CAPÍTULO 8
El Formalismo Básico

Índice del Capítulo

En este capítulo definiremos una notación simple y abstracta para escribir y manipular programas. Esta notacií, que llamaremos *formalismo básico*, se basa en la programación funcional, siendo más simple que la mayoría de los lenguaje de programación existentes y lo suficientemente rica para expresar programas funcionales.

$$f(x) = x^3$$

Para economizar paréntesis (y por otras razones que no detallamos) usaremos un punto para denotar la aplicación de una función a un argumento; así la aplicación de la función f al argumento x se escribirá como $f.x$. Por otro lado, para evitar confundir la igualdad ($=$) con la

definición de funciones, usaremos un símbolo ligeramente diferente: $\doteq$. Con estos cambios notacionales la función que eleva al cubo se escribirá como:

$$f.x \doteq x^3$$

En lo que sigue, tendremos cuidado de no confundir a la función f con su aplicación a un argumento $f.x$, pues ésto último representa un valor, mientras que una función es un concepto totalmente diferente.

La regla básica para aplicación de funciones, es la regla de "sustitución de iguales por iguales", que ya hemos definido en el capítulo 1 como *Regla de Leibniz*. Sean $f : A \to B$ una función y x e y valores de tipo A:

$$Leibniz : \frac{x = y}{f.x = f.y}$$

Recordemos que, mediante esta regla, la igualdad entre las expresiones x e y, asegura la igualdad de la aplicación de función entre $f.x$ y $f.y$.

De acuerdo a lo visto en el Capítulo 6, en expresiones cuantificadas con el operador universal, podemos reescribir la regla de Leibniz así:

$$(\forall\, f, x, y :: x = y \Rightarrow f.x = f.y)$$

8.1. Definiciones y expresiones

Uno de los elementos que forman el formalismo básico presentado aquí, es el conjunto de *expresiones*. Al igual que en las matemáticas, las expresiones son usadas sólo para denotar valores. Estos valores pertenecerán a conjuntos que definiremos más adelante como Tipos.

Es importante distinguir entre las expresiones y el valor que éstas denotan. Por ejemplo, veamos cuál es el valor que denota la expresión $cuadrado.(3 + 4)$, donde la función $cuadrado$ se define como: $cuadrado.x \doteq x \times x$. Para ello evaluaremos la expresión aplicando definiciones de funciones.

$$\begin{array}{ll} & cuadrado.(3 + 4) \\ = & \quad \langle \text{definición de } + \rangle \\ & cuadrado.\, 7 \\ = & \quad \langle \text{definición de cuadrado} \rangle \\ & 7 \times 7 \\ = & \quad \langle \text{definición de } \times \rangle \\ & 49 \end{array}$$

Aquí vemos claramente que la expresión $cuadrado.(3 + 4)$ denota el número abstracto cuarenta y nueve, al igual que la expresión 49. La igualdad a la que hemos llegado $cuadrado.(3 + 4) = 49$, significa que ambas expresiones denotan el mismo valor, pero no hay que confundir el número denotado por la expresión decimal 49 con la expresión misma.

La expresión 49 no puede reducirse más, hemos llegado a la forma canónica de una expresión. Llamaremos *reducción* o *simplificación* al proceso de evaluación de una expresión y diremos que una expresión está en *forma canónica* o en *forma normal* si no puede reducirse más.

En nuestro formalismo básico la única forma de referirse a valores será a través de expresiones. El proceso de cómputo consistirá, en general, en reducir una expresión dada a su forma canónica cuando ésta exista.

Otro elemento que formará parte del formalismo básico será el conjunto de *definiciones*, con las cuales podremos introducir nuevos valores al formalismo y definir operaciones para manejarlos.

Una definición es una asociación de una expresión a un nombre. La forma de una definición será la siguiente:

$$f.x \doteq E$$

donde las variables en x pueden ocurrir en E. Si la secuencia de variables x es vacía, diremos que f es una *constante*. Si f ocurre en E, se dice que f es recursiva.

Dado un conjunto de definiciones, es posible utilizar éstas, para formar expresiones. Por ejemplo:

Son definiciones: $pi \doteq 3{,}1416$
$cuadrado.x \doteq x \times x$

Son expresiones: pi
$pi \times cuadrado.(3 \times 5)$

Un *programa funcional* es un conjunto de definiciones. La ejecución de un programa funcional consistirá en la evaluación de una expresión y reducción a su forma canónica.

8.1.1. Reglas para el cálculo con definiciones

En un contexto de definiciones, las reglas básicas para manejar expresiones son las reglas de *plegado* y *desplegado* (más conocidas por sus nombres en inglés: folding y unfolding). Cada una de éstas es la inversa de la otra.

