

# **METODOLOGÍA DE LA PROGRAMACIÓN**

**GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN**

**ESCUELA DE INGENIERÍA DE BILBAO (UPV/EHU)**

**DEPARTAMENTO DE LENGUAJES Y SISTEMAS INFORMÁTICOS**

**1º**

**Grupo 01**

## **TEMA 6**

### **TRANSFORMACIÓN DE PROGRAMAS RECURSIVOS**

**MÉTODO DE BURSTALL: TRANSFORMACIÓN DE RECURSIVO A ITERATIVO**

**2022-2023**

Profesor: José Gaintzarain Ibarmia  
Despacho P3I 40  
Horario de tutorías: mirar en GAUR



**ÍNDICE**

6.1. INTRODUCCIÓN Y MOTIVACIÓN .....	5
6.2. PASOS GENERALES DEL MÉTODO DE BURSTALL.....	7
6.3. APLICACIÓN DEL MÉTODO .....	9
6.3.1. <i>Funciones recursivas elementales (un caso simple y uno recursivo)</i> .....	9
6.3.2. <i>Funciones con operación no asociativa</i> .....	16
6.3.3. <i>Funciones con más de un caso simple</i> .....	22
6.3.4. <i>Funciones con más de un caso recursivo</i> .....	28
6.4. RESUMEN Y CONCLUSIONES.....	36



### 6.1. INTRODUCCIÓN Y MOTIVACIÓN

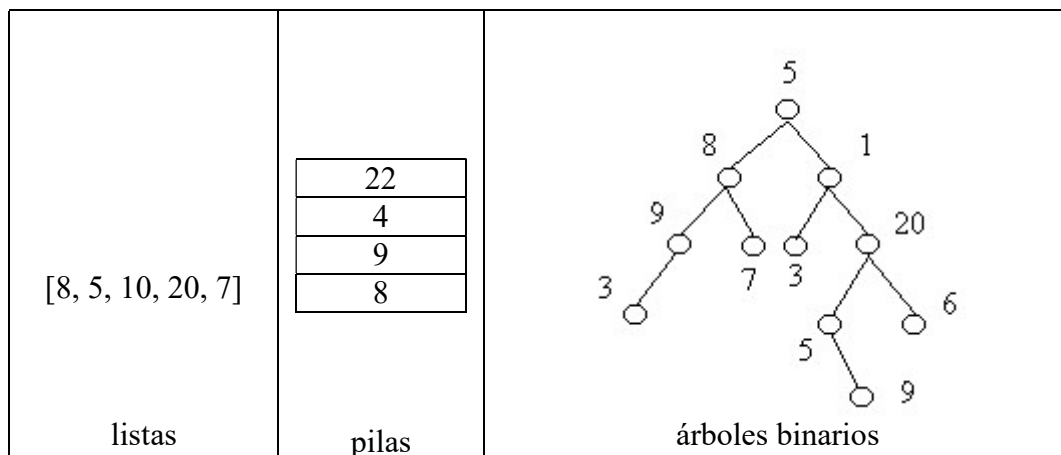
Hay problemas para los cuales la solución más directa, simple y clara es un programa recursivo. Entre ellos podemos citar tres tipos generales:

- Los problemas de cuyo planteamiento forma parte una definición matemática inductiva. Por ejemplo, la función factorial o la sucesión de Fibonacci.

$$\begin{aligned} \text{factorial}(0) &= 1 \\ \text{factorial}(n) &= n * \text{factorial}(n - 1) \end{aligned}$$

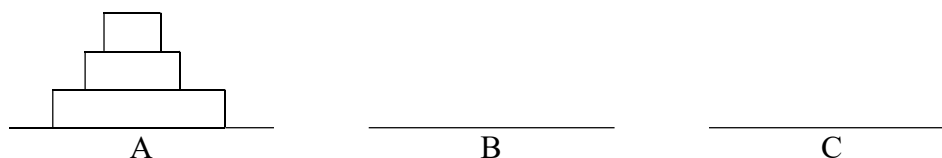
$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \end{aligned}$$

- Los problemas cuyos datos y/o resultados son estructuras recursivas. Por ejemplo, las operaciones sobre listas, pilas y árboles binarios.



- Los problemas que son tareas cuya solución más inmediata se basa en la descomposición en subtarefas del mismo tipo. Por ejemplo, el problema de las torres de Hanoi.

¿Cómo mover las tres cajas que están en la posición A a la posición B, si cada vez se puede mover sólo una caja y si sobre una caja no se puede colocar otra caja que sea mayor que ella? A modo de ayuda se puede utilizar la posición C.



También es muy común que, por el contrario, la solución iterativa a dichos problemas no sea simple de diseñar.

La simplicidad y claridad de la solución recursiva conlleva, generalmente, una prueba de corrección también sencilla. Realmente en muchos casos el programa recursivo (aun escrito en un lenguaje imperativo) es muy similar a un programa declarativo o especificación ecuacional.

En ocasiones, la solución recursiva más directa es ineficiente. Los problemas que habitualmente causan la ineficiencia de programas recursivos son las repeticiones de cálculos y el tiempo y el espacio dedicado a almacenamiento y recuperación de parámetros y variables locales. Es por ello que nos puede interesar transformar metódicamente la solución recursiva (simple y clara) en una iterativa. Utilizando este tipo de transformación tenemos la solución recursiva como punto de partida y, además, la solución iterativa puede quedar justificada y documentada en base a la original (que era directa y cercana a una especificación).

El método de Burstall es una técnica que sirve para transformar funciones recursivas en iterativas y se basa en el desplegado-plegado y en el invariante conservado por la función recursiva.

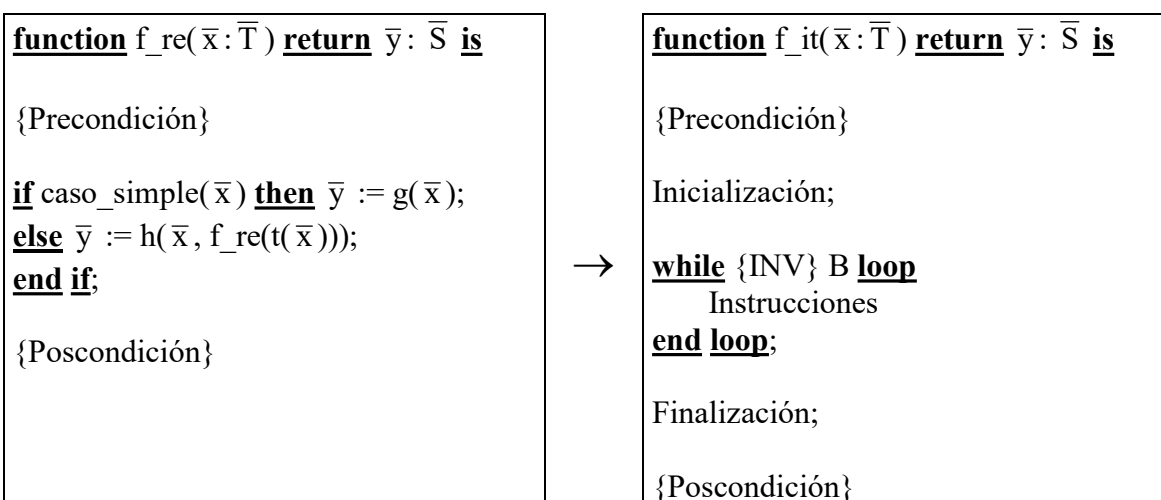
El conocimiento de este tipo de métodos (como formación en programación) es especialmente interesante por conllevar el estudio profundo de la recursión y su relación con la iteración.

Cuando se parte de un programa, la transformación puede ser considerada un método de diseño horizontal de programas en contraposición al diseño descendente. Pero también puede ser aplicada a especificaciones ecuacionales, y en ese caso la transformación puede ser vista o entendida como método de diseño de programas a partir de especificaciones.

