Metodología de la Programación
Tema 5
Especificación ecuacional de Tipos Abstractos de Datos (TAD)

5.1. Introducción

5.1.1. Objetivo

El objetivo es estudiar cómo se definen nuevos Tipos Abstractos de Datos (TAD) utilizando la técnica de la especificación ecuacional (o especificación algebraica).

Para definir un nuevo tipo de datos hay que indicar lo siguiente:

- Cuáles son los valores que pertenecen al tipo.
- Qué operaciones se pueden realizar con los elementos de ese tipo.
- Qué propiedades cumplen las operaciones definidas sobre ese tipo.

Por ejemplo, en el caso de los números enteros los valores son los propios números (…, -2, -1, 0, 1, 2, 3, …), las operaciones son la suma, la resta, etc y las propiedades de esas operaciones nos dicen que la suma es conmutativa y asociativa, que la resta no es conmutativa, el producto es distributivo sobre la suma, etc.

Tanto para indicar qué valores pertenecen a un tipo como para definir operaciones sobre un tipo, la mayoría de las veces se suele utilizar la recursividad. Por otro lado, a la hora de probar propiedades la herramienta principal es la inducción.

5.1.2. Haskell

Haskell es un lenguaje de programación funcional, es decir, se basa en funciones. Por tanto, el tipo de programación llevado a cabo por medio de Haskell (y otros lenguajes similares) se conoce como *programación funcional*. En el lenguaje Haskell todas las funciones son "puras" en el sentido matemático: las funciones no producen efectos laterales (ni siquiera las funciones de Entrada/Salida). Debido a ello, todas las expresiones construidas en Haskell son "transparentes" (tienen la propiedad conocida como "transparencia referencial").

Se puede decir que en Haskell un programa es una colección de funciones. Cada función puede ser ejecutada de manera independiente o puede ser utilizada por otra función (en ambos casos sustituyendo los parámetros formales por argumentos reales).

La página oficial de Haskell es http://haskell.org desde donde se puede instalar la plataforma Haskell para distintos sistemas operativos (dirección web https://www.haskell.org/platform/). En https://www.haskell.org/downloads podemos encontrar más opciones. Además también se dispone del compilador GHC en la dirección http://www.haskell.org/ghc/.

A la hora de escribir programas en el lenguaje Haskell, se puede utilizar notepad++ como editor. Este editor se puede descargar desde http://notepad-plus-plus.org.

5.2. Tipos simples predefinidos de Haskell

5.2.1. Bool

- Valores: {True, False}
- Operadores:
  - ✓ && (and)
  - ✓ || (or)
  - ✓ not (not)
- Propiedades (donde φ, ψ y γ representan expresiones booleanas):
  - ✓ φ && ψ ≡ ψ && φ (&& es conmutativo)
  - ✓ φ || ψ ≡ ψ || φ (|| es conmutativo)
  - ✓ not(φ && ψ) ≡ (not φ) || (not ψ) (ley de De Morgan)
  - ✓ not(φ || ψ) ≡ (not φ) && (not ψ) (ley de De Morgan)
  - ✓ φ && (ψ || γ) ≡ (φ && ψ) || (φ && γ) (Distributividad)
  - ✓ φ || (ψ&& γ) ≡ (φ || ψ) && (φ || γ) (Distributividad)
  - ✓ False || φ ≡ φ (False es el elemento neutro para ||)
  - ✓ False && φ ≡ False (False es el elemento nulo para &&)
  - ✓ True || φ ≡ True (False es el elemento nulo para ||)
  - ✓ True && φ ≡ φ (True es el elemento neutro para &&)
  - ✓ …

5.2.2. Int

- Valores: números enteros con intervalo limitado (minBound, maxBound)
- Operadores aritméticos:
  - ✓ - (signo negativo)
  - ✓ + (suma)
  - ✓ – (resta)
  - ✓ * (producto)
  - ✓ ^ (potencia, 2 ^ 3 = 8)
  - ✓ div (división entera; dos maneras de escribir: 16 `div` 3 = 5 ó div 16 3 = 5, donde ` es la tilde que está junto a la tecla de la letra P)
  - ✓ mod (resto de la división entera; dos maneras de escribir: 16 `mod` 3 = 1 ó mod 16 3 = 1, donde ` es la tilde que está junto a la tecla de la letra P)
- Propiedades:
  - ✓ x + y = y + x Conmutatividad
  - ✓ x + (y + z) = (x + y) + z Asociatividad
  - ✓ x * (y + z) = (x * y) + (x * z) Distributividad
  - ✓ 0 + x = x 0 es elemento neutro para +
  - ✓ 1 * x = x 1 es elemento neutro para *
  - ✓ 0 * x = 0 0 es elemento nulo para *
  - ✓ …

5.2.3. Integer

- Valores: números enteros sin límite
- Operadores aritméticos: los mismos que Int

- Propiedades: los mismos que Int

5.2.4. Float, Double

- Valores: números reales
- Operadores aritméticos: No los vamos a utilizar
- Propiedades: Parecido al caso Int

5.2.5. Char

- Valores: caracteres ('a', …, 'z', 'A', …, 'Z', …) Utilizando la comilla simple que está al lado de la tecla del 0
- Operadores: No los vamos a utilizar
- Propiedades: no los usaremos

5.2.6. Operaciones relacionales para los tipos básicos

Los operadores relacionales que se muestran a continuación están definidos para todos los tipos básicos mencionados:

- ✓ == (igual)
- ✓ /= (distinto)
- ✓ > (mayor que)
- ✓ >= (mayor o igual que)
- ✓ < (menor)
- ✓ <= (menor o igual que)

5.2.7. Definición de nuevas operaciones para los tipos básicos

Aparte de utilizar las operaciones presentadas hasta ahora, también es posible definir nuevas operaciones. Al definir las operaciones, colocaremos un número junto a cada ecuación. De esa manera podremos hacer referencias a las distintas ecuaciones. Vamos a dar tres ejemplos sencillos:

- **par**: dado un número entero, devuelve True si es par y False si es impar.

  Tipo de la operación:
  ```
  par:: (Int) -> Bool
  ```
  Ecuación que define la operación:
  ```
  par(x) = (x `mod` 2) == 0                (1)
  ```

  La definición de la función *par* viene a decir que dado un número x, la función *par* devuelve como resultado el resultado de comparar el valor x `mod` 2 y el valor 0. El resultado de esa comparación será True o False.

  Otra definición alternativa de *par* sería la siguiente:
  ```
  par(x)
      | (x `mod` 2) == 0     = True        (1)
      | otherwise            = False       (2)
  ```

En esta otra definición de la función *par* se indica que si el valor x `mod` 2 y el valor 0 son iguales entonces la función *par* devuelve True y si no la función *par* devuelve False.

- **impar**: dado un número entero, devuelve True si es impar y False si es par.

  Tipo de la operación:
  impar:: (Int) -> Bool

  Ecuación que define la operación:
  impar(x) = not(par(x)) (1)

  En esta definición se indica que dado un número entero x, la función *impar* devuelve lo contrario de lo que devolvería la función *par*. Por tanto la idea es calcular el resultado que devuelve *par* para el valor x y luego negar ese resultado mediante el operador lógico *not*.

  Aquí vemos que al definir una nueva función se pueden utilizar funciones que se han definido previamente.

- **max3**: dados tres números enteros devuelve el mayor:

  Tipo de la operación:
  max3:: (Int, Int, Int) -> Int
  Ecuación que define la operación:

```
max3(x, y, z)
        | x >= y && x >= z      = x      (1)
        | y > x && y >= z       = y      (2)
        | otherwise             = z      (3)
```

  Mediante esas tres ecuaciones se indica que si se cumple x >= y && x >= z, entonces se devolverá x como resultado. Si no, si se cumple y > x && y >= z, entonces se devolverá el valor y como resultado y si no se devolverá el valor z como resultado.

  Tal como ocurre en este ejemplo, cuando se tiene una lista de casos o condiciones, Haskell evalúa esas condiciones empezando desde arriba y se devuelve como resultado lo indicado en el caso correspondiente a la primera condición que se cumpla.

5.3. Listas

Una lista se representa habitualmente de la siguiente forma: [2, 7, 35, -8, 0]
Esa lista sería una lista de enteros. Su tipo sería [Int]. Una lista vacía se representa de esta forma: []

El tipo de las listas es un tipo de datos paramétrico [t] ya que dependiendo del parámetro t podemos tener listas de booleanos, listas de enteros, listas de listas de enteros, etc. Por tanto se pueden definir listas de cualquier tipo que ya esté definido pero todos los elementos de la lista han de ser del mismo tipo.

Ejemplos:

[True, True, True, False, True] Su tipo sería [Bool] (lista de booleanos)
[[0, 5], [], [77, -50, 3]] Su tipo sería [[Int]] (lista de listas de enteros)

5.3.1. Operaciones constructoras para listas

Las operaciones constructoras sirven para expresar de manera formal y general qué valores pertenecen al tipo.

Toda lista puede ser construida utilizando las siguientes dos operaciones:

- [] **Generar lista vacía**
- : **Añadir un elemento por la izquierda**

La primera genera una lista vacía y la segunda sirve para añadir elementos de uno en uno por la izquierda.

Por ejemplo, la lista [6, 10] se construiría así:

6:10:[]

Ello viene a decir que esa lista se construye generando primero una lista vacía, añadiendo luego el elemento 10 a la lista vacía y añadiendo 6 a la lista que contiene el 10.

[] → [10] → [6, 10]

[2, 7, 35, -8, 0] se construiría de la siguiente forma: 2:7:35:-8:0:[]

Para Haskell ambas representaciones ([6,10] y 6:10:[]) son equivalentes, entiende las dos versiones, pero de cara a definir nuevas operaciones utilizando especificación ecuacional y de cara a probar propiedades de las nuevas operaciones la segunda versión es más adecuada.

Toda lista tiene uno de los dos siguientes formatos:

- ✓ [] (es una lista vacía)
- ✓ x:s (es una lista no vacía donde el primer elemento es x y el resto de elementos conforman una sublista s) (en vez de x y s se pueden utilizar otros nombres)

[2, 7, 35, -8, 0] es 2:7:35:-8:0:[]
x: s
s es 7:35:-8:0:[]

Cuando una lista tiene dos o más elementos, es posible decir que la lista tiene el formato

x:z:s

el primer elemento es **x**, el segundo **z** y el resto de elementos conforman una sublista **s**.

