

Introducción a Programación Funcional

Programación Avanzada

UNRC

Pablo Castro

Un Lenguaje Funcional Simple

Definiremos un lenguaje funcional que tiene:

- Tipos básicos
- Expresiones (aritméticas, booleanas, etc)
- Definiciones de funciones.

Este lenguaje nos permite expresar **programas funcionales**.

Tipos y Expresiones

Para la definición de programas funcionales utilizamos diferentes tipos básicos:

- Booleanos: *true, false*
- Numéricos: 0, 1, 1.111, 3.1415
- Caracteres: *'a', 'b', 'c', ...*

Y las expresiones correspondientes:

$p \wedge q$

$\neg p$

$x + 15 * 5$

Buscar más
ejemplos

Funciones

Para definir una función se necesitan dos cosas

- Su perfil, diciendo que parámetros toma y que devuelve,
- Su definición por medio de expresiones

Formalmente:

Nombre de la
función

$f : A \rightarrow B$

Perfil

$f.x \doteq E$

Parámetro formal

Expresión
que define la
función

Ejemplo

$dup : Num \rightarrow Num$

$dup\ x \doteq x + x$

Funciones Recursivas

Las funciones recursivas nos permiten hacer cálculos complejos

Una función f se dice **recursiva** si en su definición aparece f

Veamos un ejemplo:

$pow : Num \rightarrow Num$

$pow.x \doteq \text{if } x = 0 \text{ then } 1 \text{ else } 2 * pow.(x - 1)$

Caso Base

Caso Recursivo

Evaluación

Para evaluar cualquier función utilizamos sustituciones:

$pow.3$

$= [Def. \textit{pow}]$

$2 * (pow.2)$

$= [Def. \textit{pow}]$

$2 * (2 * (pow.1))$

$= [Def. \textit{pow}]$

$2 * (2 * (2 * (pow.0)))$

$= [Def. \textit{pow}]$

$2 * (2 * (2 * (1)))$

$= [Arit.]$

8

Si no tuviera caso base la
función no estaría definida!

Caso Base

Pattern Matching

Podemos definir las funciones por casos según la forma de sus argumentos:

Si el
parámetro
es 0

$pow : Num \rightarrow Num$

$pow.0 = 1$

$pow.(n + 1) = 2 * pow.n$

Si el parámetro **tiene la
forma** $n+1$

El pattern matching hace
transparente cómo los
diferentes casos
depende de la forma de
los parámetros

Programas Funcionales

Un **programa funcional** es un conjunto definiciones de funciones

Dado un programa funcional podemos evaluar expresiones siguiendo las definiciones dadas:

$fact : Num \times Num \rightarrow Num$

$fact(x, y) \doteq \text{if } x = 0 \text{ then } y \text{ else } fact.(n - 1, x * y)$

Otra definición posible de factorial

$factorial : Num \rightarrow Num$

$factorial.n \doteq fact(n, 1)$

Evaluar $fact.3$

Listas

Las listas son una secuencia lineal de elementos del mismo tipo:

$[x_0, x_1, x_2, \dots, x_n]$

Lista con n elementos

Si x_0, x_1, x_2, \dots son de tipo A, entonces la lista tiene tipo [A].

$[1 + 1, 2 * 3 + 100, 3/10]$

Tiene tipo [Num]

$[true, false, true \wedge false]$

Tiene tipo [Bool]

$[[], [1, 2], [3 + 4, 15 * 100]]$

Tiene tipo [[Num]]

Construyendo Listas

Las listas se definen inductivamente mediante dos operaciones:

$[]$

La lista vacía, es una lista sin elementos

$\triangleright : A \rightarrow [A] \rightarrow [A]$

Concatena un elemento a la cabeza de una lista

Toda lista se puede definir con estos constructores.

