Un Pequeño Lenguaje Funcional: Tipos Básicos, Expresiones y Funciones.

Pablo F. Castro

Programación Avanzada, Universidad Nacional de Río Cuarto, Departamento de Computación

2014

Introducción

Definiremos un lenguaje simple que nos permitirá razonar sobre programas funcionales de una forma rigurosa. Sus principales componentes son:

- Definiciones de funciones recursivas.
- Expresiones.
- Tipos básicos de datos.

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

Funciones

Una función tiene un nombre y un tipo, para indicar el nombre y el tipo de una función utilizaremos la siguiente notación:

$$f : T_1 \to \cdots \to T_n \to T$$

Donde $T_1, \ldots, T_n$ son los tipos de los parámetros de f y T es el tipo del resultado. Un ejemplo:

$$+ : Nat \to Nat \to Nat$$

Es decir, la suma toma dos naturales y devuelve un natural. Usualmente, decimos que $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

Notación

Recordar: en vez de usar la notación $f(x)$, para denotar la aplicación de la función f al valor x, utilizamos la notación $f.x$. Cuando f tiene muchos parámetros, utilizamos varias veces el punto. Es decir, en vez de $f(x, y)$ usamos $f.x.y$.

Funciones y la Regla de Leibniz

Las funciones cumplen la regla de Leibniz por definición, es decir:

Regla de Leibniz para Funciones

Para cualquier función f se cumple $\langle \forall x, y : x = y \Rightarrow f.x = f.y \rangle$

La aplicación de funciones tiene la mayor precedencia con respecto a otros operadores. Es decir, para cualquier operador $\oplus$, tenemos:

$$f.x \oplus y = (f.x) \oplus y$$

Las funciones pueden tomar otras funciones como parámetros, estas son llamadas **funciones de alto orden**. Por ejemplo, la composición de funciones toma dos funciones y devuelve su composición.:

$$\circ : (B \to C) \to (A \to B) \to (A \to C)$$
$$\circ .g.f.x = g.(f.x)$$

Usualmente, la composición de funciones se escribe: $g \circ f$.

Expresiones

Durante la construcción de programas, utilizamos diferentes tipos de expresiones para denotar valores de diferentes tipos. Algunos ejemplos son:

- Booleanas: *true*, *false*, *true* $\wedge$ *false*, ...
- Numéricas: 42, $6 * 7$, 3,14159, ...
- Caracteres: *a*, *c*, *z*, ...

Cada una de ellas denota un valor. Algunas denotan el mismo valor, por ejemplo, $42 = 6 * 7$. Notar que 42 y $6 * 7$ son expresiones **diferentes**, aunque denoten el mismo número.

Definiciones

Recordar que utilizaremos el símbolo $\doteq$ para introducir definiciones, las cuales nos permiten utilizar nuevos valores. La definición de una función es del estilo:

$$f.x \doteq E$$

Donde: f es el nombre de la función, x es su parámetro y E es la expresión que la define.

Constantes

Cuando en la definición de una función f no hay parámetros, se dice que f es una constante. Por ejemplo: $cuadrado \doteq 2 * 2$.

Funciones Recursivas

Cuando en la definición de una función f aparece f en la expresión de la izquierda, se dice que f es **recursiva**.

Ejemplos

Por ejemplo, la siguiente es una definición recursiva:

$$\begin{aligned} f.0 &\doteq 1 \\ f.(n+1) &\doteq 2 * f(n) \end{aligned}$$

Para evaluar un resultado, por ejemplo $f.3$, hacemos lo siguiente:

$$\begin{aligned} & f.3 \\ = & \text{ [Def. de } f\text{]} \\ & 2 * f.2 \\ = & \text{ [Def. de } f\text{]} \\ & 2 * (2 * f.1) \\ = & \text{ [Def. de } f\text{]} \\ & 2 * (2 * (2 * f.0)) \\ = & \text{ [Def. de } f\text{]} \\ & 2 * 2 * 2 * 1 = 8 \end{aligned}$$

Definiciones

Dado un conjunto de definiciones, podemos utilizar estas definiciones para escribir expresiones, por ejemplo:

$$cuadrado.x \doteq x * x$$

$$pi \doteq 3.1416$$

$$area.r \doteq pi * cuadrado.r$$

Programas Funcionales

Un **programa funcional** es un conjunto de definiciones. La **ejecución** de un programa funcional se obtiene evaluando alguna expresión siguiendo las definiciones dadas.

Reglas de Cálculo para las Definiciones

Daremos algunas reglas para razonar dentro de este formalismo.

- Generalmente, consideraremos todas las reglas y axiomas de los dominios matemáticos que utilicemos, por ejemplo, las propiedades aritméticas.

Las dos reglas principales para manipular definiciones son el **plegado** (folding) y **desplegado** (unfolding) de definiciones.