[2, 7, 35, -8, 0] es 2:7:35:-8:0:[]
x:z:s

El **tipo de las dos operaciones** constructoras [] y : es el siguiente:

```
[] :: [t]
: :: (t, [t]) -> [t]
```

En el primer caso se indica que [] genera una lista vacía del tipo t que deseemos. La variable t representa a cualquier tipo. Como la operación no tiene argumentos de entrada, no se pone ->.

En el segundo caso se indica que dando a la operación **:** un elemento de tipo t y una lista de tipo t, devuelve una lista de tipo t. La notación **::** se utiliza al dar el tipo de los elementos y de las funciones, los datos de entrada de una función irán entre paréntesis y separados por coma y el resultado irá tras -> (si hay datos). Puede que no haya datos, que haya uno o que haya varios pero siempre habrá un resultado.

5.3.2. Algunas operaciones no constructoras para listas

A continuación, se definen algunas operaciones que sirven para realizar cálculos con listas. Para definir una función u operación hay que dar el tipo de la función y las ecuaciones que indican qué hace la función.

- **primero**: Dada una lista, devuelve el primer elemento de la lista. Si la lista es vacía se genera un error (se muestra un mensaje de error).

  Ejemplos:

  ```
  primero([4, 7, 8]) = 4
  primero([False, True]) = False
  primero([[0, 5], [], [77, -50, 3]]) = [0, 5]
  ```

  Tipo de la operación:
  primero:: ([t]) -> t

  Ecuaciones que definen la operación:
  primero([]) = error "Lista vacía. No se puede obtener el primer elemento." (1)
  primero(x:s) = x (2)

  Para generar mensajes de error se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **resto**: Dada una lista, devuelve la lista que se obtiene al eliminar el primer elemento de la lista. Si la lista es vacía se genera un error (se muestra un mensaje de error).

  Ejemplos:
  resto([4, 7, 8]) = [7, 8]
  resto([False, True]) = [True]
  resto([[0, 5], [], [77, -50, 3]]) = [[], [77, -50, 3]]

  Tipo de la operación:
  resto:: ([t]) -> [t]

  Ecuaciones que definen la operación:
  resto([]) = error "Lista vacía. No se puede obtener el resto." (1)
  resto(x:s) = s (2)

  Para generar el mensaje de error correspondiente, se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **es_vacia**: Dada una lista, devuelve True si la lista es vacía y devuelve False si no es vacía.

  Ejemplos:
  es_vacia([]) = True
  es_vacia([4, 7, 8]) = False
  es_vacia([False, True]) = False

  Tipo de la operación:
  es_vacia:: ([t]) -> Bool

  Ecuaciones que definen la operación:
  es_vacia([]) = True (1)
  es_vacia(x:s) = False (2)

  Si la lista dada como dato de entrada es vacía (es decir, tiene la forma []) entonces se devuelve True. Si la lista dada como dato de entrada no es vacía (es decir, tiene la forma x:s siendo x el primer elemento y s la sublista formada por el resto de los elementos), entonces se devuelve False.

- **esta**: Dados un elemento de tipo t y una lista de tipo t, devuelve True si el elemento está en la lista y devuelve False en caso contrario.

  Ejemplos:

  esta(8, []) = False
  esta(8, [4, 8, 7, 8]) = True
  esta(False, [False, True, True]) = True
  esta([10, 9], [[0, 5], [], [77, -50, 3]]) = False

  Tipo de la operación:
  esta:: (t, [t]) -> Bool
  ¡Aviso! En este caso al pasar la definición de la operación **esta** al ordenador, el tipo de la operación no se puede poner de manera explícita porque la igualdad (==) en general no está definida siempre.

  Ecuaciones que definen la operación:

```
esta(x, [])  = False                (1)
esta(x, y:s)
    | x == y  = True                (2)
    | x /= y  = esta(x, s)          (3)
```

  La segunda ecuación viene a decir que para decidir si un elemento x está en una lista no vacía y:s se procederá de la siguiente forma: si x es igual al primer elemento de la lista (es decir, al valor y) entonces la respuesta es True pero si x no es igual al primer elemento, entonces hay que mirar si x está en la sublista s.

- **longitud**: Dada una lista de tipo t, devuelve el número de elementos de la lista.

  Ejemplos:

  longitud([]) = 0
  longitud([5, 8, 7, 8]) = 4
  longitud([False, True, True]) = 3
  longitud([[0, 5], [], [77, -50, 3]]) = 3

  Tipo de la operación:
  longitud:: ([t]) -> Int

  Ecuaciones que definen la operación:

```
longitud([])   = 0                  (1)
longitud(x:r)  = 1 + longitud(r)    (2)
```

  Si la lista dada como dato de entrada es vacía (es decir, tiene la forma []) entonces la longitud o número de elementos es 0. Si la lista dada como dato de entrada no es vacía (es decir, tiene la forma x:r siendo x el primer elemento y r la sublista formada por el resto de los elementos), entonces la lista tiene por lo menos un elemento (que es x) y luego hay que contar el número de elementos en la sublista r.

- **numpares**: Dada una lista de tipo Int, devuelve el número de elementos pares de la lista.

  Ejemplos:

  numpares([]) = 0
  numpares([5, 8, 7, 8]) = 2
  numpares([7, 11, 9]) = 0
  numpares([0, 3, 3, 5]) = 1

  Tipo de la operación:
  numpares:: ([Int]) -> Int

  Ecuaciones que definen la operación:

  numpares([]) = 0 (1)
  numpares (x:r)
  | par(x) = 1 + numpares(r) (2)
  | impar(x) = numpares(r) (3)

  Si la lista dada como dato de entrada es vacía (es decir, tiene la forma []) entonces se devuelve 0 ya que esa lista no contiene ningún elemento par. Si la lista dada como dato de entrada no es vacía (es decir, tiene la forma x:r siendo x el primer elemento y r la sublista formada por el resto de los elementos), entonces si el primer elemento (que es x) es par, el número de elementos pares es 1 + el número de elementos pares de r. En cambio, si x no es par, el número de elementos pares coincide con el número de elementos pares en r, ya que x es impar.

  Al definir la función *numpares* se han utilizado las funciones *par* e *impar* definidas con anterioridad. Al definir nuevas funciones, podemos utilizar como funciones auxiliares las funciones que se hayan definido previamente.

- ++: Dadas dos listas de tipo t, devuelve la lista que se obtiene al concatenarlas o juntarlas.

  Ejemplos:

  [] ++ [] = []
  [4, 8, 7, 8] ++ [] = [4, 8, 7, 8]
  [1, 8] ++ [4, 8, 7] = [1, 8, 4, 8, 7]
  [False, True, True] ++ [False, False] = [False, True, True, False, False]

  Tipo de la operación:
  ++:: ([t], [t]) -> [t]

  Ecuaciones que definen la operación:
  [] ++ s = s (1)
  (x:r) ++ s = x:(r ++ s) (2)

  La segunda ecuación viene a decir que para concatenar o juntar una lista x:r y una lista s podemos concatenar r y s y luego añadir el elemento x a la nueva lista r ++ s.

- **intercalar**: Dadas dos listas de tipo t, devuelve la lista que se obtiene intercalando los elementos de las dos listas iniciales, de tal forma que el primer elemento de la primera lista sea el primer elemento de la nueva lista. Si las listas no tienen la misma longitud, se produce un error.

  intercalar([**7, 5, 4**], [8, 2, 0]) = [**7**, 8, **5**, 2, **4**, 0]

  Tipo de la operación:
  intercalar:: ([t], [t]) -> [t]

  Ecuaciones que definen la operación:

```
intercalar([], s)
        | not es_vacia(s)        = error "Listas de longitud diferente."        (1)
        | otherwise              = []                                           (2)

intercalar(x:r, s)
        | longitud(x:r) /= longitud(s)  = error "Listas de longitud diferente."  (3)
        | otherwise                     = x:(primero(s): intercalar(r, resto(s)))  (4)
```

  En este caso se han dado 4 ecuaciones para definir la función *intercalar*.

  Mediante las dos primeras ecuaciones se indica cuál es el resultado si la primera lista es vacía (es decir, si la primera lista es de la forma []). Si la segunda lista (la lista s) no es vacía, las longitudes de las dos listas (la lista [] y la lista s) serán distintas y habrá que mostrar un mensaje de error. Si la segunda lista es vacía, las dos listas tendrán la misma longitud y se devolverá la lista vacía como resultado.

  Mediante las otras dos ecuaciones se indica cuál será el resultado cuando la primera lista no sea vacía, es decir, cuando la primera lista es por ejemplo de la forma x:r, siendo x el primer elemento y r la sublista formada por los demás elementos. En la ecuación (3) se indica que si la longitud de la lista x:r y la longitud de la lista s no son iguales, se mostrará un mensaje de error. En la cuarta ecuación se indica cómo se construye la lista que contiene los elementos de la lista x:r y la lista s intercalados. El primer elemento de la nueva lista será el primer elemento de la lista x:r (es decir, el elemento x). El segundo elemento de la nueva lista será el primer elemento de la lista s, pero ese elemento no tiene nombre y hay que referirse a él como *primero(s)*. Para completar la nueva lista habrá que intercalar los demás elementos de la primera lista, es decir, los elementos de la sublista r y los demás elementos de la segunda lista. Como la sublista formada por los demás elementos de la segunda lista (es decir, los demás elementos de la lista s) no tiene un nombre, esa sublista se expresa como *resto(s)*.

