

# Notas para final de PLP

Manuel Panichelli

June 3, 2022

# Contents

<b>1</b>	<b>Paradigma funcional</b>	<b>2</b>
1.1	Haskell	2
1.1.1	Programación funcional	2
1.1.2	Curricación	4
1.1.3	Pattern matching	4
1.1.4	Tipos recursivos	5
1.1.5	Listas	5
1.1.6	No terminación y orden de evaluación	6
1.1.7	Evaluación lazy	6
1.1.8	Esquemas de recursion	6
1.1.9	Transparencia referencial	8
1.1.10	Folds	8
1.2	Cálculo Lambda Tipado	11
1.2.1	Sistema de tipado	12
1.2.2	Resultados básicos	14
1.2.3	Semántica	14
1.3	Extensiones de Cálculo Lambda	20
1.3.1	$\lambda^{bn}$ - Naturales	20
1.3.2	$\lambda^{\dots r}$ - Registros	21
1.3.3	$\lambda^{bnu}$ - Unit	22
1.3.4	Referencias	23
1.3.5	Recursión	30
1.4	Inferencia de tipos	32
1.4.1	Definición formal de inferencia	32
1.4.2	Variables de tipos	33
1.4.3	Sustitución	34

# Chapter 1

## Paradigma funcional

### 1.1 Haskell

**Def. 1.1** (Paradigma). Un **paradigma** es una forma de pensamiento.

**Def. 1.2** (Lenguaje de programación). Un **lenguaje de programación** es el lenguaje que usamos para comunicar lo que queremos que haga una computadora.

Usamos un lenguaje para describir los computos que lleva a cabo la computadora.

Es **computacionalmente completo** si puede expresar todas las funciones computables. Hay DSLs (*domain specific languages*) que no pueden expresar todo lo computable.

**Def. 1.3** (Paradigma de lenguaje de programación). Lo entendemos como un *estilo* de programación, que tiene que ver con los estilos de las soluciones. Está vinculado con lo que es para uno un modelo de cómputo.

Lo que vemos antes de la materia es el imperativo: a partir de un estado inicial llegar a un estado final. Programamos con secuencias de instrucciones para cambiar el estado.

#### 1.1.1 Programación funcional

Definiciones:

- **Programa y modelo de cómputo**: Programar es definir funciones, y ejecutar es evaluar expresiones.
- **Programa**: Es un conjunto de ecuaciones. Por ej. `doble x = x + x`
- **Expresiones**: El significado de una expresión es su valor (si es que está definido). El valor de una expresión depende solo del valor de sus sub-

expresiones. Evaluar o reducir una expresión es obtener su valor (por ej. `doble 2`  $\rightsquigarrow$  4) No toda expresión denota un valor, por ejemplo `doble true`.

- **Tipos:** El universo de valores está particionado en colecciones denominadas *tipos*, que tienen operaciones asociadas.

Haskell es **fuertemente tipado**. Toda expresión bien formada tiene un tipo, que depende del tipo de sus subexpresiones. Si no puede asignarse un tipo a una expresión, no se la considera bien formada.

```
1      :: Int
'a'    :: Char
1.2    :: Float
True   :: Bool
[1, 2, 3] :: [Int]
(1, True) :: (Int, Bool)
succ    :: Int -> Int
```

Definiciones de funciones:

```
-- Definición
doble :: Int -> Int
doble x = x + x

-- Guardas
signo :: Int -> Bool
signo n | n >= 0 = True
        | otherwise = False

-- Definiciones locales
f (x, y) = g x + y
    where g z = z + 2

-- Expresiones lambda
\x -> x + 1
```

### Tipos polimórficos

```
id x = x
id :: a -> a
-- x es de tipo a, que eventualmente se va a instanciar a algún tipo
```

**Clases de tipos:** Son como interfaces, que definen un conjunto de operaciones.

```
maximo :: Ord a => a -> a -> a
maximo x y | x > y = x
maximo _ y = y
-- Ord: (<), (<=), (>=), (>), max, min, compare
```

### Tipos algebraicos

```
data Figura = Circulo Float | Rectangulo Float Float
deriving Eq -- deriva la igualdad nativa

-- (Circulo 1) == (Circulo 1)
```

Estas cosas nos permiten hacer funciones genéricas.

**Funciones de alto orden:** las funciones son first class citizens, se pueden pasar como parámetro.

### 1.1.2 Currificación

Es un mecanismo que permite reemplazar argumentos estructurados por una secuencia de argumentos "simples". Ventajas:

- Evaluación parcial: `suma = suma 1`
- Evita escribir paréntesis (asumiendo que la aplicación asocia a izquierda).  
`suma 1 2 = ((suma 1) 2)`

#### curry y uncurry

En criollo: una equivalencia entre una func con muchos parametros (una tupla) y una funcion equivalente que va tomando de a uno y devuelve funciones.

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

suma x y = x + y
suma' :: (Int, Int) -> Int
suma' (x, y) = x + y

curry suma' 1 2 = suma' (1, 2)
curry suma' :: (Int -> (Int -> Int))

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

### 1.1.3 Pattern matching

Una forma copada de definir funciones. Es un mecanismo para comparar un valor con un patrón. Si la comparación tiene éxito se puede deconstruir un valor en sus partes.

```
data Figura = Circulo Float | Rectangulo Float Float

area :: Figura -> Float
area (Circulo radio) = pi * radio ^ 2
area (Rectangulo l1 l2) = l1 * l2
```

El patrón está formado por el constructor y las variables. Los casos se evalúan en el orden en el que están escritos.

```
esCuadrado :: Figura -> Bool
-- No vale esto?
-- esCuadrado (Rectangulo x y) = (x == y)
esCuadrado (Rectangulo x y) | (x==y) = True
esCuadrado _ = False
```

También se pueden definir funciones parciales (que no estén definidas para todo el dominio).

#### 1.1.4 Tipos recursivos

La definición de un tipo puede tener uno o más parámetros del tipo

```
data Natural = Zero | Succ Natural

Zero :: Natural          -- 0
succ Zero :: Natural     -- 1
succ (succ (succ Zero)) :: Natural -- 2

dameNumero :: Natural -> Int
dameNumero Zero = 0
dameNumero (Succ n) = dameNumero n + 1
```

#### 1.1.5 Listas

Tipo algebraico paramétrico recursivo con dos constructores:

```
[] :: [a]                -- lista vacia
(:) :: a -> [a] -> [a]   -- constructor infijo

-- Ejemplo
-- 1 : [2, 3] = [1, 2, 3]
```

Pattern matching

```
vacía :: [a] -> Bool
vacía [] = True
vacía _ = False

long :: [a] -> Int
long [] = 0
long x:xs = 1 + long xs
```

### 1.1.6 No terminación y orden de evaluación

```
-- No terminación
inf1 :: [Int]
inf1 = 1 : inf1

-- Evaluación no estricta
const :: a -> b -> a
const x y = x

-- const 42 inf1 -> 42 (pero depende del mecanismo de reducción del
-- lenguaje)
```

### 1.1.7 Evaluación lazy

el modelo de cómputo de haskell es la **reducción**. Se reemplaza un *redex* por otro usando las ecuaciones orientadas. Un redex (reducible expression) es una sub-expresión que no está en forma normal (irreducible).

Un redex debe ser una **instancia** del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las variables correspondientes ligadas. El resto de la expresión no cambia.

Haskell hace esto hasta llegar a una forma normal, un valor irreducible.

`const x y = x`. `const x y` es un redex, y lo reduzco a `x`.

Y cómo selecciono una redex? **Orden normal** (lazy). Se selecciona el redex más externo para el que se pueda conocer que ecuación del programa utilizar. En general, primero las funciones más externas y luego los argumentos, solo de ser necesarios.

Modo aplicativo: reduce primero todos los argumentos. Se hace en otros lenguajes como c.

### 1.1.8 Esquemas de recursion

Formas de recursion comunes que uno puede aprovechar usando funciones de alto orden.