Desplegado y Plegado de Definiciones

Si se tiene la definición $f.x \doteq E$, entonces:

$$f.A \doteq E[x := A]$$

La regla de **desplegado** consiste en reemplazar $f.A$ por $E[x := A]$. En cambio el **plegado**, consiste en reemplazar $E[x := A]$ por $f.A$.

Regla de Extensionalidad

Dos funciones son iguales si devuelven el mismo resultado cuando son evaluadas en los mismos argumentos. Formalmente:

Extensionalidad

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

Por ejemplo, probemos que la composición de funciones es asociativa.

$$
\begin{aligned}
& ((f \circ g) \circ h).x \\
= & \text{ [Def. de } \circ] \\
& (f \circ g).(h.x) \\
= & \text{ [Def. de } \circ] \\
& f.(g.(h.x)) \\
= & \text{ [Def. de } \circ] \\
& f.((g \circ h).x) \\
= & \text{ [Def. de } \circ] \\
& (f \circ (g \circ h)).x
\end{aligned}
$$

Definiciones Locales y Definiciones por Casos

Durante la definición de funciones podemos utilizar definiciones locales. Por ejemplo,

$$raiz1.a.b.c \qquad \doteq (-b - sqrt.disc)/(2 * a)$$
$$[\![disc \doteq b^2 - 4 * a * c]\!]$$

*raiz*1 calcula la raíz menor de un polinomio de grado 2. La definición local *disc* solo tiene sentido dentro de *raiz*1. En ciertos casos es necesario introducir definiciones por casos:

$$\begin{aligned} E \quad \doteq \quad & (B_0 \to E_0 \\ & \Box B_1 \to E_1) \end{aligned}$$

Donde B_0 y B_1 son predicados, y E_0 y E_1 son expresiones.

Demostrando Propiedades sobre Casos

Para demostrar una propiedad P sobre una definición por casos del estilo:

$$\begin{array}{lcl} E & \doteq & (B_0 \to E_0 \\ & & \Box B_1 \to E_1) \end{array}$$

Necesitamos probar lo siguiente:

- $B_0 \vee B_1$
- $B_0 \Rightarrow P.E_0$
- $B_1 \Rightarrow P.E_1$

Pattern Matching

Algunas veces el análisis por casos se puede hacer directamente discriminando por la forma de los parámetros.

Por ejemplo, podemos discriminar entre pares e impares.

$$f.(2 * n) \doteq E_0$$
$$f.(2 * n + 1) \doteq E_1$$

También entre 0 y valores mayores:

$$f.(0) \doteq E_0$$
$$f.(n + 1) \doteq E_1$$

Ejercicio: Escribir esta expresiones utilizando análisis por casos.

Tipos

A toda expresión se le puede asignar un tipo. Aquellas que no tienen tipo son consideradas incorrectas. Hay tres tipos básicos:

- *Num*: el tipo de los números.
- *Bool*: el tipo de lo booleanos.
- *Char*: el tipo de los caracteres.

A partir de estos podemos formar tipos compuestos.

- Las funciones son un tipo **compuesto**,
- Las secuencias son un tipo **compuesto**.

por ejemplo:

$$cuadrado : Num \to Num$$

algunas veces una función funciona para cualquier tipo, por ejemplo:

$$id.x \doteq x$$

En este caso decimos que $id : A \to A$, donde A es una variable sobre tipos, es decir, puede ser cualquier tipo básico o compuesto. Se dice que *id* es **polimórfico**.

Tuplas

Dados dos tipos A y B el tipo $A \times B$ contiene las tuplas $\langle a, b \rangle$ tal que $a \in B$ y $b \in B$.

Las tuplas cumplen la siguiente propiedad:

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

Ejemplos:

- $\langle 1, 2 \rangle$ tiene el tipo $Num \times Num$.
- $\langle true, 3 \rangle$ tiene el tipo $Bool \times Num$.
- $\langle c, 1 \rangle$ tiene el $Char \times Num$.

Listas

Las listas son secuencias ordenadas de elementos del mismo tipo. Para escribir la lista compuesta por la secuencia $x_1, \ldots, x_n$ usamos la siguiente notación.

$$[x_1, x_2, \ldots, x_n]$$

El tipo de una lista es escrito $[A]$, donde A es el tipo de sus elementos. Ejemplos:

- $[1, 2, 3]$, lista de tipo $[Num]$.
- $[c, b, a]$, lista de tipo $[Char]$.
- $[(c, True), (a, False)]$, lista de tipo$[Num \times Bool]$

El tipo de la lista **vacía** es un tipo polimórfico $[A]$, debido a que puede tener cualquier tipo.