## 6.2. PASOS GENERALES DEL MÉTODO DE BURSTALL

La idea clave del método de Burstall es que todo programa recursivo (o especificación ecuacional) satisface una relación de recurrencia que puede ser usada como invariante para diseñar (derivar) una iteración. La iteración se diseña siguiendo las mismas pautas que en derivación formal de iteraciones (Tema 6 del programa de la asignatura), pero aquí se cuenta además de con el invariante, con la solución recursiva. El disponer de dicha solución recursiva facilita mucho el diseño de la iteración, haciendo uso para ello de la técnica de desplegado-plegado.

Para presentar el método de Burstall se considerará que se desea realizar una transformación del siguiente estilo:



donde {INV} es el invariante, B la condición de permanencia de la iteración,  $\bar{x}$  los datos de entrada e  $\bar{y}$  los resultados.

Los pasos a seguir son los siguientes:

- **Paso 1:** Plantear la relación de recurrencia.

La relación de recurrencia se obtiene de la misma definición recursiva. Se puede decir que la definición recursiva es la relación de recurrencia.

- **Paso 2:** Obtener el invariante de la iteración.

El invariante de la iteración se obtiene generalizando la relación de recurrencia. Al generalizar dicha relación se introducen las variables nuevas que se utilizarán en la iteración. Para llevar a cabo la generalización hay que plantear un ejemplo y ver cómo se va resolviendo el algoritmo recursivo dando pasos de desplegado y plegado. La forma que adquiere la expresión tras cada paso de plegado nos indicará cómo generalizar la relación de recurrencia (sustituyendo las expresiones que van cambiando por variables nuevas).

- **Paso 3:** Inicialización (de las variables nuevas).

La inicialización de las variables nuevas se calcula teniendo en cuenta que al principio (justo antes de entrar en la iteración) ha de cumplirse el invariante. En el invariante aparecen todas las variables nuevas del programa que se está construyendo. Por tanto, ahí se verá qué valor ha de tomar cada variable nueva para que el invariante sea cierto.

- **Paso 4:** Finalización.

En este paso hay que obtener la condición B de permanencia en el while y las instrucciones que irán una vez finalizada la iteración (fuera del while).

- La iteración terminará cuando se llegue al caso simple, por tanto B será *"que no estemos en el caso simple"* o, dicho de otra forma,  $\neg B$  será *"que estemos en el caso simple"*.
- Las instrucciones del final sirven para indicar cuál es el resultado definitivo de la nueva función iterativa. Para conocer el resultado definitivo basta con considerar que las variables nuevas del invariante cumplen el caso simple y aplicar la fórmula correspondiente a dicho caso.

- **Paso 5:** Cuerpo de la iteración.

Para obtener las instrucciones que irán dentro del while hay que partir del invariante y dar un paso de desplegado y uno de plegado considerando que estamos en el caso recursivo. De la expresión que quede tras el plegado se deducirá cómo actualizar las variables nuevas en cada vuelta de la iteración.



### **6.3. APLICACIÓN DEL MÉTODO**

A continuación, se mostrará la aplicación del método en 4 casos distintos:

- A) Funciones recursivas elementales (un caso simple y uno recursivo).
- B) Funciones recursivas con operación no asociativa.
- C) Funciones con más de un caso simple.
- D) Funciones con más de un caso recursivo.

En los ejercicios nos encontraremos también con funciones recursivas en las que aparecen mezcladas las características de los casos B), C) y D):

- Operación no asociativa y más de un caso recursivo. (mezcla de B y D)
- Más de un caso simple y más de un caso recursivo. (mezcla de C y D)

#### **6.3.1. FUNCIONES RECURSIVAS ELEMENTALES (UN CASO SIMPLE Y UNO RECURSIVO)**

La aplicación del método en este caso coincidiría exactamente con lo expuesto en la presentación de los pasos generales (apartado 6.2).

Consideremos la función recursiva *sumar* que, dada una lista de enteros devuelve la suma de los elementos que conforman la lista. Ejemplo:  $\text{sumar}([5, 2, 9]) = 16$ .

En nuestros ejercicios la función recursiva vendrá dada en Haskell:

Función recursiva en Haskell:	
<code>sumar: ([Int]) → Int</code>	
<u>Precondición:</u> <code>{true}</code>	
<code>sumar([]) = 0</code>	← caso básico
<code>sumar(x:s) = x + sumar(s)</code>	← caso recursivo

Que la precondición sea 'true' indica que la lista inicial no tiene que cumplir ninguna condición.

A continuación, se reescribe la función recursiva en ADA\* (donde ADA\* es una mezcla entre ADA y Haskell)

Función recursiva en ADA\*:

**function** sumar\_re(h: [Int]) **return** r: Int **is**

Precondición: {true}

**if** es\_vacia(h) **then** r := 0;

**else** r := primero(h) + sumar\_re(resto(h));

**end if**;

Poscondición: {r = sumar(h)}

En Haskell los parámetros pueden ser representados utilizando patrones, es decir, para indicar qué hacer cuando se tenga una lista vacía, podemos poner [] como parámetro y para indicar qué hacer cuando se tenga una lista que no es vacía, podemos poner x:s (o algo similar) como parámetro. En cambio, en ADA\* los parámetros siempre han de ser variables, no se pueden utilizar expresiones como [] y x:s a modo de parámetro. Por ello, al realizar la traducción de Haskell a ADA\* se ha puesto la variable h como parámetro en vez de [] y x:s. La variable h representa una lista cualquiera y no sabemos si es vacía o no y debido a ello se ha tenido que utilizar la función *es\_vacia* en *sumar\_re* y en el caso en el que h no es vacía, se han utilizado las funciones *primero* y *resto* para acceder a sus elementos.

La función *sumar\_re* es una versión recursiva de la función original *sumar*. Por ello en la poscondición se indica que el resultado r obtenido por la función *sumar\_re* coincide con lo que obtendría la función original *sumar* para la lista de entrada h.

A la hora de dar los pasos del método consideraremos el programa recursivo escrito en ADA\*:

- **Paso 1:** Relación de recurrencia.

La relación de recurrencia es la propia definición recursiva.

$$\text{sumar\_re}(h) = \text{primero}(h) + \text{sumar\_re}(\text{resto}(h))$$

- **Paso 2:** Obtener el invariante.

Para obtener el invariante hay que generalizar la relación de recurrencia. Para ello, se puede desarrollar un ejemplo dando pasos de desplegado y plegado sucesivamente. La expresión que nos quede tras cada plegado nos indicará cómo generalizar la relación de recurrencia.

sumar_re([5, 2, 9]) =	
desplegado →	= primero([5, 2, 9]) + sumar_re(resto([5, 2, 9])) =
plegado →	= 5 + sumar_re([2, 9]) =
desplegado →	= 5 + (primero([2, 9]) + sumar_re(resto([2, 9]))) =
plegado →	= 7 + sumar_re([9]) =
desplegado →	= 7 + (primero([9]) + sumar_re(resto([9]))) =
plegado →	= 16 + sumar_re([]) =
caso básico →	= 16 + 0 =
	= 16

**Desplegar** consiste en sustituir la función *sumar\_re* por su propia definición recursiva.

**Plegar** consiste en realizar las operaciones posibles para normalizar o simplificar la expresión.

En el ejemplo desarrollado vemos que tras cada paso de plegado tenemos una expresión que es una suma entre un número entero y una llamada a la función *sumar\_re* que a su vez lleva como argumento una lista.

Ahora tenemos que generalizar la relación de recurrencia. Para generalizar la relación de recurrencia hay que fijarse en las expresiones obtenidas tras cada plegado. Los elementos o datos que van cambiando en esas expresiones obtenidas en los plegados han de ser sustituidas por **variables nuevas**.

$$\begin{array}{c}
 = 7 + \text{sumar\_re}([9]) = \\
 \underbrace{\quad\quad\quad}_u \quad \underbrace{\quad\quad\quad}_v
 \end{array}$$

Por tanto, el **invariante** de la iteración será la siguiente fórmula:

$$\text{sumar\_re}(\ell) = u + \text{sumar\_re}(v)$$

Conviene constatar que el invariante es realmente una generalización de la relación de recurrencia:

$$\text{sumar\_re}(h) = \underbrace{\text{primero}(h)}_u + \text{sumar\_re}(\underbrace{\text{resto}(h)}_v)$$

obtenida sustituyendo **primero(h)** por **u** y **resto(h)** por **v**.