$[1, 2, 3]$

Se escribe:

$1 \triangleright 2 \triangleright 3 \triangleright []$

$[x_0, x_1, x_2, \dots, x_n]$

Se escribe:

$x_0 \triangleright x_1 \triangleright x_2 \cdots \triangleright x_n \triangleright []$

Tuplas

Dados tipos A y B ,

$A \times B$

El conjunto de todos los pares (a,b) en donde a tiene tipo A y b tiene tipo B

A diferencia de las listas sus elementos no tienen que tener el mismo tipo

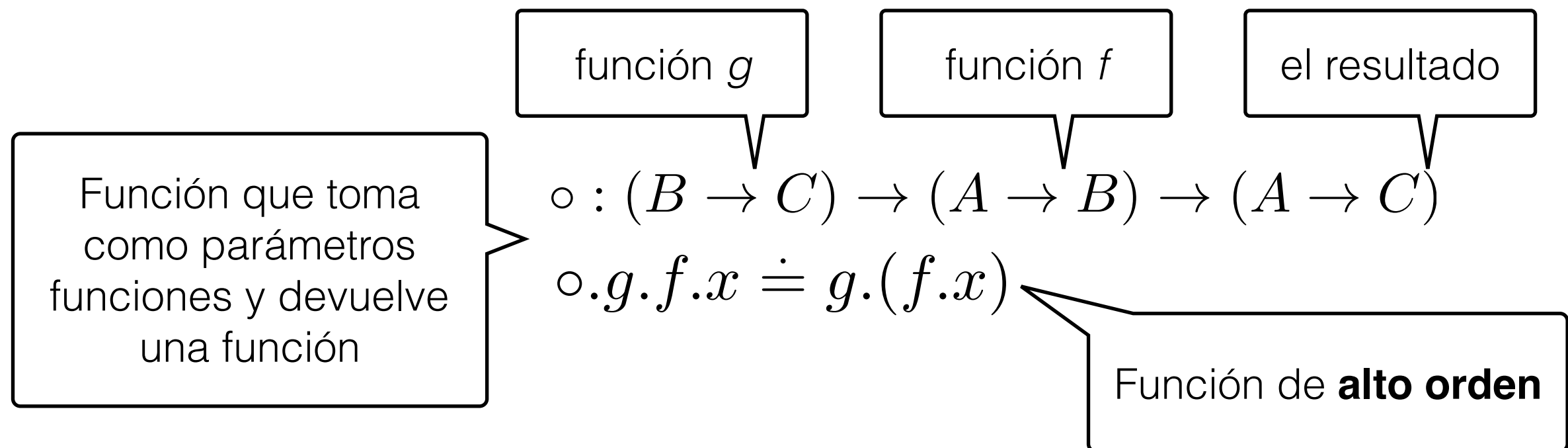
$(1, 2)$ tiene tipo: $Num \times Num$
 $('a', 1)$ tiene tipo: $Char \times Num$

Para acceder a los elementos de una tupla usamos las proyecciones:

$$\langle \forall i : 1 \leq i \leq n : \langle a_1, \dots, a_n \rangle . i = a_i \rangle$$

Funciones como Tipo de Datos

Las funciones se consideran otro tipo de datos más.



Las funciones no son diferentes de cualquier otro tipo en funcional.

Curricación

Toda función:

$$f : A_0 \times \cdots \times A_n \rightarrow B$$

Se puede reescribir como:

$$f : A_0 \rightarrow (A_1 \rightarrow \dots (A_n \rightarrow B))$$

Este proceso se llama **curricación**.

En honor a Haskell Curry

Podemos definir una función *curry* para hacer esto:

$$curry : (A \times B \rightarrow C) \rightarrow (A \rightarrow (B \rightarrow C))$$

$$curry\ f\ x\ y \doteq f(x, y)$$

Sistema de Tipos

Cada expresión bien formada es de algún tipo:

- Tipo básicos: *Num*, *Bool*, *Char*,
- Tipos Estructurados, Listas ($[A]$), Tuplas ($A \times B$) o Funciones ($A \rightarrow B$).

