Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

15 de agosto de 2017

Repaso: Expresiones y tipos básicos

Tipos elementales

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

Repaso: Expresiones y tipos básicos

Tipos elementales

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

Guardas

```haskell
signo n | n >= 0    = True
        | otherwise = False
```

Repaso: Expresiones y tipos básicos

Tipos elementales

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

Guardas

```
signo n | n >= 0    = True
        | otherwise = False
```

Pattern matching

```
longitud [] = 0
longitud (x:xs) = 1 + (longitud xs)
```

Repaso: Polimorfismo paramétrico

`todosIguales` es una función que determina si todos los elementos de una lista son iguales entre sí.

Implementar y dar el tipo de la función

```
todosIguales :: ??
todosIguales = ...
```

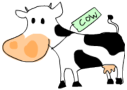
Repaso: Polimorfismo paramétrico

`todosIguales` es una función que determina si todos los elementos de una lista son iguales entre sí.

Implementar y dar el tipo de la función

```
todosIguales :: ??
todosIguales = ...
```

- El sistema de tipos de Haskell permite definir funciones para ser usadas con más de un tipo
- Su tipo se expresa con *variables de tipo*



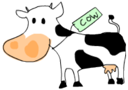
Repaso: Polimorfismo paramétrico

`todosIguales` es una función que determina si todos los elementos de una lista son iguales entre sí.

Implementar y dar el tipo de la función

```
todosIguales :: ??
todosIguales = ...
```

- El sistema de tipos de Haskell permite definir funciones para ser usadas con más de un tipo
- Su tipo se expresa con *variables de tipo*



Haskell no necesita que todos los tipos sean especificados a mano ni tampoco requiere anotaciones de tipos en el código. Para eso utiliza un Inferidor de Tipos (veremos más en λ-Cálculo).

Listas infinitas

Definición de listas

- Listas por extensión
  `[0, 3, 0, 3, 4, 5, 6]`
- Secuencias aritméticas
  `[1..4]   [5, 7..13]`
- Listas por comprensión
  `[expresion | selectores, condiciones]`
  `[(x, y) | x <-[0..3], y <-[0..3]]`

¿Las listas pueden ser infinitas?

Listas infinitas

Definición de listas

- Listas por extensión
  `[0, 3, 0, 3, 4, 5, 6]`
- Secuencias aritméticas
  `[1..4]   [5, 7..13]`
- Listas por comprensión
  `[expresion | selectores, condiciones]`
  `[(x, y) | x <-[0..3], y <-[0..3]]`

¿Las listas pueden ser infinitas?

Ejemplo

- `infinitosUnos = 1 : infinitosUnos`
- `naturales =`

Listas infinitas

Definición de listas

- Listas por extensión
  `[0, 3, 0, 3, 4, 5, 6]`
- Secuencias aritméticas
  `[1..4]   [5, 7..13]`
- Listas por comprensión
  `[expresion | selectores, condiciones]`
  `[(x, y) | x <-[0..3], y <-[0..3]]`

¿Las listas pueden ser infinitas?

Ejemplo

- `infinitosUnos = 1 : infinitosUnos`
- `naturales =  [0..]`
- `multiplosDe3 =`

Listas infinitas

Definición de listas

- Listas por extensión
  `[0, 3, 0, 3, 4, 5, 6]`
- Secuencias aritméticas
  `[1..4]   [5, 7..13]`
- Listas por comprensión
  `[expresion | selectores, condiciones]`
  `[(x, y) | x <-[0..3], y <-[0..3]]`

¿Las listas pueden ser infinitas?

Ejemplo

- `infinitosUnos = 1 : infinitosUnos`
- `naturales =  [0..]`
- `multiplosDe3 =  [0,3..]`
- `repeat "hola"`
- `primos =`

Listas infinitas

Definición de listas

- Listas por extensión
  `[0, 3, 0, 3, 4, 5, 6]`
- Secuencias aritméticas
  `[1..4]   [5, 7..13]`
- Listas por comprensión
  `[expresion | selectores, condiciones]`
  `[(x, y) | x <-[0..3], y <-[0..3]]`

¿Las listas pueden ser infinitas?

Ejemplo

- `infinitosUnos = 1 : infinitosUnos`
- `naturales =  [0..]`
- `multiplosDe3 =  [0,3..]`
- `repeat "hola"`
- `primos =  [n | n <- [2..], esPrimo n]`

Listas infinitas

Definición de listas

- Listas por extensión
  `[0, 3, 0, 3, 4, 5, 6]`
- Secuencias aritméticas
  `[1..4]   [5, 7..13]`
- Listas por comprensión
  `[expresion | selectores, condiciones]`
  `[(x, y) | x <-[0..3], y <-[0..3]]`

¿Las listas pueden ser infinitas?

Ejemplo

- `infinitosUnos = 1 : infinitosUnos`
- `naturales =  [0..]`
- `multiplosDe3 =  [0,3..]`
- `repeat "hola"`
- `primos =  [n | n <- [2..], esPrimo n]`
- ¿Qué sucede al reducir `take 2 infinitosUnos`?
- ¿Qué sucede al reducir `length naturales`?

Evaluación Lazy

Modelo de cómputo: Reducción

- Se reemplaza un *redex* (reducible expresion) utilizando las ecuaciones orientadas.
- El redex debe ser una instancia del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las correspondientes variables sustituidas.
- El resto de la expresión no cambia.

Evaluación Lazy

Modelo de cómputo: Reducción

- Se reemplaza un *redex* (reducible expresion) utilizando las ecuaciones orientadas.
- El redex debe ser una instancia del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las correspondientes variables sustituidas.
- El resto de la expresión no cambia.

Para seleccionar el redex: Orden Normal, o también llamado Lazy

Evaluación Lazy

Modelo de cómputo: Reducción

- Se reemplaza un *redex* (reducible expresion) utilizando las ecuaciones orientadas.
- El redex debe ser una instancia del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las correspondientes variables sustituidas.
- El resto de la expresión no cambia.

Para seleccionar el redex: Orden Normal, o también llamado Lazy

- Se selecciona el redex más externo y más a la izquierda para el que se pueda conocer qué ecuación del programa utilizar.
- En general: primero las funciones más externas y luego los argumentos (sólo si se necesitan).

Ejercicio

Mostrar los pasos necesarios para reducir `nUnos 2`

```
take :: Int -> [a] -> [a]
take 0 l      = []
take n []     = []
take n (x:xs) = x : (take (n-1) xs)

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

Ejercicios

Ejercicio

Mostrar los pasos necesarios para reducir `nUnos 2`

```
take :: Int -> [a] -> [a]
take 0 l       = []
take n []      = []
take n (x:xs) = x : (take (n-1) xs)

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

Digresión

- ¿Qué sucedería si usáramos otra estrategia de reducción?
- ¿Existe algún término que admita una reducción finita pero para el cual la estrategia lazy no termine?
- Si un término admite otra reducción finita además de la lazy, ¿el resultado de ambas coincide?

Funciones de alto orden

Definamos las siguientes funciones
Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Funciones de alto orden

Definamos las siguientes funciones
Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

- `mejorSegun ::`

Funciones de alto orden

Definamos las siguientes funciones
Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

- `mejorSegun :: (a -> a -> Bool) -> [a] -> a`

Funciones de alto orden

Definamos las siguientes funciones
Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

- `mejorSegun :: (a -> a -> Bool) -> [a] -> a`
- Reescribir `maximo` y `listaMasCorta` en base a `mejorSegun`

Funciones sin nombre (*lambdas*)

```
(\x -> x + 1) :: Num a => a -> a
(\x y -> "hola") :: t1 -> t2 -> [Char]
(\x y -> x + y) 10 20 ⇝ 30
```

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?

Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

22 de agosto de 2017

Hoy presentamos...

1. Esquemas de recursión sobre listas
   - Map
   - Filter
2. Folds sobre listas
   - FoldR
   - FoldL
3. Otros esquemas de recursión sobre listas
4. Tipos algebraicos
5. Recursión estructural en tipos algebraicos

Esquemas de recursión sobre listas: Map

```
map :: (a -> b) -> [a] -> [b]
```

Esquemas de recursión sobre listas: Map

```
map :: (a -> b) -> [a] -> [b]
```

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

Esquemas de recursión sobre listas: Map

```
map :: (a -> b) -> [a] -> [b]
```

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

O, hablando en francés, la función `map`

- Toma una función que sabe como convertir un tipo **a** en otro **b**,
- Y nos devuelve una función que sabe como convertir listas de **a** en listas de **b**.