#### Map

```
-- tal que dobleL xs es la lista que contiene el doble de cada elemento en xs
dobleL :: [Float] -> [Float]
dobleL [] = []
dobleL (x:xs) = 2*x : dobleL xs

-- tal que la lista esParL xs indica si el correspondiente elemento en xs es par
-- o no
esParL :: [Int] -> [Bool]
```

```

esParL [] = []
esParL (x:xs) = (even x) : esParL xs

-- tal que longL xs es la lista que contiene las longitudes de las listas en xs
longL :: [[a]] -> [Int]
longL [] = []
longL (x:xs) = (length x) : longL xs

-- esquema recursivo de map:
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map g (x:xs) = g x : map g xs

-- Con eso, se pueden reescribir como
dobleL = map ((*) 2)
esParL = map even
longL = map length

```

## Filter

```

-- tal que negativos xs contiene los elementos negativos de xs
negativos :: [Float] -> [Float]
negativos [] = []
negativos (x:xs)
  | x < 0 = x : (negativos xs)
  | otherwise = negativos xs

-- tal que la lista noVacias xs contiene las listas no vacias de xs
noVacias :: [[a]] -> [[a]]
noVacias [] = []
noVacias (l:ls)
  | (length l > 0) = l : (noVacias ls)
  | otherwise = noVacias ls

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

-- luego quedan
negativos = filter (\x -> x < 0)
noVacias = filter (\l -> length l != 0)
noVacias = filter (> 0) . length -- f o g = f(g(x))

```



### 1.1.9 Transparencia referencial

El valor de una expresion en funcional depende solo de sus subexpresiones. Esto a diferencia de imperativo que depende del estado.

Si dos expresiones son iguales, denotan el mismo valor bajo el mismo contexto.

#### 1.1.10 Folds

foldr

```
-- Funciones sobre listas

-- sumaL: suma de todos los valores de una lista de enteros
sumaL :: [Int] -> Int
sumaL [] = 0
sumaL (x:xs) = x + (sumaL xs)

-- concat: la concatenación de todos los elementos de una lista de listas
concat :: [[a]] -> [a]
concat [] = []
concat (l:ls) = l ++ (concat ls)

-- reverso: el reverso de una lista
reverso :: [a] -> [a]
reverso [] = []
reverso (x:xs) = (reverso xs) ++ [x]

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

-- Luego, con esto
sumaL = foldr (+) 0
concat = foldr (++) []
reverso = foldr (\x rec -> rec ++ [x]) []
reverso = foldr ( (flip (++) ) . (:[ ]) ) []

-- Hasta podemos definir map y filter. El fold es más general que el map y
-- filter
map f = foldr (\x rec -> f x : rec) []
map f = foldr ((:) . f) []

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

```

filter p = foldr (\x rec -> if p x then x : rec else rec) []

-- Longitud y suma con una sola pasada sobre la lista
sumaLong :: [Int] -> (Int, Int)
sumaLong = foldr (\x (r1, rn) -> (r1 + 1, rn + x)) (0, 0)

recr

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

-- ejemplo
dropWhile even [2, 4, 1, 6] = [1, 6]

-- drop while cuando se termina de cumplir devuelvo todo lo que viene "a la
-- derecha", pero cuando hago fold, lo que está a la derecha ya pasó por la
-- recursión.

dw p = first $ (foldr (\x (r1, r2)
    -> (if p x then r1 else x: r2, x: r2 ) ), ([], []))

-- Otro esquema más poderoso
g :: [a] -> b
g [] = z
g (x:xs) = f x xs (g xs)

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

dropWhile p = recr [] (\x xs rec -> if p x then rec else x:xs)

-- foldr en terminos de recr?
foldr f z = recr z (\x xs rec -> f x rec)

-- recr en términos de foldr?
recr z f = first $
    foldr
        (\x (rs1, rs2) -> (f x rs2 rs1, x:rs2))
        (z, [])

```

foldl

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

foldr = fold a la derecha, y foldl = fold a la izquierda

```
reverse = foldl (\c x -> x:c) []
reverse = foldl (flip (:)) []
```

Y uno en términos del otro? *Me falta repasar esto porque estaba matado, min 2:35:10 del video*

### Fold sobre estructuras algebraicas

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

-- Por ej.

```
Nodo 1 (Hoja 2) (Hoja 3)
```

-- Es

--

```
--   1
--  / \
-- 2   3
```

-- Y sobre ella podemos querer operaciones, como map

```
mapA :: (a -> b) -> Arbol a -> Arbol b
mapA f (Hoja x) = Hoja f x
mapA f (Nodo x izq der) = Nodo (f x) (mapA f izq) (mapA f der)
```

-- Y también podemos hacer un fold

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

```
sumaA = foldA id (\x rizq rder -> x + rizq + rder)
contarHojas = foldA 1 (\x rizq rder -> rizq + rder)
```

-- Arboles generales