Cuando una expresión E es de tipo T escribimos:

La expresiones que no pueden
asignarsele un tipo son
erróneas, o mal tipadas

$E : T$

Extensionalidad

Es una de las propiedades más importantes de funciones:

$$\langle \forall f, g :: \langle \forall x :: f.x = g.x \rangle \Rightarrow f = g \rangle$$

Dos funciones son iguales, si retornan lo mismo para iguales parámetros

Permite demostrar igualdad de funciones:

$$((h \circ g) \circ f).x$$

$$\equiv [\text{Def. } \circ]$$

$$(h \circ g).(f.x)$$

$$\equiv [\text{Def. } \circ]$$

$$h.(g.(f.x))$$

$$\equiv [\text{Def. } \circ]$$

$$h.((g \circ f).x)$$

$$\equiv [\text{Def. } \circ]$$

$$(h \circ (g \circ f)).x$$

Lo cual implica: $(h \circ g) \circ f = h \circ (g \circ f)$

Definición por Casos

Podemos definir funciones por casos:

$$f : A \rightarrow B$$

$$f.x \doteq B_0 \rightarrow E_0$$

$$\square B_1 \rightarrow E_1$$

$$\vdots$$

$$\square B_n \rightarrow E_n$$

Es una función definida con n
casos diferentes

Un ejemplo:

$$\text{max} : \text{Num} \rightarrow \text{Num} \rightarrow \text{Num} \rightarrow \text{Num}$$

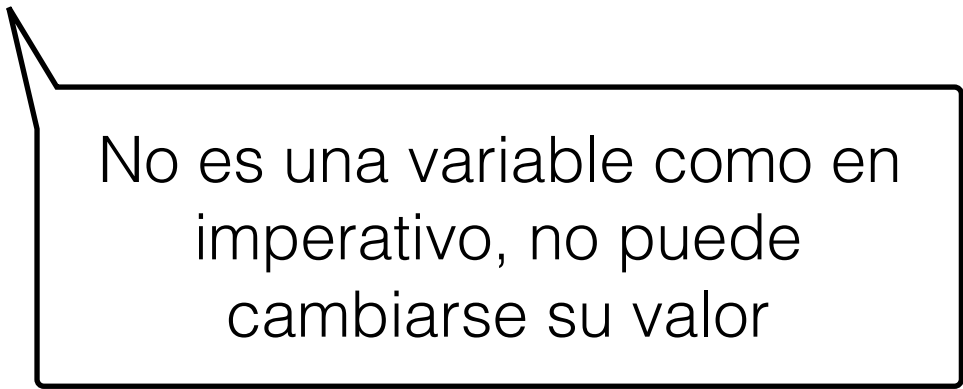
$$\text{max}.x.y.z \doteq x \leq y \wedge z \leq y \rightarrow y$$

$$\square y \leq x \wedge z \leq x \rightarrow x$$

$$\square x \leq z \wedge y \leq z \rightarrow z$$

Definiciones Locales

Podemos introducir definiciones locales para evitar redundancia y mejorar la legibilidad:

$$raiz1 : Num \rightarrow Num \rightarrow Num \rightarrow Num$$
$$raiz1.x \doteq (-b - sqrt.disc) / (2 * a)$$
$$[disc = b^2 - 4 * a * c]$$


No es una variable como en imperativo, no puede cambiarse su valor

La Importancia de las Expresiones

En funcional, la forma de computar consiste en evaluar expresiones:

- Intuitivamente, “ $5+10$ ” debe evaluar a “15”,
- Debemos decidir como evaluar expresiones como:
[$2+3$, pow.2]

Para resolver esto necesitamos las nociones de:

- Expresiones canónicas,
- Formal normal.

