Tipos Algebraico-Libres

Jesús Ravelo
Kelwin Fernández
Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información

**** **Observación:** esta versión es un borrador y posiblemente puede contener errores. Debe ser utilizada prudentemente. ****

Supongamos que deseamos realizar un tipo concreto *dinero*, el cual podría representar bolívares, euros o dólares. Al implementar este tipo de datos, tendríamos en cualquier momento el problema de no saber qué tipo de dinero estamos manejando. Los *Tipos Algebraico-Libres*, o *Tipos Libres* a secas, nos brindan una cómoda solución para resolver este tipo de problemas, entre otros. Podemos entonces implementar tipos tales como *color* que puede ser tanto amarillo como azul, rojo o verde, o un tipo *trimestreAcadémico* que puede ser septiembre-diciembre, enero-marzo o abril-julio.

La sintaxis a utilizar sería la siguiente:

freetype *dinero* ::= *bolívares*(int) | *euros*(int) | *dólares*(int)

freetype *color* ::= *amarillo* | *azul* | *rojo* | *verde*

freetype *trimestreAcadémico* ::= *sept-dic* | *ene-mar* | *abr-jul*

donde *bolívares*, *euros* y *dólares* son los constructores del tipo *dinero*; *amarillo*, *azul*, *rojo* y *verde* los del tipo *color* y *sept-dic*, *ene-mar*, *abr-jul* los del tipo *trimestreAcadémico*.

En cualquier caso, la sintaxis general de un tipo algebraico-libre es:

freetype tipo nuevo ::= $cons_0(T_0)$ | $cons_1(T_1)$ | ... | $cons_n(T_n)$

Donde cada $cons_i$ se refiere a los constructores del tipo, y cada T_i se refiere al tipo, o los tipos de objetos que debe recibir el constructor $cons_i$. Nótese que en ciertos casos, cuando se trata de constantes como en los constructores del tipo *color*, obviamos en la sintaxis el tipo en los constructores.

1. Tipos Recursivos

En la sintaxis antes presentada no se excluye la posibilidad de que alguno de los tipos de los argumentos de los constructores fuese de hecho del mismo tipo que el tipo libre que se está definiendo. Esto genera la posibilidad de tener tipos recursivos, donde al menos uno de sus constructores reciba como uno de sus argumentos un elemento de su mismo tipo. En este eje se centrará gran parte del contenido de este material.

El primer tipo recursivo que trataremos será el clásico *expr*, el cual permite representar expresiones aritméticas.

$$\begin{array}{rl} \underline{\textbf{freetype}}\ expr ::= & \underline{ctte}(\text{int}) \mid \underline{var}(\text{string}) \mid \underline{suma}(expr,\ expr) \\ & \mid \underline{dif}(expr,\ expr) \mid \underline{mult}(expr,\ expr) \\ & \mid \underline{exp}(expr,\ expr) \mid \underline{inv}(expr) \end{array}$$

donde se definen los constructores bases *ctte* y *var* como aquellos que nos permitirán representar constantes enteras (0, 3, etc) y variables (x, y, z) dentro de la expresión. Un ejemplo de constructor recursivo es el constructor *suma* que define la expresión que representa la suma de dos expresiones aritméticas; igualmente los constructores *dif* que representa la resta, *mult* la multiplicación, *exp* la potenciación e *inv* el inverso aditivo.

Podemos representar la expresión 3, por ejemplo, usando el constructor no recursivo *ctte*, el cual recibirá como argumento la constante a representar 3, es decir,

$$\underline{ctte}(3) \quad .$$

Igualmente, podríamos representar la variable x usando el constructor no recursivo *var* de la forma:

$$\underline{var}(\text{“}x\text{”}) \quad .$$

Ya habiendo entendido el manejo de los constructores bases, podemos pasar a definir expresiones más complejas, tales como $x + 3$ usando el constructor recursivo *suma*, esto es

$$\underline{suma}(\underline{var}(\text{“}x\text{”}),\ \underline{cons}(3)) \quad .$$

Finalmente, podemos escribir expresiones tan complejas como deseemos. Por ejemplo, podemos representar la expresión $(3 \cdot (x + 4))^{-(2+x)}$ utilizando nuestro tipo como:

$$\underline{exp}(\underline{mult}(\underline{ctte}(3),\ \underline{suma}(\underline{var}(\text{“}x\text{”}),\ \underline{ctte}(4))),\ \underline{inv}(\underline{suma}(\underline{ctte}(2),\ \underline{var}(\text{“}x\text{”})))) \quad .$$

1.1. Definición de Funciones

Sobre los tipos algebraico-libres podemos definir funciones. En el caso de los tipos recursivos, la definición de las funciones asociadas suelen ser definiciones inductivas, donde los casos bases comúnmente estarán dados por los constructores no recursivos.

Sobre el tipo *expr* antes planteado, definamos las siguientes funciones:

nv Calcula el número de usos del constructor $\underline{var}$ en una expresión.

nv: $expr \longrightarrow \text{int}$

$$\begin{array}{l} nv(\underline{ctte}(r)) = 0 \\ nv(\underline{var}(s)) = 1 \\ nv(\underline{suma}(e_0, e_1)) = nv(e_0) + nv(e_1) \\ nv(\underline{dif}(e_0, e_1)) = nv(e_0) + nv(e_1) \\ nv(\underline{mult}(e_0, e_1)) = nv(e_0) + nv(e_1) \\ nv(\underline{exp}(e_0, e_1)) = nv(e_0) + nv(e_1) \\ nv(\underline{inv}(e_0)) = nv(e_0) \end{array}$$

Cuando nos refiramos a parámetros en las funciones y no consideremos restricciones (tal como sucede con e_0), supondremos que estos se encuentran universalmente cuantificados (por ello, la definición anterior se supone que cuenta con la coletilla “para todo $r \in int$, todo $s \in string$, y todo $e_0, e_1 \in expr''$).

ns Calcula el número de usos del constructor $\underline{suma}$ en una expresión.

ns: $expr \longrightarrow$ int

$$
\begin{aligned}
&ns(\underline{ctte}(r)) = 0\\
&ns(\underline{var}(s)) = 0\\
&ns(\underline{suma}(e_0, e_1)) = 1 + ns(e_0) + ns(e_1)\\
&ns(\underline{dif}(e_0, e_1)) = ns(e_0) + ns(e_1)\\
&ns(\underline{mult}(e_0, e_1)) = ns(e_0) + ns(e_1)\\
&ns(\underline{exp}(e_0, e_1)) = ns(e_0) + ns(e_1)\\
&ns(\underline{inv}(e_0)) = ns(e_0)
\end{aligned}
$$

Se deja como ejercicio definir las funciones nc, nd, nm, ne, ni que calculen el número de usos de los constructores $\underline{ctte}$, $\underline{dif}$, $\underline{mult}$, $\underline{exp}$ e $\underline{inv}$ respectivamente en una expresión.

eval Evalúa una expresión, suponiendo que todas las variables valen cero.

eval: $expr \longrightarrow$ int

$$
\begin{aligned}
&eval(\underline{ctte}(r)) = r\\
&eval(\underline{var}(s)) = 0\\
&eval(\underline{suma}(e_0, e_1)) = eval(e_0) + eval(e_1)\\
&eval(\underline{dif}(e_0, e_1)) = eval(e_0) - ns(e_1)\\
&eval(\underline{mult}(e_0, e_1)) = eval(e_0) \cdot eval(e_1)\\
&eval(\underline{exp}(e_0, e_1)) = eval(e_0)^{eval(e_1)}\\
&eval(\underline{inv}(e_0)) = eval(e_0)
\end{aligned}
$$

Por ejemplo, evaluemos estas tres funciones para la expresión $2 \cdot (x + 3)$, la cual podemos representar mediante $\underline{mult}(\underline{ctte}(2), \underline{suma}(\underline{var}(\text{“}x\text{”}), \underline{ctte}(3)))$:

Podemos ver fácilmente que el número de ocurrencias de variables en la expresión es 1. Veamos formalmente, con el uso de la definición de la función nv, que esto se cumple:

$$
\begin{array}{ll}
 & nv(\underline{mult}(\underline{ctte}(2),\ \underline{suma}(\underline{var}(\text{“}x\text{”}),\ \underline{ctte}(3)))) \\
= & \quad \langle \text{ Definición } nv, \text{ caso } \underline{mult} \ \rangle \\
 & nv(\underline{ctte}(2)) + nv(\underline{suma}(\underline{var}(\text{“}x\text{”}),\ \underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } nv, \text{ caso } \underline{ctte} \ \rangle \\
 & 0 + nv(\underline{suma}(\underline{var}(\text{“}x\text{”}),\ \underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } nv, \text{ caso } \underline{suma} \ \rangle \\
 & 0 + nv(\underline{var}(\text{“}x\text{”})) + nv(\underline{ctte}(3)) \\
= & \quad \langle \text{ Definición } nv, \text{ caso } \underline{var} \ \rangle \\
 & 0 + 1 + nv(\underline{ctte}(3)) \\
= & \quad \langle \text{ Definición } nv, \text{ caso } \underline{ctte} \ \rangle \\
 & 0 + 1 + 0 \\
= & \quad \langle \text{ Aritmética } \rangle \\
 & 1
\end{array}
$$