- **intercalar2**: Dadas dos listas de tipo t, devuelve la lista que se obtiene intercalando los elementos de las dos listas iniciales, de tal forma que el primer elemento de la segunda lista sea el primer elemento de la nueva lista. Si las listas no tienen la misma longitud, se produce un error (se muestra un mensaje de error).

  intercalar2([**7, 5, 4**], [8, 2, 0]) = [8, **7**, 2, **5**, 0, **4**]

  Tipo de la operación:
  intercalar2:: ([t], [t]) -> [t]

  Ecuaciones que definen la operación:
  intercalar2([], s)
  | not es_vacia(s) = error "Listas de longitud diferente." (1)
  | otherwise = [] (2)

  intercalar2(x:r, s)
  | longitud(x:r) /= longitud(s) = error "Listas de longitud diferente." (3)
  | otherwise = primero(s):(x:intercalar2(r, resto(s))) (4)

  En estas ecuaciones la idea es la misma que en el caso de la función *intercalar* pero en la ecuación (4) los elementos se han puesto en orden inverso para poder construir la lista que se pide. Esa es la diferencia entre las funciones *intercalar* e *intercalar2*.

5.3.4. Operaciones relacionales para listas

También con las listas es posible utilizar los seis operadores relacionales mencionados en el apartado de los tipos de datos simples.

De cara a decidir si una lista es menor o mayor que otra, se sigue el mismo criterio que se utiliza para ordenar alfabéticamente las palabras.

Por ejemplo [3,3,4] es menor que [5, 1] porque 3 es menor que 5.

5.3.5. Demostración formal de propiedades: Inducción

En este apartado vamos a mostrar mediante ejemplos cómo se demuestra formalmente que una o varias operaciones definidas sobre listas cumplen una determinada propiedad. La demostración está basada en la técnica de la inducción estructural.

Ejemplo 1

Vamos a probar que para toda lista s se cumple lo siguiente:

$$\text{longitud}(s) \geq 0$$

La inducción se realizará sobre la lista s.

- Caso básico o caso simple: s es una lista vacía, es decir, s = []

  $$\text{¿longitud}([]) \geq 0?$$

  longitud([]) = 0 (por la primera ecuación de longitud)
  Por tanto sí se cumple longitud(s) ≥ 0 cuando s es [].

- Caso inductivo o caso general: s no es una lista vacía, es decir, s = x:r

  $$\text{¿longitud}(x:r) \geq 0?$$

  Tenemos que probar que la longitud de x:r es mayor o igual que 0 suponiendo que la longitud de r es mayor o igual que 0.

  - **Hipótesis de inducción** (h.i.): suponemos que r cumple la propiedad, es decir, longitud(r) ≥ 0.
  - **Paso de inducción:** suponiendo que r cumple la propiedad, probar que x:r también la cumple.

    longitud(x:r) = (por la segunda ecuación de longitud)
    = (1 + longitud(r))

    ¿Es (1 + longitud(r)) mayor o igual que 0? Sí, porque por h.i. sabemos que longitud(r) es mayor o igual que 0 y por consiguiente 1 + longitud(r) también ha de ser mayor o igual que 0.

    Una manera más técnica sería la siguiente transformación de la pregunta original:
    ¿longitud(x:r) ≥ 0? (ahora aplicamos la segunda ecuación de longitud)
    ¿1 + longitud(r) ≥ 0? (ahora restamos 1 en ambos lados de la inecuación)
    ¿1 – 1 + longitud(r) ≥ 0 – 1? (realizamos las operaciones)
    ¿longitud(r) ≥ -1? La respuesta a esta pregunta es "Sí" porque por hipótesis de inducción sabemos que longitud(r) ≥ 0 y por tanto también se cumple longitud(r) ≥ -1.

Ejemplo 2

Vamos a probar que dadas dos listas cualesquiera s y r, siempre se cumple lo siguiente:

longitud(s ++ r) = longitud(s) + longitud(r)

La inducción se realizará sobre la lista s.

- Caso básico o caso simple: s es una lista vacía, es decir, s = []

  ¿longitud([] ++ r) = longitud([]) + longitud(r)?

  Hay que comprobar que los dos lados de la ecuación tienen el mismo valor.

  longitud([] ++ r) = (por la primera ecuación de ++)
  = longitud(r)

  longitud([]) + longitud(r) = (por la primera ecuación de longitud)
  = 0 + longitud(r) = (0 es elemento neutro para la suma)
  = longitud(r)

  Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando s es [] la propiedad se cumple.

- Caso inductivo: s no es una lista vacía, es decir, s = x:w

  Tenemos que probar que
  longitud((x:w) ++ r) = longitud(x:w) + longitud(r).

  - **Hipótesis de la inducción** (h.i.): suponemos que para w y r se cumple la propiedad
    longitud(w ++ r) = longitud(w) + longitud(r)

  - **Paso de inducción:** suponiendo que para w y r se cumple la propiedad, probar que también se cumple para x:w y r. Para ello hay que comprobar que los dos lados de la ecuación tienen el mismo valor:

    longitud ((x:w) ++ r) = (utilizando la 2ª ecuación de ++)
    = longitud (x:(w ++ r)) = (utilizando la 2ª ecuación de longitud)
    = **1 + longitud (w ++ r)**

    longitud(x:w) + longitud(r) = (por la 2ª ecuación de longitud)
    = (1 + longitud(w)) + longitud(r) = (por asociatividad de la suma)
    = 1 + (longitud(w) + longitud(r)) = (por h.i.)
    = **1 + longitud(w ++ r)**

    Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando s es de la forma x:w la propiedad se cumple.

En el caso básico hemos probado que cuando s es vacía (es decir, cuando s es de la forma []) la propiedad se cumple y en el caso inductivo hemos probado que cuando s no es vacía (y por consiguiente es, por ejemplo, de la forma x:w) también se cumple. Por tanto, hemos probado que la propiedad se cumple siempre, para cualquier lista s y cualquier lista r.

Ejemplo 3

Vamos a probar que dados un elemento cualquiera x y dos listas cualesquiera s y r, siempre se cumple lo siguiente:

esta(x, s ++ r) = esta(x, s) || esta(x, r)

La inducción se realizará sobre la lista s.

- Caso básico o caso simple: s es una lista vacía, es decir, s = []

  ¿esta(x, [] ++ r) = esta(x, []) || esta(x, r)?

  Hay que comprobar que los dos lados de la ecuación tienen el mismo valor.

  esta(x, [] ++ r) = (por la primera ecuación de ++)
  = **esta(x, r)**

  esta(x, []) || esta(x, r) = (por la primera ecuación de esta)
  = False || esta(x, r) = (False es elemento neutro para ||)
  = **esta(x, r)**

  Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando s es [] la propiedad se cumple.

- Caso inductivo: s no es una lista vacía, es decir, s = z:w

  Tenemos que probar que

  esta(x, (z:w) ++ r) = esta(x, z:w) || esta(x, r)

  - **Hipótesis de la inducción** (h.i.): suponemos que para w y r se cumple la propiedad

    esta(x, w ++ r) = esta(x, w) || esta(x, r)

  - **Paso de inducción:** suponiendo que para w y r se cumple la propiedad, probar que también se cumple para z:w y r. Para ello hay que comprobar que los dos lados de la ecuación tienen el mismo valor. Puesto que al dar las ecuaciones de la función *esta*, cuando la lista dada como segundo argumento no es vacía se han distinguido dos subcasos, la prueba ha de hacerse por separado para cada uno de los subcasos.

(2) x = z

esta(x, (z:w) ++ r) = (utilizando la 2ª ecuación de ++)
= esta(x, z:(w ++ r)) = (por la 2ª ecuación de esta)
= **True**

esta(x, z:w) || esta(x, r) = (por la 2ª ecuación de esta)
= True || esta(x, r) = (True o algo es True)
= **True**

Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando s es de la forma z:w y x = z la propiedad se cumple.

(3) x ≠ z

esta(x, (z:w) ++ r) = (utilizando la 2ª ecuación de ++)
= esta(x, z:(w ++ r)) = (por la 3ª ecuación de esta)
= **esta(x, w ++ r)**

esta(x, z:w) || esta(x, r) = (por la 3ª ecuación de esta)
= esta(x, w) || esta(x, r) = (por h.i.)
= **esta(x, w ++ r)**

Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando s es de la forma z:w y x ≠ z la propiedad se cumple.

En el caso básico hemos probado que cuando s es vacía (es decir, cuando s es de la forma []) la propiedad se cumple y en el caso inductivo hemos probado que cuando s no es vacía (y, por consiguiente, es por ejemplo de la forma z:w) también se cumple (tanto cuando x = z como cuando x ≠ z). Por tanto, hemos probado que la propiedad se cumple siempre, para cualquier x, s y r.

Ejemplo 4

Vamos a probar que dadas tres listas cualesquiera s, r y q, siempre se cumple lo siguiente:

$$(s \mathbin{++} r) \mathbin{++} q = s \mathbin{++} (r \mathbin{++} q)$$

Esta propiedad viene a decir que ++ es asociativa.

La inducción se realizará sobre la lista s.

- Caso básico o caso simple: s es una lista vacía, es decir, s = []

  $$¿([] \mathbin{++} r) \mathbin{++} q = [] \mathbin{++} (r \mathbin{++} q)?$$

  Hay que comprobar que los dos lados de la ecuación tienen el mismo valor.

  ([] ++ r) ++ q = (por la primera ecuación de ++)
  = **r ++ q**

  [] ++ (r ++ q) = (por la primera ecuación de ++)
  = r ++ q

  Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando s es [] la propiedad se cumple.

- Caso inductivo: s no es una lista vacía, es decir, s = x:w

  Tenemos que probar que
  ((x:w) ++ r) ++ q = (x:w) ++(r ++ q)

  - **Hipótesis de la inducción** (h.i.): suponemos que para w, r y q se cumple la propiedad

    $$(w \mathbin{++} r) \mathbin{++} q = w \mathbin{++} (r \mathbin{++} q)$$

  - **Paso de inducción:** suponiendo que para w, r y q se cumple la propiedad, probar que también se cumple para x:w, r y q. Para ello hay que comprobar que los dos lados de la ecuación tienen el mismo valor:

    ((x:w) ++ r) ++ q = (utilizando la 2ª ecuación de ++)
    = (x:(w ++ r)) ++ q = (utilizando la 2ª ecuación de ++)
    = **x:((w ++ r) ++ q)**

    (x:w) ++ (r ++ q) = (utilizando la 2ª ecuación de ++)
    = x:(w ++ (r ++ q)) = (por h.i.)
    = **x:((w ++ r) ++ q)**

    Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando s no es vacía, es decir, es de la forma x:w la propiedad se cumple.