- **Paso 3:** Inicialización de las variables nuevas.

En este paso tenemos que decidir cómo inicializar las variables nuevas que ha introducido el invariante. Para ello, basta recordar que tras inicializar dichas variables ha de cumplirse el invariante. Por consiguiente, tras inicializar  $u$  y  $v$  tiene que cumplirse la siguiente igualdad:

$$\text{sumar\_re}(h) = u + \text{sumar\_re}(v)$$

Para que la igualdad se cumpla habrá que inicializar  $u$  y  $v$  con los valores que ocupan su lugar en la parte izquierda de la igualdad. Como en el caso de la  $u$  no hay nada en la parte izquierda y estamos sumando números, imaginamos que hay un 0:

$$0 + \text{sumar\_re}(h) = u + \text{sumar\_re}(v)$$

De esa igualdad se deduce que tenemos que inicializar  $u$  con 0 y  $v$  con la lista  $h$ :

$u := 0;$   
 $v := h;$

Se puede comprobar que, efectivamente, el invariante se cumple tras dicha inicialización:

$$\begin{aligned} \text{sumar\_re}(h) &= u + \text{sumar\_re}(v) = \\ &= 0 + \text{sumar\_re}(h) = \\ &= \text{sumar\_re}(h) \end{aligned}$$

- **Paso 4:** Finalización.

En este paso tenemos que decidir cuál es la condición para mantenerse en el while, y qué instrucción hace falta añadir una vez terminado el while.

- Para encontrar la condición B de la iteración lo más sencillo es recordar que  $\neg B$  es "*que estemos en el caso simple*". Es importante tener presente que hay que formular el caso simple con respecto a las nuevas variables. En la función *sumar\_re* el caso simple es *es\_vacia(h)* y en el paso 3 hemos visto que la variable nueva asociada a h es v, por tanto:

$$\neg B \equiv \text{es\_vacía}(v) \text{ y } B \equiv \neg \text{es\_vacía}(v)$$

- Para decidir qué instrucción hace falta ejecutar una vez terminado el while, es decir, para decidir cuál es el resultado definitivo de la función, basta con coger el invariante y considerar que estamos en el caso simple (*es\_vacia(v)*):

$$\begin{aligned} \text{sumar\_re}(h) &= u + \text{sumar\_re}(v) = \\ &= u + 0 = \\ &= u \end{aligned}$$

De ahí se deduce que la instrucción final será la siguiente:

$$r := u;$$

donde r es el resultado que devuelve la función iterativa que estamos obteniendo.

- **Paso 5:** Cuerpo de la iteración.

Derivar el cuerpo de la iteración consiste en calcular cómo actualizar las variables nuevas en cada vuelta del while.

El cuerpo de la iteración corresponde al caso recursivo de la función.

Para deducir las instrucciones que irán en el cuerpo hay que partir del invariante y dar un paso de desplegado y uno de plegado. La nueva expresión que quede tras el plegado nos indicará qué valor asignar a  $u$  y  $v$  para que el invariante se conserve:

Caso recursivo:  $\neg \text{es\_vacía}(v)$

$$\begin{array}{ll}
 \text{sumar\_re}(h) & = u + \text{sumar\_re}(v) = \\
 \text{desplegado} \rightarrow & = u + (\text{primero}(v) + \text{sumar\_re}(\text{resto}(v))) = \\
 \text{plegado} \rightarrow & = \underbrace{(u + \text{primero}(v))}_u + \underbrace{\text{sumar\_re}(\text{resto}(v))}_v
 \end{array}$$

Por tanto, la actualización consistirá en las siguientes asignaciones:

$u := u + \text{primero}(v);$   
 $v := \text{resto}(v);$

En este caso el orden de las asignaciones es importante ya que hay que actualizar  $u$  antes que  $v$  debido a que en la actualización de  $u$  se utiliza  $v$ . Por tanto, la siguiente actualización sería incorrecta:

~~$v := \text{resto}(v);$   
 $u := u + \text{primero}(v);$~~

La función iterativa resultante es la siguiente:

Función iterativa en ADA\*:

**function** sumar\_it(h: [Int]) **return** r: Int **is**

Precondición  $\equiv \{true\}$

u: Int;

v: [Int];

u := 0;

v := h;

**while** {INV}  $\neg$ es\_vacia(v) **loop**

    u := u + primero(v);

    v := resto(v);

**end loop**;

r = u;

Poscondicion  $\equiv \{r = \text{sumar}(h)\}$

Mediante la postcondición indicamos que el resultado r obtenido por la función *sumar\_it* es igual a lo que obtendría la función original *sumar* que estaba escrita en Haskell.

### 6.3.2. FUNCIONES CON OPERACIÓN NO ASOCIATIVA

En el ejemplo anterior cada paso de desplegado generaba un nuevo número y en el paso de plegado ese número se podía unificar con el que ya se tenía de antes porque la suma es asociativa:

sumar_re([5, 2, 9]) =	
desplegado →	= primero([5, 2, 9]) + sumar_re(resto([5, 2, 9])) =
plegado →	= 5 + sumar_re([2, 9]) =
desplegado →	= 5 + (primero([2, 9]) + sumar_re(resto([2, 9]))) =
	= 5 + (2 + sumar_re(resto([2, 9]))) =
por asociatividad de +	= (5 + 2) + sumar_re(resto([2, 9])) =
plegado →	= 7 + sumar_re([9]) =
desplegado →	= 7 + (primero([9]) + sumar_re(resto([9]))) =
	= 7 + (9 + sumar_re(resto([9]))) =
por asociatividad de +	= (7 + 9) + sumar_re(resto([9])) =
plegado →	= 16 + sumar_re([]) =
caso básico →	= 16 + 0 =
	= 16

En el siguiente ejemplo veremos que cuando se tiene una operación no asociativa, ésta ha de ser sustituida por una asociativa.

Sólo se considerará una operación no asociativa, la operación ':' que sirve para añadir un elemento por la izquierda a una lista. Cuando aparezca esta operación habrá que sustituirla por la operación '++', que si es asociativa.

$$([5, 7] ++ [6, 1, 8]) ++ [9, 3] = [5, 7] ++ ([6, 1, 8] ++ [9, 3])$$

$$\text{Pero } 5:(7:(9:[])) \neq (5:7):(9:[])$$

De hecho la expresión  $(5:7):(9:[])$  es una expresión incorrecta.

Consideremos la función recursiva *incr* que, dada una lista de enteros devuelve la lista cuyos elementos son los elementos de la lista inicial incrementados en 1.

Ejemplo:  $\text{incr}([5, 2]) = [6, 3]$ .

La función recursiva viene dada en Haskell:

Función recursiva en Haskell:

$\text{incr}: ([\text{Int}]) \rightarrow [\text{Int}]$

Precondición: {true}

$\text{incr}([]) = []$

← caso básico

$\text{incr}(x:s) = (x + 1):\text{incr}(s)$

← caso recursivo

Que la precondición sea 'true' indica que la lista inicial no tiene que cumplir ninguna condición.