Veamos a continuación el número de sumas contenidas en la expresión:

$$
\begin{array}{ll}
 & ns(\underline{mult}(\underline{ctte}(2),\ \underline{suma}(\underline{var}(x),\ \underline{ctte}(3)))) \\
= & \quad \langle \text{ Definición } ns, \text{ caso } \underline{mult} \ \rangle \\
 & ns(\underline{ctte}(2)) + ns(\underline{suma}(\underline{var}(x),\ \underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } ns, \text{ caso } \underline{ctte} \ \rangle \\
 & 0 + ns(\underline{suma}(\underline{var}(x),\ \underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } ns, \text{ caso } \underline{suma} \ \rangle \\
 & 0 + 1 + ns(\underline{var}(x)) + ns(\underline{ctte}(3)) \\
= & \quad \langle \text{ Definición } ns, \text{ caso } \underline{var} \ \rangle \\
 & 0 + 1 + 0 + ns(\underline{ctte}(3)) \\
= & \quad \langle \text{ Definición } ns, \text{ caso } \underline{ctte} \ \rangle \\
 & 0 + 1 + 0 + 0 \\
= & \quad \langle \text{ Aritmética } \rangle \\
 & 1
\end{array}
$$

Finalmente, evaluemos la función:

$$
\begin{aligned}
& eval(\underline{mult}(\underline{ctte}(2),\ \underline{suma}(\underline{var}(x),\ \underline{ctte}(3)))) \\
= & \quad \langle \text{ Definición } eval, \text{ caso } \underline{mult} \ \rangle \\
& eval(\underline{ctte}(2)) \cdot eval(\underline{suma}(\underline{var}(x),\ \underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } eval, \text{ caso } \underline{ctte} \ \rangle \\
& 2 \cdot eval(\underline{suma}(\underline{var}(x),\ \underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } eval, \text{ caso } \underline{suma} \ \rangle \\
& 2 \cdot (eval(\underline{var}(x)) + eval(\underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } eval, \text{ caso } \underline{var} \ \rangle \\
& 2 \cdot (0 + eval(\underline{ctte}(3))) \\
= & \quad \langle \text{ Definición } eval, \text{ caso } \underline{ctte} \ \rangle \\
& 2 \cdot (0 + 3) \\
= & \quad \langle \text{ Aritmética } \rangle \\
& 6
\end{aligned}
$$

1.2. Demostración de Propiedades

Es usual que todos los elementos de algún determinado tipo cumplan con alguna propiedad común. Por ejemplo, en el caso de nuestro tipo *expr*, en todas las expresiones aritméticas se cumple que la cantidad de operandos básicos es exactamente uno más que la cantidad de operadores binarios. En el caso de nuestros tipos recursivos, así como las funciones sobre ellos se definen inductivamente, la demostración de propiedades sobre ellos se hace por *inducción*.

Para los casos bases de la inducción, usaremos los constructores no recursivos del tipo. Luego supondremos como hipótesis inductivas que aquello que deseamos demostrar se cumple para algunos ciertos elementos pertenecientes al tipo y tomaremos como tesis que se cumple para aquellos objetos del tipo formados por algún constructor a partir de los elementos para los cuales tomamos la hipótesis.

A continuación demostraremos la siguiente propiedad sobre el tipo *expr* antes expuesto:

$$nv(e) + nc(e) = ns(e) + nd(e) + nm(e) + ne(e) + 1$$

Note que ésta es la formalización de la propiedad antes enunciada informalmente: La cantidad de operandos básicos de e es exactamente una más que su cantidad de operandos binarios.

Caso Base:

- **Caso** e=$\underline{ctte}(x)$ **:**

Partiendo del lado izquierdo:

$$\begin{array}{ll} & nv(\underline{ctte}(x)) + nc(\underline{ctte}(x)) \\ = & \quad \langle \text{ Definiciones de } nv, nc \rangle \\ & 0 + 1 \\ = & \quad \langle \text{ Neutro aditivo } \rangle \\ & 1 \end{array}$$

Partiendo del lado derecho:

$$\begin{array}{ll} & ns(\underline{var}(x)) + nd(\underline{ctte}(x)) + nm(\underline{ctte}(x)) + ne(\underline{ctte}(x)) + 1 \\ = & \quad \langle \text{ Definiciones de } ns, nd, nm, ne \rangle \\ & 0 + 0 + 0 + 0 + 1 \\ = & \quad \langle \text{ Neutro aditivo } \rangle \\ & 1 \end{array}$$

Finalmente, dado que ambos lados son iguales a uno, hemos probado que para $e = \underline{ctte}(x)$ se tiene que:

$$ns(e) + nd(e) + nm(e) + ne(e) + 1 = nv(e) + nc(e)$$

- **Caso** e=$\underline{var}$(x) **:**

Partiendo del lado izquierdo:

$$\begin{array}{ll} & nv(\underline{var}(x)) + nc(\underline{var}(x)) \\ = & \quad \langle \text{ Definiciones de } nv, nc \rangle \\ & 1 + 0 \\ = & \quad \langle \text{ Neutro aditivo } \rangle \\ & 1 \end{array}$$

Partiendo del lado derecho:

$$
\begin{array}{ll}
 & ns(\underline{var}(x)) + nd(\underline{var}(x)) + nm(\underline{var}(x)) + ne(\underline{var}(x)) + 1 \\
= & \quad \langle \text{ Definiciones de } ns,\ nd,\ nm,\ ne \rangle \\
 & 0 + 0 + 0 + 0 + 1 \\
= & \quad \langle \text{ Neutro aditivo } \rangle \\
 & 1
\end{array}
$$

Análogamente al caso anterior, hemos probado que para $e = \underline{var}(x)$ se cumple que:

$$ns(e) + nd(e) + nm(e) + ne(e) + 1 = nv(e) + nc(e)$$

Caso Inductivo:

- **Caso** $e = \underline{suma}(e_1, e_2)$

Hipótesis Inductiva (H.I.):

$$(\forall e' \mid e' \prec_i e : nv(e') + nc(e') = ns(e') + nd(e') + nm(e') + ne(e') + 1)$$

Tesis:

$$nv(e) + nc(e) = ns(e) + nd(e) + nm(e) + ne(e) + 1$$

La relación $\prec_i$ se define como la relación de "subexpresión" inmediata. En el caso de *expr* podemos definir como las subexpresiones inmediatas de la expresión

$$\underline{suma}(\underline{dif}(\underline{var}(x), \underline{ctte}(1)), \underline{ctte}(7))$$

a las expresiones $\underline{dif}(\underline{var}(x), \underline{ctte}(1))$ y $\underline{ctte}(7)$, mientras que son subexpresiones no inmediatas las expresiones $\underline{var}(x)$ y $\underline{ctte}(1)$. Es decir, la estructura a es subestructura inmediata de b si y sólo si existe un constructor $\underline{c}$ tal que $\underline{c}(\ldots, a, \ldots) = b$.

En el caso que nos ocupa, $e = \underline{suma}(e_1, e_2)$, las subexpresiones inmediatas e' que cumplen $e' \prec_i e$ son solamente $e' = e_1$ y $e' = e_2$.

$$
\begin{array}{ll}
 & ns(e) + nd(e) + nm(e) + ne(e) + 1 \\
= & \quad \langle \text{caso } e = \underline{suma}(e_1, e_2) \rangle \\
 & ns(suma(e_1, e_2)) + nd(suma(e_1, e_2)) \\
 & + nm(\underline{suma}(e_1, e_2)) + ne(\underline{suma}(e_1, e_2)) + 1 \\
= & \quad \langle \text{ Definición de } ns \rangle \\
 & 1 + ns(e_1) + ns(e_2) + nd(\underline{suma}(e_1, e_2)) \\
 & + nm(\underline{suma}(e_1, e_2)) + ne(\underline{suma}(e_1, e_2)) + 1 \\
= & \quad \langle \text{ Definiciones de } nd, nm, ne \rangle \\
 & 1 + ns(e_1) + ns(e_2) + nd(e_1) + nd(e_2) \\
 & + nm(e_1) + nm(e_2) + ne(e_1) + ne(e_2) + 1 \\
= & \quad \left\langle \begin{array}{l} \text{Nótese que estamos persiguiendo la H.I., por} \\ \text{lo tanto, nos conviene reasociar la expresión} \\ \text{usando simetría de } +, \text{ asociatividad de } + \end{array} \right\rangle \\
 & (ns(e_1) + nd(e_1) + nm(e_1) + ne(e_1) + 1) \\
 & + (ns(e_2) + nd(e_2) + nm(e_2) + ne(e_2) + 1) \\
= & \quad \langle \text{ Dado } e_1 \prec e \wedge e_2 \prec e, \text{ H.I. } \rangle \\
 & nv(e_1) + nc(e_1) + nv(e_2) + nc(e_2) \\
= & \quad \langle \text{ Simetría, Asociatividad de } + \rangle \\
 & (nv(e_1) + nv(e_2)) + (nc(e_1) + nc(e_2)) \\
= & \quad \left\langle \begin{array}{l} \text{Recordar que estamos en el caso } e = \underline{suma}(e_1, e_2), \\ \text{definición de } nv, nc \end{array} \right\rangle \\
 & nv(\underline{suma}(e_1, e_2)) + nc(\underline{suma}(e_1, e_2)) \\
= & \quad \langle \text{ caso } e = \underline{suma}(e_1, e_2) \rangle \\
 & nv(e) + nc(e)
\end{array}
$$

Se deja al lector la demostración para los casos donde $e = \underline{dif}(e_1, e_2)$, $e = \underline{mult}(e_1, e_2)$, $e = \underline{exp}(e_1, e_2)$, $e = \underline{inv}(e_1)$. Su resolución es análoga a la anterior.