```
data AG = NodoAG a [AG a]
```

```
mapAG :: (a -> b) -> AG a -> AG b
```

```
mapAG f (Nodo AG a as) -> NodoAG (f a) (map (mapAG f) as)
```

```
foldAG :: (a -> [b] -> b) -> AG a -> b
foldAG f (NodoAG a as) = f a (map (foldAG f) as)
```

Aplicar un fold con un constructor es la identidad.

## 1.2 Cálculo Lambda Tipado

Es el formalismo que está detrás de la programación funcional. Es un modelo de cómputo basado en **funciones**, introducido por Alonzo Church en 1934. Es computacionalmente completo (turing completo) y nosotros vamos a estudiar la variante tipada (Church, 1941.)

La máquina de turing es más de estado, y este con reducción de expresiones.

**Def. 1.4** (Tipos). Las **expresiones de tipos** (o tipos) de  $\lambda^b$  (lambda cálculo b de booleano) son

$$\sigma, \tau ::= Bool \mid \sigma \rightarrow \tau$$

Informalmente,

- $Bool$  es el tipo de los booleanos, y
- $\sigma \rightarrow \tau$  es el tipo de las funciones de tipo  $\sigma$  en tipo  $\tau$ .

**Ejemplo.** Ejemplos:

- $Bool \rightarrow Bool$
- $Bool \rightarrow Bool \rightarrow Bool$

**Def. 1.5** (Terminos). Los términos se escriben con las siguientes reglas de sintaxis.

Sea  $\chi$  un conjunto infinito enumerable de variables, y  $x \in \chi$ . Los **términos** de  $\lambda^b$  están dados por,

$$\begin{aligned} M, N, P, Q ::= & x \\ & | true \\ & | false \\ & | \text{if } M \text{ then } P \text{ else } Q \\ & | \lambda x : \sigma. M \\ & | M N \end{aligned}$$

**Ejemplo.** Ejemplos:

- $\lambda x : Bool.x$  ✓
- $\lambda x : Bool. \text{if } x \text{ then } false \text{ else } true$  ✓
- $\lambda f : Bool \rightarrow Bool \rightarrow Bool. \lambda x : Bool. f \ x$  ✓
- $(\lambda f : Bool \rightarrow Bool. f \ true)(\lambda y : Bool. y)$  ✓
- $true \ (\lambda x : Bool.x)$  ✓
- $x \ y$  ✓
- $\lambda x : true$  ✗

### 1.2.1 Sistema de tipado

Es un sistema formal de deducción o derivación que utiliza axiomas y reglas de inferencia para caracterizar un subconjunto de los términos llamados **tipados**. Nos permite quedarnos con algunos y rechazar otros términos en base a lo que consideremos correcto.

Definimos una **relación de tipado** en base a reglas de inferencia.

- Los **axiomas de tipado** establecen que ciertos **juicios de tipado** son derivables.
- Las **reglas de tipado** establecen que ciertos **juicios de tipado** son derivables siempre y cuando ciertos otros lo sean.

**Def. 1.6** (Variables libres). Una variable puede ocurrir **libre** o ligada en un término. Decimos que  $x$  ocurre **libre** si no se encuentra bajo el alcance de una ocurrencia de  $\lambda x$ . Caso contrario ocurre ligada.

Ejemplos:

- $\lambda x : Bool. \text{if } \underbrace{x}_{\text{ligada}} \text{ then } true \text{ else } false$
- $\lambda x : Bool. \lambda y : Bool. \text{if } true \text{ then } \underbrace{x}_{\text{ligada}} \text{ else } \underbrace{y}_{\text{ligada}}$
- $\lambda x : Bool. \text{if } \underbrace{x}_{\text{ligada}} \text{ then } true \text{ else } \underbrace{y}_{\text{libre}}$
- $(\lambda x : Bool. \text{if } \underbrace{x}_{\text{ligada}} \text{ then } true \text{ else } false) \underbrace{x}_{\text{libre}}$

La definición formal es a partir de cada término del lambda cálculo por pattern matching. FV = Free Variable

$$\begin{aligned}
FV(x) &\stackrel{\text{def}}{=} \{x\} \\
FV(true) &= FV(false) \stackrel{\text{def}}{=} \emptyset \\
FV(\text{if } M \text{ then } P \text{ else } Q) &\stackrel{\text{def}}{=} FV(M) \cup FV(P) \cup FV(Q) \\
FV(M \ N) &\stackrel{\text{def}}{=} FV(M) \cup FV(N) \\
FV(\lambda x : \sigma. M) &\stackrel{\text{def}}{=} FV(M) \setminus \{x\}
\end{aligned}$$

**Def. 1.7** (Juicio de tipado). Un **juicio de tipado** es una expresión de la forma  $\Gamma \triangleright M : \sigma$ , que se lee “El término  $M$  tiene el tipo  $\sigma$  asumiendo el contexto de tipado  $\Gamma$ ”.

Un **contexto de tipado** es un conjunto de pares  $x_i : \sigma_i$ , notado  $\{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$  donde los  $x_i$  son distintos. Usamos las letras  $\Gamma, \Delta, \dots$  para contextos de tipado.

A las variables se les anota un tipo. Uno pone las asunciones que tiene sobre el tipo de algunas variables, como  $x : Bool$ .

**Def. 1.8** (Axiomas de tipado de  $\lambda^b$ ). Son guiadas por sintaxis al igual que las variables libres

$$\frac{}{\Gamma \triangleright true : Bool}(\text{T-True}) \quad \frac{}{\Gamma \triangleright false : Bool}(\text{T-False})$$

(para cualquier contexto de tipado  $\Gamma$ )

$$\frac{x : \sigma \in \Gamma}{\Gamma \triangleright x : \sigma}(\text{T-Var})$$

$$\frac{\Gamma \triangleright M : Bool \quad \Gamma \triangleright P : \sigma \quad \Gamma \triangleright Q : \sigma}{\Gamma \triangleright \text{if } M \text{ then } P \text{ else } Q : \sigma}(\text{T-If})$$

$P$  y  $Q$  tienen que tener el mismo tipo porque queremos que la expresión siempre tipe a lo mismo.

$$\frac{\Gamma, x : \sigma \triangleright M : \tau}{\Gamma \triangleright \lambda x : \sigma. M : \sigma \rightarrow \tau}(\text{T-Abs}) \quad \frac{\Gamma \triangleright M : \sigma \rightarrow \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M \ N : \tau}(\text{T-App})$$

Si  $\Gamma \triangleright M : \sigma$  puede derivarse usando los axiomas y reglas de tipado, decimos que es **derivable**, y decimos que  $M$  es **tipable** si el juicio de tipado  $\Gamma \triangleright M : \sigma$  puede derivarse para algún  $\Gamma$  y  $\sigma$ .

Ejemplos:

- $\lambda x : Bool.x : Bool \rightarrow Bool$

$$\frac{\frac{\checkmark}{x : Bool \in \Gamma'} \quad \frac{}{\Gamma' = \Gamma \cap \{x : Bool\} \triangleright x : Bool} (T\text{-Var})}{\Gamma \triangleright \lambda x : Bool.x : Bool \rightarrow Bool} (T\text{-Abs})$$

- $\lambda x : Bool. \text{if } x \text{ then } false \text{ else } true$
- $\lambda f : Bool \rightarrow Bool. \lambda x : Bool. f \ x$
- $(\lambda f : Bool \rightarrow Bool. f \ true)(\lambda y : Bool. y)$
- $true \ (\lambda x : Bool.x)$ . No va a tipar nunca, porque  $true : Bool$  y para  $T - App$  necesitamos que sea  $\sigma \rightarrow \tau$ .
- $x \ y$ . Para usar T-App,  $x$  por sintaxis solo aplica a T-Var. La única forma de que pueda aplicar  $x$  con  $y$ ,  $x$  tiene que ser tipo flecha. Pero es una variable, entonces solo funcionaría si tenemos como asunción de tipo de  $x$  como función en  $\Gamma$ . Sin eso no se puede tipar.

### 1.2.2 Resultados básicos

*Se pueden probar por inducción en la longitud de las reglas*

**Prop. 1.1** (Unicidad de tipos). Si  $\Gamma \triangleright M : \sigma$  y  $\Gamma \triangleright M : \tau$  son derivables, entonces  $\sigma = \tau$ .

*Si una expresión tiene un tipo, ese tipo es único.*

**Prop. 1.2** (Weakening + Strengthening). Si  $\Gamma \triangleright M : \sigma$  es derivable y  $\Gamma \cap \Gamma'$  contiene a todas las variables libres de  $M$ , entonces  $\Gamma' \triangleright M : \sigma$ .

*Puedo agrandar o achicar el contexto de tipo siempre y cuando contenga las mismas variables libres.*

### 1.2.3 Semántica

Habiendo definido la sintaxis de  $\lambda^b$ , nos interesa formular como se **evalúan o ejecutan** los términos. Hay varias maneras de definir **rigurosamente** la semántica de un lenguaje de programación, nosotros vamos a definir una **semántica operacional**.