A continuación, se reescribe la función recursiva en ADA\* (donde ADA\* es una mezcla entre ADA y Haskell)

Programa recursivo en ADA*:
<b><u>function</u></b> <i>incr_re</i> (h: integer) <b><u>return</u></b> r: [Int] <b><u>is</u></b> Precondición: {true}  <b><u>if</u></b> <i>es_vacia</i> (h) <b><u>then</u></b> r := []; <b><u>else</u></b> r := (primero(h) + 1) : <i>incr_re</i> (resto(h)); <b><u>end if</u></b> ;  Poscondición: {r = <i>incr</i> (h)}

En Haskell los parámetros pueden ser representados utilizando patrones, es decir, para indicar qué hacer cuando se tenga una lista vacía, podemos poner [] como parámetro y para indicar qué hacer cuando se tenga una lista que no es vacía, podemos poner x:s (o algo similar) como parámetro. En cambio, en ADA\* los parámetros siempre han de ser variables, no se pueden utilizar expresiones como [] y x:s a modo de parámetro. Por ello, al realizar la traducción de Haskell a ADA\* se ha puesto la variable h como parámetro en vez de [] y x:s. La variable h representa una lista cualquiera y no sabemos si es vacía o no y debido a ello se ha tenido que utilizar la función *es\_vacia* en *sumar\_re* y en el caso en el que h no es vacía, se han utilizado las funciones *primero* y *resto* para acceder a sus elementos.

La función *incr\_re* es una versión recursiva de la función original *incr*. Por ello, en la postcondición se indica que el resultado r obtenido por la función *incr\_re* coincide con lo que obtendría la función original *incr* para la lista de entrada h.

A la hora de dar los pasos del método consideraremos el programa recursivo escrito en ADA\*:

- **Paso 1:** Relación de recurrencia  
La relación de recurrencia es la propia definición recursiva.

$$\text{incr\_re}(h) = (\text{primero}(h) + 1) : \text{incr\_re}(\text{resto}(h))$$

- **Paso 2:** Obtener el invariante.

Para obtener el invariante hay que generalizar la relación de recurrencia. Un buen método es desarrollar un ejemplo dando pasos de desplegado y plegado sucesivamente. La expresión que nos quede tras cada plegado nos indicará cómo generalizar la relación de recurrencia.

A continuación, se desarrollará un ejemplo:

$incr\_re([6, -1, 8, 0]) =$	
desplegado $\rightarrow$	$= (\text{primero}([6, -1, 8, 0]) + 1) : incr\_re(\text{resto}([6, -1, 8, 0])) =$
plegado $\rightarrow$	$= 7 : incr\_re([-1, 8, 0]) =$
desplegado $\rightarrow$	$= 7 : ((\text{primero}([-1, 8, 0]) + 1) : incr\_re(\text{resto}([-1, 8, 0]))) =$
plegado $\rightarrow$	$= 7 : (0 : incr\_re([8, 0])) =$
	No se pueden juntar manteniendo la operación ':'
desplegado $\rightarrow$	$= 7 : (0 : ((\text{primero}([8, 0]) + 1) : incr\_re(\text{resto}([8, 0]))) =$
plegado $\rightarrow$	$= 7 : (0 : (9 : incr\_re([0]))) =$
	No se pueden juntar manteniendo la operación ':'
desplegado $\rightarrow$	$= 7 : (0 : (9 : ((\text{primero}([0]) + 1) : incr\_re(\text{resto}([0]))))) =$
plegado $\rightarrow$	$= 7 : (0 : (9 : (1 : incr\_re([])))) =$
	No se pueden juntar manteniendo la operación ':'
caso básico $\rightarrow$	$= 7 : (0 : (9 : (1 : []))) =$
	$= [7, 0, 9, 1]$

Al no ser la operación ':' asociativa, en los pasos de plegado se han obtenido expresiones con distinta estructura y no admiten una generalización. La solución es utilizar la operación '++' en vez de ':' y para ello cuando aparezca ':' lo sustituiremos por '++' al dar el paso de plegado:

$incr\_re([6, -1, 8, 0]) =$	
desplegado $\rightarrow$	$= (\text{primero}([6, -1, 8, 0]) + 1) : incr\_re(\text{resto}([6, -1, 8, 0])) =$
plegado $\rightarrow$	$= [7] ++ incr\_re([-1, 8, 0]) =$
desplegado $\rightarrow$	$= [7] ++ ((\text{primero}([-1, 8, 0]) + 1) : incr\_re(\text{resto}([-1, 8, 0]))) =$
plegado $\rightarrow$	$= [7, 0] ++ incr\_re([8, 0]) =$
desplegado $\rightarrow$	$= [7, 0] ++ ((\text{primero}([8, 0]) + 1) : incr\_re(\text{resto}([8, 0]))) =$
plegado $\rightarrow$	$= [7, 0, 9] ++ incr\_re([0]) =$
desplegado $\rightarrow$	$= [7, 0, 9] ++ ((\text{primero}([0]) + 1) : incr\_re(\text{resto}([0]))) =$
plegado $\rightarrow$	$= [7, 0, 9, 1] ++ incr\_re([]) =$
caso básico $\rightarrow$	$= [7, 0, 9, 1] ++ [] =$
	$= [7, 0, 9, 1]$

**Desplegar** consiste en sustituir la función *incr\_re* por su propia definición recursiva.

**Plegar** consiste en realizar las operaciones posibles para normalizar o simplificar la expresión.

En este caso en el paso de plegado hay que eliminar la operación ':' y utilizar '++'.

En el ejemplo desarrollado vemos que tras cada paso de plegado tenemos una expresión donde aparece una lista concatenada a *incr\_re*, que a su vez lleva como argumento otra lista. Además, se puede observar que en cada caso las listas van cambiando.

Generalizar la relación de recurrencia consiste en sustituir por **variables nuevas** aquellos elementos que cambian tras cada desplegado/plegado.

$$= \underbrace{[7, 0]}_u ++ \underbrace{\text{incr\_re}([8, 0])}_v =$$

Por tanto, el **invariante** de la iteración será la siguiente fórmula:

$$\text{incr\_re}(h) = u ++ \text{incr\_re}(v)$$

Conviene constatar que el invariante es realmente una generalización de la relación de recurrencia:

$$\text{incr\_re}(h) = \underbrace{(\text{primero}(h) + 1)}_u : \underbrace{\text{incr\_re}(\text{resto}(h))}_v$$

obtenida sustituyendo  $(\text{primero}(h) + 1) :$  por  $u ++$  y  $\text{resto}(h)$  por  $v$ .

- **Paso 3:** Inicialización de las variables nuevas.

En este paso tenemos que decidir cómo inicializar las variables nuevas que ha introducido el invariante. Para ello, basta recordar que tras inicializar dichas variables ha de cumplirse el invariante. Por tanto, tras inicializar  $u$  y  $v$  la siguiente igualdad ha de ser cierta:

$$\text{incr\_re}(h) = u ++ \text{incr\_re}(v)$$

Para que la igualdad se cumpla habrá que inicializar  $u$  y  $v$  con los valores que ocupan su lugar en la parte izquierda de la igualdad. Como en el caso de la  $u$  no hay nada en la parte izquierda y estamos concatenando listas, imaginamos que hay una lista vacía, es decir,  $[]$ :

$$\begin{array}{c} \downarrow \\ [] ++ \text{incr\_re}(h) = u ++ \text{incr\_re}(v) \\ \uparrow \end{array}$$

De esa igualdad se deduce que tenemos que inicializar  $u$  con  $[]$  y  $v$  con la lista  $h$ :

$$\begin{array}{l} u := []; \\ v := h; \end{array}$$

Se puede comprobar que, efectivamente, el invariante se cumple tras dicha inicialización:

$$\begin{aligned} \text{incr\_re}(h) &= u ++ \text{incr\_re}(v) = \\ &= [] ++ \text{incr\_re}(\ell) = \\ &= \text{incr\_re}(h) \end{aligned}$$

- **Paso 4:** Finalización.

En este paso tenemos que decidir cuál es la condición para mantenerse en el while, y qué instrucción hace falta añadir una vez terminado el while.

