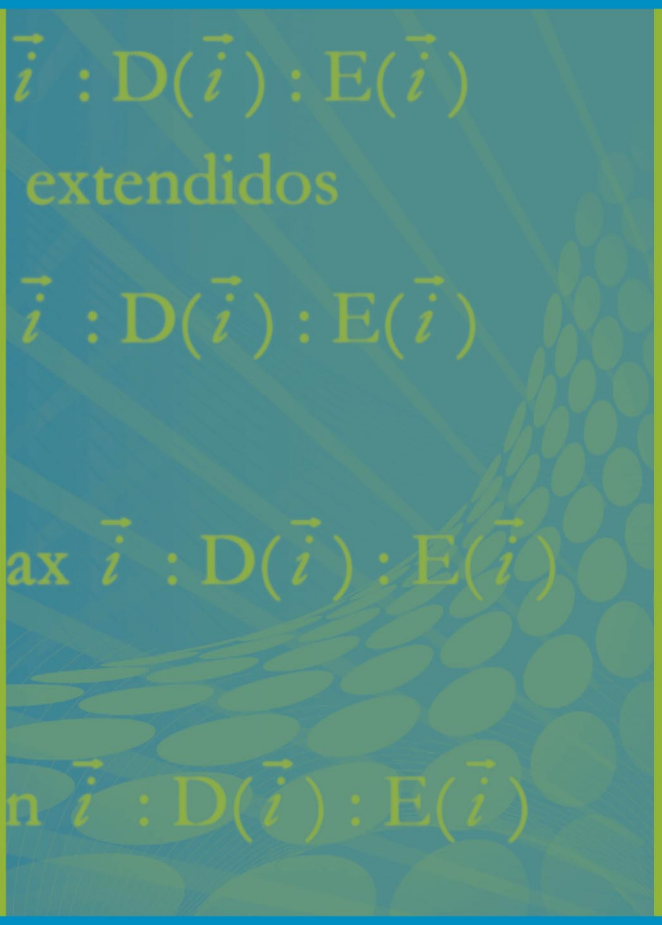


Estructuras de datos

Un enfoque moderno

Mario Rodríguez Artalejo,
Pedro Antonio González Calero y
Marco Antonio Gómez Martín

Extensión

UCM EDITORIAL COMPLUTENSE

Todos los libros publicados por Editorial Complutense a partir de enero de 2007
han superado el proceso de evaluación experta.

Donoso Cortés, 63 – 4.ª planta. 28015 Madrid
Tels.: 91 394 64 60/61. Fax: 91 394 64 58
ecsa@rect.ucm.es
www.edirorialcomplutense.com

Primera edición: Septiembre de 2011

ISBN: 978-84-9938-096-4

Esta editorial es miembro de la UNE, lo que garantiza la difusión y comercialización
de sus publicaciones a nivel nacional e internacional.

PRÓLOGO

Si has aprendido a programar de manera informal, las técnicas formales son un descubrimiento asombroso. Se convierten en una herramienta muy potente para razonar sobre los programas y explorar las alternativas de diseño de algoritmos que cualquier problema no trivial plantea. Por otra parte, si tu primera aproximación a la Programación es a través de las técnicas formales, éstas te servirán para enfocar la atención sobre aspectos concretos del problema, resolverlos por separado y razonar sobre su corrección, adquiriendo una metodología sistemática y bien fundamentada para el diseño de algoritmos que, una vez interiorizada, podrás probablemente olvidar.

Este libro es el resultado de nuestra experiencia impartiendo la asignatura Estructuras de datos y de la información en la Facultad de Informática de la Universidad Complutense de Madrid, ininterrumpidamente desde el curso 1995-96 hasta la actualidad. Pasando cronológicamente por las manos de los tres autores, empezando por las notas de clase de Mario Rodríguez, que Pedro González evolucionó y Marco Gómez siguió utilizando posteriormente. Fruto de este trabajo, además de este libro, se han generado trasparencias de apoyo a las clases e implementaciones en C++ de todos los algoritmos aquí descritos. Es este material adicional, disponible en la página http://gaia.fdi.ucm.es/people/pedro/edem/, junto con el texto del libro el que da sentido al "moderno" que aparece en el título del libro. Existen libros de estructuras de datos con un enfoque formal, alejados de los lenguajes de programación concretos, así como otros menos formales y más preocupados por proporcionar los detalles de implementación en un lenguaje concreto. Este libro pretende ser moderno aproximando ambos enfoques: formal en la presentación de los conceptos, que se acompañan de implementaciones ejecutables en el material adicional.

El libro está organizado de la siguiente forma. En el capítulo 1 se introducen las técnicas de especificación pre/post de algoritmos. Aunque se puede suponer que los estudiantes han seguido previamente un curso de "lógica para informáticos", hemos intentado que la exposición sea autocontenida, introduciendo todos los elementos de sintaxis y semántica de la lógica con signatura heterogénea que se utiliza en la especificación. Se introducen las reglas de verificación de las construcciones habituales de los lenguajes imperativos, así como las que permiten verificar procedimientos y funciones. Se presentan así mismo las medidas asintóticas de la complejidad y los métodos de análisis de la complejidad de algoritmos iterativos. A continuación, se presentan los métodos de derivación de algoritmos iterativos a partir de la especificación, con especial atención a la derivación de bucles a partir del invariante. Por último, se utilizan los métodos de derivación recién presentados para obtener los algoritmos de recorrido, búsqueda secuencial y binaria y métodos de complejidad cuadrática de ordenación de vectores.

El segundo capítulo se dedica al estudio de los algoritmos recursirvos. Siguiendo un esquema similar al del capítulo 1, se presentan consecutivamente las técnicas de especificación, derivación y análisis de algoritmos recursivos. En el apartado dedicado a la derivación se derivan los algoritmos de complejidad cuasi-lineal de ordenación rápida y ordenación por mezcla. Después de presentar el esquema general de transformación de recursión final en iteración, el capítulo concluye presentando algunas técnicas adicionales de diseño de algoritmos recursivos como son la generalización y el plegado-desplegado.

El capítulo 3 da comienzo a la segunda parte del libro, que se dedica al estudio de los tipos abstractos de datos. Después de introducir de manera informal el concepto de tipo abstracto de datos (TADs) y el papel que juegan las estructuras de datos como mecanismos de implementación, se presentan las técnicas de especificación algebraica de tipos abstractos de datos que se emplearán en el resto del libro para especificar los TADs estudiados. A continuación se hacen

algunas consideraciones generales acerca de las alternativas de implementación de un TAD a partir de su especificación, incluyendo indicaciones acerca de cómo verificar la corrección de dicha implementación. Por último, se presentan los punteros como mecanismo para construir estructuras de datos en memoria dinámica, completando así, con tipos simples, registros y vectores que se suponían conocidos, el repertorio de estructuras de datos básicas que se usará en el resto del libro para construir estructuras de datos más elaboradas.

Los capítulos 4, 5, 6 y 7 se dedican al estudio de las estructuras de datos más habituales: estructuras de datos lineales, árboles, tablas y grafos. Entre las estructuras de datos lineales nos ocupamos de especificar y presentar implementaciones alternativas para pilas, colas, colas dobles, listas y secuencias. Las secuencias, que añaden al TAD Lista la posibilidad de recorrer secuencialmente sus elementos, serán muy utilizadas como estructuras auxiliares en los capítulos posteriores. En el capítulo 5, dedicado a los árboles, se especifican e implementan los árboles binarios, los árboles de búsqueda y los árboles equilibrados AVL. Los montículos se desarrollan como ejercicios en este tema. El capítulo 6 se dedica a las tablas, especificadas como funciones de claves en valores, y que se pueden implementar eficientemente con la técnica de las tablas dispersas. Se estudian las tablas dispersas abiertas y cerradas, además de los principales métodos de localización y relocalización. Por último el capítulo 7 se dedica a la especificación e implementación de los grafos. Además de las operaciones básicas de construcción de grafos dirigidos etiquetados, se estudian las operaciones de recorrido en profundidad y anchura, así como los algoritmos de obtención de caminos mínimos.

Queremos expresar nuestro agradecimiento a las sucesivas generaciones de estudiantes de la Facultad de Informática de la Universidad Complutense de Madrid que han seguido la asignatura de Estructuras de datos y de la información y nos han ayudado a mejorar el material a partir del cual hemos preparado el libro que ahora tienes delante.

Mayo de 2011

Mario Rodríguez Artalejo
Pedro Antonio González Calero
Marco Antonio Gómez Martín

ÍNDICE DE CONTENDOS

CAPÍTULO 1
DISEÑO DE ALGORITMOS ITERATIVOS

1.1 Especificación pre/post

Los algoritmos se pueden especificar en lenguaje natural. El problema es que para que las especificaciones así expresadas sean precisas –consideren todos los casos posibles– se necesitan descripciones muy extensas. Las especificaciones formales proporcionan a la vez precisión y concisión. La tercera característica deseable de una especificación es la claridad –y aquí es donde, para un programador no entrenado en las técnicas formales, pueden resultar más ventajosas las especificaciones en lenguaje natural–. Otro problema es que los sistemas informáticos se ocupan de un rango tan amplio de cuestiones que puede resultar pretencioso pretender ser capaces de formalizar cualquier dominio, aparte de que un poco de ambigüedad puede venir bien a veces...

Los formalismos que utilizaremos para construir las especificaciones:

— Especificaciones con *precondiciones* y *postcondiciones* de la lógica de predicados para los algoritmos.

— Especificaciones *algebraicas* mediante ecuaciones para los tipos de datos.

Las especificaciones resultan útiles en las distintas fases del desarrollo de los programas:

— Antes de la construcción, como contratos que facilitan la abstracción y la división de tareas.

— Durante la construcción, porque existen técnicas que nos ayudan a construir programas a partir de las especificaciones.

— Después de la construcción, porque constituyen una documentación excelente.

Una especificación pre/post de un programa (algoritmo o acción) A es

{P} A {Q}

Que se lee como "*P* son las condiciones que ha de cumplir la entrada para que, tras ejecutar *A* se obtenga un resultado que cumpla *Q*". El problema es determinar qué describen las condiciones *P* y *Q*. Un programa lo podemos ver como un proceso que cambia el estado de una computadora [CCMRSV93].

[Peñ05] La técnica pre/post se basa en considerar que un algoritmo, contemplado como una caja negra de la cual sólo nos es posible observar sus parámetros de entrada y de salida, actúa como una función de *estados* en *estados*: comienza su ejecución en un estado inicial válido, descrito por el valor de los parámetros de entrada, y termina en un estado final en el que los parámetros de salida contienen los resultados esperados.

El problema es que los programas se ejecutan en computadoras y hay aspectos del *estado* de una computadora que no son fácilmente formalizables.

La solución que se adopta es definir el

---

estado de un programa como una descripción instantánea de los valores asociados a las variables del programa.

[Bal93] Se puede identificar intuitivamente con un "estado interno" de la máquina en la que se ejecuta el programa, en un instante determinado. ¿Es posible describir el estado de una computadora exclusivamente mediante *variables*? (El contenido de la memoria, por supuesto; el estado de otros dispositivos en última instancia ¿viene siempre dado por algún tipo de memoria –la de vídeo, por ejemplo– o registro?) ¿Está relacionado el uso de este modelo con el hecho de que las técnicas formales obvien la entrada/salida? ¿Qué es un programa?

En {P} A {Q} la *A* se puede referir a un programa entero, con lo que las variables a que hacen referencia *P* y *Q* serían las variables globales –¿después de ser inicializadas?–; o puede tratarse de un procedimiento o función, en general una *acción*, con lo que las variables serían los parámetros de entrada y salida; o, en general, un fragmento de programa o *algoritmo*.

Podemos definir programa como "una unidad que recibe una cierta *entrada* y produce un *resultado*".

Definimos una **entrada** de un programa como un estado inicial del mismo, justo antes de ser ejecutado.

Definimos un **resultado** de un programa como un estado final del mismo, justo después de ser ejecutado.

Definimos los **asertos** como las fórmulas lógicas que se refieren al estado de un programa.

En estos términos, el significado de la especificación {P} A {Q} es: si el estado inicial de *A* cumple las aserciones de *P* entonces la ejecución de *A* termina en un estado que cumple las aserciones de *Q*.

Aparece aquí la idea de "resultado garantizado si se usa correctamente" –si se cumple la precondición. Y "resultados impredecibles si no se usa como se indica".

Un ejemplo [Peñ05]: suponemos declarado el tipo

```
tipo Vect = Vector[1..1000] de ent
```

---

queremos implementar una función que dado un vector *a* de tipo *vect* y un entero *n* nos devuelva un valor booleano que nos indique si el valor de alguno de los elementos a[1], ..., a[n] es igual a la suma de todos los que le preceden en el vector. Esta especificación deja algunos puntos sin aclarar, para el usuario: ¿se puede llamar a *esSuma* con un valor negativo de *n*? ¿y con *n*=0 o con *n*>1000? y en caso afirmativo ¿qué valor devolverá la función?; y para el implementador: si n≥1 y a[1]=0 ¿la función debe devolver verdadero o falso?

```
var
  a : Vect;
  n : Ent;
  b : Bool;
{ P ≡ 0 ≤ n ≤ 1000 }
  esSuma
{ Q ≡ b ↔ ∃i : 1 ≤ i ≤ n : (a(i) = ∑j : 1 ≤ j ≤ i-1 : a(j)) }
```

esta especificación deja claro

1. no se puede llamar a la función con n negativo o $n > 1000$
2. las llamadas con $n = 0$ son correctas y en ese caso la función devuelve b = falso
3. las llamadas con $n \geq 1$ y $a[1] = 0$ han de devolver b = cierto (suponiendo que el sumatorio sobre un dominio vacío es igual a cero[1]).

1.1.1 Representación de asertos en lógica de predicados

Vamos a utilizar la lógica de predicados adaptada a los elementos que aparecen en los programas. En un lenguaje imperativo existen una serie de tipos predefinidos. Una de las primeras propiedades del estado de un programa es el tipo de sus variables. Nuestra lógica dispone de símbolos para representar a los siguientes tipos predefinidos –algunos de los cuales no suelen aparecer en los lenguajes imperativos–:

- **Nat** , para los naturales
- **Ent**, para los enteros
- **Rac**, para los racionales
- **Real**, para los reales
- **Bool**, para los booleanos
- **Car**, para los caracteres
- **Vector**, para los vectores

Esto equivale a utilizar una signatura *heterogénea* para la lógica, donde los géneros son los tipos predefinidos.

A cada tipo predefinido le asignamos como significado su dominio de valores pretendido: N para nat, Z para ent, Q para rac, R para real, {*cierto*, *falso*} para bool. Los vectores requieren un tipo más complejo. Para un vector de tipo

[1] Esto es así porque consideramos que el valor de los cuantificadores, aritméticos y booleanos, aplicados sobre un dominio nulo dan como resultado el elemento neutro de las correspondientes operaciones binarias [Bal93:pag 8].

`Vector[`v_1 `..` v_n`] de` τ

siendo los valores v_1, ..., v_n de tipo τ', su dominio será el conjunto de aplicaciones entre $D_{\tau'}$ y D_τ que notaremos $D(\tau' \rightarrow \tau)$

De esta forma, el dominio para cada uno de los tipos predefinidos queda

Tipo (sintáctico)	**Dominio (semántico)**[2]
Nat	$D_{nat} = N$
Ent	$D_{ent} = Z$
Rac	$D_{rac} = Q$
Real	$D_{real} = R$
Bool	$D_{bool} = \{cierto, falso\}$
Car	$D_{car} = \{$'0', '1', ..., '9', 'a', ..., 'z', 'A', ..., 'Z'$\}$
Vector[τ'] de τ	$D_{vector[\tau'] \text{ de } \tau} = D(\tau' \rightarrow \tau)$

Decimos entonces que un valor de tipo τ es cualquier elemento del correspondiente dominio D_τ. Los programas disponen de recipientes para dichos valores: las variables y las constantes.

La signatura que utilizaremos contiene también las operaciones habituales sobre los tipos predefinidos.

Para definir operaciones, utilizaremos *perfiles* en los que se indica el nombre de la operación, el número de argumentos que tiene junto con el tipo de cada uno de ellos, y el tipo del resultado:

$f : \tau_1 \ldots \tau_n \rightarrow \tau$

Las operaciones que podemos utilizar sobre los tipos predefinidos:

— Operaciones numéricas, definidas para los tipos numéricos: Nat, Ent, Rac y Real

$+, *, - : \tau\ \tau \rightarrow \tau$

— Operaciones constantes de tipo bool

cierto, falso : $\rightarrow$ Bool

— Operaciones booleanas

AND, OR : Bool Bool $\rightarrow$ Bool

NOT : Bool $\rightarrow$ Bool

[2] Nótese la diferencia entre las funciones constantes `cierto` y `falso` y los valores del dominio semántico *cierto* y *falso*, que es la diferencia entre un lenguaje formal, como es la lógica que estamos definiendo, y su semántica que se construye utilizando conceptos matemáticos que tienen existencia independiente de dicho lenguaje.

— Operaciones de comparación con resultado booleano definidas para los tipos ordenados: Nat, Ent, Rac, Real y Car.

$>, <, \geq, \leq : \tau\ \tau \rightarrow$ Bool

— Operaciones de máximo y mínimo para los tipos ordenados

$\max, \min : \tau\ \tau \rightarrow \tau$

— Operaciones de igualdad y desigualdad con resultado booleano, definidas para todos los tipos con igualdad, en nuestro caso todos los tipos predefinidos excepto los vectores.

$=, \neq : \tau\ \tau \rightarrow$ Bool

— Una operación polimórfica para cada uno de los tipos predefinidos que devuelve verdadero o falso dependiendo de si el argumento es de tipo δ.

$\_:\delta : \tau \rightarrow$ Bool[3]

Además, también podremos emplear en los asertos cualquier operación que haya sido *totalmente especificada*, en el sentido que veremos más adelante, entendiéndose en ese caso que su comportamiento está definido por la especificación.

Construcción de asertos

La forma más simple de aserto es una expresión de tipo Bool. Definamos primero qué entendemos por "expresión de tipo τ".

Decimos que E es una expresión de tipo τ si y sólo si es de alguna de las formas siguientes:

— Es una variable de tipo τ

— Es una constante de tipo τ

— Es $f(E_1, ..., E_n)$ siendo el perfil de f

$f : \tau_1 ... \tau_n \rightarrow \tau$

y las E_i son expresiones de tipo τ_i para $i \in \{1, ..., n\}$.

Estas expresiones se corresponden con el concepto de término en lógica de predicados. Con las expresiones construiremos aserciones que nos permiten establecer condiciones sobre el estado de los programas.

Decimos que P es una aserción (aserto o predicado) si y sólo si es:

— Atómica.

  — P es una expresión de tipo Bool

  — P es

    E = E'

[3] Vemos aquí cómo en los perfiles también se puede indicar el modo de aplicación: prefijo, infijo o postfijo.

siendo E y E' expresiones

— Compuesta

— P es de alguna de las formas

$\neg R \quad R \wedge Q \quad R \vee Q \quad R \rightarrow Q \quad R \leftrightarrow Q$

(negación, conjunción, disyunción, condicional y bicondicional)

siendo R y Q aserciones

— P es de alguna de las formas

$\forall \vec{x} : D(\vec{x}) : R(\vec{x}) \qquad \exists \vec{x} : D(\vec{x}) : R(\vec{x})$

donde $D(\vec{x})$ es una aserción de dominio para las variables $\vec{x}$, y $R(\vec{x})$ es una aserción donde intervienen las variables $\vec{x}$.

La existencia de ecuaciones se justifica por el interés en comparar expresiones de tipos en los que no existe la igualdad. Para los tipos simples coincide con el uso de las operaciones booleanas de comparación, pero los vectores –como otros tipos de datos que introduciremos más adelante– sólo se pueden comparar mediante esas aserciones ya que no disponen de operación de igualdad.

Para evitar un uso excesivo de paréntesis, definimos el orden de prioridad de las conectivas lógicas y los cuantificadores existenciales

---

De mayor a menor prioridad

$\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \delta$

donde δ denota indistintamente $\forall$ o $\exists$.

Veamos algunos ejemplos de expresiones y aserciones

— x+3 es una expresión de tipo numérico

— x > 10 es una expresión de tipo booleano y, por lo tanto una aserción atómica

— (z='a') AND (t=cierto) es una expresión de tipo booleano y una aserción atómica

— (z='a') $\wedge$ (t=cierto) es una aserción compuesta

— (x>0) $\vee$ (x<0) es una aserción compuesta

— (x>3) $\rightarrow$ (x>0) es una aserción compuesta

— $\forall$x : ent : (x*0=0) es una aserción compuesta, donde la aserción de dominio es una notación abreviada de x : ent, una expresión booleana donde se aplica la operación :ent

— $\exists$i : 1≤i≤N : (i*2=3) es una aserción compuesta, donde la aserción de dominio es una notación abreviada de (1≤i) $\wedge$ (i≥N).

El empleo de cuantificadores en las aserciones supone la utilización de variables para representar un subconjunto de los posibles valores de un cierto dominio y no como contenedores de valores asociados a una posición de memoria. Decimos que las variables cuantificadas son *mudas* en el sentido de que no representan un valor de la memoria. Esto hace que una variable sujeta a un cuantificador se pueda renombrar por otra sin que cambie el sentido de la aserción.

Una aparición de una variable se dice que **está ligada** si se encuentra en el ámbito de un cuantificador con esa variable. Diremos que es **libre** en otro caso.

Hablamos de "apariciones libres y ligadas" porque en una misma aserción una variable puede aparecer libre y ligada como en

---

$(x>0) \wedge (\exists x : Ent : x=2)$

Por claridad, trataremos de que una misma variable no aparezca libre y ligada. De esta forma, una aserción equivalente a la anterior es la que resulta de renombrar las apariciones ligadas de x por y:

$(x>0) \wedge (\exists y : Ent : y = 2)$

Para aumentar la expresividad del lenguaje que utilizaremos para describir el estado de los programas, introducimos otras formas de construir expresiones, que se caracterizan por el uso de variables ligadas

---

Las expresiones pueden ser también de la siguiente forma:

— Sumatorios

$$\sum \vec{i} : D(\vec{i}) : E(\vec{i})$$

— Productos extendidos

$$\prod \vec{i} : D(\vec{i}) : E(\vec{i})$$

— Máximos

$$\max \vec{i} : D(\vec{i}) : E(\vec{i})$$

— Mínimos

$$\min \vec{i} : D(\vec{i}) : E(\vec{i})$$

— Conteos

$$\# \vec{i} : D(\vec{i}) : P(\vec{i})$$

Siendo $D(\vec{i})$ una aserción de dominio, $E(\vec{i})$ una expresión y $P(\vec{i})$ una aserción, tales que en todas ellas pueden aparecer las variables $\vec{i}$. Cualquier aparición de las variables $\vec{i}$ se entiende

ligada en estas expresiones. Los tipos de las expresiones E($\vec{i}$) son numéricos para los sumatorios y productos extendidos y ordenados para máximos y mínimos.

Veamos algunos ejemplos:

— $\sum i : 1 \leq i \leq 100 : (i*i)$ que representa la suma de las expresiones i*i, cuando i toma valores en el dominio indicado. (se puede indicar la notación con super y subíndices que les puede resultar más familiar).

— $\prod j : 1 \leq j \leq n : j$ esta expresión obtiene el producto factorial del valor contenido en la variable *n*.

cte

N = ?; % entero ≥1

var

v: Vector [1..N] **de** Ent;

— max k : 1 ≤ k ≤ N : v(k) que representa el máximo de los valores del vector v

— min k : 1 ≤ k ≤ N : v(k) que representa el mínimo de los valores del vector v

— #i : 1 ≤ i ≤ N : (v(i) = 0) que representa el número de componentes cuyo valor es cero.

El último elemento sintáctico que nos resta por introducir son las *sustituciones*. Resulta muy importante pues es la base de la regla de verificación de una de las instrucciones más importantes en los lenguajes imperativos: la asignación. El significado intuitivo del proceso de sustitución es el siguiente: el aserto A expresa un hecho que se requiere de un estado determinado, referido al valor de x; A[x/E] expresa un hecho análogo referido al valor que toma la expresión E.

Definimos una sustitución como el proceso de reemplazar simultáneamente todas las apariciones libres de una variable por una expresión. Emplearemos la notación

[x/E]

para indicar la sustitución de la variable x por la expresión E. También

$[x_1/E_1, \ldots , x_n/E_n]$

para indicar la sustitución simultánea de las variables $x_1, \ldots , x_n$ por las expresiones $E_1, \ldots , E_n$, respectivamente.

Las sustituciones sólo se pueden realizar para variables libres, pues son éstas las que representan valores del estado. En esta definición se supone que en ningún caso se ha utilizado una variable libre y una ligada con el mismo nombre. De no ser así, es necesario realizar un renombramiento de las variables ligadas cuyos nombres coincidan con las variables libres que aparecen en las expresiones o en los asertos.

Algunos ejemplos de sustituciones (correctos e incorrectos)

es incorrecto sustituir variables ligadas

$(\forall x : 1 \le x \le 100 : (x = a))\ [x/3] \rightarrow \forall x : 1 \le 3 \le 100 : (3 = a)$

es incorrecto cualquier forma de sustitución que no sustituya todas las apariciones simultáneamente

$((x = 17) \vee (x>100))\ [x/(x*x)] \rightarrow ((x*x = 17) \vee (x>100))\ [x/(x*x)] \rightarrow$
$((x*x) * (x*x) = 17) \vee (x*x>100)$

en sustituciones múltiples es incorrecto hacer las sustituciones secuencialmente

$((x*17)*y)\ [x/(2*y), y/z] \rightarrow ((2*y)*y)\ [y/z] \rightarrow ((2*z)*z)$

Significado de los asertos

Intuitivamente un aserto expresa una afirmación que puede ser verdadera o falsa. El significado de un aserto será su verdad o su falsedad. Para describir cómo se asigna un valor de verdad a los asertos empezaremos por describir cómo se asocia ese valor con los asertos atómicos y posteriormente daremos significado a las conectivas lógicas, cubriendo así cualquier posible aserto.

Para poder afirmar si un aserto es verdadero o falso necesitamos conocer el valor de las variables libres que en él aparecen, es decir, necesitamos conocer el estado del programa. Definimos el estado de un programa como una descripción instantánea de los valores asociados a las variables del programa, es decir, una aplicación de los identificadores de la variables en valores de los dominios correspondientes, que representaremos con la notación

σ: IdVar $\rightarrow$ D

donde σ es un estado, IdVar es el conjunto de identificadores de las variables y D es la unión de los dominios de las variables.

Por ejemplo:

```
var
  x, y: Ent;
  b: Bool;
```

un estado σ es una aplicación

$\sigma : \{x,y,b\} \rightarrow D_{ent} \cup D_{bool}$

por ejemplo

$\sigma = \{(x,1), (y,2), (b,falso)\}$

El significado de los asertos se construye inductivamente a partir del significado de los predicados –las expresiones de tipo booleano–.

Dada una expresión E de tipo τ, definimos el significado de E bajo el estado σ, que notaremos

val[E, σ]

como el valor de D_τ resultante de la aplicación del significado habitual o especificado de las operaciones empleadas en E, aplicadas sobre los valores que asigna σ a las variables libres. Este significado estará definido solamente en el caso de que σ asigne valores a todas las variables libres de E.

Dado un aserto P, definimos el significado de P bajo el estado σ, que notaremos

val[P, σ]

de forma inductiva como:

— Aserciones atómicas:

— Si P es una expresión de tipo Bool tomamos el significado de P como expresión

— Si P es E = E'

$$\textbf{val}[P,\sigma] = \begin{cases} \textit{cierto} & \text{si } \textbf{val}[E,\sigma] = \textbf{val}[E',\sigma] \\ \textit{falso} & \text{en otro caso} \end{cases}$$

— Aserciones compuestas

— Si P es $\neg$R

$$\textbf{val}[P,\sigma] = \begin{cases} \textit{cierto} & \text{si } \textbf{val}[R,\sigma] = \textit{falso} \\ \textit{falso} & \text{si } \textbf{val}[R,\sigma] = \textit{cierto} \end{cases}$$

— Si P es R $\wedge$ Q

$$\textbf{val}[P,\sigma] = \begin{cases} \textit{cierto} & \text{si } \textbf{val}[R,\sigma] = \textit{cierto} \text{ y } \textbf{val}[Q,\sigma] = \textit{cierto} \\ \textit{falso} & \text{en otro caso} \end{cases}$$

— Si P es R $\vee$ Q

$$\textbf{val}[P,\sigma] = \begin{cases} \textit{cierto} & \text{si } \textbf{val}[R,\sigma] = \textit{cierto} \text{ o } \textbf{val}[Q,\sigma] = \textit{cierto} \\ \textit{falso} & \text{en otro caso} \end{cases}$$

— Si P es R $\rightarrow$ Q

$$\textbf{val}[P,\sigma] = \begin{cases} \textit{falso} & \text{si } \textbf{val}[R,\sigma] = \textit{cierto} \text{ y } \textbf{val}[Q,\sigma] = \textit{falso} \\ \textit{cierto} & \text{en otro caso} \end{cases}$$

— Si P es R $\leftrightarrow$ Q

$$\textbf{val}[P,\sigma] = \begin{cases} \textit{falso} & \text{si } \textbf{val}[R,\sigma] \text{ y } \textbf{val}[Q,\sigma] \text{ tiene significados opuestos} \\ \textit{cierto} & \text{en otro caso} \end{cases}$$

— Si P es $\forall \vec{x} : D(\vec{x}) : R(\vec{x})$ entonces

val[P, σ] = *cierto*

si y sólo si para todo estado σ' que extiende a σ y asigna valores a las variables $\vec{x}$ de tal forma que

val[D, σ'] = *cierto*

se cumple que

val[R, σ'] = *cierto*

— Si P es $\exists \vec{x} : D(\vec{x}) : R(\vec{x})$ entonces

val[P, σ] = *cierto*

si y sólo si para algún estado σ' que extiende a σ y asigna valores a las variables $\vec{x}$ de tal forma que

val[D, σ'] = *cierto*

se cumple que

val[R, σ'] = *cierto*

Este significado estará definido solamente en el caso de que σ asigne valores a todas las variables libres de P.

En el caso de los cuantificadores, consideramos que para σ'=∅ el cuantificador universal es *cierto* y el existencial *falso*.

Veamos unos ejemplos del significado de expresiones y aserciones

---

Sea σ un estado que da valores a las variables

```
var
  x, y: Ent;
  b : Bool;
```

de la siguiente forma

σ={(x, 1), (y, 2), (b, *falso*)}

E_1 es `x=y` **val**[E_1, σ] = *falso*

E_2 es `x+(2*y)` **val**[E_2, σ] = 5

P_1 es `(x=1)∧(NOT b)` **val**[P_1, σ] = *cierto*

P_2 es `∀i : 10<i<15 : (i*2<30)` **val**[P_2, σ] = *cierto*

donde los σ' serían: {(x, 1), (y, 2), (b, *falso*), (i, 11)}, ..., {(x, 1), (y, 2), (b, *falso*), (i, 14)}

De manera recíproca a cómo definimos el significado de un aserto bajo un estado definimos el conjunto de estados que hacen cierto un aserto, que nos servirá para definir más cómodamente el significado de las expresiones que aún nos restan.

Definimos el conjunto de estados que satisfacen un aserto P, que notaremos

est[P]

como

est[P] $=_{def}$ { σ | **val**[P, σ] = *cierto* }

Naturalmente, para que un estado satisfaga un aserto, el aserto debe estar definido para ese estado, es decir, el estado debe asignar valores a todas las variables libres del aserto.

Veamos algunos ejemplos de los conjuntos de estados que satisfacen ciertos asertos

---

P_1 es `falso` **est**[P_1] = $\varnothing$

(pues el valor de operación constante booleana falso siempre es *falso*)

P_2 es `cierto` **est**[P_2] = TODOS

(pues el valor de la operación constante booleana cierto es siempre *cierto*)

P_3 es `∃i : Nat : x = 2 * i`

est[P_3] = {σ | σ asigna a la variable x un valor nulo o par}

Definamos por fin el significado de las expresiones que nos restan:

— Si *E* es una expresión de la forma

$$\sum \vec{i} : D(\vec{i}) : E(\vec{i})$$

entonces

$$\textbf{val}[E,\sigma] = \sum_{\delta \in C} \textbf{val}[E', \sigma \cup \delta]$$

siendo C el conjunto de estados que satisfacen D, limitados a las variables de $\vec{i}$, y siendo $\sigma \cup \delta$ el estado que se obtiene al combinar las asignaciones de σ y δ.

Si C es el conjunto vacío, entonces se conviene en que **val**[E,σ] = 0.

— Si *E* es una expresión de la forma

$$\prod \vec{i} : D(\vec{i}) : E(\vec{i})$$

entonces

$$\textbf{val}[E,\sigma] = \prod_{\delta \in C} \textbf{val}[E', \sigma \cup \delta]$$

siendo C el conjunto de estados que satisfacen D, limitados a las variables de $\vec{i}$, y siendo $\sigma \cup \delta$ el estado que se obtiene al combinar las asignaciones de σ y δ.

Si C es el conjunto vacío, entonces se conviene en que **val**[E,σ] = 1.

— Si *E* es una expresión de la forma

$$\max \vec{i} : D(\vec{i}) : E(\vec{i})$$

entonces

$$\textbf{val}[E,\sigma] = \text{Max}_{\delta \in C}\ \textbf{val}[E', \sigma \cup \delta]$$

siendo C el conjunto de estados que satisfacen D, limitados a las variables de $\vec{i}$, y siendo $\sigma \cup \delta$ el estado que se obtiene al combinar las asignaciones de σ y δ.

Si C es el conjunto vacío, entonces se conviene en que **val**[E,σ] queda indefinido.

— Si *E* es una expresión de la forma

$$\min \vec{i} : D(\vec{i}) : E(\vec{i})$$

entonces

val[E,σ] = $\text{Min}_{\delta \in C}$ **val**[E', σ ∪ δ]

siendo C el conjunto de estados que satisfacen D, limitados a las variables de $\vec{i}$, y siendo σ ∪ δ el estado que se obtiene al combinar las asignaciones de σ y δ.

Si C es el conjunto vacío, entonces se conviene en que **val**[E,σ] queda indefinido.

— Si *E* es una expresión de la forma

$\#\vec{i} : D(\vec{i}) : P(\vec{i})$

entonces

val[E,σ] = |{σ' | **val**[D∧P, σ ∪ σ']=*cierto* }

siendo σ' el estado que se obtiene al considerar las asignaciones de variables de $\vec{i}$.

Nótese, en primer lugar, que los símbolos utilizados en el lenguaje de asertos y en el correspondiente significado difieren, con ello se quiere representar que se trata de entidades diferentes; por ejemplo, en un caso el sumatorio no es más que un elemento del lenguaje que tiene unas ciertas propiedades mientras que en el otro el sumatorio se refiere al concepto matemático conocido por todos.

Por otra parte, nótese también que cuando no hay ningún estado que satisfaga la restricción de dominio hemos elegido que el valor de la expresión sea el elemento neutro de la correspondiente operación matemática. Para el máximo y el mínimo los elementos neutros serían, respectivamente, el menor elemento posible y el mayor elemento del dominio; no hemos definido el significado porque no siempre está garantizado que existan dichos elementos.

Veamos un ejemplo:

Sea σ un estado que da valor a las variables

```
var x, y: Ent;
```

de la siguiente forma: σ={(x,10), (y,4)}

— Si E es `∑i : 1≤i≤y : (i*x)`

entonces

val[E, σ] = **val**[i*x, σ ∪ {(i, 1)}] + ... + **val**[i*x, σ ∪ {(i, 4)}]
= 1*10+ ... + 4*10
= 10+20+30+40
= 100

— Si E es `#i : 5≤i≤y : (i*x>34)`

entonces

val[E, σ] = |C|

siendo C

C = {σ' | **val**[(5≤i≤y)∧(i*x>34), σ∪σ']=*cierto*}
= {σ' | **val**[(5≤i≤y), σ∪σ']=*cierto*}∩{σ' | **val**[(i*x>34), σ∪σ']=*cierto*}
= {σ' | **val**[(5≤i≤4), σ']=*cierto*}∩{σ' | **val**[(i*10>34), σ']=*cierto*}
= ∅ ∩ {σ' | **val**[(i*10>34), σ']=*cierto*} = ∅

por tanto,

$$\mathbf{val}[E,\sigma] = |C| = |\varnothing| = 0$$

En la verificación de los algoritmos necesitaremos razonar sobre los asertos, las condiciones que describen el estado de los programas, y tendremos que poder determinar cuándo un aserto es consecuencia lógica de otro. Para ello introducimos el concepto de *fuerza de los asertos*.

Fuerza de los asertos

Intuitivamente una condición es más restrictiva que otra si es más difícil de satisfacer, o, dicho de otro modo, si se satisface en menos ocasiones. Refiriéndonos a los asertos sobre el estado de los programas, un aserto será tanto más fuerte cuantos menos estados lo satisfagan. Aunque no es exactamente esta idea la que utilizamos, pues no definimos la *fuerza* como una magnitud absoluta sino relativa, y de tal forma que puede ocurrir que dos asertos no sean comparables.

Dados dos asertos P y Q, decimos que P es más fuerte que Q si se cumple

$$\mathbf{est}[P] \subseteq \mathbf{est}[Q]$$

y lo notaremos $P \Rightarrow Q$

Decimos en este caso que Q es una consecuencia lógica de P, es decir, siempre que se cumple P se cumple también Q. P es más restrictiva que Q en el sentido de que puede haber estados que satisfagan Q pero no así P.

Por ejemplo

```
var x : Ent;
```

tenemos que $x \leq 0 \Rightarrow x \leq 10$

ya que

$\mathbf{est}[x \leq 0] = \{\sigma \mid \sigma$ asigna a x el valor cero $\}$

$\mathbf{est}[x \leq 10] = \{\sigma \mid \sigma$ asigna a x el valor cero, o el 1, o el 2, ..., o el 10 $\}$

por tanto,

$$\mathbf{est}[x \leq 0] \subseteq \mathbf{est}[x \leq 10]$$

Conviene no confundir la relación de fuerza entre aserciones $P \Rightarrow Q$ con la conectiva implicación de construcción de aserciones $P \rightarrow Q$.[4]

Se puede demostrar que `falso` es más fuerte que cualquier aserto –ya que el conjunto vacío está contenido en cualquier conjunto– y que cualquier aserto es más fuerte que el aserto `cierto` –ya que cualquier conjunto está contenido en el conjunto TODOS–.

---

[4] Lo que si es correcto es que si el aserto $P \rightarrow Q$ se satisface en cualquier estado entonces se tiene que $P \Rightarrow Q$, ya que, o bien $\mathbf{est}[P]=\varnothing$, o bien $\mathbf{est}[P] \subseteq \mathbf{est}[Q]$ debido a la definición de la semántica del conectivo $\rightarrow$.

Cuando los estados que satisfacen dos asertos son los mismos decimos que esos asertos son equivalentes

---

Dados dos asertos P y Q, decimos que P y Q son equivalentes si se cumple

est[P] = **est**[Q]

y lo notaremos $P \Leftrightarrow Q$

---

por ejemplo

```
var x : Ent;
```

tenemos que

$x=0 \Leftrightarrow (x<1) \wedge (x>-1)$

ya que

$$\textbf{est}[x=0] = \{\sigma \mid \sigma \text{ asigna a x el valor cero }\}$$

$$\textbf{est}[(x<1)\wedge(x>-1)] = \{\sigma \mid \sigma \text{ asigna a x un valor que 1 y mayor que } -1\}$$

por el conocimiento que tenemos de los números enteros podemos concluir que

est[x=0] = **est**[(x<1)∧(x>-1)]

En el razonamiento con los programas nos interesará conseguir asertos que sean fortalecimientos o debilitamientos de otros dados, es decir, asertos que sean más fuertes (satisfechos por menos estados) o más débiles (satisfechos por más estados) que uno dado. Una forma de hacerlo es añadir una condición adicional –fortalecimiento– o añadir una condición alternativa –debilitamiento–.

Dada una aserción P, decimos que P∧Q es un fortalecimiento de P, o que P se fortalece con Q.

Dada una aserción P, todo fortalecimiento de P es más fuerte que P

Dada una aserción P, decimos que P∨Q es un debilitamiento de P, o que P se debilita con Q.

Dada una aserción P, P es más fuerte que todo debilitamiento de P.

Nótese que puede ocurrir que un fortalecimiento o un debilitamiento en realidad no varíe la fuerza de un aserto –el conjunto de estado que lo satisfacen–, pero aún así se siguen cumpliendo las proposiciones sobre la fuerza con respecto a fortalecimientos y debilitamientos porque en la definición de fuerza se ha utilizado inclusión no estricta entre conjuntos –P es más fuerte que P–.

Como hemos visto en un ejemplo anterior, es posible expresar el mismo predicado de distintas formas, y en cada momento nos puede interesar más una cierta formulación. Nos interesa disponer de un conjunto de leyes de equivalencia que permitan transformar unos asertos en otros, sabiendo que se preserva el conjunto de estados que los satisfacen.

Leyes de equivalencia

Tenemos tres grupos de leyes de equivalencia, según se refieran a las operaciones, a las conectivas lógicas o a los cuantificadores.

Leyes de las operaciones y los dominios predefinidos

Son leyes que se derivan de las propiedades que verifican las operaciones habituales en los tipos predefinidos. De forma resumida dichas propiedades son:

— Conmutatividad. Para las operaciones +, *, AND, OR

— Elemento neutro. De la suma es el cero, del producto es el uno, del AND es la constante `cierto` y del OR la constante `falso`.

— Distributividad. Del producto con respecto de la suma y de la diferencia.

— Coerciones de los valores numéricos. Todos los naturales son enteros, todos los enteros son racionales, todos los racionales son reales.

Esta propiedad hace que si nos encontramos con una operación como x*y con x Real e y Ent, consideremos que se trata de una multiplicación entre reales.

Leyes de las conectivas lógicas (álgebra de Boole)

La corrección de estas leyes se basa en la semántica que hemos definido para las conectivas. Son leyes de equivalencia entre asertos como indica el uso de $\Leftrightarrow$.

— Conmutatividad

$$P \wedge Q \Leftrightarrow Q \wedge P$$
$$P \vee Q \Leftrightarrow Q \vee P$$
$$P \leftrightarrow Q \Leftrightarrow Q \leftrightarrow P$$

— Asociatividad

$$P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$$
$$P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$$

— Idempotencia

$$P \wedge P \Leftrightarrow P$$
$$P \vee P \Leftrightarrow P$$

— Distributividad

$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$
$$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$$

— Absorción

$$P \vee (P \wedge Q) \Leftrightarrow P$$
$$P \wedge (P \vee Q) \Leftrightarrow P$$

— Neutros

$$P \wedge \text{falso} \Leftrightarrow \text{falso}$$
$$P \wedge \text{cierto} \Leftrightarrow \text{falso}$$
$$P \vee \text{falso} \Leftrightarrow P$$
$$P \vee \text{cierto} \Leftrightarrow \text{cierto}$$

— Contradicción

$$P \wedge \neg P \Leftrightarrow \text{falso}$$

— Tercio excluido

$$P \vee \neg P \Leftrightarrow \text{cierto}$$

— Doble negación

$$\neg(\neg P) \Leftrightarrow P$$

— Leyes de De Morgan

$$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$$
$$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$$

— Implicación y doble implicación

$$P \rightarrow Q \Leftrightarrow \neg P \vee Q$$
$$P \leftrightarrow Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$$

Donde P, Q y R son asertos.

Leyes de los cuantificadores

La corrección de estas leyes se basa en la semántica que hemos definido para los cuantificadores. Son leyes de equivalencia entre asertos como indica el uso de $\Leftrightarrow$.

— Renombramiento de variables ligadas

$$\forall x : D(x) : P(x) \Leftrightarrow \forall y : D(x)[x/y] : P(x)[x/y]$$
$$\exists x : D(x) : P(x) \Leftrightarrow \exists y : D(x)[x/y] : P(x)[x/y]$$

siendo *y* una variable que no aparece en D ni en P

— Cuantificación en una equivalencia. Si $P \Leftrightarrow Q$

$$\forall x : D(x) : P(x) \Leftrightarrow \forall x : D(x) : Q(x)$$
$$\exists x : D(x) : P(x) \Leftrightarrow \exists x : D(x) : Q(x)$$

— Negación de un cuantificador

$$\neg(\forall x : D(x) : P(x)) \Leftrightarrow \exists x : D(x) : (\neg P(x))$$
$$\neg(\exists x : D(x) : P(x)) \Leftrightarrow \forall x : D(x) : (\neg P(x))$$

— Desplazamiento de cuantificadores

$$(\forall x : D(x) : P(x)) \wedge (\forall x : D(x) : Q(x)) \Leftrightarrow \forall x : D(x) : (P \wedge Q)(x)$$
$$(\exists x : D(x) : P(x)) \vee (\exists x : D(x) : Q(x)) \Leftrightarrow \exists x : D(x) : (P \vee Q)(x)$$

Leyes de descomposición de cuantificadores

Cuando el dominio de un cuantificador no es nulo, podemos separar uno de sus elementos y reducir en uno el dominio del cuantificador. El elemento separado se combina con el cuantificador reducido, mediante la correspondiente operación binaria. Por ejemplo

---

si $N \geq 1$
$\sum i : 1 \leq i \leq N : a(i) = a(1) + \sum i : 2 \leq i \leq N : a(i)$

---

nótese que la condición $N \geq 1$ es necesaria para garantizar que a(1) está efectivamente entre los elementos que han de ser sumados. De la misma manera, podemos separar un elemento arbitrario del dominio de un cuantificador cualquiera, salvo que sea vacío, combinándolo con el resto mediante la operación binaria asociada al cuantificador. En el caso de maximización y minimización, hemos de garantizar que el dominio original tiene al menos dos elementos, con el fin de que el cuantificador restante no se aplique sobre un dominio nulo; pero para los demás cuantificadores definidos basta con que exista en el dominio un elemento que separar.

Más formalmente, lo que estamos haciendo es aplicar propiedades como la siguiente:

---

dados dos dominios disjuntos D_1 y D_2

$$\sum i : i \in (D_1 \cup D_2) : a(i) \equiv \sum i : i \in D_1 : a(i) + \sum i : i \in D_2 : a(i)$$

Análogamente, se tienen propiedades similares para los cuantificadores producto extendido (*), existencial ($\vee$), universal ($\wedge$) y de conteo (+), utilizando la correspondiente operación binaria en lugar de la suma. ¿Y para el máximo y el mínimo?

Operaciones parciales

Para terminar por fin con el apartado dedicado a la representación de los asertos trataremos el problema de las operaciones parciales.

Con los tipos predefinidos que vamos a utilizar nos encontraremos con operaciones que no están definidas para algún valor del dominio asociado a sus argumentos. Por ejemplo, la división no está definida si el divisor es cero, es una operación parcial.

Decimos que una operación f con dominio $\tau_1, ..., \tau_n$ es parcial si para algún $i \in \{1, ..., n\}$, existe un valor en D_{τ_i}, para el que no está definida. La notaremos empleando una flecha cortada entre el dominio y el codominio declarado

f: $\tau_1 \ldots \tau_n \dashrightarrow \tau$

Decimos que es total en caso contrario.

Una operación definida con una operación total puede no estar definida debido a que uno de sus argumentos contenga una expresión que no está definida. Por ejemplo

---

dada la operación parcial

div : Ent Ent $\dashrightarrow$ Ent

la operación * : Nat Nat $\rightarrow$ Nat es total, sin embargo la expresión

(x div 0) * 2

no está definida, independientemente del estado que se tome para asignar valores a las variables libres.

El empleo de ciertas expresiones en una especificación se ve limitado por el hecho de que la expresión pueda no estar definida. Para poder controlar esta situación y obtener condiciones que reflejen el comportamiento de los programas, emplearemos *asertos de definición.*

Dada una expresión E, el aserto de definición de E es

def(E)

Dada una expresión E y un estado σ, definimos el significado de def(E) bajo σ como:

$$\mathbf{val}[\text{def}(E),\sigma] =_{\text{def}} \begin{cases} \textit{cierto} & \text{si } \mathbf{val}[E,\sigma] \text{ está definido} \\ \textit{falso} & \text{en otro caso} \end{cases}$$

Utilizaremos los asertos de definición para completar los asertos donde aparezcan expresiones con operaciones parciales. Por ejemplo

---

Sea E una expresión donde interviene una operación parcial. El aserto

E*0 = 0

no se puede garantizar que sea siempre cierto. Pues el significado del producto depende de que sus dos argumentos estén definidos. Lo que sí será cierto es el aserto

def(E) $\rightarrow$ (E*0=0)

---

ya que si el aserto de definición es cierto, el aserto de la conclusión está definido y también lo es; y si el aserto de definición es falso, la semántica de la conectiva $\rightarrow$ garantiza que el aserto es cierto. (¿No se consigue lo mismo con $\wedge$?)

Nótese que según la definición de los asertos de definición, se incluye también la condición de que las variables que aparezcan en una expresión E hayan sido declaradas. Si no fuera así, $\mathbf{val}[E,\sigma]$ no estaría definido, ya que el estado σ no les daría valor.

1.1.2 Especificación pre/post

Pasaremos ahora a definir las especificaciones con precondiciones y postcondiciones, y a establecer su significado de modo formal. De esta manera podremos emplearlas para la especificación de algoritmos y programas. Esto lo mostraremos por medio de ejemplos que ilustrarán las posibilidades del empleo de asertos para definir condiciones de los programas. Por último, estableceremos una serie de propiedades básicas de las especificaciones que estarán relacionadas con la verificación de algoritmos.

Definimos una especificación con precondiciones y postcondiciones, o especificación pre/post, para un programa o algoritmo A como un esquema:

D
{ P }
A
{ Q }

donde:

— De es un conjunto de declaraciones de constantes y variables disponibles para el programa A

— P es un aserto (la precondición)

— Q es un aserto (la postcondición)

Si las declaraciones son conocidas podremos escribir

{ P }
A
{ Q }

o bien, si conviene, { P } A { Q }.

El significado de una especificación viene dado por el cambio de estado que implica la ejecución del programa.

Dada una especificación pre/post

{ P } A { Q }

definimos su significado como:

> Si A comienza en un estado que satisface P, entonces A termina en tiempo finito en un estado que satisface Q.

En esta definición se recoge el carácter de contrato de la especificación, el implementador se compromete a obtener un programa que partiendo de un estado que verifica P llegue a un estado que verifica Q. Si el estado inicial no verifica P, el implementador no se compromete a nada.

Una especificación pre/post se puede utilizar con distintos fines dependiendo de cuáles de sus constituyentes sean conocidos. Si conocemos los tres elementos entonces el objetivo será demostrar que el programa cumple la especificación: verificación del programa. Podemos, en otro caso, partir de un fragmento de programa y una postcondición para tratar de determinar una precondi-

ción razonable que haga cierta la especificación; normalmente el objetivo de esta manipulación será obtener la postcondición del fragmento de programa que preceda a A. Podemos usar también la especificación como ayuda para el diseño del programa; en tal caso, A será desconocido, dispondremos de P y Q, y habremos de diseñar P de manera que se cumpla { P } A { Q }: derivación de programas a partir de la especificación.

Es importante hacer notar que siempre consideramos que están inicializadas todas aquellas variables a las que haga referencia la precondición.

Variables auxiliares

Antes de pasar a ejemplos concretos de especificaciones pre/post, necesitamos introducir un mecanismo más: las variables auxiliares de la especificación.

Supongamos que queremos especificar la el programa que realiza la división entera de dos números naturales:

```
var a, b, c, r :Nat;
{ b > 0 }
  dividir
{ (a = b * c + r) ∧ (r < b) }
```

Este programa verifica la especificación dada:

```
a := 0; c := 0; r := 0;
```

El problema es que no hemos sido suficientemente restrictivos. Como dice Ricardo, al escribir una especificación debemos pensar que el implementador es un ser malévolo y despreciable que intentará cumplir las condiciones del contrato de la manera que le resulte más sencilla.

El error se encuentra en la postcondición que, si bien exige que las variables del programa cumplan una cierta relación, permite que éstas tomen nuevos valores. Dicho de otra forma, la postcondición tienen sentido si `a` y `b` se refieren a los valores de entrada.

Este problema requiere el uso de un convenio. Vamos a utilizar *variables auxiliares de la especificación*, que nos servirán para recoger los valores de entrada que nos interese para escribir las especificaciones. Por ejemplo, en el caso de dividir:

```
var a, b, c, r :Nat;
{ b > 0 ∧ a = A ∧ b = B }
  dividir
{ (A = B * c + r) ∧ (r < B) }
```

Usamos las variables auxiliares A y B, cuyo valor establecemos en la precondición. Por convenio, escribimos en minúscula las variables de programa y en mayúscula las auxiliares de la especificación. El programa no puede modificar el valor de las variables de especificación ya que no

forman parte de su comportamiento visible. El siguiente es un criterio para el uso de este tipo de variables:

Si una variable de programa tiene un valor de entrada relevante, entonces la precondición debe reflejarlo con una ecuación con esa variable y una variable auxiliar.

De esta forma podremos utilizar el valor de entrada en la postcondición.

Un concepto relacionado con el uso de variables auxiliares en la precondición es el de existencia de valor. Aquellas variables para las que no se identifica un valor inicial en la precondición podemos suponer que no están inicializadas; es lo que ocurre con las variables de salida, destinadas a recoger el resultado del algoritmo. En [Peñ05] se propone el uso de un valor especial *indefinido* ($\perp$), que se añade a cada dominio, y que se supone el valor de las variables no inicializadas.

Las variables auxiliares de la especificación se utilizan también para indicar que ciertas variables de programa no cambian de valor. Así, por ejemplo, podemos reforzar aún más la postcondición de la división entera como

---

```
{ (A = B * c + r) ∧ (r < B) ∧ (a = A) ∧ (b = B) }
```

Recapitulando, en nuestras especificaciones aparecerán los siguientes tipos de variables:

- Ligadas. Variables introducidas por cuantificadores y expresiones extendidas. No representan a la memoria del programa y los estados no han de asignarles valores. Se pueden renombrar sin que se modifique el valor o el significado de la expresión o el aserto en el que aparecen.
- Libres. El resto de las variables.
  - De programa. Variables que representan el comportamiento visible del programa, pueden tener valores que el programa puede modificar. Por convenio las escribimos en minúscula.
  - Auxiliares de la especificación. No son accesibles para el programa, aunque toman valores a partir de los estados. Tienen utilidad para la especificación, porque permiten recoger los valores de las variables del programa en ciertos estados de interés como, por ejemplo, el estado inicial. También sirven para representar valores que no nos interesa almacenar en el estado del programa –no son ni un dato ni un resultado–, pero que pueden ser necesarios para escribir una condición determinada (como en la postcondición de la división entera). Por convenio, las escribimos con la primera letra mayúscula.

Algunos ejemplos de especificación pre/post

Intercambio de valores entre dos variables

La especificación se apoya en el uso de variables auxiliares de especificación:

```
var x, y : Ent;
{ x = X ∧ y = Y }
  intercambio
{ x = Y ∧ y = X }
```

Copia del valor de una variable

```
var x, y : Ent;
{ x = X }
  copia
{ y = X ∧ x = X }
```

podemos obviar la ecuación `x=X` en la postcondición si no queremos imponer esta condición adicional.

Raíz cuadrada exacta

Tenemos la declaración de variables

```
var x, y : Ent;
```

y tenemos que especificar un algoritmo que calcule en `y` la raíz cuadrada del valor de `x`, sabiendo que `x` contiene un valor que en el cuadrado de un entero.

Una primera versión

```
var x, y : Ent;
{ x = X }
  Raíz
{ y*y = X }
```

Es incompleta porque no exige a los valores de entrada las propiedades necesarias para poder ejecutar el programa.

```
var x, y : Ent;
{ x = X ∧ X = Y * Y }
  Raíz
{ y*y = X ∧ x = X }
```

Aún quedaría otro detalle. Como las variables auxiliares de la especificación no aparecen en la declaración de variables del programa, no está especificado cuál es su tipo. De esta forma, podríamos leer la precondición como que `X` es el cuadrado de un número real. Finalmente

```
var x, y : Ent;
{ x = X ∧ X = Y * Y ∧ Y : Ent }
  Raíz
{ y*y = X ∧ x = X }
```

Al escribir esta especificación uno se puede sentir tentado de escribir la postcondición como

```
{ y = Y ∧ x = X }
```

que es válida según la semántica definida para las especificaciones pre/post, pues si se parte de un estado que cumple la precondición se alcanzará un estado en el que `y` es la raíz cuadrada exacta de `x`. El problema es que para poder relacionar la postcondición con el significado que se pretende dar al programa es necesario tener en cuenta la precondición y eso hace que dicha especificación resulte poco útil en determinadas circunstancias, por ejemplo si quisiéramos derivar el algoritmo `raíz` a partir de ella.

Detección de potencias de 2

Dada la declaración de variables:

```
var
  x : Ent;
  r : Bool;
```

se trata de especificar un algoritmo que detecte si el valor de `x` es una potencia de 2, devolviendo en la variable `r` el valor `cierto` o `falso`, según sea el caso.

En la precondición sólo hemos de exigir que `x` tenga valor

```
{ x = X }
```

La postcondición es más interesante. Hemos de expresar que `r` ha de tener el mismo valor que un aserto, el aserto que se cumple cuando `x` es una potencia de 2. Supongamos que conocemos dicho aserto `P`, estaríamos tentados de escribir:

```
r = P
```

Sin embargo, en nuestro lenguaje para la construcción de asertos el igual (=) es una operación entre expresiones, y P no es una expresión. La solución consiste en recordar que toda expresión booleana es un aserto y que, en particular, `r` lo es. La equivalencia entre asertos la expresamos con la conectiva de doble implicación (↔):

```
{ r↔P ∧ x = X }
```

En cuanto a la forma de P, será algo como

```
∃i : Nat : x = 2^i
```

El problema es que no hemos incluido la exponenciación entre las operaciones disponibles para los enteros. Pero se expresa fácilmente usando el producto extendido

```
∃i : Nat : x = ( ∏j : 1≤ j ≤ i : 2 )
```

con lo que la especificación quedaría

```
var
  x : Ent;
  r : Bool;
{ x = X }
  esPotencia2?
{ r ↔ ∃i : Nat : x = ( ∏j : 1≤ j ≤ i : 2 ) ∧ x = X }
```

Máximo de un vector de enteros

Dadas las declaraciones de constantes y variables

```
cte
  N = ...;
var
```

```
  v : Vector[1..N] de Ent;
  m : Ent;
```

se trata de especificar un algoritmo que obtenga el máximo de los valores que se encuentran en el vector. Nótese que en la declaración, las constantes se nombran en mayúscula, ya que al igual que ocurre con las variables auxiliares de la especificación, el programa no las puede modificar.

En la precondición indicamos que el vector ha de tener componentes (para que exista un máximo) y estar todas ellas inicializadas. En la postcondición utilizamos la expresión extendida máximo para indicar la condición

```
cte
  N = ...;
var
  v : Vector[1..N] de Ent;
  m : Ent;
{ N ≥ 1 ∧ v = V } (* v=V expresa que todas las componentes de v tienen
valor *)
  máximo
{ m = max i : 1 ≤ i ≤ N : v(i) ∧ v = V }
```

Modificación de un vector de enteros

Partiendo de las declaraciones

```
cte
  N = ...;
var
  v : Vector[1..N] de Ent;
  x, y : Ent;
```

hemos de especificar un programa que sustituya todas las apariciones de `x en v` por `y`.

En la especificación sólo hemos de indicar que las variables de entrada tengan valor

```
N ≥ 1 ∧ v = V ∧ x = X ∧ y = Y
```

Podríamos cambiar la especificación para aceptar `N = 0`.

En la postcondición debemos indicar para cada componente del vector que si a la entrada tenía valor `x` a la salida debe tener valor `y`

```
∀i : 1 ≤ i ≤ N : (V(i) = X → v(i) = Y)
```

Pero con esto no es suficiente, porque debemos indicar además que las componentes distintas de **x** conservan su valor. Para ello debemos reforzar la postcondición con una fórmula similar a la anterior

```
∀i : 1 ≤ i ≤ N : (V(i) = X → v(i) = Y) ∧
∀i : 1 ≤ i ≤ N : (V(i) ≠ X → v(i) = V(i))
```

utilizando las leyes de equivalencia de los cuantificadores y forzando a que **x** e **y** conserven su valor llegamos a

```
cte
  N = ...;
var
  v : Vector[1..N] de Ent;
  x, y : Ent;
{ N ≥ 1 ∧ v = V ∧ x = X ∧ y = Y }
  Reemplazar
{ ∀i : 1 ≤ i ≤ N : [(V(i) = X → v(i) = Y) ∧ (V(i) ≠ X → v(i) = V(i))] ∧
  x = X ∧ y = Y }
```

División entera y resto

En el apartado donde se introdujeron las variables auxiliares de la especificación se presentó ya la especificación de la división entera, pero en aquel ejemplo era para operandos de tipo natural. Ahora, en cambio, partimos de las declaraciones

```
var x, y, c : Ent;
```

El problema es que ahora el resto puede no ser menor que el divisor, debido a los signos. Podríamos escribir una especificación donde se reflejasen las distintas posibilidades de combinación de signos de los operandos, utilizando la conectiva de implicación. Sin embargo, sería más natural si en la especificación pudiésemos utilizar la operación valor absoluto, que, desgraciadamente no se encuentra entre las operaciones disponibles para el tipo primitivo Ent. ¿Cómo podríamos incorporar la operación valor absoluto a nuestro lenguaje de asertos? Como ya dijimos al enumerar las operaciones disponibles, para incluir una nueva sólo hace falta *especificarla totalmente*. Hagámoslo:

```
var x, y : Ent;
{ x = X }
  absoluto
{ (X ≥ 0 → y = X) ∧ (X < 0 → Y = (-1) * X) ∧ x = X }
```

A partir de este momento, ya podemos utilizar la operación `abs` en los asertos que escribamos, con lo que la especificación de la división entera queda

```
var x, y, c : Ent;
{ x = X ∧ y = Y ∧ Y ≠ 0 }
  división
{ X = Y * c + R ∧ 0 ≤ abs(R) ≤ abs(Y) ∧ x = X ∧ y = Y }
```

Nótese cómo aquí introducimos una variable auxiliar de la especificación –R– para expresar una condición que debe cumplir una variable de programa –c–. En este caso la variable auxiliar sirve para representar a un valor que no viene dado por el valor de una variable de programa en ningún estado. La inclusión de variables auxiliares nuevas en la postcondición debe ser una notación abreviada para la inclusión de existenciales:

```
∃r : Ent : X = Y * c + r
```

De forma similar el módulo quedará

```
var x, y, r : Ent;
{ x = X ∧ y = Y ∧ Y ≠ 0 }
  módulo
{ X = Y * C + r ∧ 0 ≤ abs(r) ≤ abs(Y) ∧ x = X ∧ y = Y }
```

A partir de este punto podemos utilizar las operaciones `div` y `mod` en nuestros asertos, con el significado dado por estas especificaciones.

Es habitual que en la construcción de una especificación tengamos que especificar operaciones auxiliares, esto no tiene que implicar necesariamente que en la implementación también debamos realizarlas.

Propiedades de una especificación pre/post

Podemos utilizar las especificaciones pre/post para obtener el programa que las cumple –derivación– o para demostrar que un cierto programa las cumple –verificación–. En cualquier caso necesitaremos "razonar con las especificaciones".

El método de razonamiento consistirá en aplicar ciertas reglas que nos permiten asegurar que son correctas ciertas especificaciones, a partir de otras especificaciones correctas y relaciones de fuerza entre los asertos que las forman. Las reglas estarán compuestas por *premisas* y *conclusiones* separadas por una línea de quebrado de la siguiente forma:

$$\frac{\textit{premisas}}{\textit{conclusiones}}$$

El sentido de una regla de este estilo es que si se cumplen las premisas, también se cumple la conclusión. En las premisas y conclusiones aparecerán símbolos que representarán asertos y acciones genéricos.

Hay ciertas propiedades que cumplen los asertos en virtud únicamente de su definición y características. Podemos presentar estas propiedades como *reglas de verificación básicas*, pues nos servirán en el proceso de verificación de un algoritmo, o en la obtención de otras reglas.

Dados los asertos P, P', P_1, P_2, Q, Q', Q_1 y Q_2 y la acción algorítmica A se tienen las siguientes reglas de verificación básicas

— Reforzamiento de la precondición

$$\frac{\{ P' \}\ A\ \{Q\ \} \quad P \Rightarrow P'}{\{ P \}\ A\ \{ Q \}}$$

— Debilitamiento de la postcondición

$$\frac{\{ P \}\ A\ \{Q\ \} \quad Q \Rightarrow Q'}{\{ P \}\ A\ \{ Q' \}}$$

— Conjunción en la postcondición

$$\frac{\{ P \}\ A\ \{Q_1\ \} \quad \{ P \}\ A\ \{Q_2\ \}}{\{ P \}\ A\ \{ Q_1 \wedge Q_2 \}}$$

— Disyunción en la precondición

$$\frac{\{ P_1 \}\ A\ \{Q\ \} \quad \{ P_2 \}\ A\ \{Q\ \}}{\{ P_1 \vee P_2 \}\ A\ \{ Q \}}$$

— Precondición falsa

$$\frac{}{\{ \text{falso} \}\ A\ \{ Q \}}$$

— Postcondición falsa

$$\frac{P \Leftrightarrow \text{falso}}{\{ P \}\ A\ \{ \text{falso} \}}$$

Para demostrar estas reglas basta con aplicar las definiciones de especificación y de fuerza de los asertos.

Estas reglas se suelen aplicar de atrás adelante, para probar que se cumple la conclusión probamos que se cumplen las premisas. Por ejemplo, aplicando la regla de la disyunción en la postcondición, para demostrar la corrección de un algoritmo para la especificación:

```
var x, y : Ent;
{ x = X ∧ y = Y }
  sumaYDoble
{ x = 2 * X ∧ y = X + Y }
```

podemos demostrar la corrección de ese algoritmo con respecto a las especificaciones:

```
{ x = X ∧ y = Y }  sumaYDoble { x = 2 * X }
```

y

```
{ x = X ∧ y = Y }  sumaYDoble { y = X + Y }
```

Las leyes de equivalencia junto con estas reglas proporcionan un *cálculo* para razonar con predicados y especificaciones, alternativo al razonamiento en términos de estados. Las leyes se toman como *axiomas* del cálculo y las reglas como un *procedimiento de deducción* para obtener nuevas leyes. Se demuestra que dicho cálculo es *correcto*, lo que quiere decir que toda equivalencia deducida por él es válida en la lógica de predicados. Sin embargo, cuando se admiten dominios de valores cualesquiera, el cálculo no es *completo*, lo que significa que no toda equivalencia válida en la lógica de predicados es deducible mediante el cálculo.

Estas reglas no forman un método completo de verificación de algoritmos. Necesitamos poder entrar en la forma de las acciones A para poder llevar a cabo la verificación. Como veremos en el apartado dedicado a las estructuras algorítmicas básicas.

Otro método de razonamiento sobre las especificaciones radica en el uso del transformador de predicados *precondición más débil* que nos permitirá razonar sobre la corrección de los programas utilizando solamente asertos –con los que razonaremos utilizando el cálculo de predicados–.

Precondición más débil

Las reglas de verificación básicas son un mecanismo incompleto para demostrar la corrección de los programas; vamos a introducir otro concepto que nos permitirá razonar dentro del ámbito de la lógica de predicados de primer orden.

Supongamos que nos encontramos con una acción algorítmica A y queremos comprobar si cumple la especificación formada por P y Q como precondición y postcondición, respectivamente, con unas ciertas declaraciones que omitimos. Esto es, hemos de comprobar:

```
{ P }  A  { Q }
```

La idea de la verificación es proceder de atrás hacia delante. La especificación se cumplirá cuando partiendo de un estado que cumpla P se llegue a un estado que cumpla Q. Vamos a pre-

guntarnos cómo son los estados tales que partiendo desde ellos y ejecutando A se llega a Q. Estos estados se caracterizarán por unas condiciones, por un aserto R que cumplirá:

$\{ R \}\ A\ \{ Q \}$

En este punto, si resulta que P es más fuerte que R, puede aplicarse la regla de reforzamiento de la precondición y obtener así que el algoritmo es correcto.

Analizaremos para cada posible acción A de un lenguaje concreto cuál es ese aserto R. Antes formalizaremos este concepto y estudiaremos algunas de sus propiedades.

Dadas una acción A y un aserto Q, definimos la precondición más débil de A inducida por Q, que notaremos

pmd(A, Q)

como el aserto que cumple

$est[pmd(A, Q)] = \{ \sigma \mid A \text{ iniciado en } \sigma \text{ termina en un estado } \sigma' \in est[Q]$

Nótese que la precondición así definida es la más débil posible pues incluye a **todos** los estados que cumplen la propiedad de ser iniciales de A para obtener Q.

Aplicando la definición de especificación pre/post y la de precondición más débil tenemos que

---

Dados una acción A y un aserto Q se cumple

$\{ pmd(A, Q) \}\ A\ \{ Q \}$

Si se cumple lo anterior y aplicando la regla de reforzamiento de la precondición obtenemos la propiedad básica de las precondiciones más débiles

---

Dados los asertos P y Q, y una acción A se tiene la siguiente regla de verificación:

$$\frac{P \Rightarrow pmd(A, Q)}{\{ P \}\ A\ \{ Q \}}$$

Esta es la propiedad que nos permite razonar sobre la corrección de los programas en términos de fórmulas lógicas exclusivamente –una vez obtenida la precondición más débil–.

Las precondiciones más débiles cumplen una serie de propiedades que se pueden presentar como su definición axiomática [Bal93] o como proposiciones que se demuestran utilizan las definiciones de fuerza de los asertos y de especificación pre/post. Además, en el problema 12 se pide que se explique el significado operacional de estas propiedades –incluida la propiedad básica–. Estas propiedades son:

— Monotonía:

$$\frac{Q_1 \Rightarrow Q_2}{pmd(\ A, Q_1\ ) \Rightarrow pmd(\ A, Q_2\ )}$$

— Exclusión de milagros:

$$pmd(\ A, \text{Falso}) \Leftrightarrow \text{Falso}$$

— Distributividad con respecto a $\wedge$:

$$pmd(\ A, Q_1 \wedge Q_2\ ) \Leftrightarrow pmd(\ A, Q_1\ ) \wedge pmd(\ A, Q_2\ )$$

— Distributividad con respecto a $\vee$:

$$pmd(\ A, Q_1 \vee Q_2\ ) \Leftarrow pmd(\ A, Q_1\ ) \vee pmd(\ A, Q_2\ )$$

Como ejemplo veamos la demostración de la distributividad con respecto a $\wedge$:

En la dirección $\Rightarrow$

Sea σ un estado cualquiera que satisface

$\text{pmd}(\ A, Q_1 \wedge Q_2\ )$

por tanto, A iniciado en σ termina en un estado σ' de

$\text{est}[Q_1 \wedge Q_2] = \text{est}[Q_1] \cap \text{est}[Q_2]$

como $\sigma' \in \text{est}[Q_1]$ entonces σ satisface

$\text{pmd}(\ A, Q_1)$

y como $\sigma' \in \text{est}[Q_2]$ entonces σ satisface

$\text{pmd}(\ A, Q_2)$

por tanto σ satisface

$\text{pmd}(\ A, Q_1\ ) \wedge \text{pmd}(\ A, Q_2\ )$

y al ser σ un estado cualquiera que satisface $\text{pmd}(\ A, Q_1 \wedge Q_2\ )$ se tiene

$\text{pmd}(\ A, Q_1 \wedge Q_2\ ) \Rightarrow \text{pmd}(\ A, Q_1\ ) \wedge \text{pmd}(\ A, Q_2\ )$

En la otra dirección ($\Leftarrow$) el razonamiento es reversible, fijándonos en que si σ comienza en un estado que cumple $\text{pmd}(\ A, Q_1\ ) \wedge \text{pmd}(\ A, Q_2\ )$ termina en un estado σ' que satisface $Q_1 \wedge Q_2$, con lo que se tiene

$\text{pmd}(\ A, Q_1 \wedge Q_2\ ) \Leftarrow \text{pmd}(\ A, Q_1\ ) \wedge \text{pmd}(\ A, Q_2\ )$

Aunque estas propiedades nos resulten de utilidad, necesitamos ser capaces de obtener la precondición más débil para cualquier acción algorítmica escrita en un determinado lenguaje, para lo cual debemos fijar el lenguaje que vamos a utilizar para escribir los programas y determinar para cada una de sus instrucciones cómo se obtiene la precondición más débil inducida por una postcondición dada. De esto se ocupa el siguiente apartado de este tema.

1.1.3 Especificaciones informales

Aunque las especificaciones pre/post constituyen un mecanismo preciso y conciso de describir el comportamiento de los programas, no debemos pensar que sustituyen por completo a la documentación de los mismos. Un poco de lenguaje natural ayuda a completar la información que proporciona una especificación y facilita la comprensión de los programas.

Un posible esquema de documentación de una acción:

— Nombre de la acción.

— Parámetros o variables de entrada

— Parámetros o variables de salida

— Parámetros o variables de entrada y salida

— Precondición. Una explicación de los asertos que la componen.

— Efecto. Una descripción del efecto resultante de ejecutar la acción, en términos de las entradas y las salidas.

— Postcondición. Una explicación de los asertos que la componen.

— Excepciones. Consideraciones adicionales sobre la parcialidad o carácter estricto de la acción.

Las metodologías de diseño que se estudian en Ingeniería del Software suelen incluir normas sobre la documentación.

1.2 Verificación de algoritmos

Ha llegado el momento de empezar a implementar los programas. ¿Qué lenguaje de programación vamos a utilizar? Con el objetivo de que nuestro estudio sea lo más general posible utilizaremos un lenguaje, que no se corresponde con ningún lenguaje de programación concreto, pero que incluye los elementos fundamentales comunes a los lenguajes de programación imperativa más habituales. Este lenguaje imperativo genérico que permite expresar operaciones algorítmicas es lo que denominamos un *lenguaje algorítmico*.

El lenguaje algorítmico incluye todos los tipos de valores que pueden aparecer en las especificaciones y sobre los que se pueden aplicar las mismas operaciones.

Como indicamos al final del apartado anterior, para cada una de las instrucciones de nuestro lenguaje vamos a dar una regla de verificación que permite verificar la corrección de un programa compuesto únicamente por la instrucción en cuestión. Esa regla de verificación se obtendrá en cada caso a partir de un axioma que indique una utilización correcta de cada instrucción con respecto al significado de especificación Pre/Post. Estos axiomas definen formalmente el significa-

do de cada instrucción al indicar cómo se comporta respecto de los asertos que se cumplen en el programa. Constituyen lo que se denomina una *semántica axiomática*. La obtención de estos axiomas se justificará mediante la obtención de la precondición más débil inducida por una postcondición, es decir, preguntándonos ¿cuál es la condición más débil que debe cumplir –qué es lo mínimo que debemos exigir– el estado inicial para que al ejecutar la instrucción se acabe en un estado que cumpla la postcondición.

1.2.1 Estructuras algorítmicas básicas

La instrucción `seguir`

Su sintaxis es

```
seguir
```

Intuitivamente el significado de **`seguir`** es no hacer nada.

La obtención de la pmd resulta sencilla considerando que **`seguir`** no modifica el estado del cómputo, luego ¿qué han de cumplir los estados iniciales para que después de ejecutar la instrucción seguir el estado resultante cumpla un aserto Q? pues precisamente Q:

$$pmd(\text{seguir}, Q) \Leftrightarrow Q$$

El axioma indica cómo ha de ser la precondición para, al ejecutar la instrucción, obtener una postcondición.

$$\{ Q \}$$
$$\text{seguir}$$
$$\{ Q \}$$

Y, por último, la regla de verificación nos da una visión práctica del axioma, orientada a su uso en un método que emplee razonamiento hacia atrás:

$$\frac{P \Rightarrow Q}{\{ P \}\ \text{seguir}\ \{ Q \}}$$

Por ejemplo, para demostrar la corrección de

```
var x, y : Ent;
{ P : x ≥ 1 }
  seguir
{ Q : x ≥ 0 }
```

es necesario probar que

$x \geq 1 \Rightarrow x \geq 0$

lo cual es trivialmente cierto.

La asignación simple

Su sintaxis es

```
x := E
```

donde E es una expresión del mismo tipo que la variable x. Intuitivamente, se evalúa la expresión E y se asigna su valor a la variable x, con lo que se modifica el estado –siempre que el resultado de la expresión tenga un valor distinto al valor original de x–.

¿Qué debe cumplir el estado inicial para llegar a un estado que cumpla Q? La parte interesante de Q es la que hace referencia a la *x*, pues la parte que no haga referencia simplemente se debe mantener. En cuanto a la parte que se refiere a la *x*, debemos darnos cuenta de que sea lo que sea que el aserto Q dice sobre *x*, el aserto inicial debe decirlo sobre el valor que se le ha asignado a la *x*, es decir, E. Eso se expresa como

$Q\ [x/E]$

Por otra parte, hemos de garantizar que la expresión E puede evaluarse, y por ello un estado inicial ha de cumplir también:

$def(E)$

con lo que la precondición más débil queda

$pmd(\ x := E, Q\ ) \Leftrightarrow def(E) \wedge Q\ [x/E]$

El axioma que indica cómo ha de ser la precondición para obtener una postcondición dada

{ def(E) ∧ Q [x/E] }
x := E
{ Q }
y la regla de verificación

$$\frac{P \Rightarrow def(E) \wedge Q\,[x/E]}{\{\ P\ \}\ x := E\ \{\ Q\ \}}$$

Veamos un ejemplo de verificación de la asignación (es el ejercicio 16)

```
var x, y : Ent;
{ P: x ≥ 2 ∧ 2*y > 5 ∧ x = X ∧ y = Y }
  x := 2*x + y - 1
{ Q: x - 3 > 2 }
```

aplicando la regla de verificación de la asignación, hemos de probar P ⇒ def(E) ∧ Q [x/E]

La expresión esta definida porque no contiene operaciones parciales y están declaradas las variables que contiene

def(2*x + y – 1) ⇔ 2*x + y – 1 : Ent ⇐ x:Ent ∧ y:Ent ⇐ P

Para la segunda parte hemos de realizar la sustitución

$$\begin{aligned} & & x - 3 > 2\ [x/2*x + y - 1] \\ \text{sustitución} & \Leftrightarrow & 2*x + y - 1 - 3 > 2 \\ \text{álgebra} & \Leftrightarrow & 2*x + y > 6 \\ \text{álgebra} & \Leftarrow & x \geq 2 \wedge y \geq 3 \\ \text{álgebra} & \Leftarrow & x \geq 2 \wedge 2*y > 5 \\ \text{fuerza} & \Leftarrow & x \geq 2 \wedge 2*y > 5 \wedge x = X \wedge y = Y \\ & \Leftrightarrow & P \end{aligned}$$

Obsérvese cómo el objetivo de las manipulaciones algebraicas es ir acercándonos a la forma de la precondición. En muchas ocasiones, cuando def(E) sea trivialmente cierto omitiremos su demostración.

Un aspecto básico del método de verificación que estamos presentando es que la pmd, el axioma y la regla de la asignación sólo son válidos si no existen efectos colaterales, donde

Definimos los efectos colaterales como aquellas modificaciones en el estado de cómputo de un programa no provocadas por el significado de las instrucciones.

Un efecto colateral sería, por ejemplo, que una asignación a una variable *x* modificase no sólo el valor de *x* (el significado de la instrucción) sino que también modificase el de otra variable *y*. Nuestro lenguaje algorítmico carece de efectos colaterales, pero no así la mayoría de los lenguajes de programación donde es una cuestión de disciplina evitar que se produzcan. Sin embargo, en ocasiones los efectos colaterales son necesarios si no se quiere degradar la eficiencia de manera intolerable: compartición de estructura con punteros vs. copia de las estructuras de datos para evitar la compartición. De hecho, en la segunda parte del curso realizaremos implementaciones de los tipos abstractos de datos que tienen efectos colaterales.

La asignación múltiple

Su sintaxis es

```
<x1, ... , xm> := <E1, ... , Em>
```

---

siendo las x_i variables distintas entre sí, y las E_i expresiones de los mismos tipos que los de las variables x_i, para $i \in \{1,...,m\}$. Intuitivamente, se evalúan las expresiones E_i y se modifica el estado de forma simultánea, asignando a cada variable x_i el valor de la correspondiente expresión E_i. La simultaneidad es relevante cuando en las E_i interviene alguna de las x_i.

La obtención de la pmd es una extensión de lo realizado en la asignación simple: hay que garantizar la definición de las expresiones que se asignan, y que los estados iniciales cumplen la condición expresada por la postcondición en la que cada variable x_i se sustituye por la correspondiente expresión E_i.

$$pmd(<x_1, ... , x_m> := <E_1, ... , E_m>, Q) \Leftrightarrow def(E_1) \wedge ... \wedge def(E_m) \wedge Q[x_1/E_1, ..., x_m/E_m]$$

el axioma que indica la forma de la precondición para obtener la postcondición

$$\{ def(E_1) \wedge ... \wedge def(E_m) \wedge Q[x_1/E_1, ..., x_m/E_m] \}$$
$$<x_1, ... , x_m> := <E_1, ... , E_m>$$
$$\{ Q \}$$

y la regla de verificación que se obtiene aplicando la propiedad básica de las pmd.

$$\frac{P \Rightarrow def(E_1) \wedge ... \wedge def(E_m) \wedge Q[x_1/E_1, ..., x_m/E_m]}{\{ P \} <x_1, ... , x_m> := <E_1, ... , E_m> \{ Q \}}$$

Veamos un ejemplo con el intercambio de los valores de dos variables

---

```
var x, y : Ent;
{ P: x = X ∧ y = Y }
  <x,y> := <y,x>
```

```
{ Q: x = Y ∧ y = X }
```

las expresiones están definidas al no contener operaciones parciales y estar declaradas las variables

$def(x) \wedge def(y) \Leftrightarrow x{:}Ent \wedge y{:}Ent \Leftarrow P$

y la postcondición con la sustitución

$$\begin{aligned} & x = Y \wedge y = X\ [x/y, y/x] \\ \text{sustitución} \quad & \Leftrightarrow \quad y = Y \wedge x = X \\ \text{boole} \quad & \Leftrightarrow \quad x = X \wedge y = Y \\ & \Leftrightarrow \quad P \end{aligned}$$

Asignaciones a componentes de vectores

Permitimos la asignación a componentes de vectores empleando la sintaxis de una asignación simple o compuesta, donde se utiliza la notación

$v(E_i)$

para indicar la componente del vector v en la posición del valor de E_i. Si $v(E_i)$ aparece a la izquierda del símbolo := se interpreta como un cambio de valor de esa componente, y si aparece en otro lugar se considera una inspección del valor de la componente.

Sin embargo, esto sólo es una comodidad sintáctica y la asignación a componentes de vectores no significa que cada componente del vector se pueda considerar como una variable independiente. La variable independiente es el vector como un todo y lo que en realidad tiene sentido es la modificación del valor del vector y no el de una de sus componentes.

Considerar las componentes de los vectores como variables independientes conduce a efectos negativos con respecto a la verificación de los algoritmos. En primer lugar, la regla de verificación de la asignación sería incompleta, no pudiendo demostrar la corrección de programas que sí lo son, como se aprecia en el siguiente ejemplo (es el ejercicio 21.a).

```
cte N = ...;  % Ent
var v : Vector [1..N] de Ent;
  i, j : Nat;
{ 1 ≤ i ≤ N ∧ i = j }
  v(i) := 0
{ v(j) = 0 }
```

aunque es intuitivamente correcto, no es posible verificarlo con la regla de la asignación, pues al realizar la sustitución en la postcondición queda

$v(j) = 0\ [\ v(i)/0] \Leftrightarrow v(j) = 0$

que no es más débil que la precondición.

Y, es más, la regla deja de ser correcta pues permite verificar programas erróneos (es el ejercicio 21.b, casi).

```
cte N = ...;  % Ent
var v : Vector [1..N] de Ent;
{ 1 ≤ v(2) ≤ N }
  v(v(2)) := 1
{ v(v(2)) = 1 }
```

que parece correcto y, de hecho se puede probar que lo es con la regla de verificación de la asignación simple

$$v(v(2)) = 1\ [v(v(2))/1]$$

$$\text{sustitución} \quad \Leftrightarrow \quad 1 = 1$$

$$\text{álgebra} \quad \Leftrightarrow \quad \text{cierto}$$

$$\text{fuerza} \quad \Leftrightarrow \quad 1 \leq v(2) \leq N$$

sin embargo, se puede encontrar un contraejemplo, ya que si partimos de un estado σ

$\sigma(v(1)) = 2$ y $\sigma(v(2)) = 2$

que cumple la precondición, se llega a un estado σ' con:

$\sigma'(v(1)) = 2$ y $\sigma'(v(2)) = 1$

que no verifica el aserto $v(v(2)) = 1$ pues

$val[v(v(2)) = 1, \sigma'] =$ *falso*

no siendo correcto el algoritmo con respecto a la especificación

---

Para que las cosas funcionen correctamente hemos de definir operaciones de acceso y modificación de las componentes de un vector, operaciones que tomen al vector completo como argumento.

---

inspección del valor de una componente:

valor(v, E_i) que normalmente abreviamos como $v(E_i)$

modificación del valor de una componente:

asignar(v, E_i, E) que abreviamos como $v(E_i) := E$

Los vectores verifican las ecuaciones

valor(asignar(v, i, E), i) = E

$i \neq j \Rightarrow$ valor(asignar(v, i, E), j) = valor(v, j)

que nos permite relacionar el comportamiento de ambas ecuaciones

---

Utilizando estas operaciones podemos obtener la precondición más débil de la asignación a componentes de vectores utilizando un razonamiento similar al empleado en las asignaciones simples, pero considerando que se asigna todo el vector

---

$pmd(\ v(E_i) := E, Q) \Leftrightarrow def(E) \wedge enRango(v, E_i) \wedge Q[v/asignar(v, E_i, E)]$

Nótese que el resultado de la operación asignar es un vector.

Además de exigir que la expresión que se asigna esté definida, también pedimos que la expresión cuyo valor determina el índice de la componente a modificar cumpla la condición *enRango*. Este aserto incluye la definición de la expresión E_i y solicita que su valor esté en el rango de valores que sirven de índice al vector v. En caso de conocer el rango, el aserto enRango(v, E_i) puede sustituirse por

---

$def(E_i) \wedge (li \leq E_i \leq ls)$

siendo [li..ls] el rango del vector v.

---

el axioma define las condiciones que han de cumplir los estados de entrada

---

$\{ def(E) \wedge enRango(v, E_i) \wedge Q[v/asignar(v, E_i, E)] \}$

$v(E_i) := E$

$\{ Q \}$

y la regla de verificación, aplicando la propiedad básica de las pmd

$$\frac{P \Rightarrow def(E) \wedge enRango(v, E_i) \wedge Q[v/asignar(v, E_i, E)]}{\{ P \}\ v(E_i) := E\ \{ Q \}}$$

Con esta nueva regla sí es posible verificar el ejemplo que antes no

---

```
cte N = ...;  % Ent
var v : Vector [1..N] de Ent;
  i, j : Nat;
{ 1 ≤ i ≤ N ∧ i = j }
  v(i) := 0
{ v(j) = 0 }
```

la precondición garantiza que la expresión que sirve de índice es válida

$$enRango(v, i) \Leftrightarrow 1 \leq i \leq N \wedge def(i)$$
$$\Leftarrow P$$

la expresión que se asigna está definida

$$def(0) \Leftrightarrow cierto$$
$$\Leftarrow P$$

la postcondición con la sustitución se obtiene de la precondición

$$
\begin{array}{rcl}
 & & v(j) = 0\ [v/asignar(v, i, 0)] \\
\text{vectores} & \Leftrightarrow & valor(v, j) = 0\ [v/asignar(v, i, 0)] \\
\text{sustitución} & \Leftrightarrow & valor(asignar(v, i, 0), j) = 0 \\
 & \Leftarrow & valor(asignar(v, i, 0), i) = 0 \wedge i = j \\
\text{ecuación} & \Leftrightarrow & 0 = 0 \wedge i = j \\
\text{fuerza} & \Leftarrow & P
\end{array}
$$

1.2.2 Estructuras algorítmicas compuestas

Las acciones que vamos a ver a continuación permiten combinar otras instrucciones para conseguir acciones más complejas. Veremos tres tipos de composición que son las que caracterizan la programación imperativa estructurada:

- Composición secuencial
- Composición alternativa
- Composición iterativa

Los lenguajes de programación suelen incluir más de una instrucción para cada uno de estos esquemas, pero, como veremos, las instrucciones de nuestro lenguaje algorítmico permiten construir acciones que representen a cualquier construcción disponible en un lenguaje imperativo estructurado. Al reducir el número de instrucciones facilitamos la verificación puesto que tenemos que tratar con menos reglas.

Composición secuencial

Su sintaxis es

A_1 ; A_2

siendo A_1 y A_2 acciones expresadas en el lenguaje algorítmico.

Intuitivamente, el comportamiento es que se ejecuta la acción A_1 y a su terminación (si se produce) se ejecuta la acción A_2.

Para obtener la pmd empleamos el concepto de razonamiento hacia atrás. Para llegar a una postcondición Q hemos de partir de un estado que cumpla la pmd de A_2 inducida por Q y, a su vez, para alcanzar ese estado intermedio hemos de partir de un estado que cumpla la pmd de A_1 inducida por la pmd(A_2, Q):

$$pmd(A_1;A_2, Q) \Leftrightarrow pmd(A_1, pmd(A_2, Q))$$

el axioma se obtiene de la pmd

$\{ pmd(A_1, pmd(A_2, Q)) \}$
$A_1 ; A_2$
$\{ Q \}$

La regla de verificación supone una relajación del axioma al permitir utilizar un aserto intermedio R cualquiera que garantice que:

— Un estado que cumpla R, tomado como entrada de A_2, produce un estado que cumple la postcondición

— La ejecución de A_1 con entrada en un estado que cumple la precondición produce un estado que cumple R

La pmd de la segunda acción es un aserto intermedio válido, pero la regla de verificación permite considerar otros

---

$$\frac{\{ P \}\ A_1\ \{ R \} \quad \{ R \}\ A_2\ \{ Q \}}{\{ P \}\ A_1 ; A_2\ \{ Q \}}$$

la obtención de esta regla se puede justificar por las propiedades de las pmd

— Por la definición de pmd a partir de $\{ P \}\ A_1\ \{ R \}$

$$P \Rightarrow pmd(A_1, R)$$

— Por la definición de pmd a partir de $\{ R \}\ A_2\ \{ Q \}$

$$R \Rightarrow pmd(A_2, Q)$$

— Por la propiedad de monotonía de las pmd, aplicada a la anterior relación, con respecto a la acción A_1 ($P \Rightarrow Q$ // $pmd(A,P) \Rightarrow pmd(A,Q)$)

$$pmd(A_1, R) \Rightarrow pmd(A_1, pmd(A_2, Q))$$

— Por transitividad de la fuerza de los asertos (ya que es una inclusión de conjuntos)

$$P \Rightarrow pmd(A_1, pmd(A_2, Q)) \ (\Leftrightarrow pmd(A_1;A_2, Q))$$

— Y por la propiedad básica de las pmd se obtiene la conclusión de la regla

$$\{ P \}\ \ A_1 ; A_2\ \{ Q \}$$

Veamos un ejemplo de verificación de la composición secuencial. Vamos a verificar un programa que, dada un cantidad positiva de dinero obtiene a cuántas monedas de 5 y 1 pesetas se corresponde.

```
var c, d, p : Ent;
```

```
%
%  c = cantidad, d = monedas de 5, p = monedas de 1
%
{ P: c = C ∧ C > 0 }
  d := c div 5;
  p := c mod 5
{ Q: C = d*5 + p ∧ 0 ≤ d < C ∧ 0 ≤ p < 5 ∧ c = C }
```

En este programa usamos las operaciones div y mod con el significado dado por la especificación que vimos en el apartado dedicado a los ejemplos de especificación.