- Denotacional. Darle una *denotación* a cada símbolo del lenguaje, qué denota matemáticamente. Y uno define la semántica en términos de como las funciones van denotando cosas, con recursión o puntos fijos.
- Axiomática. Cuando cursamos algo1, definimos pre y pos condiciones. Predicados definen el significado de una operación. Las triplas de hoare y esas cosas.

- **Operacional** consiste en
  - interpretar a los **términos como estados** de una máquina abstracta, y
  - definir una **función de transición** que indica dado un estado cuál es el siguiente.

El **significado** de un término  $M$  es el estado final que alcanza la máquina al comenzar con  $M$  como estado inicial. Hay dos formas de definir semántica operacional,

- **Small-step**: la función de transición describe un paso de computación. Esta vamos a hacer nosotros.
- **Big-step** (o **Natural Semantics**): la función de transición, en un paso, evalúa el término a su resultado.

**Def. 1.9** (Juicios). La formulación se hace a través de **juicios de evaluación**  $M \rightarrow N$ , que se leen “el término  $M$  reduce, en un paso, al término  $N$ ”.

El significado de un juicio de evaluación se establece a través de:

- **Axiomas de evaluación**, que establecen que ciertos juicios de evaluación son derivables.
- **Reglas de evaluación**, que establecen que ciertos juicios de evaluación son derivables siempre y cuando ciertos otros lo sean

(análogo a axiomas y reglas de tipado)

### Semántica operacional small-step de $\lambda^b$

Además de introducir la función de transición es necesario introducir también los **valores**, los posibles resultados de evaluación de términos bien-tipados (derivables) y cerrados (sin variables libres).

Valores

$$V ::= true \mid false$$

todo término bien-tipado y cerrado de tipo Bool evalúa, en cero (directamente) o más pasos, a true o false. Se puede demostrar formalmente.

Juicio de evaluación en un paso:

$$\frac{}{\text{if } true \text{ then } M_2 \text{ else } M_3 \rightarrow M_2} \text{(E-IfTrue)}$$

$$\frac{}{\text{if } false \text{ then } M_2 \text{ else } M_3 \rightarrow M_3} \text{(E-IfFalse)}$$



$$\frac{M_1 \rightarrow M'_1}{\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \rightarrow \text{if } M'_1 \text{ then } M_2 \text{ else } M_3} \text{(E-If)}$$

Ejemplo de derivación

$\text{if } (\text{if } \text{false} \text{ then } \text{false} \text{ else } \text{true}) \text{ then } \text{false} \text{ else } \text{true}$   
 $\rightarrow_{(\text{E-If}, \text{E-IfFalse})} \text{if } \text{true} \text{ then } \text{false} \text{ else } \text{true}$   
 $\rightarrow_{(\text{E-IfTrue})} \text{false}$

**Obs. 1.1.** No existe  $M$  tal que  $\text{true} \rightarrow M$  o  $\text{false} \rightarrow M$ . No los puedo reducir más.

**Obs. 1.2.** La estrategia de evaluación corresponde con el orden habitual de los lenguajes de programación.

1. Primero evaluar la guarda del condicional.
2. Una vez que la guarda sea un valor, seguir con la expresión del then o del else, según corresponda.

Por ejemplo,

$\text{if } \text{true} \text{ then } (\text{if } \text{false} \text{ then } \text{false} \text{ else } \text{true}) \text{ else } \text{true}$   
 $\rightarrow \text{if } \text{true} \text{ then } \text{true} \text{ else } \text{true}$

y,

$\text{if } \text{true} \text{ then } (\text{if } \text{false} \text{ then } \text{false} \text{ else } \text{true}) \text{ else } \text{true}$   
 $\rightarrow \text{if } \text{false} \text{ then } \text{false} \text{ else } \text{true}$

**Lema 1.1** (Determinismo del juicio de evaluación en un paso). Si  $M \rightarrow M'$  y  $M \rightarrow M''$ , entonces  $M' = M''$ .

**Def. 1.10** (Forma normal). Una forma normal es un término que no puede reducirse o evaluarse más. i.e  $M$  tal que no existe  $N$ ,  $M \rightarrow N$ .

**Lema 1.2.** Todo valor está en forma normal.

No vale el recíproco en  $\lambda^b$ , puedo tener cosas que están en forma normal pero que no sean valores, como términos que no estén bien tipados o que no sean cerrados. Ejemplos:

- $\text{if } x \text{ then } \text{true} \text{ else } \text{false}$ : No tengo forma de reducir el  $x$ .
- $x$ . No tengo forma de reducirla pero no es ni true ni false
- $\text{true false}$ .

Pero si vale en el cálculo de las expresiones booleanas cerradas.

### Evaluación en muchos pasos

El juicio de **evaluación en muchos pasos**  $\rightarrow$  es la clausura reflexiva, transitiva de  $\rightarrow$ . Es decir, la menor relación tal que

1. Si  $M \rightarrow M'$ , entonces  $M \rightarrow M'$
2.  $M \rightarrow M$  para todo  $M$ .
3. Si  $M \rightarrow M'$  y  $M' \rightarrow M''$ , entonces  $M \rightarrow M''$ .

*captura la evolución en 0 y 1 pasos y la transitiva.*

**Ejemplo.**

$\text{if } \text{true} \text{ then } ( \text{if } \text{false} \text{ then } \text{false} \text{ else } \text{true} ) \text{ else } \text{true} \rightarrow \text{true}$

**Prop. 1.3** (Unicidad de formas normales). Si  $M \rightarrow U$  y  $M \rightarrow V$  con  $U, V$  formas normales, entonces  $U = V$

*aplicamos las reglas y llegamos a dos terminos entonces tienen que ser iguales.*

**Prop. 1.4** (Terminación). Para todo  $M$  existe una forma normal  $N$  tal que  $M \rightarrow N$ .

*no me quedo ciclando, en una cantidad finita de pasos llego a una forma normal.*

### Extendiendo semántica operacional con funciones

Valores

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

vamos a introducir una noción de evaluación tal que valgan los lemas previos y también el siguiente resultado

**Teorema 1.1.** Todo término bien tipado y cerrado de tipo

- $\text{Bool}$  evalúa, en **cero o más** pasos a true o false.
- $\sigma \rightarrow \tau$  en **cero o más pasos** a  $\lambda x : \sigma. M$ , para alguna variable  $x$  y término  $M$ .

Juicios de evaluación en un paso (Además de E-IfTrue, E-IfFalse y E-If):

$$\frac{M_1 \rightarrow M'_1}{M_1 M_2 \rightarrow M'_1 M_2} (\text{E-App1} / \mu) \quad \text{primero reducís la función}$$

$$\frac{M_2 \rightarrow M'_2}{(\lambda x : \sigma. M) M_2 \rightarrow (\lambda x : \sigma. M) M'_2} (\text{E-App2} / \nu) \quad \text{luego reducís el "argumento"}$$

$$\frac{}{(\lambda x : \sigma.M) \mathbf{V} \rightarrow M\{x \leftarrow \mathbf{V}\}} \text{(E-AppAbs / } \beta) \quad \text{primero reducís la función}$$

## Sustitución

La operación,

$$M\{x \leftarrow N\}$$

quiere decir "*Sustituir todas las ocurrencias **libres** de  $x$  en el término  $M$  por el término  $N$* ". Es una operación importante que se usa para darle semántica a la aplicación de funciones. Es sencilla de definir pero requiere cuidado en el tratamiento de los ligadores de variables ( $\lambda x$ ).

Se define por sintaxis,

$$\begin{aligned} x\{x \leftarrow N\} &\stackrel{\text{def}}{=} N \\ a\{x \leftarrow N\} &\stackrel{\text{def}}{=} a \quad \text{si } a \in \{true, false\} \cup \chi \setminus \{x\} \\ (\text{if } M \text{ then } P \text{ else } Q)\{x \leftarrow N\} &\stackrel{\text{def}}{=} \text{if } M\{x \leftarrow N\} \text{ then } P\{x \leftarrow N\} \text{ else } Q\{x \leftarrow N\} \\ (M_1 \ M_2)\{x \leftarrow N\} &\stackrel{\text{def}}{=} M_1\{x \leftarrow N\} \ M_2\{x \leftarrow N\} \\ (\lambda y : \sigma.M)\{x \leftarrow N\} &\stackrel{\text{def}}{=} \lambda y : \sigma.M\{x \leftarrow N\} \quad x \neq y, y \notin FV(N) \end{aligned}$$

