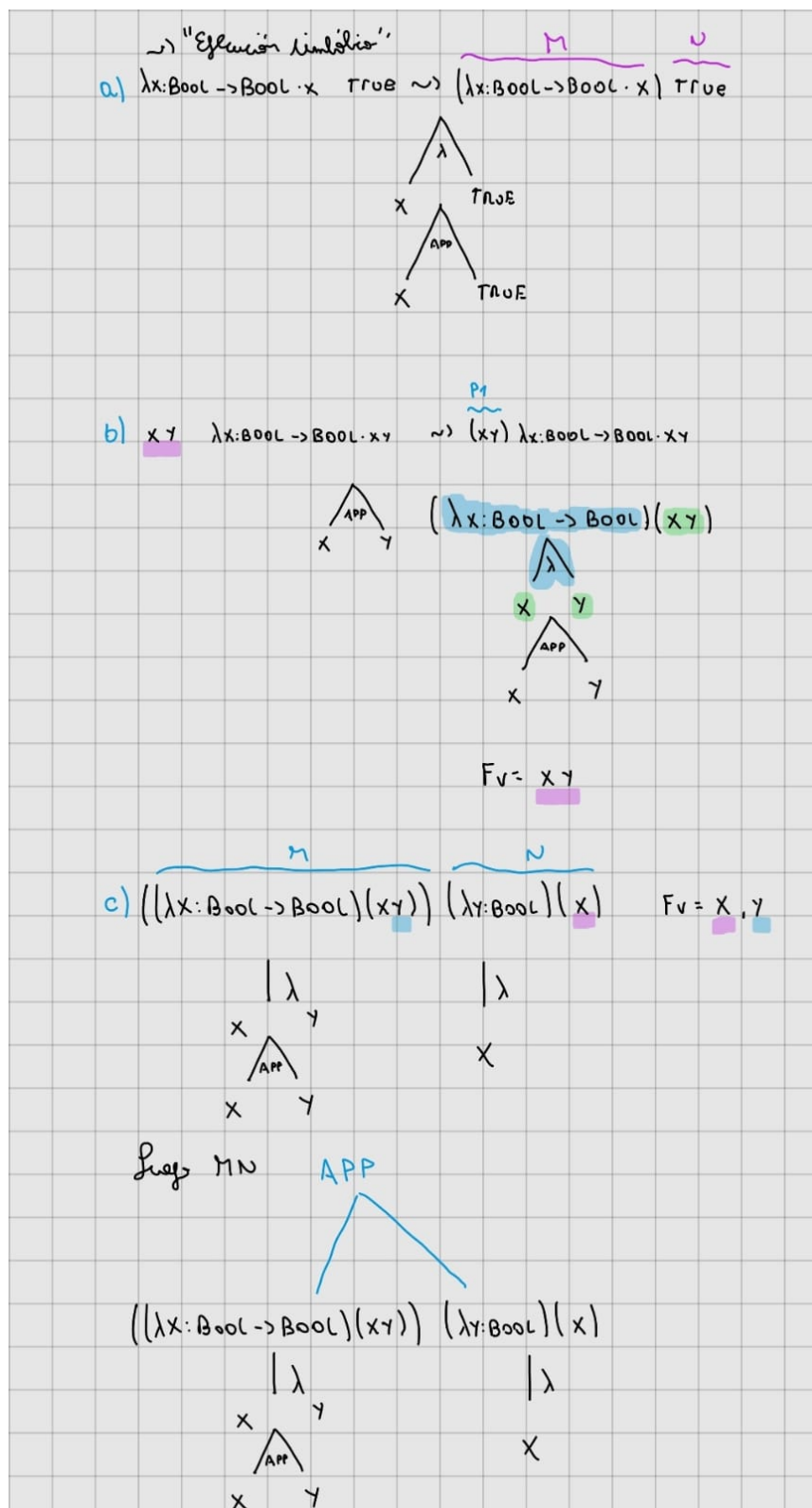
Por último podemos definir la potencia reutilizando la multiplicación

```
1 | potencia :: Integer -> Integer -> Integer
2 | potencia n m = foldNat 1 (multiplicacion n m) m
```

Es importante notar que la potencia requiere hacer foldNat nuevamente porque tenemos que hacer el proceso de multiplicación m veces.

¿Son términos válidos?

Recordemos que asocia a la derecha las implicaciones y hacia izquierda la aplicación.
Lo que hay que revisar bien siempre son todos los tipos.



La idea es ir aplicando como Haskell la asociación, separar en varios pasos la ejecución y luego aplicarlo.

Términos LPO

Sea $=\{d, f, g\}$ donde d tiene aridad 0, f aridad 2 y g aridad 3. ¿Cuales de la siguientes cadenas son términos sobre \mathcal{F} ?
Recordemos definiciones:

- Si un Símbolo de Función tiene aridad 0: es constante.