En el caso básico hemos probado que cuando s es vacía (es decir, cuando s es de la forma []) la propiedad se cumple y en el caso inductivo hemos probado que cuando s no es vacía (por consiguiente, s es, por ejemplo, de la forma x:w) también se cumple. Por tanto, hemos probado que la propiedad se cumple siempre, para cualquier s, r y q.

Ejemplo 5

Vamos a probar que para dos listas cualesquiera r y s que sean del mismo tipo y tengan la misma longitud se cumple lo siguiente:

longitud(intercalar(r, s)) = 2 * longitud(r)

La inducción se realizará sobre la lista r.

- Caso básico o caso simple: r es una lista vacía, es decir, r = [].

  ¿longitud(intercalar([], s)) = 2 * longitud([])?

  Hay que calcular los dos lados de la ecuación para comprobar que la propiedad se cumple. Recordemos que r y s tienen la misma longitud y, por tanto, si r es vacía s también será vacía:

longitud(intercalar([], s)) =	(por la primera ecuación de intercalar)
= longitud([]) =	(por la primera ecuación de longitud)
= 0	

2 * longitud([]) =	(por la primera ecuación de longitud)
= 2 * 0 =	(0 es el elemento nulo para *)
= 0	

  Los dos lados de la ecuación dan el mismo resultado, así que hemos probado que cuando r = [] la propiedad se cumple.

- Caso inductivo: r es una lista no vacía, es decir, r = x:w

  Hay que probar lo siguiente:

  longitud(intercalar(x:w, s)) = 2 * longitud(x:w)

  Hay que calcular los dos lados de la ecuación para comprobar que la propiedad se cumple para r y s, suponiendo que la propiedad sí se cumple para w y resto(s).

  - **Hipótesis de la inducción** (h.i.): suponemos que w y resto(s) cumplen la propiedad. En este caso w y resto(s) tienen la misma longitud:

    longitud (intercalar(w, resto(s))) = 2 * longitud(w)

- **Paso de inducción:** suponiendo que w y resto(s) cumplen la propiedad, probar que x:w y s también la cumplen.

  ¿longitud(intercalar(x:w, s)) = 2 * longitud(x:w)?

  Para probar que se cumple la propiedad habrá que calcular cada lado de la ecuación para verificar que dan el mismo resultado.

  longitud(intercalar(x:w, s)) = (por la 2ª ecuación de intercalar)
  = longitud(x:primero(s):intercalar(w, resto(s))) =
  (por segunda ecuación de longitud)
  = 1 + longitud(primero(s):intercalar(w, resto(s))) =
  (por segunda ecuación de longitud)
  = 1 + (1 + longitud(intercalar(w, resto(s)))) =
  (por asociatividad de la suma)
  = 2 + longitud(intercalar(w, resto(s)))

  2 * longitud(x:w) = (por segunda ecuación de longitud)
  = 2 * (1 + longitud(w)) = (por distributividad de * con respecto a +)
  = 2 + (2 * longitud(w)) = (por h.i.)
  = 2 + longitud(intercalar(w, resto(s)))

  Como se puede ver, desarrollando los dos lados de la ecuación se obtiene lo mismo y ello quiere decir que la propiedad también se cumple cuando r no es vacía, es decir, cuando tiene la forma x: w.

Por tanto hemos probado que cuando r es vacía (es decir, cuando r es de la forma []) la propiedad se cumple y que cuando no es vacía (y por consiguiente es, por ejemplo, de la forma x:w) también se cumple, lo que significa que se cumple siempre. Pero es importante recordar que la propiedad se cumple siempre que las dos listas tengan la misma longitud ya que en caso contrario estaríamos ante un caso de error. En este caso se ha partido suponiendo que las dos listas son de la misma longitud.

5.3.6. Simplificación de expresiones

Teniendo en cuenta las ecuaciones que definen las operaciones y las propiedades de las operaciones, a veces es posible simplificar expresiones. A continuación, se muestran cuatro ejemplos indicando en cada paso la ecuación que se ha utilizado para llevar a cabo la simplificación:

Ejemplo 1

longitud(resto (x:s)) → (por la segunda ecuación de resto)
longitud(s)

Ejemplo 2

longitud(resto(7:90:5:[])) → (por segunda ecuación de resto)
longitud(90:5:[]) → (por segunda ecuación de longitud)
1 + longitud(5:[]) → (por segunda ecuación de longitud)
1 + (1 + longitud([])) → (por asociatividad de la suma)
2 + longitud([]) → (por primera ecuación de longitud)
2 + 0 → (0 es el elemento neutro de la suma)
2

Ejemplo 3

longitud((x:s) ++ (resto(y:z:r))) → (por segunda ecuación de resto)
longitud((x:s) ++ (z:r)) → (por segunda ecuación de ++)
longitud(x:(s ++ (z:r))) → (por segunda ecuación de longitud)
1 + longitud(s ++ (z:r)) → (por propiedad del ejemplo 2 del apartado anterior)
1 + (longitud(s) + longitud(z:r)) → (por segunda ecuación de longitud)
1 + (longitud(s) + (1 + longitud(r))) → (la suma es asociativa)
1 + ((longitud(s) + 1) + longitud(r)) → (la suma es asociativa)
(1 + (longitud(s) + 1)) + longitud(r) → (la suma es conmutativa)
(1 + (1 + longitud(s))) + longitud(r) → (la suma es asociativa)
(2 + longitud(s)) + longitud(r)

Ejemplo 4

resto(resto(8:7:resto(1:2:[]))) → (por segunda ecuación de resto)
resto(resto(8:7:2:[])) → (por segunda ecuación de resto)
resto(7:2:[])) → (por segunda ecuación de resto)
2:[]

5.4. Pilas

Una pila se representa habitualmente de la siguiente forma:

2
7
35
-8
0

A diferencia de las listas, el tipo de las pilas no está predefinido en Haskell.

5.4.1. Operaciones constructoras para pilas

Las operaciones constructoras sirven para expresar de manera formal y general qué valores pertenecen al tipo.

Para definir un tipo nuevo hay que indicar cuáles son sus operaciones constructoras.

Definiremos el tipo de las pilas de la siguiente forma:

```
data Pila t = Pvacia | Apilar (t, Pila t)
```

Para las pilas las operaciones constructoras son Pvacia y Apilar.

Esa definición expresa que una pila cuyos elementos son de tipo t (donde t puede ser Int, Bool, etc.) o es una pila vacía (lo que se expresa mediante Pvacia) o es una pila que se ha obtenido apilando un elemento de tipo t sobre una pila de tipo t.

Las operaciones constructoras Pvacia y Apilar son del siguiente tipo:

```
Pvacia :: Pila t
Apilar:: (t, Pila t) -> Pila t
```

Es decir, la operación Pvacia genera una pila vacía de tipo t a partir de la nada (es decir, sin datos de entrada) y la operación Apilar, dados un elemento de tipo t y una pila de tipo t, devuelve una pila nueva colocando el elemento sobre la pila.

Consideremos la siguiente pila:

2
7
35

Esa pila la expresaríamos así:

Apilar(2, Apilar(7, Apilar(35, Pvacia)))

Lo que viene a indicar que para generar esa pila primero generamos la pila vacía:

Sobre la pila vacía colocamos el número 35 y así obtenemos una pila que tiene ese número:

35

Sobre esa pila colocamos el número 7 y así obtenemos la siguiente pila:

7
35

Por último, colocamos el número 2 sobre esa pila y obtenemos la siguiente pila:

2
7
35

El tipo de las pilas es un tipo de datos paramétrico ya que dependiendo del parámetro t podemos tener pilas de booleanos, pilas de enteros, pilas de listas de enteros, pilas de pilas de enteros, etc. Por tanto, se pueden definir pilas de cualquier tipo que ya esté definido pero todos los elementos de la pila han de ser del mismo tipo.

Ejemplos:

La siguiente pila es de tipo **Pila Bool**

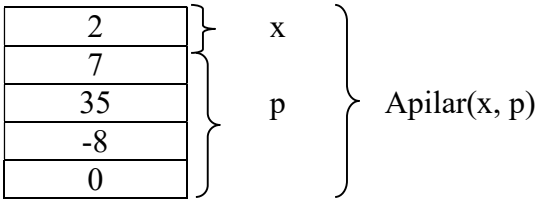
False
True
False
False

La siguiente pila es de tipo **Pila [Int]**

[0, 5]
[]
[77, -50, 3]

Toda pila tiene uno de los dos siguientes formatos:

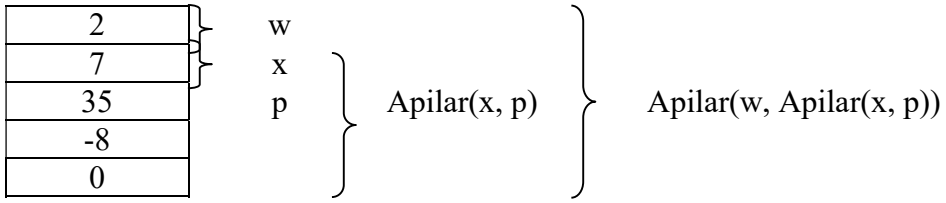
- ✓ Pvacia (es una pila vacía)
- ✓ Apilar(x, p) (es una pila no vacía donde el elemento de la cima es x y el resto de elementos conforman una subpila p) (en vez de x y p se pueden utilizar otros nombres)



Cuando una pila tiene dos o más elementos, es posible decir que la pila tiene el formato

Apilar(w, Apilar(x, p))

de tal forma que el elemento de la cima es **w**, el segundo desde la cima es **x** y el resto de elementos conforman una subpila **p**.



5.4.2. Algunas operaciones no constructoras para pilas

A continuación, se definen algunas operaciones que sirven para realizar cálculos con pilas. Para definir una función u operación hay que dar el tipo de la función y las ecuaciones que indican qué hace la función.

- **cima**: Dada una pila, devuelve el elemento que está en la cima de la pila. Si la pila es vacía se genera un error (se muestra un mensaje de error).