Esquemas de recursión sobre listas: Map

```
map :: (a -> b) -> [a] -> [b]
```

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

O, hablando en francés, la función `map`

- Toma una función que sabe como convertir un tipo **a** en otro **b**,
- Y nos devuelve una función que sabe como convertir listas de **a** en listas de **b**.

```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

Esquemas de recursión sobre listas: Map

```
map :: (a -> b) -> [a] -> [b]
```

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

O, hablando en francés, la función `map`

- Toma una función que sabe como convertir un tipo **a** en otro **b**,
- Y nos devuelve una función que sabe como convertir listas de **a** en listas de **b**.

```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

Definir utilizando map

- `longitudes :: [[a]] -> [Int]`
- `losIesimos :: [Int] -> [[a] -> a]` que devuelve una lista con las funciones que toman los iésimos de una lista.
- `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista l, devuelve la lista $[l_{i_1}, \ldots, l_{i_n}]$

Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

O, hablando en francés, la función `filter`

- Toma una función que nos dice si un elemento cumple una condicón,
- Y nos devuelve una función que sabe como convertir listas de elementos cualquiera en listas cuyos elementos cumplen la condición deseada.

Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

O, hablando en francés, la función `filter`

- Toma una función que nos dice si un elemento cumple una condicón,
- Y nos devuelve una función que sabe como convertir listas de elementos cualquiera en listas cuyos elementos cumplen la condición deseada.

```
filter p [] = []
filter p (x:xs) = if p x then x:(filter xs) else filter xs
```

Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

O, hablando en francés, la función `filter`

- Toma una función que nos dice si un elemento cumple una condicón,
- Y nos devuelve una función que sabe como convertir listas de elementos cualquiera en listas cuyos elementos cumplen la condición deseada.

```
filter p [] = []
filter p (x:xs) = if p x then x:(filter xs) else filter xs
```

Definir utilizando filter

- `deLongitudN :: Int -> [[a]] -> [[a]]`
- `soloPuntosFijos :: [Int -> Int] -> Int -> [Int -> Int]` que toma una lista de funciones y un número n. En el resultado, deja las funciones que al aplicarlas a n dan n.
- `quickSort :: Ord a => [a] -> [a]`

Esquemas de recursión sobre listas: FoldR

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Esquemas de recursión sobre listas: FoldR

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

La función `foldr` nos permite realizar recursión estructural sobre una lista.

Esquemas de recursión sobre listas: FoldR

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, hablando en francés, la función `foldr`

- Toma una función que representa el paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

Esquemas de recursión sobre listas: FoldR

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, hablando en francés, la función `foldr`

- Toma una función que representa el paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Esquemas de recursión sobre listas: FoldR

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, hablando en francés, la función `foldr`

- Toma una función que representa el paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Definir utilizando foldr

- `longitud :: [a] -> Int`
- `concatenar :: [[a]] -> [a]`
- `suma :: [Int] -> Int`

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
---> 1 + 5
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
---> 1 + 5
---> 6
```

Esquemas de recursión sobre listas: FoldR

FoldR

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
---> 1 + 5
---> 6
```

Notar que el primer `(+)` que se puede resolver es entre el último elemento de la lista y el caso base del `foldr`. Por esta razón decimos que el `foldr` *acumula* el resultado desde la **derecha**.

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

FoldL

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

Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**. Se define de la siguiente forma:

FoldL

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

Definir utilizando foldl

- `reverso :: [a] -> [a]`
- `suma :: [Int] -> Int`

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
---> (3 + 3)
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
---> (3 + 3)
---> 6
```

Esquemas de recursión sobre listas: FoldL

FoldL

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

¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
---> (3 + 3)
---> 6
```

Notar que el primer `(+)` que se puede resolver es entre el primer elemento de la lista y el caso base del `foldl`.

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
```

Usando foldl

```
suma [1..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
---> 1 + (2 + (foldr (+) 0 [3..]))
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
---> foldl (+) ((0 + 1) + 2) [3..]
```

Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
---> 1 + (2 + (foldr (+) 0 [3..]))
---> 1 + (2 + (3 + (foldr (+) 0 [4..])))
```

Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
---> foldl (+) ((0 + 1) + 2) [3..]
---> foldl (+) (((0 + 1) + 2) + 3) [4..]
```

Esquemas de recursión sobre listas: FoldR1 y FoldL1

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía.

Esquemas de recursión sobre listas: FoldR1 y FoldL1

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía.

Definir las siguientes funciones

- `ultimo :: [a] -> a`
- `maximum :: Ord a => [a] -> a`

Esquemas de recursión sobre listas: FoldR1 y FoldL1

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía.

Definir las siguientes funciones

- `ultimo :: [a] -> a`
- `maximum :: Ord a => [a] -> a`

¿Qué computan estas funciones?

- ```
  f1 :: [Bool] -> Bool
  f1 = foldr (&&) True
  ```
- ```
  f2 :: [a] -> [a]
  f2 = foldr (:) []
  ```
- ```
  f3 :: [a] -> [a] -> [a]
  f3 xs ys = foldr (:) ys xs
  ```
- ```
  f4 :: [a] -> [a]
  f4 = foldl (flip (:)) []
  ```

¡Las difíciles!

Calentando motores... (no vale recursión explícita)

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece e = foldr ...
```

¡Las difíciles!

Calentando motores... (no vale recursión explícita)

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece e = foldr ...
```

Definir la función take, ¿cuál es la diferencia?

```
take :: Int -> [a] -> [a]
take n = foldr ...
```

Break

Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b
= (a -> Bool)          – determina si es o no el caso trivial
-> (a -> b)            – resuelve el caso trivial
-> (a -> [a])          – parte el problema en sub-problemas
-> ([b] -> b)          – combina resultados
-> a                   – input
-> b                   – resultado
```

Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b
= (a -> Bool)                 -- determina si es o no el caso trivial
-> (a -> b)                   -- resuelve el caso trivial
-> (a -> [a])                 -- parte el problema en sub-problemas
-> ([b] -> b)                 -- combina resultados
-> a                          -- input
-> b                          -- resultado
```

Definir las siguientes funciones

- ```
  dc :: DivideConquer a b
  dc esTrivial resolver repartir combinar x = ...
  ```
- ```
  mergeSort :: Ord a => [a] -> [a]
  mergeSort = dc ...
  ```

Tipos algebraicos y su definición en Haskell

Tipos algebraicos

- definidos como **combinación de otros tipos**
- están formados por uno o más constructores
- cada constructor puede o no tener argumentos
- los argumentos de los constructores pueden ser recursivos
- se inspeccionan usando *pattern matching*
- se definen mediante la cláusula `data`

Algunos ejemplos

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Tipos algebraicos y su definición en Haskell

Tipos algebraicos

- definidos como **combinación de otros tipos**
- están formados por uno o más constructores
- cada constructor puede o no tener argumentos
- los argumentos de los constructores pueden ser recursivos
- se inspeccionan usando *pattern matching*
- se definen mediante la cláusula `data`

Algunos ejemplos

```haskell
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b

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

Folds sobre estructuras nuevas

¿Cómo hacemos?

Recordemos el tipo de `foldr`, el esquema de recursión estructural para listas.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

¿Por qué tiene ese tipo?

(Pista: pensar en cuáles son los constructores del tipo `[a]`).

¿Cómo hacemos?

Recordemos el tipo de `foldr`, el esquema de recursión estructural para listas.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

¿Por qué tiene ese tipo?
(Pista: pensar en cuáles son los constructores del tipo `[a]`).

Un esquema de recursión estructural espera recibir **un argumento por cada constructor** (para saber qué devolver en cada caso), y además **la estructura que va a recorrer**.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. (¡Y todos van a devolver lo mismo!)

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

Folds sobre estructuras nuevas

Definir el esquema de recursión estructural para el siguiente tipo:

```haskell
data Formula = Proposicion String | No Formula
                                  | Y Formula Formula
                                  | O Formula Formula
                                  | Imp Formula Formula
```

Folds sobre estructuras nuevas

Definir el esquema de recursión estructural para el siguiente tipo:

```
data Formula = Proposicion String | No Formula
                                  | Y Formula Formula
                                  | O Formula Formula
                                  | Imp Formula Formula
```

Ejercicio

Usando el esquema definido, escribir las funciones:

- `proposiciones :: Formula -> [String]`
- `quitarImplicaciones :: Formula -> Formula` que convierte todas las formulas de la pinta $(p \implies q)$ a $(\neg p \vee q)$
- `evaluar :: [(Proposicion, Bool)] -> Formula -> Bool` que dada una formula y los valores de verdad asignados a cada una de sus proposiciones, nos devuelve el resultado de evaluar la fórmula lógica.

