Notas para final de PLP

Manuel Panichelli

June 9, 2022

Contents

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 expresion es obtener su valor (por ej. `doble 2` $\rightsquigarrow$ `4`) No toda expresion 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 expresion 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 algebráicos

```haskell
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: `succ = 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.

```haskell
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.

```haskell
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 algebráico paramétrico recursivo con dos constructores:

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

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

Pattern matching

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

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

1.1.6 No terminación y orden de evaluación

```haskell
-- 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

```haskell
-- 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]
```

```haskell
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

```haskell
-- 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) :
```

```haskell
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 (rl, rn) -> (rl + 1, rn + x)) (0, 0)
```

recr

```haskell
-- 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 (rec 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 algebrácias

```
-- 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 λ^b (lambda cálculo b de booleano) son

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

Informalmente,

- *Bool* es el tipo de los booleanos, y
- $\sigma \to \tau$ es el tipo de las funciones de tipo σ en tipo τ.

Ejemplo. Ejemplos:

- $Bool \to Bool$
- $Bool \to Bool \to Bool$

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

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

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

Ejemplo. Ejemplos:

- $\lambda x : Bool.x$ ✓
- $\lambda x : Bool.$ if x then $false$ else $true$ ✓
- $\lambda f : Bool \to Bool \to Bool.\lambda x : Bool.f\ x$ ✓
- $(\lambda f : Bool \to Bool.f\ true)(\lambda y : Bool.y)$ ✓
- $true\ (\lambda x : Bool.x)$ ✓
- $x\ y$ ✓
- $\lambda x : true$ ✗

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 ceiertos 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 λx. Caso contrario ocurre ligada.

Ejemplos:

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

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

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

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

Un **contexto de tipado** es un conjunto de pares $x_i : \sigma_i$, notado $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ donde los x_i son distintos. Usamos las letras $\Gamma, \Delta, \ldots$ 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 λ^b)**.** Son guiadas por sintaxis al igual que las variables libres

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

(para cualquier contexto de tipado Γ)

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

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

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

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

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

Ejemplos:

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

$$\dfrac{\dfrac{\dfrac{\checkmark}{x : Bool \in \Gamma'}}{\Gamma' = \Gamma \cap \{x : Bool\} \rhd x : Bool}(\text{T-Var})}{\Gamma \rhd \lambda x : Bool.x : Bool \to Bool}(\text{T-Abs})$$

- $\lambda x : Bool.$ if x then $false$ else $true$
- $\lambda f : Bool \to Bool \to Bool.\lambda x : Bool.f\ x$
- $(\lambda f : Bool \to Bool.f\ true)(\lambda y : Bool.y)$
- $true\ (\lambda x : Bool.x)$. No va a tipar nunca, porque $true : Bool$ y para $T - App$ necesitamos que sea $\sigma \to \tau$.
- $x\ y$. 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 Γ. 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 $\Gamma \rhd M : \sigma$ y $\Gamma \rhd M : \tau$ son derivables, entonces $\sigma = \tau$.

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

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

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 λ^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. Predicaque 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** $M \to N$, 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 λ^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:

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

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

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

Ejemplo de derivación

$$\begin{aligned}&\text{if } (\text{ if } false \text{ then } false \text{ else } true) \text{ then } false \text{ else } true\\ &\to_{\text{(E-If, E-IfFalse)}} \text{ if } true \text{ then } false \text{ else } true\\ &\to_{\text{(E-IfTrue)}} false\end{aligned}$$

Obs. 1.1. No existe M tal que $true \to M$ o $false \to M$. 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,

$$\begin{aligned}&\text{if } true \text{ then } (\text{ if } false \text{ then } false \text{ else } true) \text{ else } true\\ \not\to\ &\text{if } true \text{ then } true \text{ else } true\end{aligned}$$

y,

$$\begin{aligned}&\text{if } true \text{ then } (\text{ if } false \text{ then } false \text{ else } true) \text{ else } true\\ \to\ &\text{if } false \text{ then } false \text{ else } true\end{aligned}$$

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

Lema 1.2. Todo valor está en forma normal.

No vale el recíproco en λ^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:

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

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

Evaluación en muchos pasos

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

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

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

Ejemplo.

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

Prop. 1.3 (Unicidad de formas normales)**.** Si $M \twoheadrightarrow U$ y $M \twoheadrightarrow V$ 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 $M \twoheadrightarrow N$.

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

Extendiendo semántica operacional con funciones

Valores

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

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

- $Bool$ 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):

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

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

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

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 (λx).

Se define por sintaxis,

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

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 α-equivalencia de términos**.

α-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 *naive*, 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 (α-equivalencia)**.** Dos términos M y N que difieren solamente en el nombre de sus variables ligadas se dicen α-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:

- if x then M else N (no es cerrado)
- $true\ M$ (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 reduccioens 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 $M \to M'$

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

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

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 λ^{bn} - Naturales

Tipos y términos

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

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

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 $true \mid false$ según sea cero o no.

agregamos términos para denotar ideas nuevas.

Axiomas y reglas de tipado:

$$\frac{}{\Gamma \rhd 0 : Nat}\text{(T-Zero)}$$

$$\frac{\Gamma \rhd M : Nat}{\Gamma \rhd succ(M) : Nat}\text{(T-Succ)} \qquad \frac{\Gamma \rhd M : Nat}{\Gamma \rhd pred(M) : Nat}\text{(T-Pred)}$$

$$\frac{\Gamma \rhd M : Nat}{\Gamma \rhd iszero(M) : Bool}\text{(T-IsZero)}$$

Valores

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

Juicio de evaluación en un paso

$$\frac{M_1 \to M_1'}{succ(M_1) \to succ(M_1')}\text{(E-Succ)}$$

$$\frac{}{pred(0) \to 0}\text{(E-PredZero)} \qquad \frac{}{pred(\underline{n+1}) \to \underline{n}}\text{(E-PredSucc)}$$

$$\frac{M_1 \to M_1'}{pred(M_1) \to pred(M_1')}\text{(E-Pred)}$$

$$\frac{}{iszero(0) \to true}\text{(E-IsZeroZero)} \qquad \frac{}{iszero(\underline{n+1}) \to false}\text{(E-IsZeroSucc)}$$

$$\frac{M_1 \to M_1'}{iszero(M_1) \to iszero(M_1')}\text{(E-IsZero)}$$

Además de los juicios de evaluación de un paso de λ^b **Agregar referencia**

1.3.2 $\lambda^{\cdots r}$ - Registros

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

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

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:

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

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,

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

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

Tipado:

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

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

Valores:

$$V ::= \cdots \mid \{\ l_i = V_i\ ^{i \in 1..n}\ \}$$

Semántica operacional:

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

(se reducen de izquierda a derecha)

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

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

1.3.3 λ^{bnu} - Unit

$$\sigma ::= Bool \mid Nat \mid Unit \mid \sigma \to \rho$$
$$M ::= \cdots \mid unit$$

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

Tipado:

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

Valores:

$$V ::= \cdots \mid unit$$

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

$\lambda^{\ldots let}$ - **Ligado**

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

Informalmente,

- let $x : \sigma = M$ 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,

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

  Como se puede ver, sirve para nombrar cosas que se usan más de una vez o dar declarativividad.
- let $x : Nat = \underline{2}$ in let $x : Nat = \underline{3}$ in x

Duda: Cual es la diferencia entre let $x : Nat = \underline{2}$ in M y $(\lambda x : Nat.M)\ \underline{2}$?
Sirve cuando lo querés usar más de una vez

Tipado,

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

Semántica operacional,

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

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

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

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

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), 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

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

- let $x =$ ref $\underline{2}$ in $(\lambda\_ : unit.!x)\ (x := succ(!x))$ evalúa a $\underline{3}$

  Por qué no así? let $x =$ ref $\underline{2}$ in $x := succ(!x); !x$
- let $x = \underline{2}$ in x evalúa a $\underline{2}$
- let $x =$ ref $\underline{2}$ in x evalúa a **ver**
- **aliasing**, let $x =$ ref $\underline{2}$ in let $y = x$ in $(\lambda\_ : unit.!x)\ (x := succ(!y))$

  x e y son *alias* para la misma celda de memoria. $!x == !y$.

El término

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

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

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

Informalmente, Ref σ es el tipo de las referencias de valores de tipo sigma. Por ej. Ref $(Bool \to Nat)$ es el tipo de las referencias a funciones de $Bool$ en Nat.

Términos,

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

Reglas de tipado,

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

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

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

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** μ, μ' 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 μ con una nueva asociación $I \mapsto V$ (asumiendo $I \notin Dom(\mu)$).

Y las reducciones ahora toman la forma

$$M \mid \mu \to M' \mid \mu'.$$

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

La intuición de la semántica es

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

el efecto de hacer un ref V es alocar una posición de memoria (meterlo en el store), creando una nueva direccion y asignándole el valor V.

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

$$V ::= \cdots \mid unit \mid \lambda x : \sigma.M \mid l$$

Dado que son un subcojunto 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.

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

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. Σ va a ser una función parcial de direcciones en tipos.

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

Y las reglas de tipado,

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

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

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