  Ejemplos:
  cima(Apilar(2, Apilar(7, Apilar(35, Pvacia)))) = 2
  cima(Pvacia) = error

  Tipo de la operación:
  cima:: (Pila t) -> t

  Ecuaciones que definen la operación:
  cima(Pvacia) = error "Pila vacía. No hay cima." (1)
  cima(Apilar(x, p)) = x (2)

Para generar mensajes de error se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **desapilar**: Dada una pila, devuelve la pila que se obtiene al eliminar el elemento de la cima de la pila. Si la pila es vacía se genera un error (se muestra un mensaje de error).

  Ejemplos:
  desapilar(Apilar(2, Apilar(7, Apilar(35, Pvacia)))) =
  = Apilar(7, Apilar(35, Pvacia))
  desapilar(Pvacia) = error

  Tipo de la operación:
  desapilar:: (Pila t) -> Pila t

  Ecuaciones que definen la operación:
  desapilar(Pvacia) = error "Pila vacía. No se puede desapilar." (1)
  desapilar(Apilar(x, p)) = p (2)

  Para generar el mensaje de error correspondiente, se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **es_pvacia**: Dada una pila, devuelve True si la pila es vacía y devuelve False si no es vacía.

  Ejemplos:
  es_pvacia(Pvacia) = True
  es_pvacia(Apilar(2, Apilar(7, Apilar(35, Pvacia)))) = False

  Tipo de la operación:
  es_pvacia:: (Pila t) -> Bool

  Ecuaciones que definen la operación:
  es_pvacia(Pvacia) = True (1)
  es_pvacia(Apilar(x, p)) = False (2)

  Si la pila dada como dato de entrada es vacía (es decir, tiene la forma Pvacia) entonces se devuelve True. Si la pila dada como dato de entrada no es vacía (es decir, tiene la forma Apilar(x, p) siendo x el elemento de la cima y p la subpila formada por el resto de los elementos), entonces se devuelve False.

5.5. Colas

Una cola es una estructura lineal parecida a la lista pero los elementos se han de añadir al final (por la derecha). Representaremos las colas gráficamente de la siguiente forma:

<< 2, 7, 35, -8, 0 >>

A diferencia de las listas, el tipo de las colas no está predefinido en Haskell y, por tanto, Haskell no reconoce el formato << 2, 7, 35, -8, 0 >>. Ese formato será una manera gráfica que utilizaremos nosotros para representar las colas de manera informal.

5.5.1. Operaciones constructoras para colas

Las operaciones constructoras sirven para expresar de manera formal y general qué valores pertenecen al tipo.

Para definir un tipo nuevo hay que indicar cuáles son sus operaciones constructoras.

Definiremos el tipo de las colas de la siguiente forma:

```
data Cola t = Cvacia | Poner(Cola t, t)
```

Para las colas las operaciones constructoras son Cvacia y Poner.

Esa definición expresa que una cola cuyos elementos son de tipo t (donde t puede ser Int, Bool, etc.) o es una cola vacía (lo que se expresa mediante Cvacia) o es una cola que se ha obtenido poniendo un elemento de tipo t al final de una cola de tipo t.

Las operaciones constructoras Cvacia y Poner son del siguiente tipo:

```
Cvacia :: Cola t
Poner:: (Cola t, t) -> Cola t
```

Es decir, la operación Cvacia genera una cola vacía de tipo t a partir de la nada (es decir, sin datos de entrada) y la operación Poner, dados un elemento de tipo t y una cola de tipo t, devuelve una cola nueva colocando el elemento al final de la cola.

Consideremos la siguiente cola:

<< 35, -8, 0 >>

Esa cola la expresaríamos así:

Poner(Poner(Poner(Cvacia, 35), -8), 0)

Lo que viene a indicar que para generar esa cola primero generamos la cola vacía:

<< >>

Al final de la cola vacía ponemos el número 35 y así obtenemos una cola que tiene ese número:

<< 35 >>

Al final de esa cola ponemos el número 7 y así obtenemos la siguiente pila:

<< 35, 7 >>

Por último, ponemos el número 2 al final de la cola y obtenemos la siguiente cola:

<< 35, 7, 2 >>

El tipo de las colas es un tipo de datos paramétrico ya que dependiendo del parámetro t podemos tener colas de booleanos, colas de enteros, colas de listas de enteros, colas de pilas de enteros, etc. Por tanto, se pueden definir colas de cualquier tipo que ya esté definido, pero todos los elementos de la cola han de ser del mismo tipo.

Ejemplos:

La siguiente cola es de tipo **Cola Bool**

<< True, False, False, False >>

La siguiente cola es de tipo **Cola [Int]**

<< [0, 5], [], [77, -50, 3] >>

Toda cola tiene uno de los dos siguientes formatos:

- ✓ Cvacia (es una cola vacía)
- ✓ Poner(c, x) (es una cola no vacía donde el último elemento es x y c es la subcola formada por los demás elementos) (en vez de c y x se pueden utilizar otros nombres)

Cuando una cola tiene dos o más elementos, es posible decir que la cola tiene el formato

Poner(Poner(c, x), w)

donde el último elemento de la cola es **w**, el penúltimo es **x** y el resto de elementos conforman una subcola **c**.

5.5.2. Algunas operaciones no constructoras para colas

A continuación, se definen algunas operaciones que sirven para realizar cálculos con colas. Para definir una función u operación hay que dar el tipo de la función y las ecuaciones que indican qué hace la función.

- **es_cvacia**: Dada una cola, devuelve True si la cola es vacía y devuelve False si no es vacía.

  Ejemplos:

  es_cvacia(Cvacia) = True
  es_cvacia(Poner(Poner(Poner(Pvacia, 23), 45), 0), 10) = False

  Tipo de la operación:
  es_cvacia:: (Cola t) -> Bool

  Ecuaciones que definen la operación:
  es_cvacia(Cvacia) = True (1)
  es_cvacia(Poner(c, x)) = False (2)

  Si la cola dada como dato de entrada es vacía (es decir, tiene la forma Cvacia) entonces se devuelve True. Si la cola dada como dato de entrada no es vacía (es decir, tiene la forma Poner(c, x), siendo x el último elemento de la cola y c la subcola formada por el resto de los elementos, entonces se devuelve False.

- **principio**: Dada una cola, devuelve el primer elemento de la cola. Si la cola es vacía se genera un error (se muestra un mensaje de error).

  Ejemplos:

  principio(Poner(Poner(Poner(Cvacia, 3), 8), 34), 10) = 3
  principio(Cvacia) = error

  Tipo de la operación:
  principio:: (Cola t) -> t

  Ecuaciones que definen la operación:
  principio(Cvacia) = error "Cola vacía. No hay elementos." (1)
  principio(Poner(c, x))
  | es_cvacia(c) = x (2)
  | otherwise = principio (c) (3)

  Para generar mensajes de error se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **quitar**: Dada una cola, devuelve la cola que se obtiene al eliminar el primer elemento de la cola. Si la cola es vacía se genera un error (se muestra un mensaje de error).

  Ejemplos:
  quitar(Poner(Poner (Poner (Cvacia, 2), 5), 20)) =
  = Poner(Poner(Cvacia, 5), 20)
  quitar(Cvacia) = error

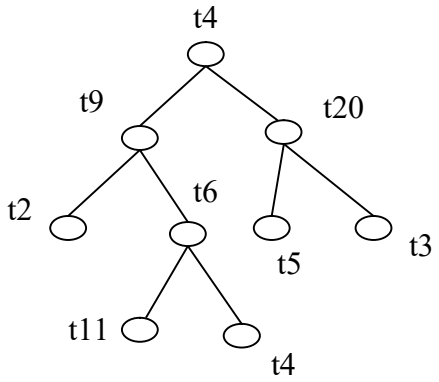
  Tipo de la operación:
  quitar:: (Cola t) -> Cola t

  Ecuaciones que definen la operación:
  quitar(Cvacia) = error "Cola vacía. No se puede avanzar." (1)
  quitar(Poner(c, x))
  | es_cvacia(c) = Cvacia (2)
  | otherwise = Poner(quitar(c), x) (3)

  Para generar el mensaje de error correspondiente, se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

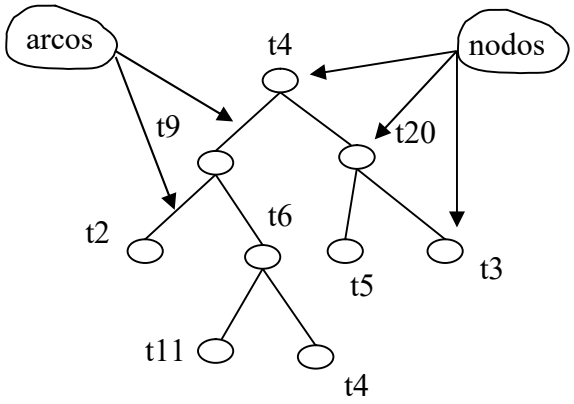
5.6. Árboles binarios

Un árbol binario de tipo t es una estructura no lineal cuyos elementos son de tipo t

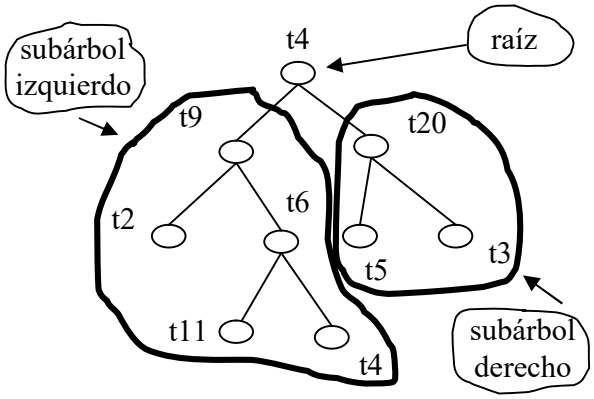


En el gráfico consideramos que los elementos t2, t3, t4, etc. son valores de tipo t.