Fin (por ahora)

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?

Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

29 de agosto de 2017

Nos faltaba...

1. Funciones como estructuras de datos

2. Generación infinita

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

```
type Conj a = (a->Bool)
```

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

```
type Conj a = (a->Bool)
```

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Operaciones sobre conjuntos

- Definir y dar el tipo de las siguientes funciones:
  - vacío
  - agregar
  - unión
  - intersección

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

```
type Conj a = (a->Bool)
```

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Operaciones sobre conjuntos

- Definir y dar el tipo de las siguientes funciones:
  - vacío
  - agregar
  - unión
  - intersección
- ¿Puede definirse la función `esVacio :: Conj a -> Bool`?

Ejercicio

Se cuenta con la siguiente representación de conjuntos, caracterizados por su función de pertenencia:

```haskell
type Conj a = (a->Bool)
```

De este modo, si `conj1` es un conjunto y `e` un elemento, la expresión `conj1 e` devuelve `True` si `e` pertenece a `conj1`, y `False` en caso contrario.

Operaciones sobre conjuntos

- Definir y dar el tipo de las siguientes funciones:
  - vacío
  - agregar
  - unión
  - intersección
- ¿Puede definirse la función `esVacio :: Conj a -> Bool`?
  ¿Y `esVacio :: Conj Bool -> Bool`?
- Definir la función `primerOcurrencia :: a -> [Conj a] -> Int` que, dados un elemento e y una lista de conjuntos (que puede ser finita o infinita), devuelva la primera posición de la lista en la cual el conjunto correspondiente contiene al elemento e. Se asume que e pertenece al menos a un conjunto de la lista.

Generación Infinita

Ejercicio: Definir

`puntosDelCuadrante :: [Punto]`

Donde `Punto`, un renombre de tipos: `type Punto = (Int, Int)`

El resultado debe ser una lista (infinita) que contenga todos los puntos del cuadrante superior derecho del plano (sin repetir).

Generación Infinita

Ejercicio: Definir

`puntosDelCuadrante :: [Punto]`

Donde `Punto`, un renombre de tipos: `type Punto = (Int, Int)`

El resultado debe ser una lista (infinita) que contenga todos los puntos del cuadrante superior derecho del plano (sin repetir).

Ejercicio de tarea: Definir

`listasPositivas :: [[Int]]`

que contenga todas las listas finitas de enteros mayores o iguales que 1.

Generación Infinita

Ejercicio: Definir

`puntosDelCuadrante :: [Punto]`

Donde `Punto`, un renombre de tipos: `type Punto = (Int, Int)`

El resultado debe ser una lista (infinita) que contenga todos los puntos del cuadrante superior derecho del plano (sin repetir).

Ejercicio de tarea: Definir

`listasPositivas :: [[Int]]`

que contenga todas las listas finitas de enteros mayores o iguales que 1.

Ayuda: Definir primero

`listasQueSuman :: Int -> [[Int]]`

que, dado un número natural n, devuelve todas las listas de enteros mayores o iguales que 1 cuya suma sea n

Cálculo Lambda I

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

29 de Agosto de 2017

Objetivo de la clase

$(\lambda x : \text{Bool}.\ \lambda y : \text{Bool} \to \text{Bool}.\ y\ (y\ x))\ ((\lambda z : \text{Bool}.\ \textit{true})\ \textit{false})\ (\lambda w : \text{Bool}.\ w)$

¿Qué significa esto? ¿Significa algo? ¿Es válido? ¿Es un valor? ¿Cómo nos damos cuenta?

Objetivo de la clase

$(\lambda x\colon \text{Bool}.\ \lambda y : \text{Bool} \to \text{Bool}.\ y\ (y\ x))\ ((\lambda z\colon \text{Bool}.\ \mathit{true})\ \mathit{false})\ (\lambda w\colon \text{Bool}.\ w)$

¿Qué significa esto? ¿Significa algo? ¿Es válido? ¿Es un valor? ¿Cómo nos damos cuenta?

Mapa del tema

■ Sintaxis	M, σ
■ Reglas de Tipado	$\Gamma \vdash M : \sigma$
■ Valores	V
■ Reglas de Evaluación	M $\to$ M'

Sintaxis

Ejercicio: ¿Cuáles son expresiones sintácticamente válidas? Dibujar el árbol sintáctico y marcar las ocurrencias libres de variables.

1. $\lambda x : \text{Bool} \to \text{Bool}.x\ \mathit{true}$
2. $(\lambda x : \text{Bool} \to \text{Nat}.x\ \mathit{true})\ (\lambda y : \text{Bool}.x)$
3. $\lambda x : \text{Nat}$
4. $\lambda x.\ x$
5. $\text{if } x \text{ then } y \text{ else } \lambda z : \text{Bool}.z$
6. $x\ (\lambda y : \text{Bool}.y)$
7. $\mathit{true}\ \ \mathit{false}$
8. $\text{succ}(M)$
9. $\text{succ } \mathit{true}$
10. $\text{if } \ \text{succ}(\mathit{true}) \text{ then } \lambda x : \text{Bool}.x$

Ejercicio: Demostrar (o explicar por qué no es posible) los siguientes juicios de tipado:

Chequeo de tipos

Ejercicio: Demostrar (o explicar por qué no es posible) los siguientes juicios de tipado:

1. $\emptyset \vdash (\lambda x : \mathit{Bool}.\ \lambda y : \mathit{Bool}.\ \mathit{if}\ x\ \mathit{then}\ \mathit{true}\ \mathit{else}\ y)\ \mathit{false} : \mathsf{Bool} \rightarrow \mathsf{Bool}$
2. $\emptyset \vdash \mathit{if}\ x\ \mathit{then}\ x\ \mathit{else}\ z : \mathsf{Bool}$

Valores

Ejercicio: ¿Cuáles de estos términos son valores?

1. if *true* then $(\lambda x : \text{Bool}.\ x)$ else $(\lambda x : \text{Bool}.\ \mathit{false})$
2. $\lambda x : \text{Bool}.\ \mathit{false}$
3. $(\lambda x : \text{Bool}.\ x)\ \mathit{false}$
4. succ(0)
5. succ(succ(0))
6. succ(pred(0))
7. $\lambda x : \text{Bool}.\ (\lambda y : \text{Bool}.x)\ \mathit{false}$
8. $\lambda x : \text{Bool} \to \text{Bool}.\ x\ \mathit{true}$

Ejercicio: ¿Cuál es el resultado de evaluar las siguientes expresiones? ¿El resultado, es siempre un valor?

1. $(\lambda x : Bool.\ \lambda y : Bool.\ if\ \ x\ \ then\, true\, else\, y)\ false$
2. $(\lambda x\colon \text{Bool}.\ \lambda y : \text{Bool} \to \text{Bool}.\ y\ (y\ x))\ ((\lambda z\colon \text{Bool}.\ true)\ false)\ (\lambda w\colon \text{Bool}.\ w)$

Simplificando la escritura

Podemos definir macros para expresiones que vayamos a utilizar con frecuencia. Por ejemplo:

- $Id_{bool} \stackrel{def}{=}$

Simplificando la escritura

Podemos definir macros para expresiones que vayamos a utilizar con frecuencia. Por ejemplo:

- $Id_{bool} \stackrel{def}{=} \lambda x : \mathrm{Bool}.x$
- and $\stackrel{def}{=}$

Simplificando la escritura

Podemos definir macros para expresiones que vayamos a utilizar con frecuencia. Por ejemplo:

- $Id_{bool} \stackrel{def}{=} \lambda x: \mathrm{Bool}.x$
- and $\stackrel{def}{=} \lambda x: \mathrm{Bool}.\lambda y: \mathrm{Bool}.\mathrm{if}\ x\ \mathrm{then}\ y\ \mathrm{else}\ \mathit{false}$

Al agregar la siguiente regla para las abstracciones:

$$\frac{M \to M'}{\lambda x\colon \tau.\ M \to \lambda x\colon \tau.\ M'}\ E-ABS$$

Ejercicio

1. Repensar el conjunto de valores para respetar esta modificación, pensar por ejemplo si ($\lambda x\colon$ Bool. $Id_{bool}\ true$) es o no un valor.