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

Semántica operacional retomada

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

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

$$M \mid \mu \to M' \mid \mu'$$

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{!M_1 \mid \mu \to !M_1' \mid \mu'}\text{(E-Deref)} \qquad \frac{\mu(l) = V}{\mu(l) = V}\text{(E-DerefLoc)}\ !l \mid \mu \to V \mid \mu$$

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

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{M_1 := M_2 \mid \mu \to M_1' := M_2 \mid \mu'}\text{(E-Assign1)} \qquad \frac{M_2 \mid \mu \to M_2' \mid \mu'}{V := M_2 \mid \mu \to V := M_2' \mid \mu'}\text{(E-Assign2)}$$

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

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{\text{ref } M_1 \mid \mu \to \text{ref } M_1' \mid \mu'}\text{(E-Ref)} \qquad \frac{l \notin Dom(\mu)}{\text{ref } V \mid \mu \to l \mid \mu \oplus (l \mapsto V)}\text{(E-RefV)}$$

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

$$\frac{M_1 \mid \mu \to M_1' \mid \mu'}{M_1\ M_2 \mid \mu \to M_1'\ M_2 \mid \mu'}\text{(E-App1)} \qquad \frac{M_2 \mid \mu \to M_2' \mid \mu'}{V_1\ M_2 \mid \mu \to V_1\ M_2' \mid \mu'}\text{(E-App2)}$$

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

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

> El store se puede modificar en el medio, por ej si tengo
>
> $$!\text{ref } V \mid \mu \to_{\text{(E-Deref, E-RefV)}} !l \mid \mu \oplus (l \mapsto V) \to_{\text{(E-DerefLoc)}} V \mid \mu \oplus (l \mapsto V)$$
>
> 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*.

> **P**ara la preservación nos gustaría definirlo como
>
> $$\Gamma|\Sigma \rhd M : \sigma \text{ y } M \mid \mu \to M' \mid \mu' \text{ entonces } \Gamma|\Sigma \rhd M' : \sigma$$
>
> pero esto tiene el problema de que nada fuerza la *coordinación* entre Σ y Γ. Por ejemplo, si tenemos

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

entonces $\Gamma|\Sigma \rhd M : Nat$ y $M \mid \mu \to true \mid \mu$ **pero** $\Gamma|\Sigma \rhd true : Nat$ 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,

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

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

Reformulamos la preservación como

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

esto es casi 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 $\lambda^{\cdots let}$)**.** Si

- $\Gamma|\Sigma \rhd M : \sigma$
- $M \mid \mu \to N \mid \mu'$ y
- $\Gamma|\Sigma \rhd \mu$

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

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

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

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

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*,

$$f = f \ldots f \ldots f \ldots$$

Términos y tipado

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

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

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

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

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.

$$\frac{M_1 \to M_1'}{\text{fix } M_1 \to \text{fix } M_1'}(\text{E-Fix})$$

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

La aplicación normal era

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

pero acá el x lo reemplazamos por el mismo fix.

Ejemplo: Factorial

Sea el término M

$$M = \lambda f : Nat \to Nat.$$
$$\lambda x : Nat.$$
$$\text{if } iszero(x) \text{ then } \underline{1} \text{ else } x * f(pred(x))$$

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

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

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

seguir

Ejemplo: Suma

Sea el término M

$$M = \lambda s : Nat \to Nat \to Nat.$$
$$\lambda x : Nat.$$
$$\lambda y : Nat.$$
$$\text{if } iszero(x) \text{ then } y \text{ else } succ(s\ pred(x)\ y)$$

En

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

Letrec

letrec es una sintaxis alternativa para definir funciones recursivas,

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

Por ejemplo

$$
\begin{array}{l}
\text{letrec } fact : Nat \to Nat = \\
\quad \lambda x : Nat.\ \text{if } iszero(x) \text{ then } \underline{1} \text{ else } x * fact(pred(x)) \\
\text{in } fact\ \underline{3}
\end{array}
$$

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

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

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 Λ (a diferencia de Λ_τ que sí), que son todos los mismos pero reemplazando $\lambda x : \sigma.M$ por $\lambda x.M$.

> **N**o van a funcionar las reglas de evaluación con Λ. 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 Λ, vamos a usar la función Erase.

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

Erase$(\cdot) : \Lambda_\tau \to \Lambda$ se define de la manera esperada.

Ejemplo. Erase$(\lambda x : Nat.\lambda f : Nat \to Nat.f\ x) = \lambda x.\lambda f.f\ x$

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. $\Gamma \rhd M : \sigma$ para algun Γ, σ, y
2. Erase$(M) = U$

Ejemplos:

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

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

  Podemos encontrar muchos Ms 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 Γ y σ tales que $\Gamma \rhd M : \sigma$ 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 σ, M_σ es una solución posible

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

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

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

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** $s, t, u, \ldots$

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

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

Ejemplo. Por ejemplo,

- $s \to t$
- $Nat \to Nat \to t$

- $Bool \to t$.

1.4.3 Sustitución

Una **sustitución** es una función que mapea variables de tipo en expresiones de tipo. Usamos S, T, para sustituciones. Formalmente, $S : \mathcal{V} \to \mathcal{T}$. 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 σ (escribimos $S\sigma$)
2. Un término M (escribimos SM)
3. Un contexto de tipado $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$

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

Notaciones

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

1.4.4 Instancia de un juicio de tipado

Def. 1.21 (Instancia de un juicio de tipado)**.** Un juicio de tipado $\Gamma' \rhd M' : \sigma'$ es una **instancia** de $\Gamma \rhd M : \sigma$ si existe una sustitución de tipos S tal que

$$\Gamma' \supseteq S\Gamma, M' = SM \text{ y } \sigma' = S\sigma$$

Es agarrar y aplicar la sust S a todas las partes. G' puede ser un poco más grande, tiene que incluir.

Prop. 1.5. Si $\Gamma \rhd M : \sigma$ es derivable, entonces cualquier instancia del mismo también lo es.

1.4.5 Función de inferencia $\mathbb{W}(\cdot)$

Vamos a definir una función $\mathbb{W}(\cdot)$ que dado un término U sin anotaciones de tipo, verifica

- **Corrección**: $\mathbb{W}(U) = \Gamma \rhd M : \sigma$ implica

- Erase$(M) = U$ y
- $\Gamma \rhd M : \sigma$ es derivable

Dado el U nos devuelve algo que chequea tipos y al borrarle los tipos nos devuelve U. I.e es correcto

- **Completitud**: Si $\Gamma \rhd M : \sigma$ es derivable y Erase$(M) = U$ entonces
  - $\mathbb{W}(U)$ tiene éxito, y
  - produce un juicio de tipado $\Gamma' \rhd M' : \sigma'$ tal que $\Gamma \rhd M : \sigma$ es instancia de él. *Se dice que $\mathbb{W}(\cdot)$ computa un **tipo principal**, el más general. Tal vez no instancia algunas cosas.*

Si existe ese M, queremos que el algoritmo de $\mathbb{W}(\cdot)$ ande. No necesariamente lo encuentra exactamente, pero al menos produce una instancia.

Tal vez sabemos que existe un juicio que tipa a un término como $Nat \to Nat$, pero en realidad se puede tipar como algo más general, como $s \to s$. El algoritmo debería devolver el más general y no la instancia.

1.4.6 Unificación

Motivación

Queremos definir $\mathbb{W}(succ(U)) \stackrel{\text{def}}{=}$. Suponemos que $\mathbb{W}(U) = \Gamma \rhd M : \tau$, y queremos saber si τ puede ser un Nat, si son *compatibles*.

Idea: yo tengo dos expr de tipos, y quiero ver si son compatibles, las puedo ver como iguales. Para eso tengo que sustituir adecuadamente. Si encuentro una sustitución que me da que son iguales, entonces son compatibles.

El proceso de determinar si existe una sustitución S tal que dos expresiones de tipos σ, τ son unificables (i.e $S\sigma = S\tau$) se llama **unificación**. Por ejemplo,

- El tipo $s \to t$ es **compatible** o **unificable** con $Nat \to u$? Si, porque podemos tomar la sustitución $S \stackrel{\text{def}}{=} \{Nat\{s\}, u\{t\}\}$.

  Observar que $S(s \to t) = Nat \to u = S(Nat \to u)$ ya que S no afecta Nat ni u.
- El tipo s es **unificable** con Nat? Si, la sust S anterior es tal que $Ss = SNat$.

Composición de sustituciones

Def. 1.22 (Composicion de sustituciones)**.** La **composición** de S y T, denotada $S \circ T$, es la sustitución que comporta

$$(S \circ T)(\sigma) = ST\sigma$$

Ejemplo 1.1. Sea $S = \{u \to Bool/t, Nat/s\}$ y $T = \{v \times Nat/u, Nat/s\}$ entonces $S \circ T = \{(v \times Nat) \to Bool/t, v \times Nat/u, Nat/s\}$

Decimos que $S = T$ si tienen el mismo soporte (variables afectadas) y $St = Tt$ para todo t en su soporte.

Prop. 1.6. Algunas props,

- $S \circ Id = Id \circ S = S$
- $S \circ (T \circ U) = (S \circ T) \circ U$ (asociatividad)
- La conmutatividad no vale.

Def. 1.23. Una sustitución S es **más general** que T si existe U tal que $T = U \circ S$.

La idea es que S es más general que T porque T se obtiene instanciando S.

Ejemplo 1.2. $S = \{t \to t/u\}$ es más general que $T = \{Nat \to Nat/u\}$, porque tomando $U = \{Nat/t\}$, $T = U \circ S$.

Lo que nos va a interesar en el algortimo es obtener el tipo o sustitución más general.

MGU

Def. 1.24 (Ecuación de unificación)**.** Una **ecuación de unificación** es una expresión de la forma $\sigma_1 \doteq \sigma_2$. Una sustitución S es la **solución** de conjunto de ecuaciones de unificación $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$

si $S\sigma_1 = S\sigma'_1, \ldots, S\sigma_n = S\sigma'_n$

Vamos a escribir ecuaciones con reestriciones que quiero que se cumplan.

Ejemplo 1.3. Ejemplos,

- La sust $\{Bool/v, Bool \times Nat/u\}$ es solución de $\{v \times Nat \to Nat \doteq u \to Nat\}$ (reemplaza de ambos lados)
- $\{Bool \times Bool/v, (Bool \times Bool) \times Nat/u\}$ también
- $\{v \times Nat/u\}$ también, y es más general.
- $\{Nat \to s \doteq t \times u\}$ **no** tiene solución.
- $\{u \to Nat \doteq u\}$ no tiene solución. (Occurs check)

Def. 1.25 (MGU)**.** Una sustitución S es un **MGU** (Unificador más general, most general unifier) de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ si

1. Es solución de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$,
2. Es más general que cualquier otra solución de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$.

Ejemplo 1.4. La sustitución $\{Bool/v, Bool \times Nat/u\}$ solución de $\{v \times Nat \to Nat \doteq u \to Nat\}$ pero no es un MGU porque es instancia de $\{v \times Nat/u\}$, que si es MGU.

Algoritmo de unificación

Teorema 1.3. Si $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$ tiene sol, existe un MGU y es único salvo renombre de variables.

El algoritmo va a tener,

- **Entrada**: Conjunto de ecuaciones de unificación $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$
- **Salida**:
  - MGU S de $\{\sigma_1 \doteq \sigma'_1, \ldots, \sigma_n \doteq \sigma'_n\}$, si tiene solución
  - falla en caso contrario.

Algoritmo de Martelli-Montanari

Es un algoritmo no determinístico, porque tiene un montón de **reglas de simplificación** que podemos aplicar en cualquier orden. Estas simplifican conjuntos de pares de tipos a unificar (los goals)

$$G_0 \mapsto G_1 \mapsto \ldots \mapsto G_n$$

termina cuando llegamos al goal vacío. Algunos pasos de simplificación usan una sustitución que representa una *solución parcial* al problema,

$$G_0 \mapsto G_1 \mapsto_{S_1} G_2 \mapsto \ldots \mapsto_{S_k} G_n$$

Si la secuencia es exitosa, el MGU es $S_k \circ \ldots \circ S_1$

Reglas:

σ, τ tipos concretos y s var de tipo

1. **Descomposición**

   $\{\sigma_1 \to \sigma_2 \doteq \tau_1 \to \tau_2\} \cup G \mapsto \{\sigma_1 \doteq \tau_1, \sigma_2 \doteq \tau_2\} \cup G$

   $\{Nat \doteq Nat\} \cup G \mapsto G$

   $\{Bool \doteq Bool\} \cup G \mapsto G$

2. **Eliminación de par trivial**

   $\{s \doteq s\} \cup G \mapsto G$

3. **Swap**: si σ no es una variable,

   $\{\sigma \doteq s\} \cup G \mapsto \{s \doteq \sigma\} \cup G$

4. **Eliminación de variable**: si $s \notin FV(\sigma)$,

   $\{s \doteq \sigma\} \cup G \mapsto_{\{\sigma/s\}} \{\sigma/s\}G$

5. **Colisión**

   $\{\sigma \doteq \tau\} \cup G \mapsto$ falla

   con $(\sigma, \tau) \in T \cup T^{-1}$ y $T = \{(Bool, Nat), (Nat, \sigma_1 \to \sigma_2), (Bool, \sigma_1 \to \sigma_2)\}$

   (T^{-1} es invertir los pares)

6. **Occur check**: si $s \neq \sigma$ y $s \in FV(\sigma)$

   $\{s \doteq \sigma\} \cup G \mapsto$ falla