Expresiones Canónicas

Muchas expresiones denotan el mismo valor:

$9, \text{pow}.3, 3 * 3, 10 - 1, \dots$

De cada conjunto de valores que denotan el mismo valor, se elige uno que es llamado la **expresión canónica** para ese valor

Ejemplos:

$9, \text{pow}.3, 3 * 3, 10 - 1, \dots$ expresión canónica: 9

$[1]++[], [1], 1 \triangleright [], \dots$ expresión canónica: [1]

Expresiones Canónicas

Definamos las expresiones canónicas para cada tipo:

- Booleanas: *true, false*
- Números: $0, 1, 2, 3, -1, 3.1415, \dots$ es decir su representación decimal.
- Pares: (E_0, E_1) en donde E_0 y E_1 son expresiones canónicas.
- Listas: $[E_0, E_1, \dots, E_n]$ donde E_i son expresiones canónicas.

Formal Normal

Dada una expresión, su **forma normal** es la expresión canónica la cual representa el mismo valor

Hay expresiones que no tienen formal normal:

$inf : Num$

$inf \doteq inf + 1$

$err : Num$

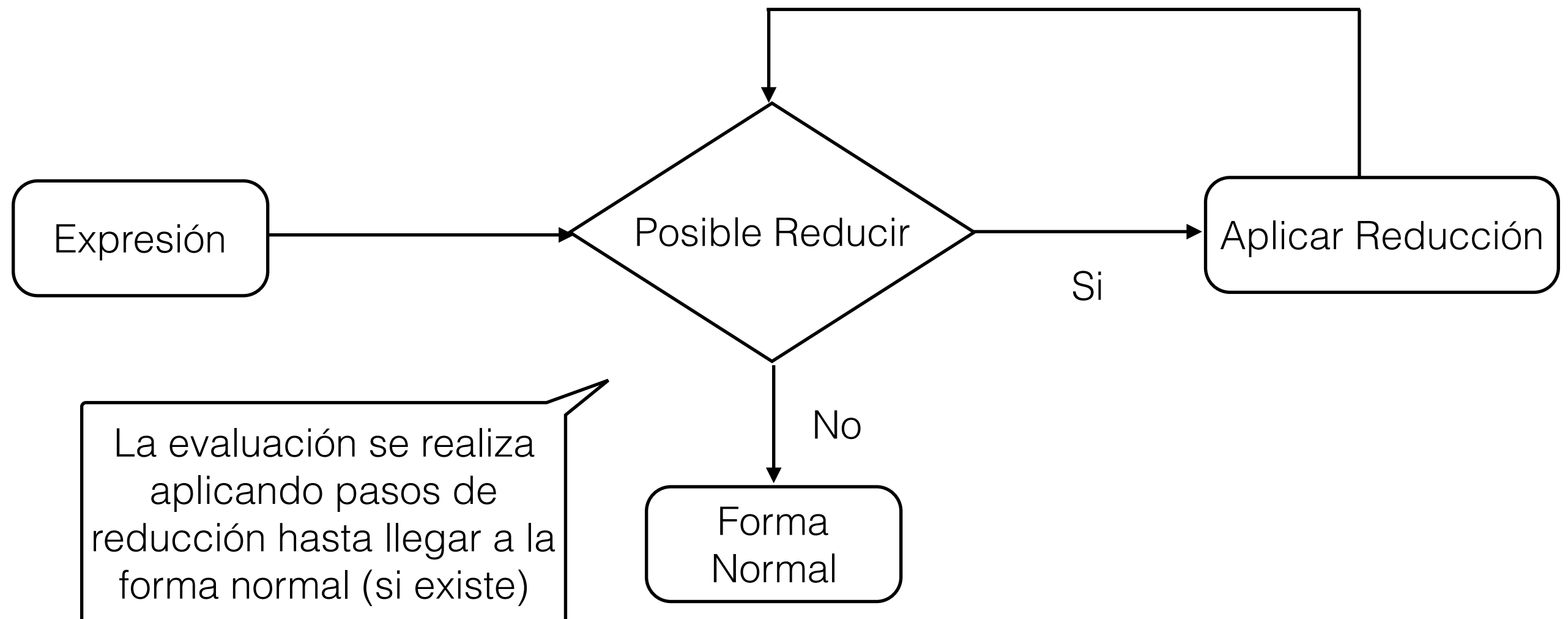
$err \doteq \frac{1}{0}$



No tienen formal normal

Evaluación de Expresiones

La **evaluación de una expresión**, dado un programa funcional, es el proceso de encontrar la forma normal de la expresión usando las definiciones dadas.



Formas de Evaluación

Veamos un ejemplo de evaluación: $cuad : Num \rightarrow Num$
 $cuad.x \doteq x * x$

$cuad.(3 * 5)$
 $= [Aritmética]$
 $cuad.15$

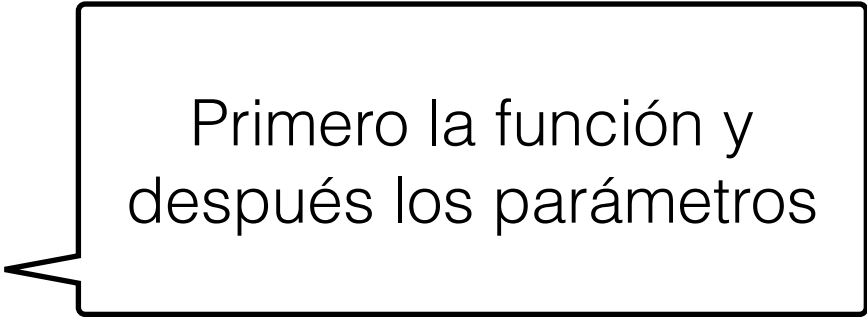
Primero evaluamos el
parámetro y después la
función

$= [Def. \textit{pow}]$
 $15 * 15$