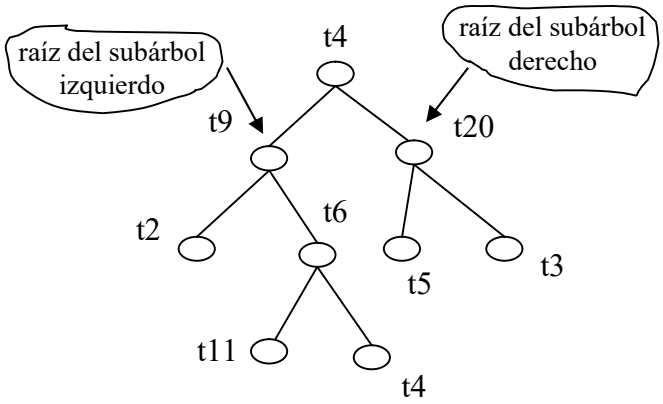
Un árbol binario está formado por nodos y arcos dirigidos. Los nodos tienen asociado un valor de tipo t.

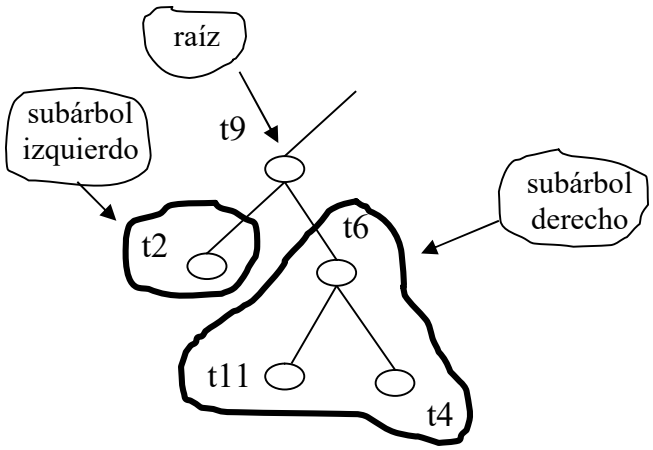


En un árbol binario se pueden diferenciar la raíz, el subárbol izquierdo y el subárbol derecho:



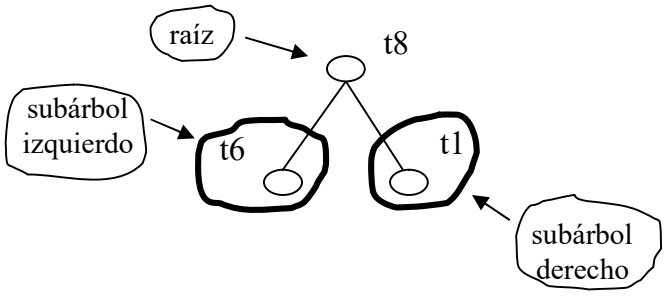
El subárbol izquierdo (al igual que el derecho) está formado a su vez por una raíz, un subárbol izquierdo y un subárbol derecho:



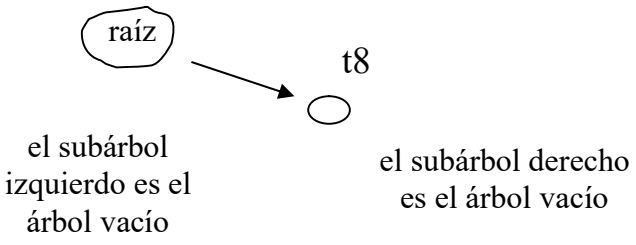


Y de esa forma un árbol puede ser descompuesto en raíz, subárbol izquierdo y subárbol derecho. Y a su vez los subárboles admiten la misma descomposición, hasta llegar a un árbol cuyos subárboles son vacíos.

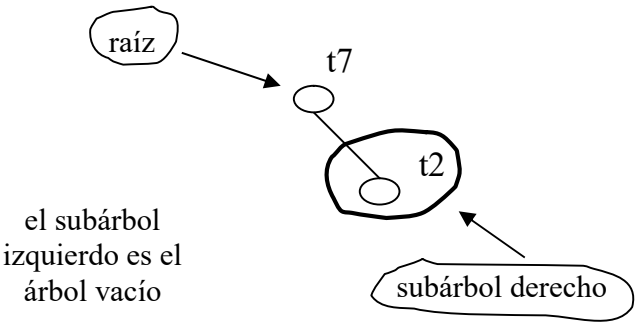
En el siguiente árbol tanto el subárbol izquierdo como el derecho constan de un único nodo:



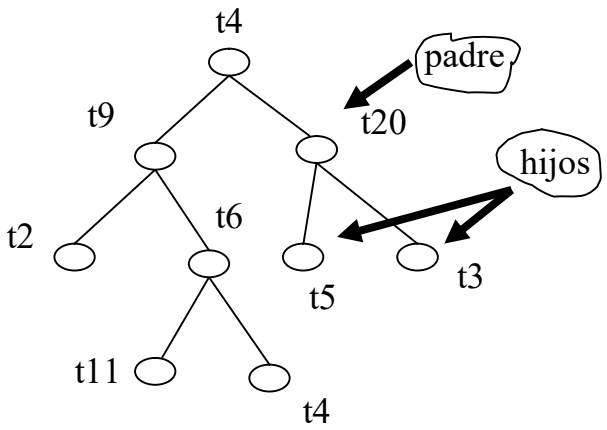
El siguiente árbol consta de un único nodo, la raíz. Los subárboles izquierdo y derecho son vacíos:



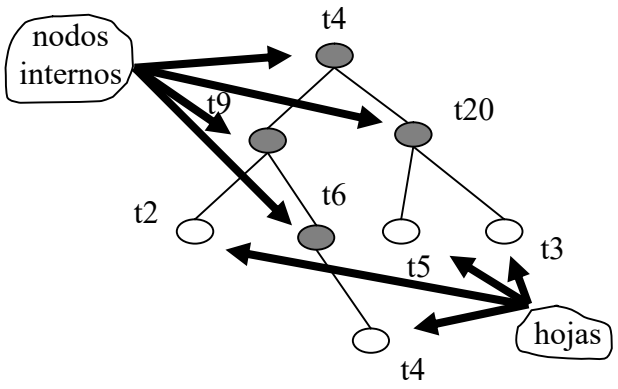
El siguiente árbol consta de dos nodos. El subárbol izquierdo es vacío:



Un nodo es el padre de las raíces de sus subárboles izquierdo y derecho. Dichas raíces reciben el nombre de hijo izquierdo e hijo derecho, respectivamente. Los arcos se entienden dirigidos de padres a hijos:



Llamaremos hoja a cualquier nodo cuyos subárboles son vacíos.
Llamaremos nodo interno a cualquier nodo que tiene al menos un subárbol no vacío.



5.6.1. Operaciones constructoras para árboles binarios

Las operaciones constructoras sirven para expresar de manera formal y general qué valores pertenecen al tipo.

El tipo de los árboles binarios no está predefinido en Haskell.
Para definir un tipo nuevo hay que indicar cuáles son sus operaciones constructoras.

Definiremos el tipo de los árboles binarios de la siguiente forma:

```
data Arbin t = Avacio | Crear (t, Arbin t, Arbin t)
```

Para los árboles binarios las operaciones constructoras son Avacio y Crear.

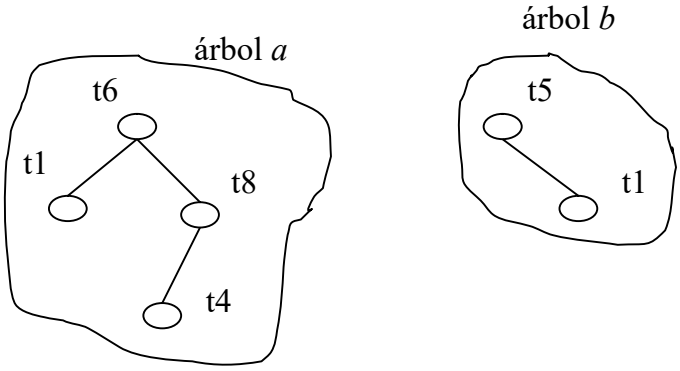
Esa definición expresa que un árbol binario cuyos elementos son de tipo t (donde t puede ser Int, Bool, etc.) o es un árbol vacío (lo que se expresa mediante Avacio) o es un árbol que se ha obtenido o creado a partir de dos árboles binarios y un elemento, haciendo que el elemento sea la raíz del nuevo árbol y que los dos árboles sean el subárbol izquierdo y el subárbol derecho para el nuevo árbol.
Dicho de otra forma, las operaciones constructoras son la que genera un árbol vacío y la que construye un nuevo árbol binario a partir de un nodo (raíz) y dos árboles binarios (subárbol izquierdo y subárbol derecho):

Las operaciones constructoras Avacio y Crear son del siguiente tipo:

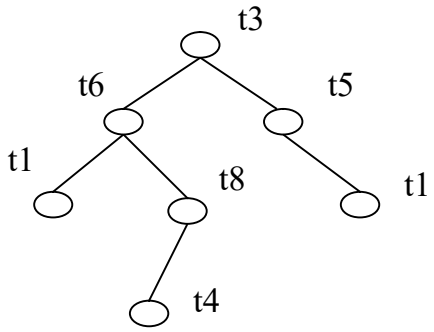
```
Avacio :: Arbin t
Crear:: (t, Arbin t, Arbin t) -> Arbin t
```

Es decir, la operación Avacio genera un árbol binario vacío de tipo t a partir de la nada (es decir, sin datos de entrada) y la operación Crear, dados un elemento de tipo t y dos árboles binarios de tipo t, devuelve un árbol binario nuevo.

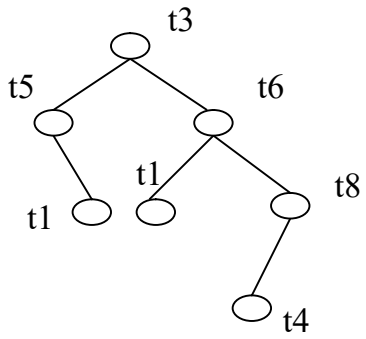
Si tenemos dos árboles *a* y *b*:



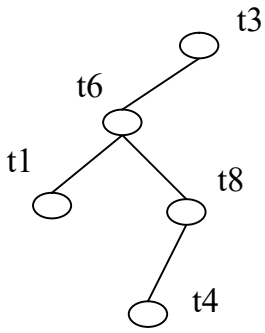
El árbol Crear(t3, *a*, *b*) es el siguiente árbol:



Y el árbol Crear(t3, *b*, *a*) es el siguiente árbol:



El árbol Crear(t3, *a*, Avacio) es el siguiente árbol:



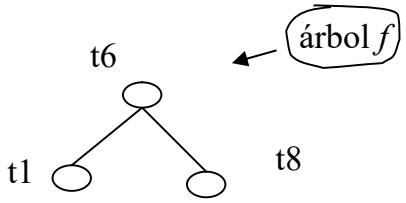
El árbol Crear(t3, Avacio, Avacio) es el siguiente árbol:

Por medio de las operaciones constructoras podemos expresar cualquier árbol binario.

