Introducción a Programación Funcional

Algoritmos y Estructuras de Datos I

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: `1, 2, 3.14`
- Caracteres: `'a', 'b', 'c' ...`

Y las expresiones correspondientes:

`p && q, 2*3, etc`

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:

Perfil

Nombre de la función

```
f :: a -> b
f x = E
```

Parámetro formal

Expresión que define la función

Ejemplo

```
dup :: Int -> Int
dup x = x+x
```

Funciones Recursivas

Las funciones recursivas nos permiten hacer cómputos complejos

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

Veamos un ejemplo:

```
pow :: Int -> Int
pow n = if n == 0 then 1 else 2 * pow (n-1)
```

Caso Base

Caso Recursivo

Evaluación

Para evaluar cualquier función utilizamos sustituciones:

```
pow 3
= [def. pow]
2 * (pow 2)
= [def. pow]
2 * (2 * pow 1)
= [def. pow]
2 * (2 * (2 * pow 0))
= [def. pow, caso base]
2 * (2 * (2 * 1))
= [Arit.]
8
```

Sustituimos por la definición de la función

Caso Base

Pattern Matching

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

Si el parámetro es 0

```
pow 0 = 1
pow n = 2 * pow (n-1)
```

Si el parámetro no es 0

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

Se evalúa desde arriba hacia abajo, el primer patrón que coincide es el que se ejecuta.

Programas Funcionales

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

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

```
fact :: (int,int) -> int
fact (n,m) = if n == 0 then m else fact(n-1,n*m)
```

Otra definición posible de factorial

```
fact (5,1)
```

Calcula factorial de 5

Listas

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

```
[x1,x2,x3,x4,x5,x6,x7]
```

Lista con n elementos

Si $x_0, x_1, x_2, \ldots$ son de tipo A, entonces la lista tiene tipo $[\mathrm{A}]$.

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

Tiene tipo [Int]

```
[True, False, True && False]
```

Tiene tipo [Bool]

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

Tiene tipo [[Int]]

Construyendo Listas

Las listas se definen inductivamente mediante dos operaciones:

`[]` — La lista vacía, es una lista sin elementos

`:` — Concatena un elemento a la cabeza de una lista

Toda lista se puede definir con estos constructores.

`[1,2,3]` se escribe: `1:(2:(3:[]))`

`[1,2,3,4,5]` se escribe: `1:(2:(3:(4:(5:[]))))`

Tuplas

Dados tipos `a,b`

`(a,b)`

El conjunto de todos los pares *(i,j)* en donde i tiene tipo **a** y **j** tiene tipo **b**

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

`(1,2)` tiene tipo: `(Int,Int)`

`('a',1)` tiene tipo: `(Char, Int)`

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

```
fst ('a',1) = 'a'

snd ('a',1) = 1
```

Funciones como Tipo de Datos

Las funciones se consideran otro tipo de datos más.

función *g*

función *f*

el resultado

Función que toma como parámetros funciones y devuelve una función

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

Función de **alto orden**

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

Currificación

Toda función: `f :: (a1,...,an) -> b`

Se puede reescribir como: `f :: a1 -> (a2 -> (a3... -> b) ... )`

Este proceso se llama **currificación.**

En honor a Haskell Curry

Podemos definir una función *curry* para hacer esto:

```
curry :: ((a,b)->c) -> (a -> b -> c)
curry f x y = 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,b)` o Funciones (`a->b`).

Cuando una expresión `E` es de tipo `T` escribimos: `E::t`

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

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.g).f) x
= [def .]
(h.g) (f x)
= [def. .]
h (g (f x))
= [def .]
h ((g.f) x)
= [def .]
(h.(g.f)) x
```

Lo cual implica: `((h.g).f) = (h.(g.f))`

Definición por Casos

Podemos definir funciones por casos:

```
f :: a -> b
f x | B0 = E0
    | B1 = E1
    .
    .
    .
    | Bn = EN
```

Es una función definida con N casos diferentes

Un ejemplo:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z | x <= y && z <= y = y
           | y <= x && z <= x = z
           | otherwise        = x
```

Se evalúa de arriba para abajo, se evalúa la primera expresión cuya guarde es verdadera

otherwise.= True

Definiciones Locales

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