Cambiando reglas semánticas

Al agregar la siguiente regla para las abstracciones:

$$\frac{M \to M'}{\lambda x : \tau.\ M \to \lambda x : \tau.\ M'}\ E - ABS$$

Ejercicio

1. Repensar el conjunto de valores para respetar esta modificación, pensar por ejemplo si (λx : Bool. Id_{bool} $true$) es o no un valor.
2. ¿Qué reglas deberían modificarse para no perder el determinismo?

Cambiando reglas semánticas

Al agregar la siguiente regla para las abstracciones:

$$\frac{M \to M'}{\lambda x\colon \tau.\ M \to \lambda x\colon \tau.\ M'}\ E-ABS$$

Ejercicio

1. Repensar el conjunto de valores para respetar esta modificación, pensar por ejemplo si $(\lambda x\colon \mathrm{Bool}.\ Id_{bool}\ true)$ es o no un valor.
2. ¿Qué reglas deberían modificarse para no perder el determinismo?
3. Utilizando la nueva regla y los valores definidos, reducir la siguiente expresión $(\lambda x\colon \mathrm{Nat} \to \mathrm{Nat}.\ x\ \underline{23})\ (\lambda y\colon \mathrm{Nat}.\ 0)$ ¿Qué se puede concluir entonces? ¿Es seguro o no agregar esta regla?

Continuará...

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?

$(\lambda x : \mathit{Clase}.\ \mathit{fin}\ x)\ (\mathit{Cálculo\ Lambda\ I})$

Machete: Tipos y Términos

Las expresiones de tipos (o simplemente tipos) son

$$\sigma \quad ::= \quad \mathit{Bool} \mid \mathit{Nat} \mid \sigma \to \rho$$

Sea $\mathcal{X}$ un conjunto infinito enumerable de variables y $x \in \mathcal{X}$. Los términos están dados por

$$
\begin{array}{rcl}
M & ::= & x \\
 & \mid & \mathit{true} \\
 & \mid & \mathit{false} \\
 & \mid & \text{if } M \text{ then } M \text{ else } M \\
 & \mid & \lambda x : \sigma.\ M \\
 & \mid & M\ M \\
 & \mid & 0 \\
 & \mid & \text{succ}(M) \\
 & \mid & \text{pred}(M) \\
 & \mid & \text{iszero}(M)
\end{array}
$$

Machete: Axiomas y reglas de tipado

$$\frac{}{\Gamma \ \vdash \ \mathit{true} : \mathit{Bool}}\ (\text{T-True}) \qquad \frac{}{\Gamma \ \vdash \ \mathit{false} : \mathit{Bool}}\ (\text{T-False})$$

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

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

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

Machete: Axiomas y reglas de tipado

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

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

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

Machete: Semántica operacional

$$V \quad ::= \quad \mathit{true} \mid \mathit{false} \mid \lambda x : \sigma.\ M \mid \underline{n}$$

donde $\underline{n}$ abrevia $\mathrm{succ}^n(0)$.

Reglas de Evaluación en un paso

$$\frac{M_1 \to M_1'}{M_1\ M_2 \to M_1'\ M_2}\ (\text{E-App1 o } \mu)$$

$$\frac{M_2 \to M_2'}{V_1\ M_2 \to V_1\ M_2'}\ (\text{E-App2 o } \nu)$$

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

Machete: Semántica operacional

$$V \quad ::= \quad \mathit{true} \mid \mathit{false} \mid \lambda x : \sigma.\ M \mid \underline{n}$$

donde $\underline{n}$ abrevia $\mathsf{succ}^n(0)$.

Reglas de Evaluación en un paso

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

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

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

Reglas de Evaluación en un paso

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

$$\frac{}{\mathsf{pred}(0) \to 0} \ (\textsc{E-PredZero}) \qquad \frac{}{\mathsf{pred}(\mathsf{succ}(\underline{n})) \to \underline{n}} \ (\textsc{E-PredSucc})$$

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

$$\frac{}{\mathsf{iszero}(0) \to \mathit{true}} \ (\textsc{E-IsZeroZero}) \qquad \frac{}{\mathsf{iszero}(\mathsf{succ}(\underline{n})) \to \mathit{false}} \ (\textsc{E-IsZeroSucc})$$

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

Cálculo lambda II

Extensiones del cálculo lambda

Paradigmas de Lenguajes de Programación

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Septiembre 2017

En clases anteriores...

- Introducción a C-λ^b como lenguaje representativo del paradigma funcional
- **Sintaxis**. Expresiones de tipos y términos
- **Sistema de tipado**. Contexto y juicios de tipado. Axiomas y reglas.
- **Semántica operacional**. Formas normales y valores. Interpretación.

Sintaxis

Expresiones de tipos

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

Términos

$$M ::= x \mid true \mid false \mid if \ M \ then \ P \ else \ Q \mid \lambda x : \sigma.M \mid M \ N$$

¿Son correctas estas expresiones?

- λx:Bool.x
- $(\lambda x$:Bool.$x)\ x$
- $y\ x$
- $if\ true\ then\ x\ else\ false$
- λx:true.x
- λx:Bool.λy:Bool.$if\ x\ then\ true\ else\ y$

Los tipos nos permiten caracterizar las expresiones del lenguaje que tienen sentido

Axiomas y reglas de tipado

$$\frac{x : \sigma \in \Gamma}{\Gamma \triangleright x : \sigma} \qquad \frac{}{\Gamma \triangleright \mathit{true} : \mathit{Bool}} \qquad \frac{}{\Gamma \triangleright \mathit{false} : \mathit{Bool}}$$

$$\frac{\Gamma \cup \{x : \sigma\} \triangleright M : \tau}{\Gamma \triangleright \lambda x : \sigma . M : \sigma \to \tau} \qquad \frac{\Gamma \triangleright M : \sigma \to \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M\ N : \tau}$$

$$\frac{\Gamma \triangleright M : \mathit{Bool} \quad \Gamma \triangleright P : \sigma \quad \Gamma \triangleright Q : \sigma}{\Gamma \triangleright \mathit{if}\ M\ \mathit{then}\ P\ \mathit{else}\ Q : \sigma}$$

Semántica

La semántica 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 estado

Evaluación

- La semántica nos permite interpretar las expresiones correctas (términos tipados) del lenguaje.
- El objetivo es saber como se evaluan o ejecutan los términos para conocer su *significado*.
- La semántica que usamos es “en un paso” (*small-step*).

Valores

¿Qué significan estas expresiones?

- $\lambda x{:}\mathsf{Bool}.x$
- $(\lambda x{:}\mathsf{Bool}.x)$ *true*
- $(\lambda x{:}\mathsf{Bool} \to \mathsf{Bool}.x)\ (\lambda x{:}\mathsf{Bool}.x)$ *false*

¿Qué son los valores?

Los valores son las expresiones con sentido "directo". Son los posibles resultados de los programas **correctos**.

Los posibles valores en el cálculo λ presentado hasta ahora son:

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

Semántica operacional (en un paso)

$$\frac{M \to M'}{M\ N \to M'\ N} \qquad \frac{N \to N'}{V\ N \to V\ N'} \qquad \frac{}{(\lambda x : \sigma.M)\ V \to M[x \leftarrow V]}$$

$$\frac{M \to M'}{\mathit{if}\ M\ \mathit{then}\ N\ \mathit{else}\ O \to \mathit{if}\ M'\ \mathit{then}\ N\ \mathit{else}\ O}$$

$$\frac{}{\mathit{if}\ \mathit{true}\ \mathit{then}\ N\ \mathit{else}\ O \to N} \qquad \frac{}{\mathit{if}\ \mathit{false}\ \mathit{then}\ N\ \mathit{else}\ O \to O}$$

Extendiendo el C-λ ...

- **Primera extensión:** los naturales
- ¿Qué se agregó?

Sintaxis para cálculo λ con pares

¿Qué hay que agregar?

- ...términos para representar el constructor y los observadores

$$M ::= \ldots \mid\ < M, N > \mid \pi_1(M) \mid \pi_2(M)$$

- ...y un tipo para estas nuevas expresiones

$$\sigma ::= \ldots \mid \sigma \times \tau$$

Reglas de tipado para pares

¿Qué hay que agregar?