Para la regla de descomposición de Nat y Nat, Bool y Bool podrían surgir por ej. de una descomposición de una func.

$$\{Nat \to t \doteq Nat \to s\} \mapsto \{Nat \doteq Nat, s \doteq t\} \mapsto \{s \doteq t\}$$

Ejemplo de secuencia exitosa

$$\begin{aligned}
& \{(Nat \to r) \to (r \to u) \doteq t \to (s \to s) \to t\} \\
\mapsto^1 \quad & \{Nat \to r \doteq t, r \to u \doteq (s \to s) \to t\} \\
\mapsto^3 \quad & \{t \doteq Nat \to r, r \to u \doteq (s \to s) \to t\} \\
\mapsto^4_{Nat \to r/t} \quad & \{r \to u \doteq (s \to s) \to (Nat \to r)\} \\
\mapsto^1 \quad & \{r \doteq s \to s, u \doteq Nat \to r\} \\
\mapsto^4_{s \to s/r} \quad & \{u \doteq Nat \to (s \to s)\} \\
\mapsto^4_{Nat \to (s \to s)/u} \quad & \emptyset
\end{aligned}$$

El MGU es

$$\begin{aligned}
& \{Nat \to (s \to s)/u\} \circ \{s \to s/r\} \circ \{Nat \to r/t\} \\
= \; & \{Nat \to (s \to s)/t, s \to s/r, Nat \to (s \to s)/u\}
\end{aligned}$$

Ejemplo de secuencia fallida

$$\begin{aligned}
& \{r \to (s \to r) \doteq s \to ((r \to Nat) \to r)\} \\
\mapsto^1 \quad & \{r \doteq s, s \to r \doteq (r \to Nat) \to r\} \\
\mapsto^4_{s/r} \quad & \{s \to s \doteq (s \to Nat) \to s\} \\
\mapsto^1 \quad & \{s \doteq s \to Nat, s \doteq s\} \\
\mapsto^6 \quad & \text{falla}
\end{aligned}$$

Teorema 1.4. El algoritmo de Martelli-Montanari siempre termina. Y sea G un conjunto de pares,

- si G tiene un unificador, el algoritmo termina exitosamente y retorna un MGU
- Si no tiene unificador, el algoritmo termina con falla

1.4.7 Algoritmo de inferencia

Términos simples

$$\mathbb{W}(0) \stackrel{\text{def}}{=} \emptyset \rhd 0 : Nat$$
$$\mathbb{W}(true) \stackrel{\text{def}}{=} \emptyset \rhd true : Bool$$
$$\mathbb{W}(false) \stackrel{\text{def}}{=} \emptyset \rhd false : Bool$$
$$\mathbb{W}(x) \stackrel{\text{def}}{=} \{x : s\} \rhd x : s, \text{ con s variable fresca}$$

Inferencia de succ

Sea $\mathbb{W}(U) = \Gamma \rhd M : \tau$, $S = \mathsf{MGU}(\{\tau \doteq Nat\})$,

$$\mathbb{W}(succ(U)) \stackrel{\text{def}}{=} S\Gamma \rhd S\ succ(M) : Nat$$

La idea para términos complejos es aplicar recursivamente el algoritmo para los sub-términos, y luego unificar los resultados.

similar para pred

Inferencia de iszero

Sea $\mathbb{W}(U) = \Gamma \rhd M : \tau$, $S = \mathsf{MGU}(\{\tau \doteq Nat\})$

$$\mathbb{W}(iszero(U)) \stackrel{\text{def}}{=} S\Gamma \rhd S\ iszero(M) : Bool$$

Inferencia de IfThenElse

Sean

- $\mathbb{W}(U) = \Gamma_1 \rhd M : \rho$
- $\mathbb{W}(V) = \Gamma_2 \rhd P : \sigma$
- $\mathbb{W}(W) = \Gamma_3 \rhd Q : \tau$
- $S = \mathsf{MGU}(\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i \text{ y } x : \sigma_2 \in \Gamma_j, i \neq j\} \cup \{\sigma \doteq \tau, \rho \doteq Bool\})$

Entonces

$$\mathbb{W}(\text{ if } U \text{ then } V \text{ else } W) \stackrel{\text{def}}{=}$$
$$S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_3 \rhd S(\text{ if } M \text{ then } P \text{ else } Q) : S\sigma$$

En un if, V y W tenían que tener el mismo tipo ($\sigma \doteq \tau$) y U tiene que ser algo compatible con bool ($\rho \doteq Bool$).

Además, Tenemos que pedir que los contextos de unificacion sean compatibles. Por ej. si tenemos $x : Bool \in \Gamma_1$ y $x : s \in \Gamma_2$, podemos hacer $Bool/s$. Pero si tenemos cosas no compatibles como $x : Bool \in \Gamma_1$ y $x : Nat \in \Gamma_2$, no hay nada que se pueda hacer.

Para el tipo del if en el juicio, podría haber elegido tanto $S\sigma$ como $S\tau$ porque son iguales. Elegimos σ forma arbitraria.

Inferencia de aplicación

Sean

- $\mathbb{W}(U) = \Gamma_1 \rhd M : \tau$
- $\mathbb{W}(V) = \Gamma_2 \rhd N : \rho$
- $S = \mathsf{MGU}(\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1 \text{ y } x : \sigma_2 \in \Gamma_2\} \cup \{\tau \doteq \rho \to t\})$, con t una variable fresca

entonces

$$\mathbb{W}(U\ V) \stackrel{\mathsf{def}}{=} S\Gamma_1 \cup S\Gamma_2 \rhd S(M\ N) : St$$

Inferencia de abstracción

Sea $\mathbb{W}(U) = \Gamma \rhd M : \rho$. Tenemos dos posibilidades,

- Si el contexto de tipado tiene información sobre x (i.e $x : \tau \in \Gamma$ para algún τ), entonces

  $$\mathbb{W}(\lambda x.U) \stackrel{\mathsf{def}}{=} \Gamma \setminus \{x : \tau\} \rhd \lambda x : \tau.M : \tau \to \rho$$

  lo sacamos de Γ porque los contextos de tipado son para variables libres, y a x la vamos a ligar.

- Si no tiene (i.e $x \notin \mathsf{Dom}(\Gamma)$) elegimos una variable fresca s y

  $$\mathbb{W}(\lambda x.U) \stackrel{\mathsf{def}}{=} \Gamma \rhd \lambda x : s.M : s \to \rho$$

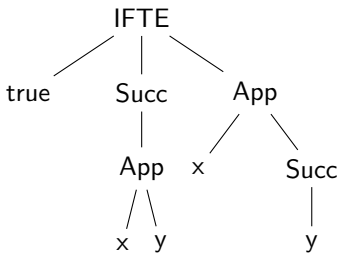
Inferencia de fix

Sean $\mathbb{W}(U) = \Gamma \rhd M : \tau$, $S = \mathsf{MGU}(\{\tau \doteq t \to t\})$ con t una variable fresca. Entonces,

$$\mathbb{W}(\mathsf{fix}\ U) \stackrel{\mathsf{def}}{=} S\Gamma \rhd S\mathsf{fix}\ (M) : St$$

Ejemplo de inferencia

Veamos la inferencia de if $true$ then $succ(x\ y)$ else $x\ (succ(y))$. Para ello es más fácil pensarlo sobre el AST, ya que el algoritmo es recursivo sobre la estructura sintáctica del término.



En la clase se ejecuta el algoritmo bottom-up, cuando yo creo que sería mucho más claro hacerlo top-down.

- $\mathbb{W}(true) = \emptyset \rhd true : Bool$
- $succ(x\ y)$

$$\begin{aligned}
\mathbb{W}(x) &= \{x : s\} \rhd x : s \\
\mathbb{W}(y) &= \{y : t\} \rhd y : t \\
\mathbb{W}(x\ y) &= \{x : t \to r, y : t\} \rhd x\ y : r \\
&\quad \text{donde } S = \mathsf{MGU}(\{s \doteq t \to r\}) = \{t \to r\{s\}\} \\
\mathbb{W}(succ(x\ y)) &= \{x : t \to Nat, y : t\} \rhd succ(x\ y) : Nat \\
&\quad \text{donde } S = \mathsf{MGU}(\{r \doteq Nat\}) = \{Nat/r\}
\end{aligned}$$

- $x\ (succ(y))$

$$\begin{aligned}
\mathbb{W}(y) &= \{y : v\} \rhd y : v \\
\mathbb{W}(succ(y)) &= \{y : Nat\} \rhd succ(y) : Nat \\
&\quad \text{donde } S = \mathsf{MGU}(\{v \doteq Nat\}) = \{Nat/v\} \\
\mathbb{W}(x) &= \{x : u\} \rhd x : u \\
\mathbb{W}(x\ succ(y)) &= \{x : Nat \to w, y : Nat\} \rhd x\ succ(y) : w \\
&\quad \text{donde } S = \mathsf{MGU}(\{u \doteq Nat \to w\}) = \{Nat \to w/u\}
\end{aligned}$$

- $M =$ if $true$ then $succ(x\ y)$ else $x\ (succ(y))$

$$\begin{aligned}
\mathbb{W}(true) &= \emptyset \rhd true : Bool \\
\mathbb{W}(succ(x\ y)) &= \{x : t \to Nat, y : t\} \rhd succ(x\ y) : Nat \\
\mathbb{W}(x\ succ(y)) &= \{x : Nat \to w, y : Nat\} \rhd x\ succ(y) : w \\
\mathbb{W}(M) &= \{x : Nat \to Nat, y : Nat\} \rhd M : Nat \\
&\quad \text{donde } S = \text{MGU}(\{t \to Nat \doteq Nat \to w, t \doteq Nat, Nat \doteq w\}) \\
&\quad = \{Nat/t, Nat/w\}
\end{aligned}$$

  $t \to Nat \doteq Nat \to w$, $t \doteq Nat$ salen de unificar contextos y $Nat \doteq w$ de que el tipo de ambas partes del if tiene que ser igual

Ejemplo de falla

$$M = \text{ if } true \text{ then } x\ \underline{2} \text{ else } x\ true$$

$$\begin{aligned}
\mathbb{W}(x) &= \{x : s\} \rhd x : s \\
\mathbb{W}(\underline{2}) &= \emptyset \rhd \underline{2} : Nat \\
\mathbb{W}(x\ \underline{2}) &= \{x : Nat \to t\} \rhd x\ \underline{2} : t
\end{aligned}$$

$$\begin{aligned}
\mathbb{W}(x) &= \{x : u\} \rhd x : u \\
\mathbb{W}(true) &= \emptyset \rhd true : Bool \\
\mathbb{W}(x\ true) &= \{x : Bool \to v\} \rhd x\ true : v
\end{aligned}$$

$$\mathbb{W}(M) = \text{falla}$$

porque no existe el MGU$(\{Nat \to t \doteq Bool \to v\})$.

1.5 Subtipado

Diego: lo más relevante es el contexto, algunos detalles no son tan importantes.

Motivación

El sistema de tipos que estudiamos descarta programas incorrectos, pero también algunos "buenos"

$$(\lambda x : \{\ a : Nat\ \}.x.a)\ \{\ a = 1,\ b = true\ \}$$

este término no tipa porque tiene que ser exactamente $\{\ a : Nat\ \}$, pero no tendría nada de malo que tenga cosas demás.

Cuando uno tipa de alguna manera lo que quiere evitar es darle semántica a cosas que no tienen sentido, pero estamos rechazando cosas que en runtime andarían bien.

1.5.1 Principio de sustitutividad

Es una relación de dos tipos,

$$\sigma <: \tau$$

Se lee como "En todo contexto donde se espera una expresión de tipo τ, puede usarse una de tipo σ en su lugar **sin** que ello genere un error".

Otras formas de verlo:

- Podemos decir que σ es un **subtipo** de τ y τ **supertipo** de σ.
- Todo valor descripto por σ también es descripto por τ
- Los elementos de σ son un subconjunto de los elementos de τ.

Y se refleja con una nueva regla de tipado llamada **subsumption**

$$\frac{\Gamma \rhd M : \sigma \quad \sigma <: \tau}{\Gamma \rhd M : \tau}\text{(T-Subs)}$$

Ejemplos:

- $M : Nat$ y se que $Nat <: Int$ puedo ver a $M : Int$.
- $M : Perro$ y se que $Perro <: Animal$ (perro es un subtipo de animal), entonces puedo ver a $M : Animal$.

Antes nuestro sistema de tipos estaba guiado por sintaxis, pero ahora tenemos algo que no necesariamente viene de ahí.

1.5.2 Reglas de subtipado

Para los tipos base asumimos que nos informan de que manera están relacionados, son como reglas *axiomáticas*

$$\frac{}{Nat <: Float}\text{(S-NatFloat)} \quad \frac{}{Int <: Float}\text{(S-IntFloat)}$$

$$\frac{}{Bool <: Nat}\text{(S-BoolNat)}$$

porque Bool se puede tomar como 0 y 1

Además, podemos definir un pre orden a partir de la relación de subtipado

$$\frac{}{\sigma <: \sigma}\text{(S-Refl)} \qquad \frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}\text{(S-Trans)}$$

por ej.

$$\frac{\dfrac{}{Nat <: Int}\checkmark \quad \dfrac{}{Int <: Float}\checkmark}{Nat <: Float}\text{(S-Trans)}$$

1.5.3 Subtipado en $\lambda^{\cdots r}$

(ver subsection 1.3.2 $\lambda^{\cdots r}$ - Registros)

Subtipado a lo ancho

$$\frac{}{\{\ l_i : \sigma_i \mid i \in 1..n{+}k\ \} <: \{\ l_i : \sigma_i \mid i \in 1..n\ \}}\text{(S-RcdWidth)}$$

Ejemplo

$$\{\ nombre : String,\ edad : Int\ \} <: \{\ nombre : String\ \}$$

si tengo un registro y lo extiendo con más cosas lo tomo como subtipo

Rcd es por ReCorD

Observaciones,

- $\sigma <: \{\}$ para todo σ record (cualquiera es más grande).
- No hay tipo τ tal que $\tau <: \sigma$ para todo σ, siempre puedo tomar un σ que tenga algún elemento de un tipo diferente. *En general va a haber alguno que sea más chico que todos, a menos que lo definamos a mano.*

Subtipado a lo profundo

$$\frac{\sigma_i <: \tau_i \quad i \in I = \{1..n\}}{\{\ l_i : \sigma_i\ \}_{i \in I} <: \{\ l_i : \tau_i\ \}_{i \in I}}\text{(S-RcdDepth)}$$

Ejemplo

$$\{\ a : Nat,\ b : Int\ \} <: \{\ a : Float,\ b : Int\ \}$$

Ejemplo

$$\frac{\dfrac{}{\{a : Nat,\ b : Nat\} <: \{a : Nat\}}\text{(S-RcdWidth)} \quad \dfrac{}{\{m : Nat\} <: \{\}}\text{(S-RcdWidth)}}{\begin{array}{c}\{\ x : \{a : Nat,\ b : Nat\}, y : \{m : Nat\}\ \} \\ <: \{\ x : \{a : Nat\}, y : \{\}\ \}\end{array}}\text{(S-RcdDepth)}$$

Permutaciones de campos

Los registros son descripciones de campos, y no deberían depender del orden dado. Hasta ahora, $\{ a : Int,\ b : String \} \neq \{ b : String,\ a : Int \}$ pero querríamos que $\{ a : Int,\ b : String \} <: \{ b : String,\ a : Int \}$ (y también vale al revés)

$$\frac{\{ k_j : \sigma_j \mid j \in 1..n \} \text{ es permutación de } \{ l_i : \tau_i \mid i \in 1..n \}}{\{ k_j : \sigma_j \mid j \in 1..n \} <: \{ l_i : \tau_i \mid i \in 1..n \}} \text{(S-RcdPerm)}$$

S-RcdPerm combinado con S-RcdWidth y S-Trans puede eliminar campos en cualquier parte de un registro, generando un supertipo (reordenando para que queden últimas las que quiero eliminar)

Regla combinada

$$\frac{\{ l_i \mid i \in 1..n \} \subseteq \{ k_j \mid j \in 1..m \} \quad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{ k_j : \sigma_j \mid j \in 1..n \} <: \{ l_i : \tau_i \mid i \in 1..m \}} \text{(S-Rcd)}$$

n y m permiten Width, que cuando coinciden sean subtipos Depth y $\subseteq$ Perm (además hace implícito que $m \geq n$)

1.5.4 Subtipado de funciones

Si tengo una función que espera recibir números positivos, entonces le voy a estar pasando todos los positivos. Si me pasan una que además se banca más, entonces mejor, no la voy a romper. Pero si me pasan una que se banca menos que los positivos, la voy a romper con algunos argumentos.

Una buena forma de pensar intuitivamente la covarianza y contravarianza es hacer ejemplos con registros, que tienen la interpretación más intuitiva de subtipado.

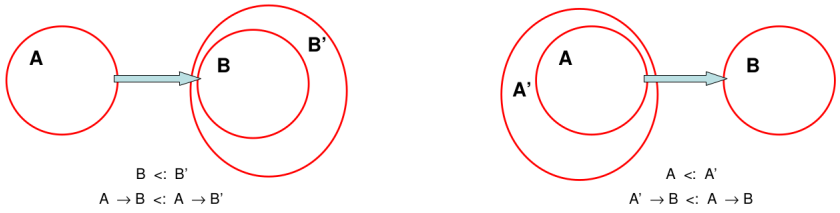


Figure 1.1: Esquema de Perla para entender el subtipado de funciones

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{(S-Arrow | S-Func)}$$

Para reemplazar una función por otra, tiene que

- Bancarse todos los argumentos, o más (contravarianza)
- El resultado tiene que ser reemplazable (covarianza)

Se dice que el constructor de tipos de función es **contravariante** en el primer argumento (dominio) y **variante** en el segundo (imagen).

1.5.5 Subtipado de referencias

Ref es **invariante**, solo se comparan referencias de tipos equivalentes

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\text{Ref } \sigma <: \text{Ref } \tau}(\text{S-Ref})$$

no hace falta que sean iguales, como con los registros y las permutaciones.

Justificación

Es ref covariante?

$$\frac{\sigma <: \tau}{\text{Ref } \sigma <: \text{Ref } \tau}(\text{S-Ref})$$

Si tengo r : Ref Nat y hago !r, que le puedo pasar? Y la escritura?

Si fuera covariante, uno esperaría que como $Int <: Float$, Ref $Int <:$ Ref $Float$. Pero si tengo