1. NB: La condición  $x \neq y, y \notin FV(N)$  **siempre** puede cumplirse renombrando apropiadamente.
2. Técnicamente, la sustitución está definida sobre **clases de  $\alpha$ -equivalencia de términos**.

## $\alpha$ -equivalencia

Si en la siguiente expresión queremos sustituir la variable  $x$  por el término  $z$ ,

$$(\lambda z : \sigma.x)\{x \leftarrow z\} = \lambda z : \sigma.z$$

y lo hacemos de forma *naïve*, convertimos una función constante en la identidad. El problema es que  $\lambda z : \sigma$  capturó la ocurrencia libre de  $z$ . Pero los nombres de las variables ligadas no son relevantes, la ecuación de arriba debería ser lo mismo que

$$(\lambda w : \sigma.x)\{x \leftarrow z\} = \lambda w : \sigma.z$$

para definir la sustitución sobre aplicaciones  $(\lambda y : \sigma.M)\{x \leftarrow N\}$  vamos a asumir que la variable ligada  $y$  se renombró de forma tal que no ocurre libre en  $N$ .

**Def. 1.11** ( $\alpha$ -equivalencia). Dos términos  $M$  y  $N$  que difieren solamente en el nombre de sus variables ligadas se dicen  $\alpha$ -equivalentes. Es una relación de equivalencia.

**Ejemplo.** Ejemplos:

- $\lambda x : Bool.x =_{\alpha} \lambda y : Bool.y$
- $\lambda x : Bool.y =_{\alpha} \lambda z : Bool.y$
- $\lambda x : Bool.y \neq_{\alpha} \lambda x : Bool.z$
- $\lambda x : Bool.\lambda x : Bool.x \neq_{\alpha} \lambda y : Bool.\lambda x : Bool.y$

*La idea detrás es agrupar expresiones que sean semánticamente equivalentes.*

### Estado de error

Un **estado de error** es un término que no es un valor pero en el que la evaluación está trabada. (Un término en forma normal que no es un valor). Representa un estado en el cual el sistema de runtime en una implementación real generaría una excepción. Ejemplos:

- $\text{if } x \text{ then } M \text{ else } N$  (no es cerrado)
- $\text{true } M$  (no es tipable)

### Objetivo de un sistema de tipos

El objetivo de un sistema de tipos es garantizar la **ausencia** de estados de error.

**Def. 1.12.** Decimos que un término **termina** o que es **fuertemente normalizante** si no hay cadenas de reducciones infinitas a partir de él.

**Teorema 1.2.** Todo término bien tipado termina. Si un término cerrado está bien tipado, entonces evalúa a un valor.

Esto es lo que nos gustaría que cumpla nuestro lenguaje.

### Corrección

Decimos que **Corrección = Progreso + Preservación**.

**Def. 1.13** (Progreso). Si  $M$  es cerrado y bien tipado, entonces

1.  $M$  es un valor, o bien
2. existe  $M'$  tal que  $M \rightarrow M'$

*La evaluación no puede trabarse para términos cerrados, bien tipados que no son valores.*

**Def. 1.14** (Preservación). Si  $\Gamma \triangleright M : \sigma$  y  $M \rightarrow N$ , entonces  $\Gamma \triangleright N : \sigma$ .

*La evaluación preserva tipos.*

## 1.3 Extensiones de Cálculo Lambda

Cada vez que extendemos un lenguaje,

- Agregamos los **tipos** si hace falta,
- Extendemos los **términos**,
- Damos la **reglas de tipado**, y finalmente
- Damos la **semántica**

### 1.3.1 $\lambda^{bn}$ - Naturales

Tipos y términos

$$\sigma ::= Bool \mid Nat \mid \sigma \rightarrow \rho$$

$$M ::= \dots \mid 0 \mid succ(M) \mid pred(M) \mid iszero(M)$$

Informalmente, la semántica de los términos es:

- $succ(M)$ : Evaluar  $M$  hasta arrojar un número e incrementarlo.
- $pred(M)$ : Evaluar  $M$  hasta arrojar un número y decrementarlo.
- $iszero(M)$ : Evaluar  $M$  hasta arrojar un número, luego retornar *true* | *false* según sea cero o no.

*agregamos términos para denotar ideas nuevas.*

Axiomas y reglas de tipado:

$$\begin{array}{c} \frac{}{\Gamma \triangleright 0 : Nat} \text{(T-Zero)} \\ \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright succ(M) : Nat} \text{(T-Succ)} \quad \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright pred(M) : Nat} \text{(T-Pred)} \\ \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright iszero(M) : Bool} \text{(T-IsZero)} \end{array}$$

Valores

$$V ::= \dots \mid \underline{n} \quad \text{donde } \underline{n} \text{ abrevia } succ^n(0)$$

Juicio de evaluación en un paso

$$\begin{array}{c}
\frac{M_1 \rightarrow M'_1}{succ(M_1) \rightarrow succ(M'_1)} \text{(E-Succ)} \\
\\
\frac{}{pred(0) \rightarrow 0} \text{(E-PredZero)} \quad \frac{}{pred(\underline{n+1}) \rightarrow \underline{n}} \text{(E-PredSucc)} \\
\\
\frac{M_1 \rightarrow M'_1}{pred(M_1) \rightarrow pred(M'_1)} \text{(E-Pred)} \\
\\
\frac{}{iszero(0) \rightarrow true} \text{(E-IsZeroZero)} \quad \frac{}{iszero(\underline{n+1}) \rightarrow false} \text{(E-IsZeroSucc)} \\
\\
\frac{M_1 \rightarrow M'_1}{iszero(M_1) \rightarrow iszero(M'_1)} \text{(E-IsZero)}
\end{array}$$

Además de los juicios de evaluación de un paso de  $\lambda^b$  **Agregar referencia**

### 1.3.2 $\lambda^{\dots r}$ - Registros

Sea  $\mathcal{L}$  un conjunto de **etiquetas**, los tipos son:

$$\sigma ::= \dots \mid \{ l_i : \sigma_i^{i \in 1..n} \}$$

Ejemplos:

- $\{ nombre : String, edad : Nat \}$
- $\{ persona : \{ nombre : String, edad : Nat \}, cuil : Nat \}$
- Son posicionales, i.e

$$\{ nombre : String, edad : Nat \} \neq \{ edad : Nat, nombre : String \}$$

Términos:

$$M ::= \dots \mid \{ l_i = M_i^{i \in 1..n} \} \mid M.I$$

Informalmente, la semántica es

- El registro  $\{ l_i = M_i^{i \in 1..n} \}$  evalúa a  $\{ l_i = V_i^{i \in 1..n} \}$  con  $V_i$  el valor al que evalúa  $M_i$ .
- $M.I$  evalúa  $M$  hasta que sea un registro valor, luego proyecta el campo correspondiente.

Ejemplos,

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

$$(\lambda p : \{ edad : Nat, esMujer : Bool \}. p.edad) \quad \{ edad = 20, esMujer = false \}$$

Tipado:

$$\frac{\Gamma \triangleright M_i : \sigma_i \text{ para cada } i \in 1..n}{\Gamma \triangleright \{ l_i = M_i \}_{i \in 1..n} : \{ \sigma_i \}_{i \in 1..n}} \text{(T-Rcd)}$$

$$\frac{\Gamma \triangleright M : \{ l_i : \sigma_i \}_{i \in 1..n} \quad j \in 1..n}{\Gamma \triangleright M.l_j : \sigma_j} \text{(T-Proj)}$$

Valores:

$$V ::= \dots \mid \{ l_i = V_i \}_{i \in 1..n}$$

Semántica operacional:

$$\frac{M_j \rightarrow M'_j}{\{ l_i = V_i \}_{i \in 1..j-1}, l_j = M_j, l_i = M_i \}_{i \in j+1..n} \rightarrow \{ l_i = V_i \}_{i \in 1..j-1}, l_j = M'_j, l_i = M_i \}_{i \in j+1..n}} \text{(E-Rcd)}$$

(se reducen de izquierda a derecha)

$$\frac{j \in 1..n}{\{ l_i = V_i \}_{i \in 1..n}.l_j \rightarrow V_j} \text{(E-ProjRcd)}$$

$$\frac{M \rightarrow M'}{M.I \rightarrow M'.I} \text{(E-Proj)}$$

### 1.3.3 $\lambda^{bnu}$ - Unit

$$\sigma ::= Bool \mid Nat \mid \text{Unit} \mid \sigma \rightarrow \rho$$

$$M ::= \dots \mid unit$$

Informalmente, *Unit* es un tipo unitario y el único valor posible de una expresión de ese tipo es *unit*. Es parecido a la idea de void en lenguajes como C o Java.

Tipado:

$$\frac{}{\Gamma \triangleright unit : Unit} \text{(T-Unit)}$$

Valores:

$$V ::= \dots \mid unit$$

No hay reglas de evaluación nuevas, ya que *unit* es un valor.

Su utilidad principal es en lenguajes con efectos laterales. Porque en ellos es útil poder evaluar varias expresiones en **secuencia**,

$$M_1; M_2 \stackrel{\text{def}}{=} (\lambda x : Unit. M_2) M_1 \quad x \notin FV(M_2)$$