1.3. Listas

El tipo *Lista* se utiliza para representar secuencias de elementos.

freetype lista(T) ::= $\underline{lvac}$ | $\underline{cons}$(T, lista(T))

Son ejemplos de secuencias $<>$, $< 1, 0 >$, $< 2, 1, 0 >$, $<' l',' i',' b',' r',' e' >$, $< true, false, true, true, false >$.

Usaremos el constructor $\underline{lvac}$ para representar la secuencia vacía $<>$. Teniendo una *lista*, podremos construir una lista con un elemento más al

comienzo usando el constructor $\underline{cons}$. Podemos representar la secuencia unitaria con el entero 0 mediante $\underline{cons}(0, \underline{lvac})$, luego, podemos representar la secuencia $< 1, 0 >$ mediante $\underline{cons}(1, \underline{cons}(0, \underline{lvac}))$ y así sucesivamente. La cuarta secuencia antes presentada ($<' l', ' i', ' b', ' r', ' e' >$), se puede representar usando el tipo *lista* como $\underline{cons}('l', \underline{cons}('i', \underline{cons}('b', \underline{cons}('r', \underline{cons}('e', \underline{lvac})))))$.

Sobre el tipo *lista*, podemos definir funciones tales como el tamaño, reverso de la lista, concatenación de dos listas, agregar al final, agregar en una cierta posición, pertenencia de un elemento a la lista, entre otras.

Definiremos las funciones tamaño, agregar al final y reverso debido a que las necesitaremos próximamente; las otras funciones son dejadas como ejercicio.

tam: lista(T) $\longrightarrow$ int

$$\begin{cases} tam(lvac) = 0 \\ tam(cons(x, l)) = 1 + tam(l) \end{cases}$$

agregar: lista(T) $\times$ T $\longrightarrow$ lista(T)

$$\begin{cases} agregar(lvac, t) = cons(t, lvac) \\ agregar(cons(t_1, l), t_2) = cons(t_1, agregar(l, t_2)) \end{cases}$$

reverso: lista(T) $\longrightarrow$ lista(T)

$$\begin{cases} reverso(lvac) = lvac \\ reverso(cons(t, l)) = agregar(reverso(l), t) \end{cases}$$

Utilizando las funciones ya definidas, y algunas de las enunciadas como ejercicio, hay varias propiedades interesantes sobre listas que podríamos enumerar. Por ejemplo:

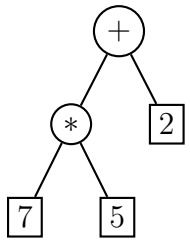
- $tam(agregar(l, a)) = tam(l) + 1$
- $tam(concat(l_0, l_1)) = tam(l_0) + tam(l_1)$
- $tam(reverso(l)) = tam(l)$
- $reverso(agregar(l, a)) = cons(a, reverso(l))$

- $reverso(concat(l_0, l_1)) = concat(reverso(l_1,), reverso(l_0))$
- $reverso(reverso(l)) = l$

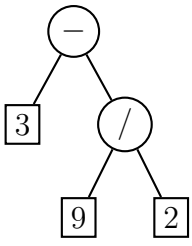
En el Apéndice A se presentan demostraciones (inductivas) de algunas de ellas, específicamente, de la cuarta y la sexta. El resto de las demostraciones se deja de ejercicio.

1.4. Árboles

Un árbol es una estructura jerárquica que permite representar información organizada de manera interesante, particularmente por supuesto cuando tal organización requiere jerarquía. Por ejemplo, volviendo a expresiones aritméticas, una expresión formada por operadores binarios y operandos reales puede organizarse como un árbol. La jerarquía implícita en la expresión $7 * 5 + 2$ puede visualizarse como el árbol



mientras para $3 - 9/2$ podemos usar el árbol



Nótese que en estos árboles tenemos un cierto tipo de información en lo que es llamado las hojas del árbol, que en este caso son números reales, mientras que en lo que es llamado los nodos internos del árbol hay otro tipo de información, en este caso operadores binarios que pueden ser representados por caracteres '+', '−', '∗', etc.

Los árboles utilizados en los ejemplos que acabamos de presentar son llamados árboles binarios, pues cada nodo interno cuanta con dos subárboles, llamados los árboles hijos de ese nodo. Un árbol también puede ser ternario si sus nodos internos tienen cada uno tres subárboles hijos; igualmente podemos tener árboles cuaternarios, etc. También podemos tener árboles en los que cada nodo puede contar con una cantidad arbitraria de subárboles hijos; a tales árboles los llamaremos generales.

Veamos ahora cómo definir tipos algebraico-libres recursivos que correspondan a distintos tipos de árboles binarios.

1.4.1. Tipo $arbbin(T_0, T_1)$

El primer tipo de árbol que consideraremos será el tipo *arbbin*, el cual, al igual que los ejemplos anteriores, contendrá información de tipos posiblemente distintos en los nodos internos y en las hojas. Definiremos formalmente a este tipo como:

freetype $arbbin(T_0,T_1) ::= \underline{hoja}(T_0) \mid \underline{rama}(T_1, arbbin(T_0,T_1), arbbin(T_0,T_1))$

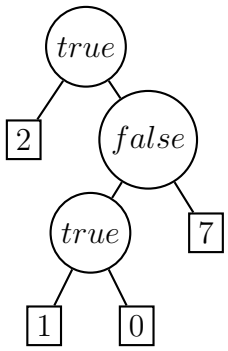
Tal como mencionamos, podemos representar los árboles presentados anteriormente usando un *arbbin*(real, char). Veamos la representación del primero:

$$\underline{rama}('+',\ \underline{rama}('*',\ \underline{hoja}(7),\ \underline{hoja}(5)),\ \underline{hoja}(2))$$

Análogamente, la representación del segundo sería:

$$\underline{rama}('-',\ \underline{hoja}(3),\ \underline{rama}('/',\ \underline{hoja}(9),\ \underline{hoja}(2)))$$

Veamos un último ejemplo de este tipo, con una instancia de un árbol con información en las hojas de tipo entero y en los nodos internos del tipo booleano. El siguiente árbol

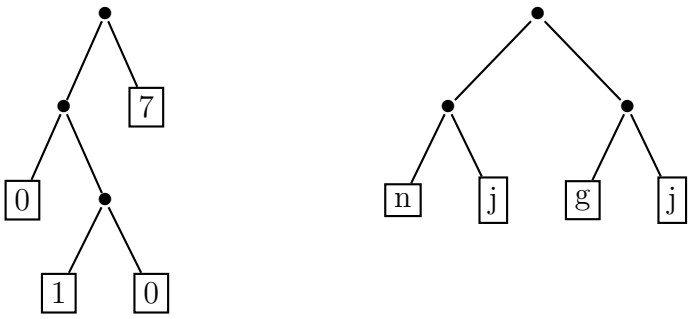


se puede representar usando un *arbbin*(int, bool) como:

$$\underline{rama}(true,\ \underline{hoja}(2),\ \underline{rama}(false,\ \underline{rama}(true,\ \underline{hoja}(1),\ \underline{hoja}(0)),\ \underline{hoja}(7)))$$

1.4.2. Tipo $arbbin'(T)$

Este tipo de árbol contendrá información únicamente en las "hojas" y no en los nodos internos. Este tipo de árboles se conoce también como árboles binarios de hojas. Podemos representar los siguientes árboles usando este tipo:



Definiremos formalmente a este tipo como:

freetype $arbbin'(T) ::= \underline{hoja}(T) \mid \underline{rama}(arbbin'(T),\ arbbin'(T))$

Usando este tipo de árbol binario, podemos representar los dos árboles presentados anteriormente como:

$$\underline{rama}(\underline{rama}(\underline{hoja}(0),\ \underline{rama}(\underline{hoja}(1),\ \underline{hoja}(0))),\ \underline{hoja}(7))$$

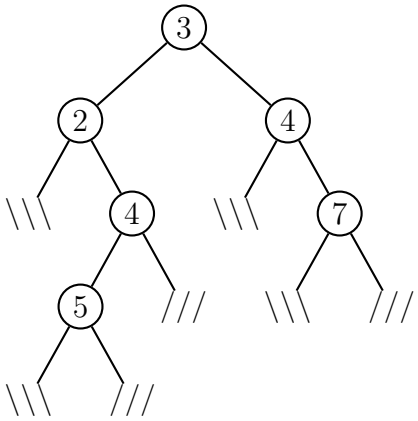
de tipo $arbbin'(int)$, y

$$\underline{rama}(\underline{rama}(\underline{hoja}('n'),\ \underline{hoja}('j')),\ \underline{rama}(\underline{hoja}('g'),\ \underline{hoja}('j'))\ )$$

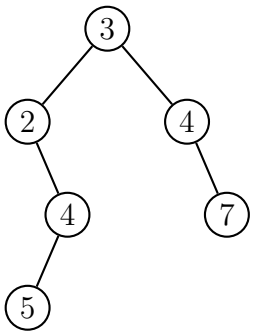
de tipo $arbbin'(char)$.