- Al menos una regla por cada forma nueva de sintaxis, porque cada una de ellas precisa poder ser tipada.
- Notar que, de no hacerlo, sería imposible construir términos tipables (útiles) con dicha forma.

Regla de tipado para el constructor

$$\frac{\Gamma \triangleright M : \sigma \quad \Gamma \triangleright N : \tau}{\Gamma \triangleright < M, N >: \sigma \times \tau}$$

Reglas de tipado para las proyecciones

$$\frac{\Gamma \triangleright M : \sigma \times \tau}{\Gamma \triangleright \pi_1(M) : \sigma}$$

$$\frac{\Gamma \triangleright N : \sigma \times \tau}{\Gamma \triangleright \pi_2(N) : \tau}$$

Semántica para pares

¿Qué reglas hay que agregar?

- Necesitamos reducir todos los pares *con sentido* que no sean valores.

¿Cuáles son los valores?

- Empecemos por ahí entonces...

Extensión de los valores

$$V ::= \ldots \mid < V, W >$$

Reglas de semántica para pares

Ahora sí, las reglas

$$\frac{M \to M'}{< M, N > \to < M', N >} \qquad \frac{N \to N'}{< V, N > \to < V, N' >}$$

Reglas de semántica para las proyecciones

$$\frac{M \to M'}{\pi_1(M) \to \pi_1(M')} \qquad \frac{M \to M'}{\pi_2(M) \to \pi_2(M')}$$

$$\frac{}{\pi_1(< V, W >) \to V} \qquad \frac{}{\pi_2(< V, W >) \to W}$$

Sintaxis para cálculo λ con árboles binarios

¿Qué hay que agregar?

- ...términos para representar los constructores y observadores
$M ::= \ldots \mid Nil_\sigma \mid Bin(M, N, O) \mid root(M) \mid right(M) \mid left(M) \mid isNil(M)$
- ...y un tipo para estas nuevas expresiones

$$\sigma ::= \ldots \mid AB_\sigma$$

Reglas de tipado para árboles binarios

¿Qué hay que agregar?

- Como antes: una regla por cada forma nueva de sintaxis, porque cada una de ellas precisa poder ser tipada.

Reglas de tipado para los constructores

$$\frac{}{\Gamma \triangleright Nil_\sigma : AB_\sigma}$$

$$\frac{\Gamma \triangleright M : AB_\sigma \quad \Gamma \triangleright O : AB_\sigma \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright Bin(M, N, O) : AB_\sigma}$$

- Nil_σ es una constante diferente según el tipo σ.
  - ¡No tenemos polimorfismo!
- Para *Bin*, en cambio, el tipo queda determinado por el tipo de los subtérminos.

Reglas de tipado para los observadores

$$\frac{\Gamma \triangleright M : AB_\sigma}{\Gamma \triangleright \mathit{root}(M) : \sigma} \qquad \frac{\Gamma \triangleright M : AB_\sigma}{\Gamma \triangleright \mathit{isNil}(M) : \mathit{Bool}}$$

$$\frac{\Gamma \triangleright M : AB_\sigma}{\Gamma \triangleright \mathit{left}(M) : AB_\sigma} \qquad \frac{\Gamma \triangleright M : AB_\sigma}{\Gamma \triangleright \mathit{right}(M) : AB_\sigma}$$

Semántica para árboles binarios

- Primero, empecemos por los valores:

$$V \ ::= \ ... \mid Nil_\sigma \mid Bin(V, W, Y)$$

Reglas de semántica para los constructores

$$\frac{M \to M'}{Bin(M, N, O) \to Bin(M', N, O)}$$

$$\frac{N \to N'}{Bin(V, N, O) \to Bin(V, N', O)}$$

$$\frac{O \to O'}{Bin(V, W, O) \to Bin(V, W, O')}$$

Reglas de semántica para los observadores (1/2)

$$\frac{M \rightarrow M'}{\mathit{left}(M) \rightarrow \mathit{left}(M')} \qquad \frac{M \rightarrow M'}{\mathit{right}(M) \rightarrow \mathit{right}(M')}$$

$$\frac{M \rightarrow M'}{\mathit{root}(M) \rightarrow \mathit{root}(M')} \qquad \frac{M \rightarrow M'}{\mathit{isNil}(M) \rightarrow \mathit{isNil}(M')}$$

Reglas de semántica para los observadores (2/2)

$$\frac{}{isNil(Nil_\sigma) \to true} \qquad \frac{}{isNil(Bin(V, W, Y)) \to false}$$

$$\frac{}{left(Bin(V, W, Y)) \to V} \qquad \frac{}{right(Bin(V, W, Y)) \to Y}$$

$$\frac{}{root(Bin(V, W, Y)) \to W}$$

Otra forma de proyectar/observar

- Vamos a ver otra forma de representar proyectores u observadores más prolija y que requiere menos reglas (aunque una construcción más sofisticada).

- Usamos los árboles nuevamente para ejemplificar, de manera que se pueda comparar correctamente ambas formas.

Sintaxis para cálculo λ con árboles binarios bis

- Los tipos quedan igual que en el caso anterior:

$$\sigma ::= \ldots \mid AB_\sigma$$

- Y los términos,

  $M ::= \ldots \mid Nil_\sigma \mid Bin(M, N, O) \mid$
  $Case_{AB_\sigma}\ M\ of\ Nil \rightsquigarrow N\ ; Bin(m, n, o) \rightsquigarrow O$

  Aquí las minúsculas (m,n,o) representan **variables**.

Reglas de tipado para árboles binarios bis

- Para los constructores son las que ya teníamos.

$$\frac{}{\Gamma \triangleright Nil_\sigma : AB_\sigma}$$

$$\frac{\Gamma \triangleright M : AB_\sigma \quad \Gamma \triangleright O : AB_\sigma \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright Bin(M, N, O) : AB_\sigma}$$

Regla de tipado para el *Case*

$$\frac{\Gamma \triangleright M : AB_\sigma \quad \Gamma \triangleright N : \tau \qquad \Gamma \cup \{m : AB_\sigma, n : \sigma, o : AB_\sigma\} \triangleright O : \tau}{\Gamma \triangleright Case_{AB_\sigma}\ M\ of\ Nil \rightsquigarrow N\ ; Bin(m, n, o) \rightsquigarrow O : \tau}$$

Semántica para los árboles binarios bis

- Tenemos los mismos valores que antes:

$$V \;::=\; ... \mid Nil_{\sigma} \mid Bin(V, W, Y)$$

Reglas de semántica para los constructores

- Análogas a las que ya teníamos.

$$\frac{M \to M'}{Bin(M, N, O) \to Bin(M', N, O)}$$

$$\frac{N \to N'}{Bin(V, N, O) \to Bin(V, N', O)}$$

$$\frac{O \to O'}{Bin(V, W, O) \to Bin(V, W, O')}$$

Reglas de semántica para el *Case*

$$\frac{M \to M'}{\begin{array}{c} Case_{AB_\sigma}\ M\ of\ Nil \rightsquigarrow N\ ; Bin(m,n,o) \rightsquigarrow O \\ \to \\ Case_{AB_\sigma}\ M'\ of\ Nil \rightsquigarrow N\ ; Bin(m,n,o) \rightsquigarrow O \end{array}}$$

$$\frac{}{Case_{AB_\sigma}\ Nil_\sigma\ of\ Nil \rightsquigarrow N\ ; Bin(m,n,o) \rightsquigarrow O \to N}$$

$$\frac{}{\begin{array}{c} Case_{AB_\sigma}\ Bin(V, W, Y)\ of\ Nil \rightsquigarrow N\ ; Bin(m,n,o) \rightsquigarrow O \\ \to O\{m \leftarrow V, n \leftarrow W, o \leftarrow Y\} \end{array}}$$

Ejercicio

Objetivo: Extender el lenguaje para soportar una estructura **fold** que servirá como esquema de recursión para los árboles binarios

- Tipos
  $\sigma ::= ... \mid AB_\sigma$

- Términos

  $M ::= ... \mid Nil_\sigma \mid Bin(M, N, O) \mid$
  $Fold\ M\ base = N;\ rec\ r_i\ e\ r_d = O$

2010-1c-1r

En la próxima clase...

Inferencia de tipos para C-λ.

- Mecanismo para reconstruir el tipo de una expresión cualquiera sin anotaciones de tipo.