Si se tiene la definición $f.x \doteq E$, entonces para una expresión A tenemos que

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

Es decir, la expresión $f.A$ corresponde a la expresión E en la que toda aparición de x fue sustituída por A. Si x es una lista de variables, A deberá serlo también y ambas deberán tener la misma longitud. En tal caso, la sustitución se realizará en forma simultánea. Por ejemplo:

$(x + y)[x, y := cuadrado.y, 3]$
$=$ $\langle$ cuadrado.y+3 $\rangle$

La regla de desplegado consiste en reemplazar $f.A$ por $E[x := A]$, mientras que la de plegado consiste en reemplazar $E[x := A]$ por $f.A$. En los cálculos que realizaremos, frecuentemente diremos "definición de f" entendiéndose si es plegado o desplegado a partir del contexto.

La regla que nos permite demostrar que dos definiciones de funciones son iguales es la siguiente:

$$(\forall\, f, g :: (\forall\, x :: f.x = g.x) \Rightarrow f = g)$$

Es decir, que las definiciones son iguales, si devuelven resultados iguales para argumentos iguales. Esta regla se conoce con el nombre *Principio de extensionalidad*. Podemos ver que lo importante acerca de una función es la correspondencia entre argumentos y resultados y no la descripción de tal correspondencia, por ejemplo, la función que duplica su argumento puede expresarse de dos formas distintas: $f.x = 2 \times x$ y $g.x = x + x$. Ambas definiciones definen la misma función pero utilizan formas diferentes para hacerlo. La extensionalidad, nos permite demostrar que $f = g$ demostrando simplemente que $f.x = g.x$ para todo x.

Por último, introduciremos mediante un ejemplo el concepto de *definiciones locales*, las cuales pueden ser inluídas en cualquier definición. Supongamos la definición de la función $f.(x, y) = (a+1)(a+2)$ donde $a = (x+y)/2$. Ésta puede ser expresada de la siguiente forma:

$$\begin{array}{lcl} f.(x,y) & \doteq & (a+1) \times (a+2) \\ & & [\![\, a \doteq (x+y)/2 \,]\!] \end{array}$$

Esta definición es más simple y clara que la que podríamos dar sin utilizar una variable local. Que la variable a esté localmente definida en la definición de f, significa que a sólo tiene sentido dentro de la definición de f. En particular, en este caso, la definición de a hace referencia a nombres de variables como x e y que sólo tienen sentido dentro de la definición de f.

Si necesitamos definir más de una variable local escribiremos una debajo de otra. Por ejemplo:

$$\begin{array}{lcl} f.(x,y) & \doteq & (a+1) \times (b+2) \\ & & [\![\, a \doteq (x+y)/2 \\ & & \;\; b \doteq (x+y)/3 \,]\!] \end{array}$$

8.2. Análisis por casos

Es frecuente la definición de funciones por *análisis por casos*. Por ejemplo :

$$f.x = \begin{cases} x+2 & \text{si } x \leq 10 \\ x-2 & \text{si } x > 10 \end{cases}$$

La definición de arriba consiste en dos expresiones, cada una de ellas contiene una expresión booleana, que llamamos *guarda*, $x \leq 10$ y $x > 10$.

Consideremos otro ejemplo, el de la función min,

$$min(x,y) = \begin{cases} x & \text{si } x \leq y \\ y & \text{si } x > y \end{cases}$$

La primer alternativa de esta definición, dice que el valor de $min(x, y)$ es x siempre que la evaluación de la expresión $x \leq y$ sea $True$. La segunda alternativa, establece que el valor de la función $min(x, y)$ será y cuando la evaluación $x > y$ dé como resultado $True$. Los dos casos, $x \leq y$ y $x > y$ abarcan todas las posibilidades, por tanto, la función min está definida para todos los posibles valores de x e y.

Adoptaremos el siguiente formato, para definir funciones mediante análisis por casos:

$$\begin{array}{lcl} f.x & \doteq & (\quad B_0 \quad \rightarrow \quad E_0 \\ & & \vdots \\ & & \Box \quad B_n \quad \rightarrow \quad E_n \\ & &) \end{array}$$

donde B_i son expresiones booleanas y E_i son expresiones del mismo tipo que $f.x$. Se entiende que el valor de $f.x$ será el valor de E_i cuando B_i sea $True$. Las expresiones B_i son las guardas.

La función min anterior la escribimos entonces como:

$$\begin{array}{lcl} min.a.b & \doteq & (\quad a \leq b \quad \rightarrow \quad a \\ & & \Box \quad a \geq b \quad \rightarrow \quad b \\ & &) \end{array}$$

De acuerdo a esta definición las guardas no son disjuntas, esto no significa que la función pueda comportarse de manera diferente en dos evaluaciones, sino que no es relevante cual de las dos guardas se elige en el caso en que ambas sean verdaderas. Esto permite cierto grado de libertad a la hora de implementar un programa.