Para aplicar la regla de la composición secuencial hemos de obtener el aserto intermedio, que en este caso será la pmd de la segunda asignación:

$$
\begin{array}{rcl}
 & & \text{pmd}(\ p := c \bmod 5,\ Q) \\
\text{definición} & \Leftrightarrow & \text{def}(c \bmod 5) \wedge Q\ [\ p/c \bmod 5] \\
\text{sustitución} & \Leftrightarrow & C = d*5 + c \bmod 5 \wedge 0 \le d < C \wedge 0 \le c \bmod 5 < 5 \wedge c = C \\
\text{álgebra} & \Leftrightarrow & C = d*5 + c \bmod 5 \wedge 0 \le d < C \wedge c \ge 0 \wedge c = C
\end{array}
$$

donde, en el primer paso aplicamos que c mod 5 está definido (cierto $\wedge$ S $\Leftrightarrow$ S); y en el último paso, hemos aplicado $0 \le c \bmod 5 < 5 \Leftrightarrow c \ge 0$ (hace falta para luego conseguir $C > 0$)

llamamos R al aserto obtenido

R: $C = d*5 + c \bmod 5 \wedge 0 \le d < C \wedge c \ge 0 \wedge c = C$

y aplicamos la regla de verificación para el aserto intermedio. Nótese que ya hemos probado que

{ R } p := c mod 5 { Q }

es correcta, pues R es la pmd(p:=c mod 5, Q). Nos queda probar

{ P } d := c div 5 { R }

aplicando la regla de verificación de la asignación tenemos que demostrar

— $P \Rightarrow \text{def}(\ c \text{ div } 5\ )$, que se cumple pues div está definido si el divisor es distinto de 0 y la variable está declarada

— $P \Rightarrow R\ [d/c \text{ div } 5]$

$$
\begin{array}{rcl}
 & & R\ [d/c \text{ div } 5] \\
\text{sustitución} & \Leftrightarrow & C = c \text{ div } 5*5 + c \bmod 5 \wedge 0 \le c \text{ div } 5 < C \wedge c \ge 0 \wedge c = C \\
\text{álgebra} & \Leftrightarrow & C = c \wedge 0 \le c \text{ div } 5 < C \wedge c \ge 0 \\
\text{fuerza} & \Leftarrow & P
\end{array}
$$

donde, en el último paso hemos aplicado

$$
\begin{array}{rcl}
C = c & \Rightarrow & C = c \\
C = c \wedge C > 0 & \Rightarrow & 0 \le c \text{ div } 5 < C \\
C = c \wedge C > 0 & \Rightarrow & c \ge 0
\end{array}
$$

Con esto hemos demostrado { P } d := c div 5 { R } y { R } p := c mod 5 { Q }, por lo que, aplicando la regla de verificación, queda demostrado

---

{ P } d := c div 5 ; p := c mod 5 { Q }

La introducción de instrucciones compuestas conduce al empleo de asertos intermedios que se emplean al realizar las verificaciones. Para facilitar la presentación de las pruebas escribimos *programas anotados* en los que se incluyen los asertos intermedios, indicando las condiciones que han de cumplir los estados en cada punto del programa. Con respecto a la composición secuencial, los programas anotados incluyen los asertos intermedios

---

{ P }
A_1 ;
{ R }
A_2 ;
{ Q }

La composición secuencial puede hacerse entre acciones del lenguaje cualesquiera, en particular entre otras composiciones secuenciales. Se cumple la propiedad de que la composición secuencial es un operador asociativo (la demostración se hace utilizando las propiedades de la pmd y es el ejercicio 26), es decir:

---

el programa
A_1 ; { A_2 ; A_3 }
es equivalente al programa
{ A_1 ; A_2 } ; A_3
y escribimos, simplemente
A_1 ; A_2 ; A_3

Con respecto a las pmd se cumple

---

pmd(A_1 ; A_2 ; A_3, Q) $\Leftrightarrow$ pmd(A_1, pmd(A_2 ; A_3, Q)) $\Leftrightarrow$ pmd(A_1, pmd(A_2 , pmd(A_3, Q)))
y, en general, la composición secuencial de *n* acciones:
pmd(A_1 ; ... ; A_n, Q) $\Leftrightarrow$ pmd(A_1, ... pmd(A_n, Q) ...)
y la regla de verificación extendida

$$\frac{\{ P \}\ A_1\ \{ R_1 \}\ \ldots\ \{ R_{n-1} \}\ A_n\ \{ Q \}}{\{ P \}\ A_1 ; \ldots ; A_n\ \{ Q \}}$$

Declaraciones locales

En ocasiones es necesario definir variables de apoyo para almacenar valores temporales de los algoritmos. Estas variables tienen una naturaleza local en tanto en cuanto su utilidad se limita a un ámbito de instrucciones reducido. Permitimos el uso de variables locales con la siguiente sintaxis:

---

var $x_1 : \tau_1$; ... ; $x_m : \tau_m$;
inicio
A
fvar

---

siendo las x_i variables nuevas, que no se han declarado previamente en el algoritmo, y A una acción del lenguaje algorítmico.

Intuitivamente, las variables locales inducen el siguiente comportamiento:

— se amplia el estado para incluir las nuevas variables

— con el estado ampliado, se ejecuta la acción A, que puede inicializar, acceder y modificar las variables locales

— al terminar la ejecución de A, las variables locales dejan de existir

---

Para obtener la pmd pensemos que la declaración local está después de la precondición y antes de la postcondición, por lo que, si las variables sólo existen dentro del ámbito indicado y son variables con nombres distintos a las que existían previamente, no pueden aparecer en la precondición ni en la postcondición. Por lo tanto la precondición se obtiene exclusivamente a partir de la acción A:

---

pmd(var ... inicio A fvar, Q) $\Leftrightarrow$ pmd(A, Q)

el axioma se obtiene directamente de la pmd

{ pmd(A, Q) }
var $x_1 : \tau_1$; ... ; $x_m : \tau_m$;
inicio
A
fvar
{ Q }

y la regla de verificación

$$\frac{\{ P \}\ A\ \{ Q \}}{\{ P \}\ \text{var ... inicio A fvar}\ \{ Q \}}$$

No todos los lenguajes de programación permiten definir variables locales en lugares arbitrarios del código. Lo que sí es más habitual es que se puedan definir en los subprogramas: procedimientos y funciones.

Composición alternativa

Es la instrucción condicional del lenguaje que permite definir una bifurcación en el flujo de control del programa. Su sintaxis es

---

si $B_1 \rightarrow A_1$
□ $B_2 \rightarrow A_2$
...
□ $B_m \rightarrow A_m$
fsi

---

siendo las B_i expresiones booleanas y las A_i acciones del lenguaje algorítmico. Cada una de las estructuras $B_i \rightarrow A_i$ se denomina *acción protegida* donde la acción A_i está protegida por la *barrera* B_i. Una barrera se dice que está *abierta* si evalúa a *cierto* y se dice que está cerrada en caso contrario.

Intuitivamente la composición alternativa funciona de la siguiente forma

---

— Se evalúan todas las barreras (por lo que éstas deben estar definidas)

— De entre las barreras abiertas, se elige una de modo *indeterminista*. Esta elección debe poder realizarse siempre (por lo que al menos una barrera debe estar abierta).

— Se ejecuta la acción asociada a la barrera seleccionada.

Para obtener la pmd hemos de considerar que todas las barreras deben estar definidas, al menos una de ellas abierta, y que si una barrera está abierta la ejecución de su acción asociada debe terminar en un estado que cumpla la postcondición. La pmd ha de garantizar estas condiciones:

$$\text{pmd}(\text{ si } B_1 \rightarrow A_1 \square \ldots \square B_m \rightarrow A_m \text{ fsi}, Q) \Leftrightarrow$$
$$\text{def}(B_1) \wedge \ldots \wedge \text{def}(B_m) \wedge [(B_1 \wedge \text{pmd}(A_1, Q)) \vee \ldots \vee (B_m \wedge \text{pmd}(A_m, Q))$$

Nótese que la disyunción de los asertos $B_i \wedge \text{pmd}(A_i, Q))$ garantiza que al menos una de las barreras está abierta y que su correspondiente acción conduce a un estado que cumple la postcondición.

---

el axioma

$\{ \text{pmd}(\text{ si } B_1 \rightarrow A_1 \square \ldots \square B_m \rightarrow A_m \text{ fsi}, Q) \}$

si $B_1 \rightarrow A_1$
$\square\ B_2 \rightarrow A_2$
...
$\square\ B_m \rightarrow A_m$
fsi
$\{ Q \}$

y la regla de verificación

$$\frac{\begin{array}{l} P \Rightarrow def(B_1) \wedge ... \wedge def(B_m) \\ P \Rightarrow B_1 \vee ... \vee B_m \\ \{ P \wedge B_1 \}\ A_1\ \{ Q \}\ ...\ \{ P \wedge B_m \}\ A_m\ \{ Q \} \end{array}}{\{ P \}\ \textbf{si}\ B_1 \rightarrow A_1 \square ... \square B_m \rightarrow A_m\ \textbf{fsi}\ \{ Q \}}$$

Veamos un ejemplo que ilustra el uso de la regla de verificación de la composición alternativa: el máximo de dos números (es el ejercicio 30, aunque allí se usa la operación **max**)

```
var x, y, z : Ent;
{ P: x = X ∧ y = Y }
    si x ≥ y z := x
     □ y ≥ x z := y
    fsi
{ Q: x = X ∧ y = Y ∧ ((z = X ∧ z ≥ Y) ∨ (z = Y ∧ z ≥ X)) }
```

— $P \Rightarrow def(x \geq y) \wedge def(y \geq x)$, lo cual se cumple ya que

$def(x \geq y) \wedge def(y \geq x) \Leftrightarrow$ cierto

al no incluir operaciones parciales y estar declaradas las variables de un tipo ordinal

— $P \Rightarrow x \geq y \vee y \geq x$, lo cual se cumple ya que

$x \geq y \vee y \geq x \Leftrightarrow$ cierto

por propiedades de los números enteros

— La corrección de cada acción en el supuesto de que la barrera esté abierta

4. $\{ P \wedge x \geq y \}\ z := x\ \{ Q \}$

que verificamos usando la regla de la asignación

$P \wedge x \geq y \Rightarrow def(x)$ al estar declarada la variable

$Q[z/x]$

sustitución $\Leftrightarrow$ $x = X \wedge y = Y \wedge ((x = X \wedge x \geq Y) \vee (x = Y \wedge x \geq X))$

Boole $\Leftrightarrow$ $x = X \wedge y = Y \wedge (x \geq Y \vee x = Y)$

Fuerza $\Leftarrow$ $x = X \wedge y = Y \wedge x \geq Y$

$\Leftrightarrow$ $P \wedge x \geq y$

5. $\{ P \wedge y \geq x \}\ z := y\ \{ Q \}$

que verificamos usando la regla de la asignación

$$P \wedge y \geq x \Rightarrow def(y) \quad \text{al estar declarada la variable}$$

$$Q[z/y]$$

$$\text{sustitución} \quad \Leftrightarrow \quad x = X \wedge y = Y \wedge ((y = X \wedge y \geq Y) \vee (y = Y \wedge y \geq X))$$

$$\text{Boole} \quad \Leftrightarrow \quad x = X \wedge y = Y \wedge (y = X \vee y \geq X)$$

$$\text{Fuerza} \quad \Leftarrow \quad x = X \wedge y = Y \wedge y \geq X$$

$$\Leftrightarrow \quad P \wedge y \geq x$$

Al igual que en la composición secuencial también podemos anotar la composición condicional con los asertos intermedios oportunos:

$\{ P \}$
si $B_1 \rightarrow \{ P \wedge B_1 \}\ A_1\ \{ Q \}$
$\square\ B_2 \rightarrow \{ P \wedge B_2 \}\ A_2\ \{ Q \}$
...
$\square\ B_m \rightarrow \{ P \wedge B_m \}\ A_m\ \{ Q \}$
fsi
$\{ Q \}$

El indeterminismo de nuestra composición secuencial no está presente en los lenguajes de programación que optan por la primera barrera abierta (Pascal) o que siguen todas las barreras abiertas (C). Si queremos que nuestros programas sean deterministas deberemos modificar las condiciones de las barreras para evitar que ciertos estados abran más de una barrera a la vez. Así por ejemplo en el anterior programa bastaría con cambiar el ≥ de la segunda barrera por un mayor estricto. Este indeterminismo [Bal93] es interesante ya que en el diseño algorítmico conviene no descartar posibilidades y tomar el mínimo de decisiones arbitrarias; así, si existen dos maneras de lograr una postcondición, el indeterminismo puede permitir que en el algoritmo consten ambas, aunque tal vez en un paso posterior de programación se descarte una de ellas.

La instrucción condicional que hemos presentado se corresponde con una versión extendida de la sentencia CASE que suelen incluir los lenguajes imperativos estructurados. La más habitual instrucción IF-THEN-ELSE se puede expresar en nuestro lenguaje como:

---

si B $\rightarrow$ A_1
□ NOT B $\rightarrow$ A_2
fsi

para facilitar su uso introducimos una nueva forma de la instrucción **si**

si B
entonces A_1

```
sino A2
fsi
```

Para verificar esta instrucción, se convierte a la forma general de composición alternativa y se aplica su regla de verificación.

Composición iterativa

La composición iterativa permite repetir la ejecución de un misma acción un cierto número de veces; un número de veces que depende del estado del programa. En el lenguaje algorítmico incluimos las siguiente instrucción iterativa, que se corresponde con el bucle *while* de los lenguajes de programación:

it B →
  A
fit

Operacionalmente se comporta de la siguiente forma:

— Se evalúa la condición de repetición B (que debe estar definida)

— Si B toma valor *falso*, se termina sin modificar el estado

— Si B es cierta, se ejecuta el cuerpo A

— Se repite el proceso a partir del nuevo estado obtenido.

Al pensar qué forma tiene la pmd surge el concepto de invariante. Supongamos la especificación de un bucle

```
{ pmd }
  it B →
       A
  fit
{ Q }
```

La idea es que dependiendo del estado de partida, el bucle se ejecutará un número diferente de veces, y todos esos estados deben satisfacer la pmd. Supongamos que partimos de un estado σ que satisface la pmd y que hace que el bucle se ejecute *n* veces y acabe en un estado que cumple Q; al ejecutarse la primera vez, termina en un estado σ_1 que servirá como entrada para la siguiente pasada por el bucle que después de ejecutarse n–1 veces acabará en un estado que cumpla Q, por lo que σ_1 también debe cumplir la pmd. Es decir, todos los estados intermedios que se alcanzan al final de cada pasada por el bucle deben cumplir la pmd pues de hecho se pueden ver como estados iniciales que después de un cierto número de pasadas *i* alcanzan la postcondición.

Al ejecutarse la última pasada por el bucle llegamos a un estado σ_n que al tomarse como entrada del bucle hará que no se cumpla la condición de repetición con lo que el bucle terminará. Pero aún así σ_n también debe cumplir la pmd pues se corresponde con la situación en que el bucle no se ejecuta ninguna vez y se alcanza directamente la postcondición. La conclusión es que la pmd es una propiedad que se debe cumplir antes y después de ejecutar cada pasada por el bucle; es una condición que permanece constante durante todas las iteraciones y que por esa razón se denomina *invariante* del bucle.

Por ejemplo,

```
<i, q, p > := <0, 0, 1>;
it i < n →
    i := i + 1;
    q := q + p;
    p := p + 2;
fit
```

los estados por los que van pasando las variables

estado	i	q	p
σ	0	0	1
σ_1	1	1	3
σ_2	2	4	5
σ_3	3	9	7
.	.	.	.
.	.	.	.
.	.	.	.

podemos observar que en todas las iteraciones las variables guardan una cierta relación entre sí, en este caso

$q = i^2 \wedge p = 2*i + 1$

Por lo tanto para encontrar la pmd debemos encontrar un aserto *I* que cumpla

(i.2) I garantiza que la condición de repetición se puede evaluar

$I \Rightarrow def(B)$

I se preserva al ejecutar cada vuelta del bucle

$\{ I \wedge B \}\ A\ \{ I \}$

(i.3) I garantiza que al terminar se cumple la postcondición

$I \wedge \neg B \Rightarrow Q$

Pero con esto no está todo, pues para que un programa sea correcto debe terminar y dado que el número de iteraciones depende del estado inicial, cabe la duda de si se alcanzará un estado que

haga falsa la condición de repetición. La pmd debe garantizar que el bucle termina, pero ¿cómo podemos asegurar la terminación de un bucle? La idea es que si un bucle termina es porque el número de veces que se ejecuta es finito; cada vez que se ejecuta una iteración, disminuye el número de iteraciones que resta por ejecutar.

Hemos de obtener una medida del número de vueltas que puede dar un bucle y garantizar que esa medida cumple dos requisitos:

— Decrece en cada vuelta

— No puede decrecer indefinidamente

Para conseguirlo utilizaremos una expresión que decrezca en cada vuelta y de la que se sepa que no puede decrecer indefinidamente: una expresión que tome valores en un dominio donde esté definida una relación de orden *bien fundamentada*.

Una relación de orden (parcial o total) se dice bien fundamentada si no existen cadenas decrecientes infinitas. (Una cadena decreciente es una sucesión de valores donde cada valor es menor que el anterior.)

Un dominio se dice bien fundamentado si en él existe un orden bien fundamentado.

La idea es que en un dominio bien fundamentado no podemos encontrar cadenas decrecientes infinitas.

Algunos ejemplos de dominios bien fundamentados:

— Todo dominio finito con un orden estricto (el orden estricto impide la repetición de los elementos de la cadena y el número finito de elementos hace que no puedan existir cadenas infinitas)

— Los naturales ($\aleph$) con el orden decreciente estricto habitual. (como mucho pueden llegar hasta 0)

Ejemplo de dominios que no son bien fundamentados

— Los reales ($\Re$) con respecto a un orden total. (podemos construir cadenas infinitas crecientes y decrecientes contenidas en cualquier intervalo de los reales)

— Los enteros (Z) con respecto a un orden total. (podemos construir cadenas infinitas crecientes y decrecientes.)

Volviendo a la pmd de la composición iterativa, para definir la pmd hemos de asegurar que

---

Existe una *expresión de acotación* C, que cumple los siguientes requisitos:

(c.1) Dentro del bucle, C está definida, toma valores en un dominio bien fundamentado y puede decrecer

$$I \wedge B \Rightarrow def(C) \wedge dec(C)$$

siendo dec(C) (que leemos "decrementable") un aserto que indica que C toma valores en un dominio bien fundamentado y que no es minimal (puede decrecer).

(c.2) Al ejecutar el cuerpo del bucle el valor de C decrece

$$\{ I \wedge B \wedge C = T \}\ A\ \{ C \prec T \}$$

donde T es un valor del dominio de C y $\prec$ es el orden bien fundamentado.

Nótese que en ocasiones será necesario incluir en el invariante condiciones que hagan referencia a la expresión de acotación, para poder así garantizar la terminación del bucle. con todo esto ya podemos escribir la pmd de la composición iterativa como

---

$$pmd(\text{ it } B \rightarrow A \text{ fit }) \Leftrightarrow I$$

siempre que existan I y C que verifiquen los requisitos (i.2), (i.3), (c.1) y (c.2).

(nótese que son c.1 y c.2 las que garantizan que el bucle termina)

el axioma se obtiene directamente de la pmd

Si existen un aserto I y una expresión C que cumplen los requisitos: (i.2), (i.3), (c.1) y (c.2), entonces

{ I }
it B →
A
fit
{ Q }

y, por fin, la regla de verificación, que también recoge que la precondición ha de ser más fuerte que la pmd:

(i.1) $P \Rightarrow I$

(i.2) $I \Rightarrow def(B)$

$\{ I \wedge B \}\ A\ \{ I \}$

(i.3) $I \wedge \neg B \Rightarrow Q$

(c.1) $I \wedge B \Rightarrow def(C) \wedge dec(C)$

(c.2) $\{ I \wedge B \wedge C = T \}\ A\ \{ C \prec T \}$

$\{ p \}$ it B → A fit $\{ Q \}$

siendo I un aserto y C una expresión

Veamos un ejemplo verificando un programa que calcula el producto de dos enteros mediante sumas. Es el ejercicio 36(a)

```
var x, y : Ent;
{ P: x = X ∧ y = Y ∧ y ≥ 0 }
  p := 0;
  it y ≠ 0 →
      p := p + x;
      y := y – 1
  fit
{ Q: p = X * Y }
```

Tenemos la composición de una asignación con un bucle. Para verificar esta composición necesitamos un aserto intermedio, y para encontrarlo tenemos dos alternativas:

— Deducir un aserto intermedio a partir de la precondición y el efecto de la asignación.

— Emplear la pmd del bucle como aserto intermedio

La complejidad es similar en ambos casos, y optamos por el primero de los métodos.

Sabemos que la precondición se mantiene pues en ella no se hace referencia a *p*; y sobre esta variable sabemos que después de la asignación su valor será cero; definimos pues un aserto intermedio *R*

R: x = X ∧ y = Y ∧ y ≥ 0 ∧ p = 0

A continuación vamos a verificar el bucle, procediendo de atrás hacia delante, tomando *R* como precondición de bucle.

Para aplicar la regla de verificación debemos obtener el invariante y la expresión de acotación.

Este algoritmo suma *x* a *p y* veces. La idea para obtener el invariante es observar que en cada pasada *p* se va acercando al valor X*Y incrementándose en *x*, hasta que al final, como se indica en la postcondición, alcanza p=X*Y. Así, en una iteración cualquiera

p + *algo* = X*Y

¿cuánto vale ese algo? El número de *x* que quedan por sumar a *p*, que son precisamente *y*.

p + x*y = X*Y

otra forma de obtenerlo sería considerar que, en una iteración cualquiera, se cumple que

p = x*(Y–y) ⇔ p = x*Y – x*y ⇔ p + x*y = x*Y

y como x = X en todos los estados ⇔ p + x*y = X*Y

otro posible invariante sería p + x*y = x*Y ∧ x = X, aunque supongo que para demostrar la corrección con él sería necesario completar la postcondición con x = X ¿o quizás no?

Por lo que se refiere a la terminación del bucle, parece que una buena expresión de acotación sería

---

C: y

---

ya que el bucle se ejecutará *y* veces, hasta que *y* tome el valor cero. Sin embargo, *y* es una variable entera, por lo que no toma valores en un dominio bien fundamentado; de hecho, si *y* tuviese un valor negativo el bucle no terminaría. Afortunadamente, el aserto *R* indica que y ≥ 0 con lo que la expresión de acotación es correcta. Aún así, es necesario que el invariante incluya esta condición para que así podamos demostrar las condiciones relacionadas con la terminación.

Con todo esto el invariante queda

I: p + x*y = X*Y ∧ y ≥ 0

y la expresión de acotación

C: y

---

pasamos entonces a demostrar las condiciones de la regla de verificación.

```
{ R: x = X ∧ y = Y ∧ y ≥ 0 ∧ p = 0 }
  it y ≠ 0 →
      p := p + x;
      y := y - 1
  fit
{ Q: p = X * Y }
```

(i.1) R ⇒ I

R ⇒ y ≥ 0

R ⇒ x = X ∧ y = Y ∧ p = 0

⇒ x*y = X*Y ∧ p = 0

⇒ 0 + x*y = X*Y ∧ p = 0

⇒ p + x*y = X*Y

donde se han aplicado propiedades del álgebra de los enteros, con lo que

R ⇒ p + x*y = X*Y ∧ y ≥ 0

⇔ I

(i.2) que pide en primer lugar I ⇒ def(B)

$$\begin{aligned} \text{def}(\, y \neq 0) &\Leftrightarrow \text{cierto} \\ &\Leftarrow I \end{aligned}$$

Además, hay que probar

$\{ I \wedge B \}$

```
p := p + x;
y := y - 1
```

$\{ I \}$

para lo cual aplicamos la regla de la composición secuencial, obteniendo la pmd de la segunda asignación inducida por la postcondición

$$\begin{aligned} & \text{def}(y-1) \wedge I[y/y-1] \\ \text{sustitución} \quad &\Leftrightarrow \text{cierto} \wedge p + x*(y-1) = X*Y \wedge y-1 \geq 0 \\ \text{álgebra} \quad &\Leftrightarrow p + x*y - x = X*Y \wedge y-1 \geq 0 \end{aligned}$$

que llamaremos R_2. Ahora hallamos las pmd de la primera asignación inducida por R_2.

$$\begin{aligned} & \text{def}(p+x) \wedge R_2[p/p+x] \\ \text{sustitución} \quad &\Leftrightarrow \text{cierto} \wedge p + x + x*y - x = X*Y \wedge y-1 \geq 0 \\ \text{álgebra} \quad &\Leftrightarrow p + x*y = X*Y \wedge y-1 \geq 0 \end{aligned}$$

que llamaremos R_1. Ahora sólo nos queda comprobar que la precondición implica a R_1

$$I \wedge B \Rightarrow R_1$$

lo cual se obtiene, aplicando propiedades del álgebra de los enteros, y razonando por partes de R_1

$$\begin{aligned} I &\Rightarrow p + x*y = X*Y \\ I \wedge B &\Rightarrow y \geq 0 \wedge y \neq 0 \\ &\Leftrightarrow y > 0 \\ &\Leftrightarrow y \geq 1 \\ &\Leftrightarrow y-1 \geq 0 \end{aligned}$$

(i.3) $I \wedge \neg B \Rightarrow Q$. Lo cual se obtiene por el siguiente razonamiento

$$\begin{aligned} I \wedge \neg B &\Rightarrow p + x*y = X*Y \wedge y = 0 \\ &\Leftrightarrow p + 0 = X*Y \\ &\Leftrightarrow p = X*Y \\ &\Leftrightarrow Q \end{aligned}$$

(c.1) $I \wedge B \Rightarrow \text{def}(C) \wedge \text{dec}(C)$

$$\begin{aligned} \text{def}(y) &\Leftrightarrow \text{cierto} \\ &\Leftarrow I \wedge B \end{aligned}$$

$$\begin{aligned} \text{dec}(y) &\Leftrightarrow y > 0 \\ &\Leftrightarrow y \geq 0 \wedge y \neq 0 \end{aligned}$$

$$\Leftarrow \quad I \wedge B$$

(c.2) Hay que probar la corrección de

$\{ I \wedge B \wedge y = T\}$

```
p := p + x;
y := y - 1
```

$\{ y < T \}$

para lo cual procedemos aplicando la regla de la composición secuencial, obteniendo la pmd de la segunda asignación inducida por la primera

$$\begin{aligned} & \text{def}(y-1) \wedge y<T[y/y-1] \\ \text{sustitución} \quad & \Leftrightarrow \text{cierto} \wedge y - 1 < T \\ \text{Boole} \quad & \Leftrightarrow y - 1 < T \end{aligned}$$

que llamaremos R_3. Ahora tendremos que obtener la pmd de la primera asignación inducida por R_3, que será $\text{def}(p+x) \wedge R_3[p/p+x]$, pero como p+x está definida y en R_3 no aparece p como variable libre, nos queda el mismo aserto R_3. Para probar la corrección de la composición nos resta demostrar

$$I \wedge B \wedge y = T \Rightarrow R_3$$

lo que se obtiene

$$\begin{aligned} y = T \quad & \Rightarrow \quad y - 1 < T \\ & \Leftrightarrow \quad R_3 \end{aligned}$$

Con esto queda demostrada la corrección del bucle para la precondición R, ahora nos queda probar

$\{ P \}$

```
p := 0
```

$\{ R \}$

aplicando la regla de la asignación

$$\begin{aligned} & \text{def}(0) \wedge R[p/0] \\ \text{sustitución} \quad & \Leftrightarrow \text{cierto} \wedge x = X \wedge y = Y \wedge y \geq 0 \wedge 0 = 0 \\ \text{Boole y álgebra} \quad & \Leftrightarrow x = X \wedge y = Y \wedge y \geq 0 \\ & \Leftrightarrow P \end{aligned}$$

---

con lo que es correcta esta asignación y, por la regla de la composición secuencial, es correcto también todo el algoritmo.

Si en lugar de postular el aserto intermedio R hubiésemos utilizado el invariante como aserto intermedio, la verificación hubiese sido similar. La condición i.1 se habría simplificado al reducirse a $I \Rightarrow I$, mientras que se habría complicado la demostración de la corrección de la asignación p := 0, al tener que demostrar $P \Rightarrow I [p/0]$; para lo cual hay que utilizar un razonamiento similar al utilizado en el ejemplo para el requisito i.1

Al igual que ocurre con las demás instrucciones compuestas, en la composición iterativa también se define un formato de programa anotado

---

{ P }
{ I ; C }
  it B →
   { I ∧ B }
      A
   { I }
  fit
{ I ∧ ¬B }
{ Q }

Para la verificación de los bucles es imprescindible encontrar un invariante y una expresión de acotación, lo cual en muchos casos no resulta trivial. Es una destreza que se desarrolla con la práctica.

Nótese que pueden existir muchos asertos invariantes de un bucle. El que se necesita ha de ser suficientemente fuerte para que, junto con ¬B, conduzca a la postcondición, y lo bastante débil para que se cumpla antes de la primera iteración (lo implique la precondición) y el cuerpo del bucle lo mantenga invariante.

Obtención de invariantes

Podemos plantear dos estrategias particulares y ciertas recomendaciones generales:

---

1. *Relajación de una conjunción en la postcondición*. Sabemos que el invariante de un bucle ha de cumplir el requisito (i.3) que establece

   $$I \wedge \neg B \Rightarrow Q$$

   por lo que podemos plantearnos que quizás sea

   $$I \wedge B \Leftrightarrow Q$$

Si la postcondición es una conjunción de asertos, quizás la condición de terminación (¬B) aparezca expresamente, y que el resto de la postcondición represente el invariante.

A veces la postcondición no refleja expresamente como una conjunción todas las condiciones que expresa, y puede que sea preciso elaborarla de alguna manera para obtener una formulación equivalente y más explícita.

En el ejemplo anterior la postcondición p = X*Y puede suponerse fortalecida por la condición de terminación quedando

---

$$p = X*Y \wedge y = 0$$

equivalente a

$$p + y = X*Y \wedge y = 0$$

y también a

$$p + y*x = X*Y \wedge y = 0$$

---

de donde se puede obtener el invariante $p + y*x = X*Y$. La condición que se refiere a la expresión de acotación se podría añadir al intentar demostrar (c.1) y no obtener que la expresión de acotación es decrementable, por lo que se haría necesario añadir $y \geq 0$. A veces se puede fortalecer el invariante según avanza la verificación, y aún así no ser necesario modificar la demostración ya realizada, pues si una versión debilitada del invariante es suficiente para demostrar un cierto requisito también lo será el invariante completo.

---

2. *Reemplazar constantes por variables.*

Partimos de la postcondición y analizamos si los asertos que incluye dependen de constantes o de variables que no son modificadas en el cuerpo del bucle. Podemos estudiar si la sustitución de esas "constantes" por variables que se modifican –en cada pasada– en el bucle nos permite obtener un invariante. Para identificar qué variables sustituir, es preciso analizar el cuerpo del bucle y ver si las acciones de éste están "construyendo" la solución que expresa la postcondición, y si hay alguna variable o variables que representan el avance de la "construcción".

Esta situación es bastante habitual cuando tenemos algoritmos que tratan vectores. Las postcondiciones incluyen asertos que se refieren a cuantificadores sobre el rango del vector, por ejemplo:

$$\forall i : 1 \leq i \leq N : v(i) = 0$$

---

aquí lo que se estaría construyendo sería "un vector con todas sus componentes a 0", de forma que en un punto intermedio de la ejecución del bucle sólo habría una parte del vector a 0. Dependiendo de si la iteración asciende o desciende por el vector, la constante candidata a ser sustituida por una variable será N o 1.

En general podemos ofrecer las siguientes

Recomendaciones generales

— Analizar la postcondición

— Estudiar si el bucle "construye" algo

— Analizar las variables que se modifican y las que no

Obtención de expresiones de acotación

Para la obtención de expresiones de acotación la estrategia es única

---

Observar la condición de terminación (¬B)

Esta condición ha de cumplirse al terminar el bucle, por lo que en ese momento la expresión de acotación que estamos buscando puede tomar un valor minimal en un cierto dominio bien fundamentado. Incluso, puede ocurrir que la propia condición de terminación indique que la expresión de acotación toma un valor minimal, como ocurría en el ejemplo anterior, donde

---

$\neg B : y = 0$

---

nos indica que *y* puede ser la expresión de acotación.

En otras ocasiones puede ser necesario algo más de elaboración. Supongamos un bucle que recorre un vector en sentido ascendente, con una condición de repetición

---

$B : i \neq N+1$

y condición de terminación

$\neg B : i = N+1$

---

que es el mayor de los valores que toma *i*; para que la expresión de acotación vaya disminuyendo hasta llegar al valor minimal podemos tomar

---

$C : N + 1 - i$

de forma que

$i = N + 1 \Leftrightarrow N + 1 - i = 0$

Resulta más complicado cuando la condición de terminación no involucra expresiones que tomen valores en dominios bien fundamentados, en cuyo caso puede ser necesario aplicarles una función de transformación para conseguirlo.

En general podemos ofrecer las siguientes

Recomendaciones generales

— Observar la condición de terminación.

— Estudiar las expresiones que involucra y las variables que contiene y que se ven modificadas en el cuerpo del bucle.

— Siempre hay que pensar que quizás es posible demostrar que el bucle no termina (probándolo con un contraejemplo).

Otras construcciones iterativas

Los lenguajes de programación imperativa suelen incluir varias instrucciones iterativas. Nosotros sólo hemos incluido una en nuestro lenguaje algorítmico, para reducir así el número de reglas de verificación. Pero esto sólo es razonable si las otras construcciones iterativas son traducibles a nuestra única instrucción, como de hecho ocurre.

A continuación, presentamos la sintaxis de otras construcciones iterativas con su correspondiente traducción a la instrucción **it**. La utilización de estas instrucciones en los algoritmos supone que a la hora de verificar la corrección es necesario acudir a su traducción a la composición iterativa simple, y verificar sobre ella.

---

1. La iteración *repetir*
   — Sintaxis

   ```
   repetir A hasta B frepetir
   ```

   siendo B una expresión de tipo Bool, y A una acción
   — Semántica

   ```
   A;
   it NOT B → A fit
   ```

2. La iteración *para* ascendente con paso 1.
3.
   — Sintaxis

   ```
   para i desde M hasta N hacer A fpara
   ```

   siendo M y N expresiones de tipo Ent, y A una acción que no afecta a la variable *i*, que es de tipo Ent.
   — Semántica

   ```
   var final : Ent:
   inicio
     i := M;
     final := N + 1;
     it i < final →
       A;
       i := i + 1
     fit
   fvar
   ```

   Obsérvese que la iteración es inexistente si el valor inicial de M es mayor que el de N.

4. La iteración *para* descendente con paso 1.
   — Sintaxis

   ```
   para i bajando desde M hasta N hacer A fpara
   ```

   siendo M y N expresiones de tipo Ent, y A una acción que no afecta a la variable *i*, que es de tipo Ent.
   — Semántica

   ```
   var final : Ent:
   inicio
     i := M;
     final := N - 1;
     it i > final →
       A;
       i := i - 1
     fit
   fvar
   ```

   Obsérvese que la iteración es inexistente si el valor inicial de M es menor que el de N.

5. La iteración *para* general.
   — Sintaxis

   ```
   para i desde M hasta N paso P hacer A fpara
   ```

   siendo M, N y P expresiones de tipo Ent, y A una acción que no afecta a la variable *i*, que es de tipo Ent.
   — Semántica

   ```
   var paso, final : Ent:
   inicio
     paso := P;
     i := M;
     final := N + paso;
     si paso > 0 →
         it i < final →
           A;
           i := i + paso
         fit
      □ paso < 0 →
         it i > final →
           A;
           i := i + paso
         fit
   fvar
   ```

1.3 Funciones y procedimientos

Las acciones parametrizadas son un mecanismo que nos permite reutilizar el mismo algoritmo en distintos puntos de un programa, aplicado sobre distintos datos.

Las acciones parametrizadas tienen un identificador asociado, con el que podemos invocarlas, e identifican algunas de sus variables como *parámetros* a través de los cuales es posible transmitir los datos y recibir los resultados.

Distinguimos entre:

- Parámetros formales: aquellas variables que se designan como parámetros al definir la acción.
- Parámetros reales: las expresiones que se utilizan en el lugar de los parámetros formales en cada llamada concreta.

El reemplazamiento de los parámetros formales por los parámetros reales se llama *paso de parámetros*. Este proceso implica un flujo de información cuyo sentido viene dado por el *modo de uso* de cada parámetro

---

Definimos el modo de uso de los parámetros, que se ha de especificar en la definición de la acción parametrizada, como una de las siguientes posibilidades

- Entrada, que notaremos como **e.**
- Salida, que notaremos como **s.**
- Entrada/Salida, que notaremos como **es**.

Cada uno de los modos de uso tiene unas restricciones con respecto a las expresiones que pueden actuales como parámetros reales y formales

---

- Parámetros de entrada
  - Real. Puede ser cualquier expresión (el paso de parámetros la evalúa y le asigna el valor al correspondiente parámetro formal)
  - Formal. Se trata como una constante local que se inicializa con el valor del parámetro real
- Parámetros de salida
  - Real. Sólo puede ser una variable, que al finalizar puede haber sido modificada. El paso de parámetros la hace corresponder con el correspondiente parámetro formal
  - Formal. Se trata de una variable local no inicializada (no se puede suponer nada sobre su valor)
- Parámetros de entrada/salida

— Real. Sólo puede ser una variable, que al finalizar puede haber sido modificada. El paso de parámetros la hace corresponder con el correspondiente parámetro formal

— Formal. Se trata de una variable local que se supone inicializada con el valor de la variable del parámetro real.

Podemos resumir en una tabla las características de los parámetros reales

Modo de uso	debe ser variable	Parámetro real es consultado	puede modificarse
e	NO	SI	NO
s	SI	NO	SI
es	SI	SI	SI

Nótese que no permitimos realizar asignaciones a los parámetros de entrada: esto puede significar la definición de variables auxiliares adicionales.

En el lenguaje algorítmico tenemos dos tipos de acciones parametrizadas: procedimientos y funciones.

1.3.1 Procedimientos

Representan a la acción parametrizada más general, donde se permiten parámetros de entrada, salida y entrada/salida.

Especificación de procedimientos

En la especificación de un procedimiento, además de la pre y la postcondición, debemos indicar su nombre y sus parámetros formales: la cabecera del procedimiento

---

Definimos una especificación de un procedimiento como un esquema EP:

proc *nombreProc* (**e** $u_1{:}\tau_1$; … ; $u_n{:}\tau_n$; **s** $v_1{:}\delta_1$; … ; $v_m{:}\delta_m$; **es** $w_1{:}\rho_1$; … ; $w_p{:}\rho_p$);

{ P_0: $P' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n \wedge w_1 = W_1 \wedge \ldots \wedge w_p = W_p$ }

{ Q_0: $Q' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n$ }

fproc

donde

— los parámetros u_i, v_j y w_k son identificadores distintos para i= 1, …, n, j = 1, …, m y k=1, …, p

— los τ_i, δ_j y ρ_k son tipos del lenguaje algorítmico y representan los tipos de los parámetros para i = 1, …, n, j=1,…,m y k=1,…,p

— P_0 no contiene expresiones con los parámetros v_j para j=1, …, m.

Básicamente lo que se pide en la especificación de los procedimientos es que se respete el comportamiento de los parámetros formales: los parámetros de entrada y entrada/salida tienen valor, se mantiene el valor de los parámetros de entrada, no hay parámetros repetidos, la precondición no hace referencia a los parámetros de salida.

Veamos un par de ejemplos:

Procedimiento que obtiene la división entera y el resto

```
proc divMod( e x, y: Nat; s c, r: Nat );
{ P₀: x = X ∧ y = Y ∧ Y > 0 }
{ Q₀: x= c*y+r ∧ r < y ∧ x = X ∧ y = Y }
```

Procedimiento que busca un valor en un vector

```
proc busca( e v : Vector[1..N] de Nat; e x : Nat; s pos : Nat );
{ P₀: v = V ∧ x = X }
{ Q₀: (1 ≤ pos ≤ N ∧ v(pos)=x) ↔ (∃i : 1 ≤ i ≤ N : v(i) = x) ∧ x = X ∧ v =
V }
```

la variable *pos* tendrá un valor fuera del rango si no existe ninguna componente igual a *x*.

Declaración de procedimientos

En la declaración se completa la especificación con las declaraciones locales y el cuerpo del procedimiento

---

Definimos una declaración de un procedimiento como un esquema DP:

proc *nombreProc* (**e** $u_1{:}\tau_1$; … ; $u_n{:}\tau_n$; **s** $v_1{:}\delta_1$; … ; $v_m{:}\delta_m$; **es** $w_1{:}\rho_1$; … ; $w_p{:}\rho_p$);
{ P_0 }
cte … ;
var … ;
inicio
    A
{ Q_0 }
fproc

donde

— la cabecera de la declaración, la precondición P_0 y la postcondición Q_0 coinciden con la especificación del procedimiento

— Las declaraciones locales pueden ser vacías, en cuyo caso no aparecerán las palabras **cte** ni **var**

— La acción *A* es una acción del lenguaje algorítmico que sólo emplea los parámetros que aparecen en la cabecera, las variables y constantes de las declaraciones locales y que respeta el modo de uso de los parámetros.

El último requisito garantiza que no existen *efectos colaterales*, es decir no hay cambios en el estado que no recojan las especificaciones. En la práctica los lenguajes permiten que no se respete el modo de uso de los parámetros, por ejemplo, se puede modificar el valor de un parámetro de entrada sin que ello afecte al parámetro real.

Por ejemplo, para el algoritmo de división entera tenemos la siguiente declaración:

```
proc divMod( e x, y: Nat; s c, r: Nat );
{ P₀: x = X ∧ y = Y ∧ Y > 0 }
inicio
  <c,r> := <0,x>;
{ I: x = c*y+r ∧ y > 0 ∧ x = X ∧ y = Y;
  C: r                                    }
  it r ≥ y →
    c := c + 1;
    r := r - y
  fit
{ Q₀: x= c*y+r ∧ r < y ∧ x = X ∧ y = Y }
```

Llamadas a procedimientos

El último mecanismo que necesitamos introducir es el de las llamadas a los procedimientos.

Definimos una llamada a un procedimiento con una especificación EP como la instrucción LP

nombreProc(E_1, …, E_n, y_1, …, y_m, z_1, …, z_p)

donde

— Las E_i son expresiones de los tipos τ_i, para i=1,…, n

— Las y_j y z_k son variables distintas que admiten los tipos δ_j y ρ_k, respectivamente, para j=1,…,m y k=1,…,p

— Las variables y_j y z_k no aparecen en las expresiones E_i, para i=1,…,n, j=1,…,m y k=1, …,p

---

La última condición es necesaria para que las reglas de verificación funcionen correctamente, como veremos después. Si no se cumpliese este requisito tendríamos que introducir variables auxiliares que nos permitiesen realizar la verificación.

Los tipos de los parámetros reales no tienen que coincidir exactamente con los de los parámetros formales cuando existen relaciones de *compatibilidad* entre tipos. Así, para un parámetro for-

mal de entrada de tipo Real podemos utilizar un parámetro real de cualquier tipo compatible con Real: Nat, Ent, Rac, Real. De igual forma, para un parámetro formal de salida de tipo Ent podemos usar un parámetro real de cualquier tipo con el que Ent sea compatible: Ent, Rac, Real. En los parámetros de entrada/salida se combinan las dos restricciones de forma que el tipo debe ser exactamente el mismo. La idea es que en el flujo de información que implica el paso de parámetros, se puede asignar a una variable de un cierto tipo un valor de un tipo "con menos información".

Verificación de procedimientos

Las llamadas a procedimientos se consideran como instrucciones del lenguaje algorítmico, y como tales, deben llevar asociada su regla de verificación.

$\{ P \}$
nombreProc(E_1, …, E_n, y_1, …, y_m, z_1, …, z_p)
$\{Q\}$

Podríamos sustituir la llamada al procedimiento por su cuerpo y verificar así, pero no queremos tener que verificar el cuerpo cada vez, nos queremos aprovechar de la abstracción funcional. De esta forma, lo que haremos será verificar por separado (y una sola vez) la declaración del procedimiento con respecto a su especificación, y la verificación de las llamadas se reducirá a verificar su corrección con respecto a la especificación del procedimiento.

Para demostrar que una declaración de procedimiento DP es correcta con respecto a su especificación EP, es necesario aplicar las reglas de verificación al cuerpo del procedimiento. Si el cuerpo del procedimiento contiene llamadas a otros procedimientos, habrá que aplicar a éstas las reglas de verificación de llamadas.

Con respecto a la corrección de las llamadas, la idea básica consiste en demostrar que la precondición de la llamada es más fuerte que la precondición del procedimiento y que la postcondición del procedimiento es más fuerte que la postcondición de la llamada. Pero cuidado: puede haber una parte del estado que no se vea afectada por la ejecución del procedimiento; las expresiones de la precondición de la llamada que no hagan referencia a variables de salida del procedimiento, deberán aparecer en la postcondición, pero no será posible obtenerlas a partir de la postcondición del procedimiento, pues éste no las afecta.

Por ejemplo

```
{ x = X ∧ y = Y ∧ y > 0 ∧ h = 2*x }
  divMod(x,y,c,r)
{ x = y*c+r ∧ r < y ∧ x = X ∧ y = Y ∧ h = 2*x }
```

---

la condición h= 2*x no se puede obtener de la postcondición del procedimiento, pues h no es una de sus variables de salida.

Con esta idea podemos expresar la pmd de una llamada a un procedimiento

---

Dada una especificación de procedimiento como en EP para la que tenemos una declaración asociada DP que es correcta

pmd(*nombreProc*(E_1, …, E_n, y_1, …, y_m, z_1, …, z_p), Q) ⇔

$$F \wedge P_0[u_1/E_1, \ldots, u_n/E_n, w_1/z_1, \ldots, w_p/z_p]$$

siendo F el aserto más débil que cumple

$$F \wedge Q_0[u_1/E_1, \ldots, u_n/E_n, v_1/y_1, \ldots, v_m/y_m, w_1/z_1, \ldots, w_p/z_p] \Rightarrow Q$$

De aquí podemos obtener directamente el axioma de la llamada y la regla de verificación. Sin embargo, la regla de verificación así obtenida no es fácil de aplicar porque es necesario encontrar en cada caso el aserto F. Es por ello que, a partir de esta misma idea –lo que se ve afectado por el procedimiento y lo que no– definimos un método de verificación de llamadas en tres pasos.

Método de verificación de llamadas:

6. Comprobar si se cumple:

   $$P \Rightarrow P_0[u_1/E_1, \ldots, u_n/E_n, w_1/z_1, \ldots, w_p/z_p]$$

   (Nótese que para que se cumpla esto es necesario que se cumpla la parte de P_0 que hace referencia a que los parámetros de entrada y entrada/salida deben tener valor y además estos valores deben ser de los tipos adecuados. Esto ha de comprobarse de forma más cuidadosa cuando se trata de tipos sobre los que hay definida alguna relación de compatibilidad.)

7. Extraer de

   $$P \wedge P_0[u_1/E_1, \ldots, u_n/E_n, w_1/z_1, \ldots, w_p/z_p]$$

   todas las condiciones que dependen de variables que han de dar algún resultado de salida

   $$y_1, \ldots, y_m, z_1, \ldots, z_p$$

   y definir *R* como el aserto restante.

   (En general no basta con eliminar los asertos donde hay variables de entrada o entrada/salida, y puede ser necesario elaborar los asertos para no perder condiciones derivadas.)

8. Comprobar el requisito

   $$R \wedge Q_0[u_1/E_1, \ldots, u_n/E_n, v_1/y_1, \ldots, v_m/y_m, w_1/z_1, \ldots, w_p/z_p] \Rightarrow Q$$

Se puede demostrar que si el método de verificación termina, la llamada es correcta, partiendo de la pmd y razonando sobre la fuerza de los asertos, teniendo en cuenta que F es el aserto más débil que puede hacer el papel de R.

Veamos un ejemplo de verificación de una llamada:

---

dada la declaración

```
proc acumula( es s : Ent; e x : Ent ) ;
{ P₀: x = X ∧ s = S }
inicio
  s := s+x
{ Q₀: s = S+x ∧ x = X }
fproc
```

supongamos que queremos verificar la llamada

```
var
  suma, b : Ent;
  a : Nat;
{ P: suma > a*b }
  acumula(suma, 2*a*b)
{ Q: suma > 3*a*b }
```

9. $P \Rightarrow P_0[s/suma, x/2*a*b]$

$$
\begin{array}{ll}
 & P_0[s/suma, x/2*a*b] \\
\Leftrightarrow & 2*a*b = X \wedge suma = S \qquad (**) \\
\Leftrightarrow & cierto \\
\Leftarrow & P
\end{array}
$$

ya que se cumple suma:Ent y 2*a*b:Ent por coerción de *a* como Ent.

(**) Este aserto es cierto si *suma* y *2*a*b* tienen valor, lo cual no está incluido explícitamente en *P*, aunque se puede considerar implícito en *suma* > *a*b*, pues este aserto sólo puede suponerse cierto si tanto *suma* como *a*b* tienen valor.

10. Extraer de

$$
\begin{array}{ll}
 & P \wedge P_0[s/suma, x/2*a*b] \\
\Leftrightarrow & suma > a*b \wedge 2*a*b = X \wedge suma = S
\end{array}
$$

las condiciones que dependen de variables de salida, en este caso *suma*. Para ello es necesario extender primero el aserto

$$
\begin{array}{ll}
 & suma > a*b \wedge 2*a*b = X \wedge suma = S \\
\Leftrightarrow & suma > a*b \wedge 2*a*b = X \wedge suma = S \wedge S > a*b
\end{array}
$$

con lo que obtenemos

R: 2*a*b = X ∧ S > a*b

11. Comprobar

$$
\begin{array}{lll}
R \wedge Q_0[s/suma, x/2*a*b] & \Rightarrow & Q \\
 & & 2*a*b = X \wedge S > a*b \wedge suma = S + 2*a*b \wedge 2*a*b = X \\
 & \Rightarrow & suma > a*b + 2*a*b \\
 & \Leftrightarrow & suma > 3*a*b \\
 & \Leftrightarrow & Q
\end{array}
$$

Con lo que la llamada es correcta.

Hay que tener cuidado con las llamadas mal construidas. Veamos un ejemplo en que la regla de verificación permite verificar la corrección de un ejemplo erróneo, porque la llamada no cumple la restricción de que las variables reales de salida o entrada/salida no pueden aparecer en las expresiones correspondientes a los parámetros reales de entrada.

{ P : suma = 5 }
Acumula(suma, suma)
{ Q: suma = 3 }

Intuitivamente el resultado debería ser 10, pero como la llamada no cumple la restricción indicada veamos qué ocurre cuando intentamos verificarla:

12. $P \Rightarrow P_0[s/suma, x/suma]$

$$\begin{aligned} & P_0[s/suma, x/suma] \\ \Leftrightarrow\ & suma = X \wedge suma = S \\ \Leftrightarrow\ & cierto \\ \Leftarrow\ & P \end{aligned}$$

ya que se cumple suma:Ent por declaración de esa variable

13. Extraer de

$$\begin{aligned} & P \wedge P_0[s/suma, x/suma] \\ \Leftrightarrow\ & suma = 5 \wedge suma = X \wedge suma = S \end{aligned}$$

las condiciones que dependen de variables de salida, en este caso *suma*. Para ello es necesario extender primero el aserto

$$\begin{aligned} & suma = 5 \wedge suma = X \wedge suma = S \\ \Leftrightarrow\ & suma = 5 \wedge suma = X \wedge suma = S \wedge X = 5 \wedge S = 5 \end{aligned}$$

con lo que obtenemos

R: X = 5 ∧ S = 5

14. Comprobar

$$R \wedge Q_0[s/suma, x/suma] \Rightarrow Q$$

$$\begin{aligned} & R \wedge Q_0[s/suma, x/suma] \\ \Leftrightarrow\ & X = 5 \wedge S = 5 \wedge suma = S + suma \wedge suma = S \\ \Leftrightarrow\ & X = 5 \wedge S = 5 \wedge S = 0 \wedge suma = S \\ \Leftrightarrow\ & falso \\ \Rightarrow\ & Q \end{aligned}$$

Con lo que la llamada es correcta.

El problema es que no estamos distinguiendo las apariciones de *suma* que representan un dato de entrada de las que representan un dato de salida. La solución sería utilizar una variable auxiliar que, antes de la llamada, inicializamos con el valor de *suma*.

Uso de los procedimientos como asertos. Entendido como un aserto, la llamada a un procedimiento es, en esencia, equivalente a establecer que se cumple la postcondición aplicada sobre los argumentos de la llamada.

Es decir dada una especificación de procedimiento EP

---

proc *nombreProc* (**e** $u_1{:}\tau_1$; … ; $u_n{:}\tau_n$; **s** $v_1{:}\delta_1$; … ; $v_m{:}\delta_m$; **es** $w_1{:}\rho_1$; … ; $w_p{:}\rho_p$);
{ P_0: $P' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n \wedge w_1 = W_1 \wedge \ldots \wedge w_p = W_p$ }
{ Q_0: $Q' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n$ }
fproc

entendemos que el aserto

nombreProc(E_1, …, E_n, y_1, …, y_m, z_1, …, z_p)

es equivalente a

$Q'[u_1/E_1, \ldots, u_n/E_n, v_1/y_1, \ldots, v_m/y_m, w_1/z_1, \ldots, w_p/z_p]$

"teniendo cuidado con los parámetros de entrada/salida, para los cuáles el comportamiento expresado por Q' tiene dos direcciones".

Nótese que de la postcondición hemos quitado los asertos que se refieren a la conservación de valores pues no tienen sentido cuando consideramos la llamada como un aserto.

Por ejemplo:

---

divMod(3, 4, cociente, resto) es equivalente a

$3 = \text{cociente}*4 + \text{resto} \wedge 0 \leq \text{resto} < 4$

1.3.2 Funciones

Una función es un caso particular de procedimiento que:

— no tiene parámetros de entrada/salida

— y tiene uno o más parámetros de salida

Definimos una sintaxis especial para este tipo de procedimientos.

Especificación de funciones

Definimos una especificación de una función como un esquema EF:

func *nombreFunc* (u_1:τ_1; … ; u_n:τ_n) **dev** v_1:δ_1; … ; v_m:δ_m;
{ P_0: $P' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n$ }
{ Q_0: $Q' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n$ }
ffunc

donde

— los parámetros u_i y v_j son identificadores distintos para i= 1, …, n, y j = 1, …, m

— los τ_i y δ_j son tipos del lenguaje algorítmico y representan los tipos de los parámetros para i = 1, …, n, y j=1,…,m

— P_0 no contiene expresiones con los parámetros v_j para j=1, …, m.

Vemos que no es necesario indicar explícitamente el modo de uso de los parámetros, al estar separados. Naturalmente, exigimos que los identificadores de los parámetros formales sean distintos entre sí, pues representan valores distintos. La precondición no puede tener condiciones que se refieran a los parámetros de salida y, finalmente, la postcondición ha de exigir que los parámetros de entrada no sean modificados. En las funciones nos referimos a los parámetros de entrada como argumentos y a los de salida como resultados.

Por ejemplo, la división y el módulo se puede especificar como una función

func divMod(x, y : Nat) **dev** c, r : Nat;
{ P_0: x = X ∧ y = Y ∧ y > 0 }
{Q_0: x = c*y+r ∧ r < y ∧ x = X ∧ y = Y }

Declaración de funciones

Es similar a la declaración de procedimientos, el cuerpo se compone de las declaraciones, una acción algorítmica y una **acción de devolución**, al final, que establece los valores de los parámetros de salida.

Definimos una declaración de función como un esquema DF:

func *nombreFunc* (u_1:τ_1; … ; u_n:τ_n) **dev** v_1:δ_1; … ; v_m:δ_m;
{ P_0 }
cte …;
var …;

inicio
    A;
$\{ Q_0 \}$
    dev $<v_1, \ldots, v_m>$
ffunc

donde

— la cabecera de la función, la precondición P_0 y la postcondición Q_0 coinciden con la especificación de la función EF

— las declaraciones locales pueden ser vacías en cuyo caso no aparecerán las palabras **cte** ni **var**.

— la acción *A* es una acción del lenguaje algorítmico que sólo emplea los parámetros que aparecen en la cabecera y las variables y constantes de las declaraciones locales, y que respeta el modo de uso de los parámetros (no modifica el valor de los parámetros de entrada).

El último requisito garantiza la ausencia de efectos colaterales. Nótese que prohibimos realizar asignaciones a las variables de entrada; los lenguajes de programación realmente no imponen esta restricción pues las variables de entrada almacenan copias de los parámetros reales.

Los valores que se devuelven deben verificar la postcondición, y es por eso que la acción de devolución aparece detrás de aquella.

Veamos algunos ejemplos de declaración de funciones:

```
func divMod( x, y : Nat ) dev c, r : Nat;
{ P₀: x = X ∧ y = Y ∧ y > 0 }
inicio
  <c,r> := <0,x>
{ I: x = c*y+r ∧ y > 0 ∧ x = X ∧ y = Y ;
  C: r                                    }
  it r ≥ y →
    c := c + 1;
    r := r - y
  fit;
{ Q₀: x = c*y+r ∧ r < y ∧ x = X ∧ y = Y }
  dev <c,r>
ffunc
```

Una cosa curiosa de este ejemplo es que vemos cómo el invariante de un bucle depende de la postcondición (el invariante debe cumplir que $I \wedge \neg B \Rightarrow Q$); pues de no ser porque tenemos que obtenerlo en la postcondición, no se nos ocurriría incluir las condiciones $x=X \wedge y=Y$ en el invariante.

Otro ejemplo:

```
func mcd ( x, y : Nat ) dev m : Nat;
{ P0: x > 0 ∧ y > 0 ∧ x = X ∧ y = Y }
var
  a, b : Nat;
inicio
  <a,b> := <x,y>;
{ I: mcd(a,b) = mcd(x,y) ∧ x = X ∧ y = Y
  C: a+b                                  }
  it a ≠ b →
    si a > b → a := a - b
    □  b > a → b := b - a
    fsi
  fit;
  m := a;
{ Q0: m = mcd(x,y) ∧ x = X ∧ y = Y }
  dev m
ffunc
```

donde el aserto mcd(x,y) es equivalente a

$\max i : i \leq x \wedge i \leq y \wedge (x \bmod i = 0) \wedge (y \bmod i = 0) : i$

Llamadas a funciones

El resultado de una llamada a una función se recoge como una asignación múltiple.

Definimos una llamada a una función con un especificación EF y una declaración DF como la instrucción

$<y_1, \ldots, y_m> := \mathit{nombreFunc}(E_1, \ldots, E_n)$

donde

- las E_i son expresiones de los tipos τ_i para $i = 1, \ldots, n$
- las y_j son variables distintas que admiten los tipos δ_j, para $j=1, \ldots, m$. Estas variables actúan como los parámetros reales de salida.
- Las variables y_j no aparecen en las expresiones E_i para $i = 1, \ldots, n$ y $j = 1, \ldots, m$.

Los comentarios sobre la compatibilidad entre tipos que hicimos al describir las llamadas a procedimientos son también aplicables aquí, teniendo en cuenta que los parámetros son variables de entrada (el valor real puede ser de un tipo con "menos información") y los resultados son variables de salida (el parámetro real puede ser de un tipo con "más información").

Operacionalmente en la llamada a una función:

— Se evalúan las expresiones de los parámetros reales de entrada

— Se considera un estado auxiliar para la ejecución de la llamada, en el que sólo intervienen los parámetros formales de la función, y se les asignan los valores correspondientes a los de entrada.

— se ejecuta el cuerpo de la función en ese estado auxiliar (incluida la declaración de variables y constantes locales que puede ampliar el estado)

— al terminar la ejecución de la acción del cuerpo (si termina) se asignan a los parámetros reales de salida los valores correspondientes que tengan los parámetros formales en el estado auxiliar final

---

Verificación de las funciones

En cuanto a la verificación de las funciones, utilizamos los mismos resultados que obtuvimos para los procedimientos, considerando que no hay ninguna variable de entrada/salida.

Dada una especificación de función como en EF para la que tenemos una declaración asociada DF que es correcta

$pmd(<y_1, \ldots, y_m> := nombreFunc(E_1, \ldots, E_n), Q) \Leftrightarrow$

$$F \wedge P_0[u_1/E_1, \ldots, u_n/E_n]$$

siendo F el aserto más débil que cumple

$$F \wedge Q_0[u_1/E_1, \ldots, u_n/E_n, v_1/y_1, \ldots, v_m/y_m] \Rightarrow Q$$

Para realizar la verificación utilizaremos el mismo método de verificación que sugerimos para los procedimientos.

Cuando la función devuelve un único resultado podemos incluir una llamada a esa función dentro de otra expresión cualquiera, aunque en ese caso será necesario utilizar una **declaración local de variable**. Veamos un ejemplo de verificación de una llamada a función donde es necesario utilizar este mecanismo:

---

tenemos una función cuya especificación es

func fact(n : Nat) **dev** x : Nat;

$\{P_0: n > 0 \wedge n = N\}$

$\{Q_0: x = n! \wedge n = N\}$

(suponemos que ya hemos especificado el factorial) y supongamos que tenemos una declaración correcta con respecto a esa especificación. Queremos verificar la llamada

var num, suma : Nat;

{ P : num > 0 }

suma := 5 * fact(num)

{ Q : suma = 5 * num! }

la llamada es correcta porque la variable que recoge el resultado no aparece en la expresión que se usa como argumento, y además los tipos son correctos. Sin embargo, necesitamos introducir una variable auxiliar de la corrección *f*

{ P : num > 0 }
var f: Nat;
inicio
f := fact(num);
suma := 5 * f
fvar
{ Q : suma = 5 * num! }

Una declaración local es correcta si lo es su cuerpo.

El cuerpo lo verificamos como una composición secuencial. El aserto intermedio que usamos en la composición secuencial será la pmd de la asegunda asignación

R_1: 5*f = 5 * num!

con lo que pasamos a verificar

{ P : num > 0 }
f := fact(num);
{ R_1 : 5*f = 5 * num! }

usando el método de verificación de las llamadas a procedimientos:

15. Comprobamos

$$P \Rightarrow P_0[n/num]$$

$$\begin{aligned} & P_0[n/num] \\ \Leftrightarrow\ & num > 0 \wedge num = N \\ \Leftrightarrow\ & P \end{aligned}$$

ya *num:Nat* por la declaración.

16. Extraer de

$$\begin{aligned} & P \wedge P_0[n/num] \\ \Leftrightarrow\ & num > 0 \wedge num = N \end{aligned}$$

todas la condiciones que dependen de las variables de salida. Como ninguna condición se refiere a *f*, tomamos el aserto completo

R : num > 0 ∧ num = N

17. Comprobar el requisito

$$R \wedge Q_0[x/f, n/num] \Rightarrow R_1$$

$$R \wedge Q_0[x/f, n/num]$$

$$\Leftrightarrow \quad num > 0 \wedge num = N \wedge f=num! \wedge num = N$$

álgebra y fuerza $\Rightarrow \quad 5*f = 5 * num!$

$$\Leftrightarrow \quad R_1$$

con lo que la llamada es correcta.

Igual que ocurre con los procedimientos, se producen errores en la verificación si intentamos verificar funciones que no se ajusten al esquema LF (las variables de salida aparezcan en las expresiones de entrada).

Uso de las funciones como asertos. Entendido como un aserto, la llamada a una función es, en esencia, equivalente a establecer que se cumple la postcondición aplicada sobre los argumentos de la llamada.

Es decir dada una especificación de función EF

---

func *nombreFunc* (u_1:τ_1; … ; u_n:τ_n) **dev** v_1:δ_1; … ; v_m:δ_m;
{ P_0: $P' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n$ }
{ Q_0: $Q' \wedge u_1 = U_1 \wedge \ldots \wedge u_n = U_n$ }
ffunc

entendemos que el aserto

$<y_1, \ldots, y_m>:=$ *nombreFunc* ($E_1, \ldots, E_n$)

es equivalente a

$Q'[u_1/E_1, \ldots, u_n/E_n, v_1/y_1, \ldots, v_m/y_m]$

---

prescindimos de los asertos relativos a la conservación del valor de los parámetros de entrada.

1.4 Análisis de algoritmos iterativos

Hasta ahora nos hemos preocupado fundamentalmente por la corrección de los algoritmos; ahora vamos a ocuparnos de su eficiencia.

Un algoritmo será tanto más eficiente cuantos menos recursos consuma. Los recursos que consume un algoritmo son tiempo (de CPU) y espacio de memoria necesario para ejecutarlo.

Se ha llegado a afirmar que la eficiencia es irrelevante pues la capacidad de las computadoras aumenta año a año, y que son otros los criterios que se deben primar al desarrollar programas: claridad, legibilidad, reusabilidad. Sin embargo, lo que ocurre es que las aplicaciones informáticas cada vez son más exigentes: procesan volúmenes mayores de datos, aplicaciones en tiempo real, …

Para darnos cuenta del impacto del consumo de recursos veamos unas tablas ideales con distintos órdenes de consumo

Programa	Consumo	n=1.000	Bytes	Tiempo

A_1	$\log_2 n$	≈ 10	10	10 segs
A_2	n	1.000	$\approx$ 1 KB	16 mins
A_3	n^2	1.000.000	$\approx$ 1 Meg	11 días

Los programadores de los algoritmos A_2 y A_3 podrían esperar que las mejoras tecnológicas hiciesen útiles sus algoritmos. Sin embargo tomemos un volumen de datos de un millón (no descabellado: bases de datos de población, resolución de impresión, pixeles en pantalla, …)

Programa	**Consumo**	**n=1.000.000**	**Bytes**	**Tiempo**
A_1	$\log_2 n$	≈ 20	10	20 segs
A_2	n	1.000.000	$\approx$ 1 KB	11 días
A_3	n^2	10^{12}	$\approx$ 1.000 Gig	30.000 años

Vemos cómo el programa A_1 se ve muy poco afectado por el aumento en el volumen de datos; mientras que los programas A_1 y A_2 tienen aumentos prohibitivos, que los descalifican para la tarea.

El programa A_1 sí puede aprovechar las mejoras tecnológicas: con un procesador dos veces más rápido puede procesar 1000 veces más información; mientras que el programa A_3 con un procesador 1.000 veces más rápido "sólo" tardaría 30 años en realizar la tarea.

Un algoritmo eficiente es un bien mayor que cualquier mejora tecnológica; y éstas se deben entender, fundamentalmente, como un medio para procesar un mayor volumen de datos y no como una forma de remediar la pobre eficiencia de nuestros algoritmos. En resumen

---

— Un algoritmo eficiente permite explotar las posibilidades de una máquina rápida

— Una máquina rápida no es suficiente si no se dispone de un algoritmo eficiente

---

Sin embargo, también hay que tener en cuenta otra máxima:

La eficiencia suele contraponerse a la claridad y exige un desarrollo más costoso.

Por ello:

— En ocasiones puede interesar sacrificar la eficiencia. Por ejemplo, si el programa va a ejecutarse pocas veces o con datos de pequeño tamaño.

— Es conveniente, en todo caso, partir de un diseño claro, aunque sea ineficiente; y optimizar posteriormente.

1.4.1 Complejidad de algoritmos

La eficiencia de los algoritmos se cuantifica por medio de **medidas de complejidad**. Consideramos dos medidas de complejidad diferentes:

— Complejidad en tiempo: tiempo de cómputo de un programa

— Complejidad en espacio: memoria que utiliza un programa en su ejecución

---

Nosotros nos centraremos en la complejidad en tiempo que suele ser más crucial (la complejidad en espacio se suele resolver añadiendo más memoria).

El tiempo (y también el espacio) que tarda en ejecutarse un programa en una máquina concreta depende de:

— El tamaño de los datos de entrada

— El valor de los datos de entrada (la mayor o menor dificultad que entrañe el procesamiento de los datos)

— La máquina y el compilador.

En nuestro estudio vamos a ignorar este tercer factor pues afecta a todos los algoritmos por igual, como un factor constante (con una máquina el doble de rápida el algoritmo tardará la mitad); y, por lo tanto, no afecta a las comparaciones entre algoritmos.

El "tamaño de los datos" se pueden referir a distintas cosas según el tipo de datos y de procesamiento que se lleve a cabo: para un vector su longitud, para un número su valor o su número de dígitos, …

Con respecto al segundo factor, el valor de los datos, y una vez fijado el tamaño de los datos, podemos adoptar dos enfoques, estudiar el **peor caso** posible, estableciendo así una cota superior; o analizar el **caso promedio**, para lo cual debemos estudiar las posibles entradas y su probabilidad.

Definimos el coste temporal del algoritmo A con entrada $\vec{x}$, que notaremos

$t_A(\vec{x})$

como el tiempo consumido durante la ejecución del algoritmo A con entrada $\vec{x}$.

A partir del coste para unos ciertos datos podemos definir el coste en el caso peor y en el caso promedio

---

Definimos la complejidad de A en el caso peor, que notaremos

$T_A(n)$

como

$T_A(n) =_{def} \max\{ t_A(\vec{x}) \mid \vec{x} \text{ de tamaño n} \}$

Definimos la complejidad de A en el caso promedio, que notaremos

$TM_A(n)$

como

$$TM_A(n) =_{def} \sum_{\vec{x} \text{ de tamaño n}} p(\vec{x}) \cdot t_A(\vec{x})$$

siendo $p(\vec{x})$ la función de probabilidad ($p(\vec{x}) \in [0,1]$) de que la entrada sea el dato $\vec{x}$.

Aunque la **complejidad en promedio parece más realista**, su estimación exige determinar la distribución de probabilidades de los datos iniciales y hacer cálculos matemáticos complejos. Lo **más habitual es trabajar con la complejidad en el caso peor**, que da una cota superior del tiempo consumido por cualquier cómputo particular con datos de tamaño n.

Vamos a realizar los cálculos de la complejidad de manera teórica a partir de las instrucciones que aparecen en los algoritmos, viendo cómo afecta cada instrucción al coste. En cierto modo, "contaremos cuántas instrucciones se ejecutan". No nos planteamos la medida empírica del tiempo de ejecución porque queremos conocer los resultados a priori.

Debido a que las diferencias en el coste se hacen significativas cuando el tamaño de los datos aumenta, no nos va a interesar encontrar la función $T_A(n)$ exacta, sino que nos bastará con encontrar otra función f(n) que sea del mismo **orden de magnitud**; para que así podamos decir que para n grande $T_A(n)$ se comporta como f(n): **medidas asintóticas de la complejidad**.

Veamos un primer ejemplo donde se calcula el orden de magnitud del coste temporal de un algoritmo que determina si una matriz cuadrada de enteros es simétrica:

```
var v : Vector [1..N,1..N] de Ent;
  b : BOOL;
{ Pre.: v = V ∧ N ≥ 1 }
var
  I, J: Ent;
inicio
  b = cierto;
  para I desde 1 hasta N-1 hacer
    para J desde I+1 hasta N hacer
      b := b AND (v(I,J) = v(J,I))
    fpara
  fpara
fvar
{ Post: v = V ∧ b ↔ ∀ i, j : 1 ≤ i < j ≤ N : v(i,j) = v(j,i) }
```

Si tomamos como instrucciones significativas las asignaciones y tomamos a N como tamaño de los datos iniciales tenemos:

$T_A(n) = 1 + \sum i : 1 \leq i \leq N\text{-}1 : N\text{-}i$

(donde N–i es el número de veces que se ejecuta el bucle interno por cada pasada por el bucle externo)

$= 1 + (N-1) + (N-2) + \ldots + 1$

$= 1 + \frac{1+(N-1)}{2} * (N–1)$

$= 1 + \frac{N*(N-1)}{2}$

$\leq N^2$ para todo $N \geq 1$

y podemos decir por tanto que el coste del algoritmo A es el del orden de magnitud de N^2.

1.4.2 Medidas asintóticas de la complejidad

Una medida asintótica es una agrupación de funciones que muestra un comportamiento similar cuando los argumentos toman valores muy grandes. Estos conjuntos de funciones se definen en términos de un función de referencia *f*. Así definimos tres medidas asintóticas según que la función de referencia sirva como cota superior, cota inferior o cota exacta de las funciones del conjunto.

En estas definiciones vamos a considerar funciones T y f con el perfil

---

$T, f: N \rightarrow R^+ \cup \{0\}$

donde N es el dominio del tamaño de los datos y R^+ el valor del coste del algoritmo (ya sea temporal o espacial, pues las definiciones que siguen son válidas para ambos casos, así como para el caso peor y el caso promedio).

Imaginamos que T representa la complejidad de un cierto algoritmo.

Definimos la medida de cota superior f, que notaremos

$O(f)$

(omicrón mayúscula, que abreviaremos por "O grande") como el conjunto

$\{ T \mid \text{existen } c \in R^+, n_0 \in N \text{ tales que, para todo } n \geq n_0, T(n) \leq c \cdot f(n) \}$

Si $T \in O(f)$ decimos que "T(n) es del orden de f(n)" y que "f es asintóticamente una cota superior del crecimiento de T" (las funciones de O(f) crecen como mucho a la misma velocidad que f).

Definimos la medida de cota inferior f, que notaremos

$\Omega(f)$

(omega mayúscula, que abreviaremos llamándola "omega grande") como el conjunto

{ T | existe $c \in R^+$, $n_0 \in N$ tales que, para todo $n \geq n_0$, $T(n) \geq c \cdot f(n)$ }

Si $T \in \Omega(f)$ podemos decir que "f es asintóticamente una cota inferior del crecimiento de T". Las funciones de $\Omega(f)$ crecen con igual o mayor rapidez que f.

(En los apuntes de Mario se establece una distinción entre $\Omega_\infty(f)$ y $\Omega(f)$.)

Definimos la medida exacta f, que notaremos

$\Theta(f)$

(theta mayúscula, que abreviaremos "Theta grande") como el conjunto

{ T | existen $c_1, c_2 \in R^+$, $n_0 \in N$, tales que, para todo $n \geq n_0$, $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$

Si $T \in \Theta(f)$ decimos que "T(n) es del orden exacto de f(n)".

Veamos algunos ejemplos:

- f es O(f)
- $(n+1)^2$ es $O(n^2)$ con $c=4$, $n_0=1$
- $3n^3+2n^2$ es $O(n^3)$ con $c=5$, $n_0=0$
- p(n) es $O(n^k)$ para todo polinomio P de grado k
- 3^n no es $O(2^n)$. Si lo fuese existiría $c \in R^+$, $n_0 \in N$ tales que

  $\forall\ n \geq n_0 : 3^n \leq c \cdot 2^n$

  $\Rightarrow\ \forall\ n \geq n_0 : (3/2)^n \leq c$

  $\Leftrightarrow$ falso

  pues $\lim_{n\to\infty}(3/2)^n = \infty$

- f es de $\Omega(f)$
- n es de $\Omega(2n)$ con $c = ½$, $n_0=0$
- $(n+1)^2$ es de $\Omega(n^2)$ con $c = ¼$ y $n_0=1$ (también valdría con c=1)
- $P(n) \in \Omega(n^k)$ para todo polinomio P de grado k
- f es $\Theta(f)$
- 2n es $\Theta(n)$ con $c_1=1$, $c_2=2$ y $n_0=0$
- $(n+1)^2$ es $\Theta(n^2)$ con $c_1=1$, $c_2 = 4$, $n_0=1$
- P(n) es $\Theta(n^k)$ para todo polinomio P de grado k

En estos ejemplos se pueden comprobar que

$\Theta(f) = O(f) \cap \Omega(f)$

o lo que es igual

$\Theta(f) \Leftrightarrow O(f) \wedge \Omega(f)$ (ejercicio 53.g)

1.4.3 Ordenes de complejidad

Cuando las medidas asintóticas se aplican sobre funciones de referencia concretas tenemos conjuntos concretos de funciones. Habitualmente nos referiremos a las medidas asintóticas aplicadas a funciones concretas como *órdenes*. Algunos órdenes tienen especial interés porque la función de referencia es simple y su comportamiento se puede estudiar fácilmente. Estos órdenes tienen nombres particulares:

— $O(1) \rightarrow$ orden constante

— $O(\log n) \rightarrow$ orden logarítmico

— $O(n) \rightarrow$ orden lineal

— $O(n \log n) \rightarrow$ orden cuasi-lineal

— $O(n^2) \rightarrow$ orden cuadrático

— $O(n^3) \rightarrow$ orden cúbico

— $O(n^k) \rightarrow$ orden polinómico

— $O(2^n) \rightarrow$ orden exponencial

donde usamos log n para referirnos a $\log_2 n$.

Reciben los mismos nombres las otras medidas asintóticas (Ω, Θ) aplicadas sobre las mismas funciones de referencia; aunque nosotros nos ocupamos fundamentalmente de la cota superior, debido a que nos fijaremos casi siempre en el caso peor, y, dentro de esos supuestos pesimistas, lo que nos interesa es la cota superior.

Se puede demostrar (en el libro de Joaquín y Mario se sugiere cómo) que se cumplen las siguientes relaciones de inclusión entre órdenes de complejidad (jerarquía de órdenes de complejidad):

$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset$

$O(n^2) \subset O(n^3) \subset \ldots \subset O(n^k) \subset \ldots$

$O(2^n) \subset O(n!)$

La jerarquía de los órdenes de complejidad nos sirve como marco de referencia para comparar la complejidad de los algoritmos. **Un algoritmo será tanto más eficiente cuanto menor sea su complejidad dentro de la jerarquía de órdenes de complejidad**.

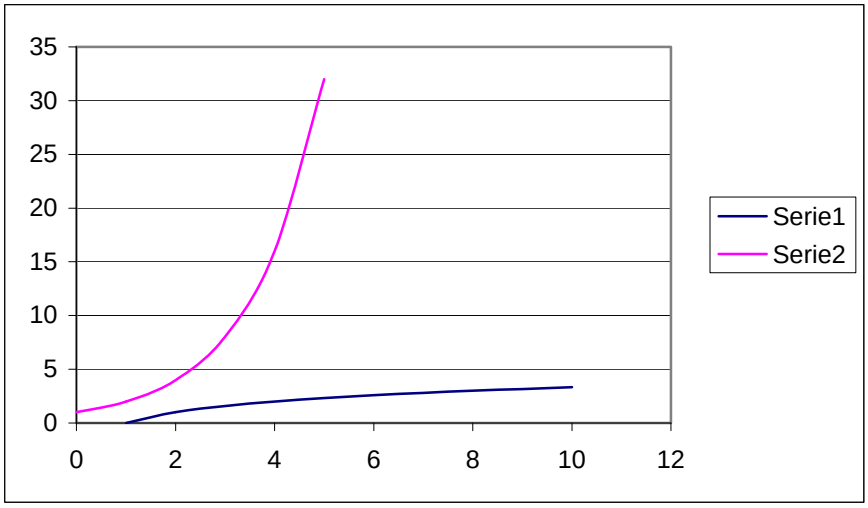
El estudio de los algoritmos para determinar el orden de magnitud de su complejidad se llama **análisis de algoritmos** (que da título a este apartado). El análisis de algoritmos sofisticados puede ser una tarea muy compleja.

Podemos establecer una comparación entre algoritmos con órdenes de complejidad diferentes, para ver cómo se ven afectados con el tamaño de los datos.

Supongamos dos algoritmos A y B que resuelven el mismo problema pero con complejidades diferentes:

$T_A \in O(\log n)$

$T_B \in O(2^n)$



tenemos por tanto que

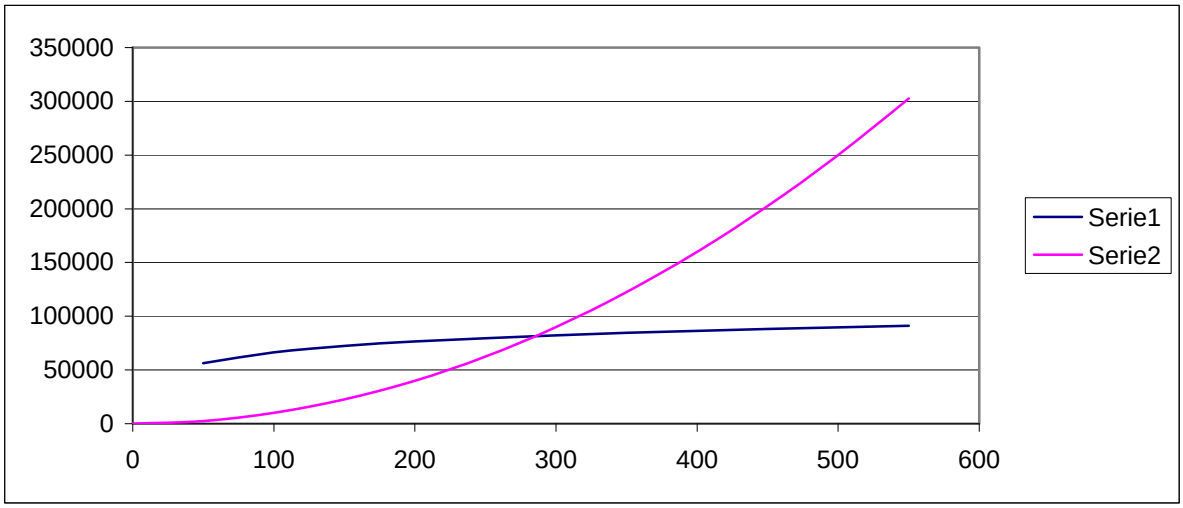
$T_A(n) \leq c \cdot \log n$ para $n \geq n_0$ $\qquad$ $T_B(n) \leq c' \cdot 2^n$ para $n \geq n_1$

---

podemos hacer algunas observaciones comparativas sobre estos dos algoritmos

---

— Si $c >> c'$ entonces B puede ser más rápido que A para datos pequeños



comparación entre n^2 y 10.000 log n

— A puede consumir más memoria que B

— Para A aumentar "mucho" el tamaño de los datos aumenta "poco" el tiempo necesario; y aumentar "poco" el tiempo disponible (la velocidad de la máquina) aumenta "mucho" el máximo tamaño tratable.

$\log(2n) = 1 + \log n$ al doblar el tamaño de los datos solo se suma 1 al tiempo

$2 \cdot \log n = \log n^2$ al doblar la velocidad los datos aumentan al cuadrado

— Para B aumentar "poco" el tamaño de los datos aumenta "mucho" el tiempo necesario; y aumentar "mucho" el tiempo disponible (la velocidad de la máquina) aumenta "poco" el máximo tamaño tratable.

$2^{2n} = (2^n)^2$ duplicar el tamaño de los datos eleva al cuadrado el tiempo necesario

$2 \cdot 2^n = 2^{n+1}$ duplicar el tiempo sólo aumenta en 1 el máximo tratable

---

Nos remitimos a la tabla que presentamos al principio del tema, donde el "consumo" se refiere en realidad a la complejidad.

Como conclusión llegamos a la idea que ya presentamos al principio del tema:

Sólo un algoritmo eficiente, con un orden de complejidad bajo puede tratar volúmenes de datos grande con un consumo de recursos razonable, y aprovechar las mejoras técnicas de forma que produzcan un incremento en el tamaño de los datos que puede tratar.

En este sentido, se suele considerar:

Un algoritmo A es

— muy eficiente si su complejidad es de orden log n

— eficiente si su complejidad es de orden n^k (polinómico)

— ineficiente si su complejidad es de orden 2^n

Y decimos que

Un algoritmo es tratable si tiene un algoritmo que lo resuelve con complejidad de orden menor que 2^n. Decimos que es intratable en caso contrario.

Para algunos problemas se ha demostrado que efectivamente no existe ningún algoritmo que los haga tratables. En cambio, para otros, simplemente no se ha encontrado ningún algoritmo, pero no se ha podido demostrar que no exista.

1.4.4 Métodos de análisis de algoritmos iterativos

Las definiciones teóricas que hemos visto hasta ahora son aplicables tanto a la complejidad temporal como a la espacial. A partir de aquí vamos a estudiar cómo se puede analizar en la práctica la complejidad temporal. En la segunda parte de la asignatura haremos algunas consideraciones sobre la complejidad espacial de las estructuras de datos utilizadas en la implementación de tipos abstractos de datos.

Vamos a ir analizando la complejidad para cada una de las estructuras de nuestro lenguaje algorítmico.

Antes, presentamos un par de reglas que sirven de utilidad para analizar los algoritmos.

Si $T_1(n) \in O(f_1(n))$ y $T_2(n) \in O(f_2(n))$, se cumplen las siguientes propiedades

— Regla de la suma

$$T_1(n) + T_2(n) \in O(\max(f_1(n), f_2(n)))$$

— Regla del producto

$$T_1(n) * T_2(n) \in O(f_1(n) \cdot f_2(n))$$

Demostración

Por la definición de O se tiene:

existen $c_1, c_2 \in R^+$; $n_1, n_2 \in N$ tales que

$\forall\ n \geq n_1\ T_1(n) \leq c_1 \cdot f_1(n)$ $\qquad$ $\forall\ n \geq n_2\ T_2(n) \leq c_2 \cdot f_2(n)$

tomando $n_0 = \max(n_1, n_2)$

— suma

$$T_1(n) + T_2(n) \leq c_1 \cdot f_1(n) + c_2\ f_2(n)$$

$$\leq (c_1+c_2) \cdot \max(f_1(n), f_2(n))$$

por lo que para $n \geq n_0$ se cumple la propiedad

— producto

$$T_1(n) \cdot T_2(n) \leq c_1 \cdot f_1(n) \cdot c_2 f_2(n)$$
$$\leq (c_1 \cdot c_2) \cdot (f_1(n) \cdot f_2(n))$$

por lo que para $n \geq n_0$ se cumple la propiedad

Vamos a ir viendo ahora cuál es la complejidad para cada una de las instrucciones. Esto es una simplificación que puede funcionar en muchos casos, pero no en todos.

Si A es

— seguir

$T_A(n) \in O(1)$

— $x := e$ ó $<x_1, \ldots, x_r> := <e_1, \ldots, e_r>$

$T_A(n) \in O(1)$ (excepto si la evaluación de e, e_i es compleja, en cuyo caso sería del orden del máximo de los órdenes de evaluar las expresiones)

— $A_1 ; A_2$ y $T_{Ai}(n) \in O(f_i(n))$

$T_A(n)= T_{A1}(n)+T_{A2}(n)$

y por la regla de la suma $T_A \in O(\max(f_1(n), f_2(n)))$

— **si** $B_1 \rightarrow A_1 \square \ldots \square B_r \rightarrow A_r$ **fsi**

$T_{Ai}(n) \in O(f_i(n))$ $(1 \leq i \leq n)$

entonces $T_A \in O(\max(f_1(n), \ldots, f_r(n)))$

siempre que podamos suponer que la evaluación de las barreras consume tiempo constante. Si esta suposición no es razonable y sabemos que

$T_{Bi}(n) \in O(g_i(n))$ $(1 \leq i \leq n)$

entonces $T_A \in O(\max(g_1(n), \ldots, g_r(n), f_1(n), \ldots, f_r(n)))$

— **it** $B \rightarrow A_1$ **fit**

$T_{A1}(n) \in O(f(n))$

$T_B(n) \in O(g(n))$

Sea $h(n) =_{def} \max(g(n), f(n))$

Entonces el tiempo de ejecución de una vuelta del bucle será $O(h(n))$

Si podemos estimar que el número de vueltas en el caso peor es O(t(n)), tendremos (usando la regla del producto):

$$T_A(n) \in O(h(n) \cdot t(n))$$

Generalmente es posible una estimación más fina de $T_A(n)$, estimando por separado el tiempo de cada vuelta y calculando un sumatorio.

Si el tiempo de ejecución de la vuelta número i es O(h(n,i)) entonces:

$$T_A(n) \in O(\sum i : 1 \leq i \leq t(n) : h(n,i))$$

Otras clases de bucles se analizan con ideas análogas.

Aparte de estas reglas generales, una idea útil en muchos casos prácticos es la de **acción característica**. Se trata de localizar en el texto del algoritmo A las acciones que se ejecutan con más frecuencia: las acciones características. El orden de magnitud de $T_A(n)$ se podrá estimar calculando el número de ejecuciones de acciones características (en realidad, se calcula una cota superior de ese número). La idea es que, puesto que en la composición secuencial se suman las complejidades, y la regla de la suma nos dice que la complejidad de una suma es igual al máximo de las complejidades de los sumandos, podemos hacer los cálculos quedándonos directamente con los máximos. Por ejemplo, en un programa que contenga un bucle, normalmente la acción característica será el cuerpo del bucle.

Veamos por último un par de ejemplo de cálculo de la complejidad.

El primero es una implementación más eficiente del algoritmo que comprueba si una matriz es simétrica

---

```
func esSim( a: Vector [1..N, 1..N] de Ent ) dev b: Bool;
{ Pre.: a = A }
var I, J: Ent;
inicio
  b := Cierto;
  I := 1;
  it I < N AND b
  →  J := I+1
     it J ≤ N AND b
     → b := b AND ( a(I,J) = a(J,I) )
       J := J +1
     fit;
     I := I+1
  fit
{ Post.: a = A ∧ ( b ↔ ∀i,j : 1 ≤ i < j ≤ N : a(i,j) = a(j,i) ) }
  dev b
ffunc
```

Nótese que este algoritmo parece mejor que el que presentamos al principio del tema porque se detiene en cuanto detecta que la matriz no es simétrica. Sin embargo, vamos a ver cómo, en el caso peor, la complejidad sigue siendo $O(N^2)$. Para obtener la complejidad en el caso promedio deberíamos tener una estimación de la probabilidad de los distintos valores de la matriz, y el cálculo sería muy complejo.

Como tamaño de los datos podemos tomar N o N^2; optamos por N porque eso nos facilitará el cálculo de la complejidad.

Suponemos que todas las asignaciones, evaluación de condiciones y acceso a vectores tienen complejidad constante O(1). Podemos obtener una expresión *fina*, contando el número de instrucciones:

---

cuerpo del bucle interno:

2 (podríamos considerar 3 si cada acceso al vector fuese 1)

coste del bucle interno

$((N-I) \cdot (2+1)) + 1$ donde 2 es el cuerpo y 1 la condición; y el otro 1 la condición de salida

$= 3\ (N-I) + 1$

coste del cuerpo del bucle interno

$1 + (3(N-I) + 1) + 1 = 3(N-I) + 3$

coste del bucle externo

$[\sum_{I=1}^{N-1} (3(N-I) + 3 + 1)] + 1$ donde el 1 interno es la condición; y el 1 externo la condición de salida

$= 3\sum_{I=1}^{N-1} (N-I) + 4(N-1) + 1 = 3 \cdot \frac{N \cdot (N-1)}{2} + 4(N-1) + 1$

$= 3/2\ N^2 - 3/2\ N + 4N - 4 + 1$

$= 3/2\ N^2 + 5/2\ N - 3$

y el coste del algoritmo, considerando las dos asignaciones iniciales

$T_A(N) = 3/2\ N^2 + 5/2\ N - 3 + 2 = 3/2\ N^2 + 5/2\ N - 1$

con lo que $T_A(N) \in O(N^2)$

Sin embargo, podemos llegar directamente al mismo resultado razonando sobre la instrucción característica:

Consideramos como acciones características las asignaciones que componen el cuerpo del bucle más interno. El bucle interno se ejecuta desde I+1 hasta N, es decir, se ejecuta N-I veces para cada valor de I, por lo tanto:

$$T(N) \in O(\sum_{I=1}^{N-1} (N-I)) = O(N^2)$$

Como segundo ejemplo consideramos el algoritmo de multiplicación eficiente del ejercicio 36(b)

```
var x, y, p : Ent;
{ Pre. P: x = X ∧ y = Y ∧ y ≥ 0 }
p := 0;
it y ≠ 0 →
  si par(y)
    entonces <x, y> := <x+x, y div 2>
    sino     <p, y> := <p+x, y-1>
  fsi
fit
{ Post. Q: p = X*Y }
```

Como tamaño de los datos tomamos Y = n

como complejidad del cuerpo del bucle tomamos la complejidad de la estructura condicional, que, al ser el máximo de la complejidad de las ramas y las barreras, y tener todas complejidad constante será O(1). Por tanto la complejidad del bucle será el número de vueltas

$T_A(n) \in O(t(n))$

La idea es que el y se divide por 2 en una o dos vueltas con lo que la complejidad es de orden O(log n). Esta idea se extrae de que si en cada pasada por el bucle se disminuye el tamaño de los datos a la mitad, hasta alcanzar el valor 1 entonces la complejidad es logarítmica:

si un algoritmo se ejecuta una vez con datos de tamaño n, la siguiente con n/2, …, hasta llegar a datos de tamaño 1

n, n/2, n/4, …, 1 esta serie es equivalente a

$n/2^0, n/2^1, n/2^2, \ldots, n/2^k$ siendo k el número de veces que se ha ejecutado el bucle

$1 = n/2^k$

$2^k = n$

$k = \log n$

por lo tanto

si $t(n) \in O(\log n)$ entonces $T_A(n) \in O(\log n)$

Sólo para este caso, vamos a ver con más detalle la demostración de que $t(n) \in O(\log n)$. Probando con distintos valores de *y* vemos que el comportamiento del algoritmo es el siguiente:

n = 0 : y = 0
n = 1 : y = 1 y = 0
n = 2 : y = 2 y = 1 y = 0
n = 3 : y = 3 y = 2 y = 1 y = 0
n = 4 : y = 4 y = 2 y = 1 y = 0
n = 5 : y = 5 y = 4 y = 2 y = 1 y = 0

formulamos la conjetura

$t(n) \leq 2 \log n$ para $n \geq 2$

esta conjetura se demuestra por inducción sobre n

base

n = 2 $t(n) = 2 = 2 \log(n)$ (se ejecuta dos veces, para y=2, y para y=1)
n = 3 $t(n) = 3 \leq 2 \log(3)$

$$3 \leq 2 \log(3) \Leftrightarrow 2^3 \leq 2^{2 \log(3)} \Leftrightarrow 2^3 \leq (2^{\log 3})^2 \Leftrightarrow 2^3 \leq 3^2 \Leftrightarrow 8 \leq 9 \Leftrightarrow \text{cierto}$$

paso inductivo

— n > 3, par n = 2n', n' ≥ 2

$$\begin{aligned} t(n) = 1 + t(n') &\leq_{HI} 1 + 2 \log(n') \\ &< 2 (1 + \log(n')) \\ &= 2 \cdot \log(2\, n') \\ &= 2 \log(n) \end{aligned}$$

se ejecuta 1 vez más las veces de n'

— n > 3, impar n = 2n'+1, n' ≥ 2

$$\begin{aligned} t(n) = 2 + t(n') &\leq_{HI} 2 + 2 \log(n') \\ &= 2 (1 + \log(n')) \\ &= 2 \cdot \log(2\, n') \\ &< 2 \log(2n' + 1) \\ &= 2 \log n \end{aligned}$$

se ejecuta 1+1 veces más las veces de n'

1.4.5 Expresiones matemáticas utilizadas

$$\sum_{i=min}^{max} i = \frac{min + max}{2} \cdot (max - min + 1)$$

donde (max–min+1) es el número de términos.