- Para encontrar la condición B de la iteración lo más sencillo es recordar que  $\neg B$  es "*que estemos en el caso simple*". Es importante insistir en que hay que formular el caso simple con respecto a las nuevas variables:

$$\begin{aligned} \neg B &\equiv \text{es\_vacía}(v) \text{ y, por tanto, } B \equiv \neg(\text{es\_vacía}(v)), \text{ es decir,} \\ &\quad \mathbf{B \equiv \text{not} (\text{es\_vacía}(v))} \end{aligned}$$

- Para decidir qué instrucción hace falta ejecutar una vez terminado el while, es decir, para averiguar cuál es el resultado definitivo de la función, basta con formular el invariante y considerar que estamos en el caso simple ( $\text{es\_vacía}(v)$ ):

$$\begin{aligned} \text{incr\_re}(h) &= u ++ \text{incr\_re}(v) = \\ &= u ++ [] \\ &= u \end{aligned}$$

Por tanto, la instrucción final será:

$$\mathbf{r := u;}$$

donde r es el resultado que devuelve la función iterativa que estamos obteniendo.

- **Paso 5:** Cuerpo de la iteración.

Derivar el cuerpo de la iteración consiste en decidir cómo actualizar las variables nuevas en cada vuelta del while.

El cuerpo de la iteración corresponde al caso recursivo de la función.

Para deducir las instrucciones que irán en el cuerpo hay que partir del invariante y dar un paso de desplegado y uno de plegado. La nueva expresión que quede tras el plegado nos indicará qué valor asignar a u y v para que el invariante se conserve:

Caso recursivo:  $\neg \text{es\_vacía}(v)$

<b>incr_re(h)</b>	= <b>u ++ incr_re(v)</b> =
desplegado $\rightarrow$	= <b>u ++ ((primero(v) + 1) : incr_re(resto(v)))</b>
plegado $\rightarrow$	= <b>(u ++ [(primero(v) + 1)]) ++ incr_re(resto(v))</b>
	<div style="display: flex; justify-content: space-around; width: 100%;"> <span style="text-align: center;">u</span> <span style="text-align: center;">v</span> </div>

Por consiguiente, la actualización consistirá en las siguientes asignaciones:

```
u := u ++ [(primero(v) + 1)];
v := resto(v);
```

En este caso el orden de las asignaciones es importante ya que hay que actualizar *u* antes que *v* debido a que en la actualización de *u* se utiliza *v*. Por tanto la siguiente actualización sería incorrecta:

```
v := resto(v);
u := u ++ [(primero(v) + 1)];
```

La función iterativa resultante es la siguiente:

Función iterativa en ADA*:
<b><u>function</u></b> incr_it(h: [Int]) <b><u>return</u></b> r: [Int] <b><u>is</u></b>  Precondición $\equiv \{\text{true}\}$  u, v: [Int];  u := []; v := h; <b><u>while</u></b> {INV} <b><u>not</u></b> es_vacia(v) <b><u>loop</u></b> u := u ++ [(primero(v) + 1)]; v := resto(v); <b><u>end loop</u></b> ; r = u; Postcondición $\equiv \{r = \text{incr}(h)\}$

Mediante la postcondición indicamos que el resultado *r* obtenido por la función *incr\_it* es igual a lo que obtendría la función original *incr* que estaba escrita en Haskell.

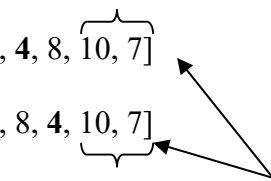
### 6.3.3. FUNCIONES CON MÁS DE UN CASO SIMPLE

La existencia de más de un caso simple afectará al paso 4, tanto en la condición de permanencia del while como en lo que respecta a la determinación del resultado final.

Consideremos la función recursiva que, dadas dos listas de enteros, devuelve la lista que se obtiene intercalando los elementos de las dos listas. Si una de las listas tiene más elementos que la otra, los elementos que quedan sin poder intercalar irán al final de la nueva lista.

Ejemplo:

$\text{intercalar}([2, 4], [5, 8, 10, 7]) = [2, 5, 4, 8, 10, 7]$   
 $\text{intercalar}([5, 8, 10, 7], [2, 4]) = [5, 2, 8, 4, 10, 7]$



elementos que han quedado sin intercalar por contener una de las listas más elementos que la otra

La función recursiva viene dada en Haskell:

Especificación ecuacional o función recursiva en Haskell:

$\text{intercalar} : ([\text{Int}], [\text{Int}]) \rightarrow [\text{Int}]$

Precondición: {true}

$\text{intercalar}([], s) = s$

$\text{intercalar}(x : r, s)$

    |  $\text{es\_vacía}(s)$                      $= x : r$

    |  $\text{otherwise}$                      $= [x] ++ [\text{primero}(s)] ++ \text{intercalar}(r, \text{resto}(s))$

Que la precondición sea 'true' indica que las listas iniciales no tienen que cumplir ninguna condición.

A continuación se reescribe la función recursiva en ADA\* (donde ADA\* es una mezcla entre ADA y Haskell):

Función recursiva en ADA\*:

**function**  $\text{intercalar\_re}(h, s : [\text{Int}])$  **return**  $q : [\text{Int}]$  **is**

Precondición  $\equiv \{\text{true}\}$

**if**  $\text{es\_vacía}(h)$  **then**  $q := s$ ;

**elsif**  $\text{es\_vacía}(s)$  **then**  $q := h$ ;

**else**  $q := [\text{primero}(h)] ++ [\text{primero}(s)] ++ \text{intercalar\_re}(\text{resto}(h), \text{resto}(s))$ ;

**end if**;

Postcondición  $\equiv \{q = \text{intercalar}(h, s)\}$

En Haskell los parámetros pueden ser representados utilizando patrones, es decir, para indicar qué hacer cuando se tenga una lista vacía, podemos poner `[]` como parámetro y para indicar qué hacer cuando se tenga una lista que no es vacía, podemos poner `x:r` (o algo similar) como parámetro. En cambio, en ADA\* los parámetros siempre han de ser variables, no se pueden utilizar expresiones como `[]` y `x:r` a modo de parámetro. Por ello, al realizar la traducción de Haskell a ADA\* se ha puesto la variable `h` como parámetro en vez de `[]` y `x:r`. La variable `h` representa una lista cualquiera y no sabemos si es vacía o no y debido a ello se ha tenido que utilizar la función *es\_vacia* en *intercalar\_re* y en el caso en el que `h` y `s` no sean vacías, se han utilizado las funciones *primero* y *resto* para acceder a sus elementos.

Mediante la postcondición se indica que el resultado `q` que obtiene la función *intercalar\_re* es igual al resultado que obtendría la función *intercalar* para las listas `h` y `s`. La poscondición es necesaria porque la función *intercalar\_re* es una versión de *intercalar*, pero no es la misma, por eso hay que decir que hacen lo mismo.

A la hora de dar los pasos del método consideraremos el programa recursivo escrito en ADA\*:

- **Paso 1:** Relación de recurrencia.

La relación de recurrencia es la propia definición recursiva.

```
intercalar_re(h, s) = [primero(h)] ++ [primero(s)]
                    ++ intercalar_re(resto(h), resto(s))
```

- **Paso 2:** Obtener el invariante.

Para obtener el invariante hay que generalizar la relación de recurrencia. Desarrollar un ejemplo, dando pasos de desplegado y plegado sucesivamente, puede aportar la información necesaria. La expresión que nos quede tras cada plegado nos indicará cómo generalizar la relación de recurrencia.

	<code>intercalar_re([8, 0], [3, 9, 4, 7]) =</code>
desplegado →	<code>= primero([8, 0]) ++ primero([3, 9, 4, 7]) ++</code> <code>intercalar_re(resto([8, 0]), resto([3, 9, 4, 7])) =</code>
plegado →	<code>= [8, 3] ++ intercalar_re([0], [9, 4, 7]) =</code>
desplegado →	<code>= [8, 3] ++ (primero([0]) ++ primero([9, 4, 7]) ++</code> <code>intercalar_re(resto([0]), resto([9, 4, 7]))) =</code>
plegado →	<code>= [8, 3, 0, 9] ++ intercalar_re([], [4, 7]) =</code>
caso básico →	<code>= [8, 3, 0, 9] ++ [4, 7] =</code> <code>= [8, 3, 0, 9, 4, 7]</code>

**Desplegar** consiste en sustituir la función *intercalar\_re* por su propia definición recursiva.

**Plegar** consiste en realizar las operaciones posibles para volver al formato original o para simplificar la expresión.