¡Eso es todo!

$$(\lambda x : \mathit{Clase}.\mathit{fin}\ x)\ \mathit{LambdaCalculo2}$$

Inferencia de Tipos

PLP

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

12 de septiembre de 2017

Motivación

Ejemplo

Dada la expresión $\lambda x\colon \mathsf{Nat}\,.\,\mathsf{isZero}(x)$, ¿qué tipo tiene?

$$\emptyset \rhd \lambda x\colon \mathsf{Nat}\,.\,\mathsf{isZero}(x) : \mathsf{Nat} \to \mathsf{Bool}$$

¿Cómo hicimos? ¿Se puede automatizar? ¿Podríamos no escribir el : Nat?
¿Y para $\lambda x.\lambda y.(x\ y)\ (\lambda z.x)$?

Inferencia

Dada una expresión, ¿tiene tipo? ¿Cuál es este tipo? ¿Es el más general? ¿Qué necesitamos saber del contexto?

Más ejemplos *a ojo*

¿Qué tipo tiene? ¿En qué contexto? ¿En qué contexto?
¿Es lo más general posible?

- $\emptyset \triangleright \lambda x{:}\mathsf{Nat}.\ \mathsf{succ}(x) : \mathsf{Nat} \rightarrow \mathsf{Nat}$
- $\{y{:}\mathsf{Nat}\} \triangleright \lambda x{:}t.\ \mathsf{succ}(y)\ : t \rightarrow \mathsf{Nat}$
- $(\lambda x.\mathsf{isZero}(x))\ \mathsf{true}$ no tiene tipo.
- $\emptyset \triangleright \lambda x : \mathsf{Nat}.\ x : \mathsf{Nat} \rightarrow \mathsf{Nat}$ no es lo más general.
- $\emptyset \triangleright \lambda x : t.\ x : t \rightarrow t$ es lo más general.

Generalidad

Dijimos que queremos el juicio *más general*. ¿Qué significa ser el más general?

Todos los juicios derivables para $\lambda x.x$ son instancias de $\emptyset \rhd \lambda x : t\,.\,x : t \to t$. Por ejemplo:

- $\emptyset \rhd \lambda x : \mathsf{Nat}.\ x : \mathsf{Nat} \to \mathsf{Nat}$
- $\emptyset \rhd \lambda x : \mathsf{Bool}.\ x : \mathsf{Bool} \to \mathsf{Bool}$
- $\{y : \mathsf{Bool}\} \rhd \lambda x : r \to \mathsf{Nat}.\ x : (r \to \mathsf{Nat}) \to (r \to \mathsf{Nat})$
- $\ldots$

Recordemos algunas reglas de tipado

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\,(\textsc{T-Var}) \qquad \frac{\Gamma \cup \{x : \sigma\} \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\,(\textsc{T-Abs})$$

$$\frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\,N : \tau}\,(\textsc{T-App})$$

Tipado vs. Inferencia

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

$$\mathbb{W}(x) \quad \stackrel{\text{def}}{=} \quad \{x : t\} \rhd x : t, \quad t \text{ variable fresca}$$

Tipado vs. Inferencia

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

Otra forma de escribirlo:

- Sea $\mathbb{W}(U)$=$\Gamma \rhd M : \rho$
- Si el contexto tiene información de tipos para x (i.e. $x:\tau \in \Gamma$ para algún τ), entonces

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

- Si el contexto no tiene información de tipos para x (i.e. $x \notin \mathrm{Dom}(\Gamma)$) elegimos una variable fresca t y entonces

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

$$\tau = \begin{cases} \alpha \text{ si } x:\alpha \in \Gamma \\ \text{variable fresca en otro caso.} \end{cases}$$

$\Gamma' = \Gamma \ominus \{x\}$

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

Tipado vs. Inferencia

$$\frac{\Gamma \triangleright M : \sigma \to \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M\,N : \tau}\ (\textrm{T-App})$$

- Sea
  - $\mathbb{W}(U) = \Gamma_1 \triangleright M : \tau$
  - $\mathbb{W}(V) = \Gamma_2 \triangleright N : \rho$
- Sea

$$S = MGU\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1 \wedge x : \sigma_2 \in \Gamma_2\}$$
$$\cup$$
$$\{\tau \doteq \rho \to t\} \text{ con } t \text{ una variable fresca}$$

- Entonces

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

Apliquémoslo

Utilizar el algoritmo $\mathbb{W}$ para las siguientes expresiones:

- $\lambda f \,.\, \lambda x \,.\, f(f\ x)$
- $x(\lambda x \,.\, x)$
- $\lambda x \,.\, x\ y\ x$

Extensiones al algoritmo

En general

- Agregar casos nuevos al algoritmo.
- Menos frecuentemente, modificar casos existentes.

Para incorporar nuevos términos

- Nuevas reglas de tipado $\Rightarrow$ nuevos casos del algoritmo $\mathbb{W}$.
- Anotar las expresiones con sus tipos.

Extensión del lenguaje

Abstracciones sobre pares

$$M ::= \ldots | \lambda\langle x, y\rangle : \langle\sigma \times \tau\rangle . M$$

$$M' ::= \ldots | \lambda\langle x, y\rangle . M'$$

$$\frac{\Gamma, x : \sigma, y : \tau \triangleright M : \rho}{\Gamma \triangleright \lambda\langle x, y\rangle : \langle\sigma \times \tau\rangle . M : \langle\sigma \times \tau\rangle \to \rho}$$

Extender el algoritmo

$$\mathbb{W}(\lambda\langle x, y\rangle . U) \stackrel{\text{def}}{=} \Gamma' \triangleright \lambda\langle x, y\rangle : \langle\tau_x \times \tau_y\rangle . M : \langle\tau_x \times \tau_y\rangle \to \rho$$

donde

$$\tau_x = \begin{cases} \alpha \text{ si } x : \alpha \in \Gamma \\ \text{variable fresca si no.} \end{cases} \qquad \tau_y = \begin{cases} \beta \text{ si } y : \beta \in \Gamma \\ \text{variable fresca si no.} \end{cases}$$

$$\Gamma' = \Gamma \ominus \{x, y\}$$

Extensiones del lenguaje

Listas

$\sigma ::= \ldots \mid [\sigma]$

$M, N, O ::= \ldots \mid [\,]_\sigma \mid M :: N \mid \mathit{Case}\ M\ \mathit{of}\ [\,] \rightsquigarrow N \,;\, h :: t \rightsquigarrow O$

$$\frac{}{\Gamma \triangleright [\,]_\sigma\ : [\sigma]} \qquad \frac{\Gamma \triangleright M : \sigma \qquad \Gamma \triangleright N : [\sigma]}{\Gamma \triangleright M :: N : [\sigma]}$$

$$\frac{\begin{array}{c}\Gamma \triangleright M : [\sigma] \qquad \Gamma \triangleright N : \tau \\ \Gamma \cup \{h : \sigma, t : [\sigma]\} \triangleright O : \tau\end{array}}{\Gamma \triangleright \mathit{Case}\ M\ \mathit{of}\ [\,] \rightsquigarrow N \,;\, h :: t \rightsquigarrow O : \tau}$$

Extender el algoritmo

$\mathbb{W}([\,]) \stackrel{\text{def}}{=} ?$

$\mathbb{W}(U_1 :: U_2) \stackrel{\text{def}}{=} ?$

$\mathbb{W}(\mathit{Case}\ U_1\ \mathit{of}\ [\,] \rightsquigarrow U_2 \,;\, h :: t \rightsquigarrow U_3) \stackrel{\text{def}}{=} ?$

Ahora usémoslo

$\mathbb{W}(\mathit{Case}\ \mathrm{succ}(0) :: x\ \mathit{of}\ [\,] \rightsquigarrow x \,;\, x :: y \rightsquigarrow \mathrm{succ}(x) :: [\,]) = ?$

Otra extensión
Switch de naturales

Switch

Extender el algoritmo de inferencia $\mathbb{W}$ para que soporte el tipado del *switch* de números naturales, similar al de C o C++. La extensión de la sintaxis es la siguiente:

$M = \ldots \mid \texttt{switch}\ M\ \{\texttt{case}\ \underline{n_1} :\ M_1\ \ \ldots\ \texttt{case}\ \underline{n_k} :\ M_k\ \texttt{default} :\ M_{k+1}\}$

donde cada $\underline{n_i}$ es un numeral (un *valor* de tipo Nat, como 0, succ(0), succ(succ(0)), etc.). Esto forma parte de la sintaxis y no hace falta verificarlo en el algoritmo.