```
let r = ref 3  // r : Ref Int
in
    r := 2.1;  // Ref Int <: Ref Float, T-Sub, r: Ref Float
    !r
    // r: Int
    // Pero 2.1 no es int!
```

se rompe con la asignación.

(el 2 es un int que lo puedo ver como 2.0, float, pero el 2.1 es un float y no lo puedo ver como int)

Es ref contravariante?

$$\frac{\sigma <: \tau}{\text{Ref } \tau <: \text{Ref } \sigma}(\text{S-Ref})$$

Como $Int <: Float$ y suponemos ref contravariante, Ref $Float <:$ Ref Int

```
let r = ref 2.1 // r: Ref Float
in
    !r
    // Por Ref Float <: Ref Int y T-Sub, r: Ref Int
    // r: Int
    // Pero 2.1 no es Int!
```

Refinado de Ref

Para permitir algún tipo de subtipado, se agregan nuevas clases referencias de solo lectura y solo escritura. Source σ de lectura y Sink σ de escritura.

$$\frac{\Gamma|\Sigma \rhd M : \text{Source } \sigma}{\Gamma|\Sigma \rhd !M : \sigma} \quad \frac{\Gamma|\Sigma \rhd M : \text{Sink } \sigma \quad \Gamma|\Sigma \rhd N : \sigma}{\Gamma|\Sigma \rhd M := N : Unit}$$

- Source (lectura) es covariante

$$\frac{\sigma <: \tau}{\text{Source } \sigma <: \text{Source } \tau}(\text{S-Source}) \quad \frac{Int <: Float}{\text{Source } Int <: \text{Source } Float}$$

  $!r$ puede verse como $Float$ aunque r sea de tipo Source Int.

  Si tengo un ref 3 y lo desreferencio, puedo verlo como int o float 3.0