En el ejemplo desarrollado vemos que tras cada paso de plegado tenemos una expresión donde aparece una lista concatenada a *intercalar\_re*. Dicha lista y los dos argumentos de *intercalar\_re* cambian con cada desplegado/plegado.

Generalizar la relación de recurrencia consiste en sustituir por **variables nuevas** aquellos elementos que cambian tras cada plegado:

$$= \underbrace{[8, 3, 0, 9]}_u ++ \text{intercalar\_re}(\underbrace{[]}_v, \underbrace{[4, 7]}_w) =$$

Por tanto, el **invariante** de la iteración será la siguiente fórmula:

$$\text{intercalar\_re}(h, s) = u ++ \text{intercalar\_re}(v, w)$$

Conviene constatar que el invariante es realmente una generalización de la relación de recurrencia:

$$\text{intercalar\_re}(h, s) = \underbrace{[primero(h)] ++ [primero(s)]}_u ++ \text{intercalar\_re}(\underbrace{\text{resto}(h)}_v, \underbrace{\text{resto}(s)}_w)$$

obtenida sustituyendo **[primero(h)] ++ [primero(s)]** por **u**, **resto(h)** por **v** y **resto(s)** por **w**.

- **Paso 3:** Inicialización de las variables nuevas.

En este paso tenemos que calcular cómo inicializar las variables nuevas que ha introducido el invariante. Para ello, basta recordar que tras inicializar dichas variables ha de cumplirse el invariante. Por tanto, tras inicializar **u**, **v** y **w** tiene que cumplirse la siguiente ecuación que es el invariante:

$$\text{intercalar\_re}(h, s) = u ++ \text{intercalar\_re}(v, w)$$

Para que la igualdad se cumpla habrá que inicializar **u**, **v** y **w** con los valores que ocupan su lugar en la parte izquierda de la igualdad. Como en el caso de la **u** no hay nada en la parte izquierda y estamos concatenando listas, imaginamos que hay una lista vacía, es decir, **[]**:

$$\boxed{[]} ++ \text{intercalar\_re}(h, s) = u ++ \text{intercalar\_re}(v, w)$$



De esa igualdad se deduce que tenemos que inicializar  $u$  con  $[]$ ,  $v$  con la lista  $h$  y  $w$  con la lista  $s$ :

$$\begin{aligned}u &:= []; \\ v &:= h; \\ w &:= s;\end{aligned}$$

Se puede comprobar que, efectivamente, el invariante se cumple tras dicha inicialización:

$$\begin{aligned}\text{intercalar\_re}(h, s) &= u ++ \text{intercalar\_re}(v, w) = \\ &= [] ++ \text{intercalar\_re}(h, s) = \\ &= \text{intercalar\_re}(h, s)\end{aligned}$$

- **Paso 4:** Finalización.

En este paso tenemos que decidir cuál es la condición para mantenerse en el while, y qué instrucción hace falta añadir una vez terminado el while.

- Para encontrar la condición  $B$  de la iteración lo más sencillo es recordar que  $\neg B$  es "*que estemos en un caso simple*". Es importante incidir en que los casos simples han de formularse en función de las nuevas variables. Al haber dos casos simples  $\neg B$  será la **disyunción de los dos casos simples**:

$$\neg B \equiv \text{es\_vacía}(v) \vee \text{es\_vacía}(w)$$

y, por tanto,

$$B \equiv \neg(\text{es\_vacía}(v) \vee \text{es\_vacía}(w)),$$

es decir,  $B \equiv \underline{\text{not}}(\text{es\_vacía}(v) \text{ or } \text{es\_vacía}(w))$ , que también podría escribirse como  $\underline{\text{not}}(\text{es\_vacía}(v)) \text{ and } \underline{\text{not}}(\text{es\_vacía}(w))$ .

- Para decidir cuál es el resultado final, habrá que considerar los dos casos simples por separado y ello originará la necesidad de una instrucción condicional:

✓ Primer caso simple:  $\text{es\_vacía}(v)$

$$\begin{aligned}\text{intercalar\_re}(h, r) &= u ++ \text{intercalar\_re}(v, w) = \\ &= u ++ w\end{aligned}$$

Por tanto, en este caso la instrucción final será:

$$q := u ++ w;$$

donde  $q$  es el resultado que devuelve la función iterativa que estamos construyendo.

✓ Segundo caso simple: es\_vacia (w)

$$\begin{aligned} \text{intercalar\_re}(h, s) &= u ++ \text{intercalar\_re}(v, w) = \\ &= u ++ v \end{aligned}$$

De ahí se infiere que la instrucción final para este caso será:

$$q := u ++ v;$$

donde q es el resultado que devuelve la función iterativa que estamos construyendo.

En definitiva, tendremos la siguiente instrucción condicional una vez terminado el while:

```
if es_vacia(v) then q := u ++ w;
else q := u ++ v;
end if;
```

- **Paso 5:** Cuerpo de la iteración

Al derivar el cuerpo de la iteración se ha de decidir en qué forma actualizar las variables nuevas en cada vuelta del while.

El cuerpo de la iteración corresponde al caso recursivo de la función.

Para deducir las instrucciones que irán dentro del while hay que partir del invariante y dar un paso de desplegado y uno de plegado. La nueva expresión que quede tras el plegado nos indicará qué valor asignar a u, v y w para que el invariante se conserve:

Caso recursivo:  $\neg(\text{es\_vacía}(v) \vee \text{es\_vacía}(w))$

$$\begin{aligned} \text{intercalar\_re}(h, s) &= u ++ \text{intercalar\_re}(v, w) = \\ \text{desplegado} \rightarrow &= u ++ ([\text{primero}(v)] ++ [\text{primero}(w)] \\ &\quad ++ \text{intercalar\_re}(\text{resto}(v), \text{resto}(w))) = \end{aligned}$$

$$\begin{aligned} \text{plegado} \rightarrow &= \overbrace{(u ++ [\text{primero}(v)] ++ [\text{primero}(w)])}^u \\ &\quad ++ \text{intercalar\_re}(\underbrace{\text{resto}(v)}_v, \underbrace{\text{resto}(w)}_w) \end{aligned}$$

Por consiguiente, la actualización consistirá en las siguientes asignaciones:

```
u := u ++ [primero(v)] ++ [primero(w)];
v := resto(v);
w := resto(w);
```

El orden de las asignaciones es importante ya que hay que actualizar u antes que v y w debido a que en la actualización de u se utilizan v y w. En cambio, en cuanto a v y w, el orden de actualización no es importante, es decir, la siguiente actualización es correcta también:

```
u := u ++ [primero(v)] ++ [primero(w)];
w := resto(w);
v := resto(v);
```

Pero la siguiente es incorrecta:

```
v := resto(v);
u := u ++ [primero(v)] ++ [primero(w)];
w := resto(w);
```

A continuación, se muestra la función iterativa resultante:

Función iterativa en ADA\*:

**function** intercalar\_it (h, s: [Int]) **return** q: [Int] **is**

Precondición  $\equiv \{\text{true}\}$

u, v, w: [Int];

u := [];

v := h;

w := s;

**while** {INV} **not** (es\_vacia(v) **or** es\_vacia(w)) **loop**

    u := u ++ [primero(v)] ++ [primero(w)];

    v := resto(v);

    w := resto(w);

**end loop**;

**if** es\_vacia(v) **then** q := u ++ w;

**else** q := u ++ v;

**end if**;

Poscondición  $\equiv \{q = \text{intercalar}(h, s)\}$

Mediante la postcondición indicamos que el resultado q obtenido por la función *intercalar\_it* es igual a lo que obtendría la función original *intercalar* que estaba escrita en Haskell.