```
raiz1 :: Float -> Float -> Float -> Float
raiz1 a b c = (-b - sqrt disc)/2*a
            where disc = b^2 - (4*a*c)
```

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

Recordar:

$$r1 = \frac{-b - \sqrt{b^2 - (4 * a * c)}}{2 * a}$$

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, pow 3, 3*3, 10-1`

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

Ejemplos:

`9, pow 3, 3*3, 10-1` expresión canónica: `9`

`[1]++[], 1, 1:[]` expresión canónica: `[1]`

Expresiones Canónicas

Definamos las expresiones canónicas para cada tipo:

- Booleanas: **True, False**
- Números:`0,1,2,3,−1,3.1415` es decir su representación decimal.
- Pares: `(E0,E1)` en donde `E0` y `E1` son expresiones canónicas.
- Listas: `[E0,…,En]` donde son `Ei` 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:

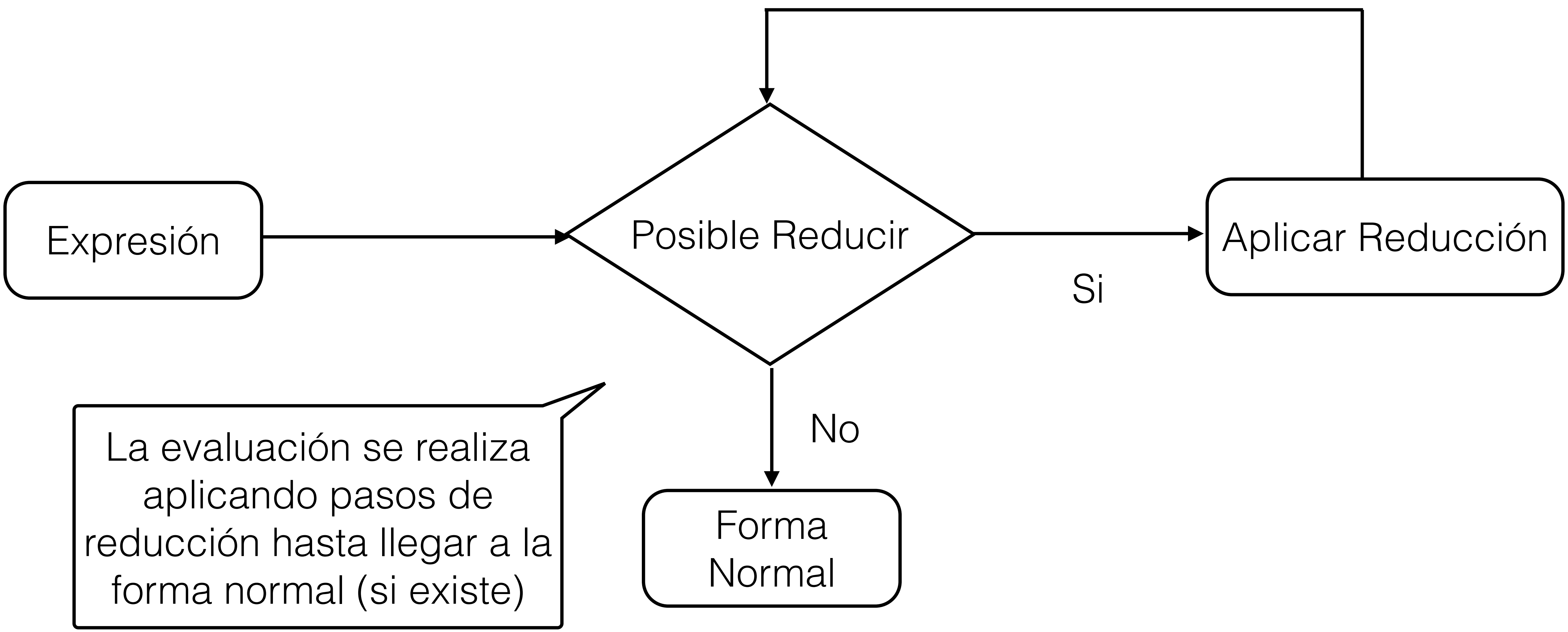
```
inf :: Int
inf = inf + 1


err = 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 :: Int -> Int
cuad x = x*x
```

Primero evaluamos el parámetro y después la función

```
cuad (3*5)
= [Arit.]
cuad 15
= [def cuad]
15 * 15
= [Arit.]
225
```

Formas de Evaluación

Podríamos evaluar la expresión de otra forma:

```
cuad (3*5)
= [def. cuad]
(3*5) * (3*5)
= [Arit.]
15 * (3*5)
= [Arit.]
15 * 15
= [Arit.]
225
```

Primero la función y después los parámetros

Los parámetros son evaluados cuando se necesitan para el operador de más afuera

Evaluaciones Aplicativa y Normal

Es decir, primero se reducen los parámetros de izquierda a derecha

Orden Aplicativo: se reduce siempre la expresión más adentro (de izquierda a derecha)

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

Los parámetros son reducidos cuando se necesitan para evaluar la expresión de más afuera

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

Aplicativa vs Normal

Veamos la siguiente función:

```
K :: a -> b -> c
K x y = x
```

Evaluación Aplicativa:

```
K 3 inf
= [eval. aplicativa y def. de inf]
K 3 (inf+1)
= [eval. aplicativa y def. de inf]
K 3 ((inf+1)+1)
=
No termina
```

No termina

Evaluación Normal:

```
K 3 inf
= [eval. normal]
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)
= [eval. lazy]
x * x
where x = cuad 3
= [def. cuad]
x * x
where x = 3*3
= [Arit.]
x * x
where x = 9
= [sustitución]
9 * 9
= [Arit.]
81
```

La expresiones son evaluadas una sola vez

A lo sumo usa tantos pasos como la evaluación aplicativa

Siempre encuentra la formal normal, cuando existe

Utiliza más memoria que la forma aplicativa