```
let r = ref 3
in
    !r  // por Source Int <: Source Float
:: Float
```

  Si espero leer una ref a T, puedo esperar una ref a un tipo más bajo, más informativo.

- Sink (escritura) es contravariante.

$$\frac{\tau <: \sigma}{\text{Source } \sigma <: \text{Source } \tau}(\text{S-Sink}) \quad \frac{Int <: Float}{\text{Sink } Float <: \text{Sink } Int}$$

  Por ejemplo,

```
let r = ref 2.1
in
    r := 3; // Usando Sink Float <: Sink Int
    !r
:: Float
```

Puedo coercionar 3 a 3.0 y no se rompe nada.

Si espero escribir sobre una Ref a T, puedo esperar una Ref a un tipo más alto, menos informativo.

Se pueden relacionar con Ref,

$$\frac{}{\text{Ref } \tau <: \text{Source } \tau}(\text{S-RefSource}) \qquad \frac{}{\text{Ref } \tau <: \text{Sink } \tau}(\text{S-RefSink})$$

1.5.6 Algoritmo

Muy lindo todo, pero como lo usamos para tipar? Hasta ahora nuestras reglas eran dirigidas por sintaxis, por lo que es inmediato implementar un algoritmo de chequeo de tipos a partir de ellas.

Pero con T-Subs, está guiada por la *oportunidad del tipo*, la podés aplicar cuando quieras como convenga. Esto hace que **no sea evidente como implementar un algoritmo** de chequeo de tipos a partir de las reglas, que no sea determinístico. Nos podemos sacar de encima el problema?

Propuesta N°1 de sistema: cambio en la regla de aplicación

La única regla en la que realmente hace falta subtipar es en la aplicación. Definimos una variante del sistema de tipado dirigida por sintaxis y la notamos con $\mapsto$.

$$\frac{\Gamma \mapsto M : \sigma \to \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M\ N : \tau}(\text{T-App})$$

(y cambiando el resto del sistema para que use el tipado alternativo $\mapsto$)

Prop. 1.7. Se puede probar por inducción en las reglas de tipado que

1. $\Gamma \mapsto M : \sigma$ implica que $\Gamma \rhd M : \sigma$
2. $\Gamma \rhd M : \sigma$ implica que existe τ tal que $\Gamma \mapsto M : \tau$ con $\tau <: \sigma$.

Pero nos falta cubrir como implementar la relación $<:$. (recordar subsection 1.5.2 Reglas de subtipado y S-Rcd). Las reglas S-Refl y S-Trans no están guiadas por la sintaxis. Para sacarlas,

Diego: Lo que sigue ahora es algo anecdótico

- S-Refl está para resolver cosas de la pinta $Nat <: Nat$, entonces podemos remover la necesidad de la reflexividad agregando axiomas para cada tipo: $Nat <: Nat$, $Float <: Float$, $Bool <: Bool$.

  Esto se puede probar en general pero no lo probamos

- Con un argumento similar, se puede demostrar que no es necesaria S-Trans agregando todas las combinaciones.

Sacando esas dos reglas, la parte de subtipado puede ser guiada por sintaxis? Si, el algoritmo es el siguiente

```
subtype(S, T) {
    // Si es un axioma, sabemos que si. Sino,
    // func
    if S == S1 -> S2 and T == T1 -> T2:
        return subtype(T1, S2) and subtype(S2, T2)

    // reg
    if S == {kj: Sj, j in 1..m} and T == {li: Ti, i in 1..n}:
        return {li, i in 1..n} subseteq {kj, j in 1..m} and
            forall i exists j kj = li and subtype(Sj, Ti)

    return false
}
```

Chapter 2

Paradigma de objetos

El modelo de cómputo que está detrás de POO, también puede caer diseño orientado a objetos pero es mucho más complejo. Acá vamos a modelar una parte chiquita.

2.1 Programación Orientada a Objetos

Conceptos y metáfora

- Todo programa es una simulación, y cada entidad del sistema siendo simulado se representa en el programa a través de un **objeto**
- Los objetos son la forma de abstraer un concepto físico o conceptual del mundo real
- El modelo de cómputo consiste en envío de mensajes: un sistema está formado por objetos que *colaboran* entre sí mediante mensajes.
- Los **mensajes** son solicitudes para que un objeto lleve a cabo operación. El **receptor** (el objeto que lo recibe) decide como llevarla a cabo, cuya implementación está descripta por un **método**.
- El conjunto de mensajes que responde un objeto se denomina **interfaz** o **protocolo**.
- Los objetos pueden tener **estado** interno que altere el comportamiento de los métodos. Se representa a través de un conjunto de **colaboradores internos** (también llamados **atributos** o **variables instancia**)

Ejemplo:

```
unRectangulo
    interfaz: area
    atributos: alto y ancho
    método: area = function() { return alto * ancho }
```

La única manera de interactuar con un objeto es a través de su protocolo. Su implementación no puede depender de detalles de implementación de otros objetos (principio heredado de TADs)

Def (Principio de ocultamiento de la información)**.** El estado de un objeto es **privado** y solamente puede ser consultado o modificado por sus métodos. *(No todos los lenguajes imponen esta restricción)*

2.1.1 Method dispatch

Cómo hacemos por atrás para saber qué método de un objeto ejecutar cuando le llega un mensaje? El proceso que establece la asociación mensaje-método a ejecutar se llama **method dispatch**.

Si se hace en tiempo de *compilación* (se puede determinar a partir del código fuente) es **method dispatch estático**. En cambio, si se hace en runtime es **method dispatch dinámico.**

2.1.2 Corrientes

Quien es responsabile de conocer los métodos de los objetos? Hay dos alternativas conocidas: **clasificación** y **prototipado**

2.1.3 Clasificación

Es la más mainstream. Las clases modelan **conceptos abstractos** del dominio de problema. Definen el comportamiento y la forma de un conjunto de objetos que instancian (sus **instancias**). Todo objeto es instancia de alguna clase. Son templates que tienen métodos, atributos y después se usan para instanciar objetos concretos.

Tienen

- Nombre
- Definición de variables de instancia
- Métodos de instancia. Por cada uno nombre, parámetros y cuerpo.

clase Point

Variables de instancia 'xCoord' e 'yCoord'
Métodos

```
x
   ^xCoord
y
   ^yCoord

dist: aPoint
"Answer the distance between aPoint and the receiver."
   | dx dy |
   dx := aPoint x - xCoord.
   dy := aPoint y - yCoord.
   ^ (dx * dx + (dy * dy)) sqrt
```

Figure 2.1: Ejemplo de clase en sintaxis de Smalltalk

clase INode

Métodos de clase

```
l: leftchild r:rightchild
    "Creates an interior node"
    ...
```

Vars. de instancia 'left right'
Métodos de instancia

```
sum
    ^ left sum + right sum
```

clase Leaf

Métodos de clase

```
new: anInteger
    "Creates a leaf"
    ...
```

Vars. de instancia 'value'
Métodos de instancia

```
sum
    ^value
```

Ejemplos

```
1) Leaf new: 5
2) (INode l: (Leaf new: 3) r: (Leaf new: 4)) sum
```

Figure 2.2: Ejemplo de clase Node en sintaxis de Smalltalk

Self y super

self es una pseudovariable que durante la ejecución de un método referencia al receptor del mensaje. Se liga automáticamente y no puede ser asignada.

clase INode

Métodos de clase

```
l: leftchild r:rightchild
    "Creates an interior node"
    ...
```

Var. de instancia 'left right'

Métodos de instancia

```
l
    ^left
r
    ^right
sum
    ^ (self l) sum + (self r) sum
```

Figure 2.3: Ejemplo uso de self en Smalltalk

super es otra pseudovariable que referencia al objeto que recibe el mensaje. Cambia el proceso de activación al momento del envío de un mensaje.

Una expresión de la forma `super msg` que aparece en el cuerpo de un método `m` provoca que el **method lookup** se haga desde el padre de la clase anfitriona de `m`.

```
Object subclass: #C1
Métodos de instancia
m1
  ^self m2
m2
  ^13
```

```
C1 subclass: #C2
Métodos de instancia
m1
  ^22
m2
  ^23
m3
  ^super m1
```

```
C2 subclass: #C3
Métodos de instancia
m1
  ^32
m2
  ^33
```

Sigamos la ejecución de

```
(C2 new) m3. ----¿ Qué valor devuelve?   23
(C3 new) m3. ----¿ Qué valor devuelve?   33
```

Figure 2.4: Ejemplo `super` y `self`

Jerarquía de clases

Es común que nuevas clases se definan como *extensiones* de clases existentes, agregando o cambiando el comportamiento de algunos métodos y agregando nuevas variables de instancia o clase. Por eso, una clase puede **heredar de** o **extender** de

una clase existente (llamada **superclase**). La transitividad de esta relación induce nociones de **ancestros** y **descendientes**.

Hay dos tipos

- **Simple**: una clase tiene un único padre (salvo la raíz). Esta es la que usan la mayoría de los lenguajes OO.
- **Múltiple**: una clase puede tener más de una clase padre.

  Complica el proceso de method dispatch, ya que si tengo un método m definido en más de una superclase, cual uso? Hay dos soluciones posibles:

  – Establecer un *orden de búsqueda* sobre las superclases

  – Pedir que se *redefinan* en la clase nueva todos los métodos que estén en más de una clase padre.