La función factorial puede definirse usando análisis por casos así:

$$\begin{array}{lcl} fac.n & \doteq & (\quad n = 0 \quad \rightarrow \quad 1 \\ & & \Box \quad n \neq 0 \quad \rightarrow \quad n \times fac.(n-1) \\ & &) \end{array}$$

Observemos que esta es una definición recursiva.

Veremos ahora el uso de definiciones locales en una definición que utiliza análisis por casos, por ejemplo:

$$\begin{array}{lcl} f.x.y & \doteq & (\quad x > 10 \quad \rightarrow \quad x + a \\ & & \Box \quad x \leq 10 \quad \rightarrow \quad x - a \\ & & \qquad\qquad\qquad\quad [\![a \doteq (x+y)/2]\!] \\ & &) \end{array}$$

Por último, también es posible definir expresiones mediante análisis por casos, sea por ejemplo, una expresión definida de acuerdo a dos alternativas:

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

Existen reglas que permiten manipular tales expresiones. Supongamos que queremos demostrar que la expresión E definida arriba cumple una cierta propiedad P, es decir queremos probar que se satisface $P.E$. Para esto es suficiente demostrar que se cumple la conjunción de:

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

El primer punto requiere que al menos una de las guardas sea verdadera. El segundo y el tercero, piden que suponiendo la guarda verdadera, se pueda probar el caso correspondiente. Esto nos da un método de demostración y por tanto de derivación de programas que utilizaremos más adelante.

8.3. Pattern Matching

Vamos a introducir una abreviatura cómoda para escribir ciertos análisis por casos que ocurren con frecuencia. Muchas veces las guardas se usan para discernir la forma del argumento y hacer referencia a sus componentes. Por ejemplo, en el caso del factorial, las guardas se usan para determinar si el argumento es igual a 0 o no. Una forma alternativa de describir esta función es usar en los argumentos de la definición de función un patrón o *pattern*, el cual permite distinguir si el número es 0 o no. Para ello, se usa el hecho de que un número natural es o bien 0 o bien de la forma $(n+1)$, con n cualquier natural (incluído el 0). Luego la función factorial podría definirse como:

$$\begin{aligned} fac.0 &\doteq 1 \\ fac.(n+1) &\doteq (n+1) \times fac.n \end{aligned}$$

Es importante notar que el pattern sirve no sólo para distinguir los casos sino también para tener un nombre en el cuerpo de la definición (en este caso n) el cual refiera a la componente del pattern.

Otros patterns posibles para los naturales podrían ser 0, 1 y $n+2$, con los cuales una definición de función sería de la forma:

$$\begin{aligned} f.0 &\doteq E_0 \\ f.1 &\doteq E_1 \\ f.(n+2) &\doteq E_2 \end{aligned}$$

Aquí los patterns están asociados a tres guardas, dos consideran los casos en que el argumento es 0 y 1 y el último para el caso en que el argumento es 2 o más. Esta definición se traduce a análisis por casos como:

$$\begin{array}{lcll} f.n & \doteq & (& n = 0 \ \rightarrow \ E_0 \\ & & \square & n = 1 \ \rightarrow \ E_1 \\ & & \square & n \geq 2 \ \rightarrow \ E_2[n := n - 2] \\ & &) & \end{array}$$

Otra posibilidad para el caso de los naturales es separar entre pares e impares, por ejemplo la función:

$$\begin{array}{rcl} f.(2 \times n) & \doteq & E_0 \\ f.(2 \times n + 1) & \doteq & E_1 \end{array}$$

se traduce en una definición que utiliza análisis por casos, definiendo el predicado par,

$$\begin{array}{lcll} f.n & \doteq & (& par.n \ \rightarrow \ E_0[n := \frac{n}{2}] \\ & & \square & \neg par.n \ \rightarrow \ E_1[n := \frac{n-1}{2}] \\ & &) & \end{array}$$

8.4. Tipos

Toda expresión tiene un tipo asociado, lo cual significa que la expresión se evalúa como un miembro del conjunto descripto por su tipo.

Si a una expresión no se le puede asignar un tipo, la misma será considerada incorrecta.

En nuestra notación de programas utilizaremos 5 tipos básicos. Tres de ellos son tipos numéricos: $\mathbf{Nat}$, que incluye los números naturales; $\mathbf{Int}$, los enteros y $\mathbf{Real}$, los números reales. Los 2 restantes son $\mathbf{Bool}$, para los valores $True$ y $False$; y $\mathbf{Char}$, para los caracteres.

Ejemplos:	$1/2$	es de tipo $\mathbf{Real}$
	'c'	es de tipo $\mathbf{Char}$
	par	es de tipo $\mathbf{Int} \rightarrow \mathbf{Bool}$

Además de los tipos básicos existen los tipos compuestos, que se forman a partir de otros tipos. En esta sección definiremos 2 de ellos: tuplas y listas.