### 6.3.4. FUNCIONES CON MÁS DE UN CASO RECURSIVO

La existencia de más de un caso recursivo afectará a los pasos 1, 2 y 5, es decir, a la relación de recurrencia, al cálculo del invariante y al cuerpo de la iteración:

- Habrá tantas relaciones de recurrencia como casos recursivos.
- El invariante tendrá que ser una generalización de todas las relaciones de recurrencia. Sólo habrá un invariante.
- Habrá que analizar cómo actualizar las variables nuevas en cada caso recursivo y ello hará que el cuerpo de la iteración sea una sentencia condicional con tantas ramas como casos recursivos.

Consideremos la función recursiva *divpart* que, dados un número entero  $x$  y una lista de enteros que no contiene el 0, obtiene una lista nueva en la cual todos los elementos de la lista que dividen a  $x$  están al principio y todos los elementos que no dividen a  $x$  al final.

Ejemplo:

$\text{divpart}(20, [3, 8, 10, 9, 4, 7]) = [10, 4, 7, 9, 8]$

Es conveniente fijarse en que en la nueva lista primero están todos los que dividen a 20 en el mismo orden que en la lista inicial y luego vienen los que no dividen a 20 pero en orden inverso.

La función recursiva viene dada en Haskell:

Especificación en Haskell:		
$\text{divpart}: (\text{Int}, [\text{Int}]) \rightarrow [\text{Int}]$		
<u>Precondición:</u> $\{\neg \text{esta}(0, s)\}$		
$\text{divpart}(x, s)$		
$\text{es\_vacía}(s)$		$= []$
$x \bmod \text{primero}(s) == 0$		$= [\text{primero}(s)] ++ \text{divpart}(x, \text{resto}(s))$
$x \bmod \text{primero}(s) \neq 0$		$= \text{divpart}(x, \text{resto}(s)) ++ [\text{primero}(s)]$

En este caso para distinguir entre lista vacía y no vacía no se han utilizado los patrones  $[]$  y  $z:r$  de manera explícita. Se le ha llamado  $s$  a la lista y luego para saber si la lista es vacía y para acceder a los elementos se han utilizado las funciones *es\_vacia*, *primero* y *resto*. Haskell admite las dos variantes y en los enunciados a veces la función vendrá dada utilizando como parámetros los patrones  $[]$  y  $z:r$  y en otros casos el parámetro vendrá representado por una variable.

Mediante la precondición ' $\neg \text{esta}(0, s)$ ' se indica que la lista de entrada no contendrá ningún 0.

A continuación se reescribe la función recursiva en ADA\* (donde ADA\* es una mezcla entre ADA y Haskell):

<p>Función recursiva en ADA*:</p> <pre> <b>function</b> divpart_re(x: Int, s: [Int]) <b>return</b> h: [Int] <b>is</b> Precondición <math>\equiv \{\neg \text{esta}(0, s)\}</math> <b>if</b> es_vacia(s) <b>then</b> h := []; <b>elsif</b> x <b>mod</b> primero(s) = 0 <b>then</b> h := [primero(s)] ++ divpart_re(x, resto(s)); <b>else</b> h := divpart_re(x, resto(s)) ++ [primero(s)]; <b>end if</b>; {h = divpart(x, s)} </pre>
---

En este caso la traducción de la función de Haskell a ADA\* ha sido muy directa porque en la función escrita en Haskell la lista de entrada aparecía directamente como s y no como [] y z:r.

A la hora de dar los pasos del método consideraremos el programa recursivo escrito en ADA\*:

- **Paso 1:** Relación de recurrencia.  
La relación de recurrencia es la propia definición recursiva. En este caso hay dos relaciones de recurrencia, una por cada caso recursivo.

x mod primero(s) = 0:

$$\text{divpart\_re}(x, s) = [\text{primero}(s)] ++ \text{divpart\_re}(x, \text{resto}(s))$$

x mod primero(s)  $\neq$  0:

$$\text{divpart\_re}(x, s) = \text{divpart\_re}(x, \text{resto}(s)) ++ [\text{primero}(s)]$$

- **Paso 2:** Obtener el invariante.

Para obtener el invariante hay que generalizar las dos relaciones de recurrencia. Para generalizar las relaciones de recurrencia se desarrollará un ejemplo dando pasos de desplegado y plegado sucesivamente. La expresión que nos quede tras cada plegado nos indicará cómo generalizar las relaciones de recurrencia.

```
divpart_re(20, [8, 10, 9, 4, 7]) =
desplegado →      = divpart_re(20, resto([8, 10, 9, 4, 7])) ++ [primero([8, 10, 9, 4, 7])] =
plegado →          = divpart_re(20, [10, 9, 4, 7]) ++ [8] =
desplegado →      = ([primero([10, 9, 4, 7])] ++ divpart_re(20, resto([10, 9, 4, 7]))) ++ [8] =
plegado →          = [10] ++ divpart_re(20, [9, 4, 7]) ++ [8] =
desplegado →      = [10] ++ (divpart_re(20, resto([9, 4, 7])) ++ [primero([9, 4, 7])]) ++ [8] =
plegado →          = [10] ++ divpart_re(20, [4, 7]) ++ [9, 8] =
desplegado →      = [10] ++ ([primero([4, 7])] ++ divpart_re(20, resto([4, 7]))) ++ [9, 8] =
plegado →          = [10, 4] ++ divpart_re(20, [7]) ++ [9, 8] =
desplegado →      = [10, 4] ++ (divpart_re(20, resto([7])) ++ [primero([7])]) ++ [9, 8] =
plegado →          = [10, 4] ++ divpart_re(20, []) ++ [7, 9, 8] =
caso simple →      = [10, 4] ++ [] ++ [7, 9, 8] =
                  = [10, 4, 7, 9, 8]
```

El **desplegado** se realiza reemplazando la función *divpart\_re* por su propia definición recursiva.

En el **plegado** se llevan a cabo las operaciones necesarias para que la expresión vuelva a tener la forma original, es decir, para simplificarla.

En el ejemplo desarrollado vemos que tras cada paso de plegado tenemos una expresión donde la llamada recursiva aparece concatenada a una lista por la izquierda y a otra por la derecha. Las dos listas concatenadas y la lista argumento de *divpart\_re* son las que van cambiando.

Generalizar la relación de recurrencia supondrá sustituir por **variables nuevas** aquellos elementos que van cambiando:

$$= \underbrace{[10, 4]}_u ++ \underbrace{\text{divpart\_re}(20, [])}_v ++ \underbrace{[7, 9, 8]}_w =$$

Por tanto, el **invariante** de la iteración será la siguiente fórmula:

$$\text{divpart\_re}(x, s) = u ++ \text{divpart\_re}(x, v) ++ w$$

Como el primer argumento de *divpart\_re* no cambia, no hay que sustituirlo por una variable.

Conviene constatar que el invariante es realmente una generalización de las dos relaciones de recurrencia:

$x \bmod \text{primero}(s) = 0$ :

$$\text{divpart\_re}(x, s) = \underbrace{[\text{primero}(s)]}_{u} ++ \text{divpart\_re}(x, \underbrace{\text{resto}(s)}_v) ++ \underbrace{[]}_w$$

$x \bmod \text{primero}(s) \neq 0$ :

$$\text{divpart\_re}(x, s) = \underbrace{[]}_u ++ \text{divpart\_re}(x, \underbrace{\text{resto}(s)}_v) ++ \underbrace{[\text{primero}(s)]}_w$$

El invariante se ha obtenido sustituyendo  $[\text{primero}(s)]$  y  $[]$  por  $u$ ,  $\text{resto}(s)$  por  $v$  y  $[]$  y  $[\text{primero}(s)]$  por  $w$ . En el caso de las listas vacías añadidas en las relaciones de recurrencia, son listas que uno puede suponer que están ahí para que las dos relaciones de recurrencia tengan el mismo formato: lista ++ función ++ lista.

- **Paso 3:** Inicialización de las variables nuevas.