```
Object subclass: #Point
instanceVarNames: 'xCoord yCoord'

Métodos de clase
x: p1 y: p2
    ^self new setX: p1 setY: p2

Métodos de instancia
x
    ^xCoord
y
    ^yCoord

setX: xValue setY:yValue
   xCoord :=  xValue.
   yCoord :=  yValue.
```

```
Point subclass: #ColorPoint
instanceVarNames: 'color'

Métodos de clase
x: p1 y: p2 color: aColor
    |instance|
    instance := self x: p1 y: p2.
    instance color: aColor.
    ^instance


Métodos de instancia
color: aColor
    color := aColor

color
    ^color
```

USO

```
ColorPoint x: 10 y: 20 color: red.
```

Figure 2.5: Ejemplo de herencia en Smalltalk

2.1.4 Prototipado

Construye instancias concretas que se interpretan como representantes canónicos de instancias (llamados prototipos). El resto de las instancias se generan clonando los prototipos (de forma shallow). Y los clones se pueden cambiar.

Ejemplo de creado de objeto en js

```
let celda = {
    contenido: 0,
    get: function() { return this.contenido; },
```

```
    set: function(n) { this.contenido = n; }
}

// Generar objetos
Celda = function() {
    this.contenido = 0;
    this.get = function() { return this.contenido; };
    this.set = function(n) { this.contenido = n; };
}

otracelda = new Celda();
```

2.2 Cálculo de objetos

Acá vamos a hacer semántica big step en vez de small step. De un gran paso llegás al valor. Vamos a ver un cálculo de objetos no tipado basado en Abadi y Cardelli, 98 que se llama ς calculo (sigma pero una sigma cheta)

Ingredientes

- Los objetos son la única estructura computacional
- Los objetos son registros que tienen métodos como atributos (un campo normal va a ser un método que devuelve siempre lo mismo)
- Cada método tiene una única variable ligada que representa a `self` (o `this`) y un cuerpo que produce el resultado, que puede depender o no de self.
- Proveen dos operaciones: envío de mensaje (invocar un método) o redefinición de un método.

2.2.1 Sintaxis

$$
\begin{array}{lll}
o, b ::= & x & \textit{variable} \\
& \mid [\ l_i = \varsigma(x_i) b_i\ ^{i \in 1..n}\ ] & \textit{objeto} \\
& \mid o.l & \textit{selección / envío de mensaje} \\
& \mid o.l \Leftarrow \varsigma(x) b & \textit{redefinición de método}
\end{array}
$$

Ejemplo.

$$o \stackrel{\text{def}}{=} [\ l_1 = \varsigma(x_1)[\ ],\ l_2 = \varsigma(x_2) x_2.l_1\ ]$$

- l_1 retorna el objeto vacío
- l_2 envía el mensaje l_1 a self (representado por el parámetro x_2)

2.2.2 Atributos vs métodos

El cálculo ς no incluye explícitamente atributos (campos), sino que se representan como métodos que no usan al parámetro self. De esta manera, el envío de un mensaje representa también a la selección de un atributo y la redefinición de un método representa también la asignación de un atributo

Como abusos de notación, vamos a

- Usar $[\ \ldots, l = b, \ldots\ ]$ en vez de $l = \varsigma(x)b$ cuando no se usa x en b,
- Usar $o.l := b$ en vez de $o.l \Leftarrow \varsigma(x)b$ cuando x no se usa en b.

2.2.3 Variables libres

ς es un ligador del parámetro self x_i en el cuerpo b_i de la expresión $\varsigma(x_i)b_i$.

Esto es similar a lc, probablemente en semántica vamos a tener que hacer una sustitución para ligarla.

Def. 2.1 (Variables libres)**.**

$$
\begin{aligned}
\mathsf{fv}(\varsigma(x)b) &= \mathsf{fv}(b) \setminus \{x\} \\
\mathsf{fv}(x) &= \{x\} \\
\mathsf{fv}([\ l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}\ ]) &= \bigcup_{i\in 1..n} \mathsf{fv}(\varsigma(x_i)b_i) \\
\mathsf{fv}(o.l) &= \mathsf{fv}(o) \\
\mathsf{fv}(o.l \Leftarrow \varsigma(x)b) &= \mathsf{fv}(o.l) \cup \mathsf{fv}(\varsigma(x)b)
\end{aligned}
$$

Un término o es **cerrado** si $\mathsf{fv}(o) = \emptyset$.

2.2.4 Sustitución

$$
\begin{aligned}
x\{c/x\} &= c \\
y\{c/x\} &= y \\
([\ l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}\ ])\{c/x\} &= [\ l_i = (\varsigma(x_i)b_i)\{c/x\}^{i\in 1..n}\ ] \\
(o.l)\{c/x\} &= o\{x/x\}.l \\
o.l \Leftarrow \varsigma(y)b\{c/x\} &= (o\{c/x\}).l \Leftarrow ((\varsigma(y)b)\{c/x\}) \\
(\varsigma(y)b)\{c/x\} &= \varsigma(y')b\{y'/y\}\{c/x\} \\
&\qquad \text{con } y' \notin \mathsf{fv}(\varsigma(y)b) \cup \mathsf{fv}(c) \cup \{x\}
\end{aligned}
$$

Acomodamos las cosas para que no haya interferencias en la sustitución como clashing de nombres (primer reemplazo) y una vez que tengamos eso, aplicamos la sustitución que queremos.

2.2.5 Equivalencia de términos ($\equiv$)

Los términos $\varsigma(x)b$ y $\varsigma(y)(b\{y/x\})$ con $y \notin \text{fv}(b)$ se consideran equivalentes (α-conversión).

También, dos objetos que difieren en el orden de sus componentes son considerados equivalentes.

$$[\ l_1 = \varsigma(x_1)[\ ],\ l_2 = \varsigma(x_2)x_2.l_1\ ] \equiv [\ l_2 = \varsigma(x_3)x_3.l_1,\ l_1 = \varsigma(x_1)[\ ]\ ]$$

2.2.6 Semántica operacional

> **E**n Ic, las reducciones llevaban de $M \to M'$ y eventualmente aplicando muchas reducciones llegábamos a un valor. Acá vamos a llegar en una sola.

Los valores van a ser objetos

$$v ::= [\ l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}\ ]$$

Y vamos a aplicar reducciones **big-step** $\longrightarrow$, que en un paso de reducción pasan de una expresión a un valor.

$$\frac{}{v \longrightarrow v}\text{[Obj]}$$

$$\frac{o \longrightarrow v' \quad v' \equiv [\ l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}\ ] \quad b_j\{v'/x_j\} \longrightarrow v \quad j \in 1..n}{o.l_j \longrightarrow v}\text{[Sel]}$$

$$\frac{o \longrightarrow [\ l_i = \varsigma(x_i)b_i\ ^{i\in 1..n}\ ] \quad j \in 1..n}{o.l_j \Leftarrow \varsigma(x)b \longrightarrow [\ l_j = \varsigma(x)b,\ l_i = \varsigma(x_i)b_i^{i\in 1..n-\{j\}}\ ]}\text{[Upd]}$$

En Sel reduzco o hasta un valor v' para saber quien es self, ligo self en b_j que es el cuerpo del método correspondiente a l_j, y el resultado es reducir eso a un valor

Ejemplo. Reducción de $o = [\ a = [],\ l = \varsigma(x)x.a\ ].l$

$$\frac{\underbrace{[\ a = [],\ l = \varsigma(x)x.a\ ]}_{o}.l \longrightarrow [] \qquad \overbrace{(x.a)\{o/x\}}^{=o.a} \longrightarrow []}{\dfrac{o \longrightarrow o}{}\text{[Obj]}}\text{[Sel]}$$

$$\frac{o \longrightarrow o}{}\text{[Obj]} \qquad \frac{\dfrac{o \longrightarrow o}{}\text{[Obj]} \qquad \overbrace{[]\{o/x\}}^{=[]} \longrightarrow []}{}\text{[Sel]}$$

Ejemplo. Reducción de $([\ a = [],\ l = \varsigma(x)x.a\ ].l \Leftarrow \varsigma(y)[]).l$

$$\dfrac{\dfrac{\dfrac{}{o \longrightarrow o}[\text{Obj}]}{\underbrace{u \longrightarrow [\ a = [],\ l = [\,]\ ]}_{o}}[\text{Upd}] \qquad \overbrace{[]\{u/x\}}^{=[]} \longrightarrow [\,]}{\underbrace{([\ a = [],\ l = \varsigma(x)x.a\ ].l \Leftarrow \varsigma(y)[]).l}_{u} \longrightarrow [\,]}[\text{Sel}]}[\text{Obj}]$$

Ejemplo. Podemos tener problemas, por ej. el siguiente tiene una reducción infinita $\underbrace{[\ a = \varsigma(x)x.a\ ]}_{o}.a$

es como una llamada recursiva, pero sin caso base. La evaluación de esta expresión se indefine. Es análogo a fix $\lambda x : \sigma.x$.

$$\dfrac{\dfrac{}{o \longrightarrow o}[\text{Obj}] \qquad \overbrace{x.a\{o/x\}}^{= o.a} \longrightarrow}{o.a \longrightarrow}[\text{Sel}]}[\text{Sel}]$$

2.2.7 Ejemplo: Naturales

Vamos a asumir que existen los objetos `true` y `false` que corresponden a booleanos.

> **S**e podrían definir estilo Smalltalk, y tener métodos `ifTrue` e `ifFalse` que según el objeto ejecuten lo que le pasás o no.

$$\begin{aligned}
\texttt{zero} \stackrel{\text{def}}{=} [\ & \mathit{iszero} = \texttt{true}, \\
& \mathit{pred} = \varsigma(x)x, \\
& \mathit{succ} = \varsigma(x)(x.\mathit{iszero} := \texttt{false}).\mathit{pred} := x\ ] \\
\texttt{uno} \stackrel{\text{def}}{=}\ & \texttt{zero}.\mathit{succ} \\
& \longrightarrow \underbrace{[\ \mathit{iszero} = \texttt{false}, \mathit{pred} = \texttt{zero}, \mathit{succ} = \ldots\ ]}_{\texttt{uno}'} \\
\texttt{dos} \stackrel{\text{def}}{=}\ & \texttt{zero}.\mathit{succ}.\mathit{succ} \\
& \longrightarrow [\ \mathit{iszero} = \texttt{false}, \mathit{pred} = \texttt{uno}', \mathit{succ} = \ldots\ ]
\end{aligned}$$

2.2.8 Codificando calculo λ

Podemos simular el cálculo λ no tipado,