Listas

Las listas se definen inductivamente a partir de los operadores:

- [] lista vacía.
- $\triangleright : A \to [A] \to [A]$, agrega un elemento a la izquierda.

Cualquier lista puede escribirse utilizando estos operadores. Por ejemplo.

- $[1, 2, 1]$ se escribe $1 \triangleright 2 \triangleright 1 \triangleright []$.
- $[a]$ se escribe $a \triangleright []$
- En general: $[x_1, \ldots, x_n]$ se escribe $x_1 \triangleright \cdots \triangleright x_n \triangleright []$

Propiedad

Dada dos listas $x \triangleright xs$ y $y \triangleright ys$ de tipo A, tenemos que:

$$x \triangleright xs = y \triangleright ys \equiv x = y \wedge xs = ys$$

Otros operadores sobre listas

Además consideramos los siguientes operadores:

- $+\!\!+ : [A] \to [A] \to [A]$, dadas dos listas, devuelve su concatenación.
- $\# : [A] \to Num$, devuelve la longitud de una lista.
- $\uparrow : [A] \to Num \to [A]$, tomar n, dada una lista y un número n, devuelve la lista de los primeros n.
- $\downarrow : [A] \to Num \to [A]$, tirar n, dada una lista y un número n, tira los primeros n elementos de la lista, y devuelve el resto.

La Importancia de las Expresiones

En funcional, la forma de computar es por medio de la evaluación de expresiones.

- Intuitivamente, "$5 + 10$" tiene que evaluar a "15", ambas son expresiones representando el valor **quince**.
- Debemos decidir a cómo evaluamos expresiones como: $[2 + 3, \mathit{cuadrado}.2]$, $[1, 2] +\!\!+ [2]$, etc.

Para resolver esto último debemos introducir las nociones de:

- Expresiones canónicas,
- Formal normal.

Forma Normal y Valores Canónicos

Muchas expresiones del mismo tipo denotan el mismo valor, por ejemplo:

$$9, \mathit{cuadrado}.3, 3 * 3, 10 - 1, \ldots$$

También:

$$[1] +\!\!+ [], [1], 1 \triangleright [], \ldots$$

En cada tipo elegimos un subconjunto de expresiones, que son las expresiones más *simples* que representan un valor.

Expresiones Canónicas

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

Por ejemplo, para $9, \mathit{cuadrado}.3, 3 * 3, 10 - 1, \ldots$ la expresión canónica es 9. Para $[1] +\!\!+ [], [1], 1 \triangleright [], \ldots$ la expresión canónica es $[1]$. Llamaremos a la expresión canónica que representa el mismo valor que una expresión su **forma normal**. Por ejemplo, 9 es la forma normal de $10 - 1$.

Expresiones Canónicas

Definamos las expresiones canónicas para las expresiones de distintos tipos:

- Booleanas: *true*, *false*.
- Números: $-1, 0, 1, 2, 3, 3{,}15, \ldots$, es decir, las representaciones decimales de los números.
- Pares: (E_0, E_1), en donde E_0 y E_1 son expresiones canónicas.
- Listas: $[E_0, E_1, E_2, E_3, \ldots, E_n]$, en donde cada E_i es una expresión canónica.

Algunas expresiones no tienen forma normal, por ejemplo:

$$inf : Num$$
$$inf \doteq inf + 1$$

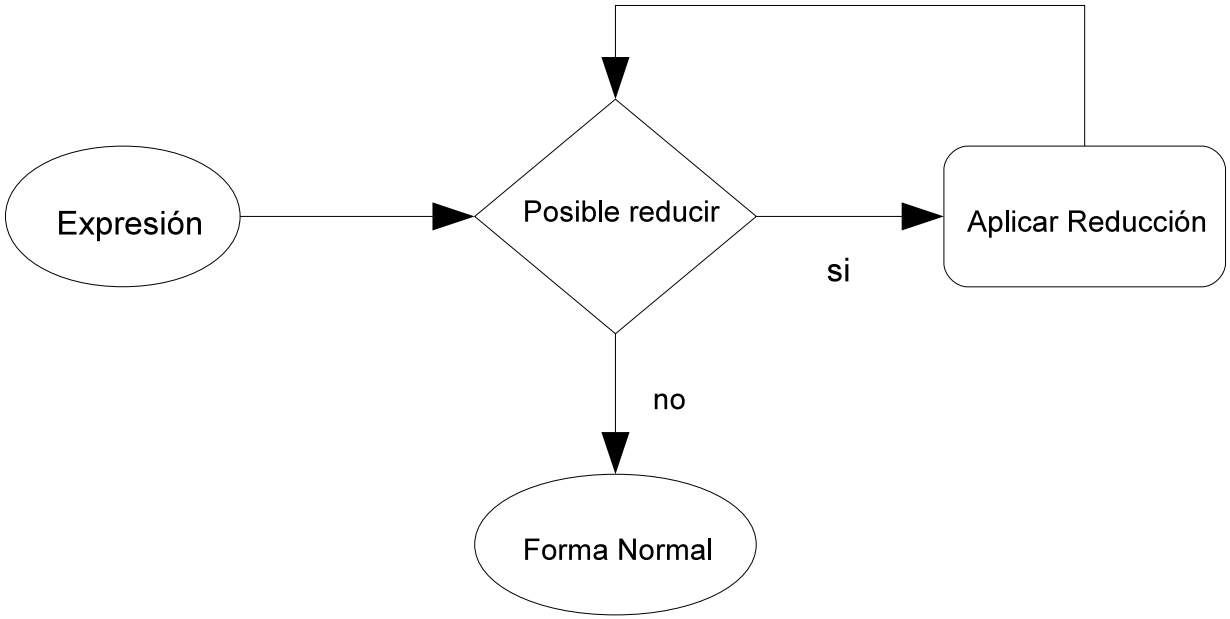
también:

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

Evaluación de Programas

La evaluación de un programa, dada un expresión, es el proceso de encontrar la forma normal de la expresión, utilizando las definiciones del programa.

Esto es realizado usando varios pasos de reducción hasta obtener la forma normal (si existe) de una expresión. La forma de evaluar la expresión es la siguiente:



Evaluación de Expresiones

Por ejemplo, apliquemos el proceso de evaluación al siguiente ejemplo:

$$
\begin{aligned}
& \mathit{cuadrado}.(3 * 5) \\
= & \text{ [Aritmética]} \\
& \mathit{cuadrado}.15 \\
= & \text{ [Def. de } \mathit{cuadrado}\text{]} \\
& 15 * 15 \\
= & \text{ [Aritmética]} \\
& 225
\end{aligned}
$$

Evaluación de Expresiones (Cont.)

También podríamos evaluar esta expresión de la siguiente forma:

$$
\begin{aligned}
& \mathit{cuadrado}.(3 * 5) \\
= & \ [\text{Def.}\mathit{cuadrado}] \\
& (3 * 5) * (3 * 5) \\
= & \ [\text{Arit.}] \\
& 15 * (3 * 5) \\
= & \ [\text{Arit}]. \\
& 15 * 15 \\
= & \ [\text{Arit}] \\
& 225
\end{aligned}
$$

Formas de Reducción

En la última reducción hemos elegido para reducir la expresión mas afuera, en cambio en la primera reducción elegimos la expresión más adentro.

Orden Aplicativo

Se reduce siempre la expresión más adentro y más a la izquierda.

Orden Normal

Se elige siempre la expresión más a la izquierda y más afuera.

Una importante propiedad es la siguiente:

Si hay una forma normal el orden normal siempre la encuentra.

Formas de Evaluación

Consideremos, por ejemplo, $K.x.y \doteq x$. Tratemos de reducir $K.3.inf$ con orden aplicativo.

$$
\begin{aligned}
& K.3.inf \\
= & \ [\text{Def.}inf] \\
& K.3.(inf + 1) \\
= & \ [\text{Def.}inf] \\
& K.3.((inf + 1) + 1) \\
= & \ [\text{Def.}inf] \\
& K.3.(((inf + 1) + 1) + 1) \\
\vdots &
\end{aligned}
$$

Formas de Evaluación

En cambio con orden normal:

$$
\begin{aligned}
& K.3.inf \\
= & \text{ [Def.}K\text{]} \\
& 3
\end{aligned}
$$

El orden normal es a veces menos eficiente pero tiene la ventaja que siempre encuentra la forma normal.

Evaluación Lazy

Los lenguajes funcionales modernos utilizan técnicas de evaluación más eficientes.

Evaluación Lazy

Se evalua la expresión más afuera más a la izquierda, en donde la misma expresión no es evaluada dos veces.

Supongamos la expresión Cuadrado.Cuadrado.3:

$$
\begin{aligned}
& \mathit{Cuadrado}.\mathit{Cuadrado}.3 \\
= & \text{ [Def. local]} \\
& x * x \\
& [\![x = \mathit{Cuadrado}.3]\!]
\end{aligned}
$$

Evaluación Lazy (Cont)

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

$x * x$

$[\![x = 3 * 3]\!]$

$= \text{[Aritmética]}$

$x * x$

$[\![x = 9]\!]$

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

$9 * 9$

$= \text{[Aritmética]}$

81

Evaluación Lazy

La evaluación lazy tiene dos características importantes:

- Encuentra la forma normal de una expresión, cuando esta existe,
- No requiere más pasos que el orden aplicativo.

La mayoría de los lenguajes funcionales modernos (por ejemplo Haskell) utilizan la evaluación lazy.