La regla de tipado es la siguiente:

$$\frac{\begin{array}{c}\Gamma \triangleright M : Nat \quad \forall i,j(1 \leq i,j \leq k \wedge i \neq j \Rightarrow n_i \neq n_j) \\ \Gamma \triangleright N_1 : \sigma \quad \ldots \quad \Gamma \triangleright N_k : \sigma \quad \Gamma \triangleright N : \sigma\end{array}}{\Gamma \triangleright \texttt{switch}\ M\ \{\texttt{case}\ \underline{n_1} :\ N_1\ \ldots\ \texttt{case}\ \underline{n_k} :\ N_k\ \texttt{default} : N\} : \sigma}$$

Otra extensión del lenguaje
Letrec

En este ejercicio modificaremos el algoritmo de inferencia para incorporar la posibilidad de utilizar letrec en nuestro cálculo.

$M ::= \ldots \mid \text{letrec } f = M \text{ in } N$

Permite por ejemplo representar el factorial de 10 de la siguiente manera:

$\text{letrec } f = (\lambda x : \text{Nat} . \text{if isZero}(x) \text{ then } \underline{1} \text{ else } x \times f \ (\text{Pred}(x))) \text{ in } f \ \underline{10}$

Para ello se agrega la siguiente regla de tipado:

$$\frac{\Gamma \cup \{f : \pi \to \tau\} \rhd M : \pi \to \tau \qquad \Gamma \cup \{f : \pi \to \tau\} \rhd N : \sigma}{\Gamma \rhd \text{letrec } f = M \text{ in } N : \sigma}$$

Extendemos el algoritmo

$$\frac{\Gamma \cup \{f : \pi \to \tau\} \rhd M : \pi \to \tau \qquad \Gamma \cup \{f : \pi \to \tau\} \rhd N : \sigma}{\Gamma \rhd \text{letrec } f = M \text{ in } N : \sigma}$$

$\mathbb{W}(\text{letrec } f = U_1 \text{ in } U_2) \stackrel{\text{def}}{=} S\,\Gamma'_1 \cup S\,\Gamma'_2 \rhd S\,(\text{letrec } f = M_1 \text{ in } M_2) : S\,\tau_2$
donde

- $\mathbb{W}(U_1) = \Gamma_1 \rhd M_1 : \tau_1$
- $\mathbb{W}(U_2) = \Gamma_2 \rhd M_2 : \tau_2$
- $\tau_{f1} = \begin{cases} \alpha_1 \text{ si } f : \alpha_1 \in \Gamma_1 \\ \text{variable fresca en otro caso.} \end{cases}$
- $\tau_{f2} = \begin{cases} \alpha_2 \text{ si } f : \alpha_2 \in \Gamma_2 \\ \text{variable fresca en otro caso.} \end{cases}$
- $\Gamma'_1 = \Gamma_1 \ominus \{f\}$ y $\Gamma'_2 = \Gamma_2 \ominus \{f\}$
- $\begin{array}{rcl} S & = & \mathtt{mgu}\ \{\tau_{f1} \doteq \tau_{f2}, \tau_1 \doteq t_1 \to t_2, \tau_1 \doteq \tau_{f1}\} \\ & \cup & \{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma'_1, x : \sigma_2 \in \Gamma'_2\} \end{array}$

  t_1 y t_2 variables frescas

Otra forma de escribirlo

$$\frac{\Gamma \cup \{f : \pi \to \tau\} \rhd M : \pi \to \tau \qquad \Gamma \cup \{f : \pi \to \tau\} \rhd N : \sigma}{\Gamma \rhd \mathsf{letrec}\ f = M \ \mathsf{in}\ N : \sigma}$$

$\mathbb{W}(\mathsf{letrec}\ f = U_1 \ \mathsf{in}\ U_2) \stackrel{\text{def}}{=} S\,\Gamma'_1 \cup S\,\Gamma'_2 \rhd S\,(\mathsf{letrec}\ f = M \ \mathsf{in}\ N) : S\,\sigma$
donde

- $\mathbb{W}(U_1) = \Gamma_1 \rhd M : \gamma$
- $\mathbb{W}(U_2) = \Gamma_2 \rhd N : \sigma$
- $\tau_f = \begin{cases} \alpha_1 \text{ si } f : \alpha_1 \in \Gamma_1 \\ \alpha_2 \text{ si } f \notin \mathtt{dom}(\Gamma_1) \text{ y } f : \alpha_2 \in \Gamma_2 \\ \text{variable fresca en otro caso.} \end{cases}$
- $\Gamma'_1 = \Gamma_1 \ominus \{f\}$ y $\Gamma'_2 = \Gamma_2 \ominus \{f\}$
- $\begin{array}{rcl} S & = & \mathtt{mgu}\ \{\gamma \doteq t_1 \to t_2, \gamma \doteq \tau_f\} \\ & \cup & \{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\} \end{array}$

  t_1 y t_2 variables frescas

Moraleja

Algunas conclusiones

- Los llamados recursivos devuelven un contexto, un término anotado y un tipo. No podemos asumir nada sobre ellos.
- Cuando la regla tiene tipos iguales o tipos con una forma específica: unificar.
- Si hay contextos repetidos en las premisas, unificarlos.
- Cuando la regla liga variables:
  - Obtener su tipo del Γ obtenido recursivamente.
  - Si no figuran: variable fresca.
  - Sacarlas del Γ del resultado (y del que se vaya a unificar).
- Decorar los términos según corresponda.
- Si la regla tiene restricciones adicionales, se incorporan como posibles casos de falla.

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?

Taller de Programación
Algoritmo de Inferencia de Tipos

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Septiembre 2017

Algoritmo de Inferencia

Entrada	una expresión U de λ sin anotaciones
Salida	el juicio de tipado $\Gamma \rhd M : \sigma$ más general para U, o bien una falla si U no es tipable
Estrategia	recursión sobre la estructura de U

Tipos de datos y funciones auxiliares

Expresiones de tipo:

$$\sigma \quad ::= \quad s \,|\, Nat \,|\, Bool \,|\, \sigma \to \tau$$

Tipos de datos y funciones auxiliares

Expresiones de tipo:

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

En Haskell:

```
data Type = TVar Int
          | TNat
          | TBool
          | TFun Type Type
```

Tipos de datos y funciones auxiliares

Expresiones anotadas:

$$\begin{array}{rcl}
M & ::= & x \\
 & | & 0 \mid \mathit{succ}(M) \mid \mathit{pred}(M) \mid \mathit{iszero}(M) \\
 & | & \mathit{true} \mid \mathit{false} \mid \mathit{if}\ \ M\ \ \mathit{then}\ \ P\ \ \mathit{else}\ \ Q \\
 & | & \lambda x : \sigma . M \\
 & | & M\, N
\end{array}$$

Tipos de datos y funciones auxiliares

Expresiones sin anotar:

$$\begin{array}{rcl} M & ::= & x \\ & \mid & \mathit{0} \mid \mathit{succ}(M) \mid \mathit{pred}(M) \mid \mathit{iszero}(M) \\ & \mid & \mathit{true} \mid \mathit{false} \mid \mathit{if}\ \ M\ \ \mathit{then}\ \ P\ \ \mathit{else}\ \ Q \\ & \mid & \lambda x.M \\ & \mid & M\,N \end{array}$$

Tipos de datos y funciones auxiliares

```
type Symbol = String

data Exp a = VarExp Symbol
           | ZeroExp
           | SuccExp (Exp a)
           | PredExp (Exp a)
           | IsZeroExp (Exp a)
           | TrueExp
           | FalseExp
           | IfExp (Exp a) (Exp a) (Exp a)
           | LamExp Symbol a (Exp a)
           | AppExp (Exp a) (Exp a)
```

Tipos de datos y funciones auxiliares

```
type Symbol = String

data Exp a = VarExp Symbol
           | ZeroExp
           | SuccExp (Exp a)
           | PredExp (Exp a)
           | IsZeroExp (Exp a)
           | TrueExp
           | FalseExp
           | IfExp (Exp a) (Exp a) (Exp a)
           | LamExp Symbol a (Exp a)
           | AppExp (Exp a) (Exp a)

type AnnotExp = Exp Type
```

Tipos de datos y funciones auxiliares

```
type Symbol = String

data Exp a = VarExp Symbol
           | ZeroExp
           | SuccExp (Exp a)
           | PredExp (Exp a)
           | IsZeroExp (Exp a)
           | TrueExp
           | FalseExp
           | IfExp (Exp a) (Exp a) (Exp a)
           | LamExp Symbol a (Exp a)
           | AppExp (Exp a) (Exp a)

type AnnotExp = Exp Type

type PlainExp = Exp ()
```