$$M ::= M\ N \mid \lambda x.M \mid x$$

Queremos hacer esto porque los métodos del cálculo sigma no tienen argumentos, solo self. Idea intuitiva:

- Representar las funciones como objetos

$$[\ arg = \ldots,\ val = \ldots\ ].$$

- Al aplicarlas, primero se asigna el valor del argumento al atributo arg y luego se envía el mensaje val que evalúa el cuerpo de la función
- De esa forma, una evaluación (o aplicación) $(f\ v)$ se traduce en $(o_f.arg := o_v).val$

Son esencialmente *method objects* de ing1

Vamos a definir una función $[\![\cdot]\!] : M \to a$ que dado un término nos da el objeto que lo codifica.

$$\begin{aligned}
[\![x]\!] &\stackrel{\text{def}}{=} x \\
[\![M\ N]\!] &\stackrel{\text{def}}{=} ([\![M]\!].arg := [\![N]\!]).val \\
[\![\lambda x.M]\!] &\stackrel{\text{def}}{=} [\ val = \varsigma(y)[\![M]\!]\{y.arg/x\}, \\
&\qquad\quad arg = \varsigma(y)y.arg\ ] \\
&\qquad\quad \text{con } y \notin \mathsf{fv}(M)
\end{aligned}$$

en $\lambda x.M\ M$ tiene apariciones libres de x, entonces las reemplazamos por $y.arg$ para que funcione bien la semántica de la aplicación. En la codificación de la función, arg lo dejamos inicialmente como algo indefinido, porque no tiene sentido que tenga nada real asignado ya que siempre lo vamos a reemplazar.

Si hacemos $[\![\lambda x.x]\!].val$, se cuelga.

Ejemplo 2.1. Ejemplos de codificación

$$\begin{aligned}
[\![\lambda x.x]\!] &\stackrel{\text{def}}{=} [\ val = \varsigma(y)[\![x]\!]\{y.arg/x\},\ arg = \varsigma(y)y.arg\ ] \\
&= [\ val = \varsigma(y)x\{y.arg/x\},\ arg = \varsigma(y)y.arg\ ] \\
&= [\ val = \varsigma(y)y.arg,\ arg = \varsigma(y)y.arg\ ]
\end{aligned}$$

$$\begin{aligned}
[\![(\lambda x.x)\ M]\!] &\stackrel{\text{def}}{=} ([\![\lambda x.x]\!].arg := [\![M]\!]).val \\
&= [\ val = \varsigma(y)y.arg,\ arg = \varsigma(y)y.arg\ ].arg := [\![M]\!].val \\
&\quad \longrightarrow [\![M]\!] \\
&\quad \text{siempre que } [\![M]\!] \text{ sea un objeto}
\end{aligned}$$

Métodos con parámetros

Usando este truquito de codificación podemos extender a nuestros métodos para que reciban parámetros. Un método que espera un parámetro es un método cuya definición codifica a una función

$$\varsigma(y)[\![\lambda x.M]\!]$$

como notación, vamos a escribir

- $\lambda(x)M$ en vez de $[\![\lambda x.M]\!]$
- $M(N)$ en vez de $[\![M\ N]\!]$.

Ejemplo: PUnto en el plano

Lo definimos arrancando en el origen de coordenadas pero puede ser desplazado

$$\begin{aligned}\texttt{origen} \stackrel{\text{def}}{=} [\ & x = 0,\\ & y = 0,\\ & mv\_x = \varsigma(p)\lambda(d_x)p.x := p.x + d_x,\\ & mv\_y = \varsigma(p)\lambda(d_y)p.y := p.y + d_y\ ]\end{aligned}$$

$$\texttt{unidad} \stackrel{\text{def}}{=} \texttt{origen}.mv\_x(1).mv\_y(1)$$

2.2.9 Codificación de clases

Partimos de una def de objetos que tenia métodos. Le dimos semántica, agregamos un encoding (no cambió la semántica) de parámetros, y ahora vamos a hacer otro encoding para **generadores de objetos** como los que vimos en prototipado pero ahora clases

(Stateless) Trait

Un **trait** es una colección de ciertos métodos. Los stateless son un conjunto particular de los traits que no tienen estado: no especifican variables ni estado ni acceden al estado (i.e self). Una clase de construye a partir de traits, a veces se usan para interfaces.

Ejemplo 2.2. Ejemplo de trait

$$\begin{aligned}\texttt{CompT} \stackrel{\text{def}}{=} [\ & eq = \varsigma(t)\lambda(x)\lambda(y)(x.comp(y)) == 0,\\ & lt = \varsigma(t)\lambda(x)\lambda(y)(x.comp(y)) < 0\ ]\end{aligned}$$

Observar que en el cuerpo de los métodos eq y lt, no se usa t (self).

Los podemos pensar como una colección de **pre-métodos**: algo que eventualmente será un método

- Un pre método es $\varsigma(t)\lambda(y)b$ con $t \notin \text{fv}(\lambda(y)b)$ (i.e no usan self)
- En este caso por notación al ser un atributo podíamos omitir el $\varsigma(t)$ y escribir directamente $\lambda(y)b$, por lo que los traits pasarían a ser

$$\mathtt{t} = [\ l_i = \lambda(y_i)b_i\ ^{i\in 1..n}\ ]$$

A partir de un trait $\mathtt{t} = [\ l_i = \lambda(y_i)b_i\ ^{i\in 1..n}\ ]$ podemos definir un constructor de objetos (cuando t es completo, tiene todos los métodos que necesita)

$$new \stackrel{\text{def}}{=} \lambda(z)[\ l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}\ ]$$

$$\begin{aligned} o &\stackrel{\text{def}}{=} new(t) \\ &\approx [\ l_i = \varsigma(s)t.l_i(s)^{i\in 1..n}\ ] \\ &\approx [\ l_i = \varsigma(y_i)b_i\ ^{i\in 1..n}\ ] \end{aligned}$$

new aprovecha el trait para crear un método real (uno en el que el self importa). Probablemente el y_i sea lo que querramos usar después como self.

Ejemplo. Ejemplo de new

$$\begin{aligned} \mathtt{CompT} &\stackrel{\text{def}}{=} [\ eq = \varsigma(t)\lambda(x)\lambda(y)(x.comp(y)) == 0, \\ &\qquad lt = \varsigma(t)\lambda(x)\lambda(y)(x.comp(y)) < 0\ ] \\ new &\stackrel{\text{def}}{=} \lambda(z)[\ l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}\ ] \\ new(\mathtt{CompT}) &\approx [\ eq = \varsigma(s)\mathtt{CompT}.eq(s), \\ &\qquad lt = \varsigma(s)\mathtt{CompT}.lt(s)\ ] \\ &\approx [\ eq = \varsigma(x)\lambda(y)(x.comp(y)) == 0, \\ &\qquad lt = \varsigma(x)\lambda(y)(x.comp(y)) < 0\ ] \end{aligned}$$

Acá el objeto creado es inutilizable, porque `CompT` usa $comp$ que no es un método que define (no es completo).

Clases

Una **clase** va a ser un *trait* stateless (completo) que además provea un método new.

$$\mathsf{c} \stackrel{\mathsf{def}}{=} [\ new = \lambda(z)[\ l_i = \varsigma(s)z.l_i(s)^{\,i\in 1..n}\ ],\\ l_i = \lambda(s)b_i\ ^{\,i\in 1..n}\ ]$$

Luego,

$$o \stackrel{\mathsf{def}}{=} \mathsf{c}.new\\ \longrightarrow [\ l_i = \varsigma(s)c.l_i(s)^{\,i\in 1..n}\ ]\\ \approx [\ l_i = \varsigma(s)b_i\ ^{\,i\in 1..n}\ ]$$

Ejemplo. Clase Contador

$$\texttt{Contador} \stackrel{\mathsf{def}}{=} [\ new = \lambda(z)[\quad v = \varsigma(s)z.v(s),\\ inc = \varsigma(s)z.inc(s),\\ get = \varsigma(s)z.get(s)\ ],\\ v = \lambda(s)0,\\ inc = \lambda(s)s.v := s.v + 1,\\ get = \lambda(s)s.v\ ]$$

Herencia

Si tenemos una clase

$$\mathsf{c} \stackrel{\mathsf{def}}{=} [\ new = \lambda(z)[\ l_i = \varsigma(s)z.l_i(s)^{\,i\in 1..n}\ ],\\ l_i = \lambda(s)b_i\ ^{\,i\in 1..n}\ ]$$

Queremos definir c′ como subclase de c que agregue los pre-métodos $\lambda(s)b_k^{k\in n+1..n+m}$. Alcanza con agregar los métodos nuevos, y para los viejos referenciar a la superclase.

$$\mathsf{c} \stackrel{\mathsf{def}}{=} [\ new = \lambda(z)[\ l_i = \varsigma(s)z.l_i(s)^{\,i\in 1..n+m}\ ],\\ l_j = c.l_j^{\,j\in 1..n},\\ l_k = \lambda(s)b_k^{k\in n+1..n+m}\ ]$$

también si quisiéramos podríamos redefinir pre-métodos. En vez de delegar a la superclase, los definimos y ya.

Chapter 3

Paradigma lógico

Hasta ahora, vimos

- **Imperativo**: tenemos un estado compuesto por variables que se va modificando con las instrucciones, al final determinando un estado final.
- **Funcional**: tenemos expresiones que se van reduciendo hasta una forma que si todo sale bien es un valor
- **Objetos**: objetos como una forma de modelar la realidad, que colaboran mediante mensajes. Los métodos se implementan de forma imperativa, pero no es lo fundamental del paradigma. Eso es la colaboración mediante mensajes.

Se basa en el uso de la lógica como forma de programar. En vez de pensar en un algoritmo dado un problema, nos mantenemos en la misma "esfera". En vez de programar un problema, quedarnos en el *qué*.

Vamos a especificar **hechos** y **reglas de inferencia** y un **goal** (objetivo) a probar. Un motor de inferencia trata de probar que el goal es consecuencia de los hechos y reglas.

Es **declarativo**: se especifican hechos, reglas y goals sin indicar *cómo* se obtiene el último a partir de los primeros.

3.1 Prolog

Vamos a usar **Prolog**:

- Los programas se escriben en un subconjunto de la logica de primer orden (cláusulas de Horn)
- El mecanismo teórico en el que se basa es el **método de resolución** (forma de dada una fórmula ver si es SAT o UNSAT)

- Para motivarlo, primero vamos a ver como es resolución en lógica proposicional

Ejemplo. Ejemplo de programa de prolog