1.5 Derivación de algoritmos iterativos

Una vez que hemos estudiado las técnicas para verificar la corrección de programas ya escritos vamos a estudiar las técnicas que permiten diseñar programas a partir de la especificación.

Utilizamos las reglas de verificación para diseñar los programas, de forma que para los programas obtenidos ya esté demostrada su corrección.

Lo principal que aporta este método es que descompone la tarea de la programación en distintas subtareas, de forma que nos podemos concentrar en distintos aspectos del proceso de diseño del programa.

1.5.1 El método de derivación

Se trata de derivar un programa que cumpla la especificación

$\{ P \}\ A\ \{ Q \}$

---

hay que comenzar tomando una decisión acerca de la estructura de A al nivel más externo. Es decir, hay que decidir cuál de las estructuras disponibles en el lenguaje algorítmico es la que corresponde a la acción A: una acción simple, una composición secuencial, una distinción de casos, o una repetición. Vamos a ir considerando cada caso

---

— Asignación.

Usamos esta instrucción si podemos encontrar una asignación A tal que

$$P \Rightarrow pmd(A, Q)$$

El caso más habitual es cuando la única diferencia entre la postcondición y la precondición es una igualdad de la forma

$$x = E$$

donde E es una expresión que se puede obtener mediante operaciones predefinidas o especificadas.

— Composición secuencial

Usamos esta opción cuando decidimos que para obtener la postcondición Q es necesario obtener un aserto intermedio R, que nos acerca a Q. En ese caso, la derivación de A se transforma en la derivación de dos acciones A_1 y A_2 que cumplan

$$\{ P \}\ A_1\ \{ R \} \qquad \text{y} \qquad \{ R \}\ A_2\ \{ Q \}$$

— Composición alternativa

Usamos esta opción cuando decidimos que los datos de entrada **no admiten un tratamiento uniforme** para obtener la postcondición.

Deberemos encontrar condiciones B_1, …, B_n que **discriminen los casos homogéneos**, de tal forma que se cumpla

$$P \Rightarrow B_1 \vee \ldots \vee B_n$$

Y entonces, para cada condición B_i deberemos derivar las acciones A_i que verifiquen

$$\{ P \wedge B_i \}\ A_i\ \{Q\} \qquad \text{para } i= 1, \ldots, n$$

Y así, el algoritmo resultante será de la forma:

$$A \equiv \textbf{si}\ B_1 \rightarrow A_1 \square \ldots \square B_n \rightarrow A_n\ \textbf{fsi}$$

— Composición iterativa.

Usamos esta opción cuando decidimos que la postcondición debe obtenerse repitiendo una determinada acción.

Para obtener el algoritmo utilizamos el método de derivación de bucles

---

La opción más interesante, habitual y difícil es esta última, que tiene su propio método y que es de lo que nos ocupamos en el resto de este tema.

Derivación de bucles

La derivación de un bucle se hace a través de los siguientes pasos:

— **B.1**. Invariante

Obtenemos el invariante a partir de la postcondición.

— **B.2**. Condición de repetición

Con el invariante y la postcondición obtenemos las condición de terminación

$$I \wedge \neg B \Rightarrow Q$$

y de ahí, la condición de repetición. (Hemos de demostrar que efectivamente se cumple i.3)

En ocasiones los pasos B.1 y B.2 se realizan simultáneamente.

— **B.3**. Definición de la condición de repetición

Comprobamos si

$$I \Rightarrow \text{def}(B)$$

En ocasiones no será así porque en B (y en I) hayan aparecido variables nuevas (sustituir constantes por variables). En ese caso se deberán introducir con una declaración local.

— **B.4**. Expresión de acotación

Considerando I, B se construye una expresión de acotación C de modo que

$$I \wedge B \Rightarrow def(C) \wedge dec(C)$$

(Recuérdese que para obtener expresiones de acotación, la recomendación era observar la condición de terminación, así como las variables que se modificaban en el bucle).

— **B.5**. Acción de inicialización.

Si la precondición no garantiza que se cumple el invariante antes de entrar en el bucle, entonces es necesario derivar una acción de inicialización que establezca el invariante.

Esta acción se deriva a partir de la especificación

$$\{P\}\ A_0\ \{I\}$$

— **B.6**. Acción de avance.

Se deriva una acción A_2 que haga avanzar al bucle hacia su terminación. Se puede derivar a partir de la especificación

$$\{I \wedge B \wedge C = T\}\ A_2\ \{c \prec T\}$$

— **B.7**. ¿Es suficiente con la acción de avance?

Nos preguntamos si la acción de avance constituye el cuerpo del bucle. Para ello, debemos demostrar

$$\{B \wedge I\}\ A_2\ \{I\}$$

Si lo cumple, entonces hemos acabado de obtener un algoritmo correcto.

— **B.8**. Restablecimiento del invariante

Si la acción de avance no es suficiente (lo más habitual), entonces necesitamos derivar una nueva acción A_1 que se encargue de restablecer el invariante dentro del bucle, de forma que

$$\{I \wedge B\}\ A_1\ ;\ A_2\ \{I\}$$

la acción A_1 se puede derivar de la especificación

$$\{I \wedge B\}\ A_1\ \{R\}$$

donde $R \equiv pmd(A_2, I)$

— **B.9**. Terminación.

Se comprueba que después de introducir la acción A_1 se sigue cumpliendo

$$\{I \wedge B \wedge C = T\}\ \ A_1\ ;\ A_2\ \{c \prec T\}$$

Si se comprueba, entonces hemos acabado de obtener un algoritmo correcto

---

El esquema general resultante de la derivación de un bucle es de la forma:

```
{ P }
  var x1:τ1; …; xn:τn;
  inicio
  A0
{ I; C }
  it B →
    { I ∧ B }
      A1;
    { R }
      A2;
    { I }
  fit;
{ I ∧ ¬B }
  fvar
{ Q }
```

Insistir otra vez en que lo que nos aporta este método es que nos obliga a concentrarnos en las distintas partes del problema, comprobando paso a paso que "vamos bien"; y, si nos equivocamos, tenemos puntos bien definidos a los que volver.

1.5.2 Dos ejemplos de derivación

Estos dos ejemplos se diferencian por la forma en que se obtiene el invariante a partir de la postcondición. Las técnicas que usaremos son las mismas que comentamos al final del apartado de verificación de bucles: tomar una parte de la conjunción que forma la postcondición; y sustituir constantes por variables en las postcondición.

Hay otra recomendación general en cuanto a la construcción de invariantes a partir de una especificación:

Si la precondición y la postcondición del bucle contienen una igualdad x=X, el invariante también debe incluirla.

No parece razonable –aunque no sea imposible– que una variable empiece tomando valor, que se ve modificado por el bucle, para que al final vuelva a tomar el valor inicial.

Existe una diferencia esencial en la obtención del invariante para una verificación a posteriori y una derivación: en la derivación tenemos menos información, y por lo tanto menos restricciones, sobre el bucle, de forma que disponemos de más libertad a la hora de seleccionar el invariante.

Obtención del invariante por relajación de una conjunción en la postcondición

El razonamiento, como ya indicamos se basa en considerar la condición i.3 de la verificación de bucles

$$I \wedge \neg B \Rightarrow Q$$

y preguntarnos si no será

$$I \wedge \neg B \Leftrightarrow Q$$

pudiéndose así obtener el invariante y la condición de terminación directamente –o con alguna reformulación– de la postcondición.

El ejemplo que vamos a derivar es la división entera y el cociente de dos números enteros positivos.

---

var x, y, c, r : Ent;

{P: $x = X \wedge y = Y \wedge x \geq 0 \wedge y > 0$ }

divMod

{Q: $x = X \wedge y = Y \wedge x = c*y + r \wedge 0 \leq r < y$ }

Empezamos por determinar de manera informal "la idea del algoritmo". En este caso, nos proponemos obtener el cociente como el número de veces que hay que restarle *y* a *x* para obtener un valor menor que *y*; siendo ese valor el resto. Esta idea nos sugiere una solución iterativa, repetimos la resta de *y*.

$$\underbrace{x - y - y \ldots - y}_{c \text{ veces}} = r$$

Una vez determinado que nuestra algoritmo va a ser un bucle, aplicamos los pasos de la derivación de bucles.

— **B.1**. Invariante

las ecuaciones relativas a la conservación de valores pasan a formar parte del invariante

$$x = X \wedge y = Y$$

lo siguiente es darnos cuenta de que la parte de la postcondición relativa a la condición de terminación es $r < y$: restaremos hasta que el resto sea menor que *y*. Por lo tanto el invariante es el resto de la postcondición

$$I: x = X \wedge y = Y \wedge x = c*y + r \wedge r \geq 0 \wedge y > 0$$

---

donde hemos obtenido $r \geq 0 \wedge y > 0$, a partir de $0 \leq r < y$. ¿Podríamos obviar $y > 0$? En general, "mejor que sobre", siempre y cuando las condiciones que incluimos en el invariante sean

fáciles de conseguir a partir de la precondición; o como, en este caso, aparezcan explícitamente en la precondición. Pensemos, que el invariante ha de ser lo bastante débil para que se pueda obtener de la precondición, y lo bastante fuerte que, junto con la condición de terminación, implique a la postcondición. Por lo tanto, todo lo que sea reforzar el invariante con condiciones que aparecen en la precondición es, en principio, beneficioso. En este caso, la condición y>0 es necesaria para demostrar que la expresión de acotación (*r*) es decrementable; podría ser al intentar demostrar este punto cuando nos diésemos cuenta de la necesidad de incluirla. En cualquier momento podemos incluir reforzamientos del invariante, sin que ello afecte a una gran parte de la demostración; tan sólo afectará a la demostración de P $\Rightarrow$ I, aunque no sea éste el caso, pues estamos reforzando el invariante con una condición que aparece en la precondición.

Para reconocer que este es un invariante válido debemos ya tener en la cabeza la forma básica del bucle, y darnos cuenta de que vamos a ir aumentando *c*, 1 a 1 en cada pasada, y vamos a ir restándole *y* al valor de *r*; con lo que en todo momento se cumple la relación indicada: x = y*c+r.

— **B.2**. Condición de repetición

Con el razonamiento anterior la condición de terminación

$\neg B : r < y$

y probamos que efectivamente se cumple

$I \wedge \neg B \Rightarrow Q$

$$\begin{aligned} I \wedge \neg B &\Leftrightarrow x = X \wedge y = Y \wedge x = c*y+r \wedge r \geq 0 \wedge y > 0 \wedge r < y \\ &\Leftrightarrow x = X \wedge y = Y \wedge x = c*y+r \wedge 0 \leq r < y \\ &\Leftrightarrow Q \end{aligned}$$

y por tanto, la condición de repetición

$B : r \geq y$

— **B.3**. Definición de la condición de repetición

Comprobamos si

$I \Rightarrow def(B)$

$$\begin{aligned} & def(r \geq y) \\ \Leftrightarrow\ & cierto \\ \Leftrightarrow\ & I \end{aligned}$$

ya que las variables están declaradas

— **B.4**. Expresión de acotación

Fijándonos en B ($r \geq y$) y considerando que *r* va a ir decreciendo en cada pasada por el bucle, tomamos *r* como expresión de acotación

$C : r$

para la que debemos probar

$I \wedge B \Rightarrow def(C) \wedge dec(C)$

$$\begin{aligned} & def(r) \\ \Leftrightarrow\ & cierto \\ \Leftarrow\ & I \wedge B \end{aligned}$$

$$
\begin{aligned}
& \texttt{dec(C)} \\
\Leftrightarrow\ & \texttt{r > 0} \\
\Leftarrow\ & \texttt{r} \geq \texttt{y} \wedge \texttt{y > 0} \\
\Leftarrow\ & \texttt{I} \wedge \texttt{B}
\end{aligned}
$$

para esta última prueba nos hace falta $y > 0$, que está implícita en la postcondición y explícita en la postcondición; podría ser aquí donde nos diésemos cuenta de que debe formar parte del invariante.

— **B.5**. Acción de inicialización.

Observamos que no se puede deducir el invariante de la precondición, y que se hace necesario incluir una acción de inicialización. Lo más fácil es asignar valores a las variables de tal modo que se cumplan los asertos del invariante. La fórmula problemática es

$$x = y*c + r$$

tomamos como acción de inicialización

$$<c,r> := <0,x>$$

con lo que nos quedaría por demostrar

$$\{P\}\ <c,r> := <0,x>\ \{I\}$$

que es, efectivamente, correcto

— **B.6**. Acción de avance.

Tenemos que si *x* es menor que *y* entonces el bucle no se ejecuta ninguna vez ($r=x<y$). Si *x* es mayor que *y*, entonces entramos en el bucle, que debe decrementar el valor de *r*. Según nuestra idea inicial del bucle, le restamos a *r* el valor de *y*, haciendo que sea esa nuestra acción de avance

$$r := r - y$$

para la que debemos probar que se cumple

$$\{ I \wedge B \wedge r = T \}\ r := r - y\ \{ r < T \}$$

que es efectivamente correcto

— **B.7**. ¿Es suficiente con la acción de avance?

Nos preguntamos si la acción de avance constituye el cuerpo del bucle. Para ello, debemos demostrar

$$\{ B \wedge I \}\ r := r-y\ \{ I \}$$

$$
\begin{aligned}
& \texttt{def(r-y)} \wedge \texttt{I[r/r-y]} \\
\Leftrightarrow\ & \texttt{cierto} \wedge \texttt{x = X} \wedge \texttt{y = Y} \wedge \texttt{x = c*y + (r-y)} \wedge \texttt{y > 0} \wedge \texttt{r - y} \geq \texttt{0} \\
\Leftrightarrow\ & \texttt{x = X} \wedge \texttt{y = Y} \wedge \texttt{x = (c-1)*y + r} \wedge \texttt{y > 0} \wedge \texttt{r} \geq \texttt{y}
\end{aligned}
$$

tenemos entonces

$$I \Rightarrow x = X \wedge y = Y \wedge y > 0$$

$$B \Leftrightarrow r \geq y$$

pero no podemos demostrar

$$I \wedge B \Rightarrow x = (c-1)*y + r$$

por lo que debemos pasar a B.8 e intentar restablecer el invariante, tomando como aserto intermedio *R*, el que acabamos de obtener como precondición más débil de la acción de avance inducido por el invariante

— **B.8**. Restablecimiento del invariante

$$\{ I \wedge B : x = X \wedge y = Y \wedge x = c*y + r \wedge r \geq 0 \wedge y > 0 \wedge r \geq y \}$$
$$A_1$$
$$\{ R : x = X \wedge y = Y \wedge x = (c-1)*y + r \wedge y > 0 \wedge r \geq y \}$$

Vemos que la única diferencia se puede solventar con una asignación

$$c := c + 1$$

de forma que al sustituir c/c+1 en la postcondición, lleguemos a la precondición.

Efectivamente se puede demostrar

$$I \wedge B \Rightarrow R[c/c+1]$$

— **B.9**. Terminación.

Por último, nos queda por comprobar que esta nueva acción en el cuerpo del bucle no afecta a la terminación, y se verifica

$$\{ I \wedge B \wedge r = T \}$$
$$c := c + 1;$$
$$r := r - y;$$
$$\{ r < T \}$$

informalmente se puede afirmar que se sigue verificando pues la acción de restablecimiento del invariante no afecta a la expresión de acotación *r*. Pero, para completar el proceso, sería necesario demostrarlo, obteniendo la pmd y verificando que $I \wedge B \wedge r = T$ es más fuerte que ella.

Finalmente el diseño del algoritmo anotado queda:

```
var x, y, c, r : Ent;
{ P : x = X ∧ y = Y ∧ y > 0 ∧ x ≥ 0 }
 <c,r> := <0,x>;
{ I : x = X ∧ y = Y ∧ x = c*y + r ∧ r ≥ 0 ∧ y > 0
 C : r                                              }
 it r ≥ y →
   { I ∧ r ≥ y }
     c := c + 1;
   { x = X ∧ y = Y ∧ x = (c–1)*y + r ∧ r ≥ y ∧ y > 0 }
     r := r – y;
```

```
   { I }
 fit
{ I ∧ r < y }
{ Q : x = X ∧ y = Y ∧ x = y * c + r ∧ 0 ≤ r < y }
```

En cuanto a la complejidad del algoritmo; si tomamos como acción característica el cuerpo del bucle, que es O(1), tenemos que el bucle se ejecuta X div Y veces, con lo que su complejidad es O(X div Y).

Obtención del invariante reemplazando constantes por variables

Esta es la segunda directriz general acerca de la obtención de invariantes. Es típica en bucles que "construyen" algo, y en bucles sobre vectores; sustituimos una constante por una variable que va indicando el avance del bucle. Esta técnica es recomendable cuando existen asertos o expresiones extendidos (∀, ∃, ∑, Π, max, min, #) cuyos asertos de dominio incluyen constantes, o variables que se supone que no se modifican en el bucle.

Como consejo general, el invariante debe contener un aserto que indique el rango de las nuevas variables. Este aserto nos será útil para demostrar la terminación del bucle.

Al obtener un invariante por reemplazamiento de constantes, todas las variables nuevas deben tener un aserto que indique su rango de posibles valores.

Vamos a derivar un algoritmo para el cálculo del producto escalar de dos vectores.

```
cte N = …; % Nat
var u, v : Vector [1..N] de Real;
  r : Real;
{ P : v = V ∧ u = U }
 prodEscalar
{ Q : r = ∑ i : 1 ≤ i ≤ N : u(i)*v(i) ∧ u = U ∧ v = V }
```

La idea del bucle es un bucle, donde, en cada pasada, sumamos al resultado parcial el resultado de multiplicar la componente *j*.

— **B.1**. Invariante

Sustituimos la constante *N* de la postcondición por una nueva variable *j*. Además conservamos los asertos de conservación de valores que aparecen tanto en la precondición como en la postcondición, e introducimos un aserto sobre el rango de la nueva variable. Si no lo introdujéramos aquí, veríamos luego que nos hace falta para demostrar que la expresión de acotación puede decrecer.

I: $r = \sum i : 1 \leq i \leq j : u(i) * v(i) \wedge u = U \wedge v = V \wedge 1 \leq j \leq N$

aquí hemos introducido el rango de la nueva variable, suponiendo que toma valores en el rango de índices del vector; luego veremos que no es así y que debemos ampliar el rango con 0, para capturar así el primer estado antes de ejecutar el bucle por primera vez. "En la vida real" podríamos darnos cuenta aquí directamente de que el rango es $0 \leq j \leq N$

— **B.2**. Condición de terminación

termina cuando *j* alcanza el valor *N*, según se indica en la postcondición

$\neg B : j = N$

probamos $I \wedge \neg B \Rightarrow Q$

$$
\begin{aligned}
& I \wedge \neg B \\
\Leftrightarrow\ & r = \sum i : 1 \leq i \leq j : u(i) * v(i) \wedge u = U \wedge v = V \wedge 1 \leq j \leq N \wedge j = N \\
\Leftrightarrow\ & r = \sum i : 1 \leq i \leq N : u(i) * v(i) \wedge u = U \wedge v = V \wedge 1 \leq N \leq N \wedge j = N \\
\Rightarrow\ & r = \sum i : 1 \leq i \leq N : u(i) * v(i) \wedge u = U \wedge v = V \\
\Leftrightarrow\ & Q
\end{aligned}
$$

la condición de repetición queda entonces

$B : j \neq N$

— **B.3**. Definición de la expresión de repetición

debemos probar si $I \Rightarrow def(B)$

$def(j \neq N) \Leftrightarrow$ falso

pues la variable *j* no está declarada. Debemos pues incluir una declaración local de variables que incluya al algoritmo que estamos derivando

var j : Ent;
inicio
A
fvar

en cuyo caso

$def(j \neq N) \Leftrightarrow$ cierto $\Leftarrow I$

— **B.4**. Expresión de acotación

j va a ir aumentando hasta alcanzar el valor de N (condición de terminación), para que la expresión de acotación decrezca en cada pasada, tomamos:

$C : N - j$

Probamos que

$I \wedge B \Rightarrow def(C) \wedge dec(C)$

$$
\begin{aligned}
& def(N - j) \\
\Leftrightarrow\ & cierto \\
\Leftarrow\ & I \wedge B
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{dec(N - j)} \\
\Leftrightarrow\ & \texttt{N - j > 0} \\
\Leftrightarrow\ & \texttt{N > j} \\
\Leftarrow\ & \texttt{1} \le \texttt{j} \le \texttt{N} \wedge \texttt{j} \ne \texttt{N} \Leftarrow \texttt{I} \wedge \texttt{B}
\end{aligned}
$$

— **B.5**. Acción de inicialización

comprobamos si $P \Rightarrow I$

pero no es así porque en I aparece

$r = \sum i : 1 \le i \le j : u(i) * v(i)$

y en la precondición ni siquiera aparece *j*, por lo tanto es necesaria una acción de inicialización. Esta será una asignación, y como siempre intentamos que sea de tal forma que se obtenga la condición del invariante por anulación de la "expresión conflictiva". Probamos

$<r,j> := <0,0>$

tendríamos que demostrar entonces

$\{ P \}\ <r,j> := <0,0>\ \{ I \}$

$$
\begin{aligned}
& \texttt{def(0)} \wedge \texttt{def(0)} \wedge \texttt{I[r/0,j/0]} \\
\Leftrightarrow\ & \texttt{u = U} \wedge \texttt{v = V} \wedge \texttt{0 = } \sum \texttt{ i : 1} \le \texttt{i} \le \texttt{0 : u(i) * v(i)} \wedge \texttt{1} \le \texttt{0} \le \texttt{N}
\end{aligned}
$$

el anterior aserto es falso, pues lo es $1 \le 0 \le N$, cambiamos el invariante y tomamos

$I : U = U \wedge v = V \wedge r = \sum i : 1 \le i \le j : u(i) * v(i) \wedge 0 \le j \le N$

esto no invalida las demostraciones hechas hasta ahora, y nos permite seguir con la verificación de la acción de inicialización

$$
\begin{aligned}
& \texttt{pmd(<r,j> := <0,0>, I)} \\
\Leftrightarrow\ & \texttt{u = U} \wedge \texttt{v = V} \wedge \texttt{0 = } \sum \texttt{ i : 1} \le \texttt{i} \le \texttt{0 : u(i) * v(i)} \wedge \texttt{0} \le \texttt{0} \le \texttt{N} \\
\Leftrightarrow\ & \texttt{u = U} \wedge \texttt{v = V} \wedge \texttt{0 = 0} \wedge \texttt{0} \le \texttt{N} \\
\Leftarrow\ & \texttt{u = U} \wedge \texttt{v = V} \wedge \texttt{N} \ge \texttt{1} \Leftrightarrow \texttt{P}
\end{aligned}
$$

otra solución habría sido tomar $<r,j> := <u(1)*v(1), 1>$ como acción de inicialización, en cuyo caso no tendríamos que haber modificado el invariante. Pero es más elegante una solución donde todas las componentes del vector se evalúan dentro del bucle. *Moraleja*: en los bucles sobre vectores hay que ser cuidadoso con los índices. En este ejemplo también cabría la duda de si $j < N$ ó $j \le N$; optamos por la segunda opción para que *I* se siga cumpliendo después de la última pasada por el bucle.

— **B.6**. Acción de avance

Como tenemos que hacer que *j* se acerque a *N* parece evidente que la acción de avance debe ser

A_2: j := j + 1;

con lo cual debemos probar el requisito c.2

$\{ I \wedge B \wedge N–j = T \}\ j := j + 1\ \{ N–j < T \}$

$$
\begin{aligned}
& \texttt{def(j+1)} \wedge \texttt{(N-j < T [j/j+1])} \\
\Leftrightarrow\ & \texttt{cierto} \wedge \texttt{N-(j+1) < T} \\
\Leftrightarrow\ & \texttt{N-j < T + 1} \\
\Leftarrow\ & \texttt{N - j = T} \Leftarrow \texttt{I} \wedge \texttt{B} \wedge \texttt{N-j=T}
\end{aligned}
$$

— **B.7**. Comprobamos si es suficiente con la acción de avance

la pmd de la acción de avance inducida por el invariante

$$I[j/j+1]$$
$$\Leftrightarrow u = U \wedge v = V \wedge r = \sum i : 1 \le i \le j+1 : u(i)*v(i) \wedge 0 \le j + 1 \le N$$
$$\Leftrightarrow u = U \wedge v = V \wedge r = \sum i : 1 \le i \le j : u(i)*v(i)+u(j+1)*v(j+1) \wedge -1 \le j \le N-1$$

que no se puede obtener del invariante, debido al término u(j+1)*v(j+1); por lo tanto se hace necesaria una acción que restablezca el invariante. Tomando el anterior aserto como el aserto intermedio R

— **B.8**. Restablecimiento del invariante.

Tenemos que obtener una acción

$\{ I \wedge B \}\ A_1\ \{ R \}$

lo único que no se puede obtener de R a partir de $I \wedge B$ es el término adicional en el sumatorio; la solución es una asignación a *r*

r := r + u(j+1)*v(j+1)

que efectivamente se puede demostrar que verifica la condición

— **B.9**. Comprobar que se conserva la terminación del bucle

$\{ I \wedge B \wedge N-j = T \}\ A_1 ; A_2\ \{ N-j < T \}$

intuitivamente se ve que se mantiene porque A_1 no afecta a *j*. También se podría demostrar.

Con todo esto el algoritmo anotado queda:

const N = …; % Ent
var u, v : **Vector**[1..N] **de** Real;
  r : Real;
{ P : u = U ∧ v = V ∧ N ≥ 1 }
var j : Ent;
inicio
  <r,j> := <0,0>;
{ I : u = U ∧ v = V ∧ r = ∑ i : 1 ≤ i ≤ j : u(i)*v(i) ∧ 0 ≤ j ≤ N
  C : N – j }
  it j ≠ N →
    { I ∧ j ≠ N }
      r := r + v(j+1) * u(j+1);
    { R }
      j := j + 1
    { I }

fit

{ I ∧ j = N }

fvar

{ Q : u = U ∧ v = V ∧ r = ∑ i : 1 ≤ i ≤ N : u(i) * v(i) }

En cuanto a la complejidad. Si tomamos N como tamaño de los datos (n), considerando el cuerpo del bucle como acción característica es fácil ver que el algoritmo es O(n).

También podemos afirmar que $T_A(n)$ es $\Omega(n)$ pues el algoritmo accede, en cualquier caso, a todas las componentes del vector. Y de las dos afirmaciones anteriores, tenemos que $T_A(n) \in \Theta(n)$.

Podemos razonar también que cualquier algoritmo que resuelva este problema ha de ser $\Omega(n)$, pues necesariamente ha de visitar al menos una vez cada componente. Tenemos pues que la complejidad del algoritmo que hemos diseñado coincide con la cota inferior para el problema; y, por lo tanto, hemos obtenido una complejidad óptima.

En muchos casos se puede establecer la cota inferior razonando sobre que cualquier algoritmo que resuelva el problema debe inspeccionar el dato de entrada completo.

Se puede hacer una generalización de la técnica de sustituir una constante de la postcondición por una variable, que consiste en sustituir una constante por una expresión. Esa es la solución que se utiliza en el cálculo de la raíz cuadrada por defecto en tiempo logarítmico.

1.5.3 Introducción de invariantes auxiliares por razones de eficiencia

En los ejemplos y consideraciones que hemos hecho hasta ahora en la derivación de bucles sólo hemos tenido en cuenta la corrección y no así la eficiencia.

El método expuesto empieza eligiendo cuál es la estructura externa de la acción a diseñar. En los tres primeros casos no tenemos mucho que decir; en la asignación la complejidad dependerá del coste de las operaciones involucradas, en la composición secuencial del coste de las acciones que se componen y en la composición alternativa del coste de evaluar las barreras, que normalmente no admiten muchas posibilidades distintas, y del coste de cada una de las ramas, que vendrá dado por la correspondiente acción que se derive.

El caso más interesante es, otras vez, el de los bucles. Los "peligros" que pueden perjudicar a la eficiencia son:

— derivar más de un bucle por haber diseñado la acción de inicialización como un bucle

— derivar bucles anidados por haber diseñado el restablecimiento del invariante como un bucle.

Estas son las dos situaciones, sobre todo la segunda, que debemos evitar.

Para resolver el primer problema debemos elegir acciones de inicialización que contengan expresiones simples.

Para resolver el segundo problema en muchas ocasiones es útil la técnica de introducir un invariante auxiliar.

En un punto de la derivación, normalmente al intentar restablecer el invariante, nos encontramos con que tenemos que obtener el valor de una expresión compleja, típicamente, una expresión que incluye asertos u operaciones extendidos. La solución consiste en conseguir que el bucle además de lo que ya calculaba, indicado por el invariante, vaya calculando también esta expresión. Introducimos una nueva variable *y*, de forma que donde se necesite el valor de la expresión E podamos simplemente tomar el valor de *y*. Reforzamos el invariante con un invariante auxiliar de la forma:

$I_1 : y = E$

con lo que el invariante quedaría

$I : I_0 \wedge I_1$

Con este nuevo invariante será necesario rehacer una parte de la derivación, aunque otras partes pueden seguir siendo válidas, teniendo en cuenta que el nuevo invariante será un fortalecimiento del antiguo.

Veamos un ejemplo donde se necesita el uso de un invariante auxiliar: el cálculo de los números de Fibonacci.

$fib(0) = 0$

$fib(1) = 1$

$fib(n+2) = fib(n+1) + fib(n) \qquad$ si $n \geq 0$

---

partimos de la especificación

---

var n, x : Ent;

{ $P : n = N \wedge n \geq 0$ }

fibonacci

{ $Q : n = N \wedge x = fib(n)$ }

Los números de Fibonacci no se pueden especificar utilizando el lenguaje de asertos. Esto ocurre con muchas funciones definidas de forma recursiva. Entendemos en ese caso que la especificación se complementa con la definición recursiva de la función, que hemos escrito más arriba, y que especifica el comportamiento de dicha función.

La idea del algoritmo es ir obteniendo los números de Fibonacci para valores sucesivos hasta llegar al valor de n, uno en cada pasada por un bucle.

— **B.1**. Obtención del invariante

como *n* se comporta como una constante, cambiamos *n* por una variable; añadiendo además la condición de que se conserve el valor de *n*

$I_0 : x = fib(i) \wedge n = N \wedge 0 \leq i \leq n$

(lo llamamos I_0 porque ya sabemos que introduciremos un invariante auxiliar)

donde además hemos incluido un "rango razonable" para la nueva variable.

— **B.2**. Condición de terminación

El invariante será igual a la postcondición cuando i = n, tomamos pues esa condición de terminación

$\neg B : i = n$

que efectivamente verifica

$I_0 \wedge \neg B \Rightarrow Q$

de ahí que la condición de repetición quede

$B : i \neq n$

— **B.3**. Definición de la condición de repetición

La condición de repetición no está definida porque la variable *i* no está declarada. Incluimos por tanto una declaración local que rodea a la acción a derivar

var i : Ent;
inicio
A
fvar

con lo que ahora sí está definida $i \neq n$

— **B.4**. Expresión de acotación.

Fijándonos en la condición de terminación, i = n, y considerando que *i* debe ir incrementándose tomamos como expresión de acotación

$C : n - i$

que junto con la condición sobre el rango de *i* que aparece en el invariante nos asegura que esta expresión toma valores en un dominio bien fundamentado.

Probamos entonces

$I_0 \wedge B \Rightarrow def(C) \wedge dec(C)$

$$\begin{aligned} & def(\ n - i\ ) \\ \Leftrightarrow\ & cierto \\ \Leftarrow\ & I_0 \wedge B \end{aligned}$$

$$\begin{aligned} & dec(\ n-i\ ) \\ \Leftrightarrow\ & n - i > 0 \\ \Leftrightarrow\ & n > i \\ \Leftarrow\ & 0 \leq i \leq n \wedge i \neq n \Leftarrow I_0 \wedge B \end{aligned}$$

— **B.5**. Acción de inicialización

Observamos que el invariante no se puede obtener de la precondición y que es necesaria una acción de inicialización

$$\{ n = N \wedge n \geq 0 \}$$
$$A_0$$
$$\{ n = N \wedge x = fib(i) \wedge 0 \leq i \leq n \}$$

como siempre intentamos que en la inicialización se haga el menor trabajo posible. Esto se consigue con

$$<x,i> := <0,0>$$

que cumple la especificación anterior.

— **B.6**. Acción de avance

Como queremos ir obteniendo los sucesivos números de Fibonacci hasta alcanzar fib(n), vamos a hacer que *i* vaya incrementándose de 1 en 1:

$$A_2 : i := i + 1$$

Demostramos que efectivamente esta acción de avance hace avanzar el bucle

$$\{ I_0 \wedge B \wedge N - i = T \}\ i := i + 1\ \{ N - i < T \}$$

que efectivamente podemos demostrar

— **B.7**. ¿Es suficiente con la acción de avance?

Se debería cumplir

$$\{ I_0 \wedge B \}\ i := i + 1\ \{ I_0 \}$$

o lo que es lo mismo

$$I_0 \wedge B \Rightarrow I_0\ [i/i+1] \Leftrightarrow x = fib(i+1) \wedge 0 \leq i + 1 \leq n \wedge n = N$$

pero no se puede demostrar

$$I_0 \wedge B \Rightarrow x = fib(i+1)$$

— **B.8** Restablecimiento del invariante

Tenemos

$$\{ n = N \wedge x = fib(i) \wedge 0 \leq i \leq n \wedge i \neq n \}$$
$$A_1$$
$$\{ n = N \wedge x = fib(i+1) \wedge 0 \leq i+1 \leq n \}$$

la solución sería una acción de la forma

$$x := fib(i+1)$$

[o lo que es igual

$$x := fib(i) + fib(i-1)$$

fib(i) lo tenemos, pero para obtener fib(i–1) tendríamos que escribir otro bucle que lo escribiera]

lo que hacemos es añadir un invariante auxiliar, exigiendo que el bucle vaya calculando también el valor de fib(i+1)

$$I_1 : y = fib(i+1)$$

con lo que el nuevo invariante queda

$$I \Leftrightarrow I_0 \wedge I_1 \Leftrightarrow n = N \wedge x = fib(i) \wedge 0 \leq i \leq n \wedge y = fib(i+1)$$

y con el nuevo invariante reiniciamos la derivación:

— **B.2**. La condición de repetición es la misma y se sigue cumpliendo

$$I \wedge \neg B \Rightarrow Q$$

ya que I es un fortalecimiento de I_0.

— **B.3**. La condición de repetición sigue estando definida

$$I \Rightarrow I_0 \Rightarrow def(B)$$

— **B.4**. La expresión de acotación sigue siendo la misma, y se siguen cumpliendo sus propiedades.

$$I \wedge B \Rightarrow def(C) \wedge dec(C)$$

— **B.5**. Es necesario modificar la acción de inicialización.

Por una parte es necesario declarar la variable local *y*

var i, y : Ent;
inicio
A
fvar

La acción de inicialización, aceptando que inicializamos *i* a 0

$\{ n = N \wedge n \geq 0 \}$
inicialización
$\{ n = N \wedge x = fib(0) \wedge 0 \leq 0 \leq n \wedge y = fib(1) \}$

Lo cual se puede derivar como una asignación múltiple

A_0 : <i,x,y> := <0,0,1>

y demostrar que efectivamente es correcto $\{ P \}\ A_0\ \{ I \}$

— **B.6**. La acción de avance sigue siendo la misma, y se sigue cumpliendo el avance del bucle pues hemos construido un reforzamiento de la precondición.

— **B.7**. De nuevo comprobamos que no es suficiente con la acción de avance.

— **B.8**. Restablecimiento del invariante

$\{ n = N \wedge x = fib(i) \wedge 0 \leq i \leq n \wedge y = fib(i+1) \wedge i \neq n \}$
A_1
$\{ n = N \wedge x = fib(i+1) \wedge 0 \leq i+1 \leq n \wedge y = fib(i+2) \}$

y ahora en la precondición sí tenemos los valores necesarios para obtener las igualdades de la postcondición:

<x,y> := <y,y+x>

que podemos verificar que es efectivamente correcta respecto de la especificación

— **B.9**. Demostramos que con la acción de restablecimiento del invariante se sigue cumpliendo la terminación del bucle.

Informalmente podemos argumentar que la expresión de acotación (n–i) no se ve afectada por la acción A_1.

Finalmente, el algoritmo anotado nos queda

```
var n, x : Ent;
{ n = N ∧ n ≥ 0 }
var
  i, y : Ent;
inicio
  <i,x,y> := <0,0,1>;
{ I : n = N ∧ x = fib(i) ∧ 0 ≤ i ≤ n ∧ y = fib(i+1)
  C : n – i                                          }
  it i ≠ n →
  { I ∧ i ≠ n }
     <x,y> := <y,x+y>;
  { I[i/i+1] }
     i := i +1
   { I }
  fit
{ I ∧ i = n }
fvar
{ n = N ∧ x = fib(n) }
```

Este ejemplo no nos debe hacer pensar que siempre es posible eliminar los bucles anidados introduciendo invariantes auxiliares. Es necesario que el bucle externo sea realmente capaz de obtener el valor para el cual se requiere un bucle interno, y para ello necesitamos que ambos bucles tengan los mismos límites y se ejecuten el mismo número de veces.

1.6 Algoritmos de tratamiento de vectores

Vamos a tratar en este apartado un conjunto de algoritmos muy habituales sobre vectores. Estos algoritmos son muy utilizados porque representan operaciones habituales sobre colecciones de datos y los vectores son la estructura de datos más habitual para almacenar colecciones de datos de un mismo tipo.

Los vectores se caracterizan porque tienen **tamaño fijo**, es necesario determinar en el momento de crearlos cuál es su tamaño (aunque esto no es cierto en lenguajes más modernos, donde los vectores se crean en ejecución, pudiéndose determinar entonces el tamaño; de esta forma es

posible implementar vectores que "pueden crecer", mediante la creación de un vector de mayor longitud y la copia de las componentes del vector antiguo). Y la otra característica es que pemiten el **acceso directo** a cada una de las componentes, a través de un índice. Para que el acceso directo sea eficiente, es necesario que todo el vector esté almacenado en memoria principal, lo cual limita el tamaño máximo. En la segunda parte del curso veremos otras formas de manejar colecciones de datos que soslayan los inconvenientes de los vectores, a costa de sacrificar el acceso directo, susituyéndolo por **acceso secuencial**.

— tamaño fijo

— acceso directo ⇒ todo el vector en memoria principal

Casi todos los algoritmos de tratamiento de vectores se ajustan a uno de los dos siguientes esquemas

---

— Recorrido. Se procesan todos los elementos del vector

— Búsqueda. Se trata de encontrar la primera componente que verifica una cierta propiedad.

El tercer grupo de algoritmos que trataremos será el de los algoritmos de ordenación que no consideramos como un esquema general sino como una operación concreta.

1.6.1 Recorrido

El esquema de recorrido tiene la siguiente estructura

---

18. Seleccionar la primera componente a tratar
19. Mientras no se hayan tratado todas las componentes
    2.1 Procesar componente
    2.2 Seleccionar la siguiente componente a tratar
20. Tratamiento final

Vamos a escribir el algoritmo de recorrido en función de una acción genérica *tratar*

```
var
  v : Vector[1..N] de elem;
  …                          % otras declaraciones
{ P : v = V ∧ N ≥ 1 }
  n := 0;
{ I : 0 ≤ n ≤ N ∧ ∀i : 1 ≤ i ≤ n : tratado(v(i))
```

```
    C : N-n                                          }
    it n ≠ N
    →  tratar(v(n+1));
       n := n+1
    fit;
    finalización
{ Q : ∀i : 1 ≤ i ≤ N : tratado(v(i)) }
```

La complejidad en $O(n)$.

Nótese que no imponemos en la postcondición que se conserve el valor de *v* porque puede ocurrir que la acción *tratar* modifique el valor de las componentes del vector.

En los ejercicios ya hemos derivado ejemplos similares. Para hacer la derivación formal de un algoritmo genérico de recorrido, el invariante se obtiene sustituyendo una constante de la postcondición por una variable.

Normalmente los bucles de recorrido de vectores se escriben utilizando la construcción **para**, lo que permite una escritura más compacta.

No hemos escrito el esquema como un procedimiento o una función porque al no conocer cuál es la función de tratamiento no sabemos si el vector se modificará, o si se deberán obtener resultados adicionales. Sin embargo, una vez fijada la operación de tratamiento de las componentes, lo habitual es implementar el recorrido como una función o un procedimento según corresponda.

Algunas variaciones sobre este esquema básico:

- Recorrido de un subvector. Lo habitual es implementar el recorrido como un procedimiento o una función que reciba como parámetro los límites del subvector a recorrer.
- Recorrido de posiciones no consecutivas. Es aconsejable disponer de una función que nos permita obtener el siguiente índice a partir del actual (acción de avance).
- Forma del resultado y los argumentos.
  - El resultado es un valor que se obtiene a partir del vector de entrada, que no se modifica (p.e. obtener el máximo de las componentes)
  - El resultado es el vector de entrada modificado (p.e. todas las componentes a 0)
  - El resultado es otro vector obtenido a partir del vector de entrada (p.e. copia de un vector).
  - Se recorre más de un vector en paralelo (p.e. producto escalar)

1.6.2 Búsqueda

Bajo esta categoría encontramos a un gran número de algoritmos sobre vectores. El esquema general tiene la siguiente forma:

21. Seleccionar la primera componente donde buscar
22. Mientras no se termine y no sea encontrada la propiedad
    2.1. Comprobamos si la componente verifica la propiedad
        2.1.1 Si es así, ha sido encontrada
        2.1.2 Si no es así, seleccionamos la siguiente componente donde buscar
23. Tratamiento final

Búsqueda secuencial

El esquema de búsqueda secuencial queda en función de una cierta operación *P* que nos dice si una componente del vector cumple o no la propiedad buscada:

```
func buscarVector( v : Vector[1..N] de elem ) dev encontrado : Bool; pos :
Ent;
{ P₀ : v = V ∧ N ≥ 1 }
inicio
  <pos,encontrado> := <1,P(v(1))>;
{ I : v = V ∧ 1 ≤ pos ≤ N ∧ ∀i : 1≤i<pos : NOT P(v(i)) ∧ encontrado ↔
P(v(pos))
  C : N – pos
}
  it pos ≠ N AND NOT encontrado
  →  encontrado := P(v(pos+1));
     pos := pos + 1
  fit;
{ Q₀ : v = V ∧ 1 ≤ pos ≤ N ∧ ∀i : 1≤i<pos : NOT P(v(i)) ∧ encontrado ↔
P(v(pos))∧
     (pos = N ∨ encontrado)
}
  dev <encontrado, pos>
ffunc
```

Nótese que si *P* es una función, está permitido utilizarla en los asertos, como ya vimos en el apartado sobre procedimientos y funciones. Nótese también que debemos entender esa función como el resultado de sustituir los parámetros formales por los reales en la postcondición de la función; y, por lo tanto, si queremos relacionar ese aserto con la variable *encontrado* debemos hacerlo con el operador ↔.

Es necesario sacar la comprobación de la primera componente fuera del bucle para que el invariante esté definido a la entrada pues si inicializásemos pos=0 y encontrado=falso tendríamos

```
encontrado ↔ P(v(0))
```

Donde P(v(0)) no está definido porque no lo está v(0).

Podríamos escribir el bucle sin utilizar la variable auxiliar *encontrado* pero eso plantea dos problemas:

— hace necesario evaluar una vez más P(v(pos)) para comprobar si hemos acabado por llegar al final o por que se cumple la propiedad. Si la comprobación de la propiedad es costosa esto implica una penalización de la eficiencia. Aunque realmente en el caso peor no es significativo pues sólo implica una comprobación más sobre las N posibles.

— algunos lenguajes de programación (como Pascal) evalúan las condiciones enteras aunque puedan llegar a un resultado sólo con evaluar una parte. Hay que tener cuidado de que no pueda ocurrir que se intenta evaluar P(v(pos)) para un valor de *pos* que no pertenezca al rango del vector. **No es el caso aquí**.

Otra posible forma de escribir el algoritmo sin necesidad de comprobar la primera componente fuera del bucle:

```
func buscarVector( v : Vector[1..N] de elem ) dev encontrado : Bool; pos :
Ent;
{ P₀ : v = V ∧ N ≥ 1 }
inicio
  pos := 1;
{ I : v = V ∧ 1 ≤ pos ≤ N ∧ ∀i : 1≤i<pos : NOT P(v(i))
  C : N – pos                                                    }
  it pos ≠ N AND NOT P(v(pos))
  → pos := pos + 1
  fit;
  encontrado := P(v(pos));
{ Q₀ : v = V ∧ 1 ≤ pos ≤ N ∧ ∀i : 1≤i<pos : NOT P(v(i)) ∧ encontrado ↔
P(v(pos))∧
      (pos = N ∨ encontrado)
}
  dev <encontrado, pos>
ffunc
```

La complejidad en el caso peor es *O*(n).

Este algoritmo se puede derivar de manera muy sencilla. El invariante se obtiene directamente de la postcondición, separando el efecto de la asignación (encontrado ↔ P(v(pos))) y la condición de terminación que también aparece explícitamente en la postcondición (pos=N ∨ encontrado). En el segundo esquema la propia acción de avance restablece el invariante. En el otro ejemplo es necesario incluir una asignación a *encontrado*.

Búsqueda secuencial con centinela

Es una variación de la búsqueda secuencial donde nos aseguramos de al menos un elemento del vector cumple la propiedad; normalmente el último del vector. De esta forma simplificamos la condición de repetición del bucle pues no tenemos que preocuparnos por si nos salimos de los límites del vector.

```
func buscarVector( v : Vector[1..N] de elem ) dev encontrado : Bool; pos :
Ent;
```

```
{ P₀ : v = V ∧ 1 ≤ M ≤ N ∧ P(v(M)) }
inicio
  pos := 1;
{ I : v = V ∧ 1 ≤ pos ≤ M ∧ ∀i : 1≤i<pos : NOT P(v(i))
  C : M – pos                                                    }
  it NOT P(v(pos))
  → pos := pos + 1
  fit;
  encontrado := pos ≠ M;
{ Q₀ : v = V ∧ 1 ≤ pos ≤ M ∧ ∀i : 1≤i<pos : NOT P(v(i)) ∧ P(v(pos)) ∧
      encontrado ↔ pos ≠ M                                                }
  dev <encontrado, pos>
ffunc
```

El coste de este algoritmo es *O*(M).

Para derivarlo tenemos que el invariante aparece explícitamente en la postcondición (∀i : 1≤i<pos : NOT P(v(i))) así como la condición de terminación (P(v(pos))), y el efecto de la asignación (encontrado ↔ pos ≠ M).

La búsqueda con centinela es útil por ejemplo en una búsqueda dentro de un vector ordenado donde la condición de parada será

```
v(pos) ≥ x
```

Si el valor del centinela es el mayor posible dentro del correspondiente dominio, entonces sabemos que siempre se cumplirá la condición.

También se puede utilizar el centinela como una marca que indica donde está el final de la parte de un vector que contiene información. De forma que utilicemos sólo las *M* primeras posiciones de un vector. Esta es un técnica muy habitual. Como no podemos esperar que el centinela cumpla cualquier propiedad sobre la que podamos estar interesados, tendríamos que usar una condición de repetición de la forma:

```
NOT P(v(pos)) AND NOT P'(v(pos))
```

siendo P la propiedad buscada y P' una propiedad conocida para el centinela, que sólo él puede verificar dentro del dominio correspondiente.

Aplicaciones de la búsqueda secuencial

Muchos algoritmos sobre vectores se pueden expresar utilizando el esquema de búsqueda secuencial –con o sin centinela–. Algunos ejemplos donde se trata de determinar si uno o más vectores cumplen una determinada propiedad y lo que se hace es buscar una componente que no cumpla la propiedad.

— Determinar si dos vectores son iguales

  P : u(pos) ≠ v(pos)

— Determinar si una matriz es simétrica

  P : v(f,c) ≠ v(c,f)

— Determinar si un vector de caracteres es un palíndromo (es igual leída de izquierda a derecha y de derecha a izquierda).

  P : v(pos) ≠ v(N–pos+1)

  El bucle sólo recorrería el vector hasta N div 2

— Determinar si un vector está ordenado crecientemente

  P : v(pos) > v(pos+1)

  El recorrido sólo llegaría, como máximo, hasta N–1.

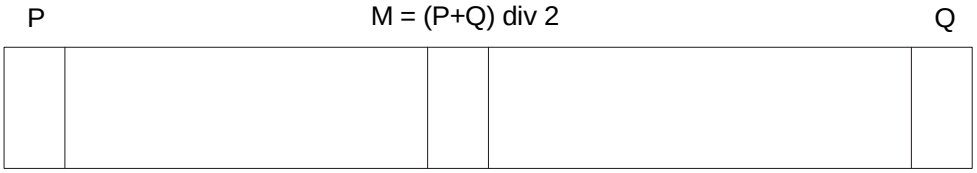
Búsqueda binaria o dicotómica

Hasta ahora no nos hemos aprovechado del acceso directo que ofrecen los vectores. Este es el primer caso donde la utilizamos.

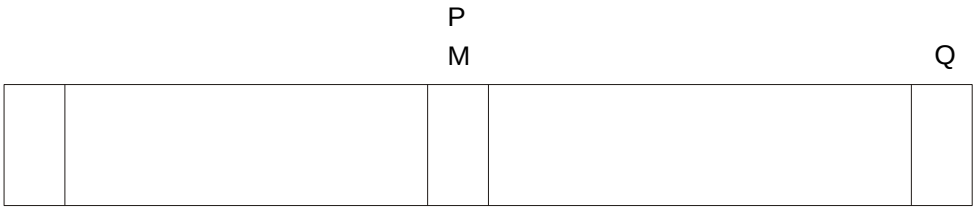
El algoritmo de búsqueda binaria sirve para

Encontrar un valor dentro de un **vector ordenado**

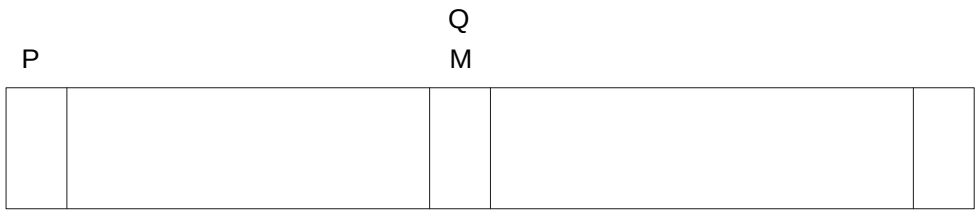
La idea es que en cada pasada por el bucle se reduce a la mitad el tamaño del subvector donde puede estar el elemento buscado.



si v(m) ≤ x entonces debemos buscar a la derecha de *m*. En realidad podríamos considerar el caso v(m) = x como un caso especial que nos hace terminar. Sin embargo, para simplificar la derivación, consideramos que siempre se llega al punto en que q=p+1; planteando el uso de una variable auxiliar *encontrado*, como una optimización posterior.



y si v(m) > x entonces debemos buscar a la izquierda de *m*



Como el tamaño de los datos se reduce a la mitad en cada pasada tenemos claramente una complejidad logarítmica en el caso peor, frente a la complejidad lineal de los algoritmos de búsqueda secuencial. El caso peor es cuando el elemento no está el vector, o tenemos la mala suerte de no encontrarlo hasta que p=q.

El tipo de los elementos del vector debe ser un tipo sobre el que esté definida la igualdad y una relación de orden

```
tipo
  elem = … ;     % tipo ordenado con igualdad
```

Debemos tener cuidado con los casos extremos, es decir si *x* es menor que todos los elementos del vector o si *x* es mayor que todos los elementos del vector. Queremos que si no se encuentra en el vector *pos* se quede apuntando a la posición anterior a la que debería ocupar el elemento; eso quiere decir que tanto 0 como N son valores admisibles para *pos*. Esto plantea un problema a la hora de escribir la postcondición; la forma más simple de escribir la postcondición es:

```
v = V ∧ ord(v) ∧ 0 ≤ pos ≤ N ∧ v(pos) ≤ x < v(pos+1)
```

Donde no hace falta indicar que todos los elementos a la izquierda de *pos* son menores que *x* y todos los elementos a la derecha son mayores, basta con que lo sean el apuntado por *pos* y el inmediatamente siguiente, ya que los demás lo serán por estar ordenado el vector. El problema es que *pos* puede valer 0, en cuyo caso no está definido v(pos), o puede valer *N* en cuyo caso no está definido v(pos+1). Se puede utilizar un artificio en la especificación que consiste en considerar un vector ficticio con valores *centinela*, $-\infty$ y $+\infty$, en las posiciones 0 y N+1, y que coincide con *v* en las demás posiciones. O bien, nos podemos aprovechar de que el cuantificador universal se considera cierto cuando el aserto de dominio define un conjunto vacío. Aplicando esta segunda idea obtenemos la siguiente especificación

```
func buscarVectorBin( v : Vector[1..N] de elem; x : elem ) dev pos : Ent;
{ P₀ : v = V ∧ x = X ∧ N ≥ 1 ∧ ord(v) }
{ Q₀ : v = V ∧ x = X ∧ ∀i : 1 ≤ i ≤ pos : v(i) ≤ x ∧ ∀i : pos+1 ≤ i ≤ N :
x<v(i) ∧
       0 ≤ pos ≤ N }
ffunc
```

Vamos a derivar el algoritmo de búsqueda binaria a partir de esta especificación

— **B.1**. Invariante

Para obtener el invariante construimos una generalización de la postcondición, teniendo en cuenta la idea de que vamos a ir acotando el subvector dentro del cual puede estar el elemento buscado, hasta llegar a un subvector con una sola componente: desde *pos* a *pos*+1. La generalización consiste en definir el subvector a explorar como el que va desde *pos* hasta el valor de una nueva variable *q*.

$$I:\ v = V \land x = X \land \forall i : 1 \le i \le pos : v(i) \le x \land \forall i : q \le i \le N : v(i) > x \land$$
$$0 \le pos < q \le N+1 \land ord(v)$$

La condición ord(v) es necesario para demostrar la corrección. La condición pos < q es necesaria para demostrar la terminación.

— **B.2**. Condición de terminación

Fijándonos en las diferencias entre el invariante y la postcondición tenemos que la condición de terminación ha de ser:

$$\neg B : q = pos+1$$

Efectivamente podemos demostrar

$$I \land \neg B \Rightarrow Q_0$$

La condición de repetición queda por tanto

$$B : q \neq pos+1$$

— **B.3**. ¿Está definida la condición de repetición?

$$I \Rightarrow def(q \neq pos+1)$$

No está definida porque *q* no está declarada. Añadimos esta nueva variable a las declaraciones locales de la función

```
var q : Ent;
```

— **B.4**. Expresión de acotación.

La idea es que en cada pasada se ha de reducir el tamaño del subvector a considerar, hasta llegar a un subvector de longitud 1. Este subvector viene dado por la diferencia entre *pos* y *q*. Por lo tanto una posible expresión de acotación:

```
C : q - pos
```

Demostramos

```
I ∧ B ⇒ def(C) ∧ dec(C)
```

— **B.5**. Acción de inicialización

Incialmente el subvector candidato es el vector entero, además podemos argumentar que elegimos los valores de *pos* y *q* para conseguir que se anulen los correspondientes cuantificadores universales:

```
A₀:     <pos,q> := <0,N+1>;
```

Con lo cual es trivial demostrar

```
{ P₀ } A₀ { I }
```

— **B.6**. Acción de avance

En la acción de avance es donde está la parte *ingeniosa* del algoritmo. En cada iteración queremos reducir el máximo posible el tamaño del subvector a explorar; como consideramos que es igualmente probable que el valor buscado se encuentre en cualquiera de las posiciones del subvector, tomamos el punto medio entre *pos* y *q*, de forma que reduzcamos a la mitad el tamaño del subvector a explorar.

```
A₂ :    m := (pos+q) div 2;
```

y asignamos *m* a *pos* o *q* según convenga para mantener el invariante:

```
si
    v(m) ≤ x → pos := m;
  □ v(m) > x → q := m
fsi
```

Demostremos ahora el avance del bucle:

```
{ I ∧ B ∧ q-pos = T } A₀ { q-pos < T }
```

Para hacerlo tomamos como hipótesis un aserto intermedio de la forma:

```
{ I ∧ B ∧ q-pos = T }
  m := (pos+q) div 2;
```

```
{ R₁ ≡ I ∧ B ∧ q-pos = T ∧ m = (pos+q) div 2 }
  si
      v(m) ≤ x → pos := m;
    □ v(m) > x → q := m
  fsi
{ q-pos < T }
```

De esta forma es trivial la corrección de la primera asignación y lo que nos queda es demostrar la corrección de la composición alternativa.

Definición de las barreras. Añadimos *m* a la declaración de variables locales

```
    def(v(m) ≤ x) ∧ def(v(m) > x)
⇔   enRango(v,m)
⇔   1 ≤ m ≤ N
⇐   0 ≤ pos < m < q ≤ N+1
⇐   0 ≤ pos < q ≤ N+1 ∧ m = (pos+q) div 2 ∧ q ≠ pos+1
⇐   R₁
```

y al menos una abierta

```
v(m) ≤ x ∨ v(m) > x ⇔ cierto ⇐ R₁
```

ahora verificamos cada rama por separado

```
{ R₁ ∧ v(m) ≤ x } pos := m { q − pos < T }

    q − pos < T [pos/m]
⇔   q − m < T
⇐   q − pos = T ∧ pos < m
⇐   q − pos = T ∧ pos < q ∧ q ≠ pos+1 ∧ m = (pos+q)
div 2
⇐   R ∧ v(m) ≤ x

{ R₁ ∧ v(m) > x } q := m { q − pos < T }

    q − pos < T [q/m]
⇔   m − pos < T
⇐   q − pos = T ∧ m < q
⇐   q − pos = T ∧ pos < q ∧ q ≠ pos+1 ∧ m = (pos+q)
div 2
⇐   R₁ ∧ v(m) > x
```

— **B.7**. ¿Es suficiente con la acción de avance?

```
{ I ∧ B }
  m := (pos+q) div 2;
{ R₂ ≡ I ∧ B ∧ m = (pos+q) div 2 }
  si
      v(m) ≤ x → pos := m;
    □ v(m) > x → q := m
  fsi
{ I }
```

La verificación de la primera asignación es trivial.
La verificación del condicional

$$def(v(m) \le x) \land def(v(m) > x) \Leftarrow R_2$$

se demuestra de la misma forma que el punto anterior

$$v(m) \le x \lor v(m) > x \Leftrightarrow cierto \Leftarrow R_2$$

demostramos ahora cada rama

$$\{ R_2 \land v(m) \le x \}\ pos := m\ \{ I \}$$

$$I\,[pos/m]$$
$$\Leftrightarrow v = V \land x = X \land 0 \le m < q \le N+1 \land \forall i : 1 \le i \le m : v(i) \le x \land \forall i : q \le i \le N : v(i) > x \land ord(v)$$

$$I \Rightarrow v = V \land x = X \land ord(v)$$
$$I \Rightarrow \forall i : q \le i \le N : v(i) > x$$
$$I \land m = (pos+q) \text{ div } 2 \land q \ne pos+1 \Rightarrow 0 \le m < q \le N+1$$
$$ord(v) \land v(m) \le x \Rightarrow \forall i : 1 \le i \le m : v(i) \le x$$

por lo tanto $R_2 \land v(m) \le x \Rightarrow I[pos/m]$

$$\{ R_2 \land v(m) > x \}\ q := m\ \{ I \}$$

$$I\,[q/m]$$
$$\Leftrightarrow v = V \land x = X \land 0 \le pos < m \le N+1 \land \forall i : 1 \le i \le pos : v(i) \le x \land \forall i : m \le i \le N : v(i) > x \land ord(v)$$

$$I \Rightarrow v = V \land x = X \land ord(v)$$
$$I \Rightarrow \forall i : 1 \le i \le pos : v(i) \le x$$
$$I \land m = (pos+q) \text{ div } 2 \land q \ne pos+1 \Rightarrow 0 \le pos < m \le N+1$$
$$ord(v) \land v(m) > x \Rightarrow \forall i : m \le i \le N : v(i) > x$$

por lo tanto $R_2 \land v(m) > x \Rightarrow I[q/m]$

Con todo esto quedaría demostrada la corrección del programa, que finalmente será

```
func buscarVectorBin( v : Vector[1..N] de elem; x : elem ) dev pos : Ent;
{ P₀ : v = V ∧ x = X ∧ N ≥ 1 ∧ ord(v) }
var
  q, m : Ent;
inicio
  <pos,q> := <0,N+1>;
{ I : x = X ∧ v = V ∧ ord(v) ∧ 0 ≤ pos < q ≤ N+1 ∧ ∀i : 1 ≤ i ≤ pos : v(i)
≤ x ∧
      ∀i : q ≤ i ≤ N : v(i) > x
  C : q – pos
}
  it q ≠ pos+1
  →  m := (pos+q) div 2;
      si
        v(m) ≤ x → pos := m;
      □ v(m) > x → q := m
      fsi
  fit;
{ Q₀ : v = V ∧ x = X ∧ ∀i : 1 ≤ i ≤ pos : v(i) ≤ x ∧ ∀i : pos+1 ≤ i ≤ N :
x<v(i) ∧
       0 ≤ pos ≤ N }
  dev pos
ffunc
```

Nótese que efectivamente *pos* se queda apuntando a la posición anterior al primer elemento que es más grande que *x*, de forma que desplazando una posición hacia la derecha todas las componentes i>pos tendríamos el hueco donde insertar *x*.

Nótese, por último, que habremos encontrado el elemento si pos ≠ 0 y v(pos) = x:

```
(∃i : 1 ≤ i ≤ N : v(i) = x) ↔ (pos ≠ 0 ∧ v(pos) = x)
```

En cuanto a la complejidad, tomando *N* como tamaño de los datos, tenemos que en cada iteración se divide a la mitad la longitud del subvector a considerar, hasta que se llega a un subvector de longitud 1.

1.6.3 Ordenación

Los algoritmos de ordenación sobre vectores son muy importantes dado lo habitual de su uso. La especificación de un procedimiento de ordenación nos dice que, dado un vector cuyos elementos son de un tipo ordenado, hemos de obtener una permutación ordenada (en orden no decreciente, por ejemplo) del vector original. Formalmente:

```
proc ordenaVector( es v : Vector[1..N] de elem );
{ P₀ : v = V ∧ N ≥ 1 }
{ Q₀ : ord(v) ∧ perm(v,V) }
fproc
```

donde los asertos `ord(v)` y `perm(v,V)` tienen el siguiente significado:

```
ord(v) ⇔ ∀i,j : 1 ≤ i < j ≤ N : v(i) ≤ v(j)

perm(v,V) ⇔ ∀i : 1 ≤ i ≤ N : (#j : 1 ≤ j ≤ N : v(j) = v(i)) =
                              (#j : 1 ≤ j ≤ N : V(j) = v(i))
```

Operación de intercambio entre posiciones de un vector

Varios algoritmos de ordenación y, en particular, los que vamos a estudiar en este capítulo, basan su funcionamiento en realizar intercambios entre las componentes del vector. Por ello, vamos a introducir una nueva sentencia en el lenguaje algorítmico que nos permita escribir de una vez dicho intercambio.

```
Int( v, Eᵢ, Eⱼ )
```

es equivalente a

```
<v(Eᵢ), v(Eⱼ)> := <v(Eⱼ), v(Eᵢ)>
```

y, a efectos de verificación es más cómodo considerar la equivalencia

```
var
  z : elem;
inicio
  z := valor(v,Eᵢ);
  v := asignar(v, Eᵢ, valor(v, Eⱼ));
  v := asignar(v, Eⱼ, z);
fvar
```

Para no recargar la verificación de los algoritmos de ordenación nos interesa observar que si la única modificación que hacemos en un vector es a través de la operación de intercambio, el vector resultante va a ser una permutación del vector original. Es decir, se puede demostrar

```
{ v = V }
  Int(v, Eᵢ, Eⱼ)
{ perm(v,V) }
```

Para demostrarlo se utiliza la segunda de las equivalencias que hemos visto más arriba. Se deja al lector encontrar esta demostración.

De esta forma, nos despreocupamos de verificar las operaciones de intercambio y obviamos la condición perm(v,V) en la postcondición de los algoritmos de ordenación.

A los algoritmos de ordenación que sólo modifican el vector por medio de intercambios se les llama **basados en intercambios**.

Cota inferior para los algoritmos de ordenación basados en intercambios

Tenemos el siguiente resultado sobre la complejidad mínima de los algoritmos de ordenación basados en intercambios:

Sea *A* un algoritmo de ordenación de vectores basado en intercambios, sea $T_A(n)$ su complejidad en tiempo en el caso peor, tomando como tamaño de los datos *n* la longitud del vector. Se verifica:

(a) $T_A(n) \in \Omega(n \cdot \log n)$

(b) $T_A(n) \in \Omega(n^2)$, en el caso de que *A* sólo efectúe intercambios de componentes vecinas.

Para demostrar (a) debemos razonar sobre el número de intercambios que es necesario realizar en el caso peor. Realizando un intercambio podemos generar 2 permutaciones distintas (realizar el intercambio o no realizarlo), con dos intercambios podemos alcanzar $4=2^2$ permutaciones, con tres intercambios $8=2^3$ permutaciones, y así sucesivamente, de forma que con *t* intercambios podemos obtener 2^t permutaciones distintas. En cada iteración decidimos si hacemos o no el intercambio con lo que escogemos una permutación y desechamos todas las demás; en el caso peor deberemos haber escogido una permutación y haber desechado un número de permutaciones igual o mayor que el número de permutaciones total, n!. De esta forma tenemos:

$$2^t \geq n!$$

$$\Rightarrow t \geq \log(n!)$$

por la fórmula de Stirling

$$\Rightarrow t \geq c \cdot n \cdot \log n$$

si hemos realizado *t* operaciones de intercambio entonces la complejidad del algoritmo debe ser

$$T_A(n) \geq t \geq c \cdot n \cdot \log n$$

y aplicando la definición de la cota inferior de la complejidad

$$T_A(n) \in \Omega(n \cdot \log n)$$

El anterior razonamiento puede hacerse en base al número de comparaciones o en base al número de intercambios. En el caso peor debemos haber hecho $2^t \geq n!$ comparaciones para así haber considerado todas las permutaciones posibles; pero podemos argumentar, si lo que contamos son las operaciones de intercambio, que el caso peor hemos hecho un número de intercambios igual al número de comparaciones.

Para demostrar la parte (b) de la anterior proposición vamos a definir una medida del grado de desorden de un vector por medio del concepto de inversiones

---

$inv(v, i, j) \Leftrightarrow_{def} i < j \wedge v(i) > v(j)$

tenemos entonces, que un vector estará ordenado si no existen inversiones en él

$ord(v) \Leftrightarrow (\#i,j : 1 \leq i < j \leq n : inv(v, i, j)) = 0$

el caso peor se tendrá cuando el vector esté ordenado en orden inverso, en cuyo caso se hará máximo el número de inversiones:

$(\#i,j : 1 \leq i < j \leq n : cierto = \sum i : 1 \leq i < n : n-i \geq c \cdot n^2$

(que es el número de pares i,j que se pueden definir con i<j)

Si usamos un algoritmo que sólo realiza intercambios entre componentes vecinas, a lo sumo podrá deshacer una inversión con cada intercambio (obsérvese que si se hacen intercambios lejanos se pueden deshacer más inversiones de una vez). Eso quiere decir que para deshacer todas las inversiones en el caso peor tendrá que hacer un número de intercambios del orden de n^2, por lo que

---

$T_A(n) \in \Omega(n^2)$

Algoritmos de ordenación

Según los resultados obtenidos en el apartado anterior y considerando su complejidad, tenemos dos grandes grupos de algoritmos de ordenación

---

Algoritmos de ordenación de complejidad $O(n^2)$

— Método de inserción. Emplea intercambios entre vecinos y se basa en tener una parte del vector ordenando e ir insertando un elemento cada vez en la parte ordenada.

— Método de selección. Emplea intercambios entre elementos lejanos y se basa en ir eligiendo cada vez el menor de los elementos que quedan en la parte no ordenada.

— Método de la burbuja. Consiste en considerar los (N–1)*(N–I) intercambios posibles entre vecinos para lograr ordenar el vector.

La implementación "natural" del método de inserción consigue complejidad lineal en el caso mejor, cuando el vector está ordenado. Una implementación "ingeniosa" del método de la burbuja consigue complejidad lineal en el caso mejor, cuando el vector está ordenado, y se detiene cuando el resto del vector ya está ordenado.

Algoritmos de ordenación de complejidad $O(n \log n)$

— Ordenación rápida (*quicksort*). Método debido a C. A. R. Hoare. Se aprovecha del acceso directo de los vectores. La idea consiste en determinar la componente donde corresponde almacenar un cierto valor del vector y situar en el subvector inferior los valores menores o iguales y el subvector superior los valores mayores. El problema se reduce entonces a ordenar los subvectores así obtenidos. Su complejidad es cuadrática en el caso peor, pero es $O(n \log n)$ en el caso promedio. Requiere un espacio auxiliar de coste $O(\log n)$.

— Ordenación por mezcla (*mergesort*). Se divide el vector en dos partes que, una vez ordenadas, se mezclan ordenadamente para obtener el resultado final. Su complejidad es cuasilineal, $O(n \log n)$, en el caso peor. Requiere un espacio auxiliar de coste $O(n)$.

— Ordenación por montículos (*heapsort*). Debido a J. W. J. Williams. Se trata de organizar los valores del vector como un montículo, para luego ir extrayendo uno a uno los valores del montículo e insertándolos al final de la parte ordenada del vector. Su complejidad es *O*(n log n) en el caso peor y no requiere espacio auxiliar si se simula el montículo sobre el propio vector.

La ordenación rápida y la ordenación por mezcla se explican en el tema de algoritmos recursivos. La ordenación por montículos se explica en el tema de árboles.

Ordenación por inserción

La idea del algoritmo es que dividimos el vector en una parte ordenada y otra desordena, y en cada pasada insertamos el primer elemento de la parte desordenada en el lugar que le corresponde dentro de la parte ordenada, realizando intercambios entre vecinos.

La forma en que escribamos la especificación del algoritmo nos conducirá al algoritmo a implementar pues dirigirá la obtención del invariante.

En la especificación sólo nos preocupamos por la condición ord(v) pues sabemos que perm(v,V) está garantizada al tratarse de un algoritmo que sólo realiza intercambios.

La postcondición:

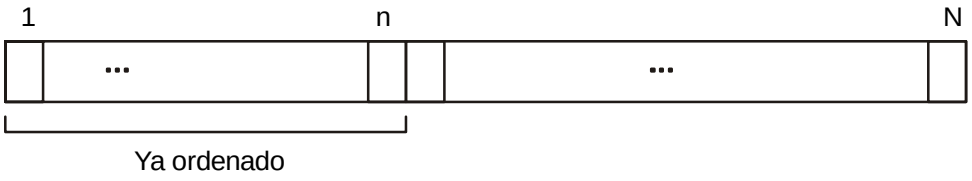
$$Q \equiv ord(v) \equiv \forall\ i,j : 1 \le i < j \le N : v(i) \le v(j)$$

La derivación formal:

— **B.1**. Obtención del invariante

El invariante se obtiene sustituyendo la constante *N* de la postcondición por una variable, y añadiendo una condición sobre los posibles valores de *n*

$$I : \forall\ i,j : 1 \le i < j \le n : v(i) \le v(j) \wedge 1 \le n \le N$$



— **B.2**. Condición de terminación

Lo que le falta al invariante para garantizar la postcondición es

$$\neg B : n = N$$

Se prueba

$I \wedge \neg B \Rightarrow Q$

— **B.3** ¿Está definida la condición de repetición?

La condición de repetición

$B : n \neq N$

No está definida porque no está declarada *n*.
Se añade *n* a las declaraciones locales.

$I \Rightarrow def(B)$

— **B.4**. Expresión de acotación

El valor de *n* se va a ir acercando paulatinamente a *N*

$C : N - n$

Se prueba

$I \wedge B \Rightarrow def(C) \wedge dec(C)$

— **B.5**. Acción de inicialización

Una forma de hacer que el invariante sea cierto de modo trivial es inicializar *n* como 1

$A_0 : n := 1$

— **B.6**. Acción de avance

$A_2 : n := n + 1$

Probamos que efectivamente

$\{ I \wedge B \wedge N-n=T \}\ n := n+1\ \{ N-n < T \}$

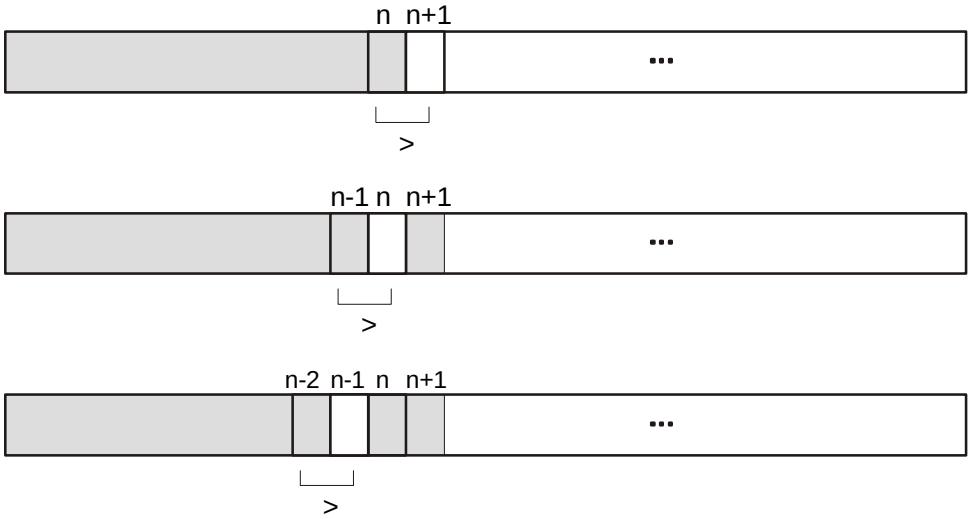
— **B.7.** ¿Es suficiente con la acción de avance?

Para ello tendría que cumplirse $I \Rightarrow I[n/n+1]$ lo cual no es cierto porque estar ordenado hasta *n* no implica estar ordenado hasta n+1

— **B.8**. Restablecimiento del invariante

$\{ I \wedge B \}$
A_1
$\{ I[n/n+1] \}$

La idea es que para restablecer el invariante tenemos que implementar un bucle que haga lo siguiente:



Tendríamos que aplicar de nuevo el proceso de derivación de bucles, pero vamos a abreviarlo.

La derivación de esta acción es el **ejercicio 75**. Allí se toma como invariante una versión reelaborada del invariante que presentamos aquí; una versión con la que es más sencillo realizar la verificación.

La postcondición es

$$\forall\ i,j : 1 \leq i < j \leq n+1 : v(i) \leq v(j) \wedge 1 \leq n+1 \leq N$$

La idea para obtener el invariante es considerar que en el subvector [1..n+1] todos los elementos están ordenados menos 1; y que ese uno es menor que todos los que están a su derecha. De forma que el subvector entero estará ordenado cuando el elemento problemático sea mayor o igual que el de su izquierda o el elemento problemático sea el primero

$$\forall\ i,j : 1 \leq i < j \leq n+1 \wedge j \neq m+1 : v(i) \leq v(j)$$
$$\wedge\ (v(m) \leq v(m+1) \vee m = 0) \wedge 0 \leq m \leq n < N$$

De esta forma tenemos el invariante y la condición de terminación

$$I_{int} : \forall\ i,j : 1 \leq i < j \leq n+1 \wedge j \neq m+1 : v(i) \leq v(j) \wedge 0 \leq m \leq n < N$$

La condición de terminación.

$$\neg B_{int} : m = 0 \vee v(m) \leq v(m+1)$$

Por lo tanto la condición de repetición

$$B_{int} : m \neq 0 \wedge v(m) > v(m+1)$$

El problema es que no se cumple

$$I_{int} \Rightarrow def(B_{int})$$

Aparte de que es necesario declarar *m*, tenemos que no es cierto

$$0 \leq m \Rightarrow enRango(v,m)$$

Es decir, cuando el elemento a insertar es menor que todos los de la parte ordenada, tenemos que llegar hasta m=0 en la última iteración, donde ya no entramos en el cuerpo del bucle. Pero aunque en esa situación ya no sería necesario evaluar v(m)>v(m+1), lo estamos haciendo y v(0)>v(1) no está definido porque no existe v(0). Una solución sería definir una variable booleana auxiliar de forma que no apareciese el acceso al vector dentro de la condición de repetición. La otra solución es tratar la primera componente del vector como un caso aparte fuera del bucle, a la terminación de éste.

El invariante hay que modificarlo, así como las condiciones de repetición y terminación

$$I_{int} : \forall\ i,j : 1 \leq i < j \leq n+1 \wedge j \neq m+1 : v(i) \leq v(j) \wedge 1 \leq m \leq n < N$$

$$\neg B_{int} : m = 1 \vee v(m) \leq v(m+1)$$

$$B_{int} : m \neq 1 \wedge v(m) > v(m+1)$$

Así, a la terminación del bucle no sabemos si v(1) es o no menor que v(2), por lo que escribimos un condicional para averiguarlo y actuar en consecuencia.

La expresión de acotación

$$C_{int} : m$$

La acción de inicialización

```
m := n
```

Con lo que tenemos que el subvector v[1..n] está ordenado porque lo garantiza la precondición *I*, y que el elemento problemático es v(m+1).

La acción de avance será

```
m := m-1
```

Habrá que restablecer el invariante, intercambiando los valores de v(m) y v(m+1) (Esta verificación es laboriosa porque está involucrado un intercambio entre posiciones de un vector y un invariante con un cuantificador existencial con una excepción).

```
Int(v, m, m+1)
```

Por último la instrucción condicional que va a continuación del bucle se encarga de intercambiar v(1) y v(2) en caso de que sea necesario.

```
si v(1) > v(2)
  entonces Int(v, 1, 2)
  sino seguir
fsi
```

Una forma de soslayar el inconveniente de escribir este condicional adicional es aprovecharse de la característica que tienen algunos lenguajes al realizar *evaluación perezosa* de las expresiones, evaluando sólo la parte de la expresión necesaria para llegar a una solución. Aprovechándonos de ello, podríamos utilizar el invariante y la condición de terminación que elegimos inicialmente. Si n embargo, a la hora de diseñar el algoritmo nos debemos abstraer de estas peculiaridades, dejando la posible modificación como una optimización posterior.

— **B.9** ¿Sigue terminando el bucle?

Informalmente podemos ver que sí pues la acción de restablecimiento del invariante no modifica el valor de *n*.

Con todo esto el algoritmo de ordenación queda de la siguiente forma:

```
proc ordenaVectorInserción( es v : Vector[1..N] de elem );
{ P₀ : v = V ∧ N ≥ 1 }
var
  n,m : Ent;
inicio
  n := 1;
{ I; C }
  it n ≠ N
  →  m := n;
    { Iint; Cint }
      it m ≠ 1 ∧ v(m) > v(m+1)
      →  Int(v, m, m+1);
          m := m – 1
      fit;
      si v(1) > v(2)
        entonces Int(v,1,2)
        sino seguir
      fsi;
      n := n + 1
  fit
{ Q₀ : ord(v) ∧ perm(v,V) }
```

```
fproc
```

En cuanto a la complejidad, el caso peor es cuando el vector está ordenado en orden decreciente, en cuyo caso tenemos $O(\sum i : 1 \leq i \leq N-1 : i) = O(N^2)$. En el caso mejor, cuando el vector está ordenado crecientemente, la complejidad es $O(N)$.

En [Kal90], pp. 172-174 se puede encontrar una derivación más formal de este algoritmo.

Ordenación por selección

Formalmente la obtención de este algoritmo se basa en considerar una formulación diferente de la postcondición:

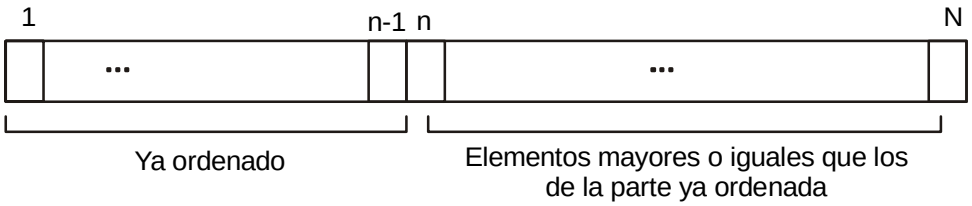
$$\forall\ i : 1 \leq i < N : (\forall\ j : i < j \leq N : v(i) \leq v(j))$$

La idea consiste en sustituir la primera aparición de N por una variable, de forma que habrá una parte del vector, que está ordenada, donde se encuentran los elementos menores que el resto. Puede comprobarse que si se sustituyesen las dos apariciones de N por una variable, obtendríamos de nuevo el método de inserción. Para hacer avanzar el bucle, seleccionaremos el menor de los elementos que quedan y lo colocaremos al final de la parte ordenada.

— **B.1** Invariante

Lo obtenemos sustituyendo la primera aparición de N en la postcondición, e introduciendo la condición sobre el posible rango de valores:

$$\forall\ i : 1 \leq i < n : (\forall\ j : i < j \leq N : v(i) \leq v(j)) \wedge 1 \leq n \leq N$$



— **B.2**. Condición de terminación

$$\neg B : n = N$$

y la condición de repetición

$$B : n \neq N$$

$$I \wedge \neg B \Rightarrow Q$$

— **B.3**. ¿Está definida la condición de repetición?

Hay que añadir *n* a las declaraciones locales.

$I \Rightarrow def(B)$

— **B.4**. Expresión de acotación.

$C : N - n$

$I \wedge B \Rightarrow def(C) \wedge dec(C)$

— **B.5**. Acción de inicialización

$A_0 : n := 1$

con lo que el invariante se hace trivialmente cierto porque el cuantificador opera sobre un dominio vacío.

$\{ P \}\ A_0\ \{ I \}$

— **B.6**. Acción de avance

vamos incrementando *n*

$A_2 : n := n + 1$

— **B.7**. Evidentemente no es suficiente con la acción de avance

— **B.8**. Restablecimiento del invariante

la postcondición a la que hay que llegar (descomponiéndola y generalizándola para ver de dónde sale el invariante)

$$I[n/n+1] \Leftrightarrow \forall\ i : 1 \le i < n : (\forall\ j : i < j \le N : v(i) \le v(j)) \wedge \forall\ j : n < j \le N : v(n) \le v(j)$$

$$\Leftrightarrow \forall\ i : 1 \le i < n : (\forall\ j : i < j \le N : v(i) \le v(j)) \wedge v(menor) = \min k : n \le k \le N : v(k) \wedge menor = n$$

La acción de restablecimiento del invariante debe recorrer el vector entre n y N para encontrar el menor de los elementos, intercambiándolo entonces con el que ocupa la posición n. Esto lo conseguimos como una composición secuencial, de un bucle que localiza el índice del menor elemento del subvector, junto con un intercambio de ese elemento con el que ocupa la posición n, que consigue que se cumpla menor = n.

El invariante del bucle se obtiene sustituyendo la N que aparece en la expresión del mínimo *m*, junto con sus condiciones de rango.

$$I_{int} : \forall\ i : 1 \le i < n : (\forall\ j : i < j \le N : v(i) \le v(j)) \wedge v(menor) = \min k : n \le k \le m : v(k) \wedge n \le m \le N \wedge n \le menor \le N$$

La primera parte del invariante hay que incluirla para luego poder obtener la postcondición I[n/n+1].

$B_{int} : m \neq N$

$C_{int} : N - m$

La derivación de esta acción es el ejercicio 77.

— **B.9**. ¿Sigue terminando?

Informalmente, el restablecimiento del invariante no afecta a *n*.

Con todo esto el procedimiento queda:

```
proc ordenaVectorSelección ( es v : Vector[1..N] de elem );
{ P₀ : v = V ∧ N ≥ 1 }
var
  n, m, menor : Ent;
inicio
  n := 1;
{ I; C }
  it n ≠ N
  →  <m, menor> := <n, n>;
    { Iint; Cint }
      it m ≠ N
      →  si v(m+1) < v(menor)
            entonces menor := m+1
            sino seguir
         fsi;
         m := m+1
      fit;
      Int(v, n, menor);
      n := n + 1
  fit
{ Q₀ : ord(v) ∧ perm(v,V) }
fproc
```

En el caso peor (y en el mejor, pues los bucles siempre se ejecutan hasta el final) este algoritmo es de orden $O(n^2)$. Aunque si contabilizásemos sólo los intercambios tenemos una complejidad de $O(n)$.

Método de la burbuja

La postcondición se escribe de la misma forma que en el método de selección.

```
∀ i : 1 ≤ i < N : ( ∀ j : i < j ≤ N : v(i) ≤ v(j) )
```

De esta forma, todo el proceso de derivación es idéntico hasta que se llega a la acción de restablecimiento del invariante. El objetivo es el mismo, conseguir que el menor de los que quedan pase a ocupar la posición *n*; pero en lugar de hacer una búsqueda y un intercambio, se van haciendo sucesivos intercambios empezando por el final.

```
I[n/n+1] ⇔ ∀ i : 1 ≤ i < n : ( ∀ j : i < j ≤ N : v(i) ≤ v(j) ) ∧
```

$$\forall\ j : n < j \leq N : v(n) \leq v(j)$$

Para obtener el invariante sustituimos *n* por una variable *m*, con lo que nos queda:

$I_{int} : \forall\ i : 1 \leq i < n : (\forall\ j : i < j \leq N : v(i) \leq v(j)) \wedge$
$\forall\ j : m < j \leq N : v(m) \leq v(j) \wedge n \leq m \leq N$
$\neg B_{int} : m = n$
$C_{int} : m$

Con todo esto el procedimiento queda:

```
proc ordenaVectorBurbuja ( es v : Vector[1..N] de elem );
{ P₀ : v = V ∧ N ≥ 1 }
var
  n, m : Ent;
inicio
  n := 1;
{ I; C }
  it n ≠ N
  →  m := N;
    { I_int; C_int }
      it m ≠ n
      →  si v(m-1) < v(m)
            entonces Int(v, m-1, m)
            sino seguir
         fsi;
         m := m-1
      fit;
      n := n + 1
  fit
{ Q₀ : ord(v) ∧ perm(v,V) }
fproc
```

La complejidad es, en cualquier caso, de orden $O(n^2)$ y, a diferencia del método de selección, el número de intercambios es también $O(n^2)$. Sin embargo, se puede hacer una optimización si observamos que cuando en un recorrido entero del bucle interno no se efectúa ningún intercambio, eso quiere decir que el subvector v[n..N] ya está ordenado, con lo cual podemos terminar. Para hacer esta optimización es necesario introducir un invariante auxiliar tanto al bucle externo como al bucle interno

$I_1 : b \rightarrow \forall\ i,j: n \leq i < j \leq N : v(i) \leq v(j)$
$I_{1\ int} : b \leftrightarrow \forall\ i,j: m \leq i < j \leq N : v(i) \leq v(j)$

No ponemos doble implicación en el invariante externo porque al principio de cada pasada por el bucle externo no sabemos si el resto del vector está ordenado o no, con lo que *b* es falso inicialmente. En el invariante interno sí podemos poner doble implicación, porque inicialmente

consideramos un subvector vacío, para el cual es cierto que está ordenado, y luego vamos actualizando *b* adecuadamente.

Cambiamos además la condición de repetición del bucle externo, incluyendo la comprobación sobre el valor de *b*.

Con todo esto el procedimiento queda:

```
proc ordenaVectorBurbuja ( es v : Vector[1..N] de elem );
{ P₀ : v = V ∧ N ≥ 1 }
var
  n, m : Ent;
  b : Bool;
inicio
  <n, b> := <1, falso>;
{ I ∧ I₁; C }
  it n ≠ N AND NOT b
  →  <m, b> := <N, cierto>;
    { I_int ∧ I_1 int; C_int }
      it m ≠ n
      →  si v(m-1) < v(m)
            entonces Int(v, m-1, m) ; b := falso
            sino seguir
         fsi;
         m := m-1
      fit;
      n := n + 1
  fit
{ Q₀ : ord(v) ∧ perm(v,V) }
fproc
```

De esta forma, la complejidad en el caso peor sigue siendo $O(n^2)$, pero ahora si el vector está ordenado la complejidad pasa a ser $O(n)$.

1.7 Ejercicios

Asertos en lógica de predicados

1. Supongamos las declaraciones

```
cte
  N = ¿;    % entero ≥ 1
var
  x, y, p: Ent;
  v : Vector [1..N] de Ent;
```

Escribe asertos que formalicen los tres enunciados siguientes. Señala las variables libres y las variables ligadas.

(a) `x` aparece como componente de `v`.
(b) `x` es mayor que todas las componentes de `v`.
(c) `x` aparece una sola vez como componente de `v`.
(d) Todas las componentes positivas de `v` están comprendidas entre `x` e `y`.
(e) `v` está ordenado en orden estrictamente creciente.
(f) `v` está ordenado en orden no decreciente.
(g) `v` está ordenado en orden estrictamente decreciente.
(h) `y` es la suma de las componentes positivas de `v`.
(i) `y` es el máximo de las componentes negativas de `v`.
(j) `p` es el menor índice de `v` que contiene el valor `x`.

2. Realiza correctamente cada uno de las sustituciones que siguen, e indica posibles formas erróneas de realizarlas. Para `v` se supone el tipo del ejercicio 1, y las restantes variables se suponen enteras.

(a) $(\exists i: 1 \le i \le n: v(i)=x\ )\ [n/n+1]$
(b) $(\forall i: n \le i \le N: v(i)>0\ )\ [n/n-1]$
(c) $(\exists i: 1 \le i \le N: v(i)=x\ )\ [x/2*i]$
(d) $(\forall i: 1 \le i \le N: v(i)>x\ )\ [i/i*i]$
(e) $(\prod i: 1 \le i \le N: x+i\ )\ [x/x-i]$
(f) $(\sum i: 1 \le i \le N: x*i\ )\ [x/i+2]$
(g) $(\prod i: n \le i \le N: x+i\ )\ [n/n-1]$
(h) $(\sum i: 1 \le i \le n: x*i\ )\ [n/n+1]$

3. Comprueba que las dos sustituciones siguientes no producen el mismo resultado. Las variables se suponen de tipo entero.

(a) `(x-17) * y [x / 2*y, y / z]`
(b) `(x-17) * y [x / 2*y] [y / z]`

4. Las variables que intervienen en lo que sigue se suponen de tipo entero. Realiza las sustituciones indicadas y simplifica el resultado.

(a) $(x^2 + 2*x + 1)\ [x / x+a]$
(b) $(x^2 \ge y)\ [x / y+1, y / x-1]$
(c) $(x \ge y+1 \wedge y \ge z)\ [x / x+3*z, y / x-y+1]$
(d) $(\exists k: 1 \le k \le n : x^2 + y^2 = k^2)\ [x / x+1, y / y-1, n / x+y]$

5. Las variables que intervienen en este ejercicio son todas de tipo entero. En cada apartado, estudia si alguno de los dos asertos es más fuerte que el otro. Razona tus respuestas.

(a) $x \geq 0$	$x \geq 0 \wedge y \geq 0$
(b) $x \geq 0 \vee y \geq 0$	$x + y = 0$
(c) $x < 0$	$x^2 + y^2 = 9$
(d) $x \geq 1 \rightarrow x \geq 0$	$x \geq 1$
(e) $\exists i: Ent : x = 2*i$	$\exists i : Ent : x = 6*i$
(f) $\exists i: Ent : x = 4*i$	$\exists i : Ent : x = 6*i$

6. En este ejercicio, P y Q representan asertos dados, y X representa un aserto incógnita. En cada caso, encuentra para X la solución más fuerte y la solución más débil que cumplan lo requerido.

(a) $X \Rightarrow P \vee Q$ (b) $X \vee Q \Rightarrow P \vee Q$

(c) $X \Rightarrow P \wedge Q$ †(d) $X \wedge Q \Rightarrow P \wedge Q$

7. Simplifica, si es posible, los asertos siguientes. Los asertos simplificados deben ser equivalentes a los dados. Las variables se suponen de tipo entero.

(a) $(x < 1) \wedge (x > -1)$ (b) $\forall i : i \geq a : x \leq i$

(c) $\exists i : i \geq 0 : (\exists j : 0 \leq j < i : x = 2*j)$

8. Simplifica los asertos de definición que siguen, suponiendo que el vector v tiene el tipo indicado en el ejercicio 1, y que las restantes variables son enteras.

(a) **def**(x div (a – b)) (b) **def**(a mod b)

(c) **def**(a + b) (d) **def**(x div y + y div x)

(e) **def**(v(i) mod b) (f) **def**($\sum i : a \leq i < b : v(i)$)

9. Aplica las leyes de descomposición de cuantificadores a los casos que siguen:

(a) $\forall i : 1 \leq i \leq n+1 : x < v(i)$ (b) $\exists i : 1 \leq i \leq n+1 : v(i) = i$

(c) $x = \sum i : n-1 \leq i < N : i*v(i)$ (d) $x = \prod i : n-1 \leq i \leq N : i+v(i)$

(e) $x = \#i : 1 \leq i \leq n+1 : v(i)=0$ (f) $x = Max\ i : 1 \leq i \leq n+1 : v(i)$

Especificaciones Pre/Post

10. Formaliza especificaciones Pre/Post para algoritmos que realicen las tareas siguientes. Distingue en cada caso entre las variables de programa, las variables auxiliares y las variables ligadas.