En ocasiones no se desea especificar un tipo en particular. En estos casos, el tipo se indicará con una letra mayúscula. Por ejemplo, la función $id.x = x$ tiene sentido como $id : \mathbf{Int} \rightarrow \mathbf{Int}$, pero también como $id : \mathbf{Char} \rightarrow \mathbf{Char}$ o $id : \mathbf{Bool} \rightarrow \mathbf{Bool}$. Si no nos interesa especificar un tipo en particular, escribiremos $id : A \rightarrow A$. La variable A se denomina *variable de tipo*. Cuando el tipo de una expresión incluya variables, diremos que es un tipo *polimórfico*.

8.4.1. Tipos básicos

Los tipos básicos mencionados anteriormente serán descriptos por las expresiones canónicas y las operaciones posibles entre elementos de ese tipo.

Tipos $\mathbf{Nat}$, $\mathbf{Int}$ y $\mathbf{Real}$: Las expresiones canónicas son las constantes (números naturales, enteros y reales). Las operaciones usadas para procesar elementos de estos tipos son: $+$,$-$,$\times$,

$/$, con las propiedades usuales. Además para los tipos $\mathbf{Nat}$ e $\mathbf{Int}$ tendremos las funciones div y mod, que devuelven, respectivamente, el cociente y el resto de la división entre dos números. Por ejemplo: $div.\,16.\,5 = 3$ y $mod.\,16.\,5 = 1$.

Tipo $\mathbf{Bool}$: Hay dos expresiones canónicas, $True$ y $False$. Una función que retorna un valor de verdad se llama predicado. Los booleanos son el resultado de los operadores de comparación $=$, $\neq$, $>$, $<$, $\geq$ y $\leq$. Estos operadores se aplican no sólo a números sino a otros tipos (por ejemplo: Char), siempre y cuando los dos argumentos a comparar tengan el mismo tipo. Es decir, cada operador de comparación es una función polimórfica de tipo $A \to A \to Bool$. Los booleanos pueden combinarse usando los operadores lógicos $\vee, \wedge, \neg,$, etc. con las propiedades usuales.

Tipo $\mathbf{Char}$: Constituído por todos los caracteres, que se denotan encerrados entre comillas simples, por ejemplo: 'a', 'B', '7'. También incluye los caracteres de control no visibles, como el espacio en blanco, el *return*, etc. Una secuencia de caracteres se denomina $\mathbf{String}$, y se denota encerrada entre comillas dobles, por ejemplo: "hola".

8.4.2. Tuplas

Una *tupla* es una colección ordenada de expresiones, no necesariamente del mismo tipo. Si b y c son expresiones, la 2-tupla (b, c) también es llamada par ordenado. También podemos formar ternas, cuaternas, y n-tuplas en general. Las tuplas tienen una utilidad concreta, supongamos que necesitamos considerar datos, correspondientes al nombre, dirección y teléfono de una persona, la estructura más adecuada para esta clase de información es una tupla. Son ejemplos de tuplas $(\text{'B'}, 3.\,8, x + 2)$, $(\text{'María'}, 45)$ y $(\text{'Peña'}, \text{'Córdoba 350'}, 4400999)$. El tipo de cada tupla se indica entre paréntesis de acuerdo al tipo de cada una de sus componentes, por ejemplo el tipo del primer ejemplo es $(\mathbf{Char}, \mathbf{Real}, \mathbf{Real})$.

Dada una n-tupla, supondremos definido el operador de indexación de tuplas

$$. : (A_0, \ldots, A_n) \to \mathbf{Nat} \to A_i$$

el cual satisface la propiedad

$$(a_0, \ldots, a_n).i = a_i$$

En el caso de los pares ordenados tendríamos la siguientes funciones:

Proyección de Pares:

- $(x, y).\,0 = x$
- $(x, y).\,1 = y$

8.4.3. Listas

Una lista es una colección de valores ordenados, todos del mismo tipo. El tipo de una lista es $[A]$, donde A es el tipo de sus elementos. La lista vacía podría considerarse de cualquier tipo,

por eso se le asigna el tipo polimórfico $[A]$, a menos que quede claro del contexto que tiene un tipo en particular.

Las listas pueden ser finitas o infinitas, pero en este curso sólo utilizaremos listas finitas. Cuando las listas son finitas se denotan entre corchetes, con sus elementos separados por comas.

A continuación presentamos ejemplos de listas:

i) $[0, 1, 2, 3]$ lista de tipo $[\mathbf{Nat}]$

ii) [('B',$True$),('C',$False$)] lista de tipo $[(\mathbf{Char}, \mathbf{Bool})]$

iii) ['a'] lista de tipo $[\mathbf{Char}]$

iv) $[[2,3, 5, 6], [1, 4,5]]$ lista de tipo $[[\mathbf{Real}]]$

Convencionalmente utilizaremos el símbolo x para indicar un elemento de una lista, xs para una lista, xss para una lista de listas. etc.