```
% Hechos
habla(ale, ruso).
habla(juan, ingles).
habla(maria, ruso).
habla(maria, ingles).

% Reglas de inferencia
seComunicaCon(X, Y) :- habla(X, L), habla(Y, L), X \= Y.

% Ejemplo de goal
seComunicaCon(X, ale)
% X = maria
seComunicaCon(X, Y)
% X = maria, Y = juan y le podemos pedir más
```

Depende de como instancie un goal, puede funcionar distinto

3.2 Resolución para Lógica Proposicional

3.2.1 Lógica proposicional

Sintaxis

Dado un conjunto $\mathcal{V}$ de **variables proposicionales**, podemos definir inductivamente al conjunto de **fórmulas proposicionales** (o **proposiciones**) **Prop** de la siguiente manera,

1. Una variable proposicional $P_0, P_1, \ldots$ es una proposición
2. Si A, B son proposiciones, entonces también lo son
   - $\neg A$ (negación)
   - $A \wedge B$ (conjunción)
   - $A \vee B$ (disyunción)
   - $A \supset B$ (implicación)
   - $A \iff B$

Ejemplo. Ejemplos de fórmulas: $A \vee \neg B, A \wedge B \supset A \vee B$

Semántica

Una **valuación** es una función $v : \mathcal{V} \to \{T, F\}$ que asigna valores de verdad a las variables proposicionales. **Satisface** una proposición A si $v \models A$ donde,

$$
\begin{aligned}
v \models P &\text{ sii } v(P) = T \\
v \models \neg A &\text{ sii } v \not\models A \text{ (i.e no } v \models A) \\
v \models A \vee B &\text{ sii } v \models A \text{ o } v \models B \\
v \models A \wedge B &\text{ sii } v \models A \text{ y } v \models B \\
v \models A \supset B &\text{ sii } v \not\models A \text{ o } v \models B \\
v \models A \iff B &\text{ sii } (v \models A \text{ sii } v \models B)
\end{aligned}
$$

Tautologías y satisfacibilidad

Una proposición A es

- una **tautología** si $v \models A$ para toda valuación v (ej: $P \vee \neg P$)
- **satisfacible** si existe una valuación v tal que $v \models A$
- **insatisfacible** si no es satisfacible

Se puede extender a un conjunto. Un conjunto de proposiciones S es

- **satisfacible** si existe una valuación v tal que para todo $A \in S$, $v \models A$. (otra forma de verlo: hace verdadera a la conjunción de todas)
- **insatisfacible** si no es satisfacible

Ejemplo. Ejemplos de tautologías

- $A \supset A$
- $\neg\neg A \supset A$
- $(A \supset B) \iff (\neg B \supset \neg A)$ (contrarecíproco)

Ejemplo. Ejemplos de proposiciones insatisfacible

- $(\neg A \vee B) \wedge (\neg A \vee \neg B) \wedge A$
- $(A \supset B) \wedge A \wedge \neg B$

Una forma de probar tautologías es con una tabla de verdad. Sin importar los valores de las variables proposicionales, tiene que dar verdadero. Para probar que algo es insat, lo mismo pero viendo que todo da falso.

Teorema 3.1. Una proposición A es una tautología sii $\neg A$ es insatisfacible.

Dem. Por la ida y la vuelta

$\Rightarrow$) Si A es taut. para toda valuación v, $v \models A$. Entonces $v \not\models \neg A$.

$\Leftarrow$) Si $\neg A$ es insatisfacible, para toda valuación v $v \not\models \neg A$. Luego $v \models A$.

□

Esto sugiere un método indirecto para probar que una prop A es una tautología, probando que $\neg A$ es insatisfacible.

Suele ser más fácil refutar que probar. Entonces si queremos probar P, refutamos $\neg P$. Esto entonces va a ser útil porque a los mecanismos de resolución les va a ser más fácil refutar.

3.2.2 Resolución

Def (Principio de demostracion por **refutación**)**.** Vamos a probar que A es **válido** mostrando que $\neg A$ es **insatisfacible**.

Hay varias técnicas de demostración por refutación

- Tableaux semántico (1960)
- Procedimiento de Davis-Putnam (1960)
- **Resolución** (1965) (nos vamos a enfocar en este)

El método de resolución fue introducido por Alan Robinson en 1965, es simple de implementar, y se usa mucho en el ámbito de demostración automática de teoremas. Tiene una única regla de inferencia: **regla de resolución**. Y si bien no es imprescindible, por conveniencia vamos a asumir que las fórmulas están en **forma normal conjuntiva** (FNC).

Forma normal conjuntiva (FNC)

Un **literal** es una variable proposicional P o su negación $\neg P$. Una proposición A está en **FNC** si es una conjunción

$$C_1 \wedge \cdots \wedge C_n$$

donde cada C_i (llamado **cláusula**) es una disyunción

$$B_{i1} \vee \cdots \vee B_{in_i}$$

y cada B_{ij} es un literal. Una FNC es entonces una "conjunción de disyunciones de literales"

Ejemplo. Ejemplos de FNC

- $(P \vee Q) \wedge (P \vee \neg Q)$ está en FNC
- $(P \vee Q) \wedge (P \vee \neg\neg Q)$ no está en FNC que tiene un doble neg en C_2.
- $(P \wedge Q) \vee P$ no está en FNC.

Teorema 3.2. Para toda proposición A puede hallarse una proposición A' en FNC que es lógicamente equivalente para A.

Decimos que A es lógicamente equivalente a B sii $A \iff B$ es una tautología.

Notación conjuntista de FNC

Tomando en cuenta que tanto el $\vee$ como $\wedge$,

- son conmutativos $((A \vee B) \iff (B \vee A))$
- son asociativos $(((A \vee B) \vee C) \iff (A \vee (B \vee C)))$
- son idempotentes $((A \vee A) \iff A)$

podemos asumir que

- Cada cláusula C_i es **distinta** (si fueran iguales, por idempotencia las podría juntar).
- Cada cláusula puede verse como un conjunto de literales distintos.

Por lo tanto podemos notar una FNC como

$$\{C_1, \ldots, C_n\}$$

donde cada C_i es un conjunto de literales

$$\{B_{i1}, \ldots, B_{in_i}\}$$

Ejemplo. La FNC $(P \vee Q) \wedge (P \vee \neg Q)$ se nota como

$$\{\{P, Q\}, \{P, \neg Q\}\}$$

Principio fundamental del método de resolución

La resolución se basa en el hecho de que la siguiente proposición es una **tautología**,

$$(A \vee P) \wedge (B \vee \neg P) \iff (A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)$$

Intuitivamente, si vale P entonces no puede valer $\neg P$, por lo que tiene que valer B valga o no A, y lo mismo al revés

Por lo tanto, el conjunto de cláusulas

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}\}$$

es **lógicamente equivalente** a

$$\{C_1, \ldots, C_m, \{A, P\}, \{B, \neg P\}, \{A, B\}\}$$

y entonces el primero va a ser **insatisfacible** sii el segundo lo es.

La cláusula $\{A, B\}$ se llama **resolvente** de las cláusulas $\{A, P\}$ y $\{B, \neg P\}$.

El resolvente de las cláusulas $\{P\}$ y $\{\neg P\}$ es la **cláusula vacía** (porque es insatisfacible) y se anota $\square$. El vacío significa *falso*, como era un y de todo, podemos concluir que es insatisfacible.

Regla de resolución

Def (Opuesto)**.** Dado un literal L, el **opuesto** de L $\overline{L}$ se define como

- $\neg P$ si $L = P$
- P si $L = \neg P$

Tenemos que definir esto para poder devolver la negación de un literal sin agregarle símbolos.

Def (Resolvente)**.** Dadas dos cláusulas C_1, C_2 una cláusula C se dice **resolvente de C_1 y C_2** sii para algún literal L, $L \in C_1, \overline{L} \in C_2$ y

$$C = (C_1 - \{L\}) \cup (C_2 - \{\overline{L}\})$$

Esto es lo de antes pero escrito más formal.

Ejemplo. Ejemplos

- Las cláusulas $\{A, B\}$ y $\{\neg A, \neg B\}$ tienen dos resolventes: $\{A, \neg A\}$ y $\{B, \neg B\}$.
- Las cláusulas $\{P\}$ y $\{\neg P\}$ tienen a la cláusula vacía $\square$ como resolvente.

Def (Regla de resolución)**.** En base a lo anterior podemos definir la **regla de resolución** como sigue,

$$\frac{\{A_1, \ldots, A_m, Q\} \quad \{B_1, \ldots, B_n, \neg Q\}}{\{A_1, \ldots, A_m, B_1, \ldots, B_n\}}$$

Ejemplo. El resultado de aplicar la regla de resolución a

$$\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$$

es

$$\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}\}$$

Método de resolución

El proceso de agregar a un conjunto S el resolvente C de dos cláusulas $C_1, C_2 \in S$ (aplicando la regla de resolución a S) se llama **paso de resolución**. (asumimos $C \notin S$)

Recordamos que los pasos de resolución preservan la insatisfacibilidad,

$$S \text{ es insatisfacible} \iff S \cup \{C\} \text{ es insatisfacible}$$

ya que eran lógicamente equivalentes, como vimos en section 3.2.2 Principio fundamental del método de resolución

Vamos a aplicar el metodo para obtener una **refutación**. Un conjunto de cláusulas se llama una refutación si contiene la cláusula vacía ($\square$).

De esa forma, el método de resolución trata de constuir una secuencia de conjuntos de cláusulas obtenidas usando pasos de resolución hasta llegar a una **refutación**

$$S_1 \Rightarrow S_2 \Rightarrow \dots S_{n-1} \Rightarrow S_n \ni \square$$

en este caso, sabemos que el *conjunto inicial* de cláusulas es insatisfacible, dado que cada paso de resolución preserva la insatisfacibilidad y el último conjunto de cláusulas es insatisfacible.

Ejemplos

Ejemplo. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$ es insatisfacible.

1. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$
2. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}\}$
3. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}, \{\neg P\}\}$
4. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}, \{\neg P\}, \square\}$

Ejemplo. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}\}$ es insatisfacible

1. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}\}$
2. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}\}$
3. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}, \{A\}\}$
4. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}, \{A\}, \square\}$

Formulas satisfacibles

Queremos mostrar que $S = \{\{A, B, C\}, \{A\}, \{B\}\}$ es insatisfacible, pero no podemos aplicar ningún paso de resolución a S porque no hay negaciones. Por lo tanto, no puede llegarse a una refutación a partir de S. En lógica proposicional, estamos seguros entonces que S debe ser satisfacible. Efectivamente, podemos tomar por ej. $v(A) = v(B) = \mathbf{T}$.

Terminación

La aplicación reiterada de la regla de resolución **siempre termina** (suponiendo que el resolvente que se agrega es nuevo). Ya que,

1. El resolvente (la cláusula nueva que se agrega) se forma con los literales distintos que aparecen en el conjunto de las cláusulas de partida S.
2. Hay una cantidad **finita** de literales en el conjunto de cláusulas de partida S.

En el peor de los casos, la regla de resolución podrá generar una nueva cláusula por cada combinación diferente de literales distintos de S. Pero sigue siendo finita y por lo tanto termina.

Corrección y completitud

Teorema 3.3. Dado un conjunto finito S de cláusulas, S es insatisfacible sii tiene una refutación.

Este resultado establece la corrección y completitud del método de resolución.

> **E**n LPO no es tan bueno. Si encuentra una refutación significa que es insatisfacible (correcto) pero puede pasar que no la encontremos (incompleto).

Resumen del algoritmo

Para probar que A es una tautología hacemos lo siguiente,

1. Calculamos la FNC de $\neg A$
2. Aplicamos el método de resolución
3. Si hallamos una refutación, entonces $\neg A$ es insatisfacible y por lo tanto A es una tautología. (Por Teo. 3.1)
4. Si no hallamos ninguna refutación, entonces $\neg A$ es satisfacible y por lo tanto A no es una tautologia. (Por negación de Teo. 3.1)