(a) Copiar el valor de una variable en otra.
(b) Intercambiar los valores de dos variables.
(c) Calcular el valor absoluto de un entero.
(d) Dada una cantidad entera no negativa de segundos, convertirla a horas, minutos y segundos.
(e) Calcular la raíz cuadrada por defecto de un entero.
(f) Calcular la raíz cuadrada exacta de un entero.
(g) Calcular el cociente y el resto de una división entera.
(h) Calcular el máximo común divisor de dos enteros.
(i) Intercambiar dos componentes de un vector.
(j) Calcular el número de ceros que aparecen en un vector de enteros.
(k) Calcular el producto escalar de dos vectores de reales.
(l) Dados dos vectores de enteros, reconocer si uno de ellos es una permutación del otro.

(m) Dados dos vectores de enteros, reconocer si uno de ellos es la imagen especular del otro.

(n) Ordenar un vector de enteros dado.

†(o) Calcular el máximo de las sumas de segmentos de un vector de enteros dado (entendiendo que los *segmentos* de un vector son los diferentes subintervalos del intervalo de índices del vector).

Reglas de verificación básicas

11. Explica operacionalmente el significado de las dos reglas que siguen, y demuestra que se deducen de las reglas de verificación básicas.

$$\frac{\{ P_1 \}\ A\ \{Q_1\} \quad \{ P_2 \}\ A\ \{ Q_2 \}}{\{ P_1 \wedge P_2 \}\ A\ \{ Q_1 \wedge Q_2 \}} \qquad \frac{\{ P_1 \}\ A\ \{Q_1\} \quad \{ P_2 \}\ A\ \{ Q_2 \}}{\{ P_1 \vee P_2 \}\ A\ \{ Q_1 \vee Q_2 \}}$$

12. Para razonar con el operador *pmd* son válidas las reglas que siguen. Explica su significado operacional.

(a) *pmd* construye la precondición más débil:

$$\frac{P \Rightarrow pmd(A, Q)}{\{ P \}\ A\ \{ Q \}}\ \text{(syss)}$$

(b) Exclusión de milagros:

$$pmd(A, \text{Falso}) \Leftrightarrow \text{Falso}$$

(c) Monotonía:

$$\frac{Q_1 \Rightarrow Q_2}{pmd(A, Q_1) \Rightarrow pmd(A, Q_2)}$$

(d) Distributividad con respecto a $\wedge$:

$$pmd(A, Q_1 \wedge Q_2) \Leftrightarrow pmd(A, Q_1) \wedge pmd(A, Q_2)$$

(e) Distributividad con respecto a $\vee$:

$$pmd(A, Q_1 \vee Q_2) \Leftarrow pmd(A, Q_1) \vee pmd(A, Q_2)$$

†13. Postulemos la existencia de una acción llamada "azar" que cumpliese la especificación siguiente:

```
var x : Ent; { cierto } azar { x = 0 ∨ x = 1 }
```

Explica el significado operacional de la especificación. ¿Puede deducirse de ella que "azar" cumpla alguna de las dos especificaciones siguientes?

```
(a) var x : Ent; { cierto } azar { x = 0 }
(b) var x : Ent; { cierto } azar { x = 1 }
```

†14. La acción "azar" del ejercicio anterior es *indeterminista* en el sentido de que un mismo estado inicial puede ser transformado en varios estados finales (el cómputo escogerá al azar uno de ellos, sin que el usuario sepa cuál). Por el contrario, una acción se llama *determinista* si el estado inicial determina unívocamente el estado final. Volviendo a pensar en el apartado (e) del ejercicio 12, demuestra que si la acción A es *determinista* entonces puede aceptarse el axioma siguiente, que falla para $A \equiv azar$.

$$pmd(\mathrm{A}, \mathrm{Q}_1 \vee \mathrm{Q}_2) \Leftrightarrow pmd(\mathrm{A}, \mathrm{Q}_1) \vee pmd(\mathrm{A}, \mathrm{Q}_2)$$

En general, este axioma no puede aceptarse como válido si la acción A es indeterminista.

La acción nula *seguir*

15. Estudia los enunciados de corrección que siguen. Verifica los que sean válidos y construye contraejemplos para los que no lo sean.

```
(a) var x, y : Ent; { x > 0 ∧ y > 0 } seguir { x+y > 0 }
(b) var x, y : Ent; { x+y > 0 } seguir { x > 0 }
(c) var x, y : Ent; { x > 0 ∨ y > 0 } seguir { x*y > 0 }
(d) var x, y : Ent; { x*y < 0 } seguir { ( x > 0 ) ↔ ( y < 0 ) }
```

Asignación

16. Verifica usando la regla de la asignación:

```
var x, y : Ent
{ x ≥ 2 ∧ 2*y > 5 } x := 2*x + y−1 { x−3 > 2 }
```

17. Verifica (a), (b) y encuentra contraejemplos para (c), (d).

```
(a) var x, y, m : Ent;
    { x ≥ 0 ∧ y ≥ 0 } m := x+y { m ≥ max(x,y) }
(b) var x, y, m : Ent;
    { x < 0 ∧ y < 0 } m := max(x,y) { m > x+y }
(c) var x, y, m : Ent;
    { x ≥ 0 ∧ y ≥ 0 } m := max(x,y) { m > x+y }
(d) var x, y, m : Ent;
    { x < 0 ∧ y < 0 } m := x+y { m ≥ max(x,y) }
```

18. Verifica usando la regla de la asignación:

```
var x, y : Ent;
{ def(e) ∧ y = 0 } x := e { y = 0 }
```

Razona: esto no sería correcto si la evaluación de *e* pudiese afectar a la variable *y*. Concluye: la regla de la asignación sólo es válida en general para expresiones cuya evaluación no cause *efectos colaterales*.

19. Verifica usando la regla de la asignación múltiple:

```
var x, X, y, Y, p : Ent;
{ X*Y = p + x*y ∧ y > 0 ∧ par(y) }
    <y, x> := <y div 2, x+x>
{ X*Y = p + x*y ∧ y ≥ 0 }
```

20. Calcula en cada apartado la *precondición más débil P* que satisface la especificación dada, suponiendo la declaración de variables indicada.

```
var x, y : Ent; n : Nat; b : Bool;
```

(a) { *P* } x := x + 2 { x ≥ 0 }
(b) { *P* } x := 2*x { x ≤ 99 }
(c) { *P* } x := x*x − 2 { x^2 − x + 1 > 0 }
(d) { *P* } x := x − 2 { x = x − 2 }
(e) { *P* } x := x **mod** 2 { x = x **mod** 2 }
(f) { *P* } x := 3*x { ∀i : 1 ≤ i ≤ n : x ≠ 6*i }
(g) { *P* } b := b **AND** (x > 0) { ¬b }
(h) { *P* } (x, y) := (y, x) { x = X ∧ y = Y }
(i) { *P* } (x, y) := (y, x) { x < 2*y }
(j) { *P* } (x, y) := (y−2, x+3) { x+y > 0 }

Asignación a vectores

21. La regla de la asignación no se puede aplicar a asignaciones de la forma

```
v(i) := e
```

tratando a la expresión v(i) como si fuese una variable. Comprueba que si admitiésemos aplicar la regla de la asignación de este modo, tendríamos:

(a) Enunciados verdaderos, pero imposibles de verificar, como:

```
var v : Vector [1..100] de Ent; i, j : Ent;
{ i = j } v(i) := 0 { v(j) = 0 }
```

(b) Enunciados falsos, pero que se pueden verificar (incorrectamente), como:

```
var v : Vector [1..100] de Ent; i, j : Ent;
{ v(1) = 2 ∧ v(2) = 2 } v(v(2)) := 1 { v(v(2)) = 1 }
```

22. Aplicando la regla correcta de asignación a vectores,

(a) calcula la precondición más débil *P* que cumple:

```
var v : Vector [1..100] de Ent;
{ P } v(v(2)) := 1 { v(v(2)) = 1 }
```

(b) deduce del apartado (a) que:

```
var v : Vector [1..100] de Ent;
{ v(1) = 1 ∧ v(2) = 2 } v(v(2)) := 1 { v(v(2)) = 1 }
```

Composición secuencial

23. Usa las reglas de la composición secuencial y de la asignación para encontrar *asertos intermedios* adecuados y completar la verificación siguiente:

```
var x, y : Ent;
{ P: x = X ∧ y = Y }
    x := x−y;
{ R1:              }
    y := x+y;
```

```
{ R₂:                }
    x := y-x
{ Q: x = Y ∧ y = X }
```

24. Supuesto que *x*, *y* sean variables de tipo entero, calcula en cada caso la *precondición más débil* que satisface la especificación.

```
(a) { P } x := x*x ; x := x + 1 { x > 0 }
(b) { P } x := x + y ; y := 2*y - x { y > 0 }
(c) { P } y := 4*x ; x := x*x - y ; x := x+4 { x > 0 }
(d) { P } x := y ; y := x { x = A ∧ y = B }
```

25. Lo que sigue demuestra que la *composición secuencial* no es *conmutativa*. Verifica:

```
(a) var x, y : Ent;
    { x = 3 ∧ y = Y } x := 0 ; y := x + y { x = 0 ∧ y = Y }
(b) var x, y : Ent;
    { x = 3 ∧ y = Y } y := x + y ; x := 0 { x = 0 ∧ y = Y+3 }
```

†26. Demuestra que la *composición secuencial* es *asociativa*, razonando que dos enunciados de corrección de la forma

{ P } A ; (B ; C) { Q } y { P } (A ; B) ; C { Q }

son siempre equivalentes. Por ello, se escribe simplemente:

{ P } A ; B ; C { Q }

Una propiedad similar vale para la composición secuencial de cualquier número de acciones.

Sugerencia: Demuestra que *pmd*(A ; (B ; C) , Q) ⇔ *pmd*((A ; B) ; C , Q)

†27. Demuestra que los dos enunciados de corrección que siguen son equivalentes; esto es, para *P* y *Q* cualesquiera, (a) se cumple si y sólo si se cumple (b).

```
(a) var x, y : Ent;                    (b) var x, y : Ent;
    { P } x := y ; y := x { Q }            { P } x := y { Q }
```

28. El siguiente algoritmo anotado resuelve el problema de convertir una cantidad entera no negativa de segundos a horas, minutos y segundos. Completa la verificación.

```
var s, m, h : Ent;
{ s = S ∧ s ≥ 0 }
    <h, s> := <s div 3600, s mod 3600>;
{ s + 3600*h = S ∧ 0 ≤ s < 3600 }
    <m, s> := <s div 60, s mod 60>
{ s + 60*m + 3600*h = S ∧ 0 ≤ s < 60 ∧ 0 ≤ m < 60 }
```

Variables locales

29. Verifica usando la regla de las variables locales:

```
var x, y : Ent;
{ x = X ∧ y = Y }
  var z : Ent;
```

```
  inicio
    z := x; x := y; y := z
  fvar
{ x = Y ∧ y = X }
```

Composición alternativa (distinción de casos)

30. Los dos algoritmos siguientes calculan el máximo de dos números enteros. Verifica y compara.

(a)
```
var x, y, z : Ent;
{ x = X ∧ y = Y }
  si
      x ≤ y → z := y
   □  y ≤ x → z := x
  fsi
{ x = X ∧ y = Y ∧ z = max(x,y) }
```

(b)
```
var x, y, z : Ent;
{ x = X ∧ y = Y }
  si x ≤ y
    entonces z := y
    sino     z := x
  fsi
{x = X ∧ y = Y ∧ z = max(x,y)}
```

31. Especifica, diseña y verifica un algoritmo que calcule el máximo de tres números enteros dados.

32. En cada uno de los casos que siguen, haz la verificación formal.

(a)
```
var x : Ent;
{ cierto }
  si
      cierto → x := 0
   □  cierto → x := 1
  fsi
{ x = 0 ∨ x = 1 }
```

(b)
```
var x : Ent;
{ cierto }
  si
      x ≥ 0 → x := x+1
   □  x ≤ 0 → x := x−1
  fsi
{ x ≠ 0 }
```

(c)
```
var x, y : Ent;
{ cierto }
  (x, y) := (y*y, x*x);
  si
      x ≥ y → x := x−y
   □  y ≥ x → y := y−x
  fsi
{ x ≥ 0 ∧ y ≥ 0 }
```

(d)
```
var x, y, z, p, q, r : Ent;
{ x = X ∧ y = Y ∧ z = Z }
  <p, q, r> := <x, y, z>;
  si
      p > q → <p, q> := <q, p>
   □  p ≤ q → seguir
  fsi;
  si
      p > r → <p, r> := <r, p>
   □  p ≤ r → seguir
  fsi;
  si
      q > r → <q, r> := <r, q>
   □  q ≤ r → seguir
  fsi
{Perm((p,q,r), (X,Y,Z)) ∧ p ≤ q ≤ r }
```

33. Calcula la precondición más débil *P* que satisface:

```
var x : Ent;
{ P }
  x := x+1;
  si
      x > 0 → x := x-2
   □  x = 0 → seguir
   □  x < 0 → x := x+3
{ x ≥ 1 }
```

34. Dado:

```
var x : Ent;
{ x = X }
  si
      x =  1 → x := -1
   □  x = -1 → x :=  1
  fsi
{ x = -X }
```

demuestra con un contraejemplo que la especificación no se cumple, y a continuación fortalece la precondición de manera que la especificación se cumpla. Haz la verificación.

†35. Usa cálculo de *pmds* para demostrar que las dos construcciones siguientes son equivalentes con respecto a cualquier especificación Pre/Post:

(a) **si** $B_1 \rightarrow A_1 \square B_2 \rightarrow A_2$ **fsi** ; A
(b) **si** $B_1 \rightarrow A_1$; A $\square$ $B_2 \rightarrow A_2$; A **fsi**

Composición iterativa (iteración)

36. Completa la verificación de los dos algoritmos de multiplicación que siguen, ya anotados con invariante y expresiones de acotación. Compara.

(a)
```
var x, y, p : Ent;
{Pre. P: x = X ∧ y = Y ∧ y ≥ 0}
  p := 0
{Inv. I: X*Y = p + x*y ∧ y ≥ 0
 Cota C: y                    }
  it y ≠ 0
  → { X*Y = p + x*y ∧ y > 0 }
     <p, y> := <p+x, y-1> { I }
  fit
{Post. Q: p = X*Y}
div 2>
```

(b)
```
var x, y, p : Ent;
{Pre. P: x = X ∧ y = Y ∧ y ≥ 0}
  p := 0;
{Inv. I: X*Y = p + x*y ∧ y ≥ 0
  Cota C: y                   }
  it y ≠ 0
  → { X*Y = p + x*y ∧ y > 0 }
    si par(y)
      entonces
                 <x, y> := <x+x, y

      sino <p, y> := <p+x, y-1>
    fsi { I }
  fit
{Post. Q: p = X*Y}
```

37. Considera la especificación:

```
var x, y, p : Ent;
{ Pre. P: x = X ∧ y = Y ∧ y ≥ 0 }
    potencia
{ Post. Q: p = X^Y }
```

Teniendo en cuenta que "la exponenciación es al producto como el producto es a la suma", construye y verifica dos algoritmos similares a los del ejercicio anterior, que cumplan esta especificación.

†38. Construye y verifica un algoritmo de división entera que cumpla la especificación siguiente, y que utilice solamente operaciones aritméticas de suma y resta.

```
var x, y, c, r : Ent;
{ Pre. P: x = X ∧ y = Y ∧ x ≥ 0 ∧ y > 0 }
    divEnt
{ Post. Q: x = X ∧ y = Y ∧ x = c*y + r ∧ 0 ≤ r < y }
```

39. Calcula la precondición más débil posible *P* que haga válida en cada caso la especificación:

(a)
```
var x : Ent;
{ P }
  it
    x ≠ 0 → x := x-1
  fit
{ x = 0 }
```

(b)
```
var x : Ent;
{ P }
  it
    x ≠ 0 → x := x-2
  fit
{ x = 0 }
```

†(c)
```
var x : Ent;
{ P }
  it x ≠ 0
  → si x > 0
       entonces x := x-2
       sino x := x+3
    fsi
  fit
{ x = 0 }
```

40. Completa la verificación del algoritmo de cálculo del m.c.d. que se presenta a continuación.

```
var x, y : Ent;
{  Pre. P: x = X ∧ y = Y ∧ x > Y ≥ 0 }
{ Inv. I: x > y ≥ 0 ∧ mcd(x,y) = mcd(X,Y)
  Cota C: y                                  }
  it y ≠ 0
  → { x > y > 0 ∧ mcd(x,y) = mcd(X,Y) }
      <x, y> := <y, x mod y>
  fit
{ Post. Q: x = mcd(X,Y) }
```

41. Verifica usando los invariantes y expresiones de acotación indicados:

(a) Suma de un vector de enteros (suponemos N ≥ 1, cte.).

```
var v : Vector [1..N] de Ent; s: Ent;
{  Pre: v = V }
var I : Ent;                        % variable local
  (s, I) := (0, 1);
{ Inv. I: v = V ∧ 1 ≤ I ≤ N+1 ∧ s = ∑i : 1 ≤ i < I : v(i)
  Cota C: N−I+1                                                  }
  it I ≠ N+1
  → s := s+v(I);
     I := I+1 { I }
  fit
fvar
{ Post: v = V ∧ s = ∑i : 1 ≤ i ≤ N : v(i) }
```

(b) Suma de una matriz de enteros (suponemos N, M ≥ 1, ctes.).

```
var v : Vector [1..N, 1..M] de Ent; s: Ent;
{  Pre: v = V }
var I, J : Ent;                     % variables locales
  (s, I) := (0, 1);
{ InvExt. I: v = V ∧ 1 ≤ I ≤ N+1 ∧
             s = ∑i,j : 1 ≤ i < I ∧ 1 ≤ j ≤ M : v(i,j)
  CotaExt. C : N−I+1
}
  it I ≠ N+1
  → J := 1;
   { InvInt. J: v = V ∧ 1 ≤ I ≤ N ∧ 1 ≤ J ≤ M+1 ∧
                  s = ∑i,j : 1 ≤ i < I ∧ 1 ≤ j ≤ M : v(i,j) +
                      ∑j : 1 ≤ j < J : v(I,j)
     CotaInt. D: M−J+1
}
     it J ≠ M+1
     → s := s+v(I,J);
        J := J+1 { J }
     fit;
  I := I + 1  { I }
  fit
fvar
{ Post: v = V ∧ s = ∑i,j : 1 ≤ i ≤ N ∧ 1 ≤ j ≤ M : v(i,j) }
```

42. Construye y verifica algoritmos para los problemas que siguen, anotando pre- y postcondiciones, invariantes y expresiones de acotación:

(a) Cálculo de la matriz producto de dos matrices dadas.
(b) Cálculo del determinante de una matriz dada.

Procedimientos

43. Escribe especificaciones de procedimientos adecuados para las siguientes tareas:

(a) Calcular el m.c.d. de dos enteros. (cfr. Ej. 40)
(b) Calcular el cociente y el resto de una división entera. (cfr. Ej. 38)

(c) Buscar un elemento dentro de un vector.
(d) Intercambiar los contenidos de dos posiciones de un vector. (cfr. Ej. 10)
(e) Ordenar un vector de enteros. (cfr. Ej. 10)

Indica en cada caso la precondición, la postcondición, los tipos y los modos de uso de los parámetros.

†**44.** Diseña un procedimiento que satisfaga la especificación del apartado (d) del ejercicio anterior, y verifica su corrección con ayuda de la regla de verificación de asignaciones a vectores.

45. Supongamos un procedimiento *acumula* que satisfaga la especificación

```
proc acumula( es s: Ent; e x: Ent );
{ Pre.: s = S ∧ x = X }
{ Post.: s = S + x ∧ x = X}
fproc
```

(a) Refuta mediante un contraejemplo:

```
{ suma = S } acumula(suma, 2*suma) { suma = S + 2*suma }
```

¿Qué falla al intentar aplicar las reglas de verificación?

(b) Verifica:

```
{ suma > a*b } acumula(suma, 2*a*b) { suma > 3*a*b }
```

Funciones

46. Algunos de los procedimientos del ejercicio 43 pueden ser planteados como funciones. Localízalos y escribe las correspondientes especificaciones.

47. Usando el algoritmo del ejercicio 40, construye una función correcta para el cálculo del m.c.d. de dos enteros.

48. Supongamos una función *doble* que satisfaga la especificación

```
func doble( x: Ent ) dev y: Ent;
{ Pre.: x = X }
{ Post.: y = 2*x ∧ x = X }
ffunc
```

(a) Refuta mediante un contraejemplo:

```
{ Cierto } x := doble(x) { x = 2*x }
```

¿Qué falla al intentar aplicar las reglas de verificación?

(b) Verifica:

```
{ x = y } x := y + doble(x) { x = 3*y }
```

OJO: Es necesario transformar previamente el planteamiento con ayuda de una variable local, para que sea aplicable la regla de verificación de llamadas. Queda:

```
{ x = y }
  var z : Ent;
  inicio
    z := doble(x); x := y + z
  fvar
{ x = 3*y }
```

Análisis de algoritmos iterativos

49. El siguiente algoritmo decide si una matriz cuadrada de enteros dada es o no simétrica:

```
func esSim( a: Vector [1..N, 1..N] de Ent ) dev b: Bool;
{ Pre.: a = A }
var I, J: Ent;
inicio
  b := Cierto;
  para I desde 1 hasta N-1 hacer
    para J desde I+1 hasta N hacer
      b := b AND (a(I,J) = a(J,I))
    fpara
  fpara
{ Post.: a = A ∧ ( b ↔ ∀i,j : 1 ≤ i < j ≤ N : a(i,j) = a(j,i) ) }
  dev b
ffunc
```

Considerando la dimensión N de la matriz como medida del tamaño de los datos, estima el tiempo de ejecución en el caso peor, en los dos supuestos siguientes:

(a) Tomando como única acción característica la asignación que aparece en el bucle más interno.
(b) Tomando como acciones características los accesos a posiciones de la matriz.

50. El siguiente algoritmo de búsqueda decide si un entero dado aparece o no dentro de un vector de enteros dado:

```
func busca( v: Vector [1..N] de Ent; x: Ent ) dev encontrado: Bool;
{ Pre.: v = V ∧ x = X }
var I: Ent;
inicio
  I := 1;
  encontrado := falso;
  it I ≤ N AND NOT encontrado
  →  encontrado := ( v(I) = x );
     I := I+1
  fit
{ Post.: v = V ∧ x = X ∧ ( encontrado ↔ ∃i : 1 ≤ i ≤ N : v(i) = x ) }
  dev encontrado
ffunc
```

Analiza la *complejidad en promedio* del algoritmo, bajo las hipótesis siguientes: (a) la probabilidad de que x aparezca en v es un valor constante p, $0 < p < 1$; y (b) la probabilidad de que x aparezca en la posición i de v (y no en posiciones anteriores) es la misma para cada índice i, $1 \leq i \leq N$.

51. Aplicando las definiciones de las notaciones O y Ω, razona.

(a) $(n+1)^2$ es $O(n^2)$ **(b)** $3n^3+2n^2$ es $O(n^3)$
(c) 3^n no es $O(2^n)$ **(d)** n^3+2n^2 es $\Omega(n^3)$

52. Demuestra que si $T_1 \in O(f_1)$ y $T_2 \in O(f_2)$, se tiene:

(a) *Regla de la suma*: $T_1(n) + T_2(n)$ es $O(\max(f_1(n), f_2(n)))$.

(b) *Regla del producto*: $T_1(n) * T_2(n)$ es $O(f_1(n) * f_2(n))$.

53. Aplicando las definiciones de las notaciones O, Ω y Θ, demuestra cada una de las afirmaciones siguientes:

(a) $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$.
(b) $O(f) \subseteq O(g) \Leftrightarrow f \in O(g)$.
(c) $O(f) = O(g) \Leftrightarrow f \in O(g) \wedge g \in O(f)$.
(d) $O(f) \subset O(g) \Leftrightarrow f \in O(g) \wedge g \notin O(f)$.
(e) $f \in O(g) \Leftrightarrow g \in \Omega(f)$.
(f) $f \in \Theta(g) \Leftrightarrow f \in O(g) \wedge g \in O(f)$.
(g) $f \in \Theta(g) \Leftrightarrow f \in O(g) \wedge f \in \Omega(g)$.

54. Demuestra las afirmaciones que siguen:

(a) $\lim_n \dfrac{f(n)}{g(n)} = 0 \Rightarrow O(f) \subset O(g)$.

(b) $\lim_n \dfrac{f(n)}{g(n)} \neq 0 \Rightarrow f \in \Theta(g)$.

55. Aplicando el ejercicio anterior, demuestra:

(a) $\dfrac{n*(n-1)}{2}$ es $\Theta(n^2)$.
(b) $p(n)$ es $\Theta(n^k)$, supuesto que $p(n)$ sea un polinomio de grado k.
(c) $\log_a n$ es $\Theta(\log_b n)$, supuesto que $a, b > 1$.

56. Supongamos dos algoritmos A, B que resuelven el mismo problema, pero tales que $T_A(n)$ es $O(\log n)$, mientras que $T_B(n)$ es $O(2^n)$. Responde razonadamente a las siguientes preguntas:

(a) ¿Se puede asegurar que el tiempo de ejecución de B será superior al de A en todos los casos?

(b) Si se duplica el tamaño de los datos, ¿cómo aumenta el tiempo de ejecución para cada uno de los dos algoritmos?

(c) Si se cambia la máquina por otra el doble de rápida, ¿cómo aumenta para cada uno de los dos algoritmos el tamaño de los datos que pueden procesarse en un tiempo fijado?

57. El algoritmo siguiente resuelve el mismo problema que el algoritmo del ejercicio 49. Analiza este algoritmo y estima el orden de magnitud de su tiempo de ejecución en el caso peor. ¿Se trata de un algoritmo más eficiente que el del ejercicio 49? ¿Por qué?

```
func esSim( a: Vector [1..N, 1..N] de Ent ) dev b: Bool;
{ Pre.: a = A }
var I, J: Ent;
inicio
  b := Cierto;
  I := 1;
  it I < N AND b
  →  J := I+1
     it J ≤ N AND b
     → b := b AND ( a(I,J) = a(J,I) )
       J := J +1
     fit;
     I := I+1
  fit
{ Post.: a = A ∧ ( b ↔ ∀i,j : 1 ≤ i < j ≤ N : a(i,j) = a(j,i) ) }
  dev b
ffunc
```

58. Analiza la complejidad del algoritmo de multiplicación del ejercicio 36(b), tomando como tamaño de los datos n = y. Demuestra que el tiempo de ejecución en el caso peor es $O(\log n)$.

(*Sugerencia*: Razonando por inducción sobre n, demuestra que para y = n ≥ 2, el cuerpo del bucle del algoritmo se ejecuta t(n) veces, siendo $t(n) \leq 2 \cdot (\log n)$.)

Derivación de algoritmos iterativos

59. Deriva un algoritmo de división entera que cumpla la especificación siguiente y que utilice solamente operaciones aritméticas de suma y resta. Utiliza la técnica de descomposición conjuntiva de la postcondición para descubrir un invariante adecuado. analiza el tiempo de ejecución del algoritmo obtenido.

```
func divMod( x, y : Ent ) dev c, r : Ent;
{ P : x = X ∧ y = Y ∧ x ≥ 0 ∧ y > 0 }
{ Q : x = X ∧ y = Y ∧ x = c*y + r ∧ 0 ≤ r < y }
ffunc
```

60. Deriva y analiza un algoritmo iterativo que cumpla la especificación siguiente, usando solamente las operaciones aritméticas de suma y producto, y descubriendo un invariante adecuado a partir de una descomposición conjuntiva de la postcondición.

```
func raíz( x : Ent ) dev r : Ent;
{ P : x = X ∧ x ≥ 0 }
{ Q : x = X ∧ 0 ≤ r ∧ r² ≤ x < (r+1)² }
ffunc
```

61. Dados un vector `v: Vector [1..N] de Ent` y un número `x: Ent`, se quiere calcular la posición de *v* con el menor índice posible que contenga el valor *x*. Formaliza una especificación pre/post y deriva a partir de ella un algoritmo correcto. OJO: no se supone que el vector esté ordenado.

62. Dados una matriz `a: Vector [1..N, 1..M] de Ent` y un número `x: Ent`, se quiere calcular una posición de *a* donde aparezca el valor *x*, con índice de fila lo menor posible, e índice de columna lo menor posible dentro de esa fila. Escribe una especificación formal y deriva a partir de ella un algoritmo correcto. OJO: no se supone que la matriz esté ordenada.

63. Deriva un algoritmo iterativo que calcule el producto escalar de dos vectores dados, a partir de la especificación siguiente:

```
func prodEsc( u, v : Vector[1..N] de Real ) dev r : Real;
{ P : N ≥ 1 ∧ v = V ∧ u = U }
{ Q : r = ∑ i : 1 ≤ i ≤ N : u(i)*v(i) ∧ u = U ∧ v = V }
ffunc
```

Para descubrir un invariante adecuado, intenta generalizar la postcondición:

(a) Sustituyendo la constante N por una nueva variable.
(b) Sustituyendo la constante 1 por una nueva variable.

Observa que (a) conduce a un bucle ascendente, mientras que (b) conduce a un bucle descendente.

64. Deriva un algoritmo iterativo que satisfaga la siguiente especificación pre/post, partiendo de una generalización conveniente de la postcondición para descubrir un invariante:

```
func sumaYresta ( v : Vector[1..N] de Ent ) dev s : Ent;
{ P : N ≥ 1 ∧ v = V }
{ Q : v = V ∧ s = ∑ i : 1 ≤ i ≤ N : (-1)^i * v(i) }
ffunc
```

65. La siguiente especificación corresponde a un algoritmo que debe decidir si un vector de enteros dado está o no ordenado.

```
func esOrd ( v : Vector[1..N] de Ent ) dev b : Bool;
{ P : N ≥ 1 ∧ v = V }
{ Q : v = V ∧ ( b ↔ ord(v) ) }
ffunc
```

donde "ord(v)" indica que *v* está ordenado ascendentemente. Observa que

```
Q ⇔ v = V ∧ ( b ↔ ord( v, 1, N ) )
```

siempre que se defina

```
ord( v, i, j ) ⇔def ∀ k,l : 1≤k<l≤j : v(k) ≤ v(l)
```

Usando la nueva forma de la postcondición, descubre un invariante y una expresión de acotación adecuados, y deriva un algoritmo correcto de complejidad *O*(N). Finalmente, trata de optimizar fortaleciendo la condición de iteración, de modo que se evite el recorrido completo de *v* cuando *v* no esté ordenado.

66. Considera otra vez la especificación pre/post del ejercicio 60. Deriva un nuevo algoritmo iterativo basándote en las dos ideas siguientes:

- Generalizar la postcondición reemplazando la expresión $(r+1)^2$ por otra que introduzca una nueva variable s^2.
- Avanzar asignando el valor medio `(r+s) div 2` a una de las dos variables *r* o *s*, según convenga para el mantenimiento del invariante.

Analiza la eficiencia del algoritmo obtenido y compara con la solución del ejercicio 60.

67. Considera la siguiente especificación para un algoritmo que debe calcular en *k* el logaritmo en base 2 por defecto de *n*:

```
func logBin ( n : Ent ) dev k : Ent;
{ P : n = N ∧ n ≥ 1 }
{ Q : n = N ∧ k ≥ 0 ∧ 2^k ≤ n < 2^(k+1) }
ffunc
```

Deriva un algoritmo correcto de complejidad *O*(log n) utilizando

Inv. I: $n = N \wedge m \geq 1 \wedge k \geq 0 \wedge 2^k * m \leq n < 2^k * (m+1)$

Cond. Rep. B.: $m \neq 1$

Cota C: m

El invariante y la condición de repetición se han descubierto a partir de una generalización de la postcondición. Explica cómo.

68. Los números de Fibonacci se definen matemáticamente por medio de la siguiente ley de recurrencia:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n–2) + fib(n–1), para todo n ≥ 2
```

Usando la técnica de introducción de un invariante auxiliar, deriva un algoritmo iterativo *O*(n) que calcule el n-ésimo número de Fibonacci, cumpliendo la especificación:

```
func fib ( n : Ent ) dev x : Ent;
{ P : n = N ∧ n ≥ 0 }
{ Q : n = N ∧ x = fib(n) }
ffunc
```

69. También en este caso se pide derivar un algoritmo correcto, usando nuevas variables locales y descubriendo un invariante auxiliar adecuado durante la derivación. La especificación y el invariante inicial que se proponen son como sigue:

```
func numParejas ( v : Vector[1..N] de Ent ) dev r : Ent;
{ P : N ≥ 1 ∧ v = V }
{ Q : v = V ∧ r = #i,j : 1 ≤ i < j ≤ N : v(i) ≤ 0 ∧ v(j) ≥ 0 }
```

```
    ffunc
```

$I_0 : v = V \wedge 0 \le n \le N \wedge r = \#i,j : 1 \le i < j \le n : v(i) \le 0 \wedge v(j) \ge 0$

Obsérvese que se cumple: $I_0 \wedge n = N \Rightarrow Q$. Se exige que el algoritmo obtenido tenga complejidad $O(n)$.

70. Especifica y deriva un algoritmo $O(n)$ que calcule el número de *concordancias* de un vector `v: Vector[1..N] de Ent` dado como parámetro. Se entiende que hay que contar una concordancia por cada pareja de índices *i,j* tales que $1 \le i < j \le N$ y *v(i)*, *v(j)* tengan el mismo signo. Por convenio, el número 0 no tiene signo, a efectos de resolver este ejercicio.

71. Especifica y deriva un algoritmo iterativo que, dado un entero no negativo *n*, calcule $r = \sum i : 0 \le i \le n : i!$. Se exige que el algoritmo obtenido sea $O(n)$ y utilice únicamente operaciones aritméticas de suma y producto.

72. Especifica y deriva un algoritmo iterativo de complejidad $O(n)$ que calcule el número de *índices puculiares* de un vector de tipo `Vector[1..N] de Ent`, entendiendo que un índice *i* ($1 \le i \le N$) es *peculiar* si *v(i)* es igual al número de ceros almacenados en *v* en posiciones con índice mayor que *i*.

Algoritmos de búsqueda en vectores

73. Aplica la técnica de derivación de algoritmos iterativos para construir funciones de búsqueda que satisfagan las especificaciones siguientes:

(a) *Búsqueda secuencial* (cfr. Ej. 61):

```
func busca( v : Vector[1..N] de Ent; x : Ent ) dev p : Ent;
```

$\{ P_0 : N \ge 1 \wedge v = V \wedge x = X \}$

$\{ Q_0 : v = V \wedge x = X \wedge 1 \le p \le N \wedge (\forall i : 1 \le i < p : v(i) \ne x) \wedge (v(p) = x \vee p = N) \}$

```
ffunc
```

(b) *Búsqueda secuencial con centinela*:

```
func busca( v : Vector[1..N] de Ent; x, q : Ent ) dev p : Ent;
```

$\{ P_0 : v = V \wedge x = X \wedge q = Q \wedge 1 \le q \le N \wedge v(q) = x \}$

$\{ Q_0 : v = V \wedge x = X \wedge q = Q \wedge 1 \le p \le q \wedge (\forall i : 1 \le i < p : v(i) \ne x) \wedge v(p) = x \}$

```
ffunc
```

Observa que el algoritmo obtenido en el apartado (b) tiene una condición de iteración más simple.

74. Deriva una función que realice un algoritmo $O(\log N)$ de *búsqueda binaria* en un vector ordenado, partiendo de la especificación siguiente:

```
func busca( v : Vector[1..N] de Ent; x : Ent ) dev p : Ent;
```

$\{ P_0 : N \ge 1 \wedge v = V \wedge x = X \wedge ord(v) \}$

$\{ Q_0 : v = V \wedge x = X \wedge 0 \le p \le N \wedge \forall i : 1 \le i \le p : v(i) \le x \wedge \forall i : p+1 \le i \le N : x < v(i) \}$

```
ffunc
```

Interpreta la postcondición. ¿Qué dice Q_0 en los casos extremos, es decir, para p=0 y p=N? ¿Cómo podemos deducir de Q_0 si x está o no en el vector?

Pista: Generaliza la postcondición observando que

$$Q_0 \Leftarrow v = V \wedge x = X \wedge 0 \le p < q \le N+1 \wedge \forall i : 1 \le i \le p : v(i) \le x \wedge \forall i : q \le i \le N : x < v(i) \wedge q = p+1$$

Otro algoritmo que usa búsqueda binaria ha aparecido en el ejercicio 66.

Algoritmos de ordenación de vectores

75. Comenta el significado de la especificación siguiente, y deriva un procedimiento $O(n)$ que la satisfaga.

```
proc inserta( es v : Vector[1..N] de Ent; e p : Ent );
{ P0 : 1 ≤ p < N ∧ v = V ∧ p = P ∧ ∀i,j : 1 ≤ i < j ≤ p : v(i) ≤ v(j) }
{ Q0 : p = P ∧ permut(v,V) ∧ ∀i,j : 1 ≤ i < j ≤ p+1 : v(i) ≤ v(j) }
fproc
```

Pista: Generaliza la postcondición observando que

$$Q_0 \Leftarrow p = P \wedge permut(v,V) \wedge 0 \le m \le p \wedge \forall i,j : 1 \le i < j \le m : v(i) \le v(j) \wedge \forall i,j : m+1 \le i < j \le p+1 : v(i) \le v(j) \wedge \forall i,j : 1 \le i \le m \wedge m+2 \le j \le p+1 : v(i) \le v(j) \wedge (m = 0 \vee v(m) \le v(m+1))$$

76. Deriva un procedimiento $O(n^2)$ para la ordenación de un vector mediante el *algoritmo de inserción*, partiendo de la especificación siguiente:

```
proc ordena( es v : Vector[1..N] de Ent );
{ P0 : v = v ∧ N ≥ 1 }
{ Q0 : permut(v,V) ∧ ord(v) }
fproc
```

siendo $ord(v) \Leftrightarrow_{def} \forall i,j : 1 \le i < j \le N : v(i) \le v(j)$

Pista: Generaliza la postcondición observando que

$$Q_0 \Leftarrow permut(v,V) \wedge \forall i,j : 1 \le i < j \le n : v(i) \le v(j) \wedge n = N$$

Descompón para encontrar un invariante y aprovecha el procedimiento del ejercicio anterior para diseñar la acción encargada de su restablecimiento.

77. Deriva una función $O(N–p+1)$ que satisfaga la especificación siguiente:

```
func buscaMin( v : Vector[1..N] de Ent; p : Ent ) dev m : Ent;
{ P0 : 1 ≤ p < N ∧ v = V ∧ p = P }
{ Q0 : p = P ∧ v = V ∧ p ≤ m ≤ N ∧ v(m) = min k : p ≤ k ≤ N : v(k) }
ffunc
```

78. Deriva un procedimiento $O(n^2)$ para la ordenación de un vector mediante el *algoritmo de selección*, partiendo de la misma especificación del ejercicio 76, pero tomando ahora

$$ord(v) \Leftrightarrow_{def} \forall i : 1 \le i < N : (\forall j : i < j \le N : v(i) \le v(j))$$

Pista: Generaliza la postcondición observando que

$$Q_0 \Leftarrow \texttt{permut(v,V)} \wedge \forall \texttt{i} : 1 \leq \texttt{i} < \texttt{n} : (\forall \texttt{j} : \texttt{i} < \texttt{j} \leq \texttt{N} : \texttt{v(i)} \leq \texttt{v(j)}) \wedge \texttt{n} = \texttt{N}$$

Descompón para encontrar un invariante y aprovecha el procedimiento del ejercicio anterior para diseñar la acción encargada de su restablecimiento.

CAPÍTULO 2
DISEÑO DE ALGORITMOS RECURSIVOS

2.1 Introducción a la recursión

En este apartado introducimos el concepto de programa recursivo, presentamos los distintos esquemas a los que se ajustan las definiciones recursivas, mostramos ejemplos y fijamos la terminología básica.

Optamos por una solución recursiva cuando sabemos cómo resolver de manera directa un problema para un cierto conjunto de datos, y para el resto de los datos somos capaces de resolverlo utilizando la solución al mismo problema con unos datos "más simples".

Cualquier solución recursiva se basa en un análisis de los datos, $\vec{x}$, para distinguir los casos de solución directa y los casos de solución recursiva:

— caso directo (caso base), $\vec{x}$ es tal que el resultado $\vec{y}$ puede calcularse directamente de forma sencilla.

— caso recursivo, sabemos cómo calcular a partir de $\vec{x}$ otros datos más pequeños $\vec{x}'$, y sabemos además cómo calcular el resultado $\vec{y}$ para $\vec{x}$ suponiendo conocido el resultado $\vec{y}'$ para $\vec{x}'$.

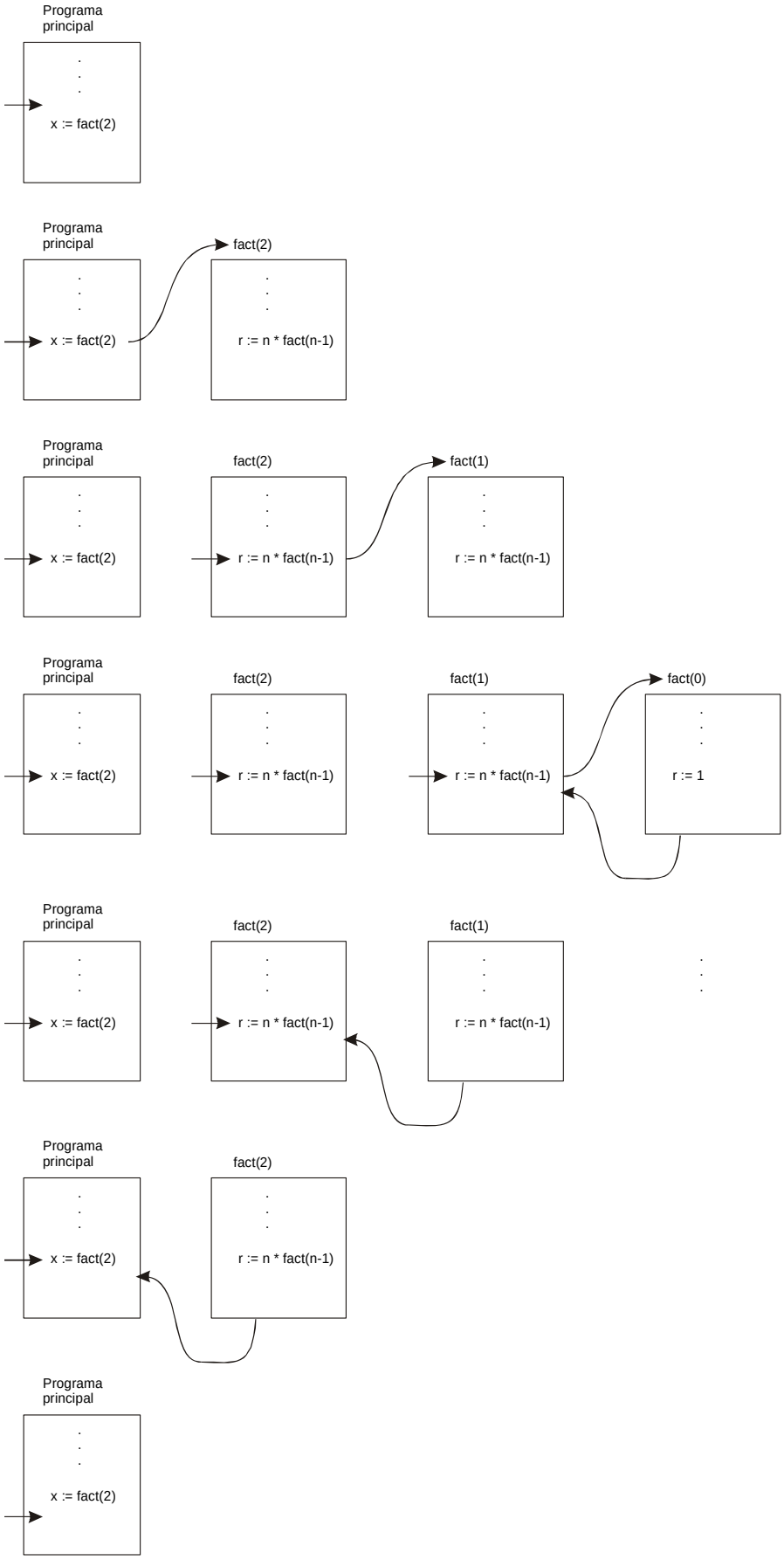
Para implementar soluciones recursivas en un lenguaje de programación tenemos que utilizar una acción que se invoque a sí misma (con datos cada vez "más simples"). Las funciones y los procedimientos son el mecanismo que permite que unas acciones invoquen a otras, y es por tanto el mecanismo que utilizamos para implementar soluciones recursivas: procedimientos o funciones que se invocan a sí mismos.

Para entender la recursividad a veces resulta útil considerar cómo se ejecutan las acciones recursivas en una computadora, como si se crearan múltiples copias del mismo código, operando sobre datos diferentes. Mercedes cuando les explicó la recursividad insistió mucho en la construcción de trazas de los procedimientos y funciones recursivas. El ejemplo clásico del factorial:

```
fun fact ( n : Nat ) dev r : Nat;
{ P₀ : N = n }
inicio
  si
    n = 0 → r := 1
  □ n > 0 → r := n * fact(n-1)
  fsi
{ Q₀ : n = N ∧ r = Π i : 1 ≤ i ≤ n : i }[1]
  dev r
ffunc
```

[1] Nótese que para n=0 el aserto de dominio del producto extendido define el conjunto vacío, y por lo tanto el producto extendido da 1 como resultado, el elemento neutro del producto.

La ejecución de fact(2)?



En realidad no se realizan copias del código sino simplemente de las variables locales y de la dirección de retorno, pero valga el símil.

La importancia de la recursión radica en que:

- Es un método muy potente de diseño y razonamiento formal
- Tiene una relación natural con la inducción.
- Facilita conceptualmente la resolución de problemas y el diseño de algoritmos
- Es fundamental en los lenguajes de programación funcionales

Al hilo de esto, señalar que hay algunos autores que consideran que la recursión es un método más natural de resolver problemas que la repetición (iteración). Es por ello que, por ejemplo, en el libro de Peña se trata primero el diseño recursivo y después el iterativo. Relacionado con esta postura nos encontramos con que se proponen cursos de introducción a la programación utilizando lenguajes funcionales, donde el mecanismo natural de resolución de problemas es la recursión. Mi opinión personal es que determinados problemas se resuelven más fácilmente con un diseño iterativo mientras que otros "piden" una solución recursiva; de modo que a veces resulta forzado intentar encontrar diseños recursivos para ciertos problemas cuya solución iterativa es directa, y viceversa.

En este tema nos vamos a limitar al estudio de **funciones** recursivas. Los procedimientos no aportan nada más y el tratamiento es menos engorroso con las funciones. Al trasladar los resultado sobre funciones a procedimientos sólo hay que tomar algunas precauciones con los parámetros de entrada/salida. Vamos a analizar los diferentes esquemas a los que se ajustan las funciones recursivas, dependiendo del número de llamadas recursivas y de la función de combinación.

1.1.1 Recursión simple

Decimos que una acción recursiva tiene recursión simple si cada caso recursivo realiza a lo sumo una llamada recursiva.

El esquema de declaración de funciones recursivas simples:

```
func nombreFunc (x1:τ1; … ; xn:τn) dev y1:δ1; … ; ym:δm;
{ P0 : P' ∧ x1 = X1 ∧ … ∧ xn = Xn }
cte … ;
var … ;
inicio
  si
     d(x⃗) → y⃗ := g(x⃗)
  □ ¬d(x⃗) → y⃗ := c(x⃗, nombreFunc( s(x⃗) ))  % abuso de notación
  fsi;
{ Q0 : Q' ∧ x1 = X1 ∧ … ∧ xn = Xn }
```

```
  dev <y₁, … , yₘ>
ffunc
```

donde

— $d(\vec{x})$ es la condición que determina el caso directo

— $\neg d(\vec{x})$ es la condición que determina el caso recursivo

— *g* calcula el resultado en el caso directo

— *s* (función sucesor) calcula los argumentos para la siguiente llamada recursiva

— *c* (función de combinación) obtiene la combinación de los resultados de la llamada recursiva *nombreFunc*($s(\vec{x})$) junto con los datos de entrada $\vec{x}$, proporcionando así el resultado de *nombreFunc*($\vec{x}$)

---

debemos tener en cuenta que

— d, g, s y c pueden ser funciones o expresiones.

— $d(\vec{x})$ y $\neg d(\vec{x})$ pueden desdoblarse en una alternativa con varios casos

veamos cómo la función factorial se ajusta a este esquema de declaración:

$d(n) \Leftrightarrow n = 0$

$\neg d(n) \Leftrightarrow n > 0$

$g(n) = 1$

$s(n) = n–1$

$c(n, fact\ (n–1)) = n * fact(n–1)$

A la recursión simple también se la conoce como **recursión lineal** porque el número de llamadas recursivas depende linealmente del tamaño de los datos.

Una caso particular de recursión simple se obtiene cuando la función de combinación se limita a transmitir el resultado de la llamada recursiva. Este tipo de recursión simple recibe el nombre de **recursión final** (*tail recursion*). Nótese que en una función recursiva final, el resultado devuelto será siempre el obtenido en uno de los casos base, ya que la función de combinación se limita a transmitir el resultado que recibe, sin modificarlo. Se llama *final* porque lo último que se hace en cada pasada es la llamada recursiva.

Las funciones recursivas finales tienen la interesante propiedad de que pueden ser traducidas de manera directa a soluciones iterativas, más eficientes.

Como ejemplo de función recursiva final veamos el algoritmo de cálculo del mcd por el método de Euclides.

```
func mcd( a, b : Ent ) dev m : Ent;
{ P₀ : a = A ∧ b = B ∧ a > 0 ∧ b > 0 }
inicio
  si
```

```
    a > b → m := mcd(a-b, b)
  □ a < b → m := mcd(a, b-a)
  □ a = b → m := a
  fsi
{ Q₀ : a = A ∧ b = B ∧ m = MCD(a,b) }
  dev m
ffunc
```

Tenemos aquí un ejemplo donde se distinguen más de dos casos, en particular dos casos recursivos y un caso base. La función es recursiva simple porque en cada caso recursivo se hace una sola llamada recursiva. Veamos cómo se ajusta al esquema de recursión simple:

$d(a,b) \Leftrightarrow a = b$		
$\neg d_1(a,b) \Leftrightarrow a > b$	$\neg d_2(a,b) \Leftrightarrow a < b$	dos caso recursivos
$g(a,b) = a$		
$s_1(a,b) = (a-b, a)$	$s_2(a,b) = (a, b-a)$	dos funciones sucesor
$c(a,b,mcd(x,y)) = mcd(x,y)$		

Observamos cómo la función de combinación se limita a propagar el resultado de la llamada recursiva y que, por lo tanto, nos encontramos ante un ejemplo de recursión final.

1.1.2 Recursión múltiple

Este tipo de recursión se caracteriza por que en cada pasada se realizan varias llamadas recursivas. El esquema correspondiente:

```
func nombreFunc (x₁:τ₁; … ; xₙ:τₙ) dev y₁:δ₁; … ; yₘ:δₘ;
{ P₀ : P' ∧ x₁ = X₁ ∧ … ∧ xₙ = Xₙ }
cte … ;
var … ;
inicio
  si
     d(x⃗) → y⃗ := g(x⃗)
  □ ¬d(x⃗) → y⃗ := c(x⃗, nombreFunc( s₁(x⃗) ), … , nombreFunc( sₖ(x⃗) ))
  fsi;
{ Q₀ : Q' ∧ x₁ = X₁ ∧ … ∧ xₙ = Xₙ }
  dev <y₁, … , yₘ>
ffunc
```

donde

— $k > 1$, indica el número de llamadas recursivas

— $d(\vec{x})$ es la condición que determina el caso directo

— $\neg d(\vec{x})$ es la condición que determina el caso recursivo

— g calcula el resultado en el caso directo

— s_i (funciones sucesor) calculan la descomposición de los datos de entrada para realizar la i-ésima llamada recursiva, para i = 1, …, k

— c (función de combinación) obtiene la combinación de los resultados de las llamadas recursivas *nombreFunc*($s_i(\vec{x})$), con i = 1, … , k, junto con los datos de entrada $\vec{x}$, proporcionando así el resultado de *nombreFunc*($\vec{x}$)

Se puede implementar de manera directa una función que calcula los números de Fibonacci siguiendo el esquema de recursión múltiple:

```
func fibo( n : Nat ) dev r : Nat;
{ P₀ : n = N }
inicio
  si
      n = 0 → r := 0
    □ n = 1 → r := 1
    □ n > 1 → r := fibo(n-1) + fibo(n-2)
  fsi
{ Q₀ : n = N ∧ r = fib(n) }
  dev r
ffunc
```

En esta función tenemos que, según el esquema

$d_1(n) \Leftrightarrow n = 0$ $\quad d_2(n) \Leftrightarrow n = 1$ $\quad$ dos caso directos

$\neg d(n) \Leftrightarrow n > 1$

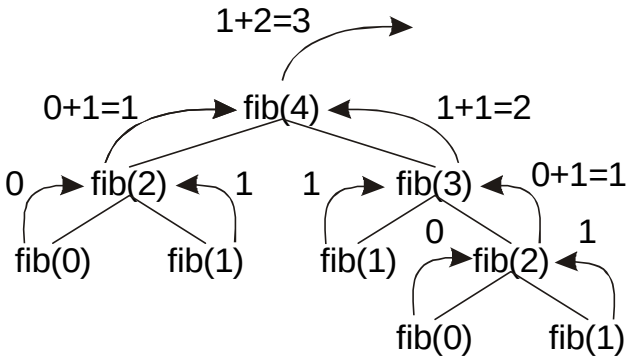
$g(0) = 0$ $\quad g(1) = 1$ $\quad$ (¿$g(n) = n$?)

$s_1(n) = n–1$ $\quad s_2(n) = n–2$

$c(n, fibo(n–1), fibo(n–2)) = fibo(n–1) + fibo(n–2)$

En la recursión múltiple el número de llamadas no depende linealmente del tamaño de los datos; en general será de mayor orden.

Podemos ver un ejemplo con las llamadas que se desencadenan al computar fib(4). Comprobamos que algunos resultados se computan más de una vez:



Para terminar con la introducción recapitulamos los distintos tipos de funciones recursivas que hemos presentado:

- Simple. A lo sumo una llamada recursiva en cada caso recursivo
  - No final. Requiere combinación de resultados
  - Final. No requiere combinación de resultados
- Múltiple. Más de una llamada recursiva en algún caso recursivo.

2.2 Verificación de algoritmos recursivos

Para verificar funciones recursivas podríamos intentar utilizar las reglas de verificación que hemos visto hasta ahora, ya que, de hecho, con la recursividad no hemos añadido ningún elemento nuevo a nuestro lenguaje algorítmico. El problema radica en que debemos verificar las llamadas recursivas

---

```
…
r := n * fact(n-1)
…
```

Y el método de verificación de llamadas depende de que la función o el procedimiento que se invocan tengan una implementación correcta. Pero eso es precisamente lo que estamos intentando demostrar cuando verificamos una función recursiva.

La solución radica en aplicar el principio de inducción, demostrando la corrección para el o los casos base, y suponiendo que la función es correcta para $s(\vec{x})$ demostrar entonces que también es correcta para $\vec{x}$.

1.2.1 Verificación de la recursión simple

Hemos de verificar el cuerpo de una función recursiva simple, que se corresponde con el esquema:

---

si
   d($\vec{x}$) → $\vec{y}$:= g($\vec{x}$)
□ ¬d($\vec{x}$) → $\vec{y}$:= c($\vec{x}$, nombreFunc(s($\vec{x}$)))
fsi;

Para ello, como hemos, dicho utilizaremos el principio de inducción, en este caso el *principio de inducción ordinaria* que, expresada sobre N queda

---

P(0)
n : Nat P(n) ⇒ P(n+1)
———————————
∀ n : Nat: P(n)

O, si consideramos que *R*(x) es la **propiedad de corrección** del cuerpo con respecto a la especificación:

---

d($\vec{x}$) ⇒ R($\vec{x}$)
R(s($\vec{x}$)) ⇒ R($\vec{x}$)
———————————
∀ $\vec{x}$ R($\vec{x}$)

Considerando que s($\vec{x}$) es el "valor anterior" de $\vec{x}$.

Para obtener las reglas de verificación vamos a empezar aplicando las reglas de verificación de la composición alternativa, pues esa es la forma que toma el cuerpo de la función recursiva. En primer lugar, se tiene que garantizar que las barreras estén definidas y que al menos una de ellas se abra:

P_0 ⇒ def(d($\vec{x}$)) ∧ def(¬d($\vec{x}$))
P_0 ⇒ d($\vec{x}$) ∨ ¬d($\vec{x}$)

Por otra parte, debe ser correcta la rama del caso directo:

{ P_0 ∧ d($\vec{x}$) } $\vec{y}$:= g($\vec{x}$) { Q_0 }

que es el caso base de la demostración por inducción antes indicada:

$d(\vec{x}) \Rightarrow R(\vec{x})$

A continuación debemos tratar la corrección de la rama recursiva, para lo cual debemos verificar:

$\{ P_0 \wedge \neg d(\vec{x}) \}\ \vec{y} := c(\vec{x}, nombreFunc(s(\vec{x})))\ \{ Q_0 \}$

como es posible que la función no sea recursiva final, y que por lo tanto la llamada recursiva aparezca dentro de una expresión, introducimos una variable auxiliar que nos facilite la verificación:

$\{ P_0 \wedge \neg d(\vec{x}) \}$
$\vec{y}' := nombreFunc(s(\vec{x}));$
$\vec{y} := c(\vec{x}, \vec{y}')$
$\{ Q_0 \}$

para verificar esta composición secuencial tomamos un aserto intermedio, obtenido como la pmd de la segunda asignación inducida por Q_0:

$\{ P_0 \wedge \neg d(\vec{x}) \}$
$\vec{y}' := nombreFunc(s(\vec{x}));$
$\{ def(c(\vec{x}, \vec{y}')) \wedge Q_0\ [\vec{y} / c(\vec{x}, \vec{y}')] \}$

la segunda asignación será correcta pues hemos tomado como aserto intermedio su pmd; por lo tanto sólo nos resta verificar la llamada recursiva. Aquí es donde aparece la **hipótesis de inducción**, verificaremos la llamada suponiendo que existe una realización correcta para dicha llamada con $s(\vec{x})$.

Aplicando el método de verificación de llamadas tenemos:

1. Se puede realizar la llamada

   $P_0 \wedge \neg d(\vec{x}) \Rightarrow P_0\ [\ \vec{x} / s(\vec{x})\ ]$

   sin embargo, es necesario tener en cuenta un detalle –en la mayoría de las ocasiones irrelevante– y es que en $P_0[\vec{x} / s(\vec{x})]$ pueden aparecer **variables auxiliares de la especificación**, con los mismos identificadores que las que aparecen en P_0; y no deben confundirse. Por ello, en realidad demostraremos:

   $P_0 \wedge \neg d(\vec{x}) \Rightarrow P_0\ [\ \vec{x} / s(\vec{x})\ ]\ [\vec{X} / \vec{X}']$

   siendo $\vec{X}'$ las variables que recogen los valores iniciales de la llamada recursiva

2. Extraer de

   $P_0 \wedge \neg d(\vec{x}) \wedge P_0\ [\ \vec{x} / s(\vec{x})\ ]\ [\vec{X} / \vec{X}']$

   las condiciones que no dependen de los parámetros de salida o entrada/salida. Como P_0 es la precondición de una función, no pueden aparecer en ella los parámetros de salida –en realidad, los parámetros de salida de la llamada recursiva–. Además observamos que si

se ha verificado el paso 1, todo lo que se puede obtener de P_0 [$\vec{x}$ / s($\vec{x}$)] [$\vec{X}$ / $\vec{X}$'] está ya en $P_0 \wedge \neg d(\vec{x})$. Por lo tanto el aserto *R* que consideramos es:

$$P_0 \wedge \neg d(\vec{x})$$

Este último razonamiento también es aplicable en el caso general de llamadas a procedimientos y funciones. La razón por la que en el caso general no aplicamos esta simplificación es que la precondición del procedimiento puede incluir condiciones que nos ayuden a reescribir la precondición de la llamada y sacar a la luz las condiciones implícitas.

3. Demostramos que:

$$P_0 \wedge \neg d(\vec{x}) \wedge Q_0[\ \vec{x}/s(\vec{x}),\ \vec{y}/\vec{y}'\ ]\ [\vec{X}/\vec{X}'] \Rightarrow def(c(\vec{x},\ \vec{y}')) \wedge Q_0[\ \vec{y}/c(\vec{x},\vec{y}')]$$

De nuevo al hacer la sustitución de la postcondición de la llamada recursiva hemos tenido cuidado de renombrar también las variables auxiliares de la especificación [$\vec{X}$ / $\vec{X}$']. Nótese que en la anterior condición es donde aparece la hipótesis de inducción: hemos de demostrar que a partir de la postcondición de la llamada recursiva se puede obtener la postcondición de la llamada actual. Abstrayendo esa condición como una condición *R*

$$R(s(\vec{x})) \Rightarrow R(\vec{x})$$

Lo último que nos queda para demostrar la corrección de una función recursiva es comprobar su terminación. Para ello vamos a definir, al igual que en los bucles, una **expresión de acotación** que dependa de los parámetros de la función y que tome valores en un dominio bien fundamentado

$$t(\vec{x}) : D_{\bar{\tau}} \rightarrow D$$

Tendremos que comprobar entonces que al entrar en la rama recursiva la expresión de acotación esté definida y sea decrementable

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow def(t(\vec{x})) \wedge dec(t(\vec{x}))$$

y que efectivamente la expresión se vaya decrementando al avanzar la recursión:

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow t(s(\vec{x})) \prec t(\vec{x})$$

Con todo esto la verificación de una función recursiva pasa por demostrar los siguientes requisitos:

(r.1) condiciones definidas

$$P_0 \Rightarrow def(d(\vec{x})) \wedge def(\neg d(\vec{x}))$$

(r.2) condiciones exhaustivas

$$P_0 \Rightarrow d(\vec{x})) \vee \neg d(\vec{x})$$

(r.3) caso base

$\{ P_0 \wedge d(\vec{x}) \}\ \vec{y} = g(\vec{x})\ \{ Q_0 \}$

(r.4) definición de la llamada

$P_0 \wedge \neg d(\vec{x}) \Rightarrow P_0\ [\vec{x}/s(\vec{x})]\ [\vec{X}/\vec{X}']$

(r.5) paso inductivo

$P_0 \wedge \neg d(\vec{x}) \wedge Q_0[\ \vec{x}/s(\vec{x}),\ \vec{y}/\vec{y}']\ [\vec{X}/\vec{X}'] \Rightarrow def(c(\vec{x}, \vec{y}')) \wedge Q_0\ [\vec{y}/c(\vec{x}, \vec{y}')]$

(r.6) expresión de acotación

$P_0 \wedge \neg d(\vec{x}) \Rightarrow def(t(\vec{x})) \wedge\ dec(t(\vec{x}))$

(r.7) descenso

$P_0 \wedge \neg d(\vec{x}) \Rightarrow t(s(\vec{x})) \prec t(\vec{x})$

Veamos un ejemplo de aplicación de las reglas de verificación de funciones recursivas, aplicada a la función factorial.

(r.1) condiciones definidas

$N = n \Rightarrow def(n=0) \wedge def(n>0) \Leftrightarrow cierto$

(r.2) condiciones exhaustivas

$N = n \Rightarrow n = 0 \vee n > 0$

$$\begin{array}{rl} & n = 0 \vee n > 0 \\ \Leftarrow & n : Nat \\ \text{declaración} \quad \Leftrightarrow & cierto \Leftarrow N = n \end{array}$$

(r.3) caso base

$\{ N = n \wedge n = 0 \}\ r := 1\ \{ Q_0\colon n = N \wedge r = \Pi\ i : 1 \le i \le n : 1 \}$

$$\begin{array}{rl} & def(1) \wedge Q_0\ [r/1] \\ \Leftrightarrow & cierto \wedge n = N \wedge 1 = \Pi\ i : 1 \le i \le n : 1 \\ \text{fortalecimiento} \quad \Leftarrow & n = N \wedge n = 0 \\ \Leftrightarrow & P_0 \wedge n = 0 \end{array}$$

(r.4) definición de la llamada

$n = N \wedge n > 0 \Rightarrow P_0\ [n/n-1]\ [N/N']$

$$\begin{array}{rl} & P_0\ [n/n-1]\ [N/N'] \\ \Leftrightarrow & n-1 = N' \\ \Leftrightarrow & n-1 : Nat \end{array}$$

$$\begin{array}{ll} & n > 0 \land n : Nat \\ \Leftrightarrow & n = N \land n > 0 \end{array}$$

(r.5) paso inductivo

$$n = N \land n > 0 \land Q_0[\ n/n-1,\ r/r'\ ]\ [N/N'] \ \Rightarrow def(n*r') \land Q_0\ [r/n*r']$$

elaboramos la parte derecha

$$\begin{array}{ll} & def(n*r') \land Q_0\ [r/n*r'] \\ \Leftrightarrow & cierto \land n = N \land n*r' = \Pi\ i : 1 \le i \le n : i \end{array}$$

veamos que de la parte izquierda (hipótesis de inducción) se deduce la parte derecha

$$\begin{array}{ll} & n = N \land n > 0 \land Q_0[\ n/n-1,\ r/r'\ ]\ [N/N'] \\ \Leftrightarrow & n = N \land n > 0 \land n-1 = N' \land r' = \Pi\ i : 1 \le i \le n-1 : i \\ \Rightarrow & n = N \land n * r' = n * (\ \Pi\ i : 1 \le i \le n-1 : i\ ) \\ \Leftrightarrow & n = N \land n * r' = \Pi\ i : 1 \le i \le n : i \\ \Leftrightarrow & def(n*r') \land Q_0\ [r/n*r'] \end{array}$$

(r.6) expresión de acotación

Como expresión de acotación tomamos

$$t(n) = n$$

demostramos

$$n = N \land n > 0 \Rightarrow def(n) \land dec(n) \Leftrightarrow n > 0$$

(r.7) descenso

$$n = N \land n > 0 \Rightarrow n-1 < n \Leftrightarrow cierto$$

Generalizaciones del esquema de recursión simple

La dos generalizaciones básicas son que nos encontremos con múltiples casos base, múltiples casos recursivos, o ambas.

Más de un caso base

Los requisitos que se ven modificados:

(r.1) todas las barreras han de estar definidas

$$P_0 \Rightarrow def(d_1(\vec{x}))\land \ \dots\ \land def(d_p(\vec{x})) \land def(\neg d(\vec{x}))$$

(r.2) condiciones exhaustivas

$P_0 \Rightarrow d_1(\vec{x}) \vee \ldots \vee d_p(\vec{x}) \vee \neg d(\vec{x})$

(r.3) se ha de comprobar la corrección de todos los casos base

$\{ P_0 \wedge d_j(\vec{x}) \}\ \vec{y} = g_j(\vec{x})\ \{ Q_0 \}$

donde las $d_j(\vec{x})$ son las condiciones de los casos directos y las $g_j(\vec{x})$ las funciones que obtiene la solución directa en cada caso, para j= 1, …, p

El resto de los requisitos no se ven modificados pues hacen referencia a los casos recursivos.

Más de un caso recursivo

Los requisitos que se ven modificados

(r.1) todas las barreras han de estar definidas

$P_0 \Rightarrow def(d(\vec{x})) \wedge def(\neg d_1(\vec{x})) \wedge \ldots \wedge def(\neg d_p(\vec{x}))$

(r.2) condiciones exhaustivas

$P_0 \Rightarrow d(\vec{x})) \vee \neg d_1(\vec{x}) \vee \ldots \vee \neg d_p(\vec{x})$

(r.4) Cada llamada recursiva ha de estar definida. Tenemos *p* requisitos de la forma $(r.4)_j$

$P_0 \wedge \neg d_j(\vec{x}) \Rightarrow P_0\ [\vec{x}/s_j(\vec{x})]\ [\vec{X}/\vec{X}']$

(r.5) Se ha de comprobar el paso inductivo para cada caso recursivo. Tenemos *p* requisitos de la forma $(r.5)_j$

$P_0 \wedge \neg d_j(\vec{x}) \wedge Q_0[\ \vec{x}/s_j(\vec{x}),\ \vec{y}/\vec{y}']\ [\vec{X}/\vec{X}'] \Rightarrow def(c_j(\vec{x},\vec{y}')) \wedge Q_0\ [\vec{y}/c_j(\vec{x},\vec{y}')]$

(r.6) Aunque la expresión de acotación es única, debemos verificar que todos los casos recursivos garantizan $(r.6)_j$

$P_0 \wedge \neg d_j(\vec{x}) \Rightarrow def(t(\vec{x})) \wedge\ dec(t(\vec{x}))$

(r.7) Se ha de comprobar el descenso de la expresión de acotación para cualquier descomposición recursiva. Tenemos requisitos $(r.7)_j$ de la forma

$P_0 \wedge \neg d_j(\vec{x}) \Rightarrow t(s_j(\vec{x})) \prec t(\vec{x})$

Siendo las $\neg d_j(\vec{x})$ las condiciones de los casos recursivos, las $s_j(\vec{x})$ las funciones que calculan las descomposiciones recursivas, y las $c_j(\vec{x},\vec{y}')$ las funciones que combinan los resultados, para j= 1, …, p

---

Cuando nos encontremos con una función recursiva que incluya más de un caso directo y más de un caso recursivo, deberemos considerar los dos conjuntos de cambios que acabamos de indicar.

Veamos un ejemplo de función recursiva con varios casos base y varios casos recursivos: el producto de dos números naturales por el método del *campesino egipcio*. La idea del método es ir dividiendo por 2 un operando y multiplicando por 2 el otro, hasta que un operando valga 1

$$a \cdot b = \begin{cases} 0 & \text{si } b = 0 \\ a & \text{si } b = 1 \\ (2 \cdot a) \cdot (b \text{ div } 2) & \text{si } b > 1, \text{ b par} \\ (2 \cdot a) \cdot (b \text{ div } 2) + a & \text{si } b > 1, \text{ b impar} \end{cases}$$

Una declaración para este algoritmo es:

```
func prod ( a, b : nat ) dev r : Nat;
{ P₀ : a = A ∧ b = B }
inicio
  si  b = 0              → r := 0
  □   b = 1              → r := a
  □   (b>1 AND par(b))      → r := prod(2*a,b div 2)
  □   (b>1 AND NOT par(b)) → r := prod(2*a,b div 2) + a
  fsi
{ Q₀ : a = A ∧ b = B ∧ r = a * b }
  dev r
ffunc
```

Nótese que el caso base b=1 es redundante y se puede eliminar.
Apliquemos el método de verificación:

(r.1) Definición de las barreras

```
a = A ∧ b = B ¿⇒? def(b=0) ∧ def(b=1) ∧ def(b>1 AND par(b)) ∧
                  def(b>1 AND NOT par(b))
          ⇔ cierto                  % suponiendo que par está definido
∀n:Nat
```

(r.2) Al menos una barrera se abre

```
a = A ∧ b = B ¿⇒? b=0 ∨ b=1 ∨ (b>1 AND par(b)) ∨ (b>1 AND NOT par(b))
```

$\Leftrightarrow$ b = 0 $\vee$ b = 1 $\vee$ (b>1 $\wedge$ (par(b) OR NOT par(b)))
$\Leftrightarrow$ b = 0 $\vee$ b = 1 $\vee$ b > 1
$\Leftrightarrow$ cierto $\Leftarrow$ a = a $\wedge$ b = B

$(r.3)_1$ Corrección del primer caso base

{ a = A $\wedge$ b = B $\wedge$ b = 0 }
  r := 0
{ a = A $\wedge$ b = B $\wedge$ r = a*b }

$(r.3)_2$ Corrección del segundo caso base

{ a = A $\wedge$ b = B $\wedge$ b = 1 }
  r := a
{ a = A $\wedge$ b = B $\wedge$ r = a*b }

$(r.4)_1$ Es posible hacer la llamada recursiva en el primer caso recursivo

a = A $\wedge$ b = B $\wedge$ (b>1 AND par(b)) ¿$\Rightarrow$? P_0 [a/2*a, b/ b div 2] [A/A',B/B']
$\Leftrightarrow$ 2*a = A' $\wedge$ b div 2 = B'
** $\Leftarrow$ a : Nat $\wedge$ b : Nat
$\Leftarrow$ a = A $\wedge$ b = B

** no es doble implicación porque –1 div 2 $\in$ N pero –1 $\notin$ N.

$(r.4)_2$ Es posible hacer la llamada recursiva en el segundo caso recursivo

a = A $\wedge$ b = B $\wedge$ (b>1 AND NOT par(b)) ¿$\Rightarrow$? P_0 [a/2*a, b/ b div 2]
[A/A',B/B']
$\Leftrightarrow$ 2*a = A' $\wedge$ b div 2 = B'
$\Leftarrow$ a : Nat $\wedge$ b : Nat
$\Leftarrow$ a = A $\wedge$ b = B

$(r.5)_1$ Paso inductivo para el primer caso recursivo

P_0 $\wedge$ (b>1 AND par(b)) $\wedge$ Q_0[a/2*a, b/b div 2, r/r] [A/A',B/B'] ¿$\Rightarrow$? Q_0
aquí no ha hecho falta introducir una variable auxiliar r' porque la función de combinación se limita a transmitir el resultado de la llamada recursiva

a=A $\wedge$ b=B $\wedge$ (b>1 AND par(b)) $\wedge$ 2*a = A' $\wedge$ b div 2 = B' $\wedge$ r = (2*a)* (b div 2) ¿$\Rightarrow$?
a = A $\wedge$ b = B $\wedge$ r = a * b

a = A $\wedge$ b = B $\Rightarrow$ a = A $\wedge$ b = B

$$
\begin{array}{ll}
 & (b > 1 \text{ AND } par(b)) \wedge r = (2*a) * (b \text{ div } 2) \\
\Rightarrow & (b > 1 \text{ AND } par(b)) \wedge r = a * 2*(b \text{ div } 2) \\
\Rightarrow & (b > 1 \text{ AND } par(b)) \wedge r = a * b \\
\Rightarrow & r = a*b
\end{array}
$$

$(r.5)_2$ Paso inductivo para el segundo caso recursivo

$P_0 \wedge (b>1 \text{ AND NOT } par(b)) \wedge Q_0\ [a/2*a,\ b/b \text{ div } 2,\ r/r'] [A/A',\ B/B'] \ ¿\Rightarrow?$
$def(r'+a) \wedge Q_0\ [r/r'+a]$

$$
\begin{array}{ll}
 & def(r'+a) \wedge Q_0\ [r/r'+a] \\
\Leftrightarrow & cierto \wedge a = A \wedge b = B \wedge r'+a = a * b \\
\Leftrightarrow & a = a \wedge b = B \wedge r' = a * (b-1)
\end{array}
$$

$P_0 \Rightarrow a = A \wedge b = B$

$(b>1 \text{ AND NOT } par(b)) \wedge Q_0\ [a/2*a,\ b/b \text{ div } 2,\ r/r'] [A/A',\ B/B']$
$\Rightarrow$
$(b>1 \text{ AND NOT } par(b)) \wedge r' = (2*a) * (b \text{ div } 2)$
$\Rightarrow$
$(b>1 \text{ AND NOT } par(b)) \wedge r' = a * 2 * (b \text{ div } 2)$
$\Rightarrow$
$(b>1 \text{ AND NOT } par(b)) \wedge r' = a * (b-1)$
$\Rightarrow$
$r' = a * (b-1)$

$(r.6)_1$ Definición de la expresión de acotación en el primer caso recursivo

Como expresión de acotación tomamos

$t(a,b) = b$

Se demuestra

$P_0 \wedge (b > 1 \text{ AND } par(b)) \Rightarrow def(b) \wedge dec(b) \Leftrightarrow cierto \wedge b > 0$

$(r.6)_2$ Definición de la expresión de acotación en el segundo caso recursivo

$P_0 \wedge (b > 1 \text{ AND NOT } par(b)) \Rightarrow def(b) \wedge dec(b) \Leftrightarrow cierto \wedge b > 0$

$(r.7)_1$ Decremento de la expresión de acotación en el primer caso recursivo

$P_0 \wedge (b > 1 \text{ AND } par(b)) \Rightarrow (b \text{ div } 2) < b$

se cumple por $b > 0$

$(r.7)_2$ Decremento de la expresión de acotación en el segundo caso recursivo

$P_0 \wedge (b > 1 \text{ AND NOT } par(b)) \Rightarrow (b \text{ div } 2) < b$

se cumple por $b > 0$

1.2.2 Verificación de la recursión múltiple

Hemos de verificar una composición alternativa de la forma:

---

si $d(\vec{x}) \rightarrow \vec{y} := g(\vec{x})$
□ $\neg d(\vec{x}) \rightarrow \vec{y} := c(\vec{x}, nombreFunc(s_1(\vec{x})), \ldots, nombreFunc(s_k(\vec{x})))$
fsi

En el que aparecen varias llamadas recursivas. Para demostrar la corrección hemos de aplicar el principio de inducción completa, que generaliza al principio de inducción *normal* cuando existe más de una descomposición recursiva. Hemos de demostrar que se cumple el paso inductivo para cada una de las descomposiciones recursivas:

$$\frac{n : Nat \qquad \forall i : Nat : 0 \leq i < n : P[i] \Rightarrow P[n]}{\forall\, n : Nat : P[n]}$$

O considerando que $R(\vec{x})$ es la propiedad de corrección del cuerpo con respecto a la especificación, aplicamos la regla:

$$\frac{d(\vec{x}) \Rightarrow R(\vec{x}) \qquad (\forall\, s_i(\vec{x}) : R(s_i(\vec{x}))) \Rightarrow R(\vec{x})}{\forall \vec{x}\ R(\vec{x})}$$

El procedimiento de verificación es análogo al de la recursión simple, excepto por lo que se refiere al caso recursivo, donde hemos de verificar:

$\{ P_0 \wedge \neg d(\vec{x}) \}\ \vec{y} := c(\vec{x}, nombreFunc(s_1(\vec{x})), \ldots, nombreFunc(s_k(\vec{x})))\ \{ Q_0 \}$

para cuya verificación introducimos *k* variables auxiliares:

$\{ P_0 \wedge \neg d(\vec{x}) \}$
$\vec{y}_1 := nombreFunc(s_1(\vec{x}));$
…
$\vec{y}_k := nombreFunc(s_k(\vec{x}));$
$\vec{y} := c(\vec{x}, \vec{y}_1, \ldots, \vec{y}_k)$

$\{ Q_0 \}$

La verificación de cada una de las llamadas recursivas introduce una serie de condiciones, que podemos trasladar al final de la composición y considerar así la verificación de una sola instrucción:

$$\{ P_0 \wedge \neg d(\vec{x}) \wedge Q_0 [\vec{x}/s_1(\vec{x}),\ \vec{y}/\vec{y}_1] [\vec{X}/\vec{X}_1] \wedge \ldots \wedge Q_0 [\vec{x}/s_k(\vec{x}),\ \vec{y}/\vec{y}_k] [\vec{X}/\vec{X}_k] \}$$
$$\vec{y} := c(\vec{x},\ \vec{y}_1, \ldots,\ \vec{y}_k)$$
$$\{ Q_0 \}$$

Para ver que este resultado de corrección es equivalente al anterior tendríamos que ir construyendo la pmd de cada una de las asignaciones intermedias, con lo cual dichas asignaciones serán correctas por el axioma de corrección de la asignación.

Aplicando la regla de verificación de la asignación, la anterior verificación es equivalente a:

$$P_0 \wedge \neg d(\vec{x}) \wedge Q_0 [\vec{x}/s_1(\vec{x}),\ \vec{y}/\vec{y}_1] [\vec{X}/\vec{X}_1] \wedge \ldots \wedge Q_0 [\vec{x}/s_k(\vec{x}),\ \vec{y}/\vec{y}_k] [\vec{X}/\vec{X}_k]$$
$$\Rightarrow$$
$$def(c(\vec{x},\ \vec{y}_1, \ldots,\ \vec{y}_k)) \wedge Q_0 [\vec{y}/c(\vec{x},\ \vec{y}_1, \ldots,\ \vec{y}_k)]$$

---

que es precisamente el paso inductivo; supuestas correctas las llamadas recursivas con las correspondientes descomposiciones, se demuestra que es correcto el paso actual.

Con todo esto los requisitos que hay que comprobar para demostrar la corrección de una función con recursión múltiple:

(r.1) condiciones definidas

$$P_0 \Rightarrow def(d(\vec{x})) \wedge def(\neg d(\vec{x}))$$

(r.2) condiciones exhaustivas

$$P_0 \Rightarrow d(\vec{x})) \vee \neg d(\vec{x})$$

(r.3) caso base

$$\{ P_0 \wedge d(\vec{x}) \}\ \vec{y} = g(\vec{x})\ \{ Q_0 \}$$

(r.4) definición de la llamada

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow P_0 [\vec{x}/s_1(\vec{x})] [\vec{X}/\vec{X}_1] \wedge \ldots \wedge P_0 [\vec{x}/s_k(\vec{x})] [\vec{X}/\vec{X}_k]$$

(r.5) paso inductivo

$P_0 \wedge \neg d(\vec{x}) \wedge Q_0\,[\vec{x}/s_1(\vec{x}),\ \vec{y}/\vec{y}_1]\,[\vec{X}/\vec{X}_1] \wedge \ldots \wedge Q_0\,[\vec{x}/s_k(\vec{x}),\ \vec{y}/\vec{y}_k]\,[\vec{X}/\vec{X}_k]$
$\Rightarrow$
$def(c(\vec{x},\ \vec{y}_1, \ldots,\ \vec{y}_k)) \wedge Q_0\,[\vec{y}/c(\vec{x},\ \vec{y}_1, \ldots,\ \vec{y}_k)]$

(r.6) expresión de acotación

$P_0 \wedge \neg d(\vec{x}) \Rightarrow def(t(\vec{x})) \wedge\ dec(t(\vec{x}))$

(r.7) descenso

$P_0 \wedge \neg d(\vec{x}) \Rightarrow t(s_1(\vec{x})) \prec t(\vec{x}) \wedge \ldots \wedge t(s_k(\vec{x})) \prec t(\vec{x})$

Generalizaciones del esquema de recursión múltiple

La dos generalizaciones básicas son que nos encontremos con múltiples casos base, múltiples casos recursivos, o ambas.

Más de un caso base

Los requisitos que se ven modificados son los mismos que en el esquema de recursión simple, pues la verificación de los dos esquemas tiene en común los pasos (r.1), (r.2) y (r.3).

Más de un caso recursivo

Los requisitos que se ven modificados

(r.1) todas las barreras han de estar definidas

$P_0 \Rightarrow def(d(\vec{x})) \wedge def(\neg d_1(\vec{x})) \wedge \ldots \wedge def(\neg d_p(\vec{x}))$

(r.2) condiciones exhaustivas

$P_0 \Rightarrow d(\vec{x})) \vee \neg d_1(\vec{x}) \vee \ldots \vee \neg d_p(\vec{x})$

(r.4) Cada llamada recursiva ha de estar definida. Para cada caso recursivo podemos tener k_j llamadas recursivas. Tenemos *p* requisitos de la forma $(r.4)_j$

$P_0 \wedge \neg d_j(\vec{x}) \Rightarrow P_0\,[\vec{x}/s_{j1}(\vec{x})]\,[\vec{X}/\vec{X}_1] \wedge \ldots \wedge P_0\,[\vec{x}/s_{jkj}(\vec{x})]\,[\vec{X}/\vec{X}_{kj}]$

(r.5) Se ha de comprobar el paso inductivo para cada caso recursivo. Tenemos *p* requisitos de la forma $(r.5)_j$[2]

$P_0 \wedge \neg d_j(\vec{x}) \wedge Q_0\,[\vec{x}/s_{j1}(\vec{x}),\ \vec{y}/\vec{y}_1]\,[\vec{X}/\vec{X}_1] \wedge\ldots\wedge Q_0\,[\vec{x}/s_{jkj}(\vec{x}),\ \vec{y}/\vec{y}_{kj}]\,[\vec{X}/\vec{X}_{kj}]$

[2] Nótese que reutilizamos las variables auxiliares en los distintos casos recursivos, porque si no deberíamos definir variables $y_{j1}, \ldots, y_{jkj}$

$\Rightarrow$
$\mathtt{def}(c_j(\vec{x}, \vec{y}_1, \ldots, \vec{y}_{k_j})) \wedge Q_0\,[\vec{y}/c_j(\vec{x}, \vec{y}_1, \ldots, \vec{y}_{k_j})]$

(r.6) Aunque la expresión de acotación es única, debemos verificar que todos los casos recursivos garantizan $(r.6)_j$
$P_0 \wedge \neg d_j(\vec{x}) \Rightarrow \mathtt{def}(t(\vec{x})) \wedge \mathtt{dec}(t(\vec{x}))$

(r.7) Se ha de comprobar el descenso de la expresión de acotación para cualquier descomposición recursiva. Tenemos requisitos $(r.7)_j$ de la forma
$P_0 \wedge \neg d_j(\vec{x}) \Rightarrow t(s_{j1}(\vec{x})) \prec t(\vec{x}) \wedge \ldots \wedge t(s_{jk_j}(\vec{x})) \prec t(\vec{x})$

Siendo las $\neg d_j(\vec{x})$ las condiciones de los casos recursivos, las $s_{ji}(\vec{x})$ las funciones que calculan las descomposiciones recursivas, y las $c_j(\vec{x}, \vec{y}_1, \ldots, \vec{y}_k)$ las funciones que combinan los resultados, para j= 1, …, p, i = 1, … , k_j

---

Si en una función aparecen varios casos directos y varios casos recursivos, tendremos que combinar los dos conjuntos de modificaciones arriba descritos.

Veamos como ejemplo de verificación de función con recursión múltiple la verificación de la función *fibo* que presentamos al principio del tema:

```
func fibo( n : Nat ) dev r : Nat;
{ P0 : n = N }
inicio
  si
      n = 0 → r := 0
    □ n = 1 → r := 1
    □ n > 1 → r := fibo(n-1) + fibo(n-2)
  fsi
{ Q0 : n = N ∧ r = fib(n) }
  dev r
ffunc
```

(r.1) Definición de las barreras

$n = N \Rightarrow \mathtt{def}(n=0) \wedge \mathtt{def}(n=1) \wedge \mathtt{def}(n>1)$

(r.2) Condiciones exhaustivas

$n = N \Rightarrow n = 0 \vee n = 1 \vee n > 1 \Leftrightarrow n : \mathtt{Nat} \Leftrightarrow \mathtt{cierto}$

$(r.3)_1$ Corrección del primer caso base

$$\{\ n = N \wedge n = 0\ \}$$
$$r := 0$$
$$\{\ n = N \wedge r = fib(n)\ \}$$

$$n = N \wedge n = 0 \Rightarrow def(0) \wedge n = N \wedge 0 = fib(n)$$

$(r.3)_2$ Corrección del segundo caso base

$$\{\ n = N \wedge n = 1\ \}$$
$$r := 1$$
$$\{\ n = N \wedge r = fib(n)\ \}$$

$$n = N \wedge n = 1 \Rightarrow def(1) \wedge n = N \wedge 1 = fib(n)$$

(r.4) Llamada recursiva

$$n = N \wedge n > 1 \Rightarrow P_0\ [n/\ n-1]\ [N/N_1] \wedge P_0\ [n/\ n-2]\ [N/N_2]$$

$$\begin{aligned} & P_0\ [n/n-1]\ [N/N_1] \\ \Leftrightarrow\ & n-1 = N_1 \\ \Leftrightarrow\ & n-1 : Nat \\ \Leftrightarrow\ & n : Nat \wedge n \geq 1 \\ \Leftarrow\ & n = N \wedge n > 1 \end{aligned}$$

$$\begin{aligned} & P_0\ [n/n-2]\ [N/N_2] \\ \Leftrightarrow\ & n-2 = N_2 \\ \Leftrightarrow\ & n-2 : Nat \\ \Leftrightarrow\ & n : Nat \wedge n \geq 2 \\ \Leftarrow\ & n = N \wedge n > 1 \end{aligned}$$

(r.5) Paso inductivo

hemos de demostrar

$$n = N \wedge n > 1 \wedge Q_0[n/n-1,\ r/r_1]\ [N/N_1] \wedge Q_0[n/n-2,\ r/r_2]\ [N/N_2]$$
$$\Rightarrow$$
$$def(r_1+r_2) \wedge Q_0(r/r_1+r_2)$$

que es equivalente a demostrar

$$n = N \wedge n > 1 \wedge n-1 = N_1 \wedge r_1 = fib(n-1) \wedge n-2 = N_2 \wedge r_2 = fib(n-2)$$
$$\Rightarrow$$
$$cierto \wedge n = N \wedge r_1+r_2 = fib(n)$$

y esto se tiene porque

$$\begin{aligned} & r_1 = fib(n-1) \wedge r_2 = fib(n-2) \\ \Rightarrow\ & r_1 + r_2 = fib(n-1)+fib(n-2) \end{aligned}$$

$$\Rightarrow \quad r_1 + r_2 = fib(n)$$

(r.6) Expresión de acotación

$$t(n) = n$$

$$n = N \wedge n > 1 \Rightarrow def(n) \wedge dec(n) \Leftrightarrow cierto \wedge n > 0$$

(r.7) Avance de la recursión

$$n = N \wedge n > 1 \Rightarrow n-1 < n \wedge n-2 < n \Leftrightarrow cierto$$

2.3 Derivación de algoritmos recursivos

El problema es, dada la especificación

{ P } A { Q }

obtener una acción *A* que la satisfaga. Podemos incluir las soluciones recursivas como una quinta opción sobre la forma de *A*, junto con: asignación, secuencia, selección e iteración.

Nos planteamos resolver *A* como una acción recursiva (siempre ha de ser una acción, procedimiento o función, ya que necesitamos la auto-invocación) cuando podemos obtener –fácilmente– una definición recursiva de la postcondición.

En esencia el método obtendrá el *planteamiento recursivo*, determinará las condiciones de los casos directos y recursivos, obtendrá la solución para el caso directo, se buscará una descomposición recursiva de los datos y se obtendrá la solución para los casos recursivos en términos de las soluciones a los "datos descompuestos". Veámoslo en detalle:

(R.1) **Planteamiento recursivo**. Se ha de encontrar una estrategia recursiva para alcanzar la postcondición, es decir, la solución. A veces, la forma de la postcondición, o de las operaciones que en ella aparecen, nos sugerirá directamente una estrategia recursiva. En otros casos es necesario encontrarla. Más adelante veremos algunas heurísticas para encontrar estrategias recursivas.

(R.2) **Análisis de casos**. Se trata de obtener las condiciones que permiten discriminar los casos directos de los recursivos. A veces será más sencillo obtener la condición del caso directo, y obtener la del recursivo como su negación, y otras ocasiones será más sencillo al revés.

Una vez obtenidas las condiciones, demostramos los requisitos (r.1) y (r.2) de la verificación de algoritmos recursivos:

$$P_0 \Rightarrow def(d(\vec{x})) \wedge def(\neg d(\vec{x}))$$
$$P_0 \Rightarrow d(\vec{x}) \vee \neg d(\vec{x})$$

(R.3) **Caso directo**. Hemos de encontrar la acción que resuelve el caso directo; la podemos derivar a partir de la especificación

$$\{\ P_0 \wedge d(\vec{x})\ \}\ A_1\ \{\ Q_0\ \}$$

De acuerdo con los esquemas de las funciones recursivas intentaremos que A_1 sea de la forma:

$$\vec{y}\ :=\ g(\vec{x})$$

Al verificar la corrección de esta acción estaremos demostrando el requisito (r.3) de la verificación de funciones recursivas.

Si hubiese más de un caso directo repetiríamos este paso para cada uno de ellos.

Si A_1 es una composición alternativa cabe plantearse si no es mejor descomponer el caso directo en varios casos directos.

(R.4) **Descomposición recursiva**. Se trata de obtener la *función siguiente* que nos proporciona los datos que empleamos para realizar la llamada recursiva, a partir de los datos de entrada:

$$s(\vec{x})$$

Si hay más de un caso recursivo obtenemos la función siguiente para cada uno de ellos.

(R.5) **Función de acotación**. Antes de pasar a derivar el caso recursivo, nos interesa determinar si la función siguiente escogida garantiza la terminación de las llamadas. Para ello hemos de obtener la función de acotación $t(\vec{x})$, definida en los casos recursivos y que toma valores no minimales en un dominio bien fundamentado. Demostraremos entonces el requisito (r.6)

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow def(t(\vec{x})) \wedge dec(t(\vec{x}))$$

Si hay más de un caso recursivo se ha de verificar esta condición para cada uno de ellos.

(R.6) **Terminación**. Demostramos entonces que la función de acotación escogida se decrementa en cada llamada –requisito (r.7)–:

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow t(s(\vec{x})) \prec t(\vec{x})$$

Si hay más de un caso recursivo se ha de comprobar este requisito para cada uno de ellos.

(R.7) **Llamada recursiva**. Pasamos a ocuparnos entonces del caso recursivo. Empezamos verificando que las descomposiciones recursivas permiten realizar las llamadas recursivas. Es decir, comprobamos el requisito (r.4):

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow P_0[x/s(\vec{x})]\ [\vec{X}/\vec{X}']$$

De nuevo, este requisito ha de comprobarse para cada descomposición recursiva.

(R.8) **Función de combinación**. Lo único que nos resta por obtener del caso recursivo es la función de combinación, ya que hay un "elemento indispensable" que suponemos incluido: la llamada o llamadas recursivas, pasando como datos las descomposiciones recursivas. En el caso de recursión simple, esta acción se puede derivar de la especificación:

$$\{\ P_0 \wedge \neg d(\vec{x}) \wedge Q_0\ [\vec{x}/s(\vec{x}),\ \vec{y}/\vec{y}']\ [\vec{X}/\vec{X}']\ \}$$
$$A_2$$
$$\{\ Q_0\ \}$$

Para ajustarnos al esquema de las funciones recursivas, intentaremos que A_2 sea de la forma:

$$\vec{y} := c(\vec{x},\ \vec{y}')$$

Con lo que la demostración de la corrección de A_2 es en realidad la comprobación de que se verifica el requisito (r.5)

$$P_0 \wedge \neg d(\vec{x}) \wedge Q_0\ [\vec{x}/s(\vec{x}),\ \vec{y}/\vec{y}']\ [\vec{X}/\vec{X}'] \Rightarrow def(c(\vec{x},\vec{y}')) \wedge Q_0\ [\vec{y}/c(\vec{x},\vec{y}')]$$

Si hubiese más de un caso recursiva habría que derivar una función de composición para cada uno de ellos.

Si tenemos varias llamadas recursivas –recursión múltiple– deberemos derivar entonces la acción A_1 a partir de la especificación:

$$\{\ P_0 \wedge \neg d(\vec{x}) \wedge Q_0\ [\vec{x}/s_1(\vec{x}),\ \vec{y}/\vec{y}_1]\ [\vec{X}/\vec{X}_1] \wedge \ldots \wedge Q_0\ [\vec{x}/s_k(\vec{x}),\ \vec{y}/\vec{y}_k]\ [\vec{X}/\vec{X}_k]\ \}$$
$$A_2$$
$$\{\ Q_0\ \}$$

donde las variables y_j representan a los resultados de las llamadas recursivas, para j=1,…, k

(R.9) **Escritura del caso recursivo**. Lo último que nos queda por decidir es si escribimos la solución del caso recursivo como una composición secuencial

$\{ P_0 \wedge \neg d(\vec{x}) \}$
$\vec{y}\text{'} := nombreFunc(s(\vec{x}));$
$\{ P_0 \wedge \neg d(\vec{x}) \wedge Q_0\ [\vec{x}/s(\vec{x}),\ \vec{y}/\vec{y}\text{'}]\ [\vec{X}/\vec{X}\text{'}] \}$
$\vec{y} := c(\vec{x}, \vec{y}\text{'})$
$\{ Q_0 \}$

cuya corrección está demostrada por el paso anterior (la llamada es correcta por la forma de la postcondición y el aserto intermedio es más fuerte que la pmd de la segunda asignación inducida por la postcondición).

O si podemos incluir la llamada o llamadas recursivas dentro de la expresión que las combina:

$\{ P_0 \wedge \neg d(\vec{x}) \}$
$\vec{y} := c(\vec{x}, nombreFunc(s(\vec{x})))$
$\{ Q_0 \}$

que también es correcta, porque para verificarla la convertiríamos a la forma anterior, sobre la que ya hemos razonado su corrección.

Este esquema se generaliza de manera inmediata para el caso de múltiples llamadas recursivas.

Veamos un primer ejemplo de aplicación del método de derivación. Vamos a obtener una función que dado un natural *n* calcule la suma de los dobles de los naturales hasta *n*:

$\sum i : 0 \le i \le n : 2*i$

La especificación:

```
func sumaDoble ( n : Nat ) dev s : Nat;
{ P0 : n = N }
{ Q0 : n = N ∧ s = ∑ i : 0 ≤ i ≤ n }
```

(R.1) Planteamiento recursivo.

La idea recursiva es obtener el resultado como:

```
sumaDoble(n) := sumaDoble(n-1) + 2*n
```

(R.2) Análisis de casos

El caso más simple es cuando n = 0, en cuyo caso el resultado es 0; por lo tanto las condiciones que distinguen los casos:

$d(n) : n = 0$
$\neg d(n) : n \neq 0$

demostramos la corrección

$n = N \Rightarrow def(n=0) \wedge def(n\neq 0) \Leftrightarrow cierto$
$n = N \Rightarrow n = 0 \vee n \neq 0 \Leftrightarrow cierto$

(R.3) Solución en el caso directo

Hemos de derivar una acción para

$$\{\ n = N \wedge n = 0\ \}$$
$$A_1$$
$$\{\ n = N \wedge s = \sum i : 0 \leq i \leq n : 2 * i\ \}$$

Considerando que n=0 es evidente que alcanzamos la postcondición con la asignación:

$$A_1\text{: } s := 0$$

(R.4) Descomposición recursiva

La descomposición recursiva ya aparecía en el planteamiento recursivo, obtenemos la suma de *n* a partir de la suma hasta n–1**:**

$$s(n) = n-1$$

(R.5) Función de acotación

El "tamaño del problema" viene dado por el valor de *n*, que ha de disminuir en las sucesivas llamadas recursivas hasta llegar a 0:

$$t(n) = n$$

Probamos los requisitos

$$n = N \wedge n \neq 0 \Rightarrow def(n) \wedge dec(n)$$

$$\begin{array}{ll} & dec(n) \\ \Leftrightarrow & n > 0 \\ \Leftarrow & n \neq 0 \wedge n : Nat \\ \Leftrightarrow & n \neq 0 \wedge n = N \end{array}$$

(R.6) Decremento de la función de acotación

Efectivamente la descomposición recursiva elegida hace que disminuya la función de acotación:

$$n = N \wedge n \neq 0 \Rightarrow n-1 < n \Leftrightarrow cierto$$

(R.7) Es posible hacer la llamada recursiva

Hemos de demostrar

$$n = N \wedge n \neq 0 \Rightarrow P_0[n/n-1][N/N']$$

$$P_0[n/n-1][N/N']$$

$$
\begin{array}{ll}
\Leftrightarrow & n - 1 = N' \\
\Leftrightarrow & n - 1 : Nat \\
\Leftarrow & n : Nat \land n \geq 1 \\
\Leftrightarrow & n : Nat \land n \neq 0
\end{array}
$$

(R.8) Función de combinación

Esta función aparecía ya en el planteamiento recursivo: la suma del resultado recursivo y 2*n. Podemos ver que se deriva de la especificación:

$$
\begin{array}{l}
\{\ P_0 \land Q_0\ [n/n-1,\ s/s'][N/N']\ :\ n=N \land n \neq 0 \land n-1=N' \land s'=\sum i : 1 \leq i \leq n-1 : 2*i\ \} \\
\quad A_2 \\
\{\ Q_0 : n = N \land s = \sum i : 1 \leq i \leq n : 2*i\ \}
\end{array}
$$

Observamos que la diferencia entre pre y postcondición se puede salvar con la asignación:

```
s := s' + 2*n
```

Verificamos que efectivamente esta sentencia verifica la especificación, pues la precondición es más fuerte que la pmd de la asignación inducida por Q_0:

$$
\begin{array}{ll}
 & def(s'+2*n) \land Q_0\ [s/s'+2*n] \\
\Leftrightarrow & cierto \land n = N \land s' + 2*n = \sum i : 1 \leq i \leq n : 2*i \\
\Leftrightarrow & n = N \land s' + 2*n = \sum i : 1 \leq i \leq n-1 : 2*i + 2*n \\
\Leftrightarrow & n = N \land s' = \sum i : 1 \leq i \leq n-1 : 2*i \\
\Leftarrow & P_0 \land n \neq 0 \land Q_0[n/n-1,\ s/s'][N/N']
\end{array}
$$

(R.9) Escritura del caso recursivo

Con lo anterior tenemos demostrado que es correcta la siguiente solución para el caso recursivo:

$$
\begin{array}{l}
\{\ P_0 \land n \neq 0\ \} \\
\quad s' := sumaDob(n-1); \\
\{\ P_0 \land n \neq 0 \land Q_0[n/n-1,\ s/s'][N/N']\ \} \\
\quad s := s' + 2*n \\
\{\ Q_0\ \}
\end{array}
$$

Nos resta plantearnos si es posible fundir la llamada recursiva en la función de combinación. En este caso sí es posible ya que la función devuelve un único resultado que se utiliza en la combinación:

$$
\begin{array}{l}
\{\ P_0 \land n \neq 0\ \} \\
\quad s := sumaDob(n-1) + 2*n \\
\{\ Q_0\ \}
\end{array}
$$

Con todo esto la función derivada queda:

```
func sumaDoble ( n : nat ) dev s : Nat;
{ P₀ : n = N }
inicio
  si
    n = 0 →
    { P₀ ∧ n = 0 }
      s := 0
    { Q₀ }
  □ n ≠ 0 →
    { P₀ ∧ n ≠ 0 }
      s := sumaDoble( n−1 ) + 2*n
    { Q₀ }
  fsi
{ Q₀ : n = N ∧ s = ∑ i : 1 ≤ i ≤ n : 2*i }
  dev s
ffunc
```

Ejemplo: suma de las componentes de un vector de enteros

En ocasiones es necesario *generalizar*, incluyendo parámetros adicionales, la función que queremos obtener para descubrir así un planteamiento recursivo. Esto es lo que ocurre en este caso. Nuestra primera idea de especificación para este problema:

```
func sumaVec ( v : Vector[1..N] de Ent ) dev s : Ent;
{ P₀ : v = V ∧ N ≥ 1 }
{ Q₀ : v = V ∧ s = ∑ i : 1 ≤ i ≤ N : v(i) }
```

(R.1) Planteamiento recursivo

el problema es que el planteamiento recursivo evidente es:

"para obtener recursivamente la suma de las *n* componentes de un vector, sumamos la primera componente a la suma del resto"

Pero para poder hacer una descomposición como esa es necesario que la función sumaVec nos permita especificar hasta qué componente queremos calcular la suma. Por lo tanto para poder llevar a la práctica este planteamiento recursivo debemos modificar la especificación de la función:

```
func sumaVec ( v : Vector[1..N] de Ent; a : Ent ) dev s : Ent;
{ P₀ : v = V ∧ N ≥ 1 ∧ b = B ∧ 1 ≤ b ≤ N }
{ Q₀ : v = V ∧ b = B ∧ s = ∑ i : a ≤ i ≤ N : v(i) }
```

Pero, ya que estamos, y pensando en darle más utilidad a nuestra función podemos generalizar por los dos lados y derivar una función que pueda obtener la suma de un subvector cualquiera:

```
func sumaVec ( v : Vector[1..N] de Ent; a, b : Ent ) dev s : Ent;
{ P₀ : v = V ∧ N ≥ 1 ∧ b = B ∧ a = A ∧ 1 ≤ a ≤ b ≤ N }
{ Q₀ : v = V ∧ b = B ∧ s = ∑ i : a ≤ i ≤ b : v(i) }
```

Cuando necesitamos modificar la especificación para llegar al planteamiento recursivo debemos indicar entonces cuál es la llamada inicial, es decir, qué valor se debe asignar a los parámetros adicionales de entrada para resolver el problema original. En este caso

```
sumaVec( v, 1, N )
```

Para facilitarle la vida a los usuarios de nuestra función podemos plantearnos la posibilidad de implementar una función con la especificación original y otra función auxiliar con la especificación generalizada. De esta forma el cuerpo de la "función principal" consistirá únicamente en la invocación inicial a la función auxiliar.

Con todo esto el planteamiento recursivo queda:

```
sumaVec( v, a, b) = v(a) + sumaVec( v, a+1, b )
```

(R.2) Análisis de casos

El caso más simple, caso directo, se tiene cuando la longitud del subvector es 1: a = b. En ese caso tenemos que la suma es directamente el valor de la componente v(a).[3]

```
d(v,a,b) : a = b
¬d(v,a,b) : a ≠ b

P₀ ⇒ def(a=b) ∧ def(a≠b)
P₀ ⇒ a = b ∨ a ≠ b
```

(R.3) Solución en el caso directo

Se deriva de la especificación

```
{ P₀ ∧ a = b }
  A₁
{ a = A ∧ b = B ∧ v = V ∧ s = ∑ i : a ≤ i ≤ b : v(i) }

A₁ : s := v(a)
```

[3] Estamos incluyendo en la precondición las restricciones que nos interesan y no nos preocupamos por garantizarlas. En esa misma línea, en este ejemplo, no nos preocupamos por el caso b < a, para el que podríamos devolver 0. Si considerásemos ese caso tendríamos una implementación más robusta.

(R.4) Descomposición recursiva

Utilizando la idea del planteamiento recursivo

$$s(v,a,b) = \langle v, a+1, b \rangle$$

(R.5) Función de acotación

La idea es que la longitud del subvector va a ir disminuyendo

$$t(v,a,b) = b-a$$

Se demuestra

$$P_0 \wedge a \neq b \Rightarrow def(b-a) \wedge dec(b-a)$$

$$\begin{aligned} & dec(b-a) \\ \Leftrightarrow\ & b-a > 0 \\ \Leftrightarrow\ & b > a \\ \Leftarrow\ & b \geq a \wedge a \neq b \Leftarrow P_0 \wedge a \neq b \end{aligned}$$

(R.6) Avance

Demostramos

$$P_0 \wedge a \neq b \Rightarrow b-(a+1) < b-a \Leftrightarrow b-a-1 < b-a \Leftrightarrow cierto$$

(R.7) Llamada recursiva

Demostramos

$$P_0 \wedge a \neq b \Rightarrow P_0[v/v,\ a/a+1,\ b/b][V/V',\ A/A',\ B/B']$$

$$\begin{aligned} & P_0[v/v,\ a/a+1,\ b/b][V/V',\ A/A',\ B/B'] \\ \Leftrightarrow\ & v = V' \wedge a+1 = A' \wedge b = B' \wedge 1 \leq a+1 \leq b \leq N \end{aligned}$$

por partes tenemos que

$v = V \wedge a = A \wedge b = B \Rightarrow v = V' \wedge a+1 = A' \wedge b = B'$

$a \leq b \wedge a \neq b \Rightarrow a+1 \leq b$

(R.8) Función de combinación

La función de combinación ya aparecía en el planteamiento recursivo, como la suma de la componente v(a) y el resultado de sumar v[a+1 .. b]. Pero podemos ver que se deduce de la especificación:

$$\{\ P_0 \wedge a \neq b \wedge Q_0[v/v,\ a/a+1,\ b/b,\ s/s'][V/V',\ A/A',\ B/B']\ \}$$
$$A_2$$
$$\{\ Q_0\ \}$$

o lo que es igual

```
{ P₀ ∧ a ≠ b ∧ v = V' ∧ a = A' ∧ b = B' ∧ s' = ∑ i : a+1 ≤ i ≤ b : v(i)
}
  A₂
{ v = V ∧ a = A ∧ b = B ∧ s = ∑ i : a ≤ i ≤ b : v(i) }
```

las diferencias se resuelven con la asignación:

```
A₂ : s := v(a) + s'
```

(R.9) Escritura del caso recursivo

¿Es posible incluir la llamada recursiva en la expresión de combinación?

```
{ P₀ ∧ a ≠ b }
  s := v(a) + sumaVec(v, a+1, b)
{ Q₀ }
```

De forma que la función resultante:

```
func sumaVec ( v : Vector[1..N] de Ent; a, b : Ent ) dev s : Ent;
{ P₀ : v = V ∧ N ≥ 1 ∧ b = B ∧ a = A ∧ 1 ≤ a ≤ b ≤ N }
  si
    a = b → s := v(a)
  □ a ≠ b → s := v(a) + sumaVec(v, a+1, b)
  fsi
{ Q₀ : v = V ∧ b = B ∧ s = ∑ i : a ≤ i ≤ b : v(i) }
  dev s
ffunc
```

Implementación recursiva de la búsqueda dicotómica (o binaria)

Queremos derivar una función recursiva que encuentre la aparición más a la derecha de un valor *x* dentro de un vector *v*. De no encontrarse, queremos que nos indique la posición donde se debería insertar. En esta función hay que ser cuidadoso con el tratamiento de los índices y tener en cuenta la posibilidad de que no esté en el vector, o que sea mayor que todos los elementos del vector o menor que todos ellos.

Vamos a utilizar la misma idea de la implementación iterativa: comparamos el elemento buscado con el elemento central del –sub–vector, y según el resultado de la comparación seguimos buscando en el subvector de la izquierda o en el de la derecha. De nuevo el planteamiento recursivo nos obliga a generalizar la función a obtener, para incluir como parámetros los límites del subvector a considerar.

En la implementación iterativa aparecían dos variables, *pos* y *q*, que servían para ir acotando el subvector a considerar; variables que íbamos modificando hasta llegar a la condición pos=q+1. Lo que ocurre es que estas variables no indican *exactamente* dónde empieza y dónde acaba el subvector a considerar, sino que indican a partir de qué posición –pos– tenemos componentes menores o iguales que *x* y partir de qué posición –q– tenemos componentes mayores que *x*. Por eso

la terminación del bucle ocurre cuando esas posiciones son consecutivas, y también es por ello por lo que no desplazamos *pos* a la derecha de *m* o *q* a la izquierda de *m*, sino que las desplazamos a *m*. De acuerdo con estas ideas, la inicialización del bucle asigna <pos,q>:=<0,N+1> que es la única forma de no hacer ninguna suposición sobre dónde está *x*.

Para implementar la idea recursiva vamos a introducir dos variables adicionales que indiquen dónde empieza y dónde acaba el vector a considerar, que es una idea ligeramente distinta de la utilizada en el algoritmo iterativo (no es que aquí no podamos utilizar esta misma idea, pero resulta más natural la que proponemos). De estar en algún sitio la componente buscada estará en el intervalo v[a..b], modificaremos los límites de forme que a la izquierda de *a* estén valores menores o iguales que el buscado y la derecha de *b* valores mayores que el buscado, pero sin presuponer nada sobre los valores de v[a..b], a diferencia del otro planteamiento donde v(a)≤x y v(b)>x, cosa que no podemos garantizar ahora si restringimos *a* y *b* a los límites válidos del vector, dentro de los cuales no sabemos si hay algún valor mayor que el buscado o algún valor menor o igual.

Si *x* no se encuentra en el vector queremos que *pos* se quede apuntando a la posición anterior a la que debería ocupar *x*.

Con todo esto la especificación queda:

```
func buscaBin( v : Vector[1..N] de Elem; x : Elem ; a, b : Ent ) dev p :
Ent;
{ P₀ : 1 ≤ a ≤ b ≤ N ∧ ord(v,a,b) }
{ Q₀ : a-1 ≤ p ≤ b ∧ v[a..p] ≤ x < v[(p+1)..b] }
```

donde:

— hemos obviado los asertos sobre la conservación de los parámetros de entrada (demasiado laborioso)

— El aserto ord(v,a,b), que ya apareció en un ejercicio indica que el vector entre las posiciones *a* y *b* está ordenado:

```
ord(v,a,b) ⇔ ∀ i, j : a ≤ i < j ≤ b : v(i) ≤ v(j)
```

Podríamos exigir simplemente ord(v), pues de hecho se cumple, pero esta otra condición es más coherente con la cabecera de la función.

— Hemos introducido una nueva notación para escribir condiciones que se cumplen en un subconjunto contiguo de un vector. Así

```
v[a..p] ≤ x
```

abrevia

```
∀ i : a ≤ i ≤ p : v(i) ≤ x
```

y

```
x < v[(p+1)..b]
```

abrevia

```
∀ i : p+1 ≤ i ≤ b : x < v(i)
```

— Los casos extremos para el valor de *pos* son

```
p = a-1 ⇒ x < v[a..b] % x no está y es menor que todas las componentes
```

```
p = b   ⇒ v[a..b] ≤ x  % v(b)=x ó x es mayor que todas las componentes
```

(R.1) Planteamiento recursivo

Tomamos el punto intermedio *m* entre *a* y *b*, si *x* es menor que v(m) entonces seguimos buscando en [a .. m–1]; si *x* es mayor que v(m) entonces seguimos buscando en [m+1 .. b] y si *x* es igual a v(m) entonces seguimos buscando en [m+1 .. b]. Se podría pensar que en este último caso es más razonable seguir buscando en [m .. b], pero esto nos puede llevar a un bucle infinito pues puede ocurrir que a=m, con lo que no decrementaríamos la longitud del subvector a considerar. Esto lleva a que el valor buscado se pueda quedar a la izquierda del subvector considerado, pero aún así el resultado será correcto. (podemos discutir este detalle al obtener la descomposición recursiva).

(R.2) Análisis de casos

Vamos a considerar como caso directo el caso en el que tenemos un subvector de longitud 1, es decir:

```
 d(v,x,a,b) : a = b
¬d(v,x,a,b) : a ≠ b
```

demostramos

```
P₀ ⇒ def(a=b) ∧ def(a≠b)
P₀ ⇒ a = b ∨ a ≠ b
```

(R.3) Solución en el caso directo.

Parece claro que tenemos que hacer una distinción de casos según el resultado de comparar *x* con la única componente del vector.

Las condiciones que discriminan los distintos tratamientos son

```
v(a) = x     v(a) < x     v(a) > x
```

Tenemos que las barreras están definidas:

```
P₀ ∧ a = b ⇒ def(v(a)=x) ∧ def(v(a)<x) ∧ def(v(a)>x)
```

puesto que

```
P₀ ⇒ enRango(v,a)
```

y al menos una se abre:

```
P₀ ∧ a = b ⇒ v(a) = x ∨ v(a) < x ∨ v(a) > x ⇔ cierto
```

derivamos entonces el código de cada una de las ramas:

```
{ P₀ ∧ a = b ∧ v(a)=x }
```

```
  A1
{ v[a..p] ≤ x < v[(p+1)..b] }

A1 : p := a

{ P0 ∧ a = b ∧ v(a)<x }
  A2
{ v[a..p] ≤ x < v[(p+1)..b] }

A2 : p := a

{ P0 ∧ a = b ∧ v(a)>x }
  A3
{ v[a..p] ≤ x < v[(p+1)..b] }

A3 : p := a - 1
```

con lo que la solución del caso directo queda

```
si
  v(a) ≤ x → p := a
□ v(a) > x → p := a-1
fsi
```

también podemos intentar obtenerla razonando sobre la postcondición, añadiéndole a=b. Podemos añadir esta condición porque sabemos que está en la precondición de la acción que queremos derivar, y sabemos que dicha acción no afectará a sus valores, por lo que debe cumplirse también en el estado final: (Esta es la primera vez que utilizamos un razonamiento de este tipo)

```
                    a-1 ≤ p ≤ b ∧ v[a..p] ≤ x < v[(p+1)..a] ∧ a = b
                ⇔   a-1 ≤ p ≤ a ∧ v[a..p] ≤ x < v[(p+1)..a] ∧ a = b
```

p puede ser *a* ó a-1

```
⇔ (a = p ∧ v[a..a] ≤ x < v[(a+1)..a] ∧ a = b) ∨
   (a-1 = p ∧ v[a..(a-1)] ≤ x < v[a..a] ∧ a = b)
⇔ (a = p ∧ v(a) ≤ x ∧ a = b) ∨ (a-1 = p ∧ x < v(a) ∧ a = b)
```

y de aquí obtenemos la solución para el caso base.

(R.4) Descomposición recursiva

La descomposición recursiva toma la forma de una selección alternativa. Obtenemos el punto intermedio y según el resultado de comparar ese punto intermedio con el valor de *x* descomponemos la llamada de una u otra forma. Podríamos haberlo derivado descomponiendo el caso recursivo en más de un caso, pero para no repetir código –el cálculo de *m*– lo hacemos de esta otra forma.

```
m := (a + b) div 2;
```

tenemos que

$$a \le b \land a \neq b \Rightarrow a < b \Rightarrow a \le m < b$$

vamos a añadir esta condición a todas las demostraciones pues partimos de un estado en el que se cumple esto, después de haber hecho la asignación a *m*

$$\{ P_0 \land \neg d(\vec{x}) \}$$
$$m := (a+b) \text{ div } 2$$
$$\{ P_0 \land \neg d(\vec{x}) \land a \le m < b \}$$
$$A_4$$
$$\{ Q_0 \}$$

la parte interesante de la postcondición es

$$v[a..p] \le x < v[(p+1)..b]$$

Vamos a derivar A_4 como una composición alternativa, según el resultado de la comparación

Si **x < v(m)** quiere decir que *m* está entre p+1 y *b*, por lo tanto podemos acotar la búsqueda con la descomposición:

$$s_1(v,x,a,b) = \langle v, x, a, m-1 \rangle$$

Por otra parte, si **x ≥ v(m)** quiere decir que *m* está entre *a* y *p* y que podemos acotar la búsqueda con la descomposición:

$$s_2(v,x,a,b) = \langle v, x, m+1, b \rangle$$

Podríamos pensar en distinguir el caso x = v(m), haciendo a = m. Pero esto puede llevar a una cadena infinita de llamadas si a=b–1 ∧ v(a)=x

(R.5) Función de acotación

Lo que va a ir disminuyendo según avanza la recursión es la longitud del subvector a considerar, por lo tanto tomamos como función de acotación:

$$t(v,x,a,b) = b-a$$

y demostramos que está definida y es decrementable

$$P_0 \land a \neq b \land a \le m < b \Rightarrow def(b-a) \land dec(b-a)$$

$$dec(b-a)$$

$$\begin{aligned} &\Leftrightarrow\ b-a > 0\\ &\Leftrightarrow\ b > a\\ &\Leftarrow\ a \le b \land a \neq b\\ &\Leftarrow\ P_0 \land a \neq b \end{aligned}$$

(R.6) Terminación

Tenemos que demostrar

$$P_0 \land a \neq b \land a \le m < b \Rightarrow t(s_1(\vec{x})) < t(\vec{x}) \land t(s_2(\vec{x})) < t(\vec{x})$$

$$\begin{aligned} &\quad (m-1) - a < b - a\\ &\Leftarrow\ m-1 < b\\ &\Leftarrow\ m < b \end{aligned}$$

$$\begin{aligned} &\quad b - (m+1) < b - a\\ &\Leftrightarrow\ b - m - 1 < b - a\\ &\Leftarrow\ b - m \le b - a\\ &\Leftrightarrow\ m \ge a \end{aligned}$$

(R.7) Llamada recursiva

Tenemos que demostrar:

$$P_0 \land a \neq b \land a \le m < b \Rightarrow P_0[\vec{x}/s_1(\vec{x})] \land P_0[\vec{x}/s_2(\vec{x})]$$

La precondición es $1 \le a \le b \le N \land ord(v,a,b)$. El requisito de la ordenación se mantiene porque el vector entero está ordenado lo que debemos comprobar es que se conserva la otra condición:

$$\begin{aligned} &\quad 1 \le a \le b \le N\ [a/a,\ b/m-1]\\ &\Leftrightarrow\ 1 \le a \le m-1 \le N\\ &¿\Leftarrow 1 \le a \le b \le N \land a \neq b \land a \le m < b? \end{aligned}$$

No es posible demostrarlo, de $a \le m$ no se puede obtener $a \le m-1$. El problema es que se puede pasar de un subvector de longitud 2 a un subvector de longitud 0:

$$\begin{aligned} &a = 1,\ b = 2,\ v(1) = 3,\ v(2) = 4,\ x = 2\\ &m = 1\ ,\ v(m) > x \Rightarrow b = a-1 \end{aligned}$$

No siempre se llega al caso base a = b, sino que también se puede llegar al caso base a = b+1, eso implica que *b* puede tomar el valor 0 y que *a* puede tomar el valor N+1. Por lo tanto hay que hacer varios cambios a lo ya obtenido:

— Hay que cambiar la especificación para que la precondición incluya:

$$1 \le a \le b+1 \le N+1 \Leftrightarrow 1 \le a \le N+1 \land 0 \le b \le N \land a \le b+1$$

— Hay que cambiar las condiciones de los casos:

$$\begin{aligned} &d_1(\vec{x})\ :\ a = b+1\\ &d_2(\vec{x})\ :\ a = b \end{aligned}$$

$d_3(\vec{x})$: a < b

— En los razonamientos sobre la terminación (R.5 y R.6) no hay que cambiar nada porque estábamos razonando a partir de $a \leq b \wedge a \neq b$ que es equivalente a razonar con a < b.

— Tampoco hay que cambiar nada en la descomposición recursiva

Ahora sí es posible demostrar:

$P_0 \wedge a \neq b \wedge a \leq m < b \Rightarrow P_0[\vec{x}/s_1(\vec{x})] \wedge P_0[\vec{x}/s_2(\vec{x})]$

$$
\begin{array}{ll}
 & 1 \leq a \leq b+1 \leq N+1 \ [a/a,\ b/m-1] \\
\Leftrightarrow & 1 \leq a \leq m-1+1 \leq N+1 \\
\Leftrightarrow & 1 \leq a \leq m \leq N+1 \\
\Leftarrow & 1 \leq a \leq b+1 \leq N+1 \wedge a \neq b \wedge a \leq m < b
\end{array}
$$

$$
\begin{array}{ll}
 & 1 \leq a \leq b+1 \leq N+1 \ [a/m+1,\ b/b] \\
\Leftrightarrow & 1 \leq m+1 \leq b+1 \leq N+1 \\
\Leftrightarrow & 0 \leq m \leq b \leq N \\
\Leftarrow & 1 \leq a \leq b+1 \leq N+1 \wedge a \neq b \wedge a \leq m < b
\end{array}
$$

$(R.3)_2$ Solución al segundo caso base

Hemos de derivar una acción a partir de:

$\{ P_0 \wedge a = b+1 \}$
  A_5
$\{ Q_0 \}$

Razonamos a partir de la postcondición y a = b+1

$$
\begin{array}{ll}
 & a-1 \leq p \leq b \wedge v[a..p] \leq v(x) < v[(p+1)..b] \wedge a = b+1 \\
\Leftrightarrow & a-1 \leq p \leq a-1 \wedge v[a..(a-1)] \leq v(x) < v[a..(a-1)] \wedge a = b+1 \\
\Leftrightarrow & a-1 = p \wedge a = b+1
\end{array}
$$

Intuitivamente podemos ver que este caso puede darse cuando v(a)>x y por lo tanto *p* debe apuntar a la posición anterior (para así apuntar a la posición anterior del lugar de inserción); o, si no admitiésemos como caso base a=b, porque v(b)≤x con lo que *p* debe apuntar a *b*, que es precisamente *a*–1.

(R.8) Función de combinación

En los dos casos de descomposición recursiva nos limitamos a propagar el resultado de la llamada recursiva:

p' := nombreFunc(s($\vec{x}$)); p := p'

$P_0 \wedge a \leq m < b \wedge a < b \wedge v(m) \leq x \wedge m+1-1 \leq p' \leq b \wedge v[(m+1)..p'] \leq x < v[(p'+1)..b]$

⇒

$a-1 \leq p' \leq b \wedge v[a..p'] \leq x < v[(p'+1)..b]$

Se cumple lo anterior porque en P_0 se incluye ord(v, a, b), por lo que v[a..m] ≤ x

De la misma forma, para la otra llamada recursiva

$P_0 \wedge a \leq m < b \wedge a < b \wedge v(m) > x \wedge a-1 \leq p' \leq m-1 \wedge v[a..p'] \leq x < v[(p'+1)..(m-1)]$

⇒

$a-1 \leq p' \leq b \wedge v[a..p'] \leq x < v[(p'+1)..b]$

También se tiene por ord(v,a,b) y por lo tanto x < v[m..b]

(R.9) Escritura de la llamada recursiva

Con todo lo anterior la solución al caso recursivo queda:

```
m := (a+b) div 2;
si
    v(m) ≤ x → p := buscaBin( v, x, m+1, b )
  □ v(m) > x → p := buscaBin( v, x, a, m-1 )
fsi
```

Y finalmente la función completa:

```
func buscaBin( v : Vector[1..N] de Elem; x : Elem ; a, b : Ent ) dev p : Ent;
{ P₀ : 1 ≤ a ≤ b ≤ N ∧ ord(v,a,b) }
var
  m : Ent;
inicio
  si
    a = b+1 → p := a - 1
  □ a = b   → si
                  v(a) ≤ x → p := a
                  v(a) > x → p := a-1
              fsi
  □ a < b   → m := (a+b) div 2;
              si
                  v(m) ≤ x → p := buscaBin( v, x, m+1, b )
                □ v(m) > x → p := buscaBin( v, x, a, m-1 )
              fsi
  fsi
{ Q₀ : a-1 ≤ p ≤ b ∧ v[a..p] ≤ x < v[(p+1)..b] }
  dev p
ffunc
```

Y no se vayan todavía que aún hay más: analizando este programa con cuidado nos damos cuenta de que el caso base a = b es redundante y lo podemos eliminar.

1.3.1 Algoritmos avanzados de ordenación

Vamos a estudiar dos algoritmos de ordenación de complejidad cuasi-lineal, $O(n \log n)$: la ordenación rápida y la ordenación por mezcla. Debido a la complejidad de estos algoritmos y con el ánimo de terminar algún día la asignatura, relajaremos los requisitos formales para la obtención de los algoritmos.

Vamos a diseñar ambos algoritmos como procedimientos para evitar así el coste que supone realizar una copia del vector a ordenar (aunque en la ordenación por mezcla se utiliza un vector auxiliar) y considerando así mismo que la mayoría de los lenguajes de programación imperativa no permiten que las funciones devuelvan tipos estructurados.

Ordenación rápida

Ideado por C. A. R. Hoare en 1962.

La especificación de la que partimos:

```
proc quickSort( es v : Vector[1..N] de Elem; e a, b : Ent );
{ P₀ : v = V ∧ a = A ∧ b = B ∧ 1 ≤ a ≤ b + 1 ≤ N+1 }
{ Q₀ : a = A ∧ b = B ∧ perm(v, V, a, b) ∧ ord(v, a, b) ∧
       (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀j : b < j ≤ N : v(j) = V(j)) }
fproc
```

Como en el caso de la búsqueda binaria la necesidad de encontrar un planteamiento recursivo nos lleva a añadir dos parámetros adicionales al procedimiento, para así poder indicar el subvector de que nos ocupamos en cada llamada recursiva.

El objetivo de esta función es ordenar el subvector comprendido entre las posiciones *a* y *b*, dejando inalterado el resto del vector.

También de la misma forma que en la búsqueda dicotómica permitimos la llamada con un subvector de longitud 0

```
a = b + 1
```

Aparece una notación que no habíamos utilizado hasta ahora:

```
perm(v, V, a, b)
```

para indicar que *v* es una permutación de *V* entre las posiciones *a* y *b*; el aserto equivalente es el mismo que para perm(v, V), pero haciendo que las variables ligadas vayan entre *a* y *b*, en lugar de entre 1 y *N*.

El planteamiento recursivo consiste en colocar en su sitio el primer elemento del subvector v[a..b], dejando a su izquierdo los elementos menores o iguales y a su derecha los mayores o iguales, y ordenar recursivamente los dos fragmentos.

Análisis de casos

— a > b

Tenemos el caso directo cuando se trata de un segmento vacío.

$P_0 \wedge a > b \Rightarrow a = b+1$
$Q_0 \wedge a = b+1 \Rightarrow v = V$

— $a \leq b$

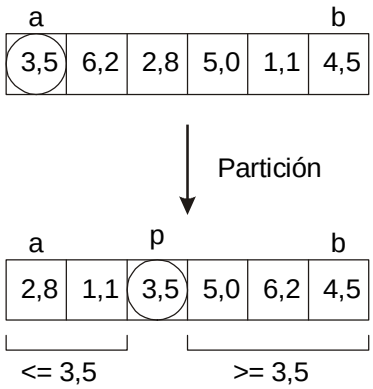
Se trata de un segmento no vacío y tenemos entonces el caso recursivo. Las ideas en las que se basa el caso recursivo son:

— considerar x = v(a) como *elemento pivote*

— reordenar parcialmente el subvector v[a..b] para conseguir que *x* quede en la posición *p* que ocupará cuando v[a..b] esté ordenado. Para ello tenemos que colocar a su izquierda los elementos menores o iguales que *x* y a su derecha los elementos mayores o iguales.

— ordenar recursivamente v[a..(p–1)] y v[(p+1)..b]. Este algoritmo ejemplifica la estrategia general de resolución de problemas *divide y vencerás*, puesto que resuelve el problema *dividiéndolo* en dos partes más pequeñas.

Por ejemplo:



Función de acotación

$t(v, a, b) = b - a + 1$

Hay que sumarle 1 porque también consideramos caso recursivo cuando a = b, y así evitamos que la función de acotación se haga 0.

Suponiendo que tenemos una implementación correcta de *partición*, sobre la que luego iremos, el algoritmo anotado nos queda:

```
proc quickSort( es v : Vector[1..N] de Elem; e a, b : Ent );
{ P₀ : v = V ∧ a = A ∧ b = B ∧ 1 ≤ a ≤ b + 1 ≤ N+1 }
var
  p : Ent;
inicio
  si
     a > b → seguir { Q₀ }
  □  a ≤ b → { P₀ ∧ a ≤ b }
                particion(v, a, b, p);
              { Q }
```

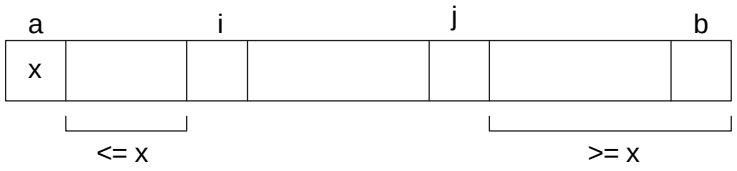
```
                { Q ∧ P₀[b/p-1] ∧ P₀[a/p+1]
                  quickSort(v, a, p-1);
                  quickSort(v, p+1, b);
                { Q ∧ Q₀[b/p-1] ∧ Q₀[a/p+1] }
                { Q₀ }
  fsi
{ Q₀ : a = A ∧ b = B ∧ perm(v, V, a, b) ∧ ord(v, a, b) ∧
       (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀j : b < j ≤ N : v(j) = V(j)) }
fproc
```

Donde Q es la postcondición de *partición*, y de la precondición la parte que nos interesa es:

```
P: 1 ≤ a ≤ b ≤ N  ⇐ P₀ ∧ a ≤ b
Q: 1 ≤ a ≤ p ≤ b ≤ N ∧ perm(v, V, a, b) ∧ v[a..(p-1)] ≤ v(p) ≤
v[(p+1)..b] ∧
   (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀i : b < i ≤ N : v(i) = V(i))
```

Diseño de partición

La idea es obtener un bucle que mantenga invariante la siguiente situación, de forma que *i* y *j* se vayan acercando hasta cruzarse, y finalmente intercambiemos v(a) y v(j)



— Invariante

```
I : a+1 ≤ i ≤ j+1 ≤ b+1 ∧ v[(a+1)..(i-1)] ≤ v(a) ≤ v[(j+1)..b]
```

también podríamos añadir

```
    ∧ perm(v, V, a, b) ∧
      (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀i : b < i ≤ N : v(i) = V(i))
```

aunque lo obviamos porque sólo vamos a modificar el vector con intercambios dentro del intervalo [a..b].

— Condición de repetición

```
¬B : i = j+1          % cuando i = j+1 hemos particionado todo el vector
B : i ≤ j
```

De esta forma al final del bucle se tiene:

```
I ∧ ¬B ⇒ I ∧ i = j+1 ⇒ v[(a+1)..j] ≤ v(a) ≤ v[i..b]
```

con lo que las acciones

```
p := j;
intercambiar(v, a, p);
```

ejecutadas a la salida del bucle harán que se alcance la postcondición

— Expresión de acotación

```
C : j – i + 1
```

— Acción de inicialización

```
<i, j> := <a+1, b>
```

Esta acción hace trivialmente cierto el invariante al hacer que los dos segmentos sean vacíos.

— Acción de avance

Habrá que hacer un análisis de casos comparando las componentes v(i) y v(j) con v(a)

```
v(i) ≤ v(a)                  incrementamos i
v(a) ≤ v(j)                  decrementamos j
v(i) > v(a) AND v(j) < v(a)  intercambiamos i con j y avanzamos ambas
```

Las dos primeras condiciones son no excluyentes y las tres condiciones juntas garantizan que al menos una de ellas se cumple.

Con todo esto el algoritmo queda:

```
proc particion (es v: Vector[1..N] de Elem; e a, b: Ent; s p : Ent)
{ P }
var
  i, j : Ent;
inicio
  <i, j> := <a+1, b>;
{ I ; C }
  it i ≤ j
  → si
       v(i) ≤ v(a) → i := i + 1
     □ v(j) ≥ v(a) → j := j – 1
     □ v(i) > v(a) AND v(j) < v(a) → intercambiar(v, i, j);
                                     <i, j> := <i+1, j–1>
     fsi
  fit
{ I ∧ i = j+1 }
  p := j;
  intercambiar(v, a, p)
```

```
{ Q }
fproc
```

Ordenación por mezcla

Por las mismas razones que en la ordenación rápida, decidimos diseñar este algoritmo como un procedimientos.

Partimos de una especificación similar a la del quickSort

```
proc mergeSort( es v : Vector[1..N] de Elem; e a, b : Ent );
{ P₀ : v = V ∧ a = A ∧ b = B ∧ 1 ≤ a ≤ b + 1 ≤ N+1 }
{ Q₀ : a = A ∧ b = B ∧ perm(v, V, a, b) ∧ ord(v, a, b) ∧
       (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀j : b < j ≤ N : v(j) = V(j)) }
fproc
```

Análisis de casos

— a ≥ b

Segmento de longitud ≤ 1.

Ya está ordenado

$P_0 \wedge a \geq b \Rightarrow a = b \vee a = b+1$

$Q_0 \wedge (a = b \vee a = b+1) \Rightarrow v = V$

En este caso no hay que hacer nada para garantizar la postcondición.

— a < b

Segmento de longitud ≥ 2.

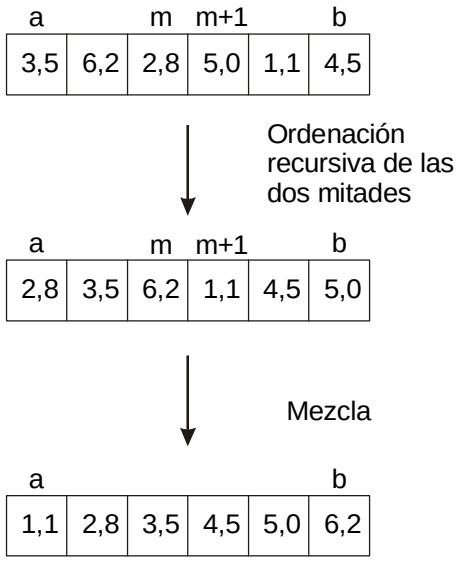
La idea del caso recursivo:

— Dividir v[a..b] en dos mitades. Al ser la longitud ≥ 2 es posible hacer la división (por eso hemos considerado como caso base el subvector de longitud 1). Cada una de las mitades tendrá una longitud estrictamente menor que el segmento original.

— Tomando m = (a+b) div 2 ordenamos recursivamente v[a..m] y v[(m+1)..b].

— Usamos un procedimientos auxiliar para mezclar las dos mitades, quedando ordenado todo v[a..b]

Ejemplo:



Función de acotación

```
t(v, a, b) = b – a + 1
```

Suponiendo que tenemos una implementación correcta para *mezcla*, el procedimiento de ordenación queda:

```
proc mergeSort( es v : Vector[1..N] de Elem; e a, b : Ent );
{ P₀ : v = V ∧ a = A ∧ b = B ∧ 1 ≤ a ≤ b + 1 ≤ N+1 }
var
  m : Ent;
inicio
  si
    a ≥ b → seguir { Q₀ }
  □ a < b → { P₀ ∧ a < b }
              m := (a+b) div 2;
            { P₀ ∧ a ≤ m < b }
            { P₀[b/m] ∧ P₀[a/m+1] }
              mergeSort( v, a, m );
              mergeSort( v, m+1, b );
            { a ≤ m < b ∧ Q₀[b/m] ∧ Q₀[a/m+1] }
              mezcla( v, a, m, b )
            { Q₀ }
  fsi
{ Q₀ : a = A ∧ b = B ∧ perm(v, V, a, b) ∧ ord(v, a, b) ∧
       (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀j : b < j ≤ N : v(j) = V(j)) }
fproc
```

Nos resta obtener el procedimiento que se encarga de mezclar los dos subvectores ordenados adyacentes.

Diseño de mezcla

El problema es que para conseguir una solución eficiente, $O(n)$, necesitamos utilizar un vector auxiliar donde iremos realizando la mezcla, para luego copiar el resultado al vector original.

La idea del algoritmo es colocarse al principio de cada segmento e ir tomando de uno u otro el menor elemento, y así ir avanzando. Uno de los subvectores se acabará primero y habrá entonces que copiar el resto del otro subvector. Finalmente copiaremos el resultado al vector original.

```
proc mezcla( es v : Vector[1..N] de Elem; e a, m, c : Ent );
{ P₀ : v = V ∧ a = A ∧ m = M ∧ c = C ∧ a ≤ m < b ∧ ord(v, a, m) ∧ ord(v,
m+1, b) }
var
  u : Vector[1..N] de Elem;
  i, j, k: Ent;
inicio
  u := v;
  <i, j, k> := <a, m+1, a>;
  it i ≤ m AND j ≤ b
  →  si
        u(i) ≤ u(j) →
           v(k) := v(i);
           i := i + 1;
      □ u(i) > u(j) →
           v(k) := u(j);
           j := j + 1;
      fsi;
      k := k + 1;
  it i ≤ m
  →  v(k) := u(i);
     <i,k> := <i+1,k+1>
  fit;
  it j ≤ b
  →  v(k) := u(j);
     <j,k> := <j+1, k+1>
  fit
{ Q₀ : a = A ∧ b = B ∧ m = M ∧ perm(V, v, a, b) ∧ ord(v, a, b) ∧
       (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀ i : b < i ≤ N : v(i) = V(i) }
```

2.4 Análisis de algoritmos recursivos

Como la recursividad no introduce nuevas instrucciones en el lenguaje algorítmico, en principio no es necesario incluir nuevos mecanismos para el cálculo de la complejidad de funciones recursivas. El problema es que cuando nos ponemos a analizar la complejidad de una función recursiva nos encontramos con que debemos conocer la complejidad de las llamadas recursivas,

es decir, necesitamos conocer la complejidad de la propia función que estamos analizando. El resultado es que la definición *natural* de la función de complejidad de una función recursiva suele ser también recursiva; a las funciones así definidas se las denomina *ecuaciones de recurrencia*. Veamos un par de ejemplos de análisis de funciones recursivas conocidas donde obtendremos las ecuaciones de recurrencia que definen su complejidad.

Cálculo del factorial

Tamaño de los datos: n

Caso directo: $T(n) = 3$ si $n = 1$

El 3 se obtiene porque hay que evaluar las dos barreras (recuérdese que en una composición alternativa se evalúan todas las barreras) y ejecutar la asignación.

Caso recursivo:

- 2 de evaluar ambas barreras
- 1 de la asignación de `n * fact(n-1)`
- 1 de evaluar la descomposición `n-1`
- T(n–1) de la llamada recursiva (el coste de la función con datos de tamaño n–1).

De esta forma las ecuaciones de recurrencia

$$T(n) = \begin{cases} 3 & \text{si } n = 1 \\ 4 + T(n-1) & \text{si } n > 1 \end{cases}$$

Método del campesino egipcio

Tamaño de los datos: $n = b$

Caso directo: $T(n) = 5$ si $n = 0, 1$

4 de evaluar todas las barreras y 1 de la correspondiente asignación

En ambos casos recursivos:

- 4 de evaluar ambas barreras
- 1 de la asignación
- 2 de evaluar la descomposición `2*a` y `b div 2`.
- T(n/2) de la llamada recursiva (el coste de la función con datos de tamaño b div 2).

De esta forma las ecuaciones de recurrencia

$$T(n) = \begin{cases} 5 & \text{si } n = 0, 1 \\ 7 + T(n/2) & \text{si } n > 1 \end{cases}$$

Las torres de Hanoi

Tamaño de los datos: *n*

Caso directo: $T(n) = 2$ si $n = 0$

2 de evaluar todas las barreras

En el caso recursivo:

— 2 de evaluar ambas barreras

— 1 del movimiento

— 2*T(n–1) de las dos llamadas recursiva

De esta forma las ecuaciones de recurrencia

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 3 + 2*T(n-1) & \text{si } n > 0 \end{cases}$$

Las recurrencias no nos dan información sobre el orden de complejidad, necesitamos una ecuación explícita.

Hay dos métodos para resolver las recurrencias:

— despliegue de recurrencias

— aplicar soluciones generales que se conocen para algunos tipos comunes de recurrencias

1.4.1 Despliegue de recurrencias

El proceso se compone de tres pasos:

— Despliegue. Sustituimos las apariciones de *T* en la recurrencia tantas veces como sea necesario hasta encontrar una fórmula en la que el número de despliegues dependa de un parámetro *k*.

— Postulado. A partir de la fórmula paramétrica resultado del paso anterior obtenemos una fórmula explícita. Para ello, obtenemos el valor de *k* que nos permite alcanzar un caso directo y sustituimos la referencia recursiva por la complejidad del caso directo. (Aquí estamos haciendo la suposición de que la recurrencia para el caso recursivo es también válida para el caso directo.)

— Demostración. Se demuestra por inducción que la fórmula obtenida cumple las ecuaciones de recurrencia. (Este paso lo vamos a obviar)

Apliquemos el método a los dos ejemplos que hemos presentado anteriormente: el factorial y la multiplicación por el método del campesino egipcio:

Factorial

$$T(n) = \begin{cases} 3 & \text{si } n = 1 \\ 4 + T(n-1) & \text{si } n > 1 \end{cases}$$

— Despliegue:

```
T(n) = 4 + T(n-1)
     = 4 + 4 + T(n-2)
     = 4 + 4 + 4 + T(n-3)
…
     = 4 · k + T(n-k)
```

— Postulado

El caso directo se tiene para $n = 1$; para alcanzarlo tenemos que hacer

```
k = n - 1

T(n) = 4 · (n-1) + T(n-(n-1))
     = 4 · n - 4 + T(1)
     = 4 · n - 4 + 3
     = 4 · n - 1
```

por lo tanto

```
T(n) ∈ O(n)
```

Campesino egipcio

$$T(n) = \begin{cases} 5 & \text{si } n = 0, 1 \\ 7 + T(n/2) & \text{si } n > 1 \end{cases}$$

— Despliegue:

```
T(n) = 7 + T(n/2)
     = 7 + 7 + T(n/2/2)
     = 7 + 7 + 7 + T(n/2/2/2)
…
     = 7 · k + T(n/2^k)
```

— Postulado

El caso directo se tiene para $n = 0,1$; para alcanzar $n=1$ tenemos que hacer

```
k = log n          % como siempre log en base 2

T(n) = 7 · log n + T(n/2^(log n))
     = 7 · log n + T(1)
```

$$= 7 \cdot \log n + 5$$

por lo tanto

$$T(n) \in O(\log n)$$

(Nótese que la igualdad $k = \log n$ sólo tiene sentido cuando *n* es una potencia de 2 y por lo tanto tiene un logaritmo exacto, ya que el número de pasos *k* tiene que ser un número entero. Por lo tanto el anterior resultado sólo sería válido para ciertos valores de *n*, sin embargo esto no es una limitación grave, porque la complejidad T(n) del algoritmo es una función monótona creciente de *n*, y por lo tanto nos basta con estudiar su comportamiento sólo en algunos puntos).

Nótese que la función de complejidad que hemos obtenido no es válida para n = 0, ya que no está definido el logaritmo de 0; sin embargo, esto no es importante ya que nos interesa el comportamiento asintótico de la función.

Torres de Hanoi

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 3 + 2*T(n-1) & \text{si } n > 0 \end{cases}$$

— Despliegue:

$$\begin{aligned} T(n) &= 3 + 2*T(n-1) \\ &= 3 + 2 * (3 + 2 * T(n-2)) \\ &= 3 + 2 * (3 + 2 * (3 + 2 * T(n-3))) \\ &= 3 + 2 * 3 + 2^2 * 3 + 2^3 * T(n-3) \\ &\dots \\ &= 3 \cdot \sum_{i=0}^{k-1} 2^i + 2^k \, T(n-k) \end{aligned}$$

— Postulado

El caso directo se tiene para $n = 0$; para alcanzarlo

$$k = n$$

$$\begin{aligned} T(n) &= 3 \cdot \sum_{i=0}^{k-1} 2^i + 2^k \, T(n-k) \\ &= 3 \cdot \sum_{i=0}^{n-1} 2^i + 2^n \, T(n-n) \\ (*) \quad &= 3 \cdot \sum_{i=0}^{n-1} 2^i + 2^n \, T(0) \\ &= 3 \cdot (2^n - 1) + 2 \cdot 2^n \\ &= 5 \cdot 2^n - 3 \end{aligned}$$

donde en (*) hemos utilizado la fórmula para la suma de progresiones geométricas:

$$\sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1} \qquad r \neq 1$$

Por lo tanto:

$$T(n) \in O(2^n)$$

Es un algoritmo de coste exponencial; es habitual que las funciones recursivas múltiples tengan costes prohibitivos.

1.4.2 Resolución general de recurrencias

Se pueden obtener unos resultados teóricos aplicables a un gran número de recurrencias, aquellas en las que la descomposición recursiva se obtiene por sustracción y aquellas otras en las que se obtiene por división.

Disminución del tamaño del problema por sustracción

Si la descomposición recursiva se obtiene restando una cierta cantidad constante, tenemos entonces que esto da lugar a recurrencias de la forma:

(D) $T(n) = G(n)$ si $0 \leq n < b$

(R) $T(n) = a \cdot T(n-b) + C(n)$ si $n \geq b$

donde:

- $a \geq 1$ es el número de llamadas recursivas
- $b \geq 1$ es la disminución del tamaño de los datos
- $G(n)$ es el coste en el caso directo
- $C(n)$ es el coste de preparación de las llamadas y de combinación de los resultados

Aplicando la técnica de despliegue sobre este esquema:

$$\begin{aligned}
T(n) &= a \cdot T(n-b) + C(n) \\
&= a \cdot (a \cdot T(n-b-b) + C(n-b)) + C(n) \\
&= a^2 \cdot T(n-2b) + a \cdot C(n-b) + C(n) \\
&= a^2 \cdot (a \cdot T(n-2b-b) + C(n-2b)) + a \cdot C(n-b) + C(n) \\
&= a^3 \cdot T(n-3b) + a^2 \cdot C(n-2b) + a \cdot C(n-b) + C(n) \\
&\ldots \\
T(n) &= a^m \cdot T(n - m \cdot b) + \sum_{i=0}^{m-1} a^i \cdot C(n - i \cdot b)
\end{aligned}$$

Como nos interesa el comportamiento asintótico podemos tomas las siguientes hipótesis simplificadoras:

$n = m \cdot b \Leftrightarrow m = n/b$ es decir, consideramos sólo los *n* que son múltiplos de *b*

$T(0) = G(0) = c_0$ de los posibles casos directos ($0 \le n < b$) nos quedamos con uno ($n = 0$)

De esta forma podemos eliminar la recurrencia de la expresión anterior, obteniendo:

$$T(n) = a^m \cdot c_0 + \sum_{i=0}^{m-1} a^i \cdot C(m \cdot b - i \cdot b)$$

Donde damos nombre a cada uno de los sumandos para poder tratarlos por separado:

$$T_1(n) = a^m \cdot c_0$$
$$T_2(n) = \sum_{i=0}^{m-1} a^i \cdot C(b \cdot (m - i))$$

Estudiamos ahora un caso particular frecuente en la práctica: cuando el coste de preparar las llamadas y combinar los resultados es de la forma:

$C(n) = c \cdot n^k$ para ciertas $c \in R^+$, $k \in N$

Y ahora estudiamos dos posibilidades dependiendo del valor de *a* –el número de llamadas recursivas–:

— a = 1

$$T_1(n) = c_0$$
$$T_2(n) = \sum_{i=0}^{m-1} c \cdot (b \cdot (m - i))^k$$
$$= c \cdot b^k \cdot \sum_{i=0}^{m-1} (m - i)^k = c \cdot b^k \cdot \frac{m^k + 1^k}{2} \cdot m$$
$$\le c \cdot b^k \cdot m^{k+1}$$
$$= c \cdot b^k \cdot (n/b)^{k+1}$$
$$= c/b \cdot n^{k+1}$$

por lo tanto

$$T_2(n) \in O(n^{k+1})$$

y como $T_2(n)$ domina a $T_1(n)$, que es una constante, tenemos

$$T(n) = c_0 + T_2(n) \in O(n^{k+1})$$

Cuando se aplica una única llamada recursiva (a = 1) obtenemos un orden de complejidad polinómico en *n*, cuyo exponente es uno más que el del coste de *C*(n)

— a > 1

$$T_1(n) = a^m \cdot c_0$$

$$T_2(n) = \sum_{i=0}^{m-1} a^i \cdot c \cdot (b \cdot (m - i))^k$$

$$= \sum_{i=0}^{m-1} a^i \cdot \frac{a^m}{a^m} c \cdot (b \cdot (m - i))^k$$ % multiplicando arriba y abajo por a^m

$$= a^m \cdot c \cdot b^k \cdot \sum_{i=0}^{m-1} \frac{a^i \cdot (m-i)^k}{a^m}$$

$$= a^m \cdot c \cdot b^k \cdot \sum_{i=0}^{m-1} \frac{(m-i)^k}{a^{m-i}}$$

y ahora hacemos un cambio de índices para así darnos cuenta de que esta serie converge.

$$j = m-i \qquad i = 0 \Leftrightarrow j = m \qquad i = m-1 \Leftrightarrow j = 1$$

entonces

$$T_2(n) = a^m \cdot c \cdot b^k \cdot \sum_{j=1}^{m} \frac{j^k}{a^j}$$

y se cumple

$$T_2(n) = a^m \cdot c \cdot b^k \cdot \sum_{j=1}^{m} \frac{j^k}{a^j} < a^m \cdot c \cdot b^k \cdot s$$

siendo *s* la suma de la serie convergente

$$s = \sum_{j=1}^{\infty} \frac{j^k}{a^j}$$

tenemos por lo tanto

$$T_2(n) \in O(a^m)$$

$$T(n) = c_0 \cdot a^m + T_2(n) \in O(a^m) = O(a^{n \text{ div } b})$$

Es decir, cuando tenemos varias llamadas recursivas (a>1) y el tamaño del problema disminuye por sustracción, la complejidad resultante es exponencial. Por lo tanto, la recursión múltiple con división del problema por sustracción conduce a algoritmos muy ineficientes; como ocurre por ejemplo con el algoritmo para las torres de Hanoi donde

$a = 2 \qquad b = 1 \qquad T(n) \in O(2^n)$

Disminución del tamaño del problema por división

Si la descomposición recursiva se obtiene dividiendo por una cierta cantidad constante, tenemos entonces que esto da lugar a recurrencias de la forma:

(D) $T(n) = G(n)$ si $0 \leq n < b$

(R) $T(n) = a \cdot T(n/b) + C(n)$ si $n \geq b$

donde:

- $a \geq 1$ es el número de llamadas recursivas
- $b \geq 2$ es el factor de disminución del tamaño de los datos
- $G(n)$ es el coste en el caso directo
- $C(n)$ es el coste de preparación de las llamadas y de combinación de los resultados

Aplicando la técnica de despliegue sobre este esquema:

$$
\begin{aligned}
T(n) &= a \cdot T(n/b) + C(n) \\
&= a \cdot (a \cdot T(n/b/b) + C(n/b)) + C(n) \\
&= a^2 \cdot T(n/b^2) + a \cdot C(n/b) + C(n) \\
&= a^2 \cdot (a \cdot T(n/b^2/b) + C(n/b^2)) + a \cdot C(n/b) + C(n) \\
&= a^3 \cdot T(n/b^3) + a^2 \cdot C(n/b^2) + a \cdot C(n/b) + C(n)
\end{aligned}
$$

…

$$T(n) = a^m \cdot T(n/b^m) + \sum_{i=0}^{m-1} a^i \cdot C(n/b^i)$$

Como nos interesa el comportamiento asintótico podemos tomas las siguientes hipótesis simplificadoras:

$n = b^m \Leftrightarrow m = \log_b n$ es decir, consideramos sólo los *n* que son potencias de *b*

$T(1) = G(1) = c_1$ de los posibles casos directos ($0 \leq n < b$) nos quedamos con uno ($n = 1$)

De esta forma podemos eliminar la recurrencia de la expresión anterior, obteniendo:

$$T(n) = a^m \cdot c_1 + \sum_{i=0}^{m-1} a^i \cdot C(n/b^i) = a^m \cdot c_1 + \sum_{i=0}^{m-1} a^i \cdot C(b^m/b^i)$$

$$T(n) = a^m \cdot c_1 + \sum_{i=0}^{m-1} a^i \cdot C(b^{m-i})$$

Donde damos nombre a cada uno de los sumandos para poder tratarlos por separado:

$$T_1(n) = a^m \cdot c_1$$

$$T_2(n) = \sum_{i=0}^{m-1} a^i \cdot C(b^{m-i})$$

podemos transformar $T_1(n)$ de la siguiente forma

$$T_1(n) = a^m \cdot c_1 = a^{\log_b n} \cdot c_1 = c_1 \cdot n^{\log_b a}$$

donde la última igualdad se tiene por

$$a^{\log_b n} = \left(b^{\log_b a}\right)^{\log_b n} = b^{(\log_b n)(\log_b a)} = n^{\log_b a}$$

Continuamos el estudio suponiendo, como ya hicimos en el estudio anterior, que

$$C(n) = c \cdot n^k \qquad \text{para ciertas } c \in R^+,\ k \in N$$

Y, de esta forma, podemos elaborar $T_2(n)$:

$$\begin{aligned} T_2(n) &= \sum_{i=0}^{m-1} a^i \cdot C(b^{m-i}) \\ &= \sum_{i=0}^{m-1} a^i \cdot c \cdot b^{(m-i)\,k} \\ &= c \cdot \sum_{i=0}^{m-1} a^i \cdot \frac{b^{m \cdot k}}{b^{m \cdot k}} \cdot b^{(m-i)\,k} \quad \text{\% multiplicando arriba y abajo por } b^{m\,k} \\ &= c \cdot b^{m \cdot k} \sum_{i=0}^{m-1} a^i \cdot b^{-i\,k} \\ &= c \cdot b^{m \cdot k} \sum_{i=0}^{m-1} \left(\frac{a}{b^k}\right)^i \end{aligned}$$

Con lo que la complejidad queda

$$T(n) = c_1 \cdot n^{\log_b a} + c \cdot b^{m \cdot k} \sum_{i=0}^{m-1} \left(\frac{a}{b^k}\right)^i$$

Estudiamos el valor del sumatorio según la relación que exista entre a y b^k.

— $a = b^k$

Tenemos entonces

$$T_1(n) = c_1 \cdot n^k$$

$$T_2(n) = c \cdot b^{m \cdot k} \sum_{i=0}^{m-1} 1^i$$

$$= c \cdot b^{m \cdot k} \cdot m$$

$$= c \cdot n^k \cdot \log_b n \qquad \% \text{ ya que } m = \log_b n$$

De esta forma

$$T(n) = c_1 \cdot n^k + c \cdot n^k \cdot \log_b n \in O(n^k \log_b n) = O(n^k \log n)$$

Vemos que $T_2(n)$ domina a $T_1(n)$. Observamos asimismo que la complejidad depende de n^k que es un término que proviene de C(n), por lo tanto la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de $C(n) = c \cdot n^k$.

— $a < b^k$

Tenemos entonces que

$$\log_b a < k$$

y, por lo tanto

$$T_1(n) = c_1 \cdot n^{\log_b a} < c_1 \cdot n^k$$

$$T_2(n) = c \cdot b^{m \cdot k} \sum_{i=0}^{m-1} \left(\frac{a}{b^k} \right)^i$$

como $a < b^k$ tenemos una serie geométrica de razón $r = a/b^k < 1$

$$T_2(n) = c \cdot b^{m \cdot k} \cdot \frac{r^m - 1}{r - 1}$$

$$= c \cdot \frac{a^m - b^{mk}}{(a / b^k) - 1}$$

si $a < b^k$ entonces $a^m < b^{mk}$ por lo tanto el término dominante en la anterior expresión es b^{mk} y podemos afirmar (nótese que la anterior expresión tiene signo positivo):

$$T_2(n) \in O(b^{mk}) = O(b^{(\log_b n)k}) = O(n^k)$$

Vemos que $T_2(n)$ domina a $T_1(n)$ y por lo tanto

$$T(n) \in O(n^k)$$

Observamos asimismo que la complejidad depende de n^k que es un término que proviene de C(n), por lo tanto la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de $C(n) = c \cdot n^k$.

— $a > b^k$

Tenemos entonces que

$$\log_b a > k$$

y, por lo tanto

$$T_1(n) = c_1 \cdot n^{\log_b a}$$

$$T_2(n) = c \cdot b^{m \cdot k} \sum_{i=0}^{m-1} \left(\frac{a}{b^k} \right)^i$$

como $a > b^k$ tenemos una serie geométrica de razón $r = a/b^k > 1$

$$T_2(n) = c \cdot b^{m \cdot k} \cdot \frac{r^m - 1}{r - 1}$$

$$= c \cdot \frac{a^m - b^{mk}}{(a / b^k) - 1}$$

si $a > b^k$ entonces $a^m > b^{mk}$ por lo tanto el término dominante en la anterior expresión es a^m y podemos afirmar (nótese que la anterior expresión tiene signo positivo):

$$T_2(n) \in O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a})$$

por lo tanto, por la regla de la suma de complejidades

$$T(n) = T_1(n) + T_2(n) \in O(n^{\log_b a})$$

Observamos aquí que $T_1(n)$ y $T_2(n)$ tienen el mismo orden de complejidad, que sólo depende de *a* y *b*. Por lo tanto, en este caso no se consigue mejorar la eficiencia global mejorando la eficiencia de C(n). Las mejoras se obtendrán disminuyendo *a* –el número de llamadas recursivas– o aumentando *b* –obteniéndose así subproblemas de menor tamaño–.

División del problema por sustracción

$$\begin{cases} c_0 & \text{si } 0 \le n < b \end{cases}$$

$$T(n) = \begin{cases} a \cdot T(n-b) + c \cdot n^k & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} O(n^{k+1}) & \text{si } a = 1 \\ O(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

Cuando se aplica una única llamada recursiva (a = 1) obtenemos un orden de complejidad polinómico en *n*, cuyo exponente es uno más que el del coste de *C*(n).

Cuando tenemos varias llamadas recursivas (a>1) y el tamaño del problema disminuye por sustracción, la complejidad resultante es exponencial. Por lo tanto, la recursión múltiple con división del problema por sustracción conduce a algoritmos muy ineficientes; como ocurre por ejemplo con el algoritmo para las torres de Hanoi donde

$a = 2 \qquad b = 1 \qquad T(n) \in O(2^n)$

División del problema por división

$$T(n) = \begin{cases} c_1 & \text{si } 0 \leq n < b \\ a \cdot T(n/b) + c \cdot n^k & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} O(n^k) & \text{si } a < b^k \\ O(n^k \cdot \log n) & \text{si } a = b^k \\ O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Si $a < b^k$ o $a = b^k$ la complejidad depende de n^k que es un término que proviene de C(n), por lo tanto la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de la preparación de las llamadas y la combinación de los resultados: $C(n) = c \cdot n^k$.

Si $a > b^k$ no se consigue mejorar la eficiencia global mejorando la eficiencia de C(n). Las mejoras se obtendrán disminuyendo *a* –el número de llamadas recursivas– o aumentando *b* –obteniéndose así subproblemas de menor tamaño–.

1.4.3 Análisis de algunos algoritmos recursivos

Ordenación por mezcla

La recurrencia es de la forma

$$T(n) = \begin{cases} c_1 & \text{si } -1 \leq n \leq 0 \end{cases}$$

$$2 \cdot T(n/2) + c \cdot n \qquad \text{si } n \geq 1$$

donde hemos considerado que el tamaño de los datos *n* es la longitud del subvector a ordenar; en el caso directo tenemos 2 pasos para evaluar las dos barreras. En cada llamada recursiva el problema se divide *aproximadamente* por 2. Nótese que la ecuación de recurrencias no se ajusta exactamente al esquema teórico (b=2 pero la expresión recurrente se aplica para $n \geq 1$ y no para $n \geq 2$); como las diferencias afectan a valores pequeños de *n* no afectan a su estudio asintótico.

En realidad no tenemos que preocuparnos por el coste en el caso directo, siempre y cuando sea un coste constante. Tampoco nos preocupamos por las constantes aditivas que aparezcan en el caso recursivo.

El coste de la combinación ($c \cdot n^k$) sale lineal porque ese es el coste del procedimientos *mezcla*.

Con todo esto tenemos

$$a = 2, b = 2, k = 1$$

estamos por tanto en el caso

$$a = b^k$$

y entonces la complejidad es:

$$O(n \cdot \log n)$$

Números de Fibonacci

$$T(0) = c_0$$
$$T(1) = c_1$$
$$T(n) = T(n-1) + T(n-2) + c \qquad \text{si } n \geq 2$$

Esta recurrencia no se ajusta a los esquemas que hemos estudiado, y por ello hacemos una simplificación:

$$T(n) \leq 2 \cdot T(n-1) + c$$

Que se corresponde al esquema de división del problema por sustracción:

$$a = 2, b = 1, k = 0$$

Como se tiene $a > 1$ la complejidad es

$$O(a^{n \text{ div } b}) = O(2^n)$$

complejidad exponencial.

Fibonacci con recursión final

Es la función *dosFib* del ejercicio 99

$T(0) = c_0$
$Tn) = T(n–1) + c \qquad n > 0$

División del problema por sustracción

$a = 1, b = 1, k = 0$

Como se tiene a = 1

$O(n^{k+1}) = O(n)$

Esto implica una enorme mejora con respecto a la versión anterior del algoritmo.

Suma recursiva de un vector de enteros

```
func sumaVec ( v : Vector[1..N] de Ent; a, b : Ent ) dev s : Ent;
```

Lo vimos en el tema de derivación de algoritmos recursivos. Suma v(a) al resultado de sumar recursivamente v[a+1..b].

Tomamos como tamaño de los datos

$n = b – a + 1$ la longitud del subvector a sumar

Recurrencia:

$T(1) = c_1$
$T(n) = T(n–1) + c$ si $n > 1$

División del problema por substracción

$a = 1, b = 1, k = 0$

Como se tiene a = 1

$O(n^{k+1}) = O(n)$

Búsqueda dicotómica

Tomamos como tamaño de los datos:

$n = b - a + 1$

Recurrencias

$T(0) = c_0$
$T(1) = c_1$
$T(n) = T(n/2) + c$ si $n > 1$ Suponiendo $n = 2^m$

División del problema por división:

$a = 1, b = 2, k = 0$
Como $a = b^k \Leftrightarrow 1 = 2^0$

$O(n^k \cdot \log n) = O(n^0 \cdot \log n) = O(\log n)$

Es una gran mejora con respecto a la búsqueda secuencial, que tiene complejidad $O(n)$. Nótese que log n no está definido para n = 0; podríamos dar como orden de complejidad $O(\log (n+1))$, o simplemente ignorar el valor n = 0, pues estamos dando una medida asintótica.

Método de ordenación rápida

También tomamos como tamaño de los datos la longitud del vector:

$n = b - a + 1$

Aquí el punto clave es cuántos elementos se quedan a la izquierda y cuántos a la derecha del elemento pivote. El caso peor es cuando no separa nada, es decir, es el mínimo o el máximo del intervalo; en ese caso:

$T(0) = c_0$
$T(n) = T(0) + T(n-1) + c \cdot n$ si $n \geq 1$

El procedimiento *partición*, que prepara las llamadas recursivas, tiene coste lineal, de ahí el término $c \cdot n$. El término T(0) que aparece en la recurrencia es una constante que ignoramos en el análisis.

Estamos entonces en el caso de división por sustracción

$a = b = k = 1$ $O(n^{k+1}) = O(n^2)$

El caso peor se produce cuando el vector está ordenado, creciente o decrecientemente.

El caso mejor se obtiene suponiendo que el pivote divide al subvector en dos mitades iguales, en cuyo caso tenemos:

$T(0) = c_0$

$T(n) = 2 \cdot T(n/2) + c \cdot n \qquad \text{si } n \geq 1$

con

$a = b = 2 \qquad k = 1$

disminución del problema por división con $a = b^k$

$O(n^k \log n) = O(n \log n)$

Con una ligera modificación del algoritmo, se puede conseguir que el caso peor no ocurra cuando el vector está ordenado: seleccionando como pivote el elemento central del intervalo.

Se puede demostrar que en promedio la complejidad coincide con la del caso mejor.

2.5 Transformación de la recursión final a forma iterativa

En general un algoritmo iterativo es más eficiente que uno recursivo porque la invocación a procedimientos o funciones tiene un cierto coste. Es por ello que tiene sentido plantearse la transformación de algoritmos recursivos en iterativos.

Los compiladores en algunos casos –recursión final– eliminan la recursión al traducir los programas a código máquina.

El inconveniente de transformar los algoritmos recursivos en iterativos radica en que puede ocurrir que el algoritmo iterativo sea menos claro, con lo cual se mejora la eficiencia a costa de perjudicar a la facilidad de mantenimiento. Como en tantas otras ocasiones es necesario llegar a compromisos.

Eliminación de la recursión final

El esquema general de una función recursiva final es como ya vimos en un tema anterior:

```
func nombreFunc ( x₁ : τ₁; … ; xₙ : τₙ ) dev y₁ : δ₁; … ; yₘ : δₘ;
{ P(x⃗) }
var
  x⃗' : τ⃗';
inicio
  si
    d(x⃗) → { P(x⃗) ∧ d(x⃗) }
              y⃗ := g(x⃗)
              { Q(x⃗, y⃗) }
   ¬d(x⃗) → { P(x⃗) ∧ ¬d(x⃗) }
              x⃗' := s(x⃗);
              { P(x⃗') }
```

```
                $\vec{y}$ := nombreFunc( $\vec{x}$' )
              { Q($\vec{x}$', $\vec{y}$) }
              { Q($\vec{x}$, $\vec{y}$) }
  fsi
{ Q($\vec{x}$, $\vec{y}$) }
  dev $\vec{y}$
ffunc
```

Intuitivamente, podemos imaginar la ejecución de una llamada de la forma

$\vec{y}$:= nombreFunc($\vec{x}$)

como un "bucle descendente"

$\vec{x} \rightarrow$ nombreFunc($\vec{x}$)
↓
nombreFunc(s($\vec{x}$))
↓
nombreFunc($s^2(\vec{x})$)
↓
…
↓
nombreFunc($s^n(\vec{x})$) → g($s^n(\vec{x})$) → $\vec{y}$

De hecho, habría otro "bucle ascendente" que iría devolviendo el valor de $\vec{y}$; sin embargo, como en ese proceso no se modifica $\vec{y}$ –recursión final– podemos ignorarlo.

Formalmente, el bucle descendente da lugar a una definición iterativa equivalente de *nombre-Func* de la siguiente forma:

```
func nombreFuncItr( x₁ : τ₁; … ; xₙ : τₙ ) dev y₁ : δ₁; … ; yₘ : δₘ;
{ P($\vec{x}$) }
var
  $\vec{x}$' : $\vec{\tau}$';
inicio
  $\vec{x}$' := $\vec{x}$;
{ I : P($\vec{x}$') ∧ nombreFunc($\vec{x}$) = nombreFunc($\vec{x}$');
  C : t($\vec{x}$')                                          }
  it ¬d($\vec{x}$')
```

```
  → x⃗' := s(x⃗')
  fit;
{ I ∧ d(x⃗') }
{ nombreFunc(x⃗) = g(x⃗') }
  y⃗ := g(x⃗')
{ y⃗ = nombreFunc(x⃗) }
{ Q(x⃗, y⃗) }
  dev y⃗
ffunc
```

Este paso se puede realizar de forma mecánica. Además la verificación del algoritmo iterativo se deduce de las condiciones conocidas para el algoritmo recursivo:

— La precondición y la postcondición coinciden

— El invariante I es P($\vec{x}$') ∧ nombreFunc($\vec{x}$') = nombreFunc($\vec{x}$). Justo antes de cada iteración es como si estuviésemos delante de una llamada recursiva, donde se cumple la precondición sustituyendo $\vec{x}$ por s($\vec{x}$). Además por ser recursiva final, se tiene la segunda parte del invariante: la función aplicada sobre cualquiera de los valores intermedios coincide con el resultado para el valor original.

— La expresión de acotación viene dada por la función de acotación del algoritmo recursivo.

La corrección del algoritmo recursivo garantiza la del algoritmo iterativo obtenido de esta manera.

1.5.1 Ejemplos

Vamos a aplicar la transformación a dos algoritmos concretos.

Factorial

Transformamos la versión recursiva final, *acuFact*, que desarrollamos en el ejercicio 82:

```
func acuFact( a, n : Nat ) dev m : Nat;
{ P₀ : a = A ∧ n = N }
var
  a', n' : Nat;
inicio
  si
     n = 0 → m := a
  □  n > 0 → <a',n'> := <a*n, n-1>;
              m := acuFact( a', n' )
  fsi
{ Q₀ : a = A ∧ n = N ∧ m = a * n! }
  dev m
ffunc
```

En esta versión hemos separado el cálculo de la descomposición recursiva de los datos para que así se ajuste exactamente al esquema teórico presentado.

Y la versión iterativa:

```
func acuFact( a, n : Nat ) dev m : Nat;
{ P₀ : a = A ∧ n = N }
var
  a', n' : Nat;
inicio
  <a', n'> := <a, n>;
  it n' > 0
  → <a', n'> := <a'*n', n'-1>;
  fit;
  m := a';
{ Q₀ : a = A ∧ n = N ∧ m = a * n! }
  dev m
ffunc
```

En realidad no harían falta las variables locales, pero si no las utilizamos no se cumplirá la parte de la especificación que se refiere a la conservación de los valores de los parámetros de entrada.

Considerando que

fact(n) = acuFact(1, n)

podemos convertir la versión iterativa en una función que calcule el factorial, eliminando el parámetro *a* e inicializando *a'* con el valor 1.

Búsqueda binaria

Partimos de la solución que obtuvimos en el tema de derivación de algoritmos recursivos:

```
func buscaBin( v : Vector[1..N] de Elem; x : Elem ; a, b : Ent ) dev p :
Ent;
{ P₀ : 1 ≤ a ≤ b ≤ N ∧ ord(v,a,b) }
var
  m : Ent;
inicio
  si
    a = b+1 → p := a - 1
  □ a = b   → si
                  v(a) ≤ x → p := a
                  v(a) > x → p := a-1
              fsi
```

```
  □ a < b   → m := (a+b) div 2;
              si
                  v(m) ≤ x → p := buscaBin( v, x, m+1, b )
                □ v(m) > x → p := buscaBin( v, x, a, m-1 )
              fsi
  fsi
{ Q₀ : a-1 ≤ p ≤ b ∧ v[a..p] ≤ x < v[(p+1)..b] }
  dev p
ffunc
```

Esta función no se ajusta exactamente al esquema teórico porque no aparecen las variables auxiliares que recogen la descomposición recursiva; las introduciremos en la versión iterativa. Además tenemos más de un caso directo e implícitamente más de un caso recursivo, lo que se traduce a composiciones alternativas en la versión iterativa.

La solución iterativa

```
func buscaBin( v : Vector[1..N] de Elem; x : Elem ; a, b : Ent ) dev p :
Ent;
{ P₀ : 1 ≤ a ≤ b ≤ N ∧ ord(v,a,b) }
var
  m, a', b' : Ent;
inicio
  <a',b'> := <a,b>;
  it a' < b'
  → m := (a'+b') div 2;
     si
       v(m) ≤ x → <a', b'> := <m+1, b'>;
     □ v(m) > x → <a', b'> := <a', m-1>;
     fsi
  fit;
  si
    a' = b'+1 → p := a' - 1
  □ a' = b'   → si
                  v(a') ≤ x → p := a'
                  v(a') > x → p := a'-1
                fsi
  fsi
{ Q₀ : a-1 ≤ p ≤ b ∧ v[a..p] ≤ x < v[(p+1)..b] }
  dev p
ffunc
```

Podemos eliminar los parámetros adicionales *a* y *b*, inicializando *a'* y *b'* con los valores 1 y *N* respectivamente. Obsérvese que el algoritmo iterativo difiere ligeramente del que derivamos en el

tema de algoritmos iterativos; la razón es que en aquél utilizábamos un planteamiento ligeramente diferente al de la implementación recursiva:

— el límite del vector que se modifica se lleva a m y no a m–1 o m+1.

— se inicializan a y b con los valores 0 y N+1 en lugar de 1 y N.

En la versión derivada directamente como iterativa no aparece la composición alternativa a continuación del bucle.

La transformación de recursivo a iterativo en funciones recursivas lineales no finales necesita en general el uso de una pila, por lo tanto posponemos su estudio al tema de ese TAD. La conversión de recursión múltiple necesita de un árbol.

2.6 Técnicas de generalización y plegado-desplegado.

En este tema vamos a ver una introducción elemental a un conjunto de técnicas que ayudan a:

— Plantear el diseño de un algoritmo recursivo a partir de su especificación, es decir, técnicas que ayudan a obtener planteamientos recursivos de los problemas.

— Transformar un algoritmo recursivo ya diseñado a otro equivalente y más eficiente. (La mejora en la eficiencia radicará fundamentalmente en la conversión a un algoritmo recursivo final que es directamente traducible a un algoritmo iterativo, más eficiente.)

En ambos casos nos preocuparemos especialmente por obtener algoritmos recursivos finales debido a la correspondencia directa que existe entre este tipo de algoritmos recursivos y algoritmos iterativos equivalentes, más eficientes.

Nos limitaremos al estudio de funciones aunque estas técnicas se pueden adaptar fácilmente al diseño de procedimientos.

1.6.1 Generalizaciones

Todas las técnicas que vamos a estudiar se basan en la idea de **generalización** (en los libros recomendados a las generalizaciones se las denomina **inmersiones**).

Decimos que una función F es una generalización de otra función f cuando:

— F tiene más parámetros de entrada y/o devuelve más resultados que f.

— Particularizando los parámetros de entrada adicionales de F a valores adecuados y/o ignorando los resultados adicionales de F se obtiene el comportamiento de f.

En esta primera parte del tema vamos a considerar generalizaciones que sólo introducen parámetros adicionales y no resultados adicionales.

Al estudiar funciones recursivas ya nos hemos encontrado con varios ejemplos de generalizaciones, por ejemplo, *acuFact* es una generalización de *fact* de forma que

---

acuFact(1, n) = fact(n)

En este ejemplo la generalización es interesante porque conduce fácilmente a una función recursiva final, lo cual no ocurre si partimos de la especificación de *fact*.

Otro ejemplo típico son las funciones recursivas sobre vectores, donde hemos obtenido generalizaciones añadiendo parámetros que indiquen el subvector a considerar en cada llamada recursiva. Por ejemplo, la función de búsqueda binaria implementada recursivamente es una generalización de la misma función implementada de forma iterativa:

---

buscaBinRec(v, x, 1, N) = buscaBin(v, x)

Aunque nosotros a la hora de implementarlas le hemos dado el mismo nombre a las dos funciones, en realidad se trata de funciones distintas pues tienen parámetros distintos.

Estos dos ejemplos ponen de manifiesto las dos razones para obtener generalizaciones que, como ya indicamos antes, son obtener soluciones recursivas más eficientes o posibilitar los planteamientos recursivos.

Vamos a caracterizar formalmente en términos de su especificación qué relación debe existir entre una función y su generalización.

Dadas dos especificaciones

(E_f) **func** f($\vec{x}$; $\vec{\tau}$) **dev** $\vec{y}$: $\vec{\sigma}$;
{ P($\vec{x}$) }
{ Q($\vec{x}$, $\vec{y}$) }
ffunc

(E_F) **func** F($\vec{a}$: $\vec{\tau}$ '; $\vec{x}$; $\vec{\tau}$) **dev** $\vec{b}$: $\vec{\sigma}$ '; $\vec{y}$: $\vec{\sigma}$;
{ P'($\vec{a}$, $\vec{x}$) }
{ Q'($\vec{a}$, $\vec{x}$, $\vec{b}$, $\vec{y}$) }
ffunc

Decimos que E_F es una generalización de E_f si se cumplen

(G1) $P(\vec{x}) \wedge \vec{a} = \text{ini}(\vec{x}) \Rightarrow P'(\vec{a}, \vec{x})$

Si estamos en un estado donde es posible invocar a *f* entonces también es posible invocar a *F*, asignando a los parámetros adicionales los valores que indica una cierta función *ini*.

(G2) $Q'(\vec{a}, \vec{x}, \vec{b}, \vec{y}) \wedge \vec{a} = \text{ini}(\vec{x}) \Rightarrow Q(\vec{x}, \vec{y})$

> De la postcondición de *F*, restringiendo los parámetros adicionales a los valores dados por *ini*($\vec{x}$), es posible obtener la postcondición de *f*.

Donde *ini* es una función que asocia a cada valor de los parámetros $\vec{x}$ el valor *ini*($\vec{x}$) de los parámetros adicionales, adecuado para que F(*ini*($\vec{x}$), $\vec{x}$) emule el comportamiento de f($\vec{x}$)

Hemos introducido aquí una nueva notación para escribir la cabecera de las funciones, denotando con $\vec{x}$ a una lista de variables y con $\vec{\tau}$ a una lista de tipos.

Veamos como ejemplo que *acuFact* es una generalización de *fact* según la definición que acabamos de dar:

```
func fact( n : Ent ) dev r : Ent;
{ P(n) : n ≥ 0 }
{ Q(n, r) : r = n! }
ffunc

func acuFact( a, n : Ent ) dev r : Ent;
{ P'(a,n) : n ≥ 0 }
{ Q(a, n, r) : r = a * n! }
ffunc
```

Efectivamente *acuFact* es una generalización de *fact* con

```
ini(n) = 1
```

Con lo que se tiene

```
n ≥ 0 ∧ a = 1 ⇒ n ≥ 0

r = a * n! ∧ a = 1 ⇒ r = n!
```

En este tema vamos a obviarlas condiciones relativas a la conservación de los valores de los parámetros, para simplificar un poco el tratamiento.

Al aplicar la técnica de las generalizaciones, partiremos de la especificación E_f e intentaremos obtener la especificación E_F de una generalización adecuada. Este proceso es esencialmente *creativo*; vamos a estudiar a continuación la aplicación de esta técnica a algunos casos particulares y presentaremos así heurísticas que pueden ayudar a descubrir las generalizaciones interesantes.

1.6.2 Generalización para la obtención de planteamientos recursivos

Vamos a estudiar dos técnicas, ejemplificándolas, una más simple que permite obtener planteamientos recursivos no finales y otra que persigue la obtención de planteamientos recursivos que admiten recursión final.

Planteamientos recursivos no finales

El objetivo es llegar a una generalización para la cual exista un planteamiento recursivo *evidente*. La idea consiste en añadir parámetros de entrada adicionales, como ocurre típicamente en los algoritmos sobre vectores, donde es necesario qué fragmento del vector se considera en cada llamada recursiva. Simplemente lo que vamos a hacer en este apartado es formalizar algo que ya hemos estado haciendo de manera informal.

Lo que se puede intentar es generalizar la postcondición introduciendo variables nuevas que serán los parámetros adicionales de la generalización. En el caso de los vectores, lo normal es generalizar la postcondición sustituyendo alguna de las constantes, que fijan los límites del vector, por variables.

Buscamos una postcondición Q' de forma que se cumpla la condición G2

$$Q'(\vec{a}\,,\ \vec{x}\,,\ \vec{y}) \wedge \vec{a} = \text{ini}(\vec{x}) \Rightarrow Q(\vec{x}\,,\ \vec{y})$$

en ese caso la nueva precondición se obtendrá añadiendo a la precondición original asertos de dominio, $D(\vec{a}\,,\ \vec{x})$, sobre los nuevos parámetros:

$$P'(\vec{a}\,,\ \vec{x}) \Leftrightarrow P(\vec{x}) \wedge D(\vec{a}\,,\ \vec{x})$$

Veamos cómo se aplica esta técnica en un ejemplo que calcula el producto escalar de dos vectores:

```
func prodEsc( u, v : Vector[1..N] de Ent ) dev p : Ent;
{ P(u, v) : cierto }
{ Q(u, v, p) : p = ∑ i : 1 ≤ i ≤ N : u(i)*v(i) }
ffunc
```

Buscamos una postcondición que generalice a la anterior, sustituyendo la constante *N* por un nuevo parámetro:

$$Q(u, v, p) \Leftarrow Q'(a, u, v, p) \wedge a = N$$

por lo tanto la nueva postcondición debe ser de la forma:

$$Q'(a, u, v, p) \Leftrightarrow p = \sum i : 1 \leq i \leq a : u(i) * v(i)$$

la nueva precondición se obtendrá añadiendo a la antigua un aserto de dominio sobre el nuevo parámetro *a*

$$P'(a, u, v) \Leftrightarrow P(u, v) \wedge 0 \leq a \leq N$$

con lo que la especificación de la generalización del producto escalar queda:

```
func prodEscGen( a : Ent; u, v : Vector[1..N] de Ent ) dev p : Ent;
{ P'(a, u, v) : 0 ≤ a ≤ N }
{ Q'(a, u, v, p) : p = ∑ i : 1 ≤ i ≤ a : u(i)*v(i) }
ffunc
```

vemos que efectivamente esta es una generalización con

```
ini(u,v) = N
```

de forma que

```
prodEscGen( N, u, v ) = prodEsc( u, v )
```

A partir de la especificación de la función generalizada es muy sencillo construir una solución recursiva:

```
func prodEscGen( a : Ent; u, v : Vector[1..N] de Ent ) dev p : Ent;
{ P'(a, u, v) : 0 ≤ a ≤ N }
inicio
  si
    a = 0 → p := 0
  □ a > 0 → p := u(a)*v(a) + prodEscGen( a-1, u, v )
  fsi
{ Q'(a, u, v, p) : p = ∑ i : 1 ≤ i ≤ a : u(i)*v(i) }
  dev p
ffunc
```

Siguiendo el mismo proceso es inmediato construir otra generalización de *prodEsc* sustituyendo por una constante 1 en lugar de *N*.

Planteamientos recursivos finales

Nuestro objetivo es, dada una especificación E_f, encontrar una especificación E_F de una función más general que admita una solución recursiva final. En una función recursiva final el resultado se obtiene en un caso directo, y para conseguirlo lo que podemos hacer es añadir nuevos parámetros que vayan acumulando el resultado obtenido hasta el momento, de forma que al llegar al caso base de la función general *F* el valor del parámetro acumulador sea precisamente el resultado de la función *f*. Formalmente, esto quiere decir que tendremos que fortalecer la precondición para exigir que alguno de los parámetros de entrada ya traiga calculado una parte del resultado.

En este tipo de generalizaciones la postcondición permanece constante (salvo la conservación de los valores de los parámetros adicionales).

Lo que tenemos que exigirle a una generalización final es, por lo tanto:

(FG2) $P'(\vec{a}, \vec{e}, \vec{x}) \wedge d(\vec{a}, \vec{e}, \vec{x}) \Rightarrow Q(\vec{x}, \vec{a})$

donde

— $\vec{a}$ son los parámetros acumuladores

— $\vec{e}$ son otros parámetros extra

— $d(\vec{a}, \vec{e}, \vec{x})$ es la condición del caso directo de F

— $Q(\vec{x}, \vec{y})$ es la postcondición de la función f

Es decir, que la precondición y la condición del caso directo de la función generalizada permiten obtener la postcondición de la función *f*, sustituyendo los parámetros de salida por los parámetros acumuladores, con lo que en el caso directo nos limitaremos a asignar el valor de los parámetros acumuladores a las variables de salida.

Esta condición hace el papel de (G2) para el caso de las generalizaciones recursivas finales. La otra condición, (G1), se que da tal cual:

(FG1) ≡ (G1) $P(\vec{x}) \wedge <\vec{a}, \vec{e}> = ini(\vec{x}) \Rightarrow P'(\vec{a}, \vec{e}, \vec{x})$

Ya sabemos que las funciones recursivas finales se puede traducir de manera inmediata a funciones iterativas; es por ello que las condiciones que aparecen en la caracterización de una generalización recursiva final se corresponden de manera directa con las de los bucles, como ya pudimos darnos cuenta cuando presentamos el esquema de traducción de una función recursiva final a otra iterativa:

$P'(\vec{a}, \vec{e}, \vec{x})$	invariante
$\neg d(\vec{a}, \vec{e}, \vec{x})$	condición de repetición
$Q(\vec{x}, \vec{y})$	postcondición

donde tenemos también que el requisito FG2 es equivalente a una de las condiciones que han de verificar los bucles:

$I \wedge \neg B \Rightarrow Q \quad \equiv \quad P' \wedge d \Rightarrow Q$

Para obtener este tipo de generalizaciones utilizamos las mismas heurísticas que hemos estudiado para la obtención de invariantes: tomar una parte de la postcondición o generalizar la postcondición sustituyendo constantes por variables. La condición que obtengamos así deberemos incluirla en la precondición de la generalización.

Como ejemplo vamos a obtener otra generalización de la función que calcula el producto escalar. Intuitivamente, vemos que tenemos que utilizar la idea de la generalización anterior para poder llegar a un planteamiento recursivo y además tenemos que introducir otro parámetro que

lleve el resultado hasta ese momento. Con esta idea planteamos la siguiente precondición para la generalización (este paso es el equivalente a obtener el invariante a partir de la postcondición, con la diferencia de que al obtener el invariante no necesitamos una variable nueva para acumular el resultado y aquí sí):

$P'(a, e, u, v) : a = \sum i : 1 \leq i \leq e : u(i)*v(i) \wedge 0 \leq e \leq N$

El siguiente paso es encontrar la condición del caso directo de forma que se garantice la condición (FG2), es decir, para conseguir que en *a* esté el resultado final deseado. Es equivalente a obtener la condición de terminación:

$$d(a, e, u, v) \Leftrightarrow e = N$$

De forma que así podemos demostrar la condición (FG2)

$$P'(a, e, u, v) \wedge d(a, e, u, v) \Rightarrow Q(u, v, a)$$

(Nótese que en *Q* aparece *a* como resultado; es decir, en el caso base simplemente se asigna a *p* el valor de *a*)

Por último, nos resta encontrar la función *ini*(u, v) que asigne valores a (a, e) de forma que se cumpla (FG1):

$$P(u, v) \wedge \langle a, e \rangle = ini(u, v) \Rightarrow P'(a, e, u, v)$$

Esto es equivalente a obtener la acción de inicialización en un bucle, tenemos que encontrar asignaciones a las variables del bucle que hagan trivialmente cierta la precondición de la función generalizada (i.e. el invariante). La forma más sencilla es consiguiendo que el dominio del sumatorio sea el conjunto vacío:

$ini(u, v) = \langle 0, 0 \rangle$

A partir de esta especificación es sencillo diseñar el siguiente algoritmo recursivo final:

```
func prodEscGenFin( a, e : Ent; u, v : Vector[1..N] de Ent ) dev p Ent;
{ P'(a, e, u, v) }
inicio
  si
    e = N → p := a
  □ e < N → p := prodEscGenFin( a + u(e+1) * v(e+1), e+1, u, v )
  fsi
{ Q(u, v, p) }
  dev p
ffunc
```

Nótese que al final de todas las llamadas recursivas el valor de *p* es el resultado final, por eso la postcondición es exactamente la misma que la de la función factorial sin ningún tipo de generalización.

Tenemos entonces que

prodEsc(u, v) = prodEscGenFin(0, 0, u, v)

De la misma se puede seguir un proceso similar para obtener una generalización sustituyendo por una nueva variable la constante 1 en lugar de *N*.

El uso de acumuladores es una técnica que tiene especial relevancia en programación funcional donde, en principio, sólo se dispone de recursión. Esta es la forma de conseguir funciones recursivas eficientes, ocurriendo incluso que el compilador sea capaz de hacer la traducción automática de recursiva final a iterativa. En programación imperativa tiene menos sentido pues en muchos casos resulta más natural obtener directamente la solución iterativa.

1.6.3 Generalización por razones de eficiencia

En ocasiones se puede mejorar la eficiencia de un algoritmo recursivo evitando el cálculo repetido de una cierta expresión en todas las llamadas recursivas, o simplificando algún cálculo sacando provecho del resultado obtenido para ese cálculo en otra llamada recursiva. En estos casos el uso de generalizaciones puede mejorar la eficiencia, introduciendo parámetros adicionales que transmitan los resultados a llamadas recursivas posteriores, o con resultados adicionales que se utilicen en llamadas recursivas anteriores. En esencia, es la misma idea que se utiliza en la introducción de invariantes auxiliares cuando se diseñan algoritmos iterativos.

Vamos a distinguir dos casos, según que la generalización consista en añadir parámetros o en añadir resultados. También hay ocasiones en que interesa utilizar simultáneamente los dos tipos de generalizaciones. En ambos casos vamos a suponer que partimos de funciones ya diseñadas y describiremos las soluciones generalizadas en términos de los cambios que sería necesario introducir en las funciones originales.

Generalización con parámetros acumuladores

Cuando se aplica esta técnica, la función más general F posee parámetros de entrada adicionales $\vec{a}$, cuya función es transmitir el valor de una cierta expresión e($\vec{x}$) que depende de los parámetros de entrada de la función original *f*, a fin de evitar su cálculo. Por lo tanto la precondición de F deberá plantearse como un fortalecimiento de la precondición de *f*, y la postcondición podrá mantenerse tal cual:

$P'(\vec{a}, \vec{x}) \Leftrightarrow P(\vec{x}) \wedge \vec{a} = e(\vec{x})$

$Q'(\vec{a}, \vec{x}, \vec{y}) \Leftrightarrow Q(\vec{x}, \vec{y})$

Suponiendo disponible un algoritmo recursivo para f (que queremos optimizar), podremos diseñar un algoritmo más eficiente para F del siguiente modo:

— Se reutiliza el texto de *f*, reemplazando e($\vec{x}$) por $\vec{a}$

— Diseñamos una nueva función sucesor s'($\vec{a}$, $\vec{x}$), a partir de la orginal s($\vec{x}$), de modo que se cumpla:

```
{ a⃗ = e(x⃗) ∧ ¬d(x⃗) }
  <a⃗', x⃗'> := s'(a⃗, x⃗)
{ x⃗' = s(x⃗) ∧ a⃗' = e(x⃗') }
```

La técnica resultará rentable cuando en el cálculo de $\vec{a}$' nos podamos aprovechar de los valores de $\vec{a}$ y $\vec{x}$ para realizar un cálculo más eficiente.

Por ejemplo una función que calcula cuántas componentes de un vector son iguales a la suma de las componentes que las preceden:

```
func numCortesGen( e : Ent; v : Vector[1..N] de Ent ) dev r : Ent;
{ P(e,v) : 0 ≤ e ≤ N }
var
  s, h : Ent;
inicio
  si
    e = N → r := 0
  □ e < N → s := 0;
             para h desde 1 hasta e hacer
               s := s + v(h);
             fpara;
             si s = v(e+1)
               entonces r := 1 + numCortesGen( e+1, v )
               sino     r := numCortesGen( e+1, v )
             fsi
  fsi
{ Q( e, v, r ) : r = # i : e+1 ≤ i ≤ N : v(i) = ∑ j : 1 ≤ j ≤ e : v(j) }
  dev r
ffunc
```

Esta primer implementación ya es una generalización pues se ha introducido el parámetros *e* para hacer posible un planteamiento recursivo: obtener el número de *cortes* a la derecha de *e*, por lo que el valor inicial de *e* ha de ser 0. Aunque desde el punto de vista del planteamiento recursivo quizás resulte más natural recorrer el vector en sentido contrario, hasta llegar a 0, no lo hacemos así para luego hacer posible la siguiente generalización, que necesita la suma de las componentes que hay a la izquierda.

numCortesGen(0, v) = numCortes(v)

Introducimos un fortalecimiento de la precondición para que la suma del vector recorrido hasta ese momento se vaya transmitiendo a las siguientes llamadas:

```
func numCortesGenEfi( e, s : Ent; v : Vector[1..N] de Ent ) dev r : Ent;
{ P'(s,e,v) : 0 ≤ e ≤ N ∧ s = ∑ i : 1 ≤ i ≤ e : v(i) }
inicio
  si
    e = N → r := 0
```

```
  □ e < N → si s = v(e+1)
              entonces r := 1 + numCortesGenEfi( e+1, s+v(e+1), v )
              sino     r := numCortesGenEfi( e+1, s+v(e+1), v )
            fsi
  fsi
{ Q( e, v, r ) : r = # i : e+1 ≤ i ≤ N : v(i) = Σ j : 1 ≤ j ≤ e : v(j) }
  dev r
ffunc
```

En este caso tenemos

ini(v) = <0, 0>

numCortesGenEfi(0, 0, v) = numCortes(v)

La primera versión tiene coste cuadrático mientras que esta lo tiene lineal.

Es inmediato obtener otra generalización, añadiendo un parámetro más, que convierta esta función en recursiva final.

Generalización con resultados acumuladores

La principal diferencia entre parámetros acumuladores y resultados acumuladores radica en el lugar donde necesitamos la expresión cuyo cálculo queremos obviar: antes de la llamada recursiva –parámetros acumuladores– o después de la llamada recursiva –resultados acumuladores–.

Utilizamos por primera vez la definición más general de generalización que introdujimos al principio del tema, porque es la primera vez que la generalización va a incluir parámetros adicionales. Recordamos que la generalización debía cumplir:

(G1) $P(\vec{x}) \wedge \vec{a} = \text{ini}(\vec{x}) \Rightarrow P'(\vec{a}, \vec{x})$

Si estamos en un estado donde es posible invocar a *f* entonces también es posible invocar a *F*, asignando a los parámetros adicionales los valores que indica una cierta función *ini*.

(G2) $Q'(\vec{a}, \vec{x}, \vec{b}, \vec{y}) \wedge \vec{a} = \text{ini}(\vec{x}) \Rightarrow Q(\vec{x}, \vec{y})$

De la postcondición de *F*, restringiendo los parámetros adicionales a los valores dados por $ini(\vec{x})$, es posible obtener la postcondición de *f*.

Donde *ini* es una función que asocia a cada valor de los parámetros $\vec{x}$ el valor $ini(\vec{x})$ de los parámetros adicionales, adecuado para que $F(ini(\vec{x}), \vec{x})$ emule el comportamiento de $f(\vec{x})$

En una generalización F con resultados acumuladores $\vec{b}$, estos parámetros de salida adicionales tendrán como misión transmitir ciertos valores $e(\vec{y})$ dependientes de los resultados $\vec{y}$ de f.

Tenemos una función f con la siguiente llamada recursiva

$\vec{y}$' := f($\vec{x}$')

y a continuación aparece una expresión de la forma

e($\vec{y}$ ', $\vec{x}$ ')

la idea es conseguir una generalización F donde

< $\vec{b}$ ', $\vec{y}$ '> := F($\vec{x}$ ')

tal que

$\vec{b}$ ' = e($\vec{y}$ ', $\vec{x}$ ')

Suponiendo que F no introduzca parámetros acumuladores tenemos la siguiente relación entre pre y postcondiciones:

P'($\vec{x}$) $\Leftrightarrow$ P($\vec{x}$)

Q'($\vec{x}$, $\vec{b}$, $\vec{y}$) $\Leftrightarrow$ Q($\vec{x}$, $\vec{y}$) $\wedge$ $\vec{b}$ = e($\vec{y}$, $\vec{x}$)

Suponiendo disponible un algoritmo recursivo para *f*, que pretendemos optimizar, el diseño de una algoritmo recursivo más eficiente para *F* se obtendrá

— Reutilizar el texto de *f*, reemplazando e($\vec{y}$ ', $\vec{x}$ ') por $\vec{b}$ ' (el valor de los resultados acumuladores en la llamada recursiva)

— Añadir el cálculo de $\vec{b}$, de manera que la parte $\vec{b}$ =e($\vec{y}$, $\vec{x}$) de la postcondición Q'($\vec{x}$, $\vec{b}$, $\vec{y}$) quede garantizada, tanto en los casos directos como en los recursivos

---

La técnica resultará rentable siempre que *F* sea más eficiente que *f*.

Es posible utilizar conjuntamente las dos técnicas de generalización para optimización, incluyendo parámetros y resultados acumuladores, como se muestra en un ejemplo de [Peña98], pág. 93, para el cálculo eficiente de la raíz cuadrado por defecto.

Vamos a ver un par de ejemplos, uno para la obtención de la suma de los cuadrados de los *n* primeros números naturales y otro para el cálculo de los números de Fibonacci.