1.4.3. Tipo $arbbin''(T)$

A continuación se presenta el tercer y último tipo de árbol binario de este material, el cual es conocido como árbol binario de nodos y se caracteriza por contener información únicamente en los nodos internos, mientras que las hojas se consideran árboles vacíos. Por ejemplo, veamos el siguiente árbol binario de nodos con información de tipo entero:



Por simplicidad, en ciertas ocasiones dibujaremos árboles como el pasado sin indicar los hijos vacíos; estos se considerarán implícitos. Con lo cual, el árbol anterior se vería gráficamente:



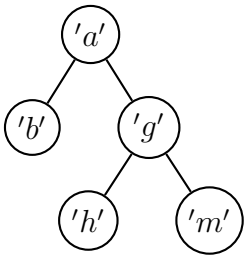
Definiremos formalmente este tipo de árboles binarios como:

freetype *arbbin"*(T) ::= $\underline{avac}$ | $\underline{nodo}(T,$ *arbbin"*$(T),$ *arbbin"*$(T))$

Con esta definición, podemos representar el árbol anterior como:

$\underline{nodo}(3,$
$\quad \underline{nodo}(2, \underline{avac}, \underline{nodo}(4, \underline{nodo}(5, \underline{avac}, \underline{avac}), \underline{avac})),$
$\quad \underline{nodo}(4, \underline{avac}, \underline{nodo}(7, \underline{avac}, \underline{avac}))$
$)$

Igualmente, podríamos representar el siguiente árbol

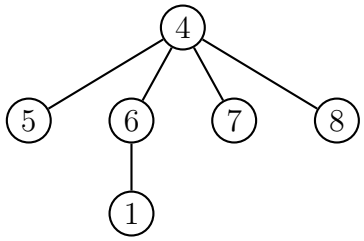


usando el tipo *arbbin"*(char) como:

$\underline{nodo}('a',\ \underline{nodo}('b',\ \underline{avac},\ \underline{avac})$
$\quad \underline{nodo}('g',\ \underline{nodo}('h',\ \underline{avac},\ \underline{avac}),$
$\quad\quad \underline{nodo}('m',\ \underline{avac},\ \underline{avac})))$

1.4.4. Tipo $arbgen(T)$

Hasta ahora hemos trabajado sólo con árboles binarios, es decir, donde cada nodo tiene dos subárboles asociados. A continuación introduciremos el tipo de árbol donde cada nodo puede tener un número arbitrario de subárboles hijos. Por ejemplo, veamos el siguiente árbol:

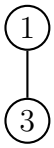


Definiremos formalmente este tipo apoyándonos en el tipo algebraico-libre *lista* definido anteriormente, de tal forma que un árbol tiene como hijos una lista (posiblemente vacía) de subárboles.

freetype $arbgen(T) ::= \underline{nodo}(T, lista(arbgen(T)))$

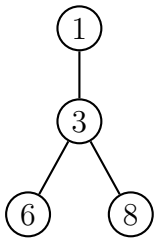
Veamos algunos ejemplos de árboles generales junto con su representación con el tipo *arbgen*:

$\underline{nodo}(1, \underline{lvac})$

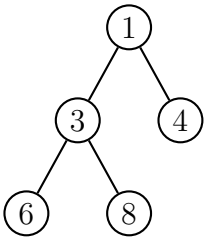


$\underline{nodo}(1,\ \underline{cons}(\underline{nodo}(3, \underline{lvac}), \underline{lvac}))$

$\underline{nodo}(1,\ \underline{cons}(\underline{nodo}(3,\ \underline{cons}(\underline{nodo}(6,\ \underline{lvac}),\ \underline{lvac})), \underline{lvac})$



$\underline{nodo}(1,$
$\quad \underline{cons}(\underline{nodo}(3,$
$\qquad \underline{cons}(\underline{nodo}(6, \underline{lvac}), \underline{cons}(\underline{nodo}(8, \underline{lvac}), lvac)),$
$\qquad \underline{lvac}))$



$\underline{nodo}(1,$
$\quad \underline{cons}(\underline{nodo}(3,$
$\qquad \underline{cons}(\underline{nodo}(6, \underline{lvac}), \underline{cons}(\underline{nodo}(8, \underline{lvac}), lvac)),$
$\qquad \underline{cons}(\underline{nodo}(4, \underline{lvac}), lvac)))$

Funciones sobre Árboles

A continuación se definirán varias funciones típicas y se analizarán varias propiedades importantes de árboles binarios de hoja (*arbbin'*) y árboles binarios de nodo (*arbbin"*).

Se define sobre el tipo de árboles binarios de hoja, las funciones que calcularán su tamaño y altura como:

tam: *arbbin'*(T) $\longrightarrow$ int

$$\begin{array}{l} tam(\underline{hoja}(x)) = 1 \\ tam(\underline{rama}(a_0, a_1)) = tam(a_0) + tam(a_1) \end{array}$$

alt: *arbbin'*(T) $\longrightarrow$ int

$$\begin{array}{l} alt(\underline{hoja}(x)) = 0 \\ alt(\underline{rama}(a_0, a_1)) = 1 + max(alt(a_0), alt(a_1)) \end{array}$$

Una propiedad importante referente a estos árboles que define cotas para su tamaño con respecto a su altura es:

$$(\forall a \mid a \in arbbin'(T) : alt(a) + 1 \leqslant tam(a) \leqslant 2^{alt(a)})$$

Informalmente podemos ver que la cota inferior se corresponde al caso en el cual el árbol se encuentra completamente desbalanceado y cada nodo interno tiene como uno de sus hijos una hoja a excepción del último que cuenta con dos árboles tipo hoja como hijos. La cota superior se da cuando el árbol se encuentra completamente lleno. La demostración inductiva de esta y otras propiedades se encuentran en el Apéndice A.

Se define sobre el tipo de árboles binarios de nodos, las funciones que calcularán su tamaño y cantidad de niveles como:

tam: *arbbin"*(T) $\longrightarrow$ int

$$\begin{array}{l} tam(\underline{avac}) = 0 \\ tam(\underline{nodo}(x, a_0, a_1)) = 1 + tam(a_0) + tam(a_1) \end{array}$$

nivs: *arbbin"*(T) $\longrightarrow$ int

$nivs(\underline{avac}) = 0$
$nivs(\underline{nodo}(x, a_0, a_1)) = 1 + max(nivs(a_0), nivs(a_1))$

Análogamente al caso de los árboles binarios de hoja, se define para los árboles binarios de nodo la siguiente propiedad

$$(\forall a \mid a \in arbbin''(T) : nivs(a) \leqslant tam(a) \leqslant 2^{nivs(a)} - 1)$$

2. Programación Imperativa con Tipos Algebraico-Libres

Sea un tipo algebraico-libre T cualquiera tal que:

freetype $T ::= \ldots \mid \underline{c}(T_0, T_1, \ldots, T_k) \mid \ldots$

Declararemos cualquier variable del tipo T tal como lo hemos venido haciendo con los tipos convencionales (int, boolean, char):

var x: T

y las manejaremos mediante el uso de:

1. **Constructores de T**

   Mediante la sintaxis:

   $$x := \underline{c}(x_0, x_1, \ldots, x_k)$$

   donde $x_0 \in T_0,\ x_1 \in T_1, \ldots,\ x_k \in T_k$. Por ejemplo:

   var *a*: *dinero*;
   $a :=$ *euros*(100);

   var *b, c*: *lista(char)*;
   $c :=$ *lvac*;
   $b :=$ *cons*('*n*', *c*);

2. **Predicado *is***

   Dada una variable x de algún tipo algebraico-libre T, el predicado "**is**" permite determinar con qué constructor de T se obtuvo el actual valor de x. El predicado "**is**" se define de la siguiente forma:

   $$(x \textbf{ is } c) \equiv (\exists a_0 \in T_0,\ a_1 \in T_1, \ldots,\ a_k \in T_k \ \mid :\ x = \underline{c}(a_0, a_1, \ldots, a_k))$$

   Por ejemplo, veamos el siguiente programa que declara e inicializa una lista y luego verifica si es vacía:

   ```
   |[
   | var  a: lista(int);
   | var  esVacia: bool;
   | a := cons(3, lvac);
   |
   | if a is lvac ⟶
   |  | esVacia := true
   | [] a is cons ⟶
   |  | esVacia := false
   | fi
   |
   | {esVacia ≡ (a is lvac)}
   ]|
   ```

3. **Instrucción *match***

   Dada una variable x de algún tipo algebraico-libre T, la instrucción "**match**" permite, suponiendo que el valor actual de x fue obtenido con un cierto constructor c, extraer los subvalores que componen al valor de x (esto es, los argumentos que se usaron con el constructor c). La instrucción "**match**" se puede definir mediante la siguiente tripleta de Hoare:

   $$\{x \textbf{ is } \underline{c}\ \}$$
   $$\quad x \textbf{ match } \underline{c}(x_0, x_1, \ldots, x_k)$$
   $$\{x\ = \underline{c}(x_0, x_1, \ldots, x_k)\}$$

   siendo $x_0,\ x_1,\ x_2, \ldots,\ x_k$ variables del tipo adecuado para el constructor. Dicha instrucción realiza una asignación simultánea a las variables $x_0,\ x_1, \ldots,\ x_k$. Por ejemplo:

```
|[
  var a, b, c: dinero;
  var va, vb: int;

  a, b := dólares(10), dólares(20);

  a match dólares(va);        /* asigna 10 a va */
  b match dólares(vb);        /* asigna 20 a vb */

  c := dólares(va + vb)
]|
```

la definición anterior de "**match**" no aplica en el caso de que alguna de las variables $x_0, x_1, \ldots, x_k$ sea la misma variable x. Para cubrir ese caso, es imprescindible que la definición utilice una variable de especificación X (entendiendo por supuesto que todas las variables $x, x_0, x_1, \ldots, x_k$ son variables de programa):

```
|[
  {x is c ∧ x = X}

      x match c(x_0, x_1, ..., x_k)

  {X = c(x_0, x_1, ..., x_k)}
]|
```

2.1. Ejemplos

A continuación se presenta una serie de ejemplos un poco más avanzados sobre el manejo de tipos algebraico-libres, en especial de los tipos *lista*, *arbbin'* y *arbbin"*.

2.1.1. Listas

El primer ejemplo que veremos calculará la productoria de los elementos de una lista de reales. Para esto, veamos la siguiente función que devuelve la productoria de los elementos de una *lista*(real):

productoria: *lista*(real) $\longrightarrow$ real

$$
\begin{array}{l}
productoria(\underline{lvac}) = 1 \\
productoria(\underline{cons}(x, a)) = x \cdot productoria(a)
\end{array}
$$

Este procedimiento tendrá un comportamiento iterativo:

```
proc productoria(in  a: lista(real); out  p: real)
  { Pre:   true }
  { Post:  p = productoria(a) }
|[
  var  l: lista(real);
  l := a;
  p := 1;

  { Inv:   p · productoria(l) = productoria(a) }
  { Cota:  ?                                   }
  do ¬(l is lvac) →
      var  e: real;
      l match cons(e, l);
      p := p · e;
  od
]|
```

Nótese que este invariante se basa en los elementos que quedan "por procesar" de la lista. Es decir, se expresa el resultado en función del segmento restante de lista que aún no hemos considerado.

El siguiente ejemplo que presentaremos sobre el tipo *lista*, se encargará de verificar si un elemento pertenece o no a una lista. Este procedimiento, al igual que el caso anterior, tendrá naturaleza iterativa.

Para el desarrollo de este ejemplo, definamos la función *pertenece* sobre el tipo *lista*:

pertenece: *lista*(T) $\times$ T $\longrightarrow \mathbb{B}$

$$
\begin{array}{ll}
pertenece(\underline{lvac}, x) \equiv & false \\
pertenece(\underline{cons}(x', l), x) \equiv & (x' = x) \\
 & \vee\ pertenece(l, x)
\end{array}
$$

```
proc buscar(in x: T; in a: lista(T); out esta: boolean)
  { Pre:  true }
  { Post:  esta ≡ pertenece(a, x) }
|[
  var l: lista(T);

  esta := false;
  l := a;

  { Inv:  esta ∨ pertenece(l, x) ≡ pertenece(a, x) }
  { Cota:  ? }
  do ¬(l is lvac) ∧ ¬esta ⟶
    var e: T;

    l match cons(e, l);

    if e = x ⟶
    |    esta := true
    [] e ≠ x ⟶
    |    skip
    fi
  od
]|
```

En este caso, al igual que en el ejemplo de productoria, hemos mantenido el estilo de invariante de elementos por procesar. Este estilo de invariantes es sumamente útil y sencillo cuando estamos trabajando con tipos algebraico-libres.

Por último, daremos una implementación recursiva de los procedimientos *buscar* y *productoria* sobre el tipo *lista*(T) y *lista*(*real*) respectivamente. Nótese que dada la naturaleza recursiva de estos tipos y de la definición de sus funciones, la implementación de estas tiende a ser casi directa.

```
proc buscar(in x: T; in a: lista(T); out esta: boolean)
  { Pre:  true }
  { Post:  esta ≡ pertenece(a, x) }
  { Cota:  ? }
|[
  if a is lvac ⟶
    esta := false;
  [] a is cons ⟶
    var l: lista(T);
    var e: T;

    a match cons(e, l);

    if e = x ⟶
       esta := true
    [] e ≠ x ⟶
       buscar(l, x, esta);
    fi
  fi
]|
```

```
proc productoria(in a: lista(real); out p: real)
  { Pre:  true }
  { Post:  p = productoria(a) }
  { Cota:  ? }
|[
  var p: real;
  if l is lvac ⟶
    p := 1
  [] l is cons ⟶
       var e: real;
       l match cons(e, l);
       productoria(l, p);
       p := e ∗ p
  fi
]|
```

Posibles Cotas

Todas las presentaciones que hemos desarrollado hasta el momento han dejado las cotas vacías. En todos estos ejemplos, podríamos haber colocado como cota el tamaño de la lista. Este tipo de cotas se conoce como cotas numéricas.

Sin embargo, existen otros tipos de cotas que son no numéricas y se basan en la relación de orden de subestructura que se establece en tipos algebraico-libres. Anteriormente definimos informalmente la relación $\prec_i$, que es una relación de subestructura inmediata; podemos generalizar esta relación y considerar no sólo los menores inmediatos. Veamos formalmente cómo definirla:

Sean $x, y \in T$,

$$x \prec_i y \equiv (\exists c \mid \textit{cesconstructores de } T : c(\ldots, x, \ldots) = y)$$

Finalmente, se define

$$x \prec y \equiv x \prec_i^+ \ y$$

donde $\prec_i^+$ es la clausura transitiva de la relación $\prec_i$.

Por ejemplo, se define la relación $\prec_i$ para el tipo *arbbin'*(T) como:

$$x \prec_i y \equiv (\exists e, a \mid e \in T, a \in \textit{arbbin}'(T) : \textit{nodo}(e, x, a) = y \vee \textit{nodo}(e, a, x) = y)$$

Teniendo esta relación podemos considerar como cota a la estructura misma, pero debemos asegurar además que existe una cota inferior que asegure terminación. Para esto veamos la siguiente definición:

Un conjunto ordenado (A, R) donde $R \subseteq A \times A$ es *Bien Fundamentado* o *Noetheriano* si y sólo si

$$\neg(\exists C \mid C \subseteq A \ \wedge \ C = \{i | i \in \mathbb{N} : x_i\} : (\forall j \mid j \in \mathbb{N} : x_{j+1} \ R \ x_j))$$

Esta definición exige que no existan cadenas infinitas R-descendientes. Note que la fórmula niega la posibilidad de $\ldots x_3 R x_2 R x_1 R x_0$. De esta forma, en algún momento nuestra relación de subsestructura debe alcanzar un mínimo, lo cual ocurre con los constructores no recursivos.

2.1.2. Árboles

Ya hemos presentado ejemplos de programación con tipos algebraico-libres sobre el tipo *lista*; a continuación presentaremos una serie de ejemplos de programación utilizando árboles.