Constructores de listas

Una lista se puede generar con los siguientes constructores:

1. $[\,]$ lista vacía.

2. $\triangleright$ agrega un elemento a una lista por la izquierda.

   Si x es de tipo A y xs es de tipo $[A]$, la operación $x \triangleright xs$ agrega x a la izquierda de la lista xs.

   Ejemplo: $[0, 1, 2, 3] = 0 \triangleright [1, 2, 3] = 0 \triangleright (1 \triangleright (2 \triangleright (3 \triangleright [\,])))$

Los tipos de estos constructores son:

$$[\,] : [A]$$

$$\triangleright : A \to [A] \to [A]$$

(8.1) Proposición. Igualdad entre Listas:

$$(x \triangleright xs) = (y \triangleright ys) \equiv (x = y) \wedge (xs = ys).$$

Operaciones entre listas

Existen cinco operaciones fundamentales sobre listas, que enunciaremos a continuación:

Concatenar $+\!\!+ : [A] \to [A] \to [A]$

Dadas dos listas b y c del mismo tipo, la concatenación de las listas b y c que denotaremos $b +\!\!+ c$ da como resultado otra lista consistente en los elementos de c ubicados a continuación de los de b:

Por ejemplo, $[0,1] +\!\!+ [2,3] = [0,1,2,3]$, $[0,1] +\!\!+ [\,] +\!\!+ [2,3] = [0,1,2,3]$, y $[0,1] +\!\!+ [1,3] = [0,1,1,3]$.

Calcular longitud $\# : [A] \to \mathbf{Nat}$

Dada una lista b, el operador $\#$ aplicado a b que denotaremos $\#b$ devuelve el número de elementos que contiene b.

Por ejemplo, $\#([0,1,2,3]) = 4$, $\#[\,] = 0$.

Tomar $n \uparrow [A] \to \mathbf{Nat} \to [A]$

Dada una lista b y un número n la operación que denotaremos $b \uparrow n$ devuelve otra lista formada por los primeros n elementos de la lista b. Cuando n es mayor que la longitud de b, $b \uparrow n = b$.

Por ejemplo, $[0,1,2,3] \uparrow 3 = [0,1,2]$, $[0,1,2,3] \uparrow 5 = [0,1,2,3]$ y $[\,] \uparrow n = [\,]$.

Tirar $n \downarrow : [A] \to \mathbf{Nat} \to [A]$

Dada una lista b y un número n la operación que denotaremos $b \downarrow n$ devuelve otra lista sin los primeros n elementos de la lista b. Cuando n es mayor que la longitud de b, $b \downarrow n = [\,]$.

Por ejemplo, $[0,1,2,3] \downarrow 3 = [3]$, $[0,1,2,3] \downarrow 5 = [\,]$ y $[\,] \downarrow n = [\,]$.

Indexar $\cdot : [A] \to \mathbf{Nat} \to A$

Dada una lista b y un número n la operación que denotaremos $b.n$ devuelve el elemento de la lista b que se encuentra en la posición indicada por n. Esta operación no está definida cuando $n > \#b - 1$.

Por ejemplo, $[0,1,2,3].\,3 = 3$, $[1,2,3].\,0 = [1]$ y $[1,2].\,3$ no está definido.

Propiedades de los operadores

A continuación enunciamos algunas propiedades importantes que poseen los operadores definidos en la sección anterior:

Concatenar:

- $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$
- $xs +\!\!+ [\,] = xs = [\,] +\!\!+ xs$
- $(x \triangleright xs) +\!\!+ ys = x \triangleright (xs +\!\!+ ys)$

- $$(xs +\!\!+ ys).i \;=\; \begin{array}{lll} (& i < \#xs & \to \; xs.i \\ \square & i \geq \#xs & \to \; ys.(i - \#xs) \\ ) & & \end{array}$$
- $(xs +\!\!+ ys) = [\,] \Leftrightarrow xs = [\,] \wedge ys = [\,]$

Calcular longitud:

- $\#[\,] = 0$
- $\#(xs +\!\!+ ys) = \#xs + \#ys$
- $$\#(xs \uparrow n) \;=\; \begin{array}{lll} (& n \leq \#xs & \to \; n \\ \square & n > \#xs & \to \; \#xs \\ ) & & \end{array}$$
- $$\#(xs \downarrow n) \;=\; \begin{array}{lll} (& n \leq \#xs & \to \; \#xs - n \\ \square & n > \#xs & \to \; 0 \\ ) & & \end{array}$$

Definiciones de operadores

Tomar n:

- $[\,] \uparrow n = [\,]$
- $(x \triangleright xs) \uparrow 0 = [\,]$
- $(x \triangleright xs) \uparrow (n + 1) = x \triangleright (xs \uparrow n)$

Tirar n:

- $[\,] \downarrow n = [\,]$
- $(x \triangleright xs) \downarrow 0 = (x \triangleright xs)$
- $(x \triangleright xs) \downarrow (n + 1) = xs \downarrow n$

Indexar:

- $(x \triangleright xs).\,0 = x$
- $(x \triangleright xs).(i + 1) = xs.i$

Longitud:

- $\#[\,] = 0$
- $\#(x \triangleright xs) = 1 + \#xs$

8.5. Ejercicios

8.1 Definir la función $sgn : Int \to Int$ que dado un número devuelve $1, 0$ o -1, en caso que el número sea positivo, cero o negativo respectivamente.

8.2 Definir la función $abs : Int \to Int$ que calcula el valor absoluto de un número.

8.3 Definir el predicado $bisiesto : Nat \to Bool$ que determina si un año es bisiesto.

Recordar: los años bisiestos son aquellos que son divisibles por 4 pero no por 100 a menos que también lo sean por 400. Por ejemplo, 1900 no es bisiesto pero 2000 sí lo es.

8.4 Definir los predicados $equi$ e $isoc$ que toman tres números, que representan las longitudes de los lados de un triángulo y determinan respectivamente si el triángulo dado es equilátero o isósceles.

8.5 Definir la función $edad : (Nat, Nat, Nat) \to (Nat, Nat, Nat) \to Nat$, que dadas dos fechas indica el tiempo en años transcurrido entre ellas. Por ejemplo:

$$edad.(25, 05, 1810).(25, 4, 2010) = 199$$
$$edad.(25, 05, 1810).(26, 05, 2010) = 200$$
$$edad.(25, 05, 1810).(20, 05, 2010) = 199$$

Observar: Sólo se calculará la cantidad de años si la primera fecha es previa a la segunda fecha dada. En caso contrario la función no estará definida. Tenga en cuenta esto a la hora de definir la función.

8.6 En un prisma rectangular, llamaremos h a la altura, b al ancho y d a la profundidad. Completar la siguiente definición del área del prisma:

$$area.h.b.d \doteq 2 \times frente + 2 \times lado + 2 \times arriba$$
$$[\![\qquad\qquad]\!]$$

donde *frente, lado* y *arriba* son las caras frontal, lateral y superior del prisma.

8.7 Definir la función $raices$, que devuelve las raíces de un polinomio de segundo grado.

8.8 Suponer el siguiente juego: m jugadores en ronda comienzan a decir números naturales consecutivamente. Cuando toca un múltiplo de 7 o un número con algún dígito igual a 7, el jugador debe decir "pip" en lugar del número. Se pide, encontrar un predicado que sea $True$ cuando el jugador debe decir "pip" y $False$ en caso contrario. Resolver el problema para $0 \leq n \leq 9999$.

Sugerencia: Definir una función $unidad$, que devuelva la cifra de las unidades de un número. Ídem con las decenas, etc: para ello usar las funciones div y mod.

8.9 Empleando la técnica de *pattern matching* defina las siguientes funciones a valores naturales según el comportamiento especificado para cada una de ellas:

1. f_1: asocia el cuadrado de su argumento si este es múltiplo de tres y el cubo de su argumento en caso contrario.
2. f_2: se comporta como f_1 exceptuando que al valor cero le asocia el valor uno.
3. f_3: se comporta como f_1 exceptuando que al valor uno le asocia el valor cero.

8.10 Redefinir a las funciones f_1, f_2 y f_3 empleando la técnica de *análisis por casos*.

8.11 Deducir el tipo de las siguientes expresiones:

a) E_1 : [0,1,2,4:**Int**, 3]

b) E_2 : [(“Lucas”, True, “0341-223344”), (“Pedro”, False, “02255-456687”), (“María”, True, “0”)]

c) E_3 : $[[1 \times 4.5, 2.5], [3 : \mathbf{Real} \times 7], []]$

8.12 Calcular el tipo de las siguientes funciones:

a) $f_1.x \doteq 3 + cuad.\, x$

b) $f_2.(a, b) \doteq a + cuad.\, b$

c) $f_3.\, 0 \doteq (0, true, 2)$
$f_3.\,(n+1) \doteq (n+1, \mathbf{par}.n, n+2)$

d) $f_4.(x, y) \doteq x$

e)
$$f_5.(a,b) \doteq \begin{array}{lll} (& \mathbf{par}.a \equiv \mathbf{par}.b & \rightarrow (true, true) \\ \square & \neg\mathbf{par}.a \equiv \mathbf{par}.b & \rightarrow (false, false) \\ ) & & \end{array}$$