```
func sumaCuad( n : Nat ) dev s : Nat;
{ P(n) : cierto }
inicio
  si
    n = 0 → s := 0
  □ n > 0 → s := n*n + sumaCuad(n-1)
  fsi
{ Q(n, s) : s = ∑ i : 0 ≤ i ≤ n : i*i }
  dev s
ffunc
```

La idea de la optimización consiste en optimizar el cálculo de n^2, conociendo del valor de $(n-1)^2$, ya que $\vec{x}' = n-1$:

$$n^2 = ((n-1)+1)^2 = (n-1)^2 + 2*(n-1) + 1$$

La idea es añadir resultados adicionales que devuelvan
n^2 $2*n+1$

La generalización queda por tanto:

```
func sumaCuadGen( n : Nat ) dev c, p, s : Nat
{ P'(n) : cierto }
inicio
  si
    n = 0 → <c, p, s> := <0, 1, 0>
  □ n > 0 → <c', p', s'> := sumaCuadGen( n-1 );
            <c, p, s> := <c'+p', p'+2, c'+p'+s'>
  fsi
{ Q'(n, c, p, c) : s = ∑ i : 0 ≤ i ≤ n : i*i ∧ c = n*n ∧ p = 2*n+1 }
  dev <c, p, s>
ffunc
```

sumaCuadGen generaliza a sumaCuad si descartamos los dos resultados adicionales:
$sumaCuad(n) = pr_3(sumaCuadGen(n))$

donde pr_3 es la función que proyecta una terna ordenada en su tercera componente.

Vamos ahora con el ejemplo de los números de Fibonacci, la optimización consiste en devolver dos números en lugar de uno:

```
func fibo(n : Nat ) dev r : Nat;
{ P(n) : cierto }
inicio
  si
    n = 0 → r := 0
  □ n = 1 → r := 1
  □ n > 1 → r := fibo(n-1) + fibo(n-2)
  fsi
{ Q(n, r) : r = fib(n) }
  dev r
ffunc
```

y la generalización

```
func fiboGen( n : Nat ) dev r, s : Nat;
{ P(n) : cierto }
var
  r', s' : Nat;
inicio
  si
    n = 0 → <r, s> := <0, 1>
  □ n > 0 → <r',s'> := fiboGen(n-1)
             <r, s> := <s', r'+s'>
  fsi
{ Q(n, r, s) : r = fib(n) ∧ s = fib(n+1) }
  dev <r, s>
ffunc

fibo(n) = pr1(fiboGen(n))
```

1.6.4 Técnicas de plegado-desplegado

En un apartado anterior hemos visto cómo generalizar una especificación E_f de manera que la especificación más general E_F haga posible un diseño recursivo final. Una idea alternativa es la de **transformación de programas**. Si suponemos ya diseñado un algoritmo recursivo simple que satisfaga la especificación E_f, puede interesar transformarlo de manera que se obtenga un algoritmo recursivo final para una función más general con especificación E_F. La versión recursiva final será, como ya sabemos, más eficiente por poderse traducir a un bucle iterativo.

Las técnicas de plegado–desplegado sirven para obtener el algoritmo recursivo final para *F* mediante manipulaciones algebraicas del algoritmo recursivo simple conocido para *f*.

Lo interesante de esta técnica es que es posible automatizar la transformación pues los pasos están bien definidos a partir de la forma de la función recursiva simple original. El inconveniente es que no es aplicable a cualquier función recursiva simple, sino que estas se deben ajustar a unos determinados esquemas donde las funciones que intervienen cumplan unas ciertas propiedades –asociatividad, elemento neutro, …–.

Vamos a estudiar tres esquemas distintos junto con el procedimiento para construir la función recursiva final a partir de una función recursiva simple que se ajuste al correspondiente esquema y cumpla las propiedades indicadas.

Plegado y desplegado I

Supongamos dada una función recursiva simple *f* cuya implementación se ajuste al siguiente esquema:

```
func f( x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P( x⃗ ) }
```

```
inicio
  si
     d(x⃗) → y⃗ := g(x⃗)
  □ ¬d(x⃗) → y⃗ := h(x⃗) ⊕ f(s(x⃗))
  fsi
{ Q(x⃗, y⃗) }
  dev y⃗
ffunc
```

siendo $\oplus$ una función

$\oplus : \vec{\sigma} \times \vec{\sigma} \rightarrow \vec{\sigma}$

que cumple las propiedades

(N) $\forall\ \vec{u} : \vec{\sigma} : \vec{0} \oplus \vec{u} = \vec{u}$ ($\vec{0}$ constante)

(A) $\forall\ \vec{u}, \vec{v}, \vec{w} : \vec{\sigma} : \vec{u} \oplus (\vec{v} \oplus \vec{w}) = (\vec{u} \oplus \vec{v}) \oplus \vec{w}$

entonces la función F, especificada como sigue, es una generalización de f que admite un algoritmo recursivo final:

```
func F (a⃗ : σ⃗ ; x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P(x⃗) }
{ y⃗ = a⃗ ⊕ f(x⃗) }
ffunc
```

Primero vamos a ver que efectivamente F es una generalización de f:

$\forall\ \vec{x} : \vec{\tau} : P(\vec{x})$

$$
\begin{array}{rcl}
 & & f(\vec{x}) \\
(N) & = & \vec{0} \oplus f(\vec{x}) \\
(E_F) & = & F(\vec{0}, \vec{x})
\end{array}
$$

Para demostrar que F admite un algoritmo recursivo final daremos el procedimiento que lo construye, a partir del algoritmo para f y la especificación de F. El algoritmo sintetizado será correcto por construcción.

Para la síntesis del algoritmo utilizamos el análisis de casos del algoritmo de f. En lo que sigue presuponemos que los valores de $\vec{x}$ cumplen $P(\vec{x})$:

Caso $d(\vec{x})$

$$
\begin{array}{rcl}
 & & F(\vec{a}, \vec{x}) \\
E_F, \text{ desplegado de } F & = & \vec{a} \oplus f(\vec{x}) \\
D_f, \text{ desplegado de } f & = & \vec{a} \oplus g(\vec{x})
\end{array}
$$

Caso $\neg$d($\vec{x}$)

$$
\begin{array}{rcl}
 & & \mathtt{F}(\vec{a}\,,\vec{x}\,) \\
\mathtt{E_F,\ desplegado\ de\ F} & = & \vec{a}\ \oplus\ \mathtt{f}(\vec{x}\,) \\
\mathtt{R_f,\ desplegado\ de\ f} & = & \vec{a}\ \oplus\ (\mathtt{h}(\vec{x}\,)\ \oplus\ \mathtt{f}(\mathtt{s}(\vec{x}\,))) \\
\mathtt{(A)} & = & (\vec{a}\ \oplus\ \mathtt{h}(\vec{x}\,))\ \oplus\ \mathtt{f}(\mathtt{s}(\vec{x}\,)) \\
\mathtt{E_F,\ plegado\ de\ F} & = & \mathtt{F}(\vec{a}\ \oplus\ \mathtt{h}(\vec{x}\,),\ \mathtt{s}(\vec{x}\,))
\end{array}
$$

Por lo tanto obtenemos el siguiente algoritmo para F

```
func F (a⃗ : σ⃗ ; x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P(x⃗ ) }
inicio
  si
     d(x⃗ ) → y⃗ := a⃗ ⊕ g(x⃗ )
  □ ¬d(x⃗ ) → y⃗ := F( a⃗ ⊕ h(x⃗ ), s(x⃗ ))
  fsi
{ y⃗ = a⃗ ⊕ f(x⃗ ) }
  dev y⃗
ffunc
```

Ejemplos de funciones recursivas que se ajustan al esquema PDP I y que admiten esta transformación:

La función *acuFact* del ejercicio 82, y que apareció al principio de este tema como ejemplo de generalización, se obtiene aplicando PDP I a la función *fact*

$$\mathtt{h(n) = n} \qquad \oplus = * \qquad \vec{0} = 1$$

La función *prodEscGen* que vimos al principio de este tema –pág. 5– como ejemplo de generalización que hace posible la recursión con planteamiento recursivo no final se ajusta al esquema PDP I con los siguiente valores para los parámetros del método:

$$\mathtt{h(u, v, n) = u(n) * v(n)} \qquad \oplus = + \qquad \vec{0} = 0$$

El resultado de la transformación es:

```
func prodEscF( b, a : Ent; u, v : Vector[1..N] de Ent ) dev p : Ent;
{ P(b, a, u, v) : 0 ≤ a ≤ N }
```

```
inicio
  si
    a = 0 → p := b
  □ a > 0 → p := prodEscF( b + u(a)*v(a), a-1, u, v )
  fsi
{ Q(b, a, u, v, p) : p = b + ∑ i : 1 ≤ i ≤ a : u(i)*v(i) }
  dev p
ffunc
```

En el ejemplo que sigue la aplicación de la técnica se complica un poco porque tenemos dos casos recursivos y la función $h(\vec{x})$ se define por distinción de casos. El algoritmo es el método de multiplicación del campesino egipcio –que presentamos en el tema de verificación de algoritmos recursivos, aunque aquí se ha eliminado uno de los casos bases por ser redundante–.

```
func prod ( x, y : Nat ) dev r : Nat;
{ P₀ : cierto }
inicio
  si  x = 0              → r := 0
  □   (x>0 AND par(x))     → r := prod(x div 2, y+y )
  □   (x>0 AND NOT par(x)) → r := y + prod(x div 2, y+y)
  fsi
{ Q₀ : r = x * y }
  dev r
ffunc
```

La transformación es aplicable con los siguientes parámetros:

$\oplus$ = +

$\vec{0}$ = 0

$$h(x,y) = \begin{cases} 0 & \text{si } par(x) \\ y & \text{si } \neg par(x) \end{cases}$$

Con todo ello la función transformada queda:

```
func prodF ( a, x, y : Nat ) dev r : Nat;
{ P₀ : cierto }
inicio
  si  x = 0              → r := a
  □   (x>0 AND par(x))     → r := prodF( a, x div 2, y+y )
  □   (x>0 AND NOT par(x)) → r := prodF( a+y, x div 2, y+y)
  fsi
{ Q₀ : r = a + x * y }
  dev r
ffunc
```

Plegado y desplegado II

Supongamos dada una función recursiva simple *f* cuya implementación se ajuste al siguiente esquema:

---

func f($\vec{x}$: $\vec{\tau}$) **dev** $\vec{y}$: $\vec{\sigma}$;
{ P($\vec{x}$) }
inicio
  si
    d($\vec{x}$) → $\vec{y}$:= g($\vec{x}$)
  □ ¬d($\vec{x}$) → $\vec{y}$:= h($\vec{x}$) ⊕ (k($\vec{x}$) ⊗ f(s($\vec{x}$)))
  fsi
{ Q($\vec{x}$, $\vec{y}$) }
  dev $\vec{y}$
ffunc

siendo ⊕, ⊗ funciones

⊕, ⊗ : $\vec{\sigma}$ x $\vec{\sigma}$ → $\vec{\sigma}$

que cumple las propiedades

(N⊕) ∀ $\vec{u}$: $\vec{\sigma}$: $\vec{0}$ ⊕ $\vec{u}$ = $\vec{u}$ ($\vec{0}$ constante)
(N⊗) ∀ $\vec{u}$: $\vec{\sigma}$: $\vec{1}$ ⊗ $\vec{u}$ = $\vec{u}$ ($\vec{1}$ constante)
(A⊕) ∀ $\vec{u}$, $\vec{v}$, $\vec{w}$: $\vec{\sigma}$: $\vec{u}$ ⊕ ($\vec{v}$ ⊕ $\vec{w}$) = ($\vec{u}$ ⊕ $\vec{v}$) ⊕ $\vec{w}$
(A⊗) ∀ $\vec{u}$, $\vec{v}$, $\vec{w}$: $\vec{\sigma}$: $\vec{u}$ ⊗ ($\vec{v}$ ⊗ $\vec{w}$) = ($\vec{u}$ ⊗ $\vec{v}$) ⊗ $\vec{w}$
(D) ∀ $\vec{u}$, $\vec{v}$, $\vec{w}$: $\vec{\sigma}$: $\vec{u}$ ⊗ ($\vec{v}$ ⊕ $\vec{w}$) = ($\vec{u}$ ⊗ $\vec{v}$) ⊕ ($\vec{u}$ ⊗ $\vec{w}$)

entonces la función *F*, especificada como sigue, es una generalización de *f* que admite un algoritmo recursivo final:

func F ($\vec{a}$: $\vec{\sigma}$; $\vec{b}$: $\vec{\sigma}$; $\vec{x}$: $\vec{\tau}$) **dev** $\vec{y}$: $\vec{\sigma}$;
{ P($\vec{x}$) }
{ $\vec{y}$ = $\vec{a}$ ⊕ ($\vec{b}$ ⊗ f($\vec{x}$)) }
ffunc

Primero vamos a ver que efectivamente *F* es una generalización de *f*:

∀ $\vec{x}$: $\vec{\tau}$: P($\vec{x}$)

f($\vec{x}$)

$$(N_{\oplus}, N_{\otimes}) = \vec{0} \oplus (\vec{1} \otimes f(\vec{x}))$$
$$(E_F) = F(\vec{0}, \vec{1}, \vec{x})$$

Para demostrar que F admite un algoritmo recursivo final daremos el procedimiento que lo construye, a partir del algoritmo para f y la especificación de F. El algoritmo sintetizado será correcto por construcción.

Para la síntesis del algoritmo utilizamos el análisis de casos del algoritmo de f. En lo que sigue presuponemos que los valores de $\vec{x}$ cumplen $P(\vec{x})$:

Caso $d(\vec{x})$

$$F(\vec{a}, \vec{b}, \vec{x})$$
$$E_F, \text{ desplegado de } F = \vec{a} \oplus (\vec{b} \otimes f(\vec{x}))$$
$$D_f, \text{ desplegado de } f = \vec{a} \oplus (\vec{b} \otimes g(\vec{x}))$$

Caso $\neg d(\vec{x})$

$$F(\vec{a}, \vec{b}, \vec{x})$$
$$E_F, \text{ desplegado de } F = \vec{a} \oplus (\vec{b} \otimes f(\vec{x}))$$
$$R_f, \text{ desplegado de } f = \vec{a} \oplus [\vec{b} \otimes [h(\vec{x}) \oplus [k(\vec{x}) \otimes f(s(\vec{x}))]]]$$
$$(D) = \vec{a} \oplus [[\vec{b} \otimes h(\vec{x})] \oplus [\vec{b} \otimes [k(\vec{x}) \otimes f(s(\vec{x}))]]]$$
$$(A_{\oplus}, A_{\otimes}) = [\vec{a} \oplus [\vec{b} \otimes h(\vec{x})]] \oplus [[\vec{b} \otimes k(\vec{x})] \otimes f(s(\vec{x}))]$$
$$E_F, \text{ plegado de } F = F(\vec{a} \oplus (\vec{b} \otimes h(\vec{x})), \vec{b} \otimes k(\vec{x}), s(\vec{x}))$$

Por lo tanto obtenemos el siguiente algoritmo para F

```
func F (a⃗ : σ⃗ ; b⃗ : τ⃗ ; x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P(x⃗) }
inicio
  si
     d(x⃗) → y⃗ := a⃗ ⊕ (b⃗ ⊗ g(x⃗))
  □ ¬d(x⃗) → y⃗ := F( a⃗ ⊕ (b⃗ ⊗ h(x⃗)), b⃗ ⊗ k(x⃗), s(x⃗) )
  fsi
{ y⃗ = a⃗ ⊕ (b⃗ ⊗ f(x⃗)) }
  dev y⃗
ffunc
```

Como ejemplo vamos a ver una función que dada una base b y un natural n obtiene la representación de n en base b –en el ejercicio 92 se obtenía esta función para el caso particular b=2–.

```
func cambioBase( b, n : Nat ) dev r : Nat;
{ P₀ : 2 ≤ b ≤ 9 }
```

```
inicio
  si
    n < b → r := n
  □ n ≥ b → r := n mod b + 10 * cambioBase(b, n div b)
  fsi
{ Q₀ : r = ∑ i : Nat : ((n div bⁱ) mod b) * 10ⁱ }
  dev r
ffunc
```

Esta función se ajusta a PDP II con los siguientes parámetros:

$\oplus$ = + $\vec{0}$ = 0

$\otimes$ = * $\vec{1}$ = 1

h(n, b) = n mod b

k(n, b) = 10

Con todo ello el resultado de la transformación:

```
func cambioBaseF( s, p, b, n : Nat ) dev r : Nat;
{ P₀ : 2 ≤ b ≤ 9 }
inicio
  si
    n < b → r := s + p * n
  □ n ≥ b → r := cambioBaseF( s + p * n mod b, p * 10, b, n div b)
  fsi
{ Q₀ : r = s + p * ∑ i : Nat : ((n div bⁱ) mod b) * 10ⁱ }
  dev r
ffunc

cambioBaseF( 0, 1, b, n ) = cambioBase( b, n )
```

La función que se obtiene con la transformación es mucho más oscura que la función original. Esto es típico de las optimizaciones, se pierde claridad a cambio de ganar eficiencia. Para documentar la solución eficiente se puede utilizar la solución intuitiva. Existen entornos –en el ámbito de la investigación– que realizan estas transformaciones automáticamente, el usuario escribe el diseño intuitivo y el sistema se encarga de optimizarlo.

Plegado y desplegado III

Supongamos dada una función recursiva simple que se ajuste al siguiente esquema:

```
func f(x⃗ : τ⃗ ; a⃗ : τ⃗ ') dev y⃗ : σ⃗ ;
{ P(x⃗ ) }
inicio
```

```
  si
     d(x⃗) → y⃗ := g(a⃗)
  □ ¬d(x⃗) → y⃗ := h(f(s(x⃗),a⃗))
  fsi
{ Q(x⃗, a⃗, y⃗) }
  dev y⃗
ffunc
```

Lo que tiene de particular esta función es que su solución se obtiene aplicando $n(\vec{x})$ veces la función *h* al resultado del caso base $g(\vec{a})$, siendo $n(\vec{x})$ el número de veces que hay que aplicar la descomposición recursiva *s* para llegar al caso base. Es decir, el resultado sólo depende de los parámetros que controlan el avance de la recursión para determinar cuántas veces se aplica la función *h*:

$$\forall\ \vec{x} : P(\vec{x}) : f(\vec{x}, \vec{a}) = h^{n(\vec{x})}(g(a))$$

Afirmamos entonces que la función *F* especificada como sigue es una generalización de *f* que admite un algoritmo recursivo final:

```
func F(x⃗ : τ⃗ ; b⃗ : σ) dev y⃗ : σ⃗ ;
{ P(x⃗) }
{ y⃗ = h^n(x⃗)(b⃗) }
ffunc
```

siendo

$$n(\vec{x}) =_{def} \min n : Nat : d(s^n(\vec{x}))$$

es decir, *n* representa el número de llamadas recursivas necesarias para alcanzar un caso directo a partir de $\vec{x}$.

Vemos que efectivamente el valor de *f* se puede obtener a partir de *F* pues se cumple

---

$$P(\vec{x}) \Rightarrow f(\vec{x}, \vec{a}) = F(\vec{x}, g(\vec{a}))$$

–Podemos obviar la siguiente demostración y considerar que la ecuación es suficientemente intuitiva–.

Esto no se ajusta al concepto de generalización que hemos manejado hasta ahora pues *F* no introduce parámetros ni resultados adicionales. Sustituye unos parámetros por otros: '$\vec{a} : \vec{\tau}$' por $\vec{b} : \vec{\sigma}$. Sí será una generalización –como veremos en un ejemplo posterior– si $\vec{a}$ es la tupla vacía y $\vec{b}$ no lo es –no puede serlo en ningún caso–.

Para demostrar la anterior ecuación se puede utilizar inducción sobre la función de acotación de *f*, $t(\vec{x})$, demostrando la ecuación cuando $\vec{x}$ cumple la condición del caso directo, y tomando como hipótesis de inducción

$$f(s(\vec{x}), a) = F(s(\vec{x}), g(\vec{a}))$$

demostrando entonces que

$$f(\vec{x}, a) = F(\vec{x}, g(\vec{a}))$$

es decir, suponiendo que es cierto para un valor menor sobre el dominio de inducción demostramos que es cierto para un valor mayor, $t(s(\vec{x})) < t(\vec{x})$

Caso $d(\vec{x})$

$$\begin{aligned} & f(\vec{x}, \vec{a}) \\ (D_f) \quad &= \quad g(\vec{a}) \\ (d(\vec{x}) \Rightarrow n(\vec{x})=0) \quad &= \quad h^{n(\vec{x})}(g(\vec{a})) \\ (E_F) \quad &= \quad F(\vec{x}, g(\vec{a})) \end{aligned}$$

Caso $\neg d(\vec{x})$

$$\begin{aligned} & f(\vec{x}, \vec{a}) \\ (R_f) \quad &= \quad h(f(s(\vec{x}), \vec{a})) \\ H.I. \quad &= \quad h(F(s(\vec{x}), g(\vec{a}))) \\ (E_F) \quad &= \quad h(h^{n(s(\vec{x}))}(g(\vec{a}))) \\ (1+n(s(\vec{x})) = n(\vec{x})) \quad &= \quad h^{n(\vec{x})}(g(\vec{a})) \\ (E_F) \quad &= \quad F(\vec{x}, g(\vec{a})) \end{aligned}$$

Una vez demostrado que *f* se puede obtener a partir de *F*, veamos cuál es el procedimiento para construir el algoritmo de *F* a partir del algoritmo de *f*.

Caso $d(\vec{x})$

$$\begin{aligned} & F(\vec{x}, \vec{b}) \\ E_F, \text{ desplegado de } F \quad &= \quad h^{n(\vec{x})}(\vec{b}) \\ (d(\vec{x}) \Rightarrow n(\vec{x})=0) \quad &= \quad \vec{b} \end{aligned}$$

Caso $\neg d(\vec{x})$

$$\begin{aligned} & F(\vec{x}, \vec{b}) \\ E_F, \text{ desplegado de } F \quad &= \quad h^{n(\vec{x})}(\vec{b}) \\ \neg d(\vec{x}) \Rightarrow n(\vec{x}) = 1+n(s(\vec{x})) \quad &= \quad h^{n(s(\vec{x}))}(h(\vec{b})) \\ E_F, \text{ plegado de } F \quad &= \quad F(s(\vec{x}), h(\vec{b})) \end{aligned}$$

Con lo que obtenemos para *F* el siguiente algoritmo:

```
func F(x⃗ : τ⃗ ; b⃗ : σ) dev y⃗ : σ⃗ ;
{ P(x⃗) }
inicio
  si
     d(x⃗) → y⃗ := b⃗
  □ ¬d(x⃗) → y⃗ := F( s(x⃗), h(b⃗) )
  fsi
{ y⃗ = h^n(x⃗)(b⃗) }
  dev y⃗
ffunc
```

Veamos algunos ejemplos de aplicación de PDP III.

Es aplicable a la generalización de *fibo* que vimos en el apartado sobre generalizaciones por razones de eficiencia:

```
func fiboGen( n : Nat ) dev r, s : Nat;
{ P(n) : cierto }
var
  r', s' : Nat;
inicio
  si
    n = 0 → <r, s> := <0, 1>
  □ n > 0 → <r',s'> := fiboGen(n-1)
            <r, s> := <s', r'+s'>
  fsi
{ Q(n, r, s) : r = fib(n) ∧ s = fib(n+1) }
  dev <r, s>
ffunc
```

Corresponde al esquema de PDP III con los siguientes parámetros:

```
x⃗ = n
a⃗ = < >
b⃗ = <r', s'>     del mismo tipo que el resultado Nat x Nat
g( ) = <0, 1>
h(r',s') = <s', r'+s'>
```

Con lo que se obtiene la función:

```
func fiboGenF( n, r', s' : Nat ) dev r, s : Nat;
{ cierto }
inicio
  si
    n = 0 → <r, s> := <r', s'>
  □ n > 0 → <r, s> := fiboGenF(n-1, s', r'+s')
  fsi
{ <r, s> = h^n(r', s') }
  dev <r, s>
ffunc
```

Como postcondición no podemos poner nada sobre los números de Fibonacci porque todas las llamadas devuelven el mismo valor –por algo es recursiva final– : los números de Fibonacci *n* y *n*+1 siendo *n* el valor con el que se hizo la primera invocación. Los que sí son números de Fibonacci son <r', s'>.

Para obtener el *n*-ésimo número de Fibonacci hacemos la siguiente invocación:

```
fiboGen( n ) = fiboGenF( n, 0, 1 ) = < fib(n), fib(n+1) >
```

Aplicando a este algoritmo la transformación de recursivo a iterativo obtendríamos aproximadamente el algoritmo que derivamos en el tema de algoritmos iterativos como ejemplo de introducción de un invariante auxiliar:

```
var n, x : Ent;
{ n = N ∧ n ≥ 0 }
var
  i, y : Ent;
inicio
  <i,x,y> := <0,0,1>;
{ I : n = N ∧ x = fib(i) ∧ 0 ≤ i ≤ n ∧ y = fib(i+1)
  C : n - i                                                    }
  it i ≠ n →
  { I ∧ i ≠ n }
      <x,y> := <y,x+y>;
  { I[i/i+1] }
      i := i +1
   { I }
  fit
{ I ∧ i = n }
fvar
{ n = N ∧ x = fib(n) }
```

La transformación PDP III también se puede aplicar a la función *sumaCuadGen* que obtuvimos en este mismo tema en el apartado dedicado a la generalización con parámetros acumuladores:

```
func sumaCuadGen( n : Nat ) dev c, p, s : Nat
{ P'(n) : cierto }
inicio
  si
    n = 0 → <c, p, s> := <0, 1, 0>
  □ n > 0 → <c', p', s'> := sumaCuadGen( n-1 );
            <c, p, s> := <c'+p', p'+2, c'+p'+s'>
  fsi
{ Q'(n, c, p, c) : s = ∑ i : 0 ≤ i ≤ n : i*i ∧ c = n*n ∧ p = 2*n+1 }
  dev <c, p, s>
ffunc
```

Con los parámetros:

$\vec{x}$ = n
$\vec{a}$ = < >
$\vec{b}$ = <c', p', s'>
g( ) = <0, 1, 0>
h<c', p', s'> = <c'+p', p'+2, c'+p'+s'>

Con lo que obtenemos el algoritmo

```
func sumaCuadGenF( n, c', p', s' : Nat ) dev c, p, s : Nat
{ cierto }
inicio
  si
    n = 0 → <c, p, s> := <c', p', s'>
  □ n > 0 → <c, p, s> := sumaCuadGenF( n-1, c'+p', p'+2, c'+p'+s' )
  fsi
{ <c, p, s> = h^n(c', p', s') }
  dev <c, p, s>
ffunc
```

Con la siguiente correspondencia entre *f* y *F*:

$$sumaCuadGen(n) = sumaCuadGenF(n, 0, 1, 0) = \langle n^2, 2n+1, sumaCuad(n) \rangle$$

2.7 Ejercicios

Introducción a la recursión

79. Construye y compara dos funciones que calculen el factorial de *n*, dado como parámetro:

(a) Una función *iterativa* (bucle con invariante $0 \le i \le n \wedge r = i!$).

(b) Una función *recursiva*.

80. Analiza cómo la función recursiva del ejercicio 79(b) se ajusta al esquema general de definición de una *función recursiva lineal* (o *simple*)

```
func nombreFunc (x1:τ1; … ; xn:τn) dev y1:δ1; … ; ym:δm;
{ P0 : P' ∧ x1 = X1 ∧ … ∧ xn = Xn }
cte … ;
var … ;
inicio
  si d(x⃗) → y⃗ := g(x⃗)
  □ ¬d(x⃗) → y⃗ := c(x⃗, nombreFunc( s(x⃗) ))
  fsi;
{ Q0 : Q' ∧ x1 = X1 ∧ … ∧ xn = Xn }
  dev <y1, … , ym>
ffunc
```

81. Construye una función recursiva simple *cuadrado* que calcule el cuadrado de un número natural *n*, basándote en el siguiente análisis de casos:

Caso directo: Si $n = 0$, entonces $n^2 = 0$

Caso recursivo: Si $n > 0$, entonces $n^2 = (n-1)^2 + 2*(n-1) + 1$

82. Una función se llama *recursiva final* si su definición se ajusta al siguiente esquema:

```
func nombreFunc (x1:τ1; … ; xn:τn) dev y1:δ1; … ; ym:δm;
{ P0 : P' ∧ x1 = X1 ∧ … ∧ xn = Xn }
cte … ;
var … ;
inicio
  si d(x⃗) → y⃗ := g(x⃗)
  □ ¬d(x⃗) → y⃗ :=nombreFunc( s(x⃗) )
  fsi;
{ Q0 : Q' ∧ x1 = X1 ∧ … ∧ xn = Xn }
  dev <y1, … , ym>
ffunc
```

Es decir, una función recursiva final es una función recursiva lineal especialmente sencilla, tal que la llamada recursiva devuelve directamente el resultado deseado.

Construye una función recursiva final que satisfaga la siguiente especificación pre/post:

```
func acuFact( a, n : Nat ) dev m : Nat;
```

```
{ P₀ : a = A ∧ n = N }
{ Q₀ : a = A ∧ n = N ∧ m = a * n! }
ffunc
```

Observa que la especificación de *acuFact* garantiza que acuFact(1,n) = n!.

83. Una función se llama *recursiva múltiple* si su definición se ajusta al siguiente esquema:

```
func nombreFunc (x₁:τ₁; … ; xₙ:τₙ) dev y₁:δ₁; … ; yₘ:δₘ;
{ P₀ : P' ∧ x₁ = X₁ ∧ … ∧ xₙ = Xₙ }
cte … ;
var … ;
inicio
  si
     d(x⃗) → y⃗ := g(x⃗)
  □ ¬d(x⃗) → y⃗ := c(x⃗, nombreFunc( s₁(x⃗) ), … , nombreFunc( sₖ(x⃗) ))
  fsi;
{ Q₀ : Q' ∧ x₁ = X₁ ∧ … ∧ xₙ = Xₙ }
  dev <y₁, … , yₘ>
ffunc
```

Construye una función recursiva múltiple *fib* con parámetro *n*, que devuelva el n-ésimo número de Fibonacci. Sigue el esquema de la definición recursiva de la sucesión de Fibonacci, según se conoce en matemáticas, distinguiendo los casos $0 \leq n \leq 1$ (caso directo) y $n \geq 2$ (caso recursivo).

84. Dibuja un diagrama en forma de árbol, representando todas las llamadas a la función *fib* que se originan a partir de la llamada incial *fib*(4). Cuenta el número total de llamadas y observa las llamadas repetidas.

85. Una función recursiva puede tener también varios casos directos y/o varios casos recursivos, en lugar de uno solo. Aunque haya varios casos recursivos, la recursión se considera lineal si éstos están separados, de tal manera que en cada llamada a la función se pase o bien por *uno solo* de los casos directos o bien por *uno solo* de los casos recursivos. Ejemplo:

(a) Función *recursiva lineal* que calcula potencias, con un caso recursivo:

```
func pot( x: Ent; y : Nat ) dev p : Ent;
{ P₀ : x = X ∧ y = Y }
var
  y' : Nat; p' : Ent;
inicio
  si
     y = 0 → p := 1
  □  y > 0 → y' := y div 2;
             p' := pot(x,y');
             si
                   par(y) → p := p'*p'
             □ NOT par(y) → p := x*p'*p'
             fsi
  fsi
{ Q₀ : x = X ∧ y = Y ∧ p = x^Y }
```

```
  dev p
ffunc
```

(b) Una función *recursiva lineal* que calcula potencias, con dos casos recursivos:

```
func pot'( x: Ent; y : Nat ) dev p : Ent;
{ P0 : x = X ∧ y = Y }
var
  x', y' : Nat; p' : Ent;
inicio
  si
                    y = 0 → p := 1

  □      y > 0 AND par(y) → <x',y'> := <x*x, y div 2);
                            p := pot'(x', y')
  □  y > 0 AND NOT par(y) →  y' := y-1;
                            p' := pot'(x,y')
                            p := x*p'
  fsi
{ Q0 : x = X ∧ y = Y ∧ p = x^Y }
  dev p
ffunc
```

Derivación y verificación de algoritmos recursivos

86. Verifica la función *fact* obtenida en el ejercicio 79(b).

87. Verifica la función *cuadrado* obtenida en el ejercicio 81.

88. Verifica la función *acuFact* obtenida en el ejercicio 82.

89. Verifica las funciones *pot* y *pot'* obtenidas en el ejercicio 85.

90. Deriva una función recursiva que calcule el producto de dos números `x,y:Nat`. Deberás obtener un algoritmo que sólo utilice las operaciones + y **div**, y que esté basado en el siguiente análisis de casos:

Caso directo: `x = 0`

Casos recursivos: `x ≠ 0 AND par(x)`
`x ≠ 0 AND NOT par(x)`

91. Deriva una función recursiva *log* que calcule la parte entera de $\texttt{lob}_{\texttt{b}}\ \texttt{n}$, siendo los datos `b, n : Nat` tales que `b ≥ 2 ∧ n ≥ 1`. El algoritmo obtenido deberá usar solamente las operaciones + y **div**.

92. Deriva una función recursiva *bin* tal que, dado un número natural *n*, *bin*(n) sea otro número natural cuya representación decimal tenga los mismos dígitos que la representación binaria de *n*. Es decir, debe tenerse: *bin*(0) = 0; *bin*(1) = 1; *bin*(2) = 10; *bin*(3) = 11; *bin*(4) = 100; etc.

93. Verifica la siguiente función:

```
func cuca( n : Ent ) dev r : Ent;
{ P₀ : n = N ∧ n ≥ 0 }
inicio
  si
     n = 0 → r := 0
  □  n = 1 → r := 1
  □  n ≥ 2 → r := 5*f(n-1) - 6*f(n-2)
  fsi
{ Q₀ : n = N ∧ r = 3^n - 2^n }
ffunc
```

94. Deriva una función recursiva doble que satisfaga la siguiente especificación:

```
func sumaVec( v : Vector[1..N] de Ent; a, b : Ent ) dev s : Ent;
{ P₀ : N ≥ 1 ∧ v = V ∧ a = A ∧ b = B ∧ 1 ≤ a ≤ b+1 ≤ N+1 }
{ Q₀ : v = V ∧ a = A ∧ b = B ∧ s = ∑ i : a ≤ i ≤ b : v(i) }
ffunc
```

Básate en el siguiente análisis de casos:

Caso directo: `a ≥ b`

`v[a..b]` tiene a lo sumo un elemento. El cálculo de *s* es simple.

Caso recursivo: a < b

`v[a..b]` tiene al menos dos elementos. Hacemos llamadas recursivas para sumar `v[a..m]` y `v[m+1..b]`, siendo `m = (a+b) div 2`.

95. Deriva una función recursiva lineal que realice el algoritmo de *búsqueda binaria* en un vector ordenado, cumpliendo la especificación siguiente:

```
func buscaBin( v : Vector[1..N] de Elem; x : Elem ; a, b : Ent ) dev p :
Ent;
{ P₀ : 1 ≤ a ≤ b+1 ≤ N+1 ∧ ord(v,a,b) }
{ Q₀ : a-1 ≤ p ≤ b ∧ v[a..p] ≤ x < v[(p+1)..b] }
ffunc
```

Compara con el algoritmo *iterativo* de búsqueda binaria del ejercicio 74.

96. Diseña un procedimiento doblemente recursivo que realice el algoritmo de ordenación rápida de un vector, según la especificación y análisis de casos que siguen:

```
proc quickSort( es v : Vector[1..N] de Elem; e a, b : Ent );
{ P₀ : v = V ∧ a = A ∧ b = B ∧ 1 ≤ a ≤ b + 1 ≤ N+1 }
{ Q₀ : a = A ∧ b = B ∧ perm(v, V, a, b) ∧ ord(v, a, b) ∧
       (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀j : b < j ≤ N : v(j) = V(j)) }
fproc
```

Caso directo: `a > b`

`v[a..b]` es vacío y ya está ordenado.

Caso recursivo: a ≤ b

`v[a..b]` es no vacío. Hacemos una llamada a un procedimiento auxiliar –*partición*(v, a, b, p)– que reorganiza `v[a..b]` desplazando v(a) a la posición *p*, dejando en `v[a..p-1]` elementos ≤ `v(p)`, y en `v[p+1..b]` elementos ≥ `v(p)`. A continuación, usamos llamadas recursivas para ordenar `v[a..p-1]` y `v[p+1..b]`.

†97. Diseña un procedimiento doblemente recursivo que realice el algoritmo de ordenación de un vector por el método de mezcla, según la especificación y análisis de casos que siguen:

```
proc mergeSort( es v : Vector[1..N] de Elem; e a, b : Ent );
{ P₀ : v = V ∧ a = A ∧ b = B ∧ 1 ≤ a ≤ b + 1 ≤ N+1 }
{ Q₀ : a = A ∧ b = B ∧ perm(v, V, a, b) ∧ ord(v, a, b) ∧
       (∀ i : 1 ≤ i < a : v(i) = V(i)) ∧ (∀j : b < j ≤ N : v(j) = V(j)) }
fproc
```

Caso directo: `a ≥ b`

`v[a..b]` tiene a lo sumo un elemento y ya está ordenado.

Caso recursivo: a < b

`v[a..b]` tiene al menos dos elementos. Calculamos `m = (a+b) div 2` y usamos llamadas recursivas para ordenar `v[a..m]` y `v[m+1..b]`. A continuación, efectuamos una llamada – *mezcla*(v, a, m, b)– a un procedimiento auxiliar cuyo efecto es mezclar `v[a..m]` y `v[m+1..b]`, dejando ordenado `v[a..b]`, y sin alterar el resto de *v*.

Indicación: El procedimiento *mezcla* necesita usar espacio auxiliar, aparte del ocupado por el propio *v*. Este espacio puede venir dado por otro vector.

98. Deriva una función recursiva final que calcule el máximo común divisor de dos números enteros positivos dados.

99. Deriva una función recursiva simple *dosFib* que satisfaga la siguiente especificación pre/post:

```
func dosFib( n : Nat ) dev r, s : Nat;
{ P₀ : n = N }
{ Q₀ : n = N ∧ r = fib(n) ∧ s = fib(n+1) }
ffunc
```

En la postcondición, `fib(n)` y `fib(n+1)` representan los números que ocupan los lugares n y n+1 en la sucesión de Fibonacci, para la cual suponemos la definición recursiva habitual en matemáticas.

100. Deriva una función recursiva que calcule el número combinatorio $\binom{n}{m}$ a partir de los datos `m, n: Nat`. Usa la recurrencia siguiente:

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m} \qquad \text{siendo } 0 < m < n$$

101. Una palabra se llama *palíndroma* si la sucesión de sus letras no varía al invertir el orden. Especifica y deriva una función recursiva final que decida si una palabra dada, representada como vector de caracteres, es o no palíndroma.

102. El problema de las torres de Hanoi consiste en trasladar una torre de *n* discos desde la varilla *ini* a la varilla *fin*, con ayuda de la varilla *aux*. Inicialmente, los *n* discos son de diferentes tamaños y están apilados de mayor a menor, con el más grande en la base. En ningún momento se permite que un disco repose sobre otro menor que él. Los movimientos permitidos consisten en desplazar el disco situado en la cima de una de las varillas a la cima de otra, respetando la condición anterior. Construye un procedimiento recursivo *hanoi* tal que la llamada *hanoi*(n, ini, fin, aux) produzca el efecto de escribir una serie de movimientos que represente una solución del problema de Hanoi. Supón disponible un procedimiento *movimiento*(i,j), cuyo efecto es escribir "Movimiento de la varilla *i* a la varilla *j*".

†103. La siguiente función recursiva se conoce como *función de Ackermann*:

```
func ack( m, n : Nat ) dev r : Nat;
inicio
  si
              m = 0 → r := n+1
  □ m > 0 AND n = 0 → r := ack(m-1, 1)
  □ m > 0 AND n > 0 → r := ack(m-1, ack(m,n-1))
  fsi;
  dev r
ffunc
```

(a) Explica por qué motivos la definición de *ack* no se ajusta a los esquemas de definición recursiva que hemos estudiado hasta ahora.
(b) Demuestra la terminación de *ack* usando un orden bien fundamentado conveniente, definido sobre $\aleph \times \aleph$.

Análisis de algoritmos recursivos

104. En cada uno de los casos que siguen, plantea una ley de recurrencia para la función T(n) que mide el tiempo de ejecución del algoritmo en el caso peor, y usa el método de desplegado para resolver la recurrencia.

(a) Función *fact* (ejercicio 79).
(b) Función *acuFact* (ejercicio 82).
(c) Funciones *pot* y *pot'* (ejercicio 85).
(d) Función *log* (ejercicio 91).
(e) Función *sumaVec* (ejercicio 94).
(f) Función *buscaBin* (ejercicio 95).
*(g) Procedimiento de ordenación *mergeSort* (ejercicio 97).
*(h) Procedimiento *hanoi* (ejercicio 102).

105. Aplica las reglas de análisis para dos tipos comunes de recurrencia a los algoritmos recursivos del ejercicio anterior. En cada caso, deberás determinar si el tamaño de los datos del problema decrece por sustracción o por división, así como los parámetros relevantes para el análisis.

***106.** En cada caso, calcula a partir de las recurrencias el orden de magnitud de T(n). Hazlo aplicando las reglas de análisis para dos tipos comunes de recurrencia.

(a) $T(1) = c_1$; $T(n) = 4 \cdot T(n/2) + n$, si $n > 1$
(b) $T(1) = c_1$; $T(n) = 4 \cdot T(n/2) + n^2$, si $n > 1$

(c) $T(1) = c_1$; $T(n) = 4 \cdot T(n/2) + n^3$, si $n > 1$

†107. Usa el método de desplegado para estimar el orden de magnitud de T(n), suponiendo que T obedezca la siguiente recurrencia:

$$T(1) = 1;\ T(n) = 2 \cdot T(n/2) + n \cdot \log n, \text{ si } n > 1$$

¿Pueden aplicarse en este caso las reglas de análisis para dos tipos comunes de recurrencia? ¿Por qué?

†108. Analiza la complejidad en tiempo del procedimiento de ordenación *quickSort* (ejercicio 96), distinguiendo dos casos:

(a) Tiempo de ejecución en el caso peor.

(b) Tiempo de ejecución bajo el supuesto de que las sucesivas particiones realizadas por el procedimiento auxiliar *partición* produzcan siempre dos partes de igual tamaño.

109. Supón un procedimiento recursivo de ordenación de vectores, llamado *badMergeSort*, que sea idéntico a *meregeSort* con la única diferencia de que el procedimiento auxiliar *mezcla* usado por *badMergeSort* opera en tiempo $O(n^2)$. Demuestra que bajo este supuesto el tiempo de ejecución de *badMergeSort* pasa a ser $O(n^2)$, frente a $O(n \cdot \log n)$ de *mergeSort*.

†110. Considera de nuevo la sucesión de Fibonacci, definida en el ejercicio 83. Sean φ y $\hat{\varphi}$ las dos raíces de la ecuación $x^2 - x - 1 = 0$, es decir:

$$\varphi = \frac{1+\sqrt{5}}{2} \qquad \hat{\varphi} = \frac{1-\sqrt{5}}{2}$$

Demuestra por inducción sobre *n* que $fib(n) = \frac{1}{\sqrt{5}} \cdot \left(\varphi^n - \hat{\varphi}^n\right)$

Pista: Usa que $\varphi^2 = \varphi + 1$ y $\hat{\varphi}^2 = \hat{\varphi} + 1$

†111. Considera la función doblemente recursiva *fib* del ejercicio 83, que calcula números de Fibonacci aplicando ingenuamente la definición matemática de la sucesión de Fibonacci. Supón que tomemos el propio valor numérico *n* como tamaño del dato *n*, y que definimos T(n) como el número de veces que se ejecuta una orden **dev** de devolución de resultado en el cómputo activado por la llamada inicial *fib(n)*.

(a) Plantea una ley de recurrencia para T(n) y demuestra por inducción sobre *n* que $T(n) = 2 \cdot fib(n+1) - 1$.

(b) Demuestra por inducción sobre *n* que $\varphi^{n-2} \leq fib(n) \leq \varphi^{n-1}$ se cumple para todo $n \geq 2$. Concluye que T(n) es $O(\varphi^n)$.

112. Aplicando las reglas de análisis para dos tipos comunes de recurrencia, demuestra que la función recursiva simple *dosFib* del ejercicio 99 consume tiempo $O(n)$. Compara con el resultado del ejercicio anterior.

Eliminación de la recursión final

113. Aplica la transformación de función recursiva final a función iterativa sobre las siguientes funciones:

(a) Función *acuFact* (ejercicio 82).

(b) Función *mcd* (ejercicio 98).

(c) Función *buscaBin* (ejercicio 95).

Técnicas de generalización

114. La siguiente especificación corresponde a una función *prodEsc* que calcula el producto escalar de dos vectores. Especifica una generalización *prodEsc'* con un parámetro de entrada adicional, de manera que *prodEsc'* admita un algoritmo recursivo.

```
func prodEsc( u, v : Vector[1..N] de Ent ) dev p : Ent;
{ P₀ : cierto }
{ Q₀ : p = ∑ i : 1 ≤ i ≤ N : u(i)*v(i) }
ffunc
```

115. Comprueba que *acuFact* (ejercicio 82) es una generalización de *fact* (ejercicio 79). Observa que el parámetro de entrada adicional de *acuFact* actúa como acumulador, posibilitando un algoritmo recursivo final.

116. Comprueba que *dosFib* (ejercicio 99) es una generalización de *fib* (ejercicio 83). En este caso, *dosFib* tiene un parámetro de salida adicional, el cual permite un algoritmo recursivo simple más eficiente que la recursión doble de *fib*.

117. Usando un parámetro de salida adicional, especifica una generalización *dosCuca* de la función *cuca* del ejercicio 93, de manera que *dosCuca* admita un algoritmo recursivo simple.

†118. Comprueba que la función *combi'* especificada como sigue es una generalización de la función *combi* del ejercicio 100, y que *combi'* admite un algoritmo recursivo simple, más eficiente que la recursión dobre de *combi*.

```
func combi'( n, m : nat ) dev v : Vector[0..N] de Ent;
{ P₀ : 0 ≤ m ≤ n ∧ m ≤ N }
{ Q₀ : ∀ i : 0 ≤ i ≤ m : v(i) = (n i) }
```

Observa que el parámetro de salida de *combi'* es más general que el de *combi*.

119. Comprueba que *sumaCuad'* es una generalización de *sumaCuad*, y construye un algoritmo recursivo lineal para *sumaCuad*':

```
func sumaCuad( n : Nat ) dev s : Nat;
{ P₀ : cierto }
{ Q₀ : s = ∑ i : 0 ≤ i ≤ n : i*i }
ffunc

func sumaCuad'( n : Nat ) dev c, p, s : Nat
{ P₀ : cierto }
{ Q₀ : s = ∑ i : 0 ≤ i ≤ n : i*i ∧ c = n*n ∧ p = 2*n+1 }
ffunc
```

Observa que $n^2+2n+1 = (n+1)^2$. Gracias a esto, los dos parámetros de salida añadidos a *sumaCuad'* (*c* y *p*) permiten programar *sumaCuad'* sin necesidad de calcular n^2 en cada llamada recursiva, mejorándose así la eficiencia.

120. Comprueba que *procesaVec'* es una generalización de *procesaVec*, y construye un algoritmo recursivo final para *procesaVec'*:

```
func procesaVec( v : Vector[1..N] de Ent ) dev r : Ent;
{ P0 : cierto }
{ Q0 : r = ∑ i : 1 ≤ i ≤ N : 2^i*v(i) }
ffunc

func procesaVec'( n, s, p : Ent; v : Vector[1..N] de Ent ) dev r : Ent;
{ P0 : 1 ≤ n ≤ N+1 ∧ p = 2^n }
{ Q0 : r = s + ∑ i : n ≤ i ≤ N : 2^i*v(i) }
ffunc
```

Observa el papel que juega cada uno de los tres parámetros adicionales de *procesaVec'*: *n* posibilita la recursión; *s* acumula un resultado parcial; y *p* mantiene una potencia de 2 para evitar su cálculo explícito en cada llamada recursiva.

Técnicas de plegado-desplegado

121. Aplica la técnica de plegado-desplegado para transformar la función recursiva lineal *fact* del ejercicio 79 en la función recursiva final *acuFact* del ejercicio 82.

122. Comprueba que *prod'* es una generalización de *prod*, y construye un algoritmo recursivo final para *prod'* usando plegado-desplegado:

```
func prod ( x, y : Nat ) dev r : Nat;
{ P0 : cierto }
inicio
  si  x = 0                → r := 0
  □   (x>0 AND par(x))       → r := prod(x div 2, y+y )
  □   (x>0 AND NOT par(x)) → r := y + prod(x div 2, y+y)
  fsi
{ Q0 : r = x * y }
  dev r
ffunc

func prod' ( a, x, y : Nat ) dev r : Nat;
{ P0 : cierto }
{ Q0 : r = a + x*y }
ffunc
```

123. Aplica plegado-desplegado para transformar en funciones recursivas finales las siguientes funciones recursivas simples:

(a) La función *pot'* del ejercicio 85(b).

(b) La función *prodEsc'* del ejercicio 114.

†124. Todas las transformaciones de los ejercicios 121, 122 y 123 se ajustan al siguiente esquema:

```
func f(x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P(x⃗ ) }
inicio
  si
     d(x⃗ ) → y⃗ := g(x⃗ )
  □ ¬d(x⃗ ) → y⃗ := h(x⃗ ) ⊕ f(s(x⃗ ))
  fsi
{ Q(x⃗ , y⃗ ) }
  dev y⃗
ffunc

func F (a⃗ : σ⃗ ; x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P(x⃗ ) }
{ y⃗ = a⃗ ⊕ f(x⃗ ) }
ffunc
```

Siendo $\oplus$ una operación asociativa y con elemento neutro $\vec{0}$. Comprueba que en este esquema general *F* es una generalización de *f*, y deriva un algoritmo recursivo final para *F* usando plegado-desplegado.

125. Comprueba que *bin'* es una generalización de *bin*, y deriva un algoritmo recursivo final para *bin'* usando plegado-desplegado. Observa que *bin* es la función recursiva simple del ejercicio 92.

```
func bin( n : Nat ) dev r : Nat;
{ P₀ : cierto }
inicio
  si
    n < 2 → r := n
  □ n ≥ 2 → r := 10 * bin(n div 2) + (n mod 2)
  fsi
{ Q₀ : r = ∑ i : Nat : ((n div 2^i) mod 2) * 10^i }
  dev r
ffunc
```

```
func bin'( s, p, n : Nat ) dev r : Nat;
{ P₀ : cierto }
{ Q₀ : r = s + p * bin(n) }
ffunc
```

126. Considera la función *cambioBase* especificada como sigue:

```
func cambioBase( b, n : Nat ) dev r : Nat;
{ P₀ : 2 ≤ b ≤ 9 }
{ Q₀ : r = ∑ i : Nat : ((n div bⁱ) mod b) * 10ⁱ }
ffunc
```

donde la postcondición quiere expresar que los dígitos de la representación decimal de *r* deben coincidir con los dígitos de *n* en base *b*. Por ejemplo, debe cumplirse *cambioBase*(2,9) = 1001.

(a) Construye un algoritmo recursivo lineal para *cambioBase*.
(b) Usando una técnica de plegado-desplegado similar a la del ejercicio 125, construye una generalización recursiva final *cambioBase'* de *cambioBase*.

†127. Las transformaciones de los ejercicios 125 y 126 se ajustan al siguiente esquema:

```
func f( x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P( x⃗ ) }
inicio
  si
     d( x⃗ ) → y⃗ := g( x⃗ )
  □ ¬d( x⃗ ) → y⃗ := h( x⃗ ) ⊕ (k( x⃗ ) ⊗ f(s( x⃗ )))
  fsi
{ Q( x⃗ , y⃗ ) }
  dev y⃗
ffunc

func F ( a⃗ : σ⃗ ; b⃗ : σ⃗ ; x⃗ : τ⃗ ) dev y⃗ : σ⃗ ;
{ P( x⃗ ) }
{ y⃗ = a⃗ ⊕ (b⃗ ⊗ f( x⃗ )) }
ffunc
```

Siendo ⊕, ⊗ operaciones asociativas con elementos neutros $\vec{0}$ y $\vec{1}$, respectivamente, y tales que ⊗ sea distributiva con respecto a ⊕. Comprueba que en este esquema general *F* es una generalización de *f*, y deriva un algoritmo recursivo final para *F* usando plegado-desplegado.

128. Comprueba que *fiboGen'* es una generalización de *fiboGen*, y deriva un algoritmo recursivo final para *fiboGen'* usando plegado-desplegado.

```
func fiboGen( n : Nat ) dev r, s : Nat;
{ P₀ : cierto }
func combina( u, v : nat ) dev u', v' : Nat;
inicio
  <u', v'> := <v, u+v>
dev <u', v'>
ffunc
var
  r', s' : Nat;
inicio
  si
    n = 0 → <r, s> := <0, 1>
  □ n > 0 → <r',s'> := fiboGen(n-1)
             <r, s> := combina( r', s');
  fsi
{ Q₀ : r = fib(n) ∧ s = fib(n+1) }
  dev <r, s>
ffunc

func fiboGen'( n, r', s' : Nat ) dev r, s : Nat;
{ P₀: cierto }
{ <r, s> = combina^n(r', s') }   % indica aplicar combina n veces
ffunc
```

129. Aplica una técnica de plegado-desplegado similar a la del ejercicio anterior para transformar en funciones recursivas finales las siguientes funciones recursivas simples:

(a) La función *dosCuca* del ejercicio 117.
(b) La función *sumaCuad'* del ejercicio 119.

†130. Las transformaciones de los ejercicios 128 y 129 se ajustan al esquema que sigue. Comprueba que F es una generalización de f, y deriva un algoritmo recursivo final para F usando plegado-desplegado.

```
func f(x⃗ : τ⃗ ; a⃗ : τ⃗ ') dev y⃗ : σ⃗ ;
{ P(x⃗ ) }
inicio
  si
     d(x⃗ ) → y⃗ := g(a⃗ )
  □ ¬d(x⃗ ) → y⃗ := h(f(s(x⃗ ),a⃗ ))
  fsi
{ Q(x⃗ , a⃗ , y⃗ ) }
  dev y⃗
ffunc
```

```
func F(x⃗ : τ⃗ ; b⃗ : σ) dev y⃗ : σ⃗ ;
{ P(x⃗) }
{ y⃗ = h^n(x⃗)(b⃗) }      % n(x) indica el menor n tal que d(s^n(x))
ffunc
```

CAPÍTULO 3
TIPOS ABSTRACTOS DE DATOS

3.1 Introducción a la programación con tipos abstractos de datos

3.1.1 La abstracción como metodología de resolución de problemas

Los problemas reales tienden a ser complejos porque involucran demasiados detalles, y resulta imposible considerar todas las posibles interacciones a la vez. Hay estudios en Psicología que afirman que en la memoria a corto plazo sólo es posible almacenar 7 elementos distintos.

Una abstracción es un modelo simplificado de un problema, donde se contemplan los aspectos de un determinado nivel, ignorándose los restantes.

Ejemplo. Para explicar a un granjero que no ha ido nunca a la ciudad cómo ir al banco en su tractor, conviene describir por separado:

- instrucciones para encontrar el banco
- instrucciones para conducir por carreteras y calles

Ejemplo. Para indicarle a un robot cómo cambiar la rueda de un coche podemos utilizar una descripción con distintos niveles de abstracción:

- Bajar del coche
  - Tirar de la manivela
  - Empujar la puerta
  - Levantarse del asiento
- Ir al maletero
- Abrir el maletero
- Sacar la rueda de repuesto
- …

Ejemplo. Para probar la corrección de un bucle

- Demostramos que hay un invariante se cumple antes y después de todas las iteraciones
  - Obtenemos el invariante
  - Demostramos que la precondición implica al invariante
  - Demostramos que el invariante implica la definición de la condición de repetición
  - Demostramos que partiendo de un estado que cumple el invariante y la condición de repetición y ejecutando el cuerpo del bucle se llega a un estado que cumple el invariante
- Demostramos que el bucle avanza
- Demostramos que al final de la ejecución del bucle se cumple la postcondición

El razonar en términos de abstracciones conlleva una serie de ventajas:

— La resolución de los problemas se simplifica

— Las soluciones son más claras y resulta más sencillo razonar sobre su corrección

— Es más fácil adaptar las soluciones a otros problemas

— Pueden obtenerse soluciones de utilidad más general (por ej. las instrucciones para llegar al banco, sirven también para peatones o ciclistas)

3.1.2 La abstracción como metodología de programación

La programación es un proceso de resolución de problemas y como tal también se beneficia del uso de abstracciones.

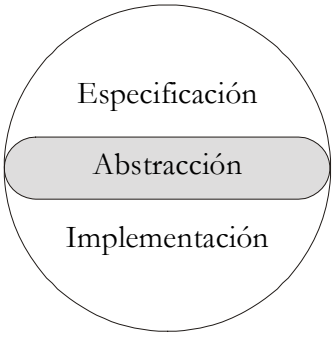
La evolución de los lenguajes de programación muestra una tendencia a incluir mecanismos de abstracción cada vez de más alto nivel. El ensamblador es una abstracción del lenguaje máquina, los lenguajes de alto nivel son una abstracción del ensamblador.

Dentro de los lenguajes de programación de alto nivel existen dos formas fundamentales de abstracción

La abstracción funcional

Son las funciones y los procedimientos. Consiste básicamente en reunir un conjunto de sentencias que realizan una determinada operación sobre unos datos y "darles un nombre" y una interfaz que las abstrae. De esta forma se puede utilizar ese conjunto de sentencias como si fuese una operación definida en el propio lenguaje, abstrayéndonos de los detalles de su implementación. Por ejemplo, para utilizar un procedimiento de ordenación no necesita saber qué método de ordenación implementa.

Para poder poner a disposición de otros –o de mi mismo, pasado el tiempo– una abstracción funcional, tengo que ser capaz de describirla de la manera más clara y precisa que sea posible, sin incluir detalles de la implementación. Eso lo consigo mediante una especificación. La abstracción es como una barrera que deja a un lado la especificación, con la que los clientes de la abstracción pueden razonar, y a otro la implementación:



La abstracción funcional es la base del método de diseño descendente, del cuál es un ejemplo "las instrucciones que han de darse a un robot para que cambie una rueda".

La abstracción de datos

Esta idea es posterior a la abstracción funcional –formulada por Guttag y otros hacia 1974–. Se trata de separar el comportamiento deseado de un tipo de datos de los detalles relativos a la representación y el manejo de los valores de ese tipo. Al igual que la abstracción funcional se puede ver como una forma de añadir operaciones a un lenguaje, la abstracción de datos se puede ver como una forma de añadir tipos al lenguaje.

Datos históricos:

El concepto de *tipo de datos* (*data types*) surgió con la aparición de los lenguajes de programación estructurada (como Pascal, derivado de Algol), en la década de los 60.

> O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare. *Structured Programming.* Academic Press 1972.
>
> N. Wirth. *The Programming Language PASCAL.* Acta Informatica 1, 1971, pp. 35-63.

El concepto de *tipo abstracto de datos* (*abstract data type*) fue introducido a mediados de la década de los 70. Pioneros: S. N. Zilles, J. V. Guttag y grupo ADJ (J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright).

> [ADJ78] J. A. Goguen, J. W. Thatcher, E. G. Wagner. "An Initial Algebra Approach to the Specification, Correctnes and Implementation of Abstract Data Types". En *Current Trends in Programming Methodology*, vol. IV, Prentice Hall, 1978.

3.1.3 Los tipos predefinidos como TADs

Los primeros ejemplos que podemos considerar son los tipos predefinidos de los lenguajes de programación de alto nivel:

— Enteros. Tenemos acceso a las representaciones abstractas de los números –en base 10–, y a las operaciones aritméticas. Sabemos qué leyes algebraicas obedecen las operaciones. La representación interna de los números –binaria en complemento a 2– y la implementación de las operaciones al nivel de la máquina –algoritmos de aritmética en complemento a 2 implementados en el procesador– son irrelevantes al nivel del usuario.

— Vectores. Disponemos de operaciones para consultar y modificar los valores almacenados en un vector. Estas operaciones obedecen a leyes algebraicas que pueden formularse con precisión –si almaceno *v* en la posición *i* de un vector, y luego consulto la posición *i* obtendré el valor *v*–, como veremos más adelante. Los detalles de representación interna y almacenamiento de vectores en la memoria de una máquina son irrelevantes para el usuario.

En cambio, los tipos de datos definidos por los programadores en la mayoría de los lenguajes de programación no son abstractos, porque:

— la representación interna de los valores del tipo es visible,

— como consecuencia, el usuario puede realizar con valores del tipo operaciones absurdas con respecto a la semántica pretendida.

Por ejemplo, si queremos definir un tipo de datos para representar fechas, podemos utilizar una representación como esta:

```
tipo
  Día = [1..31];
  Mes = enero | febrero | … | diciembre;
  Año = Ent;
  Fecha = reg
    día : Día;
    mes : Mes;
    año : Año
  freg;
```

Aparecen aquí por primera vez algunas definiciones de tipos. Los tipos que consideramos son "los de Pascal traducidos al castellano".

El hecho de que el usuario tenga libre acceso a la representación del tipo de datos, permite que se realicen operaciones semánticamente absurdas, como:

```
con fecha hacer
  mes := febrero;
  día := 30;
fcon
```

Aparece aquí por primera vez la sentencia **`con … fcon`**, que es el equivalente a la sentencia `with` de Pascal.

3.1.4 Ejemplos de especificación informal de TADs

Al igual que al manejar una abstracción funcional, lo que le proporcionamos al usuario es la especificación de la operación, al abstraer datos también debemos proporcionar una especificación con la que los usuarios puedan trabajar. Informalmente, un tipo abstracto de datos (TAD) se define especificando:

— el dominio de valores del tipo

— las operaciones del tipo –que pueden hacer referencia a otros tipos–

— el comportamiento esperado de las operaciones

Nótese que en la abstracción de datos se incluye también la abstracción funcional de las operaciones del TAD.

Todo ello sin hacer referencia a ninguna representación concreta de los datos. El usuario del TAD sólo necesita conocer la especificación.

Un ejemplo de especificación informal del tipo abstracto de datos *fecha*:

Dominio: las fechas desde el 1/1/1 –no hubo año cero, por lo que el siglo XXI empieza el 1/1/2001–. Nótese que omitimos cualquier información sobre la estructura interna con la que se representan las fechas.

Operaciones:

```
func NuevaFecha ( d : Día; m : Mes; a : Año) dev f : Fecha;
{ P₀ : d/m/a cumplen las restricciones de una fecha, según el calendario }
{ Q₀ : f representa a la fecha d/m/a }
ffunc;

func distancia ( f1, f2 : Fecha ) dev d : Ent;
{ P₀ : }
{ Q₀ : d es la distancia, medida en número de días, entre f1 y f2 }
ffunc;

func suma ( f : Fecha; d : Ent ) dev g : Fecha;
{ P₀ : }
{ Q₀ : G es la fecha resultante de sumar d días a la fecha F }
ffunc;

¡Cuidado! d puede ser un número negativo de forma que g no sea una fecha
posterior al 1/1/0

func día ( f : Fecha ) dev d : Día;
{ P₀ : }
{ Q₀ : d es el día de la fecha f }
ffunc;

func mes ( f : Fecha ) dev m : Mes;
{ P₀ : }
{ Q₀ : m es el mes de la fecha f }
ffunc;

func año ( f : Fecha ) dev a : Año;
{ P₀ : }
{ Q₀ : a es el año de la fecha f }
ffunc;

func díaSemana ( f : Fecha ) dev d : DíaSemana;
{ P₀ : }
{ Q₀ : d es el día de la semana correspondiente a la fecha f }
ffunc;
```

```
func primer ( d : DíaSemana; m : Mes; a : Año ) dev f : Fecha;
{ P0 : }
{ Q0 : f es la primera fecha del mes m del año a cuyo día de la semana es d
}
ffunc;
```

Por ejemplo, con la función `primer` podríamos calcular cuando cae el primer martes de noviembre de 1871:

```
primer( martes, noviembre, 1871 )
```

Una especificación del TAD *fecha* debería incluir además axiomas para las operaciones, que omitimos aquí.

3.1.5 Implementación de TADs: privacidad y protección

La implementación de un TAD queda separada de su especificación y no es competencia del usuario, sino del implementador –que puede ser el mismo, pero que una vez implementado el TAD se puede olvidar de sus detalles–. Consiste en:

— Definir una representación de los valores del TAD con ayuda de otros tipos ya disponibles.

— Definir las operaciones del TAD como funciones o procedimientos, usando la representación elegida, y de modo que se satisfaga la especificación.

Para hacer posible esta separación entre especificación e implementación, es necesario que nuestro lenguaje incluya mecanismos de

— **Privacidad**: la representación interna está oculta y es invisible para los usuarios.

— **Protección**: el tipo sólo puede usarse a través de sus operaciones; otros accesos incontrolados se hacen imposibles.

La característica importante es la protección, pues aunque un usuario pueda conocer el tipo representante de un TAD, no resulta un problema si sólo puede acceder a los valores a traves de las operaciones públicas del TAD.

Por desgracia, muchos lenguajes de programación no incluyen estos mecanismos y la protección del TAD se convierte entonces en una cuestión de disciplina del programador: sólo deben utilizar los valores del TAD a través de las operaciones públicas que éste haya definido, aunque el lenguaje permita conocer y acceder directamente a la representación interna de los datos.

En general cualquier TAD admite varias representaciones posibles.

Por ejemplo, una secuencia de números de longitud variable se puede implementar como un registro con un vector y un campo longitud o como un vector con una marca al final. En cualquiera de los dos casos, con tipo así definido en Pascal, el programador podría consultar por un valor de la secuencia que esté más allá del límite indicado por la longitud o por la marca; algo que no tiene sentido.

Cuando hay varias representaciones posibles, normalmente una representación concreta facilita unas determinadas operaciones y dificulta otras, con respecto a una representación alternativa.

Por ejemplo el TAD *fecha* que vimos antes:

— Podemos representar fechas con los registros que vimos antes.

  Con esta representa resulta trivial implementar *NuevaFecha*, *día*, *mes* y *año*. Mientras que sería más difícil implementar el resto de las operaciones

— Podríamos representar la fechas como el número de días transcurridos desde el 1 de Enero de 1900 –de hecho en la mayoría de los sistemas es así como se hace, con lo cual llegará un momento en que se alcanzará el máximo del tipo de números seleccionado, y la representación "dará la vuelta", algo que está relacionado con el problema del año 2000– Así es mucho más sencillo calcular el número de días transcurridos entre dos fechas dadas, pero resulta más difícil construir una fecha a partir del día/mes/año.

La elección de una implementación concreta para un TAD puede posponerse o variarse según convenga, sin afectar a los programas ya realizados o en proceso de diseño –ayudando así a la división de tareas– que usen el TAD.

3.1.6 Ventajas de la programación con TADs

El uso de TADs ayuda en:

— El diseño descendente

— El diseño de los programas

— La reutilización

Mejoras en el diseño descendente

El método de diseño descendente se ve potenciado de dos maneras distintas:

— El diseño se hace más abstracto. Ya no dependemos exclusivamente de los tipos concretos que nos ofrezca un lenguaje de programación determinado. Podemos especificar TADs adecuados y posponer los detalles de su implementación.

— El diseño puede incluir refinamiento de tipos. Al especificar un TAD necesario para el diseño, podemos dejar incompletos detalles que se completen en etapas posteriores. Por ejemplo, la decisión de cuáles deben ser las operaciones del tipo puede irse completando a lo largo del diseño. De esta forma, se va *refinando* la definición del TAD.

Veamos un ejemplo ([Kingston 90], pp. 38-39).

Problema: dado un vector v de números enteros con índices entre 1 y N y un número k, $0 \leq k \leq N$, se trata de determinar los k números mayores que aparecen en el vector –nótese que no tienen que ser k números diferentes–.

Para resolver este problema necesitamos una estructura auxiliar donde vayamos almacenando los k mayores encontrados hasta ahora. Podríamos elegir una estructura de datos concreta –por ejemplo, un vector– e incluir su gestión dentro del propio algoritmo. Los inconvenientes de esta decisión:

— La lógica del algoritmo queda oscurecida por los detalles de la representación.

— La estructura de datos elegida a priori puede no ser la más eficiente, ya que para saber cuál es la elección más eficiente debemos saber cuáles son las operaciones necesarias sobre ella, y eso no lo sabremos hasta que hayamos diseñado el algoritmo que la utiliza.

La otra opción es elegir un TAD capaz de almacenar la información necesaria e ir viendo qué operaciones nos hacen falta sobre sus valores. En este problema podemos elegir como TAD los multiconjuntos. En pseudocódigo, el algoritmo quedaría:

```
m := { v(1), …, v(k) }
para i desde k+1 hasta N hacer
  min := minimo elemento de M
  si
    v(i) > min → cambiar min por v(i) en m
  □ v(i) ≤ min → seguir
fpara
```

Si suponemos disponible un TAD para los multiconjuntos de números enteros, con operaciones adecuadas, podemos diseñar la siguiente función que resuelve el problema:

```
func mayores ( k : Nat; v : Vector[1..N] de Ent ) dev m : MCjtoEnt;
{ P₀ : 1 ≤ k ≤ N ∧ N ≥ 1 }
var
  n : Ent;
inicio
  Vacío(m);
  n := 0;
  { Inv. I: m contiene los elementos de v[1..n] }
  it n ≠ k →
    Pon(v(n+1), m);
    n := n+1
  fit
  { Inv J: m contiene los k elementos mayores de v[1..n] }
  it n ≠ N →
    si v(n+1) > min( m )
      entonces
        quitaMin( m );
        Pon( v(n+1), m )
      sino
        seguir
    fsi
  fit
{ Q₀ : m contiene los k elementos mayores de v[1..N] }
  dev m
ffunc
```

Del algoritmo resultante podemos ver qué operaciones necesitamos en el TAD *multiconjunto*

```
func Vacío dev m : MCjtoEnt
{ P₀ : }
{ Q₀ : m representa ∅ }
ffunc

proc Pon( e x : Ent; es m : MCjtoEnt );
{ P₀ : m = M ∧ e = E }
{ Q₀ : e = E ∧ m = M ∪ {x} }
fproc

proc quitaMin( es m : MCjtoEnt );
{ P₀ : m = M ∧ m no es vacío }
{ Q₀ : m es M menos una copia del elemento mínimo de M }
fproc

func min ( m : MCjtoEnt ) dev r : Ent;
{ P₀ : m = M ∧ m no es vacío }
{ Q₀ : m = M ∧ r es el elemento mínimo de m }
ffunc
```

Pon y *quitaMin* se han implementado como procedimientos para evitar la copia del multiconjunto recibido como parámetro. Un multiconjunto se podría implementar como un vector de registros de dos componentes, una que indica el número almacenado y otra el número de apariciones de ese valor; además se podría optar entre mantenerlo o no ordenado con lo que se haría más eficiente la obtención del mínimo o las operaciones poner y quitar, respectivamente.

Se pueden encontrar infinidad de ejemplos como este donde es beneficioso diseñar en términos de un TAD.

Los TADs como apoyo a la programación modular

El desarrollo de programas grandes y complejos normalmente se realiza descomponiendo el programa en unidades menores llamadas **módulos**, cada uno de los cuales encapsula una colección de declaraciones en un ámbito de visibilidad cerrado y se comunica con otros módulos a través de una interfaz bien definida.

En un diseño modular es importante que los módulos se elijan adecuadamente en cuanto a su tamaño y conexiones mutuas. Por lo general siempre resulta adecuado diseñar como módulos las implementaciones de TADs, debido a su utilidad general, su tamaño mediano, y la simplicidad de la conexión con los módulos usuarios.

En Delphi las clases constituyen un mecanismo de modularización.

La implementación de los TADs como módulos está relacionado con las ideas básicas de la orientación a objetos, donde las clases encapsulan las definiciones de tipos de datos junto con las operaciones que los manejan. Utilizando criterios de ingeniería del software, la organización del diseño en torno a los datos resulta ventajoso frente a la organización en torno a las funciones, pues los datos –objetos– suelen ser más resistentes al cambio.

Datos históricos.

La metodología de diseño modular (*modular design*) surgió como extrapolación de la técnica de diseño con TADs a aplicaciones de gran dimensión.

Una obra clásica que trata en profundidad esta metodología es

> B. H. Liskov, J. V. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

3.1.7 Tipos abstractos de datos versus estructuras de datos

En este curso, entenderemos que una *estructura de datos* (*data structure*) es la representación de un TAD mediante una combinación adecuada de los tipos de datos y constructoras de tipos disponibles en los lenguajes de programación habituales (booleanos, enteros, reales, caracteres, vectores, registros, punteros, etc.) Es decir, concebimos las estructuras de datos como **estructuras de implementación**.

Hay muchas estructuras de datos clásicas que corresponden a combinaciones interesantes de los tipos básicos: las estructuras lineales, los árboles, las tablas, los grafos, etc. Estas estructuras clásicas pueden entenderse como las implementaciones de ciertos TADs básicos asociados a ellas. Además, las combinaciones de estas estructuras sirven para implementar muchos otros TADs.

Existe la tendencia entre los alumnos a identificar el estudio de los TADs con el estudio de estas estructuras de datos clásicas. Y a considerar que los TADs asociados con ellas son entidades monolíticas cuyas especificaciones no pueden ser modificadas porque entonces "ya no sería el TAD *x*". Estos TADs representan colecciones de datos a los que se accede de un cierto modo; como tales son muy interesantes porque son muchos los programas que manejan colecciones de datos, y la literatura se ha encargado de identificar agrupaciones interesantes de operaciones de acceso, e investigar posibles implementaciones –representaciones concretas– que hagan eficientes dichas operaciones. Sin embargo, el concepto de TAD es más simple y más general a la vez, simplemente un TAD define un dominio de valores junto con las operaciones que permiten manejarlos, y esto es algo que puede variar de unas aplicaciones a otras, aún en lo que se refiere a las estructuras de datos clásicas.

3.2 Especificación algebraica de TADs

Un texto de referencia

> H. Elvig, B. Mahr. *Fundamentals of Algebraic Specification. Vol. 1*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.

Como ya hemos visto en el tema anterior es conveniente, desde el punto de vista de la abstracción, separar la especificación de la implementación.

Los componentes de la especificación:

— Dominio de valores abstracto

— Operaciones

— Axiomas

Los componentes de la implementación:

— Representación concreta de los valores –con estructuras de datos adecuadas–

— Funciones y procedimientos para las operaciones

— Ocultamiento de la representación y código internos.

Los ejemplos del tema anterior pueden sugerir la posibilidad de especificar un TAD dando un nombre a su dominio de valores, y dando nombres y especificaciones pre/post para funciones o procedimientos correspondientes a sus operaciones.

En este enfoque las especificaciones pre/post deberían reflejar los axiomas que deseamos que cumplan las operaciones del TAD.

Un enfoque diferente consiste en imaginar las operaciones de un TAD como análogas a las operaciones del álgebra y formular los axiomas que queramos exigirles por medio de *ecuaciones* –igualdades entre aplicaciones de las operaciones–. Este enfoque se llama *especificación algebraica*. (Un "álgebra" es un dominio de valores equipado con operaciones que cumplen axiomas algebraicos dados).

La principal diferencia con respecto al método de las pre y postcondiciones es que se usan ecuaciones en lugar de otros asertos más complicados. Esto tiene varias ventajas:

— El razonamiento con ecuaciones es relativamente sencillo y natural

— Las ecuaciones no sólo especifican las propiedades de las operaciones, sino que especifican también cómo construir valores de éste

— Las especificaciones basadas en ecuaciones suelen dar muchas pistas para la implementación

Una diferencia menor es la disparidad de notaciones.

3.2.1 Tipos, operaciones y ecuaciones

Una especificación algebraica consta fundamentalmente de tres componentes:

— Tipos: son nombres de dominios de valores. Entre ellos está siempre el *tipo principal* del TAD; pero puede haber también otros que se relacionen con éste, quizá pertenecientes a otros TADs especificados anteriormente. (En algunos libros a los tipos se les denomina *géneros* para hacer más hincapié en la diferencia entre dominio de valores y TAD que además está equipado con un conjunto de operaciones)

— Operaciones: deben ser funciones con un perfil asociado, que indique el tipo de cada uno de los argumentos y el tipo del resultado. En una especificación algebraica no se permiten funciones que devuelvan varios valores, ni tampoco procedimientos no funcionales. (En una implementación, como veremos, sí podrán usarse procedimientos.)

— Ecuaciones entre términos formados con las operaciones y variables de tipo adecuadas

Definimos la *signatura* de un TAD como los tipos que utiliza, junto con los nombres y los perfiles de las operaciones –sin incluir las ecuaciones–.

Algo de notación:

$\sum$: signatura de un TAD

τ, τ_i : tipos

$c : \rightarrow \tau$ operación constante de género τ

$f : \tau_1, \ldots, \tau_n \rightarrow s$ operación con perfil, donde los τ_i son los tipos de los argumentos y τ es el tipo del resultado

Ejemplos: BOOL y NAT

Vamos como primer ejemplo la especificación algebraica de los booleanos

```
tad BOOL
    tipo
       Bool

    operaciones
       Cierto, Falso : → Bool /* gen */
       (not ) : Bool → Bool   /* mod */
       ( and ), ( or ) : (Bool, Bool) → Bool /* mod */

    ecuaciones
       ∀ x : Bool
       not Cierto   = Falso
       not Falso    = Cierto
       Cierto and x = x
       Falso and x     = Falso
       Cierto or x     = Cierto
       Falso or x   = x
ftad
```

Nótese que sólo hacemos distinción de casos en uno de los argumentos. La idea intuitiva es que tenemos que escribir las ecuaciones de forma que reflejen el comportamiento de las operaciones, según el modelo mental que estamos intentando especificar.

Un ejemplo más complicado, donde aparece la cláusula `usa` para indicar que importamos otro TAD, de forma que todas las operaciones, tipos y ecuaciones de BOOL son visibles en NAT. Es como si cogiésemos la especificación de BOOL y la pegásemos en la especificación de NAT. En ocasiones puede ser necesario renombrar alguna de las operaciones del TAD usado para que no se colapsen con las operaciones que voy a definir en el TAD que lo usa.