Un árbol vacío lo expresaríamos de la siguiente forma:

Avacio

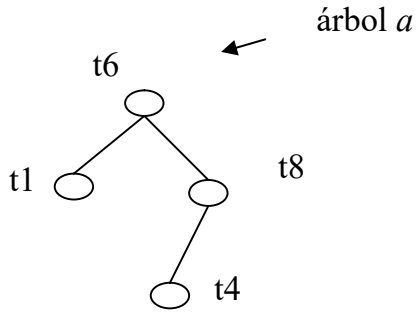
Consideremos el siguiente árbol binario, al que denominaremos *f*:



El árbol *f* lo expresaríamos de la siguiente forma:

f = Crear(t6, Crear(t1, Avacio, Avacio), Crear(t8, Avacio, Avacio))

Ahora consideremos el siguiente árbol binario *a*:



Utilizando las operaciones constructoras, ese árbol lo expresaríamos de la siguiente forma:

a = Crear(t6, Crear(t1, Avacio, Avacio),
Crear(t8, Crear(t4, Avacio, Avacio),
Avacio
)
)

5.6.2. Algunas operaciones no constructoras para árboles binarios

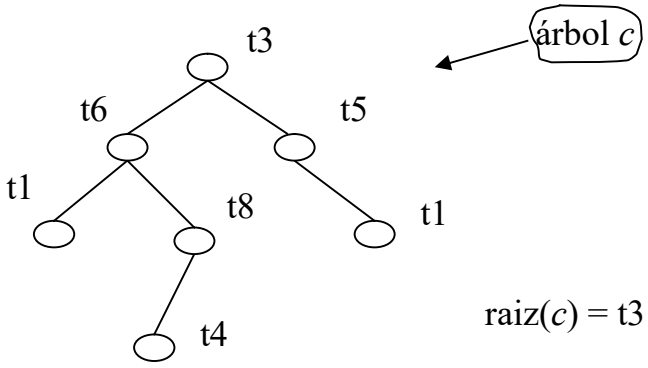
A continuación, se definen cuatro operaciones no constructoras, es decir, operaciones que sirven para realizar cálculos con árboles binarios. Para definir una función u operación hay que dar el tipo de la función y las ecuaciones que indican qué hace la función.

- **raiz**: dado un árbol devuelve el valor de la raíz. Si se aplica a árboles vacíos se genera un mensaje de error.

  Ejemplos:

  raiz(Avacio) = error

  Sea *c* el siguiente árbol binario:



raiz(*c*) = t3

Tipo de la operación:
raiz:: (Arbin t) -> t

Ecuaciones que definen la operación:
raiz(Avacio) = error "Árbol vacío. No hay raíz." (1)
raiz(Crear(x, *a*, *b*)) = x (2)

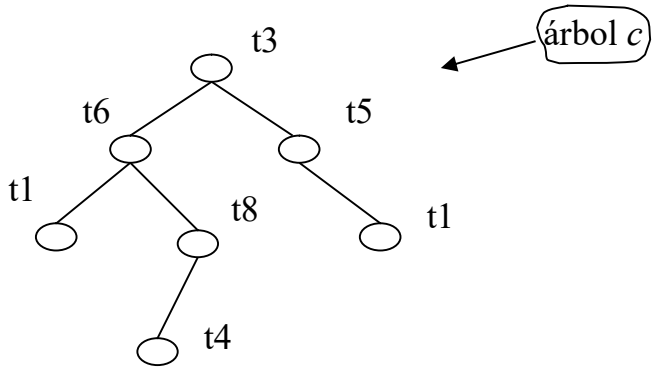
Para generar mensajes de error se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **izqdo**: dado un árbol devuelve el subárbol izquierdo. Si se aplica a árboles vacíos se genera un mensaje de error.

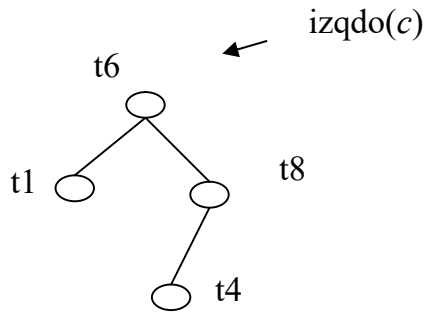
Ejemplos:

izqdo(Avacio) = error

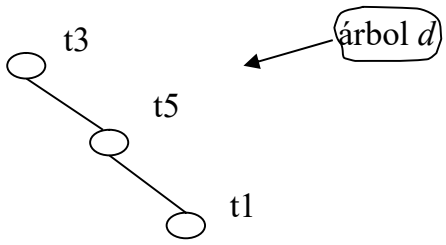
Sea *c* el siguiente árbol binario:



La operación izqdo devuelve el subárbol izquierdo:



Sea *d* el siguiente árbol binario:



izqdo(*d*) = Avacio

La operación izqdo devuelve el subárbol izquierdo:

izqdo(*d*) = Avacio

Tipo de la operación:
izqdo:: (Arbin t) -> Arbin t

Ecuaciones que definen la operación:
izqdo(Avacio) = error "Árbol vacío. No hay subárbol izquierdo." (1)
izqdo(Crear(x, *a*, *b*)) = *a* (2)

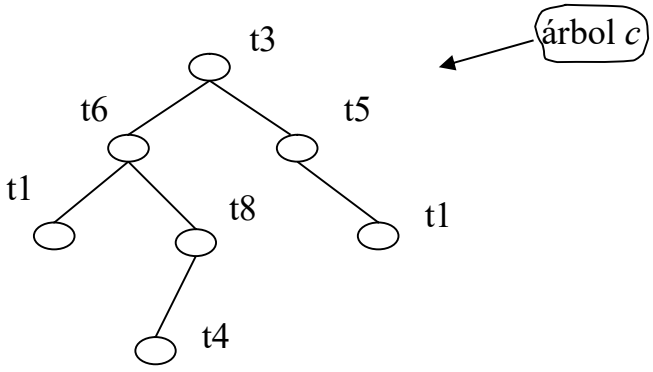
Para generar mensajes de error se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **drcho**: dado un árbol devuelve el subárbol derecho. Si se aplica a árboles vacíos se genera un mensaje de error.

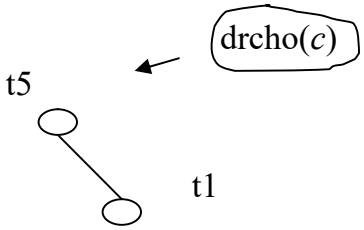
Ejemplos:

drcho(Avacio) = error

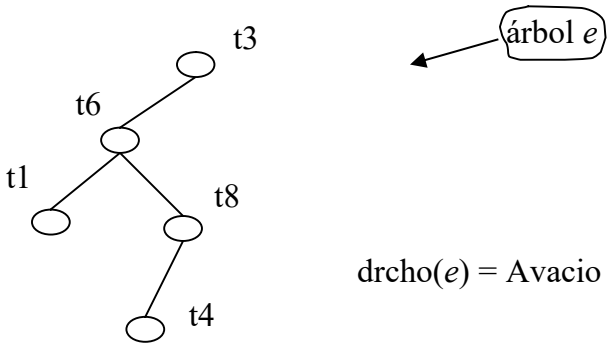
Sea *c* el siguiente árbol binario:



La operación drcho devuelve el subárbol derecho:



Sea *e* el siguiente árbol binario:



drcho(*e*) = Avacio

Tipo de la operación:
drcho:: (Arbin t) -> Arbin t

Ecuaciones que definen la operación:
drcho(Avacio) = error "Árbol vacío. No hay subárbol derecho." (1)
drcho(Crear(x, *a*, *b*)) = *b* (2)

Para generar mensajes de error se utiliza la función *error* de Haskell seguido por el mensaje que se quiera mostrar entre comillas dobles.

- **es_avacio**: Dado un árbol binario, devuelve True si el árbol es vacío y devuelve False si no es vacío.

  Ejemplos:
  es_avacio(Avacio) = True
  es_avacio(Crear(2, Crear(7, Crear(6, Avacio, Avacio)), Avacio)) = False