- La evaluación de  $M_1; M_2$  consiste en primero evaluar  $M_1$  hasta que sea un valor  $V_1$ , reemplazar las apariciones de  $x$  en  $M_2$  por  $V_1$ , y luego evaluar  $M_2$ .
- Como no hay apariciones libres de  $x$  en  $M_2$ , se evalúa  $M_1$  y luego  $M_2$ . Este comportamiento se logra con las reglas de evaluación definidas previamente.

**Agregar referencia**

### 1.3.4 Referencias

$\lambda \dots \text{let}$  - **Ligado**

$$M ::= \dots \mid \text{let } x : \sigma = M \text{ in } N$$

Informalmente,

- $\text{let } x : \sigma = M \text{ in } N$  evaluar  $M$  a un valor  $V$ , ligar  $x$  a  $V$  y evaluar  $N$ .
- Mejora la legibilidad, y la extensión **no** implica agregar nuevos tipos, es solo sintaxis.

Ejemplos,

- $\text{let } x : Nat = \underline{2} \text{ in } succ(x) \rightsquigarrow \underline{3}$
- $pred((\text{let } x : Nat = \underline{2} \text{ in } x))$
- $\text{let } f : Nat \rightarrow Nat = \lambda x : Nat. succ(n) \text{ in } f(f \ 0)$

*Como se puede ver, sirve para nombrar cosas que se usan más de una vez o dar declaratividad.*

- $\text{let } x : Nat = \underline{2} \text{ in let } x : Nat = \underline{3} \text{ in } x$



**Duda: Cual es la diferencia entre  $\text{let } x : \text{Nat} = \underline{2} \text{ in } M$  y  $(\lambda x : \text{Nat}.M) \underline{2}$ ?**

Sirve cuando lo querés usar más de una vez

Tipado,

$$\frac{\Gamma \triangleright M : \sigma_1 \quad \Gamma, x : \sigma_1 \triangleright N : \sigma_2}{\Gamma \triangleright \text{let } x : \sigma_1 = M \text{ in } N : \sigma_2} (\text{T-Let})$$

Semántica operacional,

$$\frac{M_1 \rightarrow M'_1}{\text{let } x : \sigma = M_1 \text{ in } M_2 \rightarrow \text{let } x : \sigma = M'_1 \text{ in } M_2} (\text{E-Let})$$

$$\frac{}{\text{let } x : \sigma = V_1 \text{ in } M_2 \rightarrow M_2\{x \leftarrow V_1\}} (\text{E-LetV})$$

el objetivo es llevar a  $M_1$  a un valor, y luego reemplazar las apariciones libres de  $x$  en  $M_2$  por  $V_1$ . Es parecido a la aplicación, y es como el *where* de haskell pero invertido

### Motivación

En una expresión como

$$\text{let } x : \text{Nat} = \underline{2} \text{ in } M,$$

$x$  es una variable declarada con valor 2. Pero el valor de  $x$  permanece **inalterado** a lo largo de la evaluación de  $M$ . Es **inmutable**, no existe una operación de asignación.

En programación imperativa pasa todo lo contrario, todas las variables son mutables. Por eso vamos a extender al cálculo lambda tipado con variables mutables.

### Operaciones básicas

- **Alocación** (Reserva de memoria),  $\text{ref } M$  genera una referencia fresca cuyo contenido es el valor de  $M$ .
- **Derreferenciación** (Lectura),  $!x$  sigue la referencia  $x$  y retorna su contenido.
- **Asignación**.  $x := M$  almacena en la referencia  $x$  el valor de  $M$ .

### Ejemplos

sin tipos en las let-expresiones para facilitar la lectura

- $\text{let } x = \text{ref } \underline{2} \text{ in } !x$  evalúa a  $\underline{2}$

- let  $x = \text{ref } \underline{2}$  in  $(\lambda. \text{unit}.\!x) (x := \text{succ}(\!x))$  evalúa a  $\underline{3}$   
**Por qué no así?** let  $x = \text{ref } \underline{2}$  in  $x := \text{succ}(\!x); \!x$
- let  $x = \underline{2}$  in  $x$  evalúa a  $\underline{2}$
- let  $x = \text{ref } \underline{2}$  in  $x$  evalúa a **ver**
- **aliasing**, let  $x = \text{ref } \underline{2}$  in let  $y = x$  in  $(\lambda. \text{unit}.\!x) (x := \text{succ}(\!y))$   
 $x$  e  $y$  son *alias* para la misma celda de memoria.  $\!x == \!y$ .

El término

let  $x = \text{ref } \underline{2}$  in  $x := \text{succ}(\!x)$

### Comandos

a qué evalúa? La asignación interesa por su **efecto**, y no su valor. No tiene interés preguntarse por el valor pero si tiene sentido por el efecto.

**Def. 1.15** (Comando). Vamos a definir un **Comando** como una expresión que se evalúa para causar un efecto, y a *unit* como su valor.

Un lenguaje funcional **puro** es uno en el que las expresiones son **puras** en el sentido de carecer de efectos. Este lenguaje ya no es funcional puro.

### Tipos y términos

Las **expresiones de tipos** ahora son

$$\sigma ::= \text{Bool} \mid \text{Nat} \mid \sigma \rightarrow \tau \mid \text{Unit} \mid \text{Ref } \sigma$$

Informalmente,  $\text{Ref } \sigma$  es el tipo de las referencias de valores de tipo sigma. Por ej.  $\text{Ref } (\text{Bool} \rightarrow \text{Nat})$  es el tipo de las referencias a funciones de *Bool* en *Nat*.

Términos,

$$\begin{aligned} M ::= & x \\ & \mid \lambda x : \sigma. M \\ & \mid M \ N \\ & \mid \text{unit} \\ & \mid \text{ref } M \\ & \mid \!M \\ & \mid M := N \\ & \mid \dots \end{aligned}$$

Reglas de tipado,

$$\frac{\Gamma \triangleright M_1 : \sigma}{\Gamma \triangleright \text{ref } M_1 : \text{Ref } \sigma} (\text{T-Ref})$$

$$\frac{\Gamma \triangleright M_1 : \text{Ref } \sigma}{\Gamma \triangleright !M_1 : \sigma} (\text{T-DeRef})$$

$$\frac{\Gamma \triangleright M_1 : \text{Ref } \sigma \quad \Gamma \triangleright M_2 : \sigma}{\Gamma \triangleright M_1 := M_2 : \text{Unit}} (\text{T-Assign})$$

### Semántica operacional

Qué es una referencia? Es una abstracción de una porción de memoria que se encuentra en uso. Vamos a usar **direcciones** o **locations**,  $l, l_i \in \mathcal{L}$  para representar referencias. Una **memoria** o **store**  $\mu, \mu'$  es una función parcial de **direcciones** a **valores**. Notación

- **Reescribir**:  $\mu[l \mapsto V]$  es el store resultante de **pisar**  $\mu(I)$  con  $V$ .
- **Extender**:  $\mu \oplus (l \mapsto V)$  es el **store extendido** resultante de ampliar  $\mu$  con una nueva asociación  $I \mapsto V$  (asumiendo  $I \notin \text{Dom}(\mu)$ ).

Y las reducciones ahora toman la forma

$$M \mid \mu \rightarrow M' \mid \mu'.$$

En un paso posiblemente hay un nuevo store, porque puede haber habido una operación de asignación.

La intuición de la semántica es

$$\frac{l \notin \text{Dom}(\mu)}{\text{ref } V \mid \mu \rightarrow l \mid \mu \oplus (l \mapsto V)} (\text{E-RefV})$$

el efecto de hacer un  $\text{ref } V$  es alocar una posición de memoria (meterlo en el store), creando una nueva dirección y asignándole el valor  $V$ .

Los valores posibles ahora incluyen las **direcciones**,

$$V ::= \dots \mid \text{unit} \mid \lambda x : \sigma. M \mid l$$

Dado que son un subconjunto de los términos, debemos ampliar los términos con direcciones. Estas son producto de la formalización y **no** se pretende que sean usadas por los programadores.

$$\begin{aligned}
M ::= & x \\
& | \lambda x : \sigma. M \\
& | M \ N \\
& | unit \\
& | \text{ref } M \\
& | !M \\
& | M := N \\
& | l
\end{aligned}$$

Juntar esto para que quede todos los términos juntos y redactado de una forma que la tire de una en vez de ir explorando.

### Juicios de tipado

Los valores que tienen las *locations* dependen de los valores que se almacenan en la dirección, una situación parecida a las variables libres. Entonces necesitamos un contexto de tipado para las direcciones.  $\Sigma$  va a ser una función parcial de direcciones en tipos.

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

Y las reglas de tipado,

$$\frac{\Gamma | \Sigma \triangleright M_1 : \sigma}{\Gamma | \Sigma \triangleright \text{ref } M_1 : \text{Ref } \sigma} (\text{T-Ref})$$

$$\frac{\Gamma | \Sigma \triangleright M_1 : \text{Ref } \sigma}{\Gamma | \Sigma \triangleright !M_1 : \sigma} (\text{T-DeRef})$$