```
tad NAT
    usa
       BOOL
```

```
tipo
   Nat
operaciones
   Cero : → Nat          /* gen */
   Suc _ : Nat → Nat     /* gen */
   ( + ), ( * ) : (Nat, Nat) → Nat  /* mod */
   ( == ), ( /= ) : (Nat, Nat) → Bool /* obs */
   ( ≤ ), ( ≥ ), ( < ), ( > ) : (Nat, Nat) → Bool  /* obs */
ecuaciones
   ∀ x, y : nat
   x + Cero        = x
   x + Suc(y)      = Suc(x+y)
   x * Cero           = Cero
   x * Suc(y)      = (x*y) + x
   Cero == Cero    = Cierto
   Cero == Suc(y)     = Falso
   Suc(x) == Cero     = Falso
   Suc(x) == Suc(y)   = x == y
   x /= y          = not( x == y)
   Cero ≤ y           = Cierto
   Suc(x) ≤ Cero      = Falso
   Suc(x) ≤ Suc(y) = x ≤ y
   x ≥ y           = y ≤ x
   x < y           = (x ≤ y) and (x /= y)
   x > y           = y < x
ftad
```

Nótese que las ecuaciones tienen forma de definiciones recursivas, definimos el comportamiento de la operación para el o los casos base y luego definimos una o más reglas recursivas que nos van acercando al caso base –a los términos generados–. Por ejemplo, para la suma tenemos como caso base

```
x + Cero = x
```

y como caso recursivo

```
Suc(x) + y = Suc(x+y)
```

de forma que el sumando de la izquierda se va acercando al caso base `Cero`.

Nótese la diferencia entre la operación de igualdad ==, y la igualdad algebraica entre términos. La operación de igualdad es una operación más que sirve para construir términos, mientras que las ecuaciones de la especificación indican equivalencias entre términos, por ejemplo que el término `Cero == Cero` es igual al término `Cierto`.

3.2.2 Términos de tipo τ: T_τ

Fijando un conjunto de variables X, y aplicando las operaciones del TAD podemos generar valores de distintos tipos a los que llamamos *términos*. Es decir, una signatura permite definir un

conjunto, típicamente infinito, de términos que se pueden construir utilizando las operaciones de la signatura. Nos va a interesar razonar sobre esos términos y expresar algunas propiedades que debe cumplir la especificación con respecto a ellos.

$T_{\Sigma,\tau}(X)$ conjunto de los términos de tipo τ con variables en X

Donde el conjunto de variables X hace referencia a las variables que hemos utilizado al escribir las ecuaciones.

que se define recursivamente como:

— $x \in X_\tau \Rightarrow x \in T_{\Sigma,\tau}(X)$

donde $X_\tau \subseteq X$

si *x* pertenece a las variables de tipo τ entonces pertenece al conjunto de términos de tipo τ con variables en X

— $c : \rightarrow \tau \Rightarrow c \in T_{\Sigma,\tau}(X)$

— $f : \tau_1, \ldots, \tau_n \rightarrow \tau \wedge t_i \in T_{\Sigma,\tau_i}(X)\ (1 \leq i \leq n) \Rightarrow f(t_1, \ldots, t_n) \in T_{\Sigma,\tau}(X)$

Convenios

— $T_{\Sigma,\tau}$ es $T_{\Sigma,\tau}(\varnothing)$. A los términos sin variables se les denomina *términos cerrados*

— Σ puede omitirse si se da por sabida, con lo que el conjunto de términos cerrados de tipo τ para una signatura que se da por sabida es T_τ

Ejemplos

En NAT hay dos tipos Ent y Bool, y se tiene

```
Suc(x) + Suc(Suc(y)) ∈ T_Ent(x,y)
Suc(Cero) + Suc(Suc(Cero)) ∈ T_Ent
Suc(x) == Cero ∈ T_Bool(x)
Suc(Cero) /= Cero ∈ T_Bool
```

3.2.3 Razonamiento ecuacional. T-equivalencia

Para saber qué quiere decir una especificación algebraica tenemos que determinar cuáles son las igualdades entre términos que son válidas de acuerdo con las ecuaciones de la especificación. Las ecuaciones de la especificación de un TAD pueden usarse para deducir de ellas otras ecuaciones, según las leyes de la lógica.

Sea T un TAD con tipo principal τ, dos términos `t, t': τ` se llaman T-*equivalentes* (en símbolos $t =_T t'$), si la ecuación `t = t'` se puede deducir a partir de los axiomas ecuacionales de T. (De un TAD que use a otro también se pueden deducir T-equivalencias entre términos de los tipos *importados*, sin embargo, como luego veremos, no se puede introducir ninguna equivalencia nueva, por lo que en un TAD basta con hacer referencia a las equivalencias entre términos del tipo principal.)

Las reglas del cálculo ecuacional que nos permiten determinar la igualdad algebraica entre términos son:

— Reflexividad

$$t = t$$

— Simetría

$$\frac{t = t'}{t' = t}$$

— Transitividad

$$\frac{t = t' \quad t' = t''}{t = t''}$$

— Congruencia

$$\frac{t_1 = t_1' \quad \ldots \quad t_n = t_n'}{f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}$$

— Axiomas ecuacionales

$$t\,[x_1/t_1, \ldots, x_n/t_n] = t'\,[x_1/t_1, \ldots, x_n/t_n]$$

si $t = t'$ es una ecuación de la especificación

Por ejemplo, en NAT se puede deducir la siguiente igualdad algebraica entre términos

```
Suc(Cero) * Suc(Cero) = Suc(Cero)
```

como podemos demostrar utilizando las ecuaciones de la especificación

```
          Suc(Cero) * Suc(Cero)
*.2   ==  (Suc(Cero) * Cero) + Suc(Cero)
*.1   ==  Cero + Suc(Cero)
+.2   ==  Suc(Cero + Cero)
+.1   ==  Suc( Cero )
```

donde, en todos los pasos, hemos utilizado la regla *axiomas ecuacionales* del cálculo.

Advertencia: Las notaciones

$$t = t' \quad y \quad t =_T t'$$

no deben confundirse

— t = t' indica una ecuación que puede cumplirse o no

— $t =_T t'$ indica que hemos demostrado a partir de la especificación que t = t' se cumple siempre

por ejemplo

— `Suc(x) = Suc(y)`

puede cumplirse para algunos valores de *x*, *y*

— $\texttt{Suc(x)} =_T \texttt{Suc(y)}$

es falso, pues de la especificación no se deduce que Suc(x) = Pred(y) se cumpla siempre

Diremos que $=_T$ es la *igualdad algebraica* entre términos inducida por la especificación de que se trate.

En cualquier especificación algebraica de un TAD se tiene que

— los términos sin variables denotan los valores del tipo –distintos términos pueden denotar al mismo valor–

— la T-equivalencia $=_T$ especifica las igualdades válidas entre valores del tipo

Las operaciones y ecuaciones del tipo deben elegirse de forma que estas dos condiciones se correspondan con *la intención del especificador*.

Hay que escribir las ecuaciones de forma que sea posible deducir todas las equivalencias que son válidas en el modelo en el que estamos pensando. Pero sin escribir demasiadas, para que sea posible deducir equivalencias *indeseables*.

3.2.4 Operaciones generadoras, modificadoras y observadoras

Para lograr el requisito de que el TAD se corresponda con la *intención del especificador* conviene seguir la siguiente metodología: una vez elegido el tipo principal τ, clasificamos las operaciones según el papel que queremos que jueguen en relación con el tipo principal, como:

— Generadoras (o constructoras)

Serán algunas operaciones con perfil

$\texttt{c} : \vec{\tau} \rightarrow \tau$

que estén pensadas para construir todos los valores de tipo τ

— Modificadoras

Serán las restantes operaciones con perfil

$\texttt{f} : \vec{\tau} \rightarrow \tau$

que no sean generadoras. Estas operaciones están pensadas para hacer cálculos que produzcan resultados de tipo τ.

— Observadoras

Serán algunas operaciones con perfil

$g : \vec{\tau} \to \tau'$ tal que $\tau' \mid \equiv \tau$; y algún $\tau_i \equiv \tau$

pensadas para obtener valores de otros géneros a partir de valores de tipo τ.

Volvemos sobre los ejemplos de BOOL y NAT y clasificamos las operaciones

```
Cierto, Falso : → Bool                          /* gen */
(not ) : Bool → Bool                              /* mod */
( and ), ( or ) : (Bool, Bool) → Bool             /* mod */
```

y para NAT

```
Cero : → Nat                                    /* gen */
Suc  : Nat → Nat                                  /* gen */
( + ), ( * ) : (Nat, Nat) → Nat                   /* mod */
( == ), ( /= ) : (Nat, Nat) → Bool                /* obs */
( ≤ ), ( ≥ ), ( < ), ( > ) : (Nat, Nat) → Bool      /* obs */
```

Dada esta clasificación de las operaciones, hay un tipo especialmente interesante de términos, los que sólo utilizan operaciones generadoras.

3.2.5 Términos generados: TG_τ

Dado un TAD T con tipo principal τ se llaman *términos generados* a aquellos términos de tipo τ que están construidos usando solamente operaciones generadoras de T (y términos generados de otros TADs usados por T, si es necesario). (No nos preocupamos por los términos de otros tipos que no sean τ pues esos términos deben ser equivalentes a términos generados en el TAD donde se define el correspondiente tipo.)

$TG_{\Sigma,\tau}(X) =_{def} \{ t \in T_{\Sigma,\tau}(X) \mid$ todas las operaciones usadas en t son generadoras $\}$

Al igual que en la notación de T, cuando $X = \varnothing$ ponemos $TG_{\Sigma,\tau}$, y omitimos $\sum$ si es sabida, con lo cual notamos TG_τ.

En BOOL los términos generados son

$TG_{Bool} = \{$ Cierto, Falso $\}$

En NAT son términos generados:

```
Suc(Suc(Cero))Suc(Suc(Suc(Cero)))     Suc(Suc(Cero))
```

y no son términos generados

```
Suc(Suc(Cero)) + Suc(Cero)    Suc(Cero) * Suc(Cero)
```

En general, en NAT son términos generados el conjunto

$$TG_{Nat} = \{ Cero \} \cup \{ Suc^n(Cero) \mid n > 0 \}$$

Ej. 141: Indica cuáles son los términos generados de los TADs BOOL, NAT y COMPLEJO.

La utilidad de distinguir el conjunto de términos generados dentro del conjunto de términos es poder establecer un subconjunto de términos que representen a todos los valores posibles de TAD, de forma que los términos no generados sean equivalentes a algún término generado. De esta forma nos bastará con razonar sobre los términos generados, pues el resto de términos se podrán reducir a ellos. La propiedad de que cualquier término se pueda reducir a uno generado es una propiedad que le debemos exigir a nuestras especificaciones.

3.2.6 Completitud suficiente de las generadoras

Dado el TAD T con tipo principal τ, se dice que el conjunto de operaciones generadoras de T es *suficientemente completo* si es cierta la siguiente condición: para todo término cerrado t de tipo τ debe existir un término cerrado y generado t' de tipo τ que sea T-equivalente a t

$$\forall\ t : t \in T_{\Sigma,\tau} : \exists\ t' : t' \in TG_{\Sigma,\tau} : t =_T t'$$

No nos preocupamos por los términos de otros tipos distintos de τ porque esos términos deben ser equivalentes a términos generados en el TAD donde se define su tipo. Esto ha de ser así para que el TAD usado quede protegido, como luego veremos.

La especificación de cualquier TAD siempre debe construirse de manera que el conjunto de operaciones generadoras elegido sea suficientemente completo.

Ejemplo: las generadoras de BOOL son suficientemente completas

Este tipo de demostraciones se hace por inducción sobre la estructura sintáctica de los términos –inducción estructural.

```
Base: t/Cierto

    Cierto =BOOL Cierto

Base: t/Falso

    Falso =BOOL Falso
Paso inductivo: t/not t1
```

```
                          not t1
            H.I.   =BOOL  not t1'

    t1'/Cierto

                          not Cierto
           not.1   =BOOL  Falso

    t1'/Falso

                          not Falso
           not.2   =BOOL  Cierto

Paso inductivo: t/t1 and t2

                          t1 and t2
            H.I.   =BOOL  t1' and t2'

    t1'/Cierto

                          Cierto and t2'
           and.1   =BOOL  t2'

    t1'/Falso

                          Falso and t2'
           and.2   =BOOL  Falso
```

Y de igual forma para or

Ejemplo: las generadoras de NAT son suficientemente completas

```
Base: t/Cero

    Cero =NAT Cero

Paso inductivo: t/Suc(t1)

                          Suc(t1)
            H.I.   =NAT   Suc(t1')

Paso inductivo: t/t1+t2

                          t1 + t2
            H.I.   =NAT   t1' + t2'
          Lema 1   =NAT   t''
```

El lema 1 dice que dados dos términos generados t1' y t2' existe otro término generado t'' tal que t1'+t2' $=_{NAT}$ t''. Lo demostraremos más abajo.

```
Paso inductivo: t/t1*t2

                          t1 * t2
```

```
            H.I.   =NAT t1' * t2'
          Lema 2   =NAT t''
```

El lema 2 dice que dados dos términos generados t1' y t2' existe otro término generado t'' tal que t1'*t2' $=_{NAT}$ t''. Lo demostraremos más abajo.

Lema 1

Base: t2'/Cero

```
                     t1' + Cero
         +.1   =NAT t1'
```

Paso inductivo: t2'/Suc(t3')

```
                     t1' + Suc(t3')
         +.2   =NAT Suc(t1' + t3')
        H.I.   =NAT Suc(t4')
```

Lema 2

Base: t2'/Cero

```
                     t1' * Cero
         *.1   =NAT Cero
```

Paso inductivo: t2'/Suc(t3')

```
                     t1' * Suc(t3')
         *.2   =NAT t1' * t3' + t1'
        H.I.   =NAT t4' + t1'
      Lema 1   =NAT t5'
```

Para los términos de tipo Bool no lo demostramos porque es un tipo que se define en otro TAD, lo que tendremos que demostrar es que el TAD usado queda protegido.

3.2.7 Razonamiento inductivo

El mismo tipo de demostraciones que hemos utilizado para probar la completitud suficiente de las generadoras se puede utilizar para demostrar otras ecuaciones. Hay ecuaciones cuya validez no se puede obtener simplemente con el cálculo ecuacional, sino que hay que hacer suposiciones sobre las posibles formas –sintácticas– de los términos. Aquí nos aprovechamos de la completitud suficiente de las generadoras, para así razonar solamente sobre la estructura sintáctica de los términos generados.

Ejemplo (ej. 143): Cero + y = y

Base: y/Cero

```
                     Cero + Cero
         +.1   =NAT Cero
```

Paso inductivo: y/Suc(y1)

```
                     Cero + Suc(y1)
         +.2   =NAT Suc(Cero + y1)
        H.I.   =NAT Suc(y1)
```

3.2.8 Diferentes modelos de un TAD

Una especificación algebraica define las condiciones de una familia de álgebras, aquellas que le pueden servir como modelo. Aunque nosotros cuando elegimos el nombre de las constructoras y de las operaciones ya estamos pensando en un cierto modelo y es por ello que les asignamos nombres significativos dentro de ese modelo, eso no implica que sea el único modelo posible. Un modelo de una especificación algebraica debe:

— Incluir un conjunto de valores para cada tipo utilizado en la especificación –nótese que en la especificación no se dice nada sobre la cardinalidad de ese conjunto. De hecho un álgebra donde el dominio de valores tenga un solo elemento es modelo de cualquier especificación algebraica que no tenga operaciones parciales–

— Por cada operación que aparezca en la especificación, debe incluir una función con el mismo perfil –el nombre puede ser distinto–. No puede incluir más funciones.

— Las funciones asociadas con las operaciones deben cumplir las ecuaciones de la especificación.

Ejemplo: modelo no estándar de BOOL

(ej. 145) Construye un modelo de BOOL tal que el soporte del tipo *Bool* sea un conjunto de tres elementos: *Cierto*, *Falso* y *Nodef* (que quiere representar un valor booleano "indefinido"). Interpreta las operaciones *not*, (*and*), y (*or*) en este modelo de manera que las ecuaciones de BOOL se cumplan. Compara con el *modelo de términos* de BOOL.

$A_{Bool} = \{ c, f, \bot \}$

$Cierto_A : \rightarrow A_{Bool}$
$Falso_A : \rightarrow A_{Bool}$
$not_A : A_{Bool} \rightarrow A_{Bool}$
$(and_A), (or_A) : (A_{Bool}, A_{Bool}) \rightarrow A_{Bool}$

$Cierto_A =_{def} c$
$Falso_A =_{def} f$

v	not_A v
c	f
f	c
⊥	⊥

v1 v2	and_A v
c c	c
c f	f
c ⊥	⊥
f c	f
f f	f
f ⊥	f
⊥ c	⊥
⊥ f	f
⊥ ⊥	⊥

v1 v2	or_A v
c c	c
c f	c
c ⊥	c
f c	c
f f	f
f ⊥	⊥
⊥ c	c
⊥ f	⊥
⊥ ⊥	⊥

not Cierto = Falso por la primera fila de not_A
not Falso = Cierto por la segunda fila de not_A
Cierto and x = x por las tres primeras filas de and_A

```
Falso and x = Falso   por las filas 4 a 6 de and_A
Cierto or x = Cierto    por las tres primeras filas de or_A
Falso or x = x        por las filas 4 a 6 de or_A
```

Nótese que en la última fila de not_A, y las tres últimas de and_A y not_A podríamos haber elegido cualquier otra combinación de valores pues las ecuaciones no imponen ninguna condición sobre la aplicación de las operaciones a esos valores.

Nótese asimismo que en las ecuaciones donde aparecen generadoras es necesario realizar dos sustituciones:

```
    not Cierto = Falso

se interpreta como

    not_A Cierto_A = Falso_A  ⇔  not_A c = f  ⇔  f = f
```

3.2.9 Modelo de términos

En cualquier especificación algebraica es posible construir un modelo abstracto a partir del conjunto de términos generados.

Dado un TAD T, su modelo de términos viene dado por:

— Para cada tipo τ, el dominio abstracto de este tipo es

$$A_\tau =_{def} TG_\tau$$

— La T-equivalencia establece las ecuaciones válidas entre valores abstractos

$$t =_T t' \Leftrightarrow t = t' \text{ se deduce de la especificación para } t, t' \in A_S$$

— Las operaciones abstractas definidas como sigue

$$\text{Si } f : \tau_1, \ldots, \tau_n \to \tau \quad t_i \in A_{\tau i} \ (1 \le i \le n)$$

definimos

$$f_A(t_1, \ldots, t_n) =_{def} t \in A_\tau \text{ t.q. } f(t_1, \ldots, t_n) =_T t$$

Es decir consideramos como valores los términos generados, y como resultados de las operaciones los términos generados a los que son equivalentes.

Por ejemplo, en NAT

$Suc_A\ (Suc^2(Cero)) =_T Suc^3(Cero)$
$Suc^2(Cero) +_A Suc^3(Cero) =_T Suc^5(Cero)$
$Suc^3(Cero) *_A Suc^2(Cero) =_T Suc^6(Cero)$

Este modelo da la *semántica inicial* a la especificación algebraica. Se llama inicial pues se puede establecer un isomorfismo entre ese y cualquier otro modelo de la especificación. Es el modelo con el número mínimo de ecuaciones, o dicho de otra forma, cualquier modelo de la especificación algebraica debe cumplir todas las ecuaciones que cumpla el modelo de términos.

La especificación de un TAD debe ser diseñada de manera que su modelo abstracto describa el tipo de datos que el diseñador "tiene en mente". Como veremos más adelante, las implementaciones de un TAD están obligadas a realizar una representación del modelo abstracto.

3.2.10 Protección de un TAD usado por otro

Cuando un TAD utiliza un tipo *s* definido en otro TAD, las ecuaciones observadoras se deben escribir de tal forma que no introduzcan nuevos valores de tipo *s* –basura– ni tampoco nuevas equivalencias entre valores antiguos de tipo *s* –confusión–. Si se cumplen estas dos condiciones, decimos que el TAD usado está protegido.

Sea T un TAD con tipo principal τ cuya especificación algebraica usa otro TAD S con tipo principal *s*. Se dice que T *protege* a S si se cumplen las dos condiciones siguientes:

(i) T *no introduce basura* en S, es decir: para todo término cerrado $t : s$ que se pueda construir en T, existe un término generado $t' : s$ que ya se podía construir en S y tal que $t =_T t'$.

(ii) T *no introduce confusión* en S, es decir: Si $t, t' : s$ son términos cerrados generados que ya se podían construir en S, y se tiene $t =_T t'$, entonces ya se tenía $t =_S t'$.

(a) Demuestra que el siguiente TAD que usa a BOOL no protege a BOOL, porque in<troduce basura y confusión en BOOL:

```
tad PERSONA
   usa
      BOOL
   tipo
      Persona
   operaciones
      Carmen, Roberto, Lucía, Pablo: → Persona                    /*
gen */
      feliz: Persona → Bool                                       /*
obs */
   ecuaciones
      feliz(Carmen)     = feliz(Roberto)
      feliz(Pedro)    = feliz(Lucía)
      feliz(Pedro)    = not feliz(Pablo)
      feliz(Lucía)    = Cierto
```

```
feliz(Pablo)    = Cierto
```

Introduce basura porque hay términos de tipo *Bool* para los que no se puede deducir la equivalencia con ninguno de los términos generados de BOOL –es decir, que no se sabe si son ciertos o falsos–:

```
feliz(Carmen)    feliz(Roberto)
```

E introduce confusión porque es posible obtener nuevas equivalencias entre términos:

```
                          Cierto
feliz.4   =PERSONA        feliz(Lucía)
feliz.2   =PERSONA        feliz(Pedro)
feliz.3   =PERSONA        not feliz(Pablo)
feliz.5   =PERSONA        not Cierto
  not.1   =PERSONA        Falso
```

NAT usa a BOOL dejándolo protegido:

— No se introduce basura, porque en NAT cualquier término cerrado t : Bool cumple $t =_{NAT}$ Cierto o $t =_{NAT}$ Falso. Como se puede demostrar por inducción sobre la estructura sintáctica de t.

— No se introduce confusión, porque en NAT no es posible demostrar la ecuación Cierto = Falso.

3.2.11 Generadoras no libres

Sea T un TAD con tipo principal τ. Se dice que las operaciones generadoras de T son *no libres* si existen términos generados no idénticos t y t' de tipo τ, que sean T-equivalentes. Por el contrario, se dice que las operaciones generadoras de T son *libres* si no es posible encontrar términos generados no idénticos t, t' de tipo τ que sean T-equivalentes.

Decimos que dos términos son *idénticos* cuando son iguales sintácticamente.

Los TAD que venimos estudiando hasta ahora, BOOL y NAT, presentan generadoras libres. Veamos una especificación para los enteros donde aparecen generadoras no libres:

```
tad ENT
    usa
      BOOL, NAT
    tipo
      Ent
    operaciones
      Cero : → Ent        /* gen */
      Suc, Pred : Ent → Ent  /* gen */
      (- ) : Ent → Ent       /* mod */
      ( + ), ( - ), ( * ) : (Ent, Ent) → Ent   /* mod */
      ( ^ ) : (Ent, Nat) → Ent /* mod */
    ecuaciones
      ∀ x, y : Ent : ∀ n : Nat
```

```
        Suc(Pred(x)) = x
        Pred(Suc(x)) = x
        x + Cero     = x
        x + Suc(y)   = Suc(x+y)
        x + Pred(y)    = Pred(x+y)
        x – Cero       = x
        x – Suc(y)   = Pred(x–y)
        x – Pred(y)    = Suc(x–y)
        –x           = Cero – x
        x * Cero       = Cero
        x * Suc(y)   = (x * y) + x
        x * Pred(y)    = (x * y) – x
        x ^ Cero       = Suc(Cero)
        x ^ Suc(n)   = (x ^ n) * x
ftad
```

Podemos ver que las generadoras del TAD ENT no son libres pues existen términos generados que son ENT-equivalentes entre sí:

$$\begin{aligned} &\texttt{Suc(Pred(Suc(Cero)))} \\ \texttt{Suc} \quad &=_{\texttt{ENT}} \texttt{Suc(Cero)} \end{aligned}$$

Por el contrario las generadoras de BOOL y TAD sí son libres pues la especificación no incluye ninguna ecuación que sólo involucre a generadoras, por lo tanto no es posible aplicar ninguna regla del cálculo ecuacional que no sea la reflexiva. (¿es correcto este razonamiento?).

Esto nos da una idea a utilizar en el diseño de especificaciones: si las generadoras son libres entonces no escribimos ninguna ecuación que las relacione; por el contrario si las generadoras no son libres es necesario escribir ecuaciones que permitan determinar las equivalencias que nos interesen entre términos generados. En este segundo caso puede ser útil pensar en una elección de términos canónicos.

3.2.12 Representantes canónicos de los términos: $TC_\tau \subseteq TG_\tau$

Sea T un TAD con tipo principal τ. Se llama *sistema de representantes canónicos* a cualquier subconjunto TC_τ del conjunto TG_τ de todos los términos generados y cerrados de tipo τ, tal que para cada $t \in TG_\tau$ exista un único $t' \in TC_\tau$ tal que $t =_T t'$. Es decir, si todos los elementos de TC son algebraicamente diferentes, y para cada elemento de TG se puede encontrar uno T-equivalente en TC; lo que hacemos es elegir un representante para cada una de las clases de equivalencia que es posible definir entre los elementos de TG.

Si un TAD tiene generadoras libres entonces el conjunto de representantes canónicos coincide con el de términos generados, como ocurre en BOOL y NAT. No es así cuando tratamos con TAD con generadoras no libres. De hecho el concepto de términos canónicos se introduce para conseguir un conjunto mínimo de valores que representen a todos los valores posibles, cuando tenemos TAD con generadoras no libres.

Por ejemplo, en ENT podemos elegir como términos canónicos:

$$\texttt{TC}_{\texttt{Ent}} =_{\texttt{def}} \{\texttt{Cero}\} \cup \{\texttt{Suc}^n\texttt{(Cero)} \mid n > 0\} \cup \{\texttt{Pred}^n\texttt{(Cero)} \mid n > 0\}$$

la anterior es la elección más natural, pero no la única, por ejemplo

$$TC_{Ent} =_{def} \{Suc^n(Cero) \mid n > 0 \} \cup \{Pred^n(Suc(Cero)) \mid n > 0 \}$$

Insistir en la idea expuesta anteriormente de que cuando tenemos generadoras no libres debemos escribir ecuaciones sobre las generadoras de forma que sea posible traducir cualquier término generado a un término canónico. El conjunto de representantes canónicos en el que se estaba pensando cuando se escribió la especificación de ENT era el primero que hemos escrito más arriba. Podemos ver intuitivamente que efectivamente con las dos ecuaciones dadas es posible traducir cualquier término generado a uno de la forma Cero $\vee$ Suc^n(Cero) $\vee$ $Pred^n$(Cero).

Resumiendo lo dicho hasta ahora sobre las ecuaciones, debemos escribirlas teniendo en mente las siguientes condiciones:

— Las ecuaciones deben permitir las igualdades entre términos que son posibles entre los valores del modelo "en el que estamos pensando".

— Sólo escribiremos ecuaciones entre generadoras cuando estas no sean libres.

— Las ecuaciones de las modificadoras deben permitir convertir cualquier término en uno generado –acercarnos al caso base–.

— Las ecuaciones sobre las observadoras deben proteger al tipo usado –sin introducir basura ni confusión–.

3.2.13 Operaciones privadas y ecuaciones condicionales

Vamos a introducir dos nuevos elementos que es posible utilizar en las especificaciones algebraicas:

— Las operaciones privadas. Son operaciones que no exporta el TAD a sus clientes pero que ayudan a especificar el comportamiento de las operaciones exportadas, y, en algunos casos, también a su implementación.

— Ecuaciones condicionales. Son ecuaciones de la forma

```
ecuación    si   ecuación
ecuación    si   t  donde  t ∈ T_Bool
```

Veamos un ejemplo con la especificación de los enteros con orden. Aparece aquí otro concepto nuevo: enriquecimiento de un TAD. Un enriquecimiento es un TAD que usa a otro al que extiende añadiéndole operaciones nuevas, pero sin definir ningún tipo adicional.

La idea para escribir las ecuaciones de comparación es darse cuenta de que basta con dar el modo de calcular $\leq$, y el resto de las operaciones se pueden expresar en términos de ésta –se podría hacer una elección diferente–. Y la idea para indicar el modo de calcular $\leq$ es hacerlo en términos de la resta, viendo qué signo tiene ésta. Para ello introducimos la operación auxiliar noNeg. Y para escribir las ecuaciones de noNeg tenemos que utilizar ecuaciones condicionales.

```
tad ENT-ORD
    usa
       ENT
```

```
    operaciones
      ( == ), ( /= ) : (Ent, Ent) → Bool /* obs */
      ( ≤ ), ( ≥ ), ( < ), ( > ) : (Ent, Ent) → Bool  /* obs */
    operaciones privadas
      noNeg : Ent → Bool    /* obs */

    ecuaciones
      ∀ x, y : Ent
      noNeg(Cero)          = Cierto
      noNeg(Suc(x))        = Cierto si noNeg(x) = Cierto
      noNeg(Pred(Cero))    = Falso
      noNeg(Pred(x))       = Falso si noNeg(x) = Falso
      x ≤ y              = noNeg(y – x)
      x ≥ y              = y ≤ x
      x < y              = (x ≤ y) and (x /= y)
      x > y              = y < x
      x == y             = x ≤ y and y ≤ x
      x /= y             = not(x == y)
ftad
```

La inclusión de ecuaciones condicionales modifica una de las reglas del cálculo ecuacional

— Axiomas ecuacionales

$$\frac{C[x_1/t_1, \ldots, x_n/t_n]}{t\,[x_1/t_1, \ldots, x_n/t_n] = t'\,[x_1/t_1, \ldots, x_n/t_n]}$$

si `t = t' si C` es una ecuación de la especificación

Como ejercicio se propone la especificación del TAD POLINOMIO, nada trivial, que necesita de las operaciones privadas `multMono` para expresar la multiplicación de polinomios, y `quitaExp` para expresar la operación `esNulo`.

3.2.14 Clases de tipos

Introducimos un nuevo elemento en las especificaciones: las clases de tipos. Estas especificaciones permiten especificar restricciones sobre TADs. Son útiles como luego veremos para escribir TADs parametrizados, TADs que dependen de otros, pues permiten expresar las condiciones que se les exigen a los parámetros de los tipos parametrizados.

Veamos un primer ejemplo:

```
clase ANY
    tipo
      Elem
fclase
```

La clase de tipos ANY representa a cualquier tipo. Hay una *sutilidad* y es que no todos los TAD definen como tipo principal Elem –de hecho ninguno–. Entendemos que *Elem* es sinónimo del tipo principal que se define en un TAD. Para hacerlo más formal podríamos cambiar la definición del TAD y cambiar el nombre del tipo principal; o en el TAD paramétrico que depende de ANY, podríamos hacer un renombramiento del correspondiente tipo principal a *Elem*, pero como escribimos el tipo paramétrico sin saber sobre qué tipos se instanciará, no tenemos esa información.

En las clases de tipos se pueden poner *axiomas*, es decir condiciones que no son ecuaciones. La razón de que en las especificaciones de TAD sólo se pongan ecuaciones es que así está definida de forma unívoca un álgebra con un conjunto mínimo de suposiciones: el modelo inicial. Si aceptásemos axiomas con conectivas lógicas entonces podríamos encontrar más de un álgebra "con el mismo coste":

```
a = b or a = c
```

hace posible tanto un álgebra donde *a* sea igual a *b* o igual a *c* –ambos con el mismo coste– o igual a ambas.

En las clases de tipos sí podemos poner axiomas porque no tenemos la necesidad de construir una implementación particular de la clase de tipos, sino identificar una familia de álgebras posibles.

Un par de ejemplos más: los tipos con igualdad. aparece aquí un nuevo elemento que es la herencia entre clases de tipos, indicando que la clase EQ contiene todas las operaciones y tipos especificados en la clase ANY.

```
clase EQ
    hereda
       ANY
    usa
       BOOL
    operaciones
       ( == ), ( /= ) : (Elem, Elem) → Bool
    axiomas
       % ( == ) es una operación de igualdad.

    ecuaciones
       ∀ x, y : Elem :
       x /= y   =  not (x == y)
fclase
```

Y la clase de los tipos con orden

```
clase ORD
    hereda
       EQ
    operaciones
       ( ≤ ), ( ≥ ), ( < ), ( > ) : (Elem, Elem) → Bool
```

```
    axiomas
      % ( ≤ ) es una operación de orden
    ecuaciones
      ∀ x, y : Elem :
      x ≥ y    =   y ≤ x
      x < y    =   (x ≤ y) and (x /= y)
      x > y    =   y < x
```

Escribimos las condiciones sobre == y ≤ como axiomas porque no sabemos cuáles son las constructoras del tipo concreto y por lo tanto no podemos escribir las ecuaciones concretas.

Nótese que diferentes TADs pertenecientes a la misma clase de tipos pueden tener signaturas diferentes, ya que se exige que aparezcan una ciertas operaciones, pero no exclusivamente esas.

3.2.15 TADs genéricos

Los TADs genéricos son TADs que dependen de otros –uno o más– de forma que es posible construir distintos ejemplares del TAD genérico según el tipo de los parámetros.

Los TADs genéricos representan típicamente colecciones de elementos. Estos TADs tienen un cierto comportamiento independiente del tipo de los elementos, aunque también puede haber operaciones que dependan de los elementos. Las exigencias sobre los elementos se expresan indicando la clase de tipos a la que tienen que pertenecer los parámetros admisibles.

Como ejemplo vamos a escribir la especificación de los conjuntos CJTO[E :: EQ]

```
tad CJTO[E :: EQ]
    usa
      BOOL
    tipo
      Cjto[Elem]
    operaciones
      ∅ : → Cjto[Elem]                                          /* gen */
      Pon: (Elem, Cjto[Elem]) → Cjto[Elem]                       /* gen */
      quita: (Elem, Cjto[Elem]) → Cjto[Elem]                          /*
mod */
      esVacío: Cjto[Elem] → Bool                                 /* obs */
      ( ∈ ): (Elem, Cjto[Elem]) → Bool                           /* obs */
    ecuaciones
      ∀ x, y : Elem : ∀ xs : Cjto[Elem] :
      Pon(y, Pon(x, xs))   = Pon(x, xs)            si x == y
      Pon(y, Pon(x, xs))   = Pon(x, Pon(y, xs))
      quita(x, ∅)          = ∅
**    quita(x, Pon(y, xs))   = quita(x, xs)          si x == y
      quita(x, Pon(y, xs))   = Pon(y, quita(x, xs))  si x /= y
      esVacío(∅)             = Cierto
      esVacío(Pon(x, xs))    = Falso
```

```
        x ∈ ∅                = Falso
        x ∈ Pon(y, xs)       = x == y or x ∈ xs
    ftad
```

Uno podría sentirse tentado de escribir la ecuación marcada con ** como:

```
quita(x, Pon(y, xs))   = xs          si x == y
```

el problema es que aquí no estamos teniendo en cuenta que *x* ya podía estar en el conjunto *xs*, en cuyo caso la ecuación no sería válida. Si admitiésemos la ecuación en esta segunda forma, sería posible obtener equivalencias como esta:

```
                    Vacío
quita.2   =   quita(0, Pon(0, Vacío))
  Pon.1   =   quita(0, Pon(0, Pon(0, Vacío)))
quita.2   =   Pon(0, Vacío)
```

En general podríamos obtener Vacío = t para cualquier *t* de tipo Cjto.

Ejemplares de un TAD genérico

Una vez especificado un TAD genérico, es posible declarar *ejemplares* suyos. Cada ejemplar corresponde a un cierto reemplazamiento del parámetro formal del TAD genérico por un parámetro actual.

Por ejemplo, podemos construir un ejemplar del TAD genérico para los naturales, con lo que el nuevo TAD sería

```
CJTO[E/NAT con E.Elem/NAT.Nat] abreviado como CJTO[NAT]
```

Aparece aquí un nuevo elemento sintáctico: la cualificación de los elementos de un TAD con el nombre del TAD, como en E.Elem y NAT.Nat

Podemos hacer esta instanciación porque NAT pertenece a la clase EQ puesto que contiene las operaciones == y /=, siendo == una operación de igualdad.

La especificación de este nuevo TAD se obtiene a partir de la especificación del TAD genérico:

— sustituyendo *Elem* por *Nat*, con lo que el tipo principal es Cjto[Nat]

— considerando que las ecuaciones para la operación == –la única que no se especifica en la correspondiente clase de tipos– son las ecuaciones que aparecen en la especificación de NAT.

TADs genéricos con varios parámetros

También es posible parametrizar un TAD con más de un parámetro.

Como ejemplo podemos especificar el TAD de las parejas de valores cualesquiera PAREJA[A, B :: ANY]

```
tad PAREJA[A, B :: ANY]
```

```
    tipo
      Pareja[A.Elem, B.Elem]
    operaciones
      Par : (A.Elem, B.Elem) → Pareja[A.Elem, B.Elem] /* gen */
      pr: Pareja[A.Elem, B.Elem] → A.Elem/* obs */
      sg : Pareja[A.Elem, B.Elem] → B.Elem  /* obs */
    ecuaciones
      ∀ x : A.Elem: ∀ y : B.Elem :
      pr(Par(x, y))  =  x
      sg(Par(x, y))  =  y
ftad
```

Posibles instanciaciones de esta TAD genérico:

```
PAREJA[A/NAT con A.Elem/NAT.Nat, B/BOOL con B.elem/BOOL.Bool]

o abreviadamente

PAREJA[NAT, BOOL]
El tipo principal de este ejemplar es Pareja[Nat, Bool]
```

O parejas de conjuntos

```
PAREJA[CJTO[NAT], CJTO[BOOL]]

con tipo principal Pareja[Cjto[Nat], Cjto[Bool]]
```

3.2.16 Términos definidos e indefinidos

En muchas ocasiones nos tendremos que especificar TADs que incluyan operaciones parciales. En ese caso existirán términos que no están definidos y que deben recibir un tratamiento *cuidadoso*.

Como ejemplo de TAD con operaciones parciales vamos a utilizar el TAD PILA: un apila representa una colección de valores donde es posible acceder al último elemento añadido, implementa la idea intuitiva de *pila* de objetos.

```
tad PILA[E :: ANY]
    usa
      BOOL
    tipo
      Pila[Elem]
    operaciones
      PilaVacía: → Pila[Elem]  /* gen */
      Apilar: (Elem, Pila[Elem]) → Pila[Elem]  /* gen */
      desapilar: Pila[Elem] - → Pila[Elem]  /* mod */
      cima: Pila[Elem] - → Elem   /* obs */
      esVacía: Pila[Elem] → Bool  /* obs */
```

```
    ecuaciones
      ∀ x : Elem : ∀ xs : Pila[Elem] :
      def desapilar(Apilar(x, xs))
      desapilar(Apilar(x, xs))      = xs
      def cima(Apilar(x, xs))
      cima(Apilar(x, xs))            = x
      esVacía(PilaVacía)            = Cierto
      esVacía(Apilar(x, xs))        = Falso
    errores
      desapilar(Pilavacía)
      cima(PilaVacía)
ftad
```

Dos de las operaciones se han especificado como *parciales*, según indica el uso de – → en su perfil. Cuando una operación se especifica como parcial, su valor queda indefinido en ciertos casos. En este ejemplo, *despilar* y *cima* quedan indefinidas en los casos indicados en la sección **errores** de la especificación. La implementación deberá responder adecuadamente –hay distintas posibilidades– ante la aparición de estos errores.

Axiomas de definición

Aparece un nuevo tipo de axiomas en las especificaciones: los axiomas de definición. En este ejemplo:

```
def desapilar(Apilar(x, xs))
def cima(Apilar(x, xs))
```

Nótese que sólo se han escrito axiomas de definición para las operaciones parciales. Por convenio, para cada operación declarada como total en la signatura, con un perfil de la forma:

$$f: \tau_1, \ldots, \tau_n \rightarrow \tau$$

suponemos añadido a la especificación el axioma:

$$\forall\ x_1 : \tau_1 \ldots x_n : \tau_n : \textbf{def}\ f(x_1, \ldots, x_n)$$

Por ejemplo, en las pilas se suponen los axiomas de definición:

```
∀ x : Elem : ∀ xs : Pila[Elem] :
    def PilaVacía
    def Apilar( x, xs )
    def esVacía( xs )
```

Decimos entonces que un término *t* se considera *definido* siempre que **def** t sea deducible a partir de la especificación del TAD, e *indefinido* en caso contrario.

En los axiomas de definición cabe preguntarse si no es necesario exigir que las variables estén también definidas. Sin embargo esto no es necesario; lo que se hace es modificar la regla del

cálculo referida a los axiomas ecuacionales, imponiendo que sólo se pueden hacer sustituciones de variables por términos definidos.

Modificaciones a las reglas del cálculo ecuacional

La existencia de axiomas de definición da lugar a la modificación de algunas reglas del cálculo, y a la inclusión de reglas nuevas relacionadas con la definición de los términos.

— Reflexividad

$$\frac{\textbf{def } t}{t = t}$$

— Congruencia

$$\frac{t_1 = t_1' \quad \ldots \quad t_n = t_n' \quad \textbf{def } f(t_1, \ldots, t_n)}{f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}$$

— Axiomas ecuacionales

$$\frac{C[x_1/t_1, \ldots, x_n/t_n] \quad \textbf{def } t_1 \ldots \textbf{def } t_n}{t\,[x_1/t_1, \ldots, x_n/t_n] = t'\,[x_1/t_1, \ldots, x_n/t_n]}$$

si `t = t' si C` es una ecuación de la especificación

— Igualdad existencial

$$\frac{t = t'}{\textbf{def } t \quad \textbf{def } t'}$$

por la forma como hemos modificado las reglas del cálculo, sólo vamos a poder establecer igualdades entre términos definidos, por lo tanto una igualdad entre términos implica la definición de dichos términos.

Esto implica que cuando escribamos las ecuaciones de la especificación sólo podemos utilizar términos definidos.

— Operaciones estrictas.

Convenimos en que todas las operaciones de nuestras especificaciones son estrictas, por lo que

$$\frac{\textbf{def } f(t_1, \ldots, t_n)}{\textbf{def } t_1 \ldots \textbf{def } t_n}$$

Esta regla, como las anteriores, también se puede aplicar hacia atrás de forma que si no está definido algún t_i entonces no está definida $f(t_1, \ldots, t_n)$.

— Variables definidas

def x

Las variables están definidas por hipótesis; y sólo hay que tener cuidado de que se sustituyen por términos definidos.

— Axiomas de definición

$$\frac{C[x_1/t_1, \ldots, x_n/t_n] \quad \textbf{def}\ t_1 \ldots \textbf{def}\ t_n}{\textbf{def}\ t[x_1/t_1, \ldots, x_n/t_n]}$$

si **def** t **si** C es un axioma de definición de la especificación

En las condiciones también se utiliza la igualdad existencial

Sección errores

El contenido de la sección errores se define por eliminación: si **def** t, entonces t no puede incluirse en la sección errores. Por lo tanto, en la sección errores se deben incluir aquellos términos para los que no se puede demostrar **def** t

Revisión de conceptos debido a las funciones parciales

La inclusión de funciones parciales hace que debamos revisar algunos de los conceptos que hemos presentado hasta ahora, básicamente para indicar que sólo consideramos términos definidos.

— La T-equivalencia sólo se considera entre términos definidos.

Por la forma cómo hemos escrito las reglas del cálculo sólo es posible deducir nuevas equivalencias entre términos definidos, por lo que

$$t =_T t' \Rightarrow \textbf{def}\ t \wedge \textbf{def}\ t'$$

$$TD_\tau\ (X) =_{def} \{\ t \in T_\tau(X) \mid \textbf{def}\ t\ \}$$

— De entre los términos generados interesan los definidos

$$TGD_\tau(X) =_{def} \{\ t \in TG_\tau(X) \mid \textbf{def}\ t\ \}$$

— Para que el conjunto de generadoras sea suficientemente completo, debe ocurrir que cada término cerrado y definido *t* sea T-equivalente a algún término cerrado generado y definido *t'*

$$\forall\ t : t \in TD_\tau : \exists\ t' : t' \in TGD_\tau : t =_T t'$$

— Un sistema de representantes canónicos debe elegirse como un subconjunto del conjunto de todos los términos generados cerrados y definidos, de manera que cada término cerrado y definido sea T-equivalente a un único representante canónico

$TCD_\tau(X) =_{def} \{ t \in TC_\tau(X) \mid \mathbf{def}\ t \}$

Los dos siguientes puntos se pueden obviar en la explicación

— En cuanto a la protección de los TAD usados, las condiciones se reescriben para hacer referencia sólo a términos definidos

(i) T *no introduce basura* en S, es decir: para todo término cerrado y definido `t : s` que se pueda construir en T, existe un término generado y definido `t' : s` que ya se podía construir en S y tal que $t =_T t'$.

(ii) T *no introduce confusión* en S, es decir: Si `t, t' : s` son términos cerrados generados y definidos que ya se podían construir en S, y se tiene $t =_T t'$, entonces ya se tenía $t =_S t'$.

— Por último, se modifica la definición del modelo de términos para considerar sólo los términos definidos

— Para cada tipo τ, el dominio abstracto de este tipo es

$A_\tau =_{def} TGD_\tau$

— La T-equivalencia establece las ecuaciones válidas entre valores abstractos

$t =_T t' \Leftrightarrow t = t'$ se deduce de la especificación
para $t, t' \in A_\tau$

(esta condición sólo se modifica implícitamente, al cambiar la definiciónde A_τ)

— Las operaciones abstractas definidas como sigue

Si $f : \tau_1, \ldots, \tau_n \to \tau$
$t_i \in A_{\tau i}$ $(1 \le i \le n)$

definimos

$$f_A(t_1, \ldots, t_n) =_{def} \begin{cases} t \in A_\tau \text{ t.q. } f(t_1, \ldots, t_n) =_T t & \text{si } \mathbf{def}\ f(t_1, \ldots, t_n) \\ \text{indefinido} & \text{en otro caso} \end{cases}$$

por ejemplo:

$cima_A$ (PilaVacía) está indefinido
$cima_A$ ($desapilar_A$ (Apilar(t2, Apilar(t1, PilaVacía)))) = t1

3.2.17 Igualdad existencial, débil y fuerte

En presencia de operaciones parciales se pueden definir distintos conceptos de igualdad:

- *Igualdad existencial* $=^e$
  $t =^e t' \Leftrightarrow_{def}$ t y t' están ambos definidos y valen lo mismo

  nosotros escribiremos simplemente = para la igualdad existencial. Este es el concepto de igualdad que hemos utilizado al extender las reglas del cálculo

- *Igualdad débil* $=^d$
  $t =^d t' \Leftrightarrow_{def}$ t y t' valen lo mismo si están los dos definidos

  (la igualdad débil siempre se cumple si t, t' o ambos están indefinidos)

- *Igualdad fuerte* $=^f$
  $t =^f t' \Leftrightarrow_{def}$ o bien t y t' están ambos definidos y valen lo mismo
  o bien t y t' están ambos indefinidos

  es decir, no son iguales cuando sólo uno de los dos está indefinido

Con la igualdad existencial se pueden expresar los otros dos tipos de igualdades, aprovechándonos de que

$t =^e t \rightarrow$ **def** t

$t =^d t'$ equivale a

$t =^e t' \leftarrow t =^e t \wedge t' =^e t'$

es decir, si ambos están definidos y son iguales ($=^e$) o si alguno no está definido, con lo que el antecedente de la implicación es falso, y la implicación cierta

$t =^f t'$ equivale a

$(t' =^e t' \leftarrow t =^e t) \wedge$
$(t =^e t \leftarrow t' =^e t') \wedge$
$(t =^e t' \leftarrow t =^e t \wedge t' =^e t')$

3.3 Implementación de TADs

Supongamos dada una especificación algebraica de un TAD T en el que aparecen diferentes tipos. Una *implementación* de T consiste en:

— Un *dominio concreto* D_τ para cada tipo τ incluido en T

Los dominios concretos se implementan mediante declaraciones de tipos, usando otros tipos ya implementados –por nosotros o incluidos en el lenguaje– que definen el *tipo representante* –en realidad el D_τ es el conjunto de representantes válidos del tipo τ–.

— Una *operación concreta*

$$f_C : D_{\tau 1}, \ldots, D_{\tau n} \longrightarrow D_\tau$$

para cada operación

$$f : \tau_1, \ldots, \tau_n \rightarrow \tau$$

Las operaciones concretas pueden implementarse como procedimientos aunque en la especificación sólo se admitan funciones.

De modo que satisfagan dos requisitos:

— Corrección: la implementación debe satisfacer los axiomas de la especificación

— Privacidad y protección: la estructura interna de los datos debe estar oculta; el único acceso posible al tipo debe ser a través de las operaciones públicas de éste.

En el siguiente tema veremos cómo garantizar la privacidad y protección; y más adelante en este mismo tema estudiaremos un método para probar la corrección.

Por ahora vamos a empezar con un ejemplo donde motivaremos los conceptos básicos relativos a la construcción de implementaciones correctas.

3.3.1 Implementación correcta de un TAD

Vamos a presentar las condiciones que debemos exigir a una implementación para que sea correcta con respecto a una especificación dada. Introduciremos las ideas mediante un ejemplo: la implementación del TAD PILA[NAT]. Consideramos que el TAD NAT está implementado como uno de los tipos predefinidos del lenguaje algorítmico.

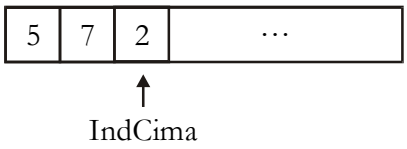
Implementación del tipo

En primer lugar, tenemos que decidir cómo implementamos el tipo *Pila[Nat]*, es decir, tenemos que elegir un *tipo representante*. En este caso elegimos representar las pilas como un registro con dos campos, uno que es un vector donde se almacenan los elementos y otro que es un índice que apunta a la cima de la pila dentro del vector

Por ejemplo la representación de

```
Apilar( 2, Apilar( 7, Apilar( 5, PilaVacía )))
```

vendría dada por:



Esta implementación tiene el inconveniente de que impone a priori un límite al tamaño máximo de la pila.

El tipo representante escogido es:

```
const
  limite = 100;
tipo
  Pila[Nat] = reg
    espacio : Vector [1..limite] de Nat;
    indCima : Nat
  freg
```

Para luego poder plantear una implementación correcta de las operaciones, conviene comenzar definiendo dos cosas:

— qué valores concretos queremos aceptar como *representantes válidos* de valores abstractos –la idea es que el tipo representante elegido puede tomar valores que no consideremos válidos–.

— cuál es el valor abstracto representado por cada valor concreto que sea un representante válido.

Empezamos formalizando qué condiciones les exigimos a los representantes válidos:

$$
\begin{aligned}
& R_{Pila[Nat]}\ (xs) \\
\Leftrightarrow_{def}\ & \\
& xs : Pila[Nat] \wedge \\
& 0 \leq xs.indCima \leq limite \wedge \\
& \forall\ i : 1 \leq i \leq xs.indCima : R_{Nat}(xs.espacio(i))
\end{aligned}
$$

La primera condición es una notación abreviada, en realidad lo que queremos decir es que *xs* es un valor del tipo que representa a *Pila[Nat]*, es decir, el registro.

Las condiciones que se les exigen a los representantes válidos se denominan *invariante de la representación*.

Notación:

TR_τ : tipo representante del tipo τ

RV_τ : conjunto de representantes válidos del tipo $\tau \equiv D_\tau$, el dominio asignado al tipo τ

R_τ : invariante de la representación del tipo τ

Los representantes válidos son aquellos valores del tipo representante que cumplen el invariante de la representación:

$RV_\tau = \{ x : TR_\tau \mid R_\tau(x) \}$

En segundo lugar indicamos una forma de obtener cuál es el valor abstracto representado por cada valor concreto que sea un representante válido. Para ello definimos de forma recursiva una *función de abstracción* de la siguiente forma:

para *xs* tal que $R_{Pila[Nat]}(xs)$

$A_{Pila[Nat]}$ (xs) $=_{def}$ hazPila(xs.espacio, xs.indCima)

hazPila(v, 0) $=_{def}$ PilaVacía

hazPila(v, n) $=_{def}$ Apilar(A_{Nat}(v(n)), hazPila(v, n-1)) **si** $1 \leq n \leq$ límite

nótese que la función de abstracción es una función que va de los representantes válidos en los términos generados definidos, es decir, el conjunto de valores para τ dado en el modelo de términos

$A_\tau : RV_\tau \rightarrow TGD_\tau$

Convenios:

— Para simplificar, escribiremos v(n) en lugar de $A_{Nat}(v(n))$

— En general, muchas veces simplificaremos omitiendo R_σ y A_σ cuando σ no sea el tipo principal que estemos considerando.

— Escribiremos R(d) y A(d) cuando se suponga conocido el tipo correspondiente.

Nótese que dos representantes válidos diferentes pueden tener asociado el mismo valor abstracto. Como se puede comprobar en el siguiente ejemplo:

xs:

indCima:	3					
espacio:	5	2	7	9	7	…

ys:

indCima:	3					
espacio:	5	2	7	0	6	…

donde se tiene que

$xs, ys \in RV_{Pila[Nat]}$; $xs \neq ys$

A(xs)

$=_{Pila[Nat]}$

A(ys)

$=_{Pila[Nat]}$

```
      Apilar( Suc^7(Cero), Apilar( Suc^2(Cero), Apilar( Suc^5(Cero),
PilaVacía)))
```

Normalmente abreviaremos la notación no utilizando los valores abstractos para los tipos primitivos, con lo que escribiríamos:

```
Apilar( 7, Apilar( 2, Apilar( 5, PilaVacía)))
```

El hecho de que dos representantes válidos diferentes puedan representar al mismo valor abstracto implica que, en general, la igualdad entre representantes no representa la igualdad entre valores abstractos. Es por ello, que, en general, no podemos utilizar la igualdad incorporada en los lenguajes sino que cada TAD en cuya especificación se incluya una operación de igualdad, deberá implementarla explícitamente.

Implementación de las operaciones

Lo que exigimos a las operaciones es que implementen el modelo de términos. Este hecho lo reflejaremos en la especificación pre/post de los subprogramas que implementan a las operaciones del TAD. Para cada operación supondremos como parte de su precondición que los argumentos son representantes válidos de valores abstractos. La postcondición, a su vez, deberá garantizar que el resultado sea un representante válido del resultado de la correspondiente operación abstracta aplicada sobre los valores abstractos que representan los argumentos. Formalmente,

la especificación pre/post para una función f_C que implementa una operación

$$f : (\tau_1, \ldots, \tau_n) - \rightarrow \tau$$

del TAD T

func f_C $(x_1 : TR_{\tau 1}, \ldots, x_n : TR_{\tau n})$ **dev** $y : TR_\tau$;

$\{ P_0 : R_{\tau 1}(x_1) \wedge \ldots \wedge R_{\tau n}(x_n) \wedge DOM_f(x_1, \ldots, x_n) \wedge LIM_f(x_1, \ldots, x_n) \}$

$\{ Q_0 : R_\tau(y) \wedge A_\tau(y) =_T f_A(A_{\tau 1}(x_1), \ldots, A_{\tau n}(x_n)) \}$

donde

TR_{τ_i}, TR_τ son los tipos representantes

R_{τ_i}, R_τ son los invariantes de la representación

A_{τ_i}, A_τ son las funciones de abstracción

DOM_f es la condición de dominio.

Para $x_i : TR_{\tau_i}$ tales que $R_{\tau_i}(x_i)$

$DOM_f(x_1, \ldots, x_n) \Leftrightarrow$ **def** $f(A_{\tau_1}(x_1), \ldots, A_{\tau_n}(x_n))$

las condiciones de DOM pueden de venir expresadas como operaciones sobre términos abstractos o como condiciones sobre el tipo representante

LIM_f expresa restricciones adicionales impuestas por la implementación

Este esquema explica la terminología *invariante de la representación*, pues es una condición que deben cumplir tanto los parámetros como los resultados de las operaciones.

Cuando LIM_f no es la condición trivial *cierto*, decimos que la implementación es *parcialmente correcta*.

Hay que tener cuidado con la aparición de f_A en la postcondición. Cuando el TAD T tiene un conjunto de generadoras libres, entonces no hay problema. Sin embargo, si el conjunto de generadoras no es libre, entonces en el modelo de términos no está determinado cuál de los términos generados pertenecientes a la misma clase de equivalencia es el resultado de la función abstracta. Podríamos interpretar que el resultado de f_A es "uno" de los términos generados que pertenecen a la misma clase de equivalencia. Otra posibilidad sería haber definido el modelo de términos utilizando el conjunto de términos canónicos en lugar del conjunto de términos generados.

Nótese que el esquema que hemos presentado generaliza a todas las posibles operaciones de un TAD, incluyendo generadoras, modificadoras y observadoras.

Veamos algunos ejemplos de cómo se implementarían las operaciones de las pilas de naturales, con la representación escogida para el tipo.

```
func PilaVacía dev xs : Pila[Nat];
{ P0 :      }
inicio
  xs.indCima := 0
{ Q0 : RPila[Nat](xs) ∧ APila[Nat](xs) =PILA[NAT] PilaVacía }
  dev xs
ffunc
```

Según el esquema teórico, en la postcondición debería aparecer $PilaVacía_A$, pero el valor de esta función es el término generado PilaVacía, que es el que hemos utilizado.

Vamos con la otra generadora:

```
func Apilar ( x : Nat; xs : Pila[Nat] ) dev ys : Pila[Nat];
{ P0 : RNat(x) ∧ RPila[Nat](xs) ∧ xs.indCima < límite }
var
  llena : Bool;

inicio
  ys := xs;
  llena := ys.indCima = límite;
  si llena
    entonces
      error("No se puede apilar porque la pila está llena")   % **
    sino
      ys.indCima := ys.indCima + 1;
      ys.espacio(ys.indCima) := x
  fsi
{ Q0 : RPila[Nat](ys) ∧ APila[Nat](ys) =PILA[NAT] Apilar(ANat(x), APila[Nat](xs)) }
  dev ys
```

```
ffunc
```

** no fijamos cuál es el tratamiento de errores. Hay muchas formas de implementarlo:

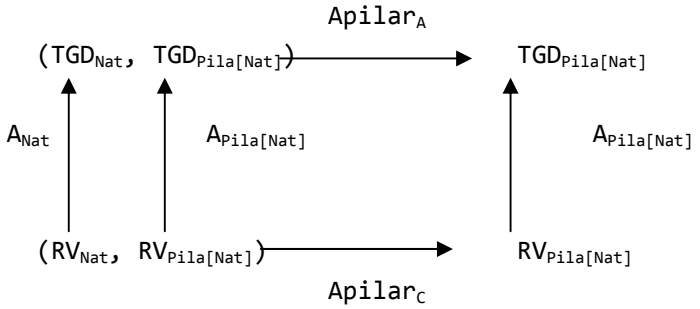
— mostrar un mensaje y detener la ejecución

— mostrar el mensaje y no detener la ejecución

— implementar un mecanismo que permite a los clientes del TAD saber si se ha producido un error y actuar en consecuencia

En la función *Apilar* aparece un ejemplo de condición en la precondición que viene impuesta por las limitaciones de la implementación: xs.indCima < límite. Tenemos por tanto que esta es una implementación parcialmente correcta. No obstante, la postcondición garantiza que *Apilar* realiza correctamente la operación abstracta en aquellos casos en que su ejecución termina normalmente.

Utilizando esta función como ejemplo, podemos representar gráficamente cómo se puede establecer una relación entre la función concreta y la abstracta a través de las funciones de abstracción:



La función *cima*

```
func cima ( xs : Pila[Nat] ) dev x : Nat;
{ P₀ : R_Pila[Nat](xs) ∧ not esVacía( A(xs) ) }
inicio
  si esVacía( xs )    % xs.indCima == 0
    entonces
      error("La pila vacía no tiene cima")  % *
    sino
      x := xs.espacio(xs.indCima)
  fsi
{ Q₀ : R_Nat(x) ∧ A_Nat(x) =_PILA[NAT] cima_A(A_Pila[Nat](xs)) } **
  dev x
ffunc
```

** podemos abreviar esta condición, no preocupándonos por los elementos, como:

```
x =_PILA[NAT] cima(A_Pila[Nat](xs))
```

En esta operación aparece una condición en la precondición de tipo DOM_f, es decir, una restricción impuesta por la especificación.

En cuanto a la operación *desapilar*:

```
func desapilar( xs : Pila[Nat] ) dev ys : Pila[Nat];
{ P0 : RPila[Nat](xs) ∧ xs.indCima > 0 }
inicio
  si xs.indCima == 0
    entonces
      error("no se puede desapilar de la pila vacía")
    sino
      ys.espacio := xs.espacio;
      ys.indCima := xs.indCima – 1
  fsi
{ Q0 : RPila[Nat](ys) ∧ APila[Nat](ys) =PILA[NAT] despilarA(APila[Nat](xs)) }
  dev ys
ffunc
```

Nótese que aquí hemos escrito la condición

```
not esVacía( A(xs) )  como xs.indCima > 0
```

nos permitimos esta libertad.

Nótese también que la operación de igualdad entre naturales la hemos escrito ==, en lugar de = como veníamos haciendo hasta ahora. La razón es que queremos distinguir la operación de igualdad de la igualdad entre términos en las especificaciones.

Nos queda por último la operación *esVacía*:

```
func esVacía ( xs : Pila[Nat] ) dev r : Bool;
{ P0 : RPila[Nat](xs) }
inicio
  r := xs.indCima == 0
{ Q0 : RBool(r) ∧ ABool(r) =PILA[NAT] esVacíaA( APila[Nat](xs) ) }
  dev r
ffunc
```

Implementación de las operaciones como funciones o como procedimientos

En los ejemplos anteriores hemos implementado todas las operaciones como funciones, según sugiere la especificación. Sin embargo, esta implementación tiene algunos inconvenientes:

— En muchos lenguajes de programación, las funciones no pueden devolver valores de tipos estructurados.

— La implementación como funciones supone realizar copias de los parámetros, lo cual consume espacio y tiempo.

— En muchos casos consideramos a los valores de un TAD como objetos mutables, de modo que el resultado de la modificadora se lo asignamos a la misma variable que contenía el valor modificado.

La solución radica en implementar las operaciones como procedimientos, que en las operaciones sobre las pilas darían lugar a las siguientes especificaciones:

```
proc PilaVacía ( s xs : Pila[Nat] );
{ P0 :      }
{ Q0 : RPila[Nat](xs) ∧ APila[Nat](xs) =PILA[NAT] PilaVacía }
fproc

proc Apilar ( e x : Nat; es xs : Pila[Nat] );
{ P0 : xs = XS ∧ RNat(x) ∧ RPila[Nat](xs) ∧ xs.indCima < límite }
{ Q0 : RPila[Nat](xs) ∧ APila[Nat](xs) =PILA[NAT] Apilar(ANat(x), APila[Nat](XS)) }
fproc

proc desapilar( es xs : Pila[Nat] );
{ P0 : xs = XS ∧ RPila[Nat](xs) ∧ xs.indCima > 0 }
{ Q0 : RPila[Nat](xs) ∧ APila[Nat](xs) =PILA[NAT] despilarA(APila[Nat](XS)) }
fproc
```

La implementación de *cima* se puede mantener como una función, siempre que el tipo de los elementos lo permita. Y, de la misma forma, podemos mantener como función la operación *esVacía*.

Es trivial modificar las implementaciones como funciones para llegar a la implementación como procedimientos.

La implementación como modelo débil de la especificación (no explicar)

El interés de construir una implementación correcta con respecto a la especificación radica en que cualquier equivalencia que sea posible deducir de la especificación también será válida para la implementación, salvo indefiniciones impuestas por las limitaciones de la implementación –de ahí la *debilidad* del modelo–. Formalmente:

Sea T una especificación algebraica de una TAD. Supongamos que se cumple:

$$t =_T t'$$

siendo

$$var(t) \cup var(t') = \{ x_1 : \tau_1, \ldots, x_n : \tau_n \}$$
$$t, t' \in TD_\tau$$

Para cualquier implementación correcta de T puede asegurarse entonces que:

$$R_{\tau_1}(x_1) \wedge \ldots \wedge R_{\tau_n}(x_n) \wedge \mathbf{def}\ t_C \wedge \mathbf{def}\ t'_C \Rightarrow A_\tau(t_C) =_T A_\tau(t'_C)$$

o, escrito de otra forma:

$$\{\ R_{\tau 1}(x_1) \wedge \ldots \wedge R_{\tau n}(x_n)\ \}$$
$$r := t_c;$$
$$r' := t'_c;$$
$$\{\ A_\tau(r) =_T A_\tau(r')\ \}$$

si este cómputo acaba –puede no acabar por limitaciones de la implementación–, entonces se cumple la postcondición. Los términos concretos se obtiene sustituyendo en los términos abstractos las operaciones por operaciones concretas. El término resultante será *ejecutable*, siempre que las operaciones del TAD se implementen todas como funciones:

$$t \equiv f(x, g(y, e)) \qquad t_c \equiv f_C(x, g_C(y, e_C))$$

Esta idea habría que formalizarla con más cuidado pues no hemos definido cuál es el término concreto, t_C, asociado con un término cualquiera, t. Lo que se pretende expresar es que la *ejecución* de dos términos equivalentes da resultados equivalentes.

Verificación de la corrección de las operaciones

Como ha ocurrido en el ejemplo anterior, no vamos a hacer las verificaciones de que efectivamente las implementaciones de las operaciones son correctas con respecto a las especificaciones pre/post que hemos escrito.

La demostración de la corrección de una implementación pasaría por la verificación de todas y cada una de las implementaciones de las operaciones. el problema radica en que la forma de las funciones de abstracción suele ser compleja y, por lo tanto, las verificaciones donde estén inmersas estas funciones también lo serán.

3.3.2 Otro ejemplo: CJTO[NAT]

Vamos a plantear tres posibles representaciones para este tipos de datos:

— Vectores de naturales

— Vectores de naturales sin repetición

— Vectores de naturales ordenados sin repetición

En los tres casos elegimos el mismo tipo representante:

```
const
  limite = 100;
tipo
  Cjto[Nat] = reg
    espacio : Vector[1..limite] de Nat;
    tamaño : Nat
  freg;
```

Vectores de naturales

El invariante de la representación:

```
    R_Cjto[Nat] (xs)
⇔def
    xs : Cjto[Nat] ∧
    0 ≤ xs.tamaño ≤ limite ∧
    ∀ i : 1 ≤ i ≤ xs.tamaño : R_Nat(xs.espacio(i))
```

La función de abstracción:

```
para xs tal que R_Cjto[Nat](xs)

    A_Cjto[Nat] (xs) =def hazCjto(xs.espacio, xs.tamaño)

    hazCjto(v, 0) =def ∅
    hazCjto(v, n) =def Pon( A_Nat(v(n)), hazCjto(v, n-1))   si 1 ≤ n ≤ límite
```

Nótese que como las constructoras no son libres, la función de abstracción puede dar como resultado términos generados distintos pero equivalentes.

Vectores de naturales sin repetición

El invariante de la representación:

```
    R_Cjto[Nat] (xs)
⇔def
    xs : Cjto[Nat] ∧
    0 ≤ xs.tamaño ≤ limite ∧
    ∀ i : 1 ≤ i ≤ xs.tamaño : R_Nat(xs.espacio(i)) ∧
    ∀ i, j : 1 ≤ i < j ≤ xs.tamaño : xs.espacio(i) /= xs.espacio(j)
```

La función de abstracción es la misma de antes. También aquí puede ocurrir que dos conjuntos equivalentes estén representados por términos generados diferentes, aunque equivalentes.

Vectores de naturales sin repetición y ordenados

El invariante de la representación:

```
    R_Cjto[Nat] (xs)
⇔def
    xs : Cjto[Nat] ∧
    0 ≤ xs.tamaño ≤ limite ∧
    ∀ i : 1 ≤ i ≤ xs.tamaño : R_Nat(xs.espacio(i)) ∧
    ∀ i, j : 1 ≤ i < j ≤ xs.tamaño : xs.espacio(i) < xs.espacio(j)
```

La función de abstracción es la misma, pero ahora cada valor concreto vendrá representado por un término generado diferente. En este caso la demostración de la corrección de las operaciones será más sencilla pues no habrá que preocuparse por equivalencias entre los términos generados.

Implementación de la operación *Pon*

Vectores de naturales:

```
proc Pon( e x : Nat; es xs : Cjto[Nat] );
{P₀ : xs = XS ∧ R(xs) ∧ tamaño < limite }  % nos olvidamos de R(x) y de los
                                            subíndices
inicio
  si tamaño == limite
    entonces
      error("No se puede insertar el elemento")
    sino
      xs.tamaño := xs.tamaño + 1;
      xs.espacio(xs.tamaño) := x
  fsi
{ Q₀ : R(xs) ∧ A(xs) =PILA[NAT] Pon( x, A(XS)) }
fproc
```

En la postcondición también hemos simplificado la notación, escribiendo Pon en lugar de Pon_A y x en lugar de $A_{Nat}(x)$.

Vectores de naturales sin repetición. Vamos a suponer implementada una función de búsqueda con la siguiente especificación:

```
func busca ( x : Nat; v : Vector[1.. limite] de Nat; a, b : Nat) dev r :
Bool;
{ P₀ : 1 ≤ a ≤ b+1 ≤ N+1 }
{ Q₀ : r ↔ ∃ i : a ≤ i ≤ b : v(i) = x }
ffunc
```

La operación *Pon*

```
proc Pon( e x : Nat; es xs : Cjto[Nat] );
{P₀ : xs = XS ∧ R(xs) ∧ tamaño < limite }
inicio
  si tamaño == limite
    entonces
      error("No se puede insertar el elemento")
    sino
      si busca( x, xs.espacio, 1, xs.tamaño )
        entonces
```

```
          seguir
        sino
          xs.tamaño := xs.tamaño + 1;
          xs.espacio(xs.tamaño) := x
      fsi
  fsi
{ Q₀ : R(xs) ∧ A(xs) =PILA[NAT] Pon( x, A(XS)) }
fproc
```

Vectores de naturales sin repetición y ordenados. Vamos a suponer implementada una función de búsqueda binaria con la siguiente especificación:

```
func buscaBin ( x : Nat; v : Vector[1.. limite] de Nat; a, b : Nat) dev p :
                                                                    Nat;
                                                                    r : Bool;
{ P₀ : 1 ≤ a ≤ b+1 ≤ N+1 ∧ ord(v[a..b]) }
{ Q₀ : a ≤ p+1 ≤ b+1 ∧ v[a..p] ≤ x < v[p+1..b] ∧ r ↔ ∃ i : a ≤ i ≤ b : v(i)
= x}

proc Pon( e x : Nat; es xs : Cjto[Nat] );
{P₀ : xs = XS ∧ R(xs) ∧ tamaño < limite }
var
  p : Nat;
  encontrado : Bool;
inicio
  si tamaño == limite
    entonces
      error("No se puede insertar el elemento")
    sino
      <p, encontrado> := buscaBin( x, xs.espacio, 1, xs.tamaño );
      si encontrado
        entonces
          seguir
        sino
          desplazaDrch( v, p+1, xs.tamaño );
          xs.tamaño := xs.tamaño + 1;
          xs.espacio( p+1 ) := x
      fsi
  fsi
{ Q₀ : R(xs) ∧ A(xs) =PILA[NAT] Pon( x, A(XS)) }
fproc
```

Y la operación *desplazaDrch*:

```
proc desplazaDrch ( es v : Vector[1..limite] de Nat; e a, b : Nat );
{ P₀ : v = V ∧ 1 ≤ a ≤ b+1 < N+1 }
```

```
var
  k : Nat;
inicio
  para k desde b+1 bajando hasta a+1 hacer
    v(k) := v(k-1)
  fpara
{ P₀ : v[a+1..b+1] = V[a..b] ∧ v[1..a] = V[1..a] ∧ v[b+2..limite] =
V[b+2..limite]}
fproc
```

Se pueden analizar las diferencias entre estas tres representaciones en términos de las operaciones *Pon* y *quita*.

3.3.3 Verificación de programas que usan TADs

Al aplicar las reglas de verificación a programas que usen TADs implementados funcionalmente, las operaciones de estos se pueden tratar del mismo modo que si fuesen operaciones predefinidas.

Esto quiere decir que, en lugar de usar las reglas de verificación para llamadas a funciones, se puede usar la regla de la asignación. Además, se pueden utilizar todas aquellas propiedades de las operaciones del TAD en cuestión que se deduzcan de la especificación de éste, ya que se supone correcta la implementación disponible para el mismo.

Si la implementación del TAD es procedimental en lugar de funcional, se pueden aplicar las mismas técnicas reemplazando previamente (a efectos de la verificación) las llamadas a procedimientos por llamadas a funciones.

Por ejemplo, supuesta una implementación procedimental del TAD PILA[NAT] queremos verificar:

```
var
  x : Nat;
  xs, ys : Pila[Nat]
{ P : xs = Apilar(1, ys) }
  x := cima(xs);
  desapilar(xs);
  x := 2*x+1
{ Q : x = 3 ∧ xs = ys }
```

Como paso intermedio reescribiremos el programa de manera que todas las operaciones de las pilas se usen como funciones:

```
var
  x : Nat;
  xs, ys : Pila[Nat]
{ P : xs = Apilar(1, ys) }
  x := cima(xs);
  xs := desapilar(xs);
```

```
  x := 2*x+1
{ Q : x = 3 ∧ xs = ys }
```

Finalmente, hacemos la verificación con ayuda de asertos intermedios adecuados, usando las propiedades especificadas para el TAD PILA[NAT]:

```
var
  x : Nat;
  xs, ys : Pila[Nat]
{ P : xs = Apilar(1, ys) }
  x := cima(xs);
{ R : x = 1 ∧ xs = Apilar(1, ys) } /* 1 */
  xs := desapilar(xs);
{ S : x = 1 ∧ xs = ys } /* 2 */
  x := 2*x+1
{ Q : x = 3 ∧ xs = ys } /* 3 */
```

Los tres pasos se verifican usando la regla de la asignación:

Verificación de /* 1 */

```
                          pmd( x:= cima(xs), R)
                     ⇔    def(cima(xs)) ∧ R[x/cima(xs)]
                     ⇔    def(cima(xs)) ∧ cima(xs) = 1 ∧ xs = Apilar(1, ys)
              PILA   ⇐    P
```

Verificación de /* 2 */

```
                          pmd(xs := desapilar(xs), S)
                     ⇔    def(desapilar(xs)) ∧ S[xs/desapilar(xs)]
                     ⇔    def(desapilar(xs)) ∧ x = 1 ∧ desapilar(xs) = ys
              PILA   ⇐    R
```

Verificación de /* 3 */

```
                          pmd( x := 2*x +1, Q)
                     ⇔    def(2*x+1) ∧ Q[x/2*x+1]
                     ⇔    2*x+1 = 3 ∧ xs = ys
 aritmética (i.e., NAT)   ⇐    S
```

3.5 Estructuras de datos dinámicas

Recordemos que consideramos como *estructuras de datos* a los tipos concretos que utilizamos para realizar los tipos definidos en los tipos abstractos de datos.

3.5.1 Estructuras de datos estáticas y estructuras de datos dinámicas

Algunas implementaciones de TADs se basan en estructuras de datos estáticas.

Una estructura de datos se llama estática cuando el espacio que va a ocupar está determinado en tiempo de compilación.

Puede que ese espacio se ubique al principio de la ejecución –variables globales– o en la invocación a procedimientos o funciones –variables locales–.

La mayoría de los tipos predefinidos de los lenguajes de programación se realizan por medio de estructuras estáticas. Son ejemplos:

- los tipos numéricos
- los caracteres
- los booleanos
- los vectores
- los registros

Muchos TADs sirven para representar colecciones de datos. La estructura de datos estática apta para representar colecciones de datos son los vectores. Sin embargo, para muchos TADs especificados por el usuario, las implementaciones basadas en vectores tienen inconvenientes graves que ya hemos mencionado en el tema 3.3 en relación con las pilas y los conjuntos, y que se resumen así:

- despilfarro de recursos si el espacio máximo reservado resulta ser excesivo
- errores de desbordamiento en tiempo de ejecución si el espacio reservado resulta ser insuficiente

---

En suma, las estructuras estáticas –los vectores– resultan demasiado rígidas para la implementación de TADs cuyas operaciones sean capaces de generar valores cuya representación necesite una cantidad de espacio potencialmente ilimitado.

Estructuras de datos dinámicas

Se llama estructura de datos dinámica a una estructura que ocupa en memoria un espacio variable durante la ejecución del programa, y que no se determina en tiempo de compilación.

Esta clase de estructuras ofrecen la posibilidad de implementar muchos TADs de forma mucho más flexible: no se malgasta espacio a priori, y el único límite en tiempo de ejecución es la cantidad total de memoria disponible.

La cuestión es ¿de qué modo podemos crear y manejar esta clase de estructuras?

Punteros

Los punteros son el medio ofrecido por diversos lenguajes de programación para construir y manejar estructuras de datos dinámicas.

Las constantes y variables que conocemos hasta ahora son objetos que tienen asociado:

— un nombre –identificador–

— un espacio de memoria

— un contenido, que es un valor de un cierto tipo –fijo en el caso de las constantes y cambiante en el caso de las variables–

Podemos utilizar aquí el conocido símil de las variables como recipiente o *cajas* de valores.

Una variable de tipo puntero tiene igualmente asignado un identificador, un espacio de memoria –que es siempre del mismo tamaño, independientemente de a dónde esté apuntado el puntero– y su valor es la dirección de otra variable:

El valor de un puntero[1] es la dirección de otra variable, tal que:

— El nombre de la variable apuntada por un puntero se puede obtener a partir del identificador del puntero. Al igual que en algunos lenguajes, nosotros obtendremos ese identificador colocando el carácter ^ detrás del identificador del puntero.

— El espacio de memoria de la variable a la que apunta un puntero se ubica dinámicamente durante la ejecución. Esta variable no existe mientras que no sea ubicada.

— Es posible "anular" una variable apuntada por un puntero, liberando así el espacio que ocupa. Después de ser anulada, la variable deja de existir.

Gráficamente lo podemos representar de la siguiente forma:

[1] Normalmente se abusa del lenguaje y se dice "puntero" cuando se quiere decir "variable de tipo puntero".

La propiedad fundamental de los punteros es que permiten obtener y devolver memoria dinámicamente durante la ejecución, es decir, crear y destruir variables dinámicamente. Esto resuelve los dos problemas que planteábamos anteriormente sobre las estructuras estáticas: sólo solicitaremos el espacio imprescindible para los datos que en cada momento necesitemos representar.

Veamos con más detalle cómo se declaran las variables de tipo puntero y cómo es posible ubicar y anular las variables a las que apuntan.

Declaración de punteros

Para cualquier tipo τ admitimos que puede formarse un nuevo tipo

---

puntero a τ

Diremos que los valores de este tipo son punteros a variables de tipo τ.

Las variables de tipo puntero se declaran como cualquier otra variable:

```
var
  p : puntero a τ;
```

El identificador de la variable a la que apunta p es p^.

Una vez creada, p^ se comporta a todos los efectos como una variable de tipo τ.

Por ejemplo:

```
tipo
  PunteroEnt = puntero a Ent;
var
  p : PunteroEnt;
```

p^ se comporta como una variable capaz de contener valores enteros.

A nivel de implementación, se tiene que:

— un puntero *p* se realiza como un número natural que es interpretado como una dirección de memoria

— la variable *p*^ apuntada por *p* se ubica en un área de memoria suficientemente amplia (cuánta, depende del tipo τ) que comienza en la dirección apuntada por p.

Operaciones básicas con punteros

Las dos operaciones básicas con los punteros son ubicar la variable a la que apuntan y anular dicha variable, liberando el espacio que ocupa, asignar un puntero a otro y comparar el valor de dos punteros. Insistimos en que la variable a la que apunta un puntero no está disponible hasta que no se ha ubicado explícitamente a través de dicho puntero.

La ubicación del puntero se hace con el siguiente procedimiento predefinido:

```
proc ubicar ( s p : puntero a τ )
```

Este es un procedimiento sobrecargado porque admite como parámetro cualquier tipo de puntero.

El efecto de ubicar es:

— Crear una variable de tipo τ

— Almacenar en *p* la dirección del espacio de memoria asignado a dicha variable.

Nótese que el parámetro *p* es exclusivamente de salida. No se tiene en cuenta si el puntero ya apunta a una variable o no. Si se invocase varias veces sucesivas al procedimiento *ubicar* sobre el mismo puntero *p*, cada vez se reservaría un espacio de memoria diferente.

En el uso de los punteros hay que ser muy cuidadoso pues los errores que se producen por su uso incorrecto suelen ser difíciles de detectar. Usando punteros es relativamente fácil dejar "colgado" al computador: accediendo a zonas de la memoria teóricamente prohibidas. La primera advertencia que hacemos es

---

La variable p^ no puede usarse jamás antes de ejecutar

```
ubicar(p).
```

Salvo si se ha ejecutado una asignación

```
p := q
```

siendo *q* un puntero del mismo tipo, tal que q^ ya tuviese espacio ubicado. En este caso, p y q apuntarán a la misma variable.

Una vez ubicada, p^ se puede utilizar como una variable cualquiera de tipo τ.

La otra característica fundamental de los punteros es que es posible liberar el espacio ocupado por las variables a las que apuntan. Esto también se hace a través de un procedimiento predefinido

---

```
proc liberar ( e p : puntero a τ )
```

El efecto de liberar es:

— destruir la variable apuntada por *p*, liberando el espacio de memoria que ocupaba

Dos advertencias sobre el uso de liberar:

— No se debe liberar una variable que no esté ubicada.

— No se puede utilizar la variable p^ después de ejecutar **liberar(p)**

Es posible también realizar asignaciones entre punteros del mismo tipo:

```
p := q      es válido si p y q son del mismo tipo
```

El efecto de una asignación como esta es que

— *p* pasa a apuntar al mismo sitio al que esté apuntado *q*.

— si antes de la asignación *p* apuntaba a una variable, después de la asignación p^ ya no es un identificador válido para dicha variable.

— si q^ no está ubicada entonces p^ tampoco lo está

— p^ y q^ son dos identificadores de la misma variable.

Advertencia en el uso de la asignación:

— No debe abandonarse nunca la variable apuntada por un puntero sin liberar previamente el espacio que ocupa, a menos que la variable sea accesible desde otro puntero.

También permitimos realizar comparaciones entre punteros:

```
p == q   p /= q    son expresiones válidas si p y q son del mismo tipo
```

Nótese que comparamos direcciones y no los valores de las variables a las que apuntan los punteros, p^ y q^.

Como conclusión de este apartado, vamos a seguir gráficamente con detalle un ejemplo del uso de las operaciones sobre punteros.

```
var
  p, q : puntero a Ent;

ubicar(p);
p^ := -3; p^:= -2*p^;
ubicar(q);
q^ := -5; q^ := p^ + q^;
p := q;     ← error
liberar(p);
liberar(q); ← error
```

Vemos cómo justo antes de empezar la ejecución ya están ubicadas las variables *p* y *q*, y que ambas contienen inicialmente *basura* (#).

3.5.2 Construcción de estructuras de datos dinámicas

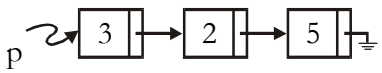
La utilidad de los punteros para construir colecciones de datos se obtiene cuando un puntero *p* apunta a una variable de tipo registro que contiene en uno de sus campos un puntero del mismo tipo que *p*. Pueden usarse entonces los punteros para encadenar entre sí registros, formando estructuras dinámicas, ya que **ubica** y **libera** podrán usarse en tiempo de ejecución para añadir o eliminar registros de la estructura.

Por ejemplo:

```
tipo
  Enlace = puntero a Nodo;
  Nodo = reg
    num : Ent;
    sig : Enlace
  freg;
```

Nótese la referencia adelantada al tipo al que apunta el puntero, esto es necesario porque estos dos tipos son mutuamente dependientes. Normalmente este tipo de dependencia sólo se permite cuando uno de los dos tipos es un puntero.

Con este tipo de punteros podemos crear estructuras como



Como resulta evidente en el anterior ejemplo, para poder construir estructuras dinámicas como esta es necesario poder indicar de alguna forma el final de las estructuras. Esto se hace con un valor especial de tipo puntero: el *puntero vacío*.

---

definimos **nil** como una constante que admite el tipo

```
puntero a τ
```

para cualquier tipo τ, y que representa a un puntero ficticio que no apunta a ninguna variable. Por lo tanto:

— nil^ está indefinido

— **ubicar**(nil), **liberar**(nil) no tienen sentido

Armados con la constante **nil** ya podemos construir la estructura dinámica que presentamos antes gráficamente:

```
var
  p, q : Enlace;

ubicar(q);
q^.num := 5;
q^.sig := nil;
p := q;

% representación gráfica del estado actual

ubicar(q);
q^.num := 2;
q^.sig := p;
p := q;

% representación gráfica del estado actual

ubicar(q);
q^.num := 3;
q^.sig := p;
p := q;

% representación gráfica del estado final
```

3.5.3 Implementación de TADs mediante estructuras de datos dinámicas

Los punteros son una técnica de programación de bajo nivel, cuyo uso no sólo es bastante engorroso, sino que además puede conducir a estructuras muy intrincadas –nada impide pensar en nodos con varios punteros cada uno, enlazándose entre sí de forma arbitrariamente compleja–. Por consiguiente:

Advertencia: el uso indiscriminado de punteros es peligroso.

Disciplina: nosotros nos limitaremos a usar punteros dentro de los módulos de implementación de TADs. Esta restricción permite sacar partido de la potencia de los punteros sin sacrificar las ventajas de una metodología de la programación basada en la abstracción. Además, está de acuerdo con la metodología de utilizar tipos ocultos. Dado que los punteros son un mecanismo para representar tipos, y puesto que sólo permitimos acceder a la representación interna de los tipos en los módulos que los implementan, es razonable prohibir el uso de punteros fuera de este ámbito.

Como ejemplo de la implementación de TADs mediante estructuras dinámicas vamos a desarrollar una implementación para las pilas.

Ejemplo: implementación dinámica de las pilas

La representación gráfica de lo que pretendemos implementar:

Los registros se suelen llamar *nodos*. La representación de la pila es el puntero *p* que señala al nodo cima. Cada nodo contiene un dato e_i que representa un elemento, y un puntero que señala al nodo situado inmediatamente debajo de él en la pila.

Tipo representante

```
tipo
  Enlace = puntero a Nodo;
  Nodo = reg
    elem : Elem;
    sig : Enlace;
  freg;
```

Nótese que con esta representación el único límite impuesto a las dimensiones de la pila es el tamaño de la memoria de la computadora.

Invariante de la representación

Sea p : Enlace, definimos

$$
\begin{aligned}
& R_{Pila[Elem]}(p) \\
\Leftrightarrow_{def} \\
& p = nil \vee \\
& (p \neq nil \wedge ubicado(p) \wedge \\
& \quad R_{Elem}(p\text{^}.elem) \wedge R_{Pila[Elem]}(p\text{^}.sig) \wedge \\
& \quad p \notin cadena(p\text{^}.sig))
\end{aligned}
$$

donde la función auxiliar cadena permite especificar que no hay dos punteros apuntando al mismo nodo:

```
cadena(nil) =def ∅
cadena( p ) =def {p} ∪ cadena(p^.sig) si p ≠ nil
```

La idea de este invariante de la representación es que de *p* arranca una cadena de punteros finita, sin repeticiones y acabada en nil, cada uno de los cuales apunta a la representación correcta de un elemento. Lo que no veo es cómo se indica en este invariante que la cadena ha de terminar.

Función de abstracción

Sea p tal que $R_{Pila[Elem]}(p)$

```
A_Pila[Elem](p) = PilaVacía  si p = nil
A_Pila[Elem](p) = Apilar( A_Elem(p^.elem), A_Pila[Elem](p^.sig) ) si p ≠ nil
```

Nótese que esta definición recursiva termina porque |cadena(p)| decrece.

Implementación de las operaciones

Podemos considerar dos posibilidades: implementación como funciones o como procedimientos. Vamos a concentrarnos primero en la implementación como funciones.

```
módulo impl PILA[ELEM]

% aquí no hace falta importar ELEM porque ya está importado en el de
especificación

  privado
    tipo
      Enlace = puntero a Nodo;
      Nodo = reg
        elem : Elem;
        sig : Enlace;
      freg;
      Pila[Elem] = Enlace;

func PilaVacía dev xs : Pila[Elem];
{ P0 : Cierto }
inicio
  xs := nil
{ Q0 : R(xs) ∧ A(xs) =PILA[ELEM] PilaVacía }
  dev xs
ffunc
```

```
func Apilar ( x : Elem; xs : Pila[Elem] ) dev ys : Pila[Elem];
{ P₀ : R(x) ∧ R(xs) }
inicio
  ubicar(ys);
  ys^.elem := x;
  ys^.sig := xs;
{ Q₀ : R(ys) ∧ A(ys) =PILA[ELEM] Apilar(A(x), A(xs)) }
  dev ys
ffunc
```

Nótese que esta implementación de apilar introduce compartición de estructura entre *xs* e *ys*. Otra opción sería realizar una copia de *xs*. A continuación haremos algunas consideraciones sobre estas cuestiones.

```
func desapilar( xs : Pila[Elem] ) dev ys : Pila[Elem];
{ P₀ : R(xs) ∧ not esVacía( A(xs) ) }
inicio
  si esVacía(xs)
    entonces
      error("no se puede desapilar de la pila vacía")
    sino
      ys := xs^.sig
  fsi
{ Q₀ : R(ys) ∧ A(ys) =PILA[ELEM] despilar(A(xs)) }
  dev ys
ffunc
```

Otra vez compartición de estructura. Nótese también que no estamos anulando la cima de *xs*.

```
func cima ( xs : Pila[Elem] ) dev x : Elem;
{ P₀ : R(xs) ∧ not esVacía( A(xs) ) }
inicio
  si esVacía(xs)
    entonces
      error("La pila vacía no tiene cima")
    sino
      x := xs^.elem
  fsi
{ Q₀ : R(x) ∧ A(x) =PILA[ELEM] cima(A(xs)) }
  dev x
ffunc

func esVacía ( xs : Pila[Elem] ) dev r : Bool;
{ P₀ : R(xs) }
inicio
  r := xs == nil
```

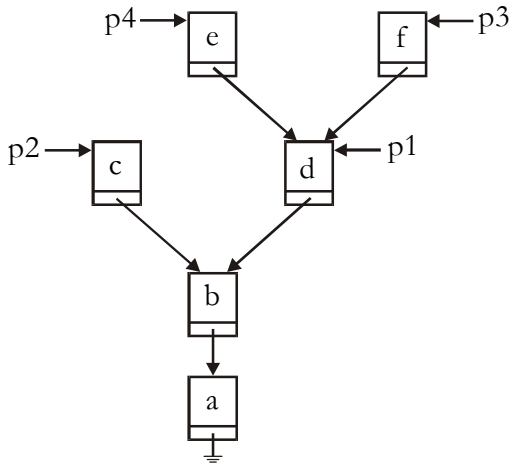
```
{ Q₀ : r ↔ esVacía( A(xs) ) } ≡ R_Bool(r) ∧ A_Bool(r) =_PILA[ELEM] esVacía(
A_Pila[Elem](xs) )
  dev r
ffunc
proc error( e Cadena : Vector[1..límite] de Car );
{ P₀ : Cierto }
inicio
  escribir(Cadena);
  abortar
{ Q₀ : Falso }
fproc
fmódulo
```

En al implementación de la operación apilar hemos introducido compartición de estructura; la razón para ello ha sido evitar el coste que supone realizar una copia del parámetro de la función. sin embargo la compartición de estructura nos ha obligado a no anular el elemento que desapilamos porque puede estar formando parte de otra pila.

Veamos un ejemplo:

```
var
  p, p1, p2, p3, p4 : Pila[Car];
  p := PilaVacía();
  p1 := Apilar( 'b', Apilar('a', p) );
  p2 := Apilar( 'c', p1 );
  p3 := Apilar( 'd', p1 );
  p4 := Apilar( 'e', p3 );
  p3 := Apilar( 'f', p3 );
  p1 := desapilar( p4 );
```

El estado al que se llega es:



¿Qué ocurre si ahora hacemos lo siguiente?

```
p  := PilaVacía();
```

```
p1 := PilaVacía(); p2 := PilaVacía();
p3 := PilaVacía(); p4 := PilaVacía();
```

Toda la memoria dinámica que ocupaban las pilas ha pasado a estar inaccesible: se ha generado *basura* en la memoria dinámica.

La solución a este problema pasa, en primer lugar, por añadir al TAD una operación que se encargue de liberar todo el espacio ocupado por un valor de ese tipo. Habrá que invocar a esta operación cada vez que deseemos reinicializar una variable con un nuevo valor. A esta operación la denominamos *anular* y en el caso de las pilas se implementaría de la siguiente manera:

```
proc anular( es xs : Pila[Elem] );
{ P₀ : R(xs) }
var
  p : Enlace;
inicio
  it not esVacía(xs) →
    p := xs;
    xs := xs^.sig;
    liberar(p)
  fit
{ Q₀ : R(xs) ∧ A(xs) =PILA[ELEM] PilaVacía }
fproc
```

Sin embargo esta solución no es completa pues los elementos también pueden ser estructuras dinámicas, en cuyo caso sería necesario anularlas:

```
proc anular( es xs : Pila[Elem] );
{ P₀ : R(xs) }
var
  p : Enlace;
inicio
  it not esVacía(xs) →
    p := xs;
    xs := xs^.sig;
    ELEM.anular(p^.elem);
    liberar(p)
  fit
{ Q₀ : R(xs) ∧ A(xs) =PILA[ELEM] PilaVacía }
fproc
```

De esta forma estamos exigiendo que los posibles parámetros de PILA[ELEM] implementen una operación *anular*. Esta restricción deberíamos expresarla en la definición de la clase de tipos. Puede plantearse que esa operación no tiene sentido si el tipo se ha implementado con una es-

tructura estática. Sin embargo, siguiendo la filosofía de que el usuario no debe conocer los detalles de la representación interna, todos los TAD deberían exportar una operación de anular que en el caso de las implementaciones estáticas no haría nada, pues al liberar el nodo ya se devuelve el espacio ocupado por el valor. Otro problema son los tipos predefinidos, para los cuales suponemos que está definida la operación *anular* y que se comporta como en el caso de las estructuras estáticas.

En cuanto al uso de *anular*

Se debe invocar *anular* antes de reinicializar una variable con otro valor. En el ejemplo anterior, se debería sustituir:

```
p1 := PilaVacía();
```

por

```
anular(p1);
p1 := PilaVacía();
```

también es necesario invocar *anular* sobre las variables locales de los procedimientos justo antes de la salida del procedimiento, siempre y cuando la variable local no comparta estructura con un parámetro de salida de dicho procedimiento.

Evidentemente sólo se debe invocar a *anular* si la variable contenía algún valor.

Pero, ¿qué ocurre con la estructura compartida? En el anterior ejemplo, si anulásemos *p1* antes de asignarle otro valor, corromperíamos el estado de todas las demás pilas, que están compartiendo estructura con *p1*.

Por lo tanto, si permitimos la compartición de estructura no podemos anular las variables, pero si no podemos anular las variables puede ocurrir que dejemos basura en la memoria dinámica. ¿cuál es la solución? Algunos lenguajes incluyen un mecanismo que se denomina "recogida automática de basura". Este mecanismo controla cuántas referencias hay a cada espacio de la memoria, de forma que cuando detecta que una zona de la memoria ya no es accesible desde ningún puntero, la libera automáticamente. En los lenguajes que tienen recogida de basura no es necesario preocuparse por las anulaciones, pues estas tendrán lugar automáticamente. El inconveniente de la recogida automática de basura es que este mecanismo ralentiza la ejecución de los programas; como es habitual, los mecanismos que facilitan la vida de los programadores tienen un cierto coste computacional.

Si queremos mantener la implementación funcional, la solución por la que debemos optar entonces es la de realizar copias de los parámetros de las operaciones para evitar así la compartición de estructura –aunque esta implementación de las pilas permite la compartición de estructura, salvo por el problema de la anulación, en algunas implementaciones de TADs es obligatorio realizar las copias de los parámetros porque si no los argumentos dejarían de ser representantes válidos del tipo en cuestión–. Sin embargo, esta solución aumenta innecesariamente el coste de las operaciones, en muchos casos de manera ridícula:

Funcionaría bien en un caso como este:

```
p2 := Apilar( 'a', p1 );
```

p2 y p1 no compartirían estructura. ¿Pero qué ocurre si lo que queremos hacer es algo como esto?