Tipos de datos y funciones auxiliares

Contexto

```
emptyEnv :: Env
extendE  :: Env -> Symbol -> Type -> Env
removeE  :: Env -> Symbol -> Env
evalE    :: Env -> Symbol -> Type
joinE    :: [Env] -> Env
domainE  :: Env -> [Symbol]
```

Sustituciones

```
emptySubst :: Subst
extendS :: Int -> Type -> Subst -> Subst
```

Sustituciones

```
emptySubst :: Subst
extendS :: Int -> Type -> Subst -> Subst

class Substitutable a where
    (<.>) :: Subst -> a -> a
    instance Substitutable Type   -- subst <.> t
    instance Substitutable Env    -- subst <.> env
    instance Substitutable Exp    -- subst <.> e
```

Tipos de datos y funciones auxiliares

Sustitiuciones y unificación

Sustituciones

```
emptySubst :: Subst
extendS :: Int -> Type -> Subst -> Subst

class Substitutable a where
    (<.>) :: Subst -> a -> a
    instance Substitutable Type   -- subst <.> t
    instance Substitutable Env    -- subst <.> env
    instance Substitutable Exp    -- subst <.> e
```

Unificación

```
type UnifGoal = (Type, Type)
data UnifResult = UOK Subst | UError Type Type
mgu :: [UnifGoal] -> UnifResult
```

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
```

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
inferType (VarExp x) = ...
    ...
```

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
inferType (VarExp x) = ...
    ...
```

$$\mathbb{W}(x) \stackrel{\text{def}}{=} \{x : s\} \triangleright x : s, \quad s \text{ variable fresca}$$

La función de inferencia

```
type TypingJudgment = (Env, AnnotExp, Type)
data Result a = OK a | Error String

inferType :: PlainExp → Result TypingJudgment
infer' :: PlainExp
                → Int
                → Result (Int, TypingJudgment)
```

¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos

  VarExp ZeroExp LamExp AppExp
- Usar *pattern matching* sobre Exp

¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos

  VarExp ZeroExp LamExp AppExp
- Usar *pattern matching* sobre Exp

Tip

```
let x = expr1 in expr2
case expr of  Pattern1 -> res1
              ...
              Paternn -> resn
```

¡A programar!

Consigna

- Completar archivo `TypeInference.hs`
- Definir la función `inferType` utilizando `infer'`
- Definir la función `infer'` para los casos

  VarExp ZeroExp LamExp AppExp
- Usar *pattern matching* sobre Exp

Prueba

- Archivo `Examples.hs`
- `expr` *n* `:: String`
- `inferExpr :: String` $\rightarrow$ `Doc`

Ejemplo

```
infer'(SuccExp e) n =
```

Ejemplo

```
infer'(SuccExp e) n =

case infer' e n of
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->





    res@(Error _) ->
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->



    res@(Error _) -> res
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of









    res@(Error _) -> res
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UOK subst ->




            UError u1 u2 ->

    res@(Error _) -> res
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UOK subst ->







            UError u1 u2 ->
                uError u1 u2
    res@(Error _) -> res
```

Ejemplo

```
infer'(SuccExp e) n =
case infer' e n of
    OK ( n', (env', e', t') ) ->
        case mgu [ (t', TNat) ] of
            UOK subst ->
                OK ( n', (



                         ) )
            UError u1 u2 ->
                uError u1 u2
    res@(Error _) -> res
```

Ejemplo

```
infer'(SuccExp e) n =

case infer' e n of

    OK ( n', (env', e', t') ) ->

        case mgu [ (t', TNat) ] of

            UOK subst ->

                OK ( n', (

                          subst <.> env',
                          subst <.> SuccExp e',
                          TNat

                         ) )

            UError u1 u2 ->

                uError u1 u2

    res@(Error _) -> res
```

Taller de Inferencia de Tipos
Machete

PLP

Septiembre 2017

1 Algoritmo de inferencia

- $\mathbb{W}(x) \stackrel{\text{def}}{=} \{x : s\} \triangleright x : s, \quad s$ variable fresca
- $\mathbb{W}(\mathit{0}) \stackrel{\text{def}}{=} \emptyset \triangleright \mathit{0} : nat$
- $\mathbb{W}(true) \stackrel{\text{def}}{=} \emptyset \triangleright true : bool$
- $\mathbb{W}(false) \stackrel{\text{def}}{=} \emptyset \triangleright false : bool$
- $\mathbb{W}(succ(U)) \stackrel{\text{def}}{=} S\Gamma \triangleright S\, succ(M) : nat$ donde
  - $\mathbb{W}(U) = \Gamma \triangleright M : \tau$
  - $S = MGU\{\tau \doteq nat\}$
- $\mathbb{W}(pred(U)) \stackrel{\text{def}}{=} S\Gamma \triangleright S\, pred(M) : nat$ donde
  - $\mathbb{W}(U) = \Gamma \triangleright M : \tau$
  - $S = MGU\{\tau \doteq nat\}$
- $\mathbb{W}(iszero(U)) \stackrel{\text{def}}{=} S\Gamma \triangleright S\, iszero(M) : bool$ donde
  - $\mathbb{W}(U) = \Gamma \triangleright M : \tau$
  - $S = MGU\{\tau \doteq nat\}$
- $\mathbb{W}(if\ U\ then\ V\ else\ W) \stackrel{\text{def}}{=} S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_3 \triangleright S(if\ M\ then\ P\ else\ Q) : S\sigma$ donde
  - $\mathbb{W}(U) = \Gamma_1 \triangleright M : \rho$
  - $\mathbb{W}(V) = \Gamma_2 \triangleright P : \sigma$
  - $\mathbb{W}(W) = \Gamma_3 \triangleright Q : \tau$
  - $S = MGU\{\sigma \doteq \tau, \rho \doteq bool\} \cup \{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i, x : \sigma_2 \in \Gamma_j, i \neq j\}$
- $\mathbb{W}(\lambda x.U) \stackrel{\text{def}}{=} \Gamma' \triangleright \lambda x : \tau'.M : \tau' \to \rho$ donde
  - $\mathbb{W}(U) = \Gamma \triangleright M : \rho$
  - $\tau' = \begin{cases} \alpha \text{ si } x : \alpha \in \Gamma \\ s \text{ con } s \text{ variable fresca en otro caso} \end{cases}$
  - $\Gamma' = \Gamma \ominus \{x\}$
- $\mathbb{W}(U\, V) \stackrel{\text{def}}{=} S\Gamma_1 \cup S\Gamma_2 \triangleright S(M\, N) : St$ donde
  - $\mathbb{W}(U) = \Gamma_1 \triangleright M : \tau$
  - $\mathbb{W}(V) = \Gamma_2 \triangleright N : \rho$
  - t variable fresca
  - $S = MGU\{\tau \doteq \rho \to t\} \cup \{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\}$

2 Código auxiliar

2.1 Expresiones

```
data Exp a = VarExp Symbol |
             ZeroExp |
             SuccExp (Exp a)|
             PredExp (Exp a) |
             IsZeroExp (Exp a) |
             TrueExp |
             FalseExp |
             IfExp (Exp a) (Exp a) (Exp a) |
             LamExp Symbol a (Exp a) |
             AppExp (Exp a) (Exp a)

type Symbol = String
type PlainExp = Exp ()
type AnnotExp = Exp Type
```

2.2 Tipos

```
data Type = TVar Int | TNat | TBool | TFun Type Type
```

2.3 Contexto

```
emptyEnv :: Env
extendE :: Env -> Symbol -> Type -> Env
removeE :: Env -> Symbol-> Env
evalE :: Env -> Symbol -> Type
joinE :: [Env] -> Env
domainE :: Env -> [Symbol]
```

2.4 Sustituciones

```
emptySubst :: Subst
extendS :: Int -> Type -> Subst -> Subst

class Substitutable a where
    (<.>) :: Subst -> a -> a
    instance Substitutable Type   -- subst <.> t
    instance Substitutable Env    -- subst <.> env
    instance Substitutable Exp    -- subst <.> e
```

2.5 Unificación

```
type UnifGoal = (Type, Type)
data UnifResult = UOK Subst | UError Type Type
mgu :: [UnifGoal] -> UnifResult
```