$$\frac{\Gamma | \Sigma \triangleright M_1 : \text{Ref } \sigma \quad \Gamma | \Sigma \triangleright M_2 : \sigma}{\Gamma | \Sigma \triangleright M_1 := M_2 : Unit} (\text{T-Assign})$$

$$\frac{\Sigma(l) = \sigma}{\Gamma | \Sigma \triangleright l : \text{Ref } \sigma} (\text{T-Loc})$$

### Semántica operacional retomada

$$V ::= true \mid false \mid 0 \mid \underline{n} \mid unit \mid \lambda x : \sigma. M \mid l$$

Juicios de evaluación en un paso. Ahora van a tener la pinta

$$M \mid \mu \rightarrow M' \mid \mu'$$

$$\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{!M_1 \mid \mu \rightarrow !M'_1 \mid \mu'} \text{(E-Deref)} \quad \frac{\mu(l) = V}{\mu(l) = V} \text{(E-DerefLoc)} !l \mid \mu \rightarrow V \mid \mu$$

antes de hacer una desreferencia, necesito que el término llegue a un valor

$$\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{M_1 := M_2 \mid \mu \rightarrow M'_1 := M_2 \mid \mu'} \text{(E-Assign1)} \quad \frac{M_2 \mid \mu \rightarrow M'_2 \mid \mu'}{V := M_2 \mid \mu \rightarrow V := M'_2 \mid \mu'} \text{(E-Assign2)}$$

$$\frac{}{l := V \mid \mu \rightarrow unit \mid \mu[l \mapsto V]} \text{(E-Assign)}$$

$$\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{\text{ref } M_1 \mid \mu \rightarrow \text{ref } M'_1 \mid \mu'} \text{(E-Ref)} \quad \frac{l \notin \text{Dom}(\mu)}{\text{ref } V \mid \mu \rightarrow l \mid \mu \oplus (l \mapsto V)} \text{(E-RefV)}$$

M puede ser complejo, así que hasta que llegue a un valor puede haber cambiado el store.

$$\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{M_1 M_2 \mid \mu \rightarrow M'_1 M_2 \mid \mu'} \text{(E-App1)} \quad \frac{M_2 \mid \mu \rightarrow M'_2 \mid \mu'}{V_1 M_2 \mid \mu \rightarrow V_1 M'_2 \mid \mu'} \text{(E-App2)}$$

$$\frac{}{\lambda x : \sigma. M \mid V \mid \mu \rightarrow M\{x \leftarrow V\} \mid \mu} \text{(E-AppAbs)}$$

El resto de las reglas son similares, pero no modifican el store.

El store se puede modificar en el medio, por ej si tengo

$$\begin{aligned} & !\text{ref } V \mid \mu \rightarrow \text{(E-Deref, E-RefV)} !l \mid \mu \oplus (l \mapsto V) \\ & \rightarrow \text{(E-DerefLoc)} V \mid \mu \oplus (l \mapsto V) \end{aligned}$$

esto es fundamental ya que rompe la idea de funcional, en la que evolucionando un término no podía haber ningún *side effect*

### Corrección

En la [section 1.2.3 Corrección](#) se habla de corrección en sistemas de tipos. Tenemos que reformularla en el marco de *referencias*.

Para la preservación nos gustaría definirlo como

$$\Gamma \mid \Sigma \triangleright M : \sigma \text{ y } M \mid \mu \rightarrow M' \mid \mu' \text{ entonces } \Gamma \mid \Sigma \triangleright M' : \sigma$$

pero esto tiene el problema de que nada fuerza la *coordinación* entre  $\Sigma$  y  $\Gamma$ . Por ejemplo, si tenemos

- $M = !l$
- $\Gamma = \emptyset$
- $\Sigma(l) = Nat$
- $\mu(l) = true$

entonces  $\Gamma|\Sigma \triangleright M : Nat$  y  $M \mid \mu \rightarrow true \mid \mu$  **pero**  $\Gamma|\Sigma \triangleright true : Nat$  no vale.

Necesitamos un mecanismo para "**tipar**" los stores. Que haya una compatibilidad entre los stores y el contexto de tipado.

**Def. 1.16** (Compatibilidad). Vamos a decir que un store es **compatible** con un juicio de tipado,

$$\Gamma|\Sigma \triangleright \mu \text{ sii}$$

1.  $Dom(\Sigma) = Dom(\mu)$  y
2.  $\Gamma|\Sigma \triangleright \mu(l) : \Sigma(l)$  para todo  $l \in Dom(\mu)$

Reformulamos la preservación como

$$\Gamma|\Sigma \triangleright M : \sigma \text{ y } M \mid \mu \rightarrow N \mid \mu' \text{ y } \Gamma|\Sigma \triangleright \mu \text{ entonces } \Gamma|\Sigma \triangleright N : \sigma$$

esto es **casí** correcto, porque no contempla la posibilidad de que haya habido algún alloc en la reducción de M a N. Puede crecer en dominio

**Def. 1.17** (Preservación para  $\lambda^{\dots let}$ ). Si

- $\Gamma|\Sigma \triangleright M : \sigma$
- $M \mid \mu \rightarrow N \mid \mu'$  y
- $\Gamma|\Sigma \triangleright \mu$

implica que existe  $\Sigma' \supseteq \Sigma$  tal que

- $\Gamma|\Sigma' \triangleright N : \Sigma$  y
- $\Gamma|\Sigma' \triangleright \mu'$

**Def. 1.18** (Progreso para  $\lambda^{\dots let}$ ). Si  $M$  es cerrado y bien tipado ( $\emptyset|\Sigma \triangleright M : \sigma$  para algún  $\Sigma, \sigma$ ) entonces:

1.  $M$  es un valor
2. o bien para cualquier store  $\mu$  tal que  $\emptyset|\Sigma \triangleright \mu$ , existe  $M'$  y  $\mu'$  tal que  $M \mid \mu \rightarrow M' \mid \mu'$

## Ejemplos

**copiar**

succ y refs

let in

ejemplo de que no todo termina

### 1.3.5 Recursión

En programación funcional es muy común tener una función definida *recursivamente*,

$$f = f \dots f \dots f \dots$$

#### Términos y tipado

$$M ::= \dots \mid \text{fix } M$$

*fix* viene de la idea de **punto fijo**, aplicar la  $f$  varias veces.

No hacen falta nuevos tipos, pero si una regla de tipado

$$\frac{\Gamma \triangleright M : \sigma_1 \rightarrow \sigma_1}{\Gamma \triangleright \text{fix } M : \sigma_1} (\text{T-Fix})$$

$M$  no puede ser cualquier cosa, tiene que ser una función que vaya del mismo dominio a la misma imagen.

#### Semántica operacional small-step

No hay valores nuevos, pero si reglas de eval en un paso nuevas.

$$\frac{M_1 \rightarrow M'_1}{\text{fix } M_1 \rightarrow \text{fix } M'_1} (\text{E-Fix})$$

$$\frac{}{\text{fix } (\lambda f : \sigma. M) \rightarrow M \{f \leftarrow \text{fix } (\lambda f : \sigma. M)\}} (\text{E-FixBeta})$$

La aplicación normal era

$$(\lambda x : \sigma. M) M_2 \rightarrow M \{x \leftarrow M_2\}$$

pero acá el  $x$  lo reemplazamos por el mismo fix.

#### Ejemplo: Factorial

Sea el término  $M$

$$\begin{aligned}
M &= \lambda f : Nat \rightarrow Nat. \\
&\quad \lambda x : Nat. \\
&\quad \text{if } iszero(x) \text{ then } \underline{1} \text{ else } x * f(pred(x))
\end{aligned}$$

$M$  tiene tipo  $(Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat)$ . En

$$\text{let } fact : Nat \rightarrow Nat = \text{fix } M \text{ in } fact \underline{2}$$

$$\begin{aligned}
\text{fix } M &= \text{fix } \lambda f. \lambda x. \text{if } iszero(x) \text{ then } \underline{1} \text{ else } x * (f \text{ } pred(x)) \\
&\rightarrow \lambda x. \text{if } iszero(x) \text{ then } 1 \\
&\text{else } x * ([\text{fix } \lambda f. \lambda x. \text{if } iszero(x) \text{ then } \underline{1} \text{ else } x * (f \text{ } pred(x))]) \text{ } pred(x)) \\
&\rightarrow \lambda x. \text{if } iszero(x) \text{ then } 1 \\
&\text{else } x * \lambda x. \text{if } iszero(x) \text{ then } 1 \text{ else } x * ([\text{fix } \lambda f. \lambda x. \text{if } iszero(x) \text{ then } \underline{1} \text{ else } x * (f \text{ } pred(x))]) \text{ } pred(x))
\end{aligned}$$

**seguir**

### Ejemplo: Suma

Sea el término  $M$

$$\begin{aligned}
M &= \lambda s : Nat \rightarrow Nat \rightarrow Nat. \\
&\quad \lambda x : Nat. \\
&\quad \lambda y : Nat. \\
&\quad \text{if } iszero(x) \text{ then } y \text{ else } succ(s \text{ } pred(x) \text{ } y)
\end{aligned}$$

En

$$\text{let } suma = \text{fix } M \text{ in } suma \underline{2} \text{ } num3$$

### Letrec

**letrec** es una sintaxis alternativa para definir funciones recursivas,

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

Por ejemplo