Definamos sobre los árboles binarios de nodos con información del tipo real la siguiente función que compute la productoria de sus nodos:

$productoria$: $arbbin"$(real) $\longrightarrow$ real

$$productoria(\underline{avac}) = 1$$
$$productoria(\underline{nodo}(x, a, b)) = x \cdot productoria(a) \cdot productoria(b)$$

Veamos primero una implementación recursiva de la función *productoria*:

```
proc productoria(in  a: arbBin"(real); out  res: real)
  { Pre:  true }
  { Post:  res = productoria(a) }
  { Cota:  a }
|[
  if a is avac ⟶
  |   res := x
  [] a is nodo ⟶
  |     var  resIzq, resDer, e: real;
  |          a0, a1: arbbin"(real);
  |
  |     a match nodo(e, a0,a1);
  |     productoria(a0, resIzq);
  |     productoria(a1, resDer);
  |
  |     res := e · resIzq · resDer
  fi
]|
```

Existen casos en donde la implementación iterativa de una función sobre tipos libres no es tan directa como en el caso recursivo; el cálculo de la productoria de un árbol binario de reales es uno de esos casos. Para su implementación, consideraremos que contamos con un tipo *pila* genérico

previamente definido, incluyendo las operaciones *empilar*, *desempilar*, *tope*, *crearVacia* y *esVacia*.

```
proc productoria(in a: arbBin"(real); out res: real)
  { Pre:  true }
  { Post: res = productoria(a) }
|[
  var p: pila(arbbin"(real));
  var res: real;

  res := 1;
  crearVacia(p);
  empilar(p, a);

  do ¬(esVacia(p)) ⟶
    var b: arbbin"(real);
    tope(p, b);
    desempilar(p);

    if b is avac ⟶
        skip
    [] b is nodo ⟶
        var i, d: arbbin"(real);
        var e: real;
        b match nodo(e, i, d);

        res := res · e;
        empilar(p, d);
        empilar(p, i)
    fi
  od
]|
```

Analicemos brevemente el uso de la pila en el procedimiento pasado. Ésta nos sirve como contenedor para el resto de los subárboles que aún no hemos procesado. En principio, el árbol que no hemos procesado es el árbol de entrada completo, luego incluimos su elemento raíz en la productoria y tenemos dos subárboles nuevos por procesar; así iremos sucesivamente hasta agotar

la pila.

No se indica el invariante por simplicidad. Sin embargo, nótese que en cada paso de la iteración se cumple que *res* multiplicado por la productoria de cada árbol contenido en la pila debe ser igual a la productoria del árbol inicial completo.

En el caso de la cota, podemos ver que en cada iteración se procesa o bien un nodo del árbol o bien una hoja. Por lo tanto, podemos establecer como cota la suma del tamaño de los árboles contenidos en la pila.

Dado que la multiplicación de reales es conmutativa y asociativa, el orden en el que realicemos las operaciones es indiferente, por lo tanto, podríamos haber considerado utilizar como estructura de almacenamiento de los árboles sin procesar una cola y el algoritmo sería igualmente correcto.

Para el siguiente ejemplo, consideraremos sobre el tipo árbol binario de hojas la siguiente función, la cual define la secuencia de elementos que forman la frontera del árbol:

$$\textit{frontera}\colon \textit{arbbin'}(\mathrm{T}) \longrightarrow \mathrm{seq}(\mathrm{T})$$

$$\begin{array}{l} \textit{frontera}(\underline{\textit{hoja(e)}}) = < e > \\ \textit{frontera}(\underline{\textit{rama}}(a, b)) = \textit{frontera}(a) \ +\!\!+ \ \textit{frontera}(b) \end{array}$$

Donde el tipo secuencia representa una serie ordenada de elementos de un cierto tipo. Utilizamos la notación $< a_1, a_2, \ldots, a_n >$ para representar una secuencia cuyos elementos son $a_1, a_2, \ldots, a_n$ en dicho orden. Análogamente $< e >$ representa la secuencia unitaria cuyo único elemento es e.

Definimos el operador binario $+\!\!+$ sobre secuencias como la concatenación de dos secuencias.

Veamos una implementación recursiva de una impresión de los elementos de la función *frontera*

```
proc frontera(in a: arbbin'(T))
  { Pre:  true }
  { Post: Han sido impresos los elementos de la
          secuencia frontera(a) }
  { Cota: a }
|[
  if a is hoja ⟶
     var e: T;
     a match hoja(e);
     print(e)
  [] a is rama ⟶
     var i, d: arbbin'(T);

     a match rama(i, d);
     frontera(i);
     frontera(d)
  fi
]|
```

Al igual que en el ejemplo de productoria, es posible de forma análoga con el uso de una pila implementar el procedimiento anterior de forma iterativa. Veamos una posible implementación:

```
proc frontera(in a: arbbin'(T))
  { Pre:  true }
  { Post: Han sido impresos los elementos de la
          secuencia frontera(a) }
|[
  var p: pila(arbbin'(T));

  crearVacia(p);
  empilar(p, a);

  do ¬(esVacia(p)) ⟶
    var b: arbbin'(T);
    tope(p, b);
    desempilar(p);

    if b is hoja ⟶
      var e: T;
        b match hoja(e);
        print(e)
    [] b is rama ⟶
        var i, d: arbbin'(T);
        b match rama(i, d);

        empilar(p, d);
        empilar(p, i)
    fi
  od
]|
```

2.1.3. Abreviación *matches*

Nótese que en casi todos los casos que hemos utilizado el predicado **is**, inmediatamente después utilizamos la instrucción **match**. Dado esto, regularmente se utiliza la abreviación **matches**, que resulta de una contracción del predicado **is** con la instrucción **match**. Sea el tipo T, definido de la siguiente manera:

$$
\begin{array}{lll}
\underline{\textbf{tipo}}\ \mathrm{T} ::= & \underline{c_0}(T_{00}, T_{01}, T_{02}, \ldots, T_{0i}) \\
\mid & \underline{c_1}(T_{10}, T_{11}, T_{12}, \ldots, T_{1j}) \\
\mid & \underline{c_2}(T_{20}, T_{21}, T_{22}, \ldots, T_{2k}) \\
\vdots & \vdots \\
\mid & \underline{c_m}(T_{m0}, T_{m1}, T_{m2}, \ldots, T_{ml})
\end{array}
$$

Consideraremos la abreviación:

$$
\begin{array}{l}
\underline{\textbf{if/do}}\ \mathrm{x}\ \textbf{matches} \\
\qquad
\begin{array}{rcl}
\underline{c_0}(t_{00}, t_{01}, t_{02}, \ldots, t_{0i}) & \longrightarrow & \ldots I_0 \ldots \\
\underline{c_1}(t_{10}, t_{11}, t_{12}, \ldots, t_{1j}) & \longrightarrow & \ldots I_1 \ldots \\
\underline{c_2}(t_{20}, t_{21}, t_{22}, \ldots, t_{2k}) & \longrightarrow & \ldots I_2 \ldots \\
\vdots & \longrightarrow & \vdots \\
\underline{c_m}(t_{m0}, t_{m1}, t_{m2}, \ldots, t_{ml}) & \longrightarrow & \ldots I_m \ldots
\end{array} \\
\underline{\textbf{fi/od}}
\end{array}
$$

la cual equivale a:

$$
\begin{array}{l}
\underline{\textbf{if/do}}\ x\ \textbf{is}\ c_0 \longrightarrow \\
\quad \mid\ |[\ \textbf{var}\ \ t_{00}\colon T_{00} \\
\quad \mid\ \mid\qquad\ \ t_{01}\colon T_{01} \\
\quad \mid\ \mid\qquad\ \ \vdots\quad \vdots \\
\quad \mid\ \mid\qquad\ \ t_{0i}\colon T_{0i} \\
\quad \mid\ \mid\ \mathrm{x}\ \textbf{match}\ c_0(t_{00}, t_{01}, \ldots, t_{0i}) \\
\quad \mid\ \mid\ \ldots I_0 \ldots \\
\quad \mid\ ]| \\
[]\ x\ \textbf{is}\ c_1 \longrightarrow \\
\quad \mid\ |[\ \textbf{var}\ \ t_{10}\colon T_{10} \\
\quad \mid\ \mid\qquad\ \ t_{11}\colon T_{11} \\
\quad \mid\ \mid\qquad\ \ \vdots\quad \vdots \\
\quad \mid\ \mid\qquad\ \ t_{1j}\colon T_{1j} \\
\quad \mid\ \mid\ \mathrm{x}\ \textbf{match}\ c_1(t_{10}, t_{11}, \ldots, t_{1j}) \\
\quad \mid\ \mid\ \ldots I_1 \ldots \\
\quad \mid\ ]| \\
\quad \mid \\
\quad \vdots \\
\quad \mid
\end{array}
$$

```
[] x is c_m ⟶
   |[  var  t_m0: T_m0
            t_m1: T_m1
             ⋮    ⋮
            t_ml: T_ml
       x match c_m(t_m0, t_ml, ..., t_ml)
       ... I_m ...
   ]|
fi/od
```

Ya que contamos con esta notación podemos traducir nuestros ejemplos de imprimir frontera de un árbol utilizando esta notación. Veamos primero el caso recursivo:

```
proc frontera(in  a: arbbin'(T))
  { Pre:   true }
  { Post:  Han sido impresos los elementos de la
           secuencia frontera(a) }
  { Cota:  a }
|[
  var  i, d: arbbin'(T);
  var  e: T;
  if a matches
    hoja(e) ⟶
          print(e)
    rama(i,d) ⟶
          frontera(i);
          frontera(d)
  fi
]|
```

Nótese que se efectúa un sólo uso del "**matches**". Adicionalmente, es importante notar que e, i y d son implícitamente declaradas.