  Tipo de la operación:
  es_avacio:: (Arbin t) -> Bool

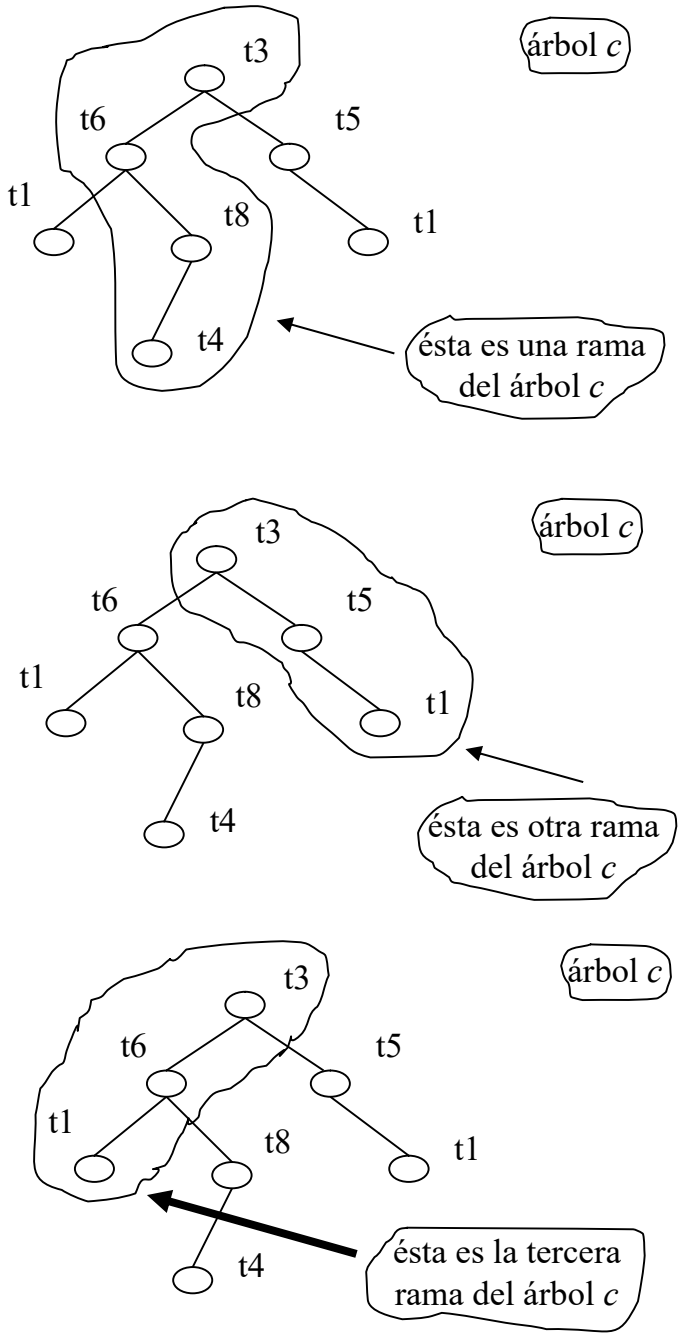
  Ecuaciones que definen la operación:
  es_avacia(Avacia) = True (1)
  es_avacia(Crear(x, *a*, *b*)) = False (2)

- **prof**: dado un árbol devuelve la profundidad del árbol, es decir, la longitud de la rama más larga. Devuelve 0 si el árbol es vacío. Una rama la conforman los nodos que van desde la raíz hasta una de las hojas

  Ejemplos:

  prof(Avacio) = 0

  Sea *c* el siguiente árbol binario:



El árbol *c* tiene tres ramas y la más larga tiene cuatro nodos, por tanto:
prof(*c*) = 4

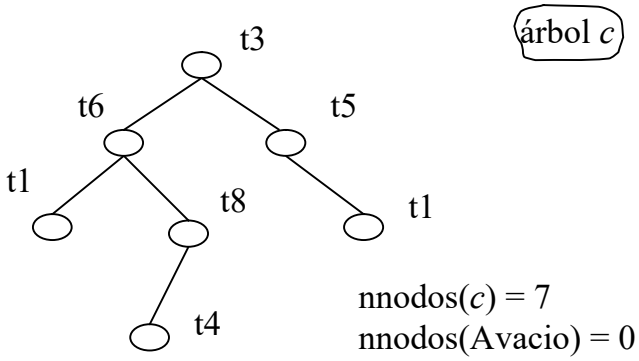
Tipo de la operación:
prof:: (Arbin t) -> Int

Ecuaciones que definen la operación:

prof(Avacio) = 0		(1)
prof(Crear(x, *a*, *b*))		
| prof(*a*) >= prof(*b*)	= 1 + prof(*a*)	(2)
| otherwise	= 1 + prof(*b*)	(3)

- **nnodos**: La operación nnodos devuelve el número de nodos del árbol binario

Ejemplos:



Tipo de la operación:
nnodos:: (Arbin t) -> Int

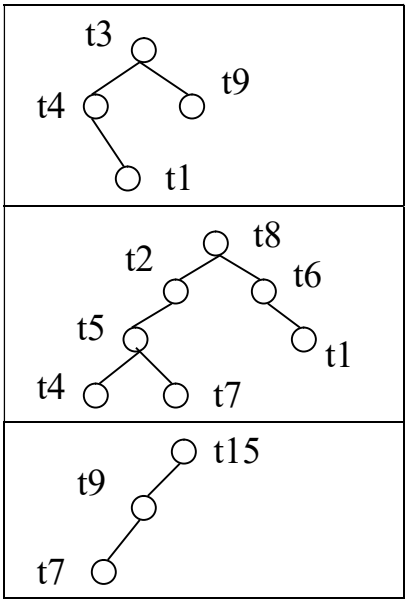
Ecuaciones que definen la operación:

nnodos(Avacio) = 0	(1)
nnodos(Crear(x, *a*, *b*)) = 1 + nnodos(*a*) + nnodos(*b*)	(2)

5.7. Ejemplos con mezcla de tipos abstractos de datos

Dada una pila de árboles binarios, la operación profmax devuelve la profundidad del árbol más profundo:

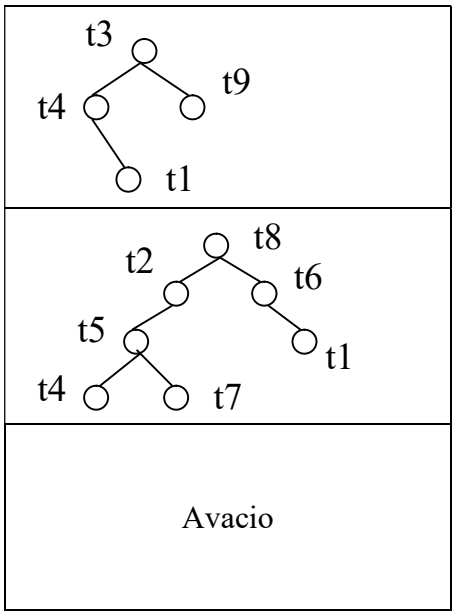
Sea la siguiente pila de árboles binarios p



La profundidad máxima se da en el árbol que está en medio y es 4:

profmax(p) = 4

Es posible que haya árboles binarios vacíos en la pila. Sea la siguiente pila q:



profmax(q) = 4

Tipo de la operación:
profmax:: Pila (Arbin t) -> Int

Ecuaciones que definen la operación:

profmax(Pvacia) = 0		(1)
profmax(Apilar(*a*, p))		
| prof(*a*) >= profmax(p)	= prof(*a*)	(2)
| otherwise	= profmax(p)	(3)

Donde *a* es un árbol binario, p una pila de árboles binarios y prof es la función que calcula la profundidad de un árbol binario.

5.8. Modularidad en Haskell

5.8.1. Definición de dos módulos con operaciones de apartados anteriores

En Haskell resulta muy sencillo definir distintos módulos y hacer uso de las funciones definidas en un módulo desde otro módulo.

Por ejemplo, teniendo en cuenta las funciones definidas en este tema, podríamos definir dos módulos:

a) Uno para funciones de los tipos básicos (par, impar, max3)
b) Otro para las funciones definidas para las listas (primero, resto, es_vacia, esta, longitud, intercalar, intercalar2).

Cada módulo irá en un fichero, así que vamos a indicar cómo quedaría cada fichero.

a) Fichero "Basicos.hs"

```
module Basicos where

par:: (Int) -> Bool
par (x) = (x `mod` 2) == 0

impar:: (Int) -> Bool
impar (x) = not (par(x))

max3:: (Int, Int, Int) -> Int
max3 (x, y, z)
                | x >= y && x >= z          = x
                | y > x && y >= z           = y
                | otherwise                 = z
```

b) Fichero "Listas.hs"

```
module Listas where
import Basicos

        primero:: ([t]) -> t
        primero([]) = error "Lista vacía. No se puede obtener el primer elemento."
        primero(x:s) = x

        resto:: ([t]) -> [t]
        resto([]) = error "Lista vacía. No se puede obtener el resto."
        resto(x:s) = s

        es_vacia:: ([t]) -> Bool
        es_vacia([]) = True
        es_vacia(x:s) = False

        esta:: (t, [t]) -> Bool
        esta(x, []) = False
        esta(x, y:s)
                | x == y        = True
                | x /= y        = esta (x, s)

        longitud:: ([t]) -> Int
        longitud([]) = 0
        longitud(x:r) = 1 + longitud(r)

        numpares:: ([Int]) -> Int
        numpares([]) = 0
        numpares(x:r)
                | par(x)        = 1 + numpares(r)
                | impar(x)      = numpares(r)

        intercalar:: ([t], [t]) -> [t]
        intercalar([], s)
                | not es_vacia(s)       = error "Listas de longitud diferente."
                | otherwise             = []
        intercalar(x:r, s)
                | longitud(x:r) /= longitud(s)  = error "Listas de longitud diferente."
                | otherwise                     = x:(primero(s): intercalar(r, resto(s)))

        intercalar2:: ([t], [t]) -> [t]
        intercalar2([], s)
                | not es_vacia(s)       = error "Listas de longitud diferente."
                | otherwise             = []
        intercalar2(x:r, s)
                | longitud(x:r) /= longitud(s)  = error "Listas de longitud diferente."
                | otherwise                     = primero(s):(x:intercalar2(r, resto(s)))
```

Las operaciones constructoras [] y : y ++ están predefinidas en Haskell.

5.9. Observaciones

5.9.1. Uso de mayúsculas y minúsculas en los nombres

De cara a utilizar mayúsculas y minúsculas es necesario seguir las siguientes reglas:

- Los nombres de los tipos de datos empiezan con mayúscula: Int, Bool, Char, Pila t,…
- Los nombres de las operaciones constructoras empiezan con mayúscula: Pvacia, Apilar, Avacio, Crear, …
- Las constantes True y False empiezan con mayúscula.
- Los nombres de las variables que sustituyen a los tipos empiezan con minúscula: t.
- Todas las demás operaciones y variables empiezan por minúscula (aunque pueden llevar mayúsculas intercaladas): par, impar, primero, resto, esta, longitud, eliminarLong, eliminarLongPar, x, s, r, p, q, …

5.9.2. Dos maneras de introducir comentarios

Para incluir comentarios en los programas Haskell hay dos posibilidades:

- Utilizando --:

  Todo lo que venga después de -- es un comentario hasta el final de la línea.

  Ejemplo:

  ```
  --Esto es un comentario pero termina donde acaba la línea
  ```

- Utilizando {- y -}:

  Todo lo que vaya entre {- y -} es un comentario. No importa el número de líneas.

  Ejemplo:

  ```
  {- Esto es un comentario pero
  podemos saltar de línea -}
  ```