f)
$$f_6.c \doteq \begin{array}{lll} (& \text{‘a’} \leq c \leq \text{‘d’} & \rightarrow 1 \\ \square & \text{‘e’} \leq c \leq \text{‘k’} & \rightarrow 2 \\ \square & \text{‘l’} \leq c \leq \text{‘p’} & \rightarrow 3 \\ \square & \text{‘q’} \leq c \leq \text{‘z’} & \rightarrow 4 \\ ) & & \end{array}$$

donde la función $cuad$ tiene asociada la siguiente signatura: $cuad : Int \rightarrow Int$, y el predicado **par** en el ítem c) tiene la siguiente signatura, $\mathbf{par} : Nat \rightarrow Bool$, y en el ítem e) tiene la siguiente signatura: $\mathbf{par} : Int \rightarrow Bool$. Este predicado determina si un número es par o no.

Recordar: En el caso de los tipos numéricos: Nat, Int, o $Real$, si al calcular el tipo asociado a una expresión aritmética, cualquiera de estos fuera válido, entonces escribiremos el tipo mas restrictivo que se ajusta a la expresión, para lo cual consideraremos que Nat es mas restrictivo que Int, e Int es mas restrictivo que $Real$.

8.13 Reescribir las siguientes funciones utilizando una definición por pattern matching si la misma se encuentra definida por análisis por casos y viceversa. Además para cada función dar su tipo.

a) $$\begin{array}{lcl} f.3 & \doteq & 0 \\ f.4 & \doteq & 0 \\ f.(n+5) & \doteq & n \times 3 \end{array}$$

b) $$\begin{array}{lcl} f.n & \doteq & (\quad impar.n \quad \to \quad n \div_{Nat} 4 \\ & & \Box \quad \neg\, impar.n \quad \to \quad n \times 2 \\ & &) \end{array}$$

donde $impar : Nat \to Bool$ y determina si un número es impar o no.

c) $$\begin{array}{lcl} f.n & \doteq & (\quad mult3.n \quad \to \quad 1 \\ & & \Box \quad \neg\, mult3.n \quad \to \quad 0 \\ & &) \end{array}$$

donde $mult3 : Nat \to Bool$ determina si el argumento dado es un múltiplo de tres o no.

d) $$\begin{array}{lcl} f.0 & \doteq & 1 \\ f.(n+1) & \doteq & f.n + 2 \times n + 1 \end{array}$$

e) $$\begin{array}{lcl} g.0 & \doteq & 1 \\ g.(2 \times n + 1) & \doteq & n \times g.n \\ g.(2 \times n + 2) & \doteq & g.(n \times n) \end{array}$$

8.14 Traduzca la siguiente definición de la función g dada por *análisis por casos* a una definición equivalente por *pattern matching*.

$$\begin{array}{l} g : \mathsf{Nat} \longrightarrow \mathsf{Nat} \\ g.y \doteq \quad (\quad b_1 \quad \longrightarrow \quad y \\ \qquad \Box \quad b_2 \quad \longrightarrow \quad y^2 \\ \qquad \Box \quad \neg b_1 \wedge \neg b_2 \quad \longrightarrow \quad y^3 \\ \qquad\qquad\qquad\qquad\quad |[b_1 \doteq (y = 5) \\ \qquad\qquad\qquad\qquad\quad\ \ b_2 \doteq (y \,\mathbf{mod}\, 2 = 0)]\!] \\ \qquad) \end{array}$$

8.15 Definir la función *esVacía*: $[A] \to Bool$ que determina si la lista dada esta vacía o no.

8.16 Definir las siguientes funciones recursivamente:

1. $duplicar : [Int] \to [Int]$ que dada una lista de enteros, duplica cada uno de sus elementos. Por ejemplo: $duplicar.[1, -7, 2] = [2, -14, 4]$.

2. Generalizando el anterior defina la función $multpor : Int \to [Int] \to [Int]$ que dado un número entero n multiplica cada una de los elemntos de la lista por este. Por ejemplo: $multipor.\,7.[1, -7, 2] = [7, -49, 14]$.
3. $sumatoria : [Int] \to Int$ que dada una lista de enteros devuelve la suma de todos sus elementos.
4. $todos0y1 : [Int] \to Bool$ que dada una lista de enteros decide si todos sus elementos son ceros o unos.
5. $hay0 : [Int] \to Bool$ que dada una lista de enteros decide si alguno de sus elementos es un cero.

8.17 Definir las siguientes funciones utilizando las funciones para listas definidas en el capítulo:

a) $hd : [A] \to A$ devuelve el primer elemento de una lista. (hd es la abreviatura de *head*).

b) $tl : [A] \to [A]$ devuelve toda la lista menos el primer elemento. (tl es la abreviatura de *tail*).

c) $last : [A] \to A$ devuelve el último elemento de una lista.

d) $init : [A] \to [A]$ devuelve toda la lista menos el último elemento.