Veamos ahora el método iterativo:

```
proc frontera(in a: arbbin'(T))
  { Pre:  true }
  { Post: Han sido impresos los elementos de la
          secuencia frontera(a) }
|[
  var p: pila(arbbin'(T));

  crearVacia(p);
  empilar(p, a);

  do ¬(esVacia(p)) ⟶
    var b, i, d: arbbin'(T);
    var e: T;

    tope(p, b);
    desempilar(p);

    if b matches
      hoja(e) ⟶
          print(e)
      rama(i,d) ⟶
          empilar(p, d);
          empilar(p, id)
    fi
  od
]|
```

Nótese que en todos estos casos hemos utilizado la abreviación en las guardas de un condicional. Sin embargo, según la definición es posible usarla en la guarda de un ciclo. Veamos un ejemplo de esto. Definamos sobre el tipo *arbbin'*(T) la siguiente función que determina la hoja "más izquierda" de un árbol:

hojaIzquierda: *arbbin'*(T) $\longrightarrow$ T

$$hojaIzquierda(hoja(e)) = e$$
$$hojaIzquierda(rama(a, b)) = hojaIzquierda(a)$$

proc hojaIzquierda(**in** a: *arbbin'(T)*; **out** e: T)
$\{$ **Pre:** $true$ $\}$
$\{$ **Post:** $e = hojaIzquierda(a)$ $\}$
|[
var b, *izq*, *der*: *arbbin'(T)*;
var e: T;

$b := a$;

$\left\{ \begin{array}{l} \textbf{Inv:} \quad hojaIzquierda(b) = hojaIzquierda(a) \\ \textbf{Cota:} \quad b \end{array} \right\}$
do b **matches** $\underline{rama}(izq,\ der) \longrightarrow$
$b := izq$
od

b **matches** $\underline{hoja}(e)$
]|

2.1.4. Recorrido en Árboles

Existen tres tipos clásicos de recorridos en árboles binarios: pre, post e inorden. El recorrido en pre-orden es tal que se lista primero el elemento en el nodo raíz y después los hijos izquierdo y derecho; en in-orden se lista primero el hijo izquierdo, luego el elemento presente en el nodo raíz y luego el hijo derecho, en el recorrido en post-orden se listan primero los hijos izquierdo y derecho y luego se lista el elemento del nodo raíz. El resultado de estos recorridos se puede definir formalmente por la secuencia que van generando.

preOrden: *arbbin"*(T) $\longrightarrow$ seq(T)

$$preOrden(\underline{avac}) = <>$$
$$preOrden(\underline{nodo}(e, a, b)) = <e> \mathbin{+\!\!+} preOrden(a) \mathbin{+\!\!+} preOrden(b)$$

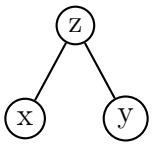
inOrden: *arbbin"*(T) $\longrightarrow$ seq(T)

$$inOrden(\underline{avac}) = <>$$
$$inOrden(\underline{nodo}(e, a, b)) = \ inOrden(a) \mathbin{+\!\!+} <e> \mathbin{+\!\!+} inOrden(b)$$

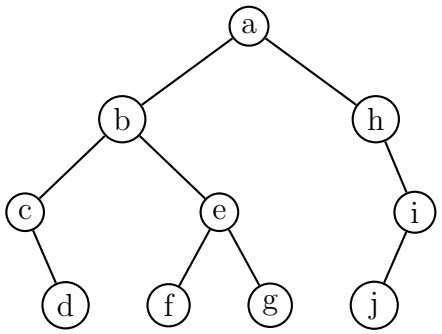
postOrden: *arbbin"*(T) $\longrightarrow$ seq(T)

$$
\begin{array}{l}
\\
postOrden(\underline{avac}) = < > \\
postOrden(\underline{nodo}(e, a, b)) = \ postOrden(a) \mathbin{+\!\!+} postOrden(b) \mathbin{+\!\!+} < e >
\end{array}
$$

Veamos los siguientes ejemplos que nos ayudarán a visualizar estos recorridos:



Pre-orden: z x y
In-orden: x z y
Post-orden: x y z



Pre-orden: a b c d e f g h i j
In-orden: c d b f e g a h j i
Post-orden: d c f g e b j i h a

Se deja como ejercicio al lector la implementación iterativa y recursiva de la impresión de los elementos de un árbol según estos recorridos.

2.1.5. Un último ejemplo sobre Árboles

Cuando definimos los distintos tipos de árboles, enunciamos unos denominados árboles generales que permiten que cada nodo tenga un número arbitrario de hijos. Veamos un ejemplo de un recorrido por niveles iterativo sobre un árbol general. Para este ejemplo, consideraremos que tenemos una *cola* FIFO con todas las operaciones necesarias. Se deja como ejercicio para

el lector definir la función *niveles* sobre el tipo que determine la secuencia resultante de recorrer un árbol general por niveles.

```
proc recorridoNiveles(in a: arbgen(T))
  { Pre: true }
  { Post: Han sido impresos los elementos de la
          secuencia niveles(a) }
|[
  var c: cola(arbgen(T));

  crearVacia(c);
  encolar(c, a);

  do ¬(esVacia(c)) ⟶
    var b: arbgen(T);
    primero(c, b);
    desencolar(c);

    var l, k: lista(arbgen(T));
    var e: T;

    b match nodo(e, l);
    print(e);

    do l ≠ lvac ⟶
      l match cons(b, l);
      encolar(c, b)
    od

  od
]|
```

3. Ejercicios

1. Definir para el tipo *expr* las funciones *nc*, *nd*, *nm*, *ne* y *ni* explicadas anteriormente en la página 4.

2. Finalizar la demostración dejada para el tipo *expr* en la página 9.

3. Para el tipo *expr* defina la función derivada, que dado un elemento del tipo *expr* y el identificador de una variable, devuelve como resultado su derivada respecto dicha variable. Puede considerar como precondición que la expresión no tendrá ningún constructor *exp*.

4. Para el tipo *lista*(T) implemente procedimientos tanto iterativos como recursivos que computen su reverso.:

5. Definir para el tipo *lista*(T) las funciones:

   a) Multiplicidad de un elemento.

   b) Eliminar la primera ocurrencia de un elemento.

   c) Eliminar todas las ocurrencias de un elemento.

6. Considere las funciones expuestas para el tipo *lista* y la función concatenar que recibe dos listas a y b y devuelve como resultado la concatenación de la lista b a la lista a. Demuestre para el tipo Lista las siguientes propiedades:

   a) $tam(l) = tam(reverso(l))$

   b) $tam(concatenar(l_0, l_1)) = tam(l_0) + tam(l_1)$

   c) $reverso(concatenar(l_1, l_2)) = concatenar(reverso(l_2), reverso(l_1))$

7. Implementar métodos recursivos e iterativos que impriman los elementos correspondientes a los recorridos en pre, in y postorden sobre *arbbin'*(T).

8. Definir una función *niveles* sobre el tipo libre *arbgen*(T) que devuelva la secuencia resultante de recorrer el árbol por niveles.

9. Dada la función espejo que dado un *arbbin''*(T), que devuelve como resultado el simétrico de un árbol y siendo $a \in arbbin''(\mathrm{T})$ demostrar:

   a) $espejo(espejo(a)) = a$

   b) $inorden(espejo(a)) = reverso(inorden(a))$

   c) $preorden(espejo(a)) = reverso(postorden(a))$

   d) $postorden(espejo(a)) = reverso(preorden(a))$

Nótese que en estos ejercicios, estamos utilizando la función *reverso* sobre secuencias y no sobre listas. Para esto, debe definir entonces la función *reverso*, tal que dada una secuencia a retorne la secuencia invertida de a.

10. Defina funciones matemáticas para el tipo $arbgen(T)$ que computen:

    a) Altura del árbol.

    b) Tamaño del árbol.

    c) Productoria de los elementos del árbol, teniendo $T = real$.

    d) Pertenencia de un elemento al árbol.

    e) Simétrico del árbol.

11. Implemente procedimientos (iterativos y recursivos) que computen las funciones del ejercicio anterior.

12. Sea el tipo *nucleotido* definido como:

    freetype *nucleotido* ::= $\underline{A} \mid \underline{C} \mid \underline{T} \mid \underline{G}$

    Y la función *complemento* definida como:

    $$\begin{array}{l} complemento\text{: } nucleotido \longrightarrow nucleotido \\ \hline \\ complemento(A) = T \\ complemento(T) = A \\ complemento(C) = G \\ complemento(G) = C \end{array}$$

    Se definen además el tipo *par* y el tipo ADN como:

    freetype *par* ::= $\underline{p}$(*nucleotido*, *nucleotido*)

    freetype ADN ::= $lista(par)$

    Se desea que defina funciones matemáticas e implemente procedimientos que computen:

a) Dado un ADN determine si su configuración es correcta, esto es, si para cada *par(a,b)* perteneciente al ADN se tiene que $complemento(a) = b$.

b) Dada una *lista(nucleotido)* devuelva una *lista(nucleotido)* la cual sea el resultado de complementar cada elemento de la primera.

Apendice A

En este apéndice se demostrarán algunas propiedades sobre tipos algebraicos-libres que se han venido planteando a lo largo de este material.