- Si f tiene una aridad n , cuando se utilice deben enviarse los n parámetros.
 - Un término tiene la pinta: $t ::= X \mid f(t_1, \dots, t_n)$
1. $g(d, d)$: No es término. g es un símbolo de función pero de aridad 3 y acá se le están enviando solo dos.
 2. $f(X, g(Y, Z), d)$: No es término. Mismo caso que arriba, f está mal aplicado y g también. Aridades incorrectas.
 3. $g(X, f(d, Z), d)$: Es un término. f y g están aplicados con su aridad esperada, y los elementos que se envían por parámetro son términos irreducibles y/o constantes.
 4. $g(X, h(Y, Z), d)$: h no está definido en nuestro conjunto de Símbolos de Funciones.

Fórmulas Válidas, uso de Predicados y Funciones

Sea c una constante, f un símbolo de función de aridad 1 y S y B dos símbolos de predicados binarios. ¿Cuales de las siguientes son fórmulas?

Recordemos la teoría

- Una constante C es una función con aridad 0.
 - Las funciones o símbolo de función esperan siempre los n argumentos con los cuales se definieron. Devuelven un elemento del dominio.
 - Los predicados S y B , binarios (reciben dos argumentos) hacen una relación específica entre dos términos irreducibles y arrojan un valor de verdad.
1. $S(c, X)$: Es una fórmula, estamos comparando una constante c con un valor irreducible X . Devuelve un valor de verdad.
 2. $B(c, f(c))$: Es una fórmula, estamos comparando una constante c con un valor irreducible que produce $f(c)$.
 3. $f(c)$: No es una fórmula. Es un valor del dominio, no reduce a un valor de verdad.
 4. $B(B(c, X), Y)$: No es una fórmula, un predicado no puede recibir como parámetro un valor de verdad ($B(c, X)$).
 5. $S(B(c), Z)$: Mismo caso que el anterior
 6. $(B(X, Y) \implies (\exists Z. S(Z, Y)))$: Es una fórmula. $B(X, Y)$ es valor de verdad, \implies define una fórmula y $(\exists Z. S(Z, Y))$ es una fórmula pues el Z que existe está en nuestro dominio, y $S(Z, Y)$ relaciona dos elementos de nuestro dominio y arroja un valor de verdad.
 7. $(S(X, Y) \implies S(Y, f(f(X))))$: Es una fórmula. $S(X, Y)$ es una fórmula, \implies es una fórmula, y $S(Y, f(f(X)))$ es una fórmula porque al aplicar dos veces f , nos termina devolviendo un elemento de nuestro dominio que luego es comparado con Y en el símbolo de predicado S .
 8. $B(X, Y) \implies f(X)$: No es una fórmula. $f(X)$ es un término o valor de nuestro dominio, no es un valor de verdad. Se le debería aplicar un símbolo de predicado para que arroje un valor de verdad.
 9. $S(X, f(Y)) \wedge B(X, Y)$: Es una fórmula.
 10. $\forall X. B(X, f(X))$: Es una fórmula. Para todo elemento posible de nuestro dominio X , al enviarlo a f nos devuelve un valor del dominio y se lo compara con el X . Luego, aplicar B devuelve un valor de verdad.
 11. $\exists X. B(Y, X(C))$: No es una fórmula, porque X es un elemento del dominio y acá se lo está usando como función de aridad 1 (creo)

Pasaje de LPO a Forma Clausal

1. Sea $\sigma = \exists X. \forall Y. (P(X, Y) \wedge Q(X) \wedge \neg R(Y))$

- Reemplazamos todas las ocurrencias de implicaciones por su equivalente. Como acá no hay, no hacemos nada.
- Reemplazamos todas las ocurrencias de las negaciones exteriores hacia adentro, como acá están todas dentro, no hacemos nada.
- Movemos todos los cuantificadores hacia afuera. Como acá no hay adentro, no hacemos nada.
- Pasamos los cuantificadores existenciales a función o constante, en este caso como el existencial está fuera de todo (no depende de ningún cuantificador universal), entonces nuestra hipotética x , será una constante. Por lo tanto $= \forall Y. (P(C, Y) \wedge Q(C) \wedge \neg R(Y))$
- Distribuimos los \vee sobre \wedge . Como acá no hay, no hacemos nada.
- Por último, los cuantificadores universales por cada \wedge distribuimos. Por lo tanto $= \forall Y. P(C, Y) \wedge \forall Y. Q(C) \wedge \forall Y. \neg R(Y)$
- Entonces la forma clausal es: $\mathcal{C} = \{\{P(C, Y)\}, \{Q(C)\}, \{\neg R(Y)\}\}$