8.18 Definir las funciones dadas en el ejercicio anterior usando la técnica de pattern matching.

8.19 Reescribir las siguientes funciones utilizando una definición por pattern matching si la misma se encuentra definida por análisis por casos y viceversa. Además para cada función dar su tipo.

a)
$$\begin{array}{lcl} h.[\,] & \doteq & [\,1\,] \\ h.(x \triangleright xs) & \doteq & [\,x+1\,] +\!\!+ h.xs \end{array}$$

b)
$$\begin{array}{lcll} f.xs & \doteq & (& \\ & & \#xs \geq 3 & \to \quad xs \downarrow 2 \\ & & \square \quad \#xs < 3 \wedge \#xs \neq 0 & \to \quad [\,0,1\,] +\!\!+ xs \\ & & \square \quad \#xs = 0 & \to \quad [\,] \\ & &) & \end{array}$$

c)
$$\begin{array}{rcl} g.(0, xs) & \doteq & xs \\ g.(2 \times n + 1, xs) & \doteq & g.(2 \times n, xs) \\ g.(2 \times n + 2, xs) & \doteq & g.(2 \times n, (2 \times n + 2) \triangleright xs) \end{array}$$

d)
$$\begin{array}{rcl} p.(0, []) & \doteq & true \\ p.(0, x \triangleright xs) & \doteq & false \\ p.(n+1, x \triangleright xs) & \doteq & (n + 1 \geq x \to p.(n + 1 - x, xs)) \end{array}$$

8.20 Dar una forma general de traducción a una definición por pattern matching y análisis por casos según corresponda de las siguientes funciones. Completar el tipo de las funciones en cada caso.

a) $$\begin{array}{lcll} f.xs & \doteq & (& \#xs = 0 \to E_0 \\ & & \square & \#xs = 1 \to E_1 \\ & & \square & \#xs \geq 2 \to E_2 \\ & &) & \end{array}$$

b) $$\begin{array}{ll} h.[\,] & \doteq E_0 \\ h.[x] & \doteq E_1 \\ h.(x' \triangleright x \triangleright xs) & \doteq E_2 \end{array}$$

c) $$\begin{array}{lcll} g.xs & \doteq & (& \#xs = 0 \vee \#xs = 1 \to E_0 \\ & & \square & \#xs \geq 2 \to E_1 \\ & &) & \end{array}$$

CAPÍTULO 9
El Modelo computacional

Índice del Capítulo

9.1. Valores

Como vimos en el capítulo anterior, la noción de "expresión" es central en programación funcional. Existen muchas formas de expresiones matemáticas, no todas están permitidas en la notación que describimos, pero todas tienen ciertas características básicas en común. La característica más importante de la notación matemática es que una expresión es utilizada solamente para describir o denotar un valor. En otras palabras, el significado de una expresión es su valor.

Es importante distinguir entre un valor y su representación por medio de expresiones. Por ejemplo, la función $cuadrado : Num \to Num$ definida como

$$cuadrado.x \doteq x \cdot x$$

permite representar al número "doscientos veinticinco" como $cuadrado.(3 \times 5)$ o también $cuadrado.\,15$. Sin embargo, ninguna de estas representaciones (ni siquiera 225) es este valor. Todas ellas son representaciones concretas del valor abstracto "doscientos veinticinco".

Lo mismo sucede con otros tipos de valores.

booleanos: Las expresiones

- $false \wedge true$

- $false \vee false$
- $\neg true$
- $false$

denotan todas el valor “falso”.

pares: Las expresiones

- $(cuadrado.(3 \cdot 5), 4)$
- $((225, 2 + 2), true).\, 0$
- $(225, 4)$

denotan todas el “par cuya primera coordenada es el número doscientos veinticinco y su segunda coordenada es el número cuatro”.

listas: Las expresiones

- $[cuadrado.\, 1, 4 - 2, -44 + 47]$
- $[[\,], [1, 2, 3]].\, 1$
- $[1, 2, 3]$

denotan la “lista con los elementos uno, dos y tres en este orden”.

números: Las expresiones

- $(2, 3).\, 1$
- $\#[0, 1, 2]$
- 3

denotan el número “tres”.

funciones: Dadas las funciones

$$\begin{array}{lcl} f.x.y & \doteq & (\quad x \geq y \quad \to \quad x \\ & & \square \quad x \leq y \quad \to \quad y \\ & &) \end{array}$$

$$\begin{array}{lcl} g.x.y & \doteq & (\quad x > y \quad \to \quad x \\ & & \square \quad x < y \quad \to \quad y \\ & & \square \quad x = y \quad \to \quad x \\ & &) \end{array}$$

las expresiones