Demostración 1

Demostremos a continuación la siguiente propiedad sobre las listas:

$$(\forall l \mid l \in Lista(T) : reverso(reverso(l)) = l)$$

Caso Base:

Para $l = \underline{lvac}$:

$$\begin{array}{ll} & reverso(reverso(\underline{lvac})) \\ = & \quad \langle \text{ Definición de reverso } \rangle \\ & reverso(\underline{lvac}) \\ = & \quad \langle \text{ Definición de reverso } \rangle \\ & \underline{lvac} \end{array}$$

Caso Inductivo:

Hipótesis Inductiva (H.I.):

$$reverso(reverso(l)) = l$$

Tesis:

$$reverso(reverso(\underline{cons}(t, l))) = \underline{cons}(t, l)$$

$$\begin{array}{ll} & reverso(reverso(\underline{cons}(t, l)) \\ = & \quad \langle \text{ Definición de reverso } \rangle \\ & reverso(agregar(t, reverso(l)) \\ = & \quad \langle \text{ Lema } \star \rangle \\ & \underline{cons}(t, reverso(reverso(l))) \\ = & \quad \langle \text{ H.I. } \rangle \\ & \underline{cons}(t, l) \end{array}$$

Lema $\star$:

$(\forall l \mid l \in Lista(T) : (\forall t \mid t \in T : reverso(agregar(l,t)) = \underline{cons}(t, reverso(l))))$

Caso Base:

Para $l = \underline{lvac}$ **:**

$$
\begin{array}{ll}
 & reverso(agregar(\underline{lvac}, t)) \\
= & \quad \langle \text{ Definición de agregar } \rangle \\
 & reverso(\underline{cons}(t, \underline{lvac})) \\
= & \quad \langle \text{ Definición de reverso } \rangle \\
 & agregar(reverso(\underline{lvac}), t) \\
= & \quad \langle \text{ Definición de reverso } \rangle \\
 & agregar(\underline{lvac}, t) \\
= & \quad \langle \text{ Definición de agregar } \rangle \\
 & \underline{cons}(t, \underline{lvac}) \\
= & \quad \langle \text{ Definición de reverso } \rangle \\
 & \underline{cons}(t, reverso(\underline{lvac}))
\end{array}
$$

Caso Inductivo:

Hipótesis Inductiva (H.I.):

$$reverso(agregar(l,t)) = \underline{cons}(t, reverso(l))$$

Tesis:

$$reverso(agregar(\underline{cons}(t_1, l), t)) = \underline{cons}(t, reverso(\underline{cons}(t_1, l)))$$

$$
\begin{array}{ll}
 & reverso(agregar(\underline{cons}(t_1, l), t)) \\
= & \quad \langle \text{ Definición de agregar } \rangle \\
 & reverso(\underline{cons}(t_1, agregar(l, t))) \\
= & \quad \langle \text{ Definición de reverso } \rangle \\
 & agregar(reverso(agregar(l, t)), t_1) \\
= & \quad \langle \text{ H.I. } \rangle \\
 & agregar(\underline{cons}(t, reverso(l)), t_1) \\
= & \quad \langle \text{ Definición de agregar } \rangle \\
 & \underline{cons}(t, agregar(reverso(l), t_1)) \\
= & \quad \langle \text{ Definición de reverso } \rangle \\
 & \underline{cons}(t, reverso(\underline{cons}(t_1, l)))
\end{array}
$$

Demostración 2

A continuación demostraremos la siguiente propiedad sobre *arbbin'*(T) que permiten acotar superior e inferiormente el tamaño de un árbol.

$$(\forall a \mid a \in arbbin'(T) : alt(a) + 1 \leqslant tam(a) \leqslant 2^{alt(a)})$$

Caso Base:

Para $a = \underline{hoja}(x)$ **:**

$$\begin{array}{ll} & alt(\underline{hoja}(x)) + 1 \leqslant tam(\underline{hoja}(x)) \leqslant 2^{alt(\underline{hoja}(x))} \\ = & \quad \langle \text{ Definiciones de } alt \text{ y } tam \rangle \\ & 0 + 1 \leqslant 1 \leqslant 2^0 \\ = & \quad \langle \text{ Neutro } +, \text{ Definición de Potencia } \rangle \\ & 1 \leqslant 1 \leqslant 1 \\ = & \quad \langle \text{ Reflexividad de } \leqslant \rangle \\ & true \end{array}$$

Caso Inductivo:

Dividiremos esta fase de la demostración en dos partes: una para demostrar la desigualdad izquierda y otra para la derecha.

Para la desigualdad izquierda:

Hipótesis Inductiva (H.I.):

$$(\forall a' \in arbbin'(T) \mid a' \prec a : alt(a') + 1 \leqslant tam(a'))$$

Tesis:

$$alt(a) + 1 \leqslant tam(a)$$

Sea $H_0 \equiv a = \underline{rama}(a_0, a_1)$. Esto será de importancia para utilizar la H.I., nótese que a_0 y a_1 son subárboles de a. Para efectos de nuestra demostración, en la hipótesis inductiva, cuando nos referimos a a' estamos considerando posibles subárboles como lo son a_0 y a_1.

$$
\begin{array}{ll}
 & alt(a) + 1 \\
= & \quad \langle \text{ H}_0 \ \rangle \\
 & alt(\underline{rama}(a_0, a_1)) + 1 \\
= & \quad \langle \text{ Definición de } alt \rangle \\
 & max(alt(a_0), alt(a_1)) + 2 \\
\leqslant & \quad \langle 0 \leqslant a \ \wedge \ 0 \leqslant b, max(a,b) \leqslant a + b; \ 0 \leqslant alt(a_0), alt(a_1) \rangle \\
 & alt(a_0) + alt(a_1) + 2 \\
\leqslant & \quad \langle \text{ Dado H}_0\text{; H.I.} \rangle \\
 & (tam(a_0) - 1) + (tam(a_1) - 1) + 2 \\
= & \quad \langle \text{ Aritmética } \rangle \\
 & tam(a_0) + tam(a_1) \\
= & \quad \langle \text{ Definición de } tam \rangle \\
 & tam(\underline{rama}(a_0, a_1)) \\
= & \quad \langle \text{ H}_0 \ \rangle \\
 & tam(a)
\end{array}
$$

Nótese que se necesita como lema que $(\forall a \mid a \in arbbin'(T) : alt(a) \geq a)$, se deja como ejercicio para el lector esta demostración.

Para la desigualdad derecha:

Hipótesis Inductiva (H.I.):

$$(\forall a' \in arbbin'(T) \mid a' \prec a : tam(a') \leqslant 2^{alt(a')})$$

Tesis:

$$tam(a) \leqslant 2^{alt(a)}$$

Análogamente, estamos considerando como a' a a_0 y a_1. Sea $H_0 \equiv a = rama(a_0, a_1)$.

La utilidad de esta hipótesis será análoga al caso anterior.

$$
\begin{array}{ll}
 & tam(a) \\
= & \quad \langle \text{ H}_0 \ \rangle \\
 & tam(\underline{rama}(a_0, a_1)) \\
= & \quad \langle \text{ Definición de } tam \rangle \\
 & tam(a_0) + tam(a_1) \\
\leqslant & \quad \langle a_0, a_1 \prec a; \ H.I. \rangle \\
 & 2^{alt(a_0)} + 2^{alt(a_1)}
\end{array}
$$

Refinemos un poco el lado derecho:

$$
\begin{array}{ll}
 & 2^{alt(a)} \\
= & \quad \langle\ \mathrm{H}_0 \rangle \\
 & 2^{alt(\underline{rama}(a_0,a_1))} \\
= & \quad \langle\ \text{Simetría } +, \text{ Definición de } alt \rangle \\
 & 2^{max(alt(a_0),alt(a_1))+1} \\
= & \quad \langle 2^a \cdot 2^b = 2^{a+b}. \text{ Nos conviene separarlo en dos términos } \rangle \\
 & 2^{max(alt(a_0),alt(a_1))} \cdot 2 \\
= & \quad \langle\ \text{Aritmética } \rangle \\
 & 2^{max(alt(a_0),alt(a_1))} + 2^{max(alt(a_0),alt(a_1))}
\end{array}
$$

Ya casi hemos conectado ambos lados. Sólo debemos relacionar cada término $2^{alt(a_0)} + 2^{alt(a_1)}$ con uno de los $2^{max(alt(a_0),alt(a_1))}$. En este punto es crucial recordar que la función exponencial es creciente y, adicionalmente que la función máximo cumple $a \leqslant max(a, b)$. Por lo tanto, es directo el hecho de que:

$$2^{alt(a_0)} + 2^{alt(a_1)} \leqslant 2^{max(alt(a_0),alt(a_1))} + 2^{max(alt(a_0),alt(a_1))}$$

Finalmente, dado que el lado izquierdo es menor o igual al lado derecho, hemos demostrado que:

$$\therefore (\forall a \mid a \in arbbin'(T) : alt(a) + 1 \leqslant tam(a) \leqslant 2^{alt(a)})$$