```

letrec fact : Nat → Nat =
  λx : Nat. if iszero(x) then 1 else x * fact(pred(x))
in fact 3

```

no es más que *syntactic sugar*. Se puede reescribir a partir de fix:

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

## 1.4 Inferencia de tipos

Consiste en transformar términos **sin** información de tipos o con información **parcial** en terminos terminales **tipables**, con toda la información de tipos.

Para eso tenemos que *inferir* la parte de los tipos que nos faltan.

Es un beneficio, porque escribir tipos puede ser error prone y tedioso. El mecanismo que vamos a ver se realiza en tiempo de compilación y no en runtime.

Para comenzar con la inferencia vamos a denotar el conjunto de términos sin anotaciones de tipos por  $\Lambda$  (a diferencia de  $\Lambda_\tau$  que sí), que son todos los mismos pero reemplazando  $\lambda x : \sigma. M$  por  $\lambda x. M$ .

**No** van a funcionar las reglas de evaluación con  $\Lambda$ . Es importante inferir los tipos antes de hacer cualquier tipo de semántica, porque sino no tendríamos garantía de no estar evaluando cosas como *true false* que no tienen sentido.

### 1.4.1 Definición formal de inferencia

Para obtener estos tipos  $\Lambda$ , vamos a usar la función Erase.

**Def. 1.19.** (Erase) Llamamos  $\text{Erase}(\cdot)$  a la función que dado un término de LC *elimina* las anotaciones de tipos de las abstracciones.

$\text{Erase}(\cdot) : \Lambda_\tau \rightarrow \Lambda$  se define de la manera esperada.

**Ejemplo.**  $\text{Erase}(\lambda x : \text{Nat}. \lambda f : \text{Nat} \rightarrow \text{Nat}. f x) = \lambda x. \lambda f. f x$

Y así podemos definir formalmente el problema de la inferencia: Dado un término  $U$  **sin** anotaciones de tipo, hallar un término estándar (**con** anotaciones de tipo)  $M$  tal que

1.  $\Gamma \triangleright M : \sigma$  para algun  $\Gamma, \sigma$ , y
2.  $\text{Erase}(M) = U$

Ejemplos:

- Para  $U = \lambda x. \text{succ}(x)$  tomamos  $M = \lambda x : \text{Nat}. \text{succ}(x)$  (no hay otra posibilidad)

- Para  $U = \lambda x. \lambda f. f x$  obtenemos infinitos  $M_s$ ,  $M_{\sigma, \tau} = \lambda x : \sigma. \lambda f : \sigma \rightarrow \tau. f x$

Podemos encontrar muchos  $M_s$  que nos sirvan.

- Para  $U = x x$  no existe ningún término  $M$  que lo cumpla.

chequeo de tipos  $\neq$  inferencia de tipos.

El chequeo de tipos consistía en dado un término estándar  $M$  determinar si existían  $\Gamma$  y  $\sigma$  tales que  $\Gamma \triangleright M : \sigma$  es tipable. El chequeo de tipos es guiado por sintaxis, la inferencia es más compleja.

*por ejemplo en matemática, no es lo mismo demostrar un teorema que tener una demostración y verificar que sea correcta.*

### 1.4.2 Variables de tipos

#### Motivación

Para formalizar la noción de que diferentes tipos pueden ser válidos para una inferencia, por ejemplo dado

$$\lambda x. \lambda f. f (f x)$$

para cada  $\sigma$ ,  $M_\sigma$  es una solución posible

$$M_\sigma = \lambda x : \sigma. \lambda f : \sigma \rightarrow \sigma. f (f x)$$

Pero  $\sigma$  representa un tipo concreto. Para escribir una única expresión que englobe a todas, usamos **variables de tipo**:  $s$  es una variable de tipo que representa una expresión de tipos arbitraria ( $Nat, Bool, Nat \rightarrow Bool$ , etc.).

$$\lambda x : s. \lambda f : s \rightarrow s. f (f x)$$

la expresión no es una solución en si misma, pero la **sustitución** de  $s$  por cualquier expresión de tipos nos da una solución válida.

**Def. 1.20** (Variables de tipo). Extendemos las **expresiones de tipo** del LC con **variables de tipo**  $s, t, u, \dots$

$$\sigma ::= s \mid Nat \mid Bool \mid \sigma \rightarrow \tau$$

Denotamos con  $\mathcal{V}$  al conjunto de variables de tipo y  $\mathcal{T}$  al de tipos así definidos.

**Ejemplo.** Por ejemplo,

- $s \rightarrow t$
- $Nat \rightarrow Nat \rightarrow t$

- $Bool \rightarrow t$ .

### 1.4.3 Sustitución

Una **sustitución** es una función que mapea variables de tipo en expresiones de tipo. Usamos  $S, T$ , para sustituciones. Formalmente,  $S : \mathcal{V} \rightarrow \mathcal{T}$ . La idea de la sustitución es instanciar las variables de tipos en tipos concretos, por ejemplo  $Ss = Int$ .

Solo nos interesan las  $S$  tales que  $\{t \in \mathcal{V} \mid St \neq t\}$  es finito (este es un detalle técnico para que no sea infinita la sustitución)

Se pueden aplicar a,

1. Una expresión de tipos  $\sigma$  (escribimos  $S\sigma$ )
2. Un término  $M$  (escribimos  $SM$ )
3. Un contexto de tipado  $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$

$$S\Gamma \stackrel{\text{def}}{=} \{x_1 : S\sigma_1, \dots, x_n : S\sigma_n\}$$

Notaciones

- El conjunto  $\{t \mid St \neq t\}$  se llama **soporte** de  $S$ , representa las variables afectadas por  $S$ .
- Notamos  $\{\sigma_1/t_1, \dots, \sigma_n/t_n\}$  para la sustitución con soporte  $\{t_1, \dots, t_n\}$
- $Id$  es la **sustitución identidad**, con soporte  $\emptyset$  (no afecta ninguna)