En este paso se ha de inferir cómo inicializar las variables nuevas que ha introducido el invariante. Para ello es suficiente tener en cuenta que tras inicializar dichas variables ha de satisfacerse el invariante. Esto es, tras inicializar  $u$ ,  $v$  y  $w$  tiene que cumplirse la siguiente fórmula, que es el invariante:

$$\text{divpart\_re}(x, s) = u ++ \text{divpart\_re}(x, v) ++ w$$

Para que la igualdad se cumpla habrá que inicializar  $u$ ,  $v$  y  $w$  con los valores que ocupan su lugar en la parte izquierda de la igualdad. Como en el caso de la  $u$  y la  $w$  no hay nada en la parte izquierda y estamos concatenando listas, imaginamos que hay listas vacías:

$$\underbrace{[]} ++ \text{divpart\_re}(x, s) ++ \underbrace{[]} = u ++ \text{divpart\_re}(x, v) ++ w$$

De esa igualdad se deduce que tenemos que inicializar  $u$  con  $[]$ ,  $v$  con la lista  $s$  y  $w$  con  $[]$ :

```
u := [];
v := s;
w := [];
```

Se puede comprobar que, efectivamente, el invariante se cumple tras dicha inicialización:

$$\begin{aligned}\text{divpart\_re}(x, s) &= u ++ \text{divpart\_re}(x, v) ++ w = \\ &= [] ++ \text{divpart\_re}(x, s) ++ [] = \\ &= \text{divpart\_re}(x, s)\end{aligned}$$

- **Paso 4: Finalización.**

En este paso tenemos que fijar la condición para mantenerse en el while y la instrucción a ejecutar una vez terminado el while.

- La condición B de la iteración se obtiene considerando primero que  $\neg B$  es "*que estemos en el caso simple*". El caso simple debe ser adaptado a las nuevas variables:

$$\begin{aligned}\neg B &\equiv \text{es\_vacía}(v) \\ &\text{y por tanto,} \\ B &\equiv \neg \text{es\_vacía}(v),\end{aligned}$$

- Para decidir cuál es el resultado final, habrá que retomar el invariante y considerar que se cumple el caso simple, es decir, que  $v$  es una lista vacía:

$$\begin{aligned}\text{divpart\_re}(x, s) &= u ++ \text{divpart\_re}(x, v) ++ w = \\ &= u ++ \text{divpart\_re}(x, []) ++ w = \\ &= u ++ w\end{aligned}$$

De ahí se deduce que la sentencia final será la siguiente:

$$\mathbf{h := u ++ w;}$$

donde  $h$  es el resultado a devolver por el programa iterativo que estamos construyendo.



- **Paso 5:** Cuerpo de la iteración.

En el cuerpo de la iteración se indica cómo actualizar las variables nuevas en cada vuelta del while.

El cuerpo de la iteración corresponde a los casos recursivos de la función.

Al haber dos casos recursivos, habrá que considerarlos por separado. Primero se cogerá el invariante y se aplicará un paso de desplegado/plegado suponiendo que estamos en el primer caso recursivo. De ahí obtendremos la manera de actualizar las variables nuevas cuando estemos en ese caso. A continuación, se procederá de la misma forma con la segunda alternativa. El cuerpo de la iteración será un if con dos ramas.

➤ Primer caso recursivo:  $x \bmod \text{primero}(v) = 0$

$$\begin{aligned}
 \text{divpart\_re}(x, s) &= u \text{ ++ } \text{divpart\_re}(x, v) \text{ ++ } w = \\
 \text{desplegado} \rightarrow &= u \text{ ++ } ([\text{primero}(v)] \text{ ++ } \\
 &\quad \text{++ divpart\_re}(x, \text{resto}(v))) \text{ ++ } w = \\
 \text{plegado} \rightarrow &= \underbrace{(u \text{ ++ } [\text{primero}(v)])}_u \text{ ++ } \underbrace{\text{divpart\_re}(x, \text{resto}(v))}_v \text{ ++ } \underbrace{w}_w
 \end{aligned}$$

Por consiguiente, la actualización consistirá en las siguientes asignaciones:

```

u := u ++ [primero(v)];
v := resto(v);
w := w;

```

El orden de las asignaciones es importante ya que hay que actualizar u antes que v debido a que en la actualización de u se utiliza v. En cambio, en cuanto a v y w o u y w, el orden de actualización no es importante, es decir, la siguiente actualización es correcta también:

```

w := w;
u := u ++ [primero(v)];
v := resto(v);

```

Pero la siguiente es incorrecta:

```

v := resto(v);
u := u ++ [primero(v)];
w := w;

```

- Segundo caso recursivo:  $x \bmod \text{primero}(v) \neq 0$

$$\begin{aligned}
 \text{divpart\_re}(x, s) &= u ++ \text{divpart\_re}(x, v) ++ w = \\
 \text{desplegado} \rightarrow &= u ++ (\text{divpart\_re}(x, \text{resto}(v)) ++ [\text{primero}(v)]) ++ w = \\
 \text{plegado} \rightarrow &= \underbrace{u}_{u} ++ \text{divpart\_re}(x, \underbrace{\text{resto}(v)}_v) ++ (\underbrace{[\text{primero}(v)] ++ w}_w)
 \end{aligned}$$

Por consiguiente, la actualización consistirá en las siguientes asignaciones:

```

u := u;
w := [primero(v)] ++ w;
v := resto(v);

```

El orden de las asignaciones es importante ya que hay que actualizar  $w$  antes que  $v$  debido a que en la actualización de  $w$  se utiliza  $v$ .

La consideración conjunta de los dos casos recursivos conduce al planteamiento de la siguiente sentencia condicional:

```

if  $x \bmod \text{primero}(s) = 0$  then  $u := u ++ [\text{primero}(v)];$ 
                                 $v := \text{resto}(v);$ 
                                 $w := w;$ 
else  $u := u;$ 
       $w := [\text{primero}(v)] ++ w;$ 
       $v := \text{resto}(v);$ 
end if;

```

Esa sentencia puede ser simplificada eliminando las asignaciones  $w := w$  y  $u := u$  y colocando la asignación  $v := \text{resto}(v)$  fuera de la condición, pero lo dejaremos así por claridad.

La función iterativa resultante es la siguiente:

Función iterativa en ADA\*:

**function** divpart\_it(x: Int, s: [Int]) **return** h: [Int] **is**

Precondición  $\equiv \{\neg \text{esta}(0, s)\}$

u, v, w: [Int];

u := [];

v := s;

u := [];

**while** {INV} **not** es\_vacia(v) **loop**

**if** x mod primero(s) = 0 **then**

        u := u ++ [primero(v)];

        v := resto(v);

        w := w;

**else**

        u := u;

        w := [primero(v)] ++ w;

        v := resto(v);

**end if**;

**end loop**;

h := u ++ w;

Poscondición  $\equiv \{h = \text{divpart}(x, s)\}$

Mediante la precondición indicamos que la lista de entrada no ha de contener ceros y mediante la postcondición indicamos que el resultado h obtenido por la función *divpart\_it* es igual a lo que obtendría la función original *divpart* que estaba escrita en Haskell.

#### **6.4. RESUMEN Y CONCLUSIONES**

En este tema se ha presentado el Método de Burstall que sirve para transformar programas recursivos en iterativos y se ha mostrado, mediante el desarrollo de ejemplos, su aplicación a distintos casos:

- A) Funciones recursivas elementales (un caso simple y uno recursivo).
- B) Funciones recursivas con operación no asociativa.
- C) Funciones con más de un caso simple.
- D) Funciones con más de un caso recursivo.

Uno de los objetivos de este tema era reflexionar sobre la relación entre la recursividad y la iteración. Analizando los cuatro casos expuestos en el apartado 7.3 se puede obtener una idea intuitiva de dicha relación:

*Cada llamada recursiva viene a ser una vuelta en el while.*

Esta conclusión es muy superficial y pretende simplemente aportar una primera idea intuitiva sobre la relación entre la recursión y la iteración.