 $= [Aritmética]$
 225

Formas de Evaluación

Podríamos evaluar la expresión de otra forma:

$cuad.(3 * 5)$
= [Def. *pow*]
 $(3 * 5) * (3 * 5)$
= [Aritmética]
 $15 * (3 * 5)$
= [Aritmética]
 $15 * 15$
= [Aritmética]
 225



Primero la función y
después los parámetros

Evaluaciones Aplicativa y Normal

Orden Aplicativo: se reduce siempre la expresión más adentro y más a la izquierda

Orden Normal: se reduce siempre la expresión más afuera y más a la izquierda

Propiedad: Si hay una forma normal el orden normal siempre la encuentra

Aplicativa vs Normal

Veamos la siguiente función:

$$K.x.y \doteq x$$

Evaluación Aplicativa:

$$k.3.inf \\ = [\text{Def. } inf]$$

$$k.3.(inf + 1) \\ = [\text{Def. } inf]$$

$$k.3.((inf + 1) + 1) \\ = [\text{Def. } inf]$$

$$k.3.(((inf + 1) + 1) + 1)$$

⋮

No termina

Evaluación Normal:

$$k.3.inf \\ = [\text{Def. } k] \\ 3$$

Devuelve la
forma normal

Evaluación Lazy

Evaluación Lazy: Se evalúa el término más afuera de izquierda a derecha, en donde la misma expresión no es evaluada dos veces

$cuad.cuad.3$

$= [Def. \text{ cuad}]$

La expresiones son evaluadas una sola vez

$x * x$

$= [Def. \text{ cuad}]$

A lo sumo usa tantos pasos como la evaluación aplicativa

$x * x$

$\llbracket x = 3 * 3 \rrbracket$

$= [Aritmética]$

Siempre encuentra la forma normal, cuando existe

$x * x$

$\llbracket x = 9 \rrbracket$

$= [Reemplazo]$

$9 * 9$

$= [Aritmética]$

81

Utiliza más memoria que la forma aplicativa