```
p1 := Apilar('a', p1);
```

hemos generado basura pues no hemos anulado el valor original de p1, por lo tanto habría que hacer algo tan artificioso como esto:

```
p2 := p1;
p1 := Apilar( 'a', p1 );
anular(p2);
```

Este último ejemplo nos sugiere otra solución que es la que realmente emplearemos: implementar las generadoras y modificadoras como procedimientos, de forma que consideremos a los valores como objetos mutables accesibles a través de una única referencia.

Realización de las operaciones como procedimientos

Planteamos la implementación de las generadoras y modificadoras como procedimientos:

```
proc PilaVacía( s xs : Pila[Elem] );
{ P₀ : Cierto }
inicio
  xs := nil
{ Q₀ : R(xs) ∧ A(xs) =PILA[ELEM] PilaVacía }
fproc

proc Apilar ( e x : Elem; es xs : Pila[Elem] );
{ P₀ : xs = XS ∧ R(x) ∧ R(xs) }
var
  p : Enlace;
inicio
  ubicar(p);
  p^.elem := x;
  p^.sig := xs;
  xs := p
{ Q₀ : R(xs) ∧ A(xs) =PILA[ELEM] Apilar(A(x), A(XS)) }
Fproc


proc desapilar( es xs : Pila[Elem] );
{ P₀ : xs = XS ∧ R(xs) ∧ not esVacía( A(xs) ) }
```

```
var
  p : Enlace;
inicio
  si esVacía(xs)
    entonces
      error("no se puede desapilar de la pila vacía")
    sino
      p := xs;
      xs := xs^.sig;
      % ELEM.anular(p^.elem);
      liberar(p)
  fsi
{ Q₀ : R(xs) ∧ A(xs) =PILA[ELEM] desapilar(A(XS)) }
fproc
```

En esta implementación podríamos dejar las operaciones *cima* y *esVacía* como funciones. Sin embargo, esto podría dar problemas para la operación *cima* si los elementos fuesen de un tipo que estuviese implementado con memoria dinámica, porque daría lugar a compartición de estructura, la solución sería devolver una copia, o que el cliente del TAD hiciera una copia si lo considerase necesario. En el siguiente apartado comentaremos la necesidad de incorporar una operación de copia en todos los TADs.

En definitiva, no prohibimos definitivamente la compartición de estructura ya que en algunos casos para evitarla habría que incurrir en graves ineficiencias. Por ejemplo, un árbol se construye a partir de la información de la raíz y dos subárboles. Para evitar la compartición, se deberían realizar copias de los subárboles, sin embargo, en la mayoría de los casos estos se han creado con el único propósito de formar parte de un árbol mayor. Como implementadores del TAD debemos informar del hecho y dejar en manos del cliente la decisión de si desea realizar copias o no.

3.5.4 Problemas del uso de la memoria dinámica en la implementación de TADs

Existen algunos inconvenientes generales en el uso de memoria dinámica:

— Agotamiento inesperado de la memoria. Evidentemente la memoria dinámica no es ilimitada. Puede ocurrir que en un punto de la ejecución ya no sea posible ubicar más variables dinámicas. Una forma de paliar este problema es comprobar antes de ubicar una nueva variable si existe espacio suficiente, para en caso negativo no intentar ubicarla. Para saber si la ubicación tendrá éxito los lenguajes de programación suelen incorporar operaciones predefinidas para obtener la memoria disponible –en Pascal *MemAvail*– y el tamaño de un cierto tipo de datos –en Pascal *SizeOf*–.

— Los errores en el uso de los punteros pueden provocar fallos irrecuperables –cuelgues del ordenador– y difíciles de detectar

— La gestión de la memoria dinámica puede aumentar el coste de algunos algoritmos –tiempo empleado en ubicación y anulación de las estructuras–. Suponiendo que la crea-

ción y destrucción de estructuras dinámicas lleva más tiempo que la creación y destrucción de estructuras estáticas, de la que se encarga automáticamente el compilador.

Existen además problemas específicos de la implementación de TADs. Lo ideal es poder resolver estos problemas de forma transparente para los clientes del TAD de forma que ni siquiera tengan que saber si la implementación del TAD es estática o dinámica. Sin embargo esto no es realista, pues resolverlo de esta forma nos obligaría a realizar algunas operaciones de manera innecesariamente ineficiente –necesidad de realizar copias y anulaciones–. La solución por la que optamos es dejar una parte de las decisiones en manos del cliente del TAD, que, por lo tanto, debe ser consciente de si se trata de una representación estática o dinámica.

Almacenamiento de los datos en disco

Los datos representados por estructuras dinámicas no pueden escribirse y leerse en archivos directamente, porque los valores de los punteros –direcciones– en cada ejecución particular serán diferentes.

La escritura de estructuras dinámicas se limitará a escribir la información –los elementos– sin guardar los valores de los punteros. La lectura deberá reconstruir las estructuras dinámicas, a partir de los datos leídos del archivo, solicitando nueva memoria dinámica donde almacenar los datos leídos.

Los TADs deberán exportar operaciones de escritura y lectura de archivo siempre que estas puedan resultar necesarias para los usuarios:

```
guardar( x, a )    recuperar( x, a )
```

Asignación

Si *x, y* son variables de un mismo tipo abstracto, el efecto de la asignación es diferente dependiendo de si el tipo representante es estático o dinámico.

— En una implementación estática la asignación implica copia

```
x := y
```

causa que se copie en la zona de memoria designada por *x* el valor almacenado en la zona de memoria designada por *y*. Se mantiene la independencia entre los dos valores, las modificaciones de uno de ellos no afectarán al otro.

— En una implementación dinámica la asignación implica compartición de estructura

```
x := y
```

causa que *x, y* apunten a la misma estructura dinámica. Los cambios realizados en la estructura a través de una variable afectarán a la estructura puntada por la otra.

Por ejemplo:

```
var
  p, q : Pila[Nat];
```

```
PilaVacía(p);
Apilar(1, p);
q := p;
despilar(q);
```

p dejaría de ser un representante válido pues estaría apuntando a un nodo que ha sido liberado al desapilar de *q*.

Una solución a este problema sería imponer la disciplina de que no se pueden realizar asignaciones entre valores de TADs, y que en caso de ser necesarias se exporte en el TAD una operación encargada de ello. Esta operación tendría semántica de copia. Sin embargo, esta solución es demasiado restrictiva pues en algunos casos al cliente puede no preocuparle la compartición de estructura. Lo que debemos hacer entonces es exportar en todos los TADs una operación que permita realizar copias de los valores del TAD, para que así el cliente pueda elegir entre las dos opciones.

```
proc copiar ( e x : τ; s y : τ )
```

de forma que se pueda sustituir

```
x := y   por    copiar(y, x)
```

En el ejemplo de las pilas la operación de copia se puede implementar de la siguiente forma:

```
proc copiar ( e xs : Pila[Elem]; s ys : Pila[Elem] );
{ P₀ : R(xs) }
var
  p, q1, q2 : Enalce;
inicio
  si esVacía(xs)
    entonces
      PilaVacía(ys)
    sino
      ubicar( ys );
      ys^.elem := xs^.elem; % ELEM.copiar( xs^.elem, ys^.elem )
      p := xs^.sig;
      q1 := ys;
      it p /= nil →
        ubicar(q2);
        q2^.elem := p^.elem; % ELEM.copiar( p^.elem, q2^.elem )
      q1^.sig := q2;
      p := p^.sig;
```

```
      q1 := q2
    fit;
    q1^.sig := nil
  fsi
{ Q₀ : R(ys) ∧ A(xs) = A(ys) }
fproc
```

En el caso de estructuras estáticas *copiar* se implementaría como una simple asignación.

Parámetros de entrada

Si alguna de las operaciones de un TAD se realiza como función o procedimiento con parámetro de entrada x : τ, y si el tipo representante de τ es dinámico, el hecho de que el parámetro sea de entrada no es suficiente para garantizar que el valor apuntado no se modifica; sólo queda protegido el puntero –dirección–, pero no la estructura apuntada. Debemos imponernos por tanto la disciplina de no modificar en ningún caso el valor de los parámetros de entrada.

Otro problema es que a través de los parámetros de entrada se puede dar lugar otra vez a compartición de estructura. Por ejemplo, así ocurre con la implementación procedimental que hemos dado para la operación *Apilar* donde aparece la asignación

```
…
p^.elem := x
…
```

si Elem está implementado con una estructura dinámica, esto daría lugar a compartición de estructura:

```
var
  p : Pila[Nat];
  pp : Pila[Pila[Nat]];

PilaVacía(pp);
PilaVacía(p);
Apilar(1, p);
apilar( p , pp );   ← compartición de estructura
desapilar( p );     ← se ha corrompido pp
```

Una solución sería realizar copias de los parámetros de entrada antes de incorporarlos a la estructura. Sin embargo, de nuevo, esto produce una ineficiencia intolerable. La solución es advertir de esta circunstancia a los clientes del TAD, para que sean ellos quienes decidan si conviene o no realizar una copia de los parámetros antes de realizar la invocación.

Igualdad

El tipo representante de un tipo abstracto puede no admitir el uso de la comparación de igualdad. Y aunque la admitiese, la identidad entre valores del tipo representante en general no corresponderá a la igualdad entre los valores abstractos representados. Este problema se presenta por igual en las implementación estáticas –como ya vimos en el ejemplo de la implementación estática de las pilas– como en las dinámicas; aunque en las estáticas puede o no ocurrir mientras que en las dinámicas sucede siempre.

La solución radica en que el TAD exporte una operación de comparación:

```
func iguales( x, y : τ ) dev r : Bool;
```

Puede ocurrir que la propia signatura de la especificación algebraica del TAD incluyese una operación de igualdad. si esto no es así, y si el módulo de datos tampoco la exporta, los usuarios del TAD deberán entender que no se les permite realizar comparaciones de igualdad.

Como ejemplo, vemos cómo se implementaría la función iguales para el caso de las pilas:

```
func iguales( xs, ys : Pila[Elem] ) dev r : Bool;
{ P₀ : R(xs) ∧ R(ys) }
var
  p, q : Enlace;
inicio
  <p, q> := <xs, ys>;
  r := cierto;
  it (p /= nil) and (q /=nil) and r →
    r := p^.elem == q^.elem;  % r := ELEM.iguales( p^.elem, q^.elem )
    <p, q> := <p^.sig, q^.sig>
  fit;
  r := r and (p == nil) and (q == nil);
{ Q₀ : r ↔ A(xs) = A(ys) }
  dev r
ffunc
```

Generación de basura

Como ya hemos indicado al comentar la implementación dinámica de las pilas, puede ocurrir que zonas de la memoria dinámica estén marcadas como ocupadas pero no sean accesibles desde ningún puntero.

Esta situación puede ocurrir con:

— Variables locales a los procedimientos que almacenan referencias a estructuras dinámicas. Dichas estructuras se convertirán en basura a la salida del procedimiento, ya que, nor-

malmente, el compilador recoge automáticamente la memoria ocupada por las variables estáticas, pero no así la memoria dinámica.

— Variables que se reinicializan al ser utilizadas como parámetro de salida de un procedimiento. La estructura a la que apuntaba la variable anteriormente se convierte en basura.

— Variables que se reinicializan mediante una asignación. La estructura a la que apuntaba la variable anteriormente se convierte en basura.

En estos tres casos no tiene que generarse basura necesariamente, sólo será así cuando la variable afectada sea la única referencia a la correspondiente estructura dinámica, es decir, si dicha estructura no está compartida por más de una referencia.

La solución como ya vimos más arriba radica en que el módulo que implementa el TAD exporte una operación que libere la estructura dinámica que representa al valor:

```
proc anular( es x : τ );
```

Para los TAD que implementen el tipo con una estructura estática operación anular no hará nada.

Los clientes del TAD deberán invocar a la operación anular:

— sobre las variables locales antes de terminar cualquier procedimiento

— antes de cualquier reinicialización de una variable

Siempre y cuando la variable anulada sea la única referencia a la estructura dinámica.

Referencias perdidas

Este problema se produce cuando un puntero *p* queda con un valor diferente de *nil*, pero habiendo sido liberado el espacio al que apunta. Por ejemplo, la siguiente serie de acciones haría que la referencia de *p* quedase perdida:

```
ubicar(q); p := q; liberar(q)
```

La solución radica en no liberar ninguna estructura que esté siendo compartida por más de una referencia.

Estructura compartida

Este problema ha aparecido en prácticamente todos los que hemos descrito hasta ahora. Decimos que hay estructura compartida cuando dos o más punteros comparten todo o una parte de la estructura dinámica a la que apuntan. El problema es que los cambios realizados en la estructu-

ra a través de un puntero afectan a la estructura apuntada por los demás, pudiendo incluso ocurrir que se deje de verificar el invariante de la representación.

En los anteriores puntos ya hemos ido viendo las situaciones que pueden dar lugar a compartición de estructura:

— asignaciones entre punteros

— operaciones que construyen una estructura dinámica añadiéndole nodos a otra ya existente.

Los problemas que plantea la compartición:

— Complica la anulación pues no se deben anular estructuras compartidas

— Puede provocar efectos colaterales: las modificaciones de una variable afectan a otras

---

Se pueden tomas dos posturas ante la compartición de estructura:

— Prohibir la compartición de estructura. La ventaja es que no se producen efectos colaterales inesperados y el inconveniente es que se aumenta el número de copias y anulaciones necesarias con lo que se degrada, en muchos casos de forma intolerable, la eficiencia de los algoritmos.

— Permitir la compartición de estructura.

  — En los lenguajes con recogida automática de basura no existe el problema de la anulación. El usuario es responsable de realizar las copias que sea necesario.

  — En los lenguajes sin recogida automática de basura el cliente del TAD es el responsable de su buen uso, realizando las copias y anulaciones que sea necesario.

3.6 Ejercicios

Introducción a la programación con TADs

131. En cada uno de los apartados siguientes, escribe cabeceras de funciones que especifiquen el comportamiento deseado para las operaciones del TAD que se sugiere en cada caso, sin hacer ninguna suposición acerca de la representación concreta de los datos del TAD.

(a) TAD de los números complejos, con el tipo *Complejo* y operaciones adecuadas para construir números complejos y calcular con ellos.

(b) El TAD para multiconjuntos de números enteros, con el tipo *MCjtoEnt* y operaciones que permitan: crear un multiconjunto vacío; añadir un nuevo elemento a un multiconjunto; reconocer si un multiconjunto es vacío; determinar el elemento mínimo de un multiconjunto no vacío; y finalmente, quitar una copia del elemento mínimo de un multiconjunto no vacío.

(c) TAD para trabajar con fechas, disponiendo de un tipo *Fecha* y operaciones que permitan: crear una fecha; determinar el día del mes, el día de la semana, el mes y el año de una fecha dada; calcular la distancia en días entre dos fechas dadas; "sumar" un entero a una fecha dada; y calcular la primera fecha correspondiente a un día de la semana, mes y año dados.

(d) TAD para polinomios en una indeterminada con coeficientes enteros, con un tipo *Poli* y operaciones que permitan: crear el polinomio nulo; añadir un nuevo monomio a un polinomio; sumar y multiplicar polinomios; evaluar un polinomio para un valor entero dado de la indeterminada; calcular el coeficiente asociado a un exponente dado en un polinomio dado; y reconocer si un polinomio dado es nulo.

132. La función *mayores* que se especifica a continuación resuelve el problema de calcular los k mayores elementos de un vector de enteros dado (con posibles repeticiones). Impleméntala por medio de un algoritmo iterativo, suponiendo disponible y utilizable el TAD *MCjtoEnt* del ejercicio 131(b). Observa que no necesitas saber cuál es la representación concreta de los multiconjuntos.

```
func mayores ( k : Nat; v : Vector[1..N] de Ent ) dev m : MCjtoEnt;
{ P₀ : 1 ≤ k ≤ N ∧ N ≥ 1 }
{ Q₀ : m contiene los k elementos mayores de v[1..N] }
ffunc
```

133. Para cada uno de los TADs del ejercicio 131, sugiere una o varias posibilidades para representar los datos del TAD usando tipos de datos conocidos (tales como registros, vectores, etc.). ¿Influye la elección de la representación en la eficiencia de las operaciones del TAD? ¿De qué manera?

Especificación algebraica de TADs

134. Especifica algebraicamente un TAD BOOL que ofrezca el tipo *Bool* de los valores booleanos, junto con las operaciones constantes *Cierto*, *Falso* y las operaciones booleanas *not*, (*and*) y (*or*)

135. Usando BOOL, especifica algebraicamente un TAD NAT que ofrezca el tipo *Nat* de los números naturales, con operaciones *Cero*, *Suc*, (+), (*), (==), (/=), (≤), (≥), (<) y (>).

136. Escribe la *signatura* de un TAD REAL que ofrezca el tipo *Real* de los números reales, junto con algunas operaciones básicas.

†137. Usando el TAD REAL del ejercicio anterior, especifica algebraicamente un TAD COMPLEJO que ofrezca el tipo *Complejo* junto con las operaciones consideradas en el ejercicio 131(a).

138. Usa *razonamiento ecuacional* en NAT para demostrar la validez de las ecuaciones siguientes:

(a) `Suc(Cero) + Suc(Cero) = Suc(Suc(Cero))`

(b) `Suc(Cero) * Suc(Cero) = Suc(Cero)`

(c) `( Suc(Cero) * Suc(Cero) == Suc(Cero) ) = Cierto`

(d) `( Suc(Suc(Cero)) + Suc(Suc(Cero)) ≤ Suc(Suc(Suc(Cero))) ) = Falso`

139. Sea T un TAD con tipo principal τ, Dos términos `t, t': τ` se llaman T-*equivalentes* (en símbolos $t =_T t'$), si la ecuación `t = t'` se puede deducir a partir de los axiomas ecuacionales de T.

(a) Comprueba que la T-equivalencia es una relación de equivalencia.

(b) Encuentra varios términos que sean NAT-equivalentes a `Suc(Cero)`.

140. Observa las especificaciones de los TADs BOOL, NAT y COMPLEJO. En cada caso, clasifica las operaciones en tres clases: *generadoras*, *modificadoras* y *observadoras*.

141. Dado un TAD T con tipo principal τ, se llaman *términos generados* a aquellos términos de tipo τ que se pueden construir usando únicamente operaciones generadoras de T (y términos generados de otros TADs usados por T, si es necesario). Indica cuáles son los términos generados de los TADs BOOL, NAT y COMPLEJO.

†142. Sea T un TAD con tipo principal τ. Se dice que el conjunto de operaciones generadoras de T es *suficientemente completo* si es cierta la siguiente condición: para todo término cerrado t de tipo τ debe existir un término cerrado y generado t' de tipo τ que sea T-equivalente a t.

Demuestra que los conjunto de generadoras de los TADs BOOL y NAT son suficientemente completos.

Importante: La especificación de cualquier TAD siempre debe construirse de manera que el conjunto de operaciones generadoras elegido sea suficientemente completo.

143. Usa *razonamiento inductivo* en NAT para demostrar la validez de las ecuaciones siguientes:

(a) `Cero + y = y`

(b) `Suc(x) + y = Suc(x+y)`

(c) `x + y = y + x`

Pista: Para (a), (b) usa inducción sobre y. Para (c) usa inducción sobre x, aplicando (a), (b).

144. Usa razonamiento inductivo en BOOL para demostrar que las operaciones (*and*) y (*or*) son conmutativas; es decir, demuestra que para `x, y : Bool` cualesquiera se cumplen las ecuaciones:

(a) `x and y = y and x`

(b) `x or y = y or x`

†**145.** Construye un modelo de BOOL tal que el soporte del tipo *Bool* sea un conjunto de tres elementos: *Cierto*, *Falso* y *Nodef* (que quiere representar un valor booleano "indefinido"). Interpreta las operaciones *not*, (*and*), y (*or*) en este modelo de manera que las ecuaciones de BOOL se cumplan. Compara con el *modelo de términos* de BOOL.

†**146.** Sea T un TAD con tipo principal τ cuya especificación algebraica usa otro TAD S con tipo principal *s*. Se dice que T *protege* a S si se cumplen las dos condiciones siguientes:

(i) T *no introduce basura* en S, es decir: para todo término cerrado $t : s$ que se pueda construir en T, existe un término generado $t' : s$ que ya se podía construir en S y tal que $t =_T t'$.

(ii) T *no introduce confusión* en S, es decir: Si $t, t' : s$ son términos cerrados generados que ya se podían construir en S, y se tiene $t =_T t'$, entonces ya se tenía $t =_S t'$.

(a) Demuestra que el siguiente TAD que usa a BOOL no protege a BOOL, porque introduce basura y confusión en BOOL:

```
tad PERSONA
   usa
      BOOL
   tipo
      Persona
   operaciones
      Carmen, Roberto, Lucía, Pablo: → Persona                    /*
gen */
      feliz: Persona → Bool                                       /*
obs */
   ecuaciones
      feliz(Carmen)     = feliz(Roberto)
      feliz(Pedro)    = feliz(Lucía)
      feliz(Pedro)    = not feliz(Pablo)
      feliz(Lucía)    = Cierto
      feliz(Pablo)    = Cierto
ftad
```

(b) Construye un TAD que use a BOOL introduciendo basura, pero no confusión.

(c) Construye un TAD que use a BOOL introduciendo confusión, pero no basura.

†(d) Comprueba que NAT, que usa a BOOL, deja a BOOL protegido.

Importante: Siempre que un TAD T se especifique usando otro TAD S ya especificado anteriormente, debe hacerse de manera que S quede protegido.

147. Usando NAT, especifica un TAD ENT que ofrezca el tipo *Ent* de los números enteros, con operaciones *Cero*, *Suc*, *Pred*, (+), (–), (*) y (^) debe especificarse de modo que represente la operación de exponenciación con base entera y exponente natural.

†148. Sea T un TAD con tipo principal τ. Se dice que las operaciones generadoras de T son *no libres* si existen términos generados no idénticos t y t' de tipo τ, que sean T-equivalentes. Por el contrario, se dice que las operaciones generadoras de T son *libres* si no es posible encontrar términos generados no idénticos t, t' de tipo τ que sean T-equivalentes.

(a) Demuestra que las generadoras del TAD ENT no son libres.

(b) Demuestra que las generadoras de los TADs BOOL y NAT son libres.

149. Sea T un TAD con tipo principal τ. Se llama *sistema de representantes canónicos* a cualquier subconjunto TC_τ del conjunto TG_τ de todos los términos generados y cerrados de tipo τ, tal que para cada $t \in TG_\tau$ exista un único $t' \in TC_\tau$ tal que $t =_T t'$. Construye sistemas de representantes canónicos para los TADs BOOL, NAT y ENT. Observa que en el caso de ENT hay varias formas posibles de elegir los representantes canónicos, debido a que las generadoras no son libres.

†150. Especifica algebraicamente un TAD ENT-ORD que *enriquezca* el TAD ENT añadiendo las operaciones de comparación (==), (/=), (≤), (≥), (<) y (>).

Pista: Es necesario utilizar una *operación privada* que reconozca los enteros no negativos. Esta operación *noNeg*: Ent → Bool se tiene que especificar usando *ecuaciones condicionales*.

151. Especifica algebraicamente un TAD POLI que ofrezca el tipo *Poli* de los polinomios en una indeterminada con coeficientes enteros, junto con las operaciones consideradas en el ejercicio 131(d). Observa que las operaciones generadoras de este TAD no son libres, y que resulta conveniente utilizar operaciones privadas en la especificación.

TADs genéricos

152. Llamaremos *clase de tipos* a una familia de TADs caracterizados por disponer de determinados tipos y operaciones, indicados en la especificación de la clase. Especifica las siguientes clases de tipos:

(a) ANY: Clase formada por todos los TADs que tengan un tipo principal *Elem*.

(b) EQ: Clase formada por todos los TADs de la clase ANY que posean además operaciones de igualdad y desigualdad (==) Y (/=).

(c) ORD: Clase formada por todos los TADs de la clase EQ que posean además operaciones de orden (≤), (≥), (<) y (>).

153. Los TADs que pertenecen a una cierta clase de tipos se llaman *miembros* o *ejemplares* de la clase. Indica ejemplares de las clases ANY, EQ y ORD. Observa que diferentes TADs miembros de una misma clase de tipos pueden tener diferentes signaturas.

154. Los TADs *parametrizados* (también llamados *genéricos*) tienen un parámetro formal que representa a otro TAD, obligado a ser miembro de cierta clase de tipos. Construye una especificación de un TAD genérico CJTO[E :: EQ] que ofrezca el tipo *Cjto[Elem]* formado por los conjuntos (finitos) de elementos de tipo *Elem* (dado por el TAD parámetro E), junto con operaciones para crear un conjunto vacío, añadir y quitar un elemento a un conjunto, reconocer el conjunto vacío, y reconocer si un elemento dado pertenece a un conjunto dado.

155. Una vez especificado un TAD genérico, es posible declarar *ejemplares* suyos. Cada ejemplar corresponde a un cierto reemplazamiento del parámetro formal del TAD genérico por un parámetro actual. Declara dos ejemplares diferentes del TAD genérico del ejercicio anterior, tomando como parámetro actual NAT y ENT-ORD, respectivamente.

156. Enriquece la especificación del TAD genérico CJTO[E :: EQ], añadiendo nuevas operaciones para calcular uniones, intersecciones, diferencias y cardinales de conjuntos, y operaciones de comparación (==), (/=) entre conjuntos. Naturalmente, debes añadir también las ecuaciones necesarias para especificar el comportamiento de las nuevas operaciones. Las ecuaciones de (==) deben construirse de manera que, dados dos términos generados cualesquiera `t,t' : Cjto[Elem]`, se verifiquen las dos condiciones siguientes:

`(i) t = t'` es deducible si y sólo si es deducible `(t == t') = Cierto`.

`(ii) t = t'` no es deducible si y sólo si es deducible `(t == t') = Falso`.

157. Un TAD genérico puede tener más de un parámetro formal. Formula una especificación algebraica de un TAD genérico PAREJA[A, B :: ANY] que ofrezca el tipo *Pareja[A.Elem, B.Elem*], junto con operaciones adecuadas para construir parejas y tener acceso a las dos componentes de cada pareja.

158. Enriquece la especificación del TAD del ejercicio anterior, obteniendo un nuevo TAD PAREJA-ORD[A,B :: ORD] que ofrezca operaciones de igualdad y orden entre las parejas. Observa que ahora es necesario exigir que los parámetros formales A, B representen TADs miembros de la clase de tipos ORD. Cualquier ejemplar de PAREJA-ORD también será miembro de la clase ORD.

TADs con operaciones parciales

159. Especifica algebraicamente un TAD genérico PILA[E :: ANY] que ofrezca el tipo *Pila[Elem]* formado por las pilas de elementos de tipo *Elem* (dado por el TAD parámetro), junto con operaciones que permitan crear una pila vacía, apilar un elemento en una pila, consultar y desapilar el elemento de la cima de una pila no vacía, y reconocer si una pila es vacía o no. Observa que las operaciones *desapilar* y *cima* son *parciales*, por lo cual la especificación del TAD incluye cláusulas que determinan su dominio de definición.

160. Siempre que un TAD tenga operaciones parciales, puede ocurrir que ciertos términos estén indefinidos. Un término t se considera *definido* siempre que **def** t sea deducible a partir de la especificación del TAD, e *indefinido* en caso contrario. Declara el TAD PILA[NAT] de las pilas de números naturales como ejemplar del TAD genérico del ejercicio anterior, y escribe varios ejemplos de términos definidos e indefinidos de tipo *Pila[Nat]*.

161. Para TADs con operaciones parciales, es necesario revisar algunos de los conceptos que hemos estudiado anteriormente. Así:

- La T-equivalencia (ej. 139) se considera entre términos definidos.
- De entre los términos generados (ej. 141) interesan los definidos.
- Para que el conjunto de generadoras sea suficientemente completo (ej. 142), debe ocurrir que cada término cerrado y definido t sea T-equivalente a algún término cerrado generado y definido t'.

– Un sistema de representantes canónicos (ej. 149) debe elegirse como un subconjunto del conjunto de todos los términos cerrados generados y definidos, de manera que cada término cerrado y definido sea T-equivalente a un único representante canónico.

Como aplicación de estas ideas:

(a) Comprueba que las generadoras de PILA[NAT] son suficientemente completas.

(b) Comprueba que las siguientes ecuaciones son equivalencias válidas en PILA[NAT]:

```
(b.1)  cima(desapilar(Apilar(Cero, Apilar(Suc(Cero), PilaVacía))))
       =
       Suc(Cero)

(b.2)  Apilar(Suc(Cero), desapilar(Apilar(Cero,
                                   Apilar(Suc(Cero), PilaVacía))))
       =
       Apilar(Suc(Cero), Apilar(Suc(Cero), PilaVacía))
```

162. Para trabajar con TADs que tengan operaciones parciales, conviene distinguir tres modalidades de igualdad entre términos:

(a) *Igualdad existencial* $=^e$
$t =^e t' \Leftrightarrow_{def}$ t y t' están ambos definidos y valen lo mismo
(nosotros escribiremos simplemente = para la igualdad existencial)

(b) *Igualdad débil* $=^d$
$t =^d t' \Leftrightarrow_{def}$ t y t' valen lo mismo si están los dos definidos
(la igualdad débil siempre se cumple si t, t' o ambos están indefinidos)

(c) *Igualdad fuerte* $=^f$
$t =^f t' \Leftrightarrow_{def}$ o bien t y t' están ambos definidos y valen lo mismo
o bien t y t' están ambos indefinidos

Demuestra que las igualdades débiles y fuertes siempre se pueden especificar usando ecuaciones condicionales con igualdad existencial.

OJO: En cualquier especificación que use ecuaciones condicionales, las ecuaciones de las condiciones deben usar igualdad existencial.

163. Enriquece el TAD de las pilas añadiendo dos nuevas operaciones, especificadas informalmente como sigue:

– *fondo*: Pila[Elem] – → Elem
Observadora. Obtiene el elemento del fondo de una pila no vacía.

– *inversa*: Pila[Elem] → Pila[Elem]
Modificadora. Invierte el orden en el que están apilados los elementos.

Construye ecuaciones adecuadas para especificar algebraicamente las nuevas operaciones. Como ayuda para la especificación de *inversa*, especifica una operación *privada* más general:

– *apilarInversa*: (Pila[Elem], Pila[Elem]) → Pila[Elem]
Modificadora. Toma los elementos de una pila y los apila en orden inverso sobre una segunda pila.

164. Especifica un TAD genérico MCJTO-MIN[E :: ORD] que ofrezca el tipo *MCjto[Elem]* formado por los multiconjuntos de elementos de tipo *Elem* (dado por el TAD parámetro), junto con operaciones similares a las consideradas en el ejercicio 131(b). Observa que el TAD del ejercicio 131(b) se puede especificar como ejemplar del TAD genérico de este ejercicio.

Implementación correcta de TADs

165. Diseña una representación del TAD PILA[NAT] basada en vectores. Define el *tipo representante*, el *invariante de la representación* y la *función de abstracción*.

166. Plantea una implementación de las operaciones del TAD PILA[NAT] usando la representación establecida en el ejercicio anterior. Formula la especificación Pre/Post adecuada para el procedimiento o función que realice cada operación.

167. Diseña tres posibles representaciones para el TAD CJTO[NAT], formulando en cada caso el tipo representante, el invariante de la representación y la función de abstracción.

(a) Usando vectores de naturales.

(b) Usando vectores de naturales sin repeticiones.

(c) Usando vectores de naturales sin repeticiones y ordenados.

168. Plantea implementaciones de la operación *Pon* usando las tres representaciones del ejercicio anterior. Compara.

169. Diseña implementaciones de la operación *quita* usando las tres representaciones del ejercicio anterior. Compara.

†**170.** Plantea dos implementaciones para el TAD POLI del ejercicio 151,

(a) representando los polinomios mediante vectores de parejas, cada pareja formada por un coeficiente y un exponente.

(b) representando los polinomios mediante vectores de parejas como en (a), pero exigiendo que el vector esté ordenado por exponentes.

En cada caso, formula el invariante de la representación, la función de abstracción, y las especificaciones Pre/Post de los procedimientos y funciones que realicen las operaciones de POLI. Intenta comparar la eficiencia de las dos implementaciones.

Implementación modular de TADs

171. Plantea una implementación modular del TAD COMPLEJO del ejercicio 137, en dos fases:

(a) Construcción del *módulo de especificación* (o *interfaz*).

(b) Construcción del *módulo de implementación*.

Observa que el módulo de implementación *oculta* el tipo representante y la implementación de las operaciones, de manera que los *clientes* del módulo no tienen acceso a ellas.

172. Plantea una implementación modular de un ejemplar PILA[ELEM] del TAD genérico PILA[E :: ANY] del ejercicio 159, en dos fases:

(a) Construcción del módulo de especificación, incluyendo una *cláusula de importación* para importar el tipo *elem* de otro módulo ELEM que se supone disponible. La idea es que ELEM implementa un parámetro actual para PILA.

(b) Construcción del módulo de implementación, ocultando el tipo representante, la implementación de las operaciones y los procedimientos de tratamiento de errores.

173. Plantea implementaciones modulares para otros TADs ya estudiados, tales como POLI, ejemplares de CJTO[E :: EQ], y ejemplares de MCJTO-MIN[E :: ORD].

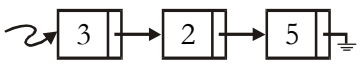
Punteros

174. Representa gráficamente el efecto de ejecutar lo siguiente:

```
var
  p, q : puntero a Ent;

ubicar(p);
p^ := -3; p^:= -2*p^;
ubicar(q);
q^ := -5; q^ := p^ + q^;
p := q;
liberar(p);
liberar(q);
```

¿Daría lo mismo omitir la orden **liberar(p)**? ¿Queda liberado al final todo el espacio que se ha ido solicitando?

175. Escribe un fragmento de programa cuya ejecución construya la estructura correspondiente a la representación gráfica que sigue:



176. Escribe ahora un fragmento de programa que libere el espacio ocupado por la estructura creada en el ejercicio anterior. Ten en cuenta que cada llamada liberará el espacio de un solo registro.

CAPÍTULO 4
TIPOS DE DATOS CON ESTRUCTURA LINEAL

4.1 Pilas

En el tema dedicado a la implementación de TADs ya hemos presentado su especificación así como un par de implementaciones: estática y dinámica. Por eso en este capítulo nos limitaremos a presentar la técnica de análisis de la complejidad de TADs conocida como análisis amortizado, ejemplificada sobre las pilas, y los algoritmos para convertir funciones recursivas lineales no finales en funciones iterativas, con ayuda de una pila.

4.1.1 Análisis de la complejidad de implementaciones de TADs

El análisis de la complejidad de las implementaciones de TADs se basa en los conceptos generales que hemos estudiado en la primera parte de la asignatura: complejidad en el caso peor y en promedio; órdenes de magnitud asintóticos; … Estas mismas ideas se pueden aplicar para analizar por separado la complejidad de los algoritmos que implementan cada una de las operaciones de un TAD.

En este apartado haremos algunas consideraciones sobre la complejidad espacial y nos concentraremos en el problema específico de analizar la complejidad de algoritmos que utilizan TADs, donde interesa obtener la complejidad de secuencias de llamadas a las operaciones del TAD.

Complejidad en espacio

El análisis del espacio tiene especial relevancia en las implementaciones de TADs, ya que éstas emplean muchas veces estructuras de datos muy voluminosas.

Ya hemos visto ejemplos donde distintas elecciones del tipo representante dan lugar a requisitos de espacio diferentes. Sin embargo lo más habitual es que las distintas representaciones correspondan a un coste en espacio del mismo orden de magnitud asintótico. En tales casos, puede interesar analizar con más detalle el coste "real" teniendo en cuenta los valores concretos de las constantes multiplicativas, ya que las optimizaciones solamente se podrán plantear a este nivel de análisis.

Para más detalles puede consultarse [Fra94] pp: 110-111.

Complejidad temporal: análisis de sucesiones de llamadas

Al analizar la complejidad de algoritmos que utilizan TADs interesa ocuparse de sucesiones de llamadas, ya que, en muchos casos, las operaciones modifican el tamaño del valor, de forma que el coste temporal de una llamada se ve afectado por las llamadas que sobre ese mismo valor se han realizado con anterioridad.

Dada una implementación de un TAD con operaciones:

$f_1, f_2, \ldots, f_r$

Sean

$T_1, T_2, \ldots, T_r$

las funciones que miden la complejidad en tiempo, en el caso peor, de los algoritmos que implementan las *r* operaciones.

Queremos estimar el coste de realizar *m* llamadas a operaciones del TAD, siendo:

t_i tiempo consumido por la *i*-ésima llamada
j_i índice de la operación de la *i*-ésima llamada ($1 \le j_i \le r$)
n_{i-1}, n_i tamaño de los datos antes y después de la *i*-ésima llamada
n $\max : 1 \le i \le m : n_i$
m_j número de llamadas a f_j ($1 \le j \le r$)

Entonces son correctas las siguientes estimaciones:

$$\sum_{i=1}^{m} t_i$$

$$\le$$

$$\sum_{i=1}^{m} T_{ji}(n_{i-1})$$

$\le$ (si las T_j son monótonas)

$$\sum_{i=1}^{m} T_{ji}(n)$$

$$= m_1 \cdot T_1(n) + \ldots + m_r \cdot T_r(n)$$

Nótese que este análisis estándar se puede realizar sin conocer los detalles de la implementación, basta con que sean conocidas las funciones $T_j(n)$.

Análisis amortizado de sucesiones de llamadas

Los resultados del análisis estándar que acabamos de realizar siempre dan una cota superior correcta, pero muchas veces ésta es demasiado grande. El análisis estándar es demasiado pesimista, básicamente debido a la simplificación que supone sustituir los diferentes tamaños n_{i-1} por el máximo *n* de todos ellos. Esto podemos observarlo con un ejemplo.

Consideremos una variante del TAD PILA[E :: ELEM] donde la operación *desapilar* se sustituya por

$$\text{desapilarK} : (\text{Nat}, \text{Pila[Elem]}) \rightarrow \text{Pila[Elem]}$$

que es capaz de desapilar *k* elementos de golpe, mediante una llamada *desapilarK(k, p)*.

En una implementación típica de este TAD, la complejidad de las operaciones en el caso peor sería como sigue:

f_j	$T_j(n)$
PilaVacía	O(1)
Apilar	O(1)
desapilarK	O(n)
esVacía	O(1)
cima	O(1)

suponiendo que *n* mida el tamaño de la pila. Nótese que *desapilarK* es *O*(n) porque el caso peor se da cuando se desapilan todos los elementos.

Para una sucesión compuesta por m_1 llamadas a *Apilar* y m_2 llamadas a *desapilarK*, el análisis estándar nos daría como estimación de la complejidad:

$$O(m_1 + m_2 \cdot n)$$

Esta estimación se puede mejorar bastante si suponemos que la pila está inicialmente vacía ($n_0=0$) y tenemos en cuenta que, bajo este supuesto, el número total de elementos desapilados no puede exceder al número total de elementos apilados. Este razonamiento demuestra que

$$O(m_1)$$

es también una cota superior válida –en realidad sería $O(2 \cdot m_1)$–.

Utilizando la técnica del *análisis amortizado* podemos llegar formalmente a este resultado. Esta técnica puede dar cotas superiores más ajustadas que el análisis estándar. La idea consiste en sustituir los *tiempos* por *tiempos amortizados* definidos de la siguiente forma:

$$\text{tiempo amortizado} =_{def} \text{tiempo} + \Delta \text{ potencial}$$

$$a_i = t_i + p_i - p_{i-1}$$

siendo

a_i tiempo amortizado de la *i*-ésima llamada
t_i tiempo consumido por la *i*-ésima llamada
p_i potencial de los datos después de la *i*-ésima llamada
p_{i-1} potencial de los datos antes de la *i*-ésima llamada

El potencial de una estructura de datos *X* se calcula como una función $p(X) \geq 0$ que debe definirse intentando medir la susceptibilidad de la estructura a la ejecución de operaciones costosas; si el potencial es mayor entonces es posible realizar operaciones más costosas. En muchas ocasiones el potencial viene dado por el tamaño de la estructura, en cuyo caso

---

$$p_i = n_i$$

Con esta definición podemos realizar la siguiente estimación para una sucesión de *m* llamadas:

$$\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} (a_i + p_{i-1} - p_i) = (\sum_{i=1}^{m} a_i) + (p_0 - p_m) \leq \text{(suponiendo } p_0 \leq p_m) \sum_{i=1}^{m} a_i$$

Para estimar una cota superior de los tiempos amortizados, definimos el *tiempo amortizado en el caso peor para datos de tamaño n* $A_j(n)$

$$A_j(n) =_{def} \text{Max } \{ a_j(X) \mid \text{tamaño de } X = n \} \quad (1 \leq j \leq r)$$

siendo

$$a_j(X) =_{def} t_j(X) + p(f_j(X)) - p(X) \quad (1 \leq j \leq r)$$

siendo

$$t_j(X) =_{def} \text{tiempo de ejecución de } f_j(X) \quad (1 \leq j \leq r)$$

y donde estamos suponiendo que si $X : \tau$ entonces $f_j : \tau \rightarrow \tau$, en realidad $p(f_j(X))$ denota al potencial de la estructura que representa al valor del tipo principal del TAD, después de realizar la operación f_j.

En muchos casos se verifica que

$$A_j(n) = T_j(n) + \Delta p$$

Una vez definido el tiempo amortizado en el caso peor para datos de tamaño *n* podemos seguir con el análisis del coste de las *m* llamadas

$$\sum_{i=1}^{m} a_i \leq \sum_{i=1}^{m} A_{ji}(n_{i-1}) \leq \text{(si las } A_j \text{ son monótonas)} \sum_{i=1}^{m} A_{ji}(n) = m_1 \cdot A_1(n) + \ldots + m_r \cdot A_r(n)$$

El análisis amortizado dará una estimación del tiempo más ajustada que el análisis estándar si la función potencial se elige adecuadamente. Volviendo al ejemplo de las pilas con operación *desapilarK*, podemos definir el potencial de una pila como su tamaño *n*:

$\Delta p = \Delta n$

f_j	$T_j(n)$	$A_j(n) = T_j(n) + \Delta n$
PilaVacía	$O(1)$	$O(2)$
Apilar	$O(1)$	$O(2)$
desapilarK	$O(n)$	$O(0)$
esVacía	$O(1)$	$O(1)$
cima	$O(1)$	$O(1)$

Nótese que la complejidad amortizada de *desapilarK* es $O(0)$ porque el coste de la operación se compensa con el incremento negativo del tamaño:

$$A_{desapilarK}(n) = Max \{ k + (n-k) - n \mid 0 \leq k \leq n \} = 0$$

Volviendo al ejemplo de las m_1 llamadas a *Apilar* y m_2 llamadas a *desapilarK*, tenemos que el resultado anterior se podrá aplicar siempre que el tamaño final sea mayor o igual que el tamaño inicial –en particular, esto ocurrirá siempre que la pila esté vacía inicialmente–, en cuyo caso:

$$O(m_1 \cdot 2 + m_2 \cdot 0) = O(m_1)$$

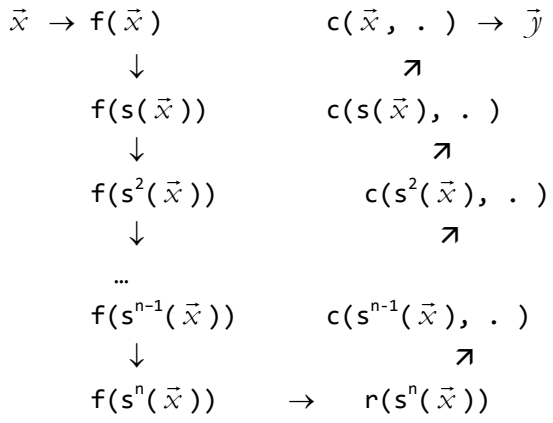
como habíamos deducido razonando informalmente.

4.1.2 Eliminación de la recursión lineal no final

El esquema general de definición de una función recursiva lineal según aparece en el ejercicio 186:

```
func f( x : T ) dev y : S;
{ P₀ : P(x); Cota : t(x) }
var
  x' : T; y' : S;
% Otras posibles declaraciones locales
inicio
  si d(x)
    entonces
    { P(x) ∧ d(x) }
      y ;= r(x)
    { Q(x, y) }
    sino
    { P(x) ∧ ¬d(x) }
      x' := s(x);
    { x' = s(x) ∧ P(x') }
      y' := f(x');
    { x' = s(x) ∧ Q(x', y') }
      y := c(x, y');
    { Q(x, y) }
  fsi
{ Q₀ : Q(x, y) }
  dev y
ffunc
```

Intuitivamente, la ejecución de una llamada recursiva $\vec{y}$:= f($\vec{x}$) consiste en un "bucle descendente" seguido de un "bucle ascendente":



El bucle descendente reitera el cálculo de parámetros para nuevas llamadas recursivas, mientras que el bucle ascendente reitera el procesamiento de los resultados devueltos por las llamadas. La transformación a iterativo se basa en la idea intuitiva que hemos expuesto antes: se utilizan dos bucles compuestos secuencialmente, de forma que en el primero se van obteniendo las descomposiciones recursivas hasta llegar al caso base, y en el segundo se aplica sucesivamente la función de combinación. Vemos que los datos $s^{n-i}(\vec{x})$ –esto es, los parámetros actuales de las diferentes llamadas– deben estar disponibles durante la ejecución del bucle ascendente–. Según la forma cómo se obtengan los valores de $s^{n-i}(\vec{x})$ en el bucle ascendente distinguimos tres casos, que vienen dados por la forma de la función de descomposición recursiva:

— Caso especial. La función s de descomposición recursiva, posee una función inversa calculable s^{-1}. En este caso, los datos $s^{n-i}(\vec{x})$ pueden calcularse sobre la marcha, usando s^{-1}.

— Caso general. La función s no posee inversa –o, aunque la posea, ésta no es calculable, o su cálculo resulta demasiado costoso–. En este caso, los datos $s^{n-i}(\vec{x})$ se irán almacenando en una pila en el curso del bucle descendente y se irán recuperando de ésta en el curso del bucle ascendente.

— Combinación de los casos especial y general. No es necesario apilar toda la información correspondiente a $s^{n-i}(\vec{x})$. Para cada parámetro real $s^{n-i}(\vec{x})$ es suficiente con apilar un dato más simple $\vec{u}_{n-i}$, elegido de tal forma que sea fácil calcular $s^{n-i}(\vec{x})$ a partir de $\vec{u}_{n-i}$ – dato obtenido de la pila– y $s^{n-i+1}(\vec{x})$ –obtenido en la iteración anterior.

Caso especial

El esquema de la transformación es el que se muestra en el ejercicio 186:

```
func f_it( x : T ) dev y : S;
{ P_0 : P(x) }
var
  x' : T;
% Otras posibles declaraciones locales
inicio
  x' := x;
{ Inv. I: R(x', x); Cota: t(x') }
  it ¬d(x') →
  { I ∧ ¬d(x') }
    x' := s(x')
  { I }
  fit;
{ I ∧ d(x') }
  y := r(x');
{ Inv. J: R(x', x) ∧ y = f(x'); Cota: m(x', x) }
  it x' /= x →
  { J ∧ x' ≠ x }
    x' := s^-1(x');
```

```
    y := c(x', y)
  { J }
  fit
{ J ∧ x' = x }
{ y = f(x) }
{ Q₀ : Q(x, y) }
  dev y
ffunc
```

Donde el aserto R(x', x) que aparece en los dos invariantes representa:

$$R(x', x) \Leftrightarrow_{def} \exists\, n : Nat : (x' = s^n(x) \wedge P(x') \wedge \forall\, i : 0 \leq i < n : (\, P(s^i(x)) \wedge \neg d(s^i(x)) \,))$$

que expresa que *x'* desciende de *x* después de un cierto número de llamadas recursivas, es decir, después de un cierto número de aplicaciones de la función *s*, de forma que todos los valores intermedios cumplen la precondición de la función y la barrera del caso recursivo.

La cota del algoritmo descendente es la misma que se haya utilizado para la función recursiva.

y la cota del bucle ascendente, queda

$$m(x', x) =_{def} \min n : Nat : x' = s^n(x)$$

que expresa el número de llamadas recursivas necesarias para alcanzar *x'* desde *x*, cantidad que va disminuyendo pues en cada iteración nos vamos acercando al valor de *x* al ir aplicando s^{-1}.

Como ejemplo de la aplicación de este método veamos cómo podemos transformar la versión recursiva no final de la función factorial:

```
s(n) = n-1    s⁻¹(n) = n+1

func fact ( n : Nat ) dev r : Nat;
{ P₀ : cierto }
var
  n' : Nat;
inicio
  n' := n;
{ I: R(n', n); C : n' }
  it n' /= 0 →
    n' := n' - 1
  fit;
```

```
  r := 0;
  { J : R(n', n) ∧ r = n'!; D : n-n' }
  it n' /= n →
    n' := n'+1;
    r := r * n'
  fit
{ Q₀ : r = n! }
  dev r
ffunc
```

Caso general

Este es el caso donde nos ayudamos de una pila para almacenar los sucesivos valores del parámetro $\vec{x}$. Como se muestra en el ejercicio 187:

```
func f_it( x : T ) dev y : S;
{ P₀ : P(x) }
var
  x' : T;
  xs : Pila[T];
% Otras posibles declaraciones locales
inicio
  x' := x;
  PilaVacía(xs);
{ Inv. I: R(xs, x', x); Cota: t(x') }
  it ¬d(x') →
  { I ∧ ¬d(x') }
    Apilar(x', xs);
    x' := s(x')
  { I }
  fit;
{ I ∧ d(x') }
  y := r(x');
{ Inv. J: R(xs, x', x) ∧ y = f(x'); Cota: tamaño(xs) }
  it NOT esVacía(xs) →
  { J ∧ ¬esVacía(xs) }
    x' := cima(xs);
    y := c(x', y);
    desapilar(xs)
  { J }
  fit
{ J ∧ esVacía(xs) }
{ y = f(x) }
{ Q₀ : Q(x, y) }
```

```
  dev y
ffunc
```

donde, la condición R que aparece en ambos invariantes

$$R(xs, x', x) \Leftrightarrow_{def} \exists\, n : Nat : (x' = s^n(x) \wedge P(x') \wedge \forall\, i : 0 \le i < n : (P(s^i(x)) \wedge \neg d(s^i(x)) \wedge s^i(x) = elem(i, xs)))$$

donde *elem*(i, xs) indica el elemento que ocupa el lugar *i* en la pila *xs*, contando el elemento del fondo como *elem*(0, xs)

Idea: expresa que *x'* desciende de *x* después de un cierto número de llamadas recursivas, cuyos parámetros reales, hasta *x'* exclusive, están apilados en *xs*

La cota del algoritmo ascendente viene dada por el tamaño de la pila, que se va decrementando en 1 con cada iteración

$$tamaño(xs) =_{def} \text{número de elementos apilados en xs}$$

Idea: expresa el número de llamadas recursivas necesarias para alcanzar *x'* desde *x*

Como ejemplo, veamos cómo se aplica este esquema a la función *bin* que obtiene la representación binaria de un número decimal:

```
func bin( n : Nat ) dev r : Nat;
{ P0 : n = N }
inicio
  si
    n < 2 → r := n
  □ n ≥ 2 → r := 10 * bin(n div 2) + (n mod 2)
  fsi
{ Q0 : n = N ∧ r = ∑ i : Nat : ((n div 2^i) mod 2) * 10^i }
  dev r
ffunc
```

La transformación a iterativo:

```
func bin( n : Nat ) dev r : Nat;
{ P0 : n = N }
var
  n' : Nat;
  ns : Pila[Nat];
inicio
  n' := n;
```

```
  PilaVacía(ns);
{ I : R(ns, n', n) ; C : n' }
  it n' ≥ 2 →
    Apilar(n', ns);
    n' := n' div 2    % n' := s(n')
  fit;
  r := n';
{ J : R(ns, n', n) ∧ r = ∑ i : Nat : ((n' div 2^i) mod 2) * 10^i; D :
tamaño(ns) }
  it NOT esVacía(ns) →
    n' := cima(ns);
    r := 10 * r + n' mod 2;   % r := c(n',r)
    desapilar(ns)
  fit
{ Q₀ : n = N ∧ r = ∑ i : Nat : ((n div 2^i) mod 2) * 10^i }
  dev r
ffunc
```

Combinación de los casos especial y general

Como indicamos anteriormente estamos en este caso cuando sólo es necesario apilar una parte de la información que contienen los parámetros $\vec{x}$ de forma que luego sea posible reconstruir dicho parámetro a partir de la información apilada y s($\vec{x}$). Formalmente, hemos de encontrar dos funciones *g* y *h* que verifiquen:

```
¬d(x) ⇒ u = g(x) está definido y cumple que x = h(u, s(x))
```

Modificando el esquema del ejercicio 187 según esta idea, se obtiene la siguiente versión iterativa de *f* según se muestra en el ejercicio 189:

```
func f_it( x : T ) dev y : S;
{ P₀ : P(x) }
var
  x' : T; u : Z;
  us : Pila[Z];
% Otras posibles declaraciones locales
inicio
  x' := x;
  PilaVacía(us);
{ Inv. I: R(us, x', x); Cota: t(x') }
  it ¬d(x') →
  { I ∧ ¬d(x') }
    u := g(x');
```

```
    Apilar(u, us);
    x' := s(x')
  { I }
  fit;
{ I ∧ d(x') }
  y := r(x');
{ Inv. J: R(us, x', x) ∧ y = f(x'); Cota: tamaño(us) }
  it NOT esVacía(us) →
  { J ∧ ¬esVacía(us) }
    u := cima(us);
    x' := h(u,x');
    y := c(x', y);
    desapilar(us)
  { J }
  fit
{ J ∧ esVacía(us) }
{ y = f(x) }
{ Q₀ : Q(x, y) }
  dev y
ffunc
```

Donde el aserto R que aparece en ambos invariantes:

$$R(us, x', x) \Leftrightarrow_{def} \exists\, n : Nat : (x' = s^n(x) \wedge P(x') \wedge \forall\, i : 0 \leq i < n : (P(s^i(x)) \wedge \neg d(s^i(x)) \wedge g(s^i(x)) = elem(i, us)))$$

donde *elem*(i, us) indica el elemento que ocupa el lugar *i* en la pila *us*, contando el elemento del fondo como *elem*(0, us)

Idea: expresa que *x'* desciende de *x* después de un cierto número de llamadas recursivas, y que en *us* está apilada información suficiente para recuperar los parámetros reales de dichas llamadas

Y la cota del bucle ascendente como en el caso anterior viene dada por el número de elementos de la pila

$$tamaño(us) =_{def} \text{número de elementos apilados en us}$$

Idea: expresa el número de llamadas recursivas necesarias para alcanzar *x'* desde *x*

Vamos a aplicar esta técnica al mismo ejemplo del caso anterior: la función *bin*.

En la función *bin* la descomposición recursiva es:

```
s(n) = n div 2;
```

Aquí no hemos podido aplicar la técnica del caso especial porque no existe la inversa de esta función: para obtener *n* a partir de *n div 2* tenemos que saber si *n* es par o impar, por eso hemos tenido que aplicar la transformación del caso general, e ir apilando los sucesivos valores de *n*. Sin embargo, no hace falta apilar *n*, basta con apilar su paridad, pues a partir de *n div 2* y la paridad de *n* podemos obtener *n*. Por lo tanto:

```
g(n) = par(n)

                ⎧ 2 * s(n)      si u
h(u,s(n)) =     ⎨
                ⎩ 2 * s(n) + 1    si ¬u
```

Con lo que la versión iterativa aplicando este nuevo esquema quedará:

```
func bin( n : Nat ) dev r : Nat;
{ P₀ : n = N }
var
  n' : Nat;
  u  : Bool;
  us : Pila[Nat];
inicio
  n' := n;
  PilaVacía(us);
{ I : R(us, n', n) ; C : n' }
  it n' ≥ 2 →
    u := par(n');     % u := g(n')
    Apilar(u, us);
    n' := n' div 2    % n' := s(n')
  fit;
  r := n';
{ J : R(us, n', n) ∧ r = ∑ i : Nat : ((n' div 2^i) mod 2) * 10^i; D :
tamaño(us) }
  it NOT esVacía(us) →
    u := cima(us);
    si
          u → n' := 2 * n'      ⎫
                                ⎬   n' := h(u, n')
    □ NOT u → n' := 2 * n' + 1  ⎭
    fsi
    r := 10 * r + n' mod 2;   % r := c(n',r)
    desapilar(us)
  fit
{ Q₀ : n = N ∧ r = ∑ i : Nat : ((n div 2^i) mod 2) * 10^i }
  dev r
ffunc
```

4.2 Colas

Este TAD representa una colección de datos del mismo tipo donde las operaciones de acceso hacen que su comportamiento se asemeje al de una cola de personas esperando a ser atendidas: los elementos se añaden por el final y se eliminan por el principio, o dicho de otra forma, el primero en llegar es el primero en salir –en ser atendido–: FIFO –*first in first out*–.

4.2.1 Especificación

La especificación según aparece en la página 9 de las hojas de TADs es:

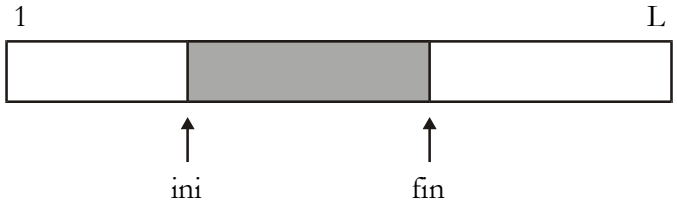
```
tad COLA [E :: ANY]
    usa
      BOOL
    tipo
      Cola[Elem]
    operaciones
      ColaVacía: → Cola[Elem]                                  /* gen */
      Añadir: (Elem, Cola[Elem]) → Cola[Elem]                  /* gen */
      avanzar: Cola[Elem] – → Cola[Elem]                       /* mod */
      primero: Cola[Elem] – → Elem                             /* obs */
      esVacía: Cola[Elem] → Bool                               /* obs */
    ecuaciones
      ∀ x : Elem : ∀ xs : Cola[Elem] :
      def avanzar(Añadir(x, xs))
      avanzar(Añadir(x, xs))      = ColaVacía                 si  esVacía(xs)
      avanzar(Añadir(x, xs))      = Añadir(x, avanzar(xs))    si ¬esVacía(xs)
      def primero(Añadir(x, xs))
      primero(Añadir(x, xs))      = x                       si  esVacía(xs)
      primero(Añadir(x, xs))      = primero(xs)             si ¬esVacía(xs)
      esVacía(ColaVacía)          = Cierto
      esVacía(Añadir(x, xs))      = Falso
    errores
      avanzar(Colavacía)
      primero(ColaVacía)
ftad
```

4.2.2 Implementación

Implementación estática basada en un vector

Tipo representante

Idea gráfica:



Inicialmente *ini* y *fin* están al principio del vector, pero tras un cierto número de invocaciones a *Añadir* y *avanzar* llegaremos a un estado como el indicado en la figura.

El tipo representante:

```
const
  límite = 100;
tipo
  Cola[Elem] = reg
    ini, fin : Nat;
    espacio : Vector [1..límite] de Elem;
  freg;
```

Invariante de la representación

Dada xs : Cola[Elem]

$$
\begin{aligned}
&\texttt{R(xs)}\\
\Leftrightarrow_{\texttt{def}}\ &\texttt{1} \leq \texttt{xs.ini} \leq \texttt{xs.fin + 1} \leq \texttt{límite + 1} \wedge\\
&\forall\ \texttt{i} : \texttt{xs.ini} \leq \texttt{i} \leq \texttt{xs.fin} : \texttt{R(xs.espacio(i))}
\end{aligned}
$$

Nótese que el aserto de dominio implica: $1 \leq \text{xs.ini} \leq \text{límite}+1 \wedge 0 \leq \text{xs.fin} \leq \text{límite}$

Nótese también que estamos permitiendo xs.ini = xs.fin+1, con lo que representamos a una cola vacía, como veremos a continuación al formalizar la función de abstracción.

Función de abstracción

Dada xs : Cola[Elem] tal que R(xs):

```
A(xs) =def hazCola(xs.espacio, xs.ini, xs.fin)

hazCola(e, i, f) = ColaVacía                                  si i = f+1
hazCola(e, i, f) = Añadir( A(e(f)), hazCola( e, i, f–1) )     si i ≤ f
```

Implementación procedimental de las operaciones

Debido a los problemas, que ya comentamos en el caso de las pilas, que conlleva la implementación funcional, pasamos directamente a la implementación con procedimientos.

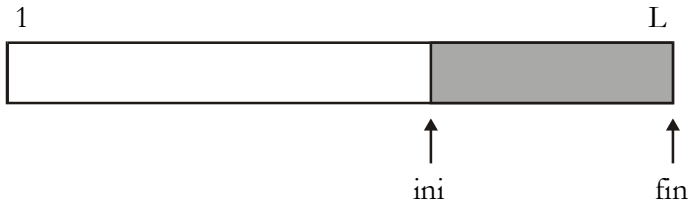
```
proc ColaVacía (s xs : Cola[Elem] );    % O(1)
{ P0 : Cierto }
inicio
  xs.ini := 1;
  xs.fin := 0;
{ Q0 : R(xs) ∧ A(xs) =COLA[ELEM] ColaVacía }
fproc

proc Añadir ( e x : Elem; es xs : Cola[Elem] );   % O(1)
{ P0 : xs = XS ∧ R(x) ∧ R(xs) ∧ xs.fin < límite }
inicio
  si xs.fin == límite
    entonces
      error("Cola llena")
    sino
      xs.fin := xs.fin+1;
      xs.espacio(xs.fin) := x;     % compartición de estructura
  fsi
{ Q0 : R(xs) ∧ A(xs) =COLA[ELEM] Añadir(A(x), A(XS)) }
fproc

proc avanzar ( es xs : Cola[Elem] );    % O(1)
{ P0 : xs = XS ∧ R(xs) ∧ NOT esVacía(A(xs)) % xs.ini ≤ xs.fin }
inicio
  si esVacía(xs)
    entonces
      error("Cola vacía")
```

```
    sino
      % ELEM.anular(xs.espacio(xs.ini))
      xs.ini := xs.ini+1
  fsi
{ Q₀ : R(xs) ∧ A(xs) =COLA[ELEM] avanzar(A(XS)) }
fproc

func primero ( xs : Cola[Elem] ) dev x : Elem;    % O(1)
{ P₀ : R(xs) ∧ NOT esVacía(A(xs)) }
inicio
  si esVacía(xs)
    entonces
      error("Cola vacía")
    sino
      x := xs.espacio(xs.ini)
  fsi
{ Q₀ : A(x) =COLA[ELEM] primero(A(xs)) }
  dev x
ffunc

func esVacía ( xs : Cola[Elem] ) dev r : Bool ;   % O(1)
{ P₀ : R(xs) }
inicio
  r := xs.ini == xs.fin+1
{ Q₀ : A(r) =COLA[ELEM] esVacía(A(xs)) }
  dev r
ffunc
```

Desventaja de esta implementación

El espacio ocupado por la representación de la cola se va desplazando hacia la derecha al ejecutar *Añadir* y *avanzar*.

Cuando *fin* alcanzar el valor *límite* no se pueden añadir nuevos elementos, aunque quede sitio en *espacio[1..ini–1]*.



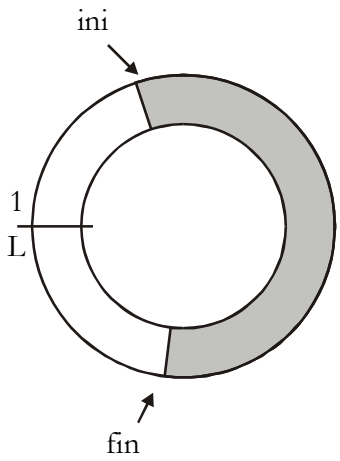
Una solución sería que cuando llegásemos a esta situación se desplazasen todos los elementos de la pila hacia la izquierda haciendo *ini=1*. Sin embargo esto penalizaría en esos casos la eficiencia de la operación *Añadir* –*O*(n)–. Es mejor la solución descrita a continuación.

Implementación basada en un vector circular

Esta implementación está pensada para resolver el problema de la anterior; cuando *fin* alcanza el valor *límite* y queda espacio libre entre 1 y *ini–1* se utiliza ese espacio para seguir añadiendo elementos. Hay que tener cuidado porque ahora puede suceder que *ini* > *fin*, y en particular *ini=fin+1* puede implicar que la cola esté llena o vacía.

Tipo representante

Idea gráfica:



Para que se vea mejor, se podría mostrar un ejemplo de cómo evolucionaría la cola ante un cierto número de invocaciones de *Añadir* y *avanzar*.

El tipo representante:

```
const
  límite = 100;
tipo
  Cola[Elem] = reg
    ini, fin, tam : Nat;
    espacio : Vector [1..límite] de Elem;
  freg;
```

Usamos el campo *tam* para guardar el número de elementos almacenados en cada instante. Este campo es necesario para distinguir en la situación *ini=fin+1* si tenemos una cola vacía o llena. en [Fran93] se describen otras formas de resolver este problema.

Para escribir el invariante de la representación, la función de abstracción y las operaciones vamos a utilizar dos funciones auxiliares *pred* y *suc* que restan y suman 1 "módulo límite". En la implementación de las operaciones podríamos utilizar directamente la operación **mod**, aunque para ello sería mejor idea definir el vector entre 0 y límite–1. Sin embargo, en la función de abstracción nos hace falta una operación de restar 1 que cumpla límite–1 = 1, para lo cual no disponemos de ninguna función.

Por lo tanto definimos las siguientes funciones auxiliares:

$$suc, pred: [1..límite] \to [1..límite]$$

$$suc(i) =_{def} \begin{cases} i+1 & \textbf{si } i < limite \\ 1 & \textbf{si } i = límite \end{cases}$$

$$pred(i) =_{def} \begin{cases} i-1 & \textbf{si } i > 1 \\ límite & \textbf{si } i = 1 \end{cases}$$

Necesitamos definir también la aplicación repetida de *suc*:

$$\text{dados } i \in [1..límite],\ j \geq 0 \text{ definimos}$$

$$suc^j(i) =_{def} \begin{cases} i & \textbf{si } j = 0 \\ suc^{j-1}(suc(i)) & \textbf{si } j > 0 \end{cases}$$

Convenimos por último:

$$suc^{-1}(i) =_{def} pred(i) \quad \% \text{ es necesario para escribir el invariante R}$$

Invariante de la representación

Dado xs : Cola[Elem]

$$\begin{aligned} & R(xs) \\ \Leftrightarrow_{def} \\ & 0 \leq xs.tam \leq límite \ \wedge \\ & 1 \leq xs.ini \leq límite \ \wedge\ xs.fin = suc^{xs.tam-1}(xs.ini) \ \wedge \\ & \forall\ i : 0 \leq i \leq xs.tam-1 : R(xs.espacio(suc^i(xs.ini))) \end{aligned}$$

Función de abstracción

Dado xs: Cola[elem] tal que R(xs):

$$A(xs) =_{def} hazCola(xs.esp, xs.tam, xs.fin)$$

$$hazCola(e, t, f) =_{def} ColaVacía \quad \textbf{si } t = 0$$

```
hazCola(e, t, f) =def Añadir( A(e(f)), hazCola(e, t–1, pred(f)) ) si t > 0
```

Implementación procedimental de las operaciones

En la implementación de las operaciones suponemos disponible la función *suc* con el comportamiento antes descrito.

```
proc ColaVacía (s xs : Cola[Elem] );    % O(1)
{ P0 : Cierto }
inicio
  xs.ini := 1;
  xs.fin := límite;
  xs.tam := 0;
{ Q0 : R(xs) ∧ A(xs) =COLA[ELEM] ColaVacía }
fproc
```

Nótese que con esta inicialización se cumple la parte del invariante de la representación:

$$xs.fin = suc^{xs.tam-1}(xs.ini)$$

para lo cual hay que aplicar el convenio

$$suc^{-1}(i) = pred(i)$$

```
proc Añadir ( e x : Elem; es xs : Cola[Elem] );   % O(1)
{ P0 : xs = XS ∧ R(x) ∧ R(xs) ∧ xs.tam < límite }
inicio
  si xs.tam == límite
    entonces
      error("Cola llena")
    sino
      xs.tam := xs.tam+1;
      xs.fin := suc(xs.fin);
      xs.espacio(xs.fin) := x;     % compartición de estructura
  fsi
{ Q0 : R(xs) ∧ A(xs) =COLA[ELEM] Añadir(A(x), A(XS)) }
fproc

proc avanzar ( es xs : Cola[Elem] );    % O(1)
{ P0 : xs = XS ∧ R(xs) ∧ NOT esVacía(A(xs)) % xs.tam > 0 }
inicio
  si esVacía(xs)
```

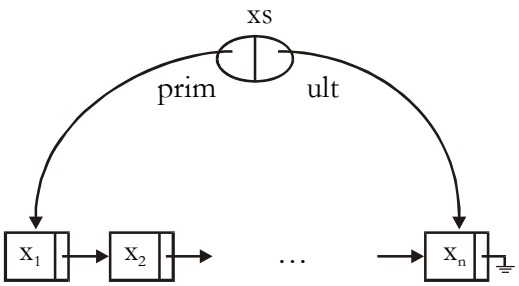
```
    entonces
      error("Cola vacía")
    sino
      % ELEM.anular(xs.espacio(xs.ini))
      xs.tam := xs.tam - 1;
      xs.ini := suc(xs.ini)
  fsi
{ Q₀ : R(xs) ∧ A(xs) =COLA[ELEM] avanzar(A(XS)) }
fproc

func primero ( xs : Cola[Elem] ) dev x : Elem;    % O(1)
{ P₀ : R(xs) ∧ NOT esVacía(A(xs)) }
inicio
  si esVacía(xs)
    entonces
      error("Cola vacía")
    sino
      x := xs.espacio(xs.ini)
  fsi
{ Q₀ : A(x) =COLA[ELEM] primero(A(xs)) }
  dev x
ffunc

func esVacía ( xs : Cola[Elem] ) dev r : Bool ;   % O(1)
{ P₀ : R(xs) }
inicio
  r := xs.tam == 0
{ Q₀ : A(r) =COLA[ELEM] esVacía(A(xs)) }
  dev r
ffunc
```

Implementación dinámica

La idea es tener acceso directo al principio y al final de la cola para así poder realizar las operaciones de manera eficiente.

Tipo representante

Idea gráfica:



Existe también la posibilidad de implementar el nodo cabecera con memoria dinámica, con esto conseguiríamos que en aquellos lenguajes donde las funciones no pueden devolver tipos estructurados, devolvieran valores de tipo cola –y en Modula-2 estaríamos obligados si quisiéramos que el tipo fuese oculto–.

El tipo representante:

```
tipo
  Enlace = puntero a Nodo;
  Nodo = reg
    elem : Elem;
    sig : Enlace
  freg;
  Cola[Elem] = reg
    prim, ult : Enlace
  freg;
```

Invariante de la representación

Dada xs : Cola[Elem]

$$R(xs) \Leftrightarrow_{def} R_{CV}(xs) \vee R_{CNV}(xs)$$

escribimos por separado las condiciones que debe cumplir y una cola vacía y otra que no lo esté.

$$R_{CV}(xs)$$
$$\Leftrightarrow_{def}$$
$$xs.prim = nil \wedge xs.ult = nil$$

$$R_{CNV}(xs)$$
$$\Leftrightarrow_{def}$$
$$xs.prim \neq nil \wedge xs.ult \neq nil \wedge$$
$$buenaCola(xs.prim, xs.ult)$$

entre otras cosas, en el predicado *buenaCola* tenemos que expresar la condición de que desde *p* es posible llegar a *q*

```
        buenaCola(p,q)
⇔def
        (p = q ∧ ubicado(p) ∧ R(p^.elem) ∧ p^.sig = nil) ∨
        (p ≠ q ∧ ubicado(p) ∧ R(p^.elem) ∧
         p ∉ cadena(p^.sig) ∧ buenaCola(p^.sig, q))

cadena(p) =def ∅                          si p = nil
cadena(p) =def {p} ∪ cadena(p^.sig)       si p ≠ nil
```

Función de abstracción

Dada xs : Cola[Elem] tal que R(xs):

```
A(xs) =def ColaVacía                  si RCV(xs)
A(xs) =def hazCola(xs.prim, xs.ult)   si RCNV(xs)
```

para construir la cola tenemos que añadir elementos hasta que lleguemos a una cola con un solo elemento, donde se cumple p=q

```
hazCola(p,q) =def Añadir(A(p^.elem), ColaVacía)     si p = q
hazCola(p,q) =def ponerPrimero( A(p^.elem), hazCola(p^.sig, q)) si p ≠ q

ponerPrimero( y, ColaVacía ) =def Añadir( y, colaVacía)
ponerPrimero( y, Añadir( x, xs ) ) =def Añadir( x, PonerPrimero( y, xs ) )
```

es necesario el predicado auxiliar *ponerPrimero* porque el orden de recorrido del valor representante de la cola es el inverso al orden de inserción en la misma.

Implementación procedimental de las operaciones

```
proc ColaVacía (s xs : Cola[Elem] );    % O(1)
{ P0 : Cierto }
inicio
  xs.prim := nil;
  xs.ult := nil;
{ Q0 : R(xs) ∧ A(xs) =COLA[ELEM] ColaVacía }
fproc
```

```
proc Añadir ( e x : Elem; es xs : Cola[Elem] );   % O(1)
{ P₀ : xs = XS ∧ R(x) ∧ R(xs) ∧ xs.tam < límite }
var
  p : Enlace;
inicio
  ubicar(p);
  p^.elem := x;    % compartición de estructura
  p^.sig := nil;
  si esVacía(xs)
    entonces
      xs.prim := p;
      xs.ult := p
    sino
      xs.ult^.sig := p;
      xs.ult := p
  fsi
{ Q₀ : R(xs) ∧ A(xs) =COLA[ELEM] Añadir(A(x), A(XS)) }
fproc

proc avanzar ( es xs : Cola[Elem] );    % O(1)
{ P₀ : xs = XS ∧ R(xs) ∧ NOT esVacía(A(xs)) % xs.tam > 0 }
var
  p : Enlace;
inicio
  si esVacía(xs)
    entonces
      error("Cola vacía")
    sino
      p := xs.prim;
      xs.prim := xs.prim^.sig;
      si xs.prim == nil
        entonces
          xs.ult := nil    % sólo tenía un elemento
        sino
          seguir
      fsi;
      % ELEM.anular(p^.elem)
      liberar(p)
  fsi
{ Q₀ : R(xs) ∧ A(xs) =COLA[ELEM] avanzar(A(XS)) }
fproc

func primero ( xs : Cola[Elem] ) dev x : Elem;    % O(1)
{ P₀ : R(xs) ∧ NOT esVacía(A(xs)) }
inicio
  si esVacía(xs)
```

```
    entonces
      error("Cola vacía")
    sino
      x := xs.prim^.elem   % compartición de estructura
  fsi
{ Q₀ : A(x) =COLA[ELEM] primero(A(xs)) }
  dev x
ffunc

func esVacía ( xs : Cola[Elem] ) dev r : Bool ;   % O(1)
{ P₀ : R(xs) }
inicio
  r := xs.prim == nil
{ Q₀ : A(r) =COLA[ELEM] esVacía(A(xs)) }
  dev r
ffunc
```

Como cualquier implementación de un TAD, debemos incluir una operación que realice copias de los valores representados:

```
func copiar ( xs : Cola[Elem] ) dev ys : Cola[Elem];
{ P₀ : R(xs) }
var
  p, q, r : Enlace; % p recorre la original y q,r la copia
inicio
  si esVacía(xs)
    entonces
      ys.prim := nil;
      ys.ult := nil;
    sino
      p := xs.prim;
      ubicar(q);
      ys.prim := q;
      q^.elem := p^.elem;  % q^.elem := ELEM.copiar(p^.elem)
      { I : p ∈ cadena(xs.prim) ∧
            los punteros de cadena(ys.prim) apuntan a copias de los nodos de
            cadena(xs.prim) hasta p^ inclusive ∧
            q = último(cadena(ys.prim));
        C : |cadena(p)|
}
      it p^.sig /= nil →
        p := p^.sig;
        ubicar(r);
        q^.sig := r;
        q := r;
```

```
        q^.elem := p^.elem;% q^.elem := ELEM.copiar(p^.elem)
      fit;
      q^.sig := nil;
      ys.ult := q
  fsi
{ Q₀ : R(ys) ∧ A(xs) =COLA[ELEM] A(ys) ∧
      xs, ys apuntan a estructuras que no comparten nodos }
  dev ys
ffunc
```

4.3 Colas dobles

Son una generalización de las colas y las pilas, donde es posible insertar, consultar y eliminar tanto por el principio como por el final.

4.3.1 Especificación

La especificación según aparece en la página 10 de las hojas con los TADs:

```
tad DCOLA [E :: ANY]
    usa
      BOOL
    tipo
      DCola[Elem]
    operaciones
      DColaVacía: → DCola[Elem]                                  /* gen */
      PonDetrás: (Elem, DCola[Elem]) → DCola[Elem]                    /*
gen */
      ponDelante: (Elem, DCola[Elem]) → DCola[Elem]              /* mod */
      quitaUlt: DCola[Elem] – → DCola[Elem]                      /* mod */
      último: DCola[Elem] – → Elem                               /* obs */
      quitaPrim: DCola[Elem] – → DCola[Elem]                          /*
mod */
      primero: DCola[Elem] – → Elem                              /* obs */
      esVacía: DCola[Elem] → Bool /* obs */
    ecuaciones
      ∀ x, y : Elem : ∀ xs : DCola[Elem] :
      ponDelante(y, DColaVacía)        = PonDetrás(y, DColaVacía)
      ponDelante( y, PonDetrás(x, xs)) = PonDetrás(x, ponDelante(y, xs))
      def quitaUlt(xs) si ¬esVacía(xs)
      quitaUlt(PonDetrás(x,xs))        = xs

      def último(xs) si ¬esVacía(xs)
```

```
        último(PonDetrás(x, xs))          = x
        def quitaPrim(xs) si ¬esVacía(xs)
        quitaPrim(PonDetrás(x, xs))       = DColaVacía      si   esVacía(xs)
        quitaPrim(PonDetrás(x, xs))       = PonDetrás(x, quitaPrim(xs))
                                                          si ¬esVacía(xs)
        def primero(xs) si ¬esVacía(xs)
        primero(PonDetrás(x, xs))         = x           si  esVacía(xs)
        primero(PonDetrás(x, xs))         = primero(xs) si ¬esVacía(xs)
        esVacía(DColaVacía)               = Cierto
        esVacía(PonDetrás(x, xs))         = Falso
    errores
        quitaUlt(DColavacía)
        último(DColaVacía)
        quitaPrim(DColavacía)
        primero(DColaVacía)
ftad
```

Se podría elegir como generador *PonDetrás* o *ponDelante*. Hemos elegido *PonDetrás* para que la especificación sea más parecida a la de las colas normales, ya que *PonDetrás* ≡ *Añadir*.

Se puede establecer de manera evidente la siguiente correspondencia entre las operaciones de las colas y las de las colas dobles:

COLA[ELEM]	DCOLA[ELEM]
ColaVacía	DColaVacía
Añadir	PonDetrás
avanzar	quitaPrim
primero	primero
esVacía	esVacía
	ponDelante
	quitaUlt
	último

4.3.2 Implementación

Básicamente podemos considerar las mismas implementaciones utilizadas para las colas, a excepción de la primera que, como ya vimos, no es muy conveniente.

Implementación basada en un vector circular

Tipo representante

Igual que en el caso de las colas ordinarias, dándole le nombre *DCola[Elem]*.

Invariante de la representación

Igual que el caso de las colas ordinarias.

Función de abstracción

Es igual que el caso de las colas ordinarias, sustituyendo *ColaVacía* por *DColaVacía* y *Añadir* por *PonDetrás*:

Dado xs: Cola[elem] tal que R(xs):

```
A(xs) =def hazDCola(xs.esp, xs.tam, xs.fin)


hazDCola(e, t, f) =def DColaVacía   si t = 0
hazDCola(e, t, f) =def PonDetrás( A(e(f)), hazDCola(e, t-1, pred(f)) ) si t >
0
```

Implementación procedimental de las operaciones

Todas las operaciones consumen tiempo $O(1)$ en el caso peor. Aquellas operaciones que tienen una análoga en COLA[ELEM] se implementan igual; el resto de manera análoga.

```
proc ponDelante ( e x : Elem; es xs : DCola[Elem] );   % O(1)
{ P0 : xs = XS ∧ R(x) ∧ R(xs) ∧ xs.tam < límite }
inicio
  si xs.tam == límite
    entonces
      error("Cola llena")
    sino
      xs.tam := xs.tam+1;
      xs.ini := pred(xs.ini);
      xs.espacio(xs.ini) := x;     % compartición de estructura
  fsi
{ Q0 : R(xs) ∧ A(xs) =DCOLA[ELEM] ponDelante(A(x), A(XS)) }
fproc
proc quitaUlt ( es xs : Cola[Elem] );   % O(1)
{ P0 : xs = XS ∧ R(xs) ∧ NOT esVacía(A(xs)) % xs.tam > 0 }
inicio
  si esVacía(xs)
    entonces
      error("Cola vacía")
    sino
```

```
      % ELEM.anular(xs.espacio(xs.fin))
      xs.tam := xs.tam – 1;
      xs.fin := pred(xs.fin)
  fsi
{ Q₀ : R(xs) ∧ A(xs) =DCOLA[ELEM] quitaUlt(A(XS)) }
fproc
```

Se queda como ejercicio la implementación de *último*.

Implementación dinámica

Tipo representante, invariante de la representación y función de abstracción

Es igual que el caso de las colas ordinarias, sustituyendo *ColaVacía* por *DColaVacía* y *Añadir* por *PonDetrás*.

Implementación procedimental de las operaciones

Aquellas operaciones que tienen una análoga en COLA[ELEM] se implementan igual, con un tiempo de ejecución en el caso pero de $O(1)$.

Se puede proponer como ejercicio la implementación de:

```
proc ponDelante (e x : Elem; es xs : DCola[Elem] );  % O(1)
{ P₀ : x = XS ∧ R(x) ∧ R(xs) }
% algoritmo análogo a Añadir en COLA[ELEM]
{ Q₀ : R(x) ∧ A(x) =DCOLA[ELEM] ponDelante(A(x), A(XS)) }
fproc

proc último ( xs: DCola[Elem] ) dev x : Elem;   % O(1)
{ P₀ : R(xs) ∧ NOT esVacía(A(xs)) }
% algoritmo análogo a primero en COLA[ELEM]
{ Q₀ : R(x) ∧ A(x) =DCOLA[ELEM] último( A(xs) ) }
fproc
```

Más interesante resulta la implementación de *quitaUlt* que necesita recorrer la cola con lo que resulta en un complejidad de $O(n)$. La razón es que el puntero al penúltimo nodo sólo se puede obtener recorriendo la cola:

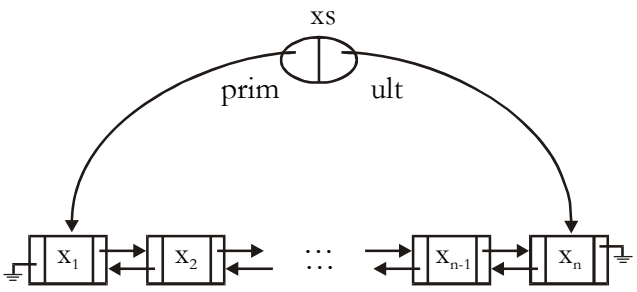
```
proc quitaUlt ( es xs : Cola[Elem] );   % O(n)
{ P₀ : xs = XS ∧ R(xs) ∧ NOT esVacía(A(xs)) }
var
  p : Enlace;
inicio
  si esVacía(xs)
    entonces
      error("Cola vacía")
```

```
    sino
      p := xs.prim;
      si p == xs.ult
        entonces
          xs.prim := nil;
          xs.ult := nil;
          ELEM.anular(p^.elem);
          liberar(p)
        sino
          it p^.sig /= xs.ult →
            p := p^.sig
          fit;
          p^.sig := nil;
          ELEM.anular(xs.ult^.elem);
          liberar(xs.ult);
          xs.ult := p
      fsi
  fsi
{ Q₀ : R(xs) ∧ A(xs) =DCOLA[ELEM] quitaUlt(A(XS)) }
fproc
```

Para conseguir que también esta operación tenga complejidad $O(1)$ se puede utilizar la siguiente representación.

Implementación dinámica con encadenamiento doble

Tipo representante



Con esta representación tenemos acceso en tiempo constante al penúltimo elemento. El tipo representante queda:

```
tipo
  Enlace = puntero a Nodo;
  Nodo = reg
    elem : Elem;
    sig, ant : Enlace
  freg;
```

```
DCola[Elem] = reg
  prim, ult : Enlace
freg;
```

Esta representación permite una implementación procedimental de todas las operaciones con complejidad de $O(1)$.

4.4 Listas

Es un TAD que representa a una colección de elementos de un mismo tipo que generaliza a pilas y colas porque incluye una operación que permite acceder al elemento *i*-ésimo. Incluye además operaciones de concatenación, conteo de los elementos, …

4.4.1 Especificación

Como se recoge en la página 11 de las especificaciones de TADs

```
tad LISTA[E :: ANY]
    usa
      BOOL, NAT
    tipo
      Lista[Elem]
    operaciones
```

A la derecha aparecen notaciones habituales, que usaremos indistintamente con el nombre de las operaciones

```
      Nula: → Lista[Elem]                              /* gen */   % []
      Cons: (Elem, Lista[Elem]) → Lista[Elem]          /* gen */   %
[x/xs]
      [ _ ]: Elem → Lista[Elem]                        /* mod */   % [x]
      ponDr: (Lista[Elem], Elem) → Lista[Elem]         /* mod */   %
[xs\x]
      primero: Lista[Elem] – → Elem                    /* obs */
```

El resultado de quitarle el primer elemento a la lista

```
      resto: Lista[Elem] – → Lista[Elem]               /* mod */
      último: Lista[Elem] – → Elem                     /* obs */
```

El resultado de quitarle el último elemento a la lista

```
      inicio: Lista[Elem] – → Lista[Elem]              /* mod */
      ( ++ ): (Lista[Elem], Lista[Elem]) → Lista[Elem] /* mod */
      nula?: Lista[Elem] → Bool                        /* obs */
      (# ): Lista[Elem] → Nat                          /* obs */
      ( !! ): (Lista[Elem], Nat) – → Elem              /* obs */
```

Las generadoras son libres

```
    ecuaciones
      ∀ x, y: E.Elem : ∀ xs, ys : Lista[Elem] : ∀ n : Nat :
      [ x ]                          = Cons(x, Nula)
      Nula ++ ys                     = ys
      Cons(x, xs) ++ ys              = Cons(x, (xs ++ ys))
      ponDr(xs, x)                   = xs ++ [x]
      nula?(Nula)                    = Cierto
      nula?(Cons(x, xs))             = Falso
      def primero(xs) si NOT nula?(xs)
      primero(Cons(x, xs))           = x
      def resto(xs)   si NOT nula?(xs)
      resto(Cons(x, xs))             = xs
      def último(xs)  si NOT nula?(xs)
      último(Cons(x, xs))            = x                 si
nula?(xs)
      último(Cons(x, xs))            = último(xs)        si NOT
nula?(xs)
      def inicio(xs)  si NOT nula?(xs)
      inicio(Cons(x, xs))            = Nula              si
nula?(xs)
      inicio(Cons(x, xs))            = Cons(x, inicio(xs)) si NOT
nula?(xs)
      # Nula                         = Cero
      # Cons(x, xs)                  = Suc(# xs)
```

Por primera aparece la igualdad fuerte en las especificaciones. Recordemos que la igualda fuerte representa el hecho de que ambos términos están definidos y son equivalentes, o que ambos términos están indefinidos. El problema con esta operación es que no podemos escribir una ecuación que especifique su comportamiento y que esté garantizado que sólo involucra términos definidos; es por ello que utilizamos la igualdad fuerte.

```
      def xs !! si Suc(Cero) ≤ n ≤ (# xs)
      Cons(x, xs) !! Suc(Cero)       = x
      Cons(x, Cons(y, ys)) !! Suc(Suc(n)) =f Cons(y,ys) !! Suc(n)
    errores
      primero(Nula)
      resto(Nula)
      último(Nula)
      inicio(Nula)
      xs !! n si (n == Cero) OR (n > (# xs))
ftad
```

Notaciones habituales para representar listas (siendo x_i : Elem)

```
      [ x1, … , xn ]
      [ x1, … , xn / xs ]
      [ xs \ x1, … , xn ]
```

Muchas de las operaciones de las listas tienen una análoga en DCOLA[ELEM], y es por ello que las representaciones son similares. Se verifican las siguientes equivalencias entre las operaciones:

Operaciones de LISTA	Operaciones de DCOLA
Nula	DColaVacía
Cons	ponDelante
[_]	—
ponDr	PonDetrás
primero	primero
resto	quitaPrim
último	último
inicio	quitaUlt
(++)	—
nula?	esVacía
(#)	—
(!!)	—

4.4.2 Implementación

Normalmente se utiliza siempre la implementación dinámica.

Representación estática

Podría basarse en un vector o un vector circular como hemos visto para los TAD COLA[ELEM] y DCOLA[ELEM]. Esta representación permitiría una implementación procedimental con tiempo $O(1)$ para todas las operaciones de LISTA que tienen una análoga en DCOLA. También sería fácil implementar:

— [_] como **proc** en tiempo $O(1)$

— (#) como **func** en tiempo $O(1)$

— (!!) como **func** en tiempo $O(1)$

Sin embargo se plantean inconvenientes con esta representación:

— El uso de la operación (++) tiende a desbordar fácilmente los límites de espacio de una representación estática.

— En muchas aplicaciones de las listas es más natural usar (++) y otras operaciones como funciones –en algoritmos recursivos, sobre todo si se viene de la *cultura funcional*–. Una implementación funcional con representación estática malgasta muchos espacio (¿por qué?)

En cualquier caso es muy sencillo llevar a cabo esta implementación.

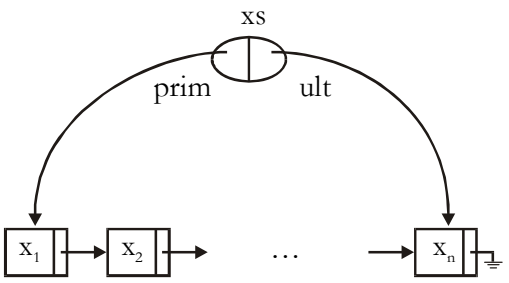
Representación dinámica como la utilizada para las pilas

La idea gráfica:

El problema es que esta representación sólo es adecuada para implementar eficientemente las operaciones que acceden a una lista por su extremo inicial. en particular, se pueden implementar en tiempo $O(1)$ las operaciones: *Nula, Cons, primero, resto, nula?*. Sin embargo, no permite una implementación eficiente de las operaciones que accede a una lista por el final.

Representación dinámica como la utilizada para las colas

La idea gráfica de esta representación



Tipo representante

```
tipo
  Enlace = puntero a Nodo;
  Nodo = reg
    elem : Elem;
    sig : Enlace
  freg;
  Lista[Elem] = reg
    prim, ult : Enlace
  freg;
```

Invariante de la representación

Es exactamente igual que el invariante de la representación que vimos para las colas. Si la lista está vacía entonces *prim* y *ult* valen *nil*. Si no está vacía, entonces ambos son distintos de *nil*, y se debe cumplir que desde el nodo apuntado por *prim* se llegue al nodo apuntado por *ult*, que será el último de la cadena, donde no se introduce circularidad y donde todos los punteros están ubicados y los elementos son representantes válidos.

Función de abstracción

Es muy similar a la de las colas, aunque más simple porque ahora la generadora que añade elementos, lo hace por el principio, con lo que no resulta necesaria introducir la función auxiliar *ponerPrimero*.

Dada xs : Lista[Elem] tal que R(xs):

```
A(xs) =def Nula                          si RLV(xs)
A(xs) =def hazLista(xs.prim, xs.ult)     si RLNV(xs)
```

para construir la lista tenemos que añadir elementos hasta que lleguemos a una lista con un solo elemento, donde se cumple p=q

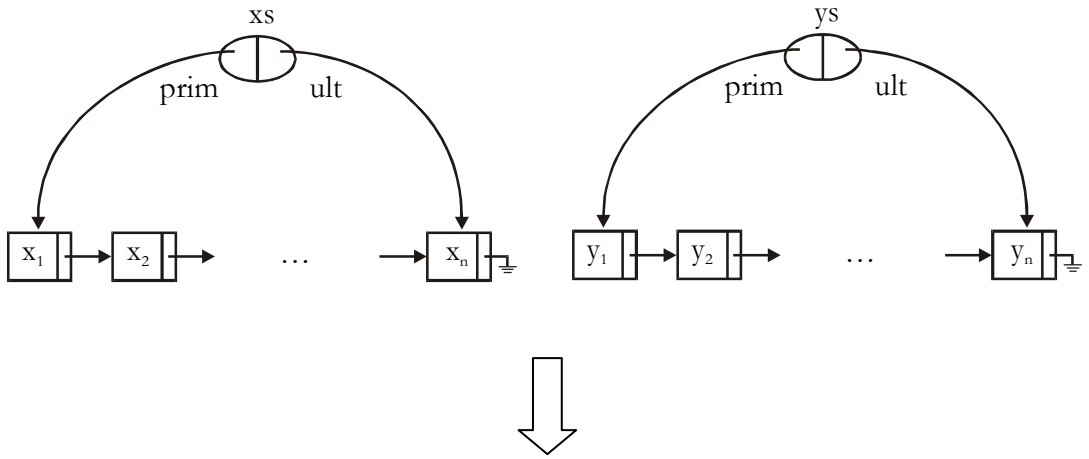
```
hazLista(p,q) =def Cons(A(p^.elem), Nula)   si p = q
hazLista(p,q) =def Cons( A(p^.elem), hazLista(p^.sig, q))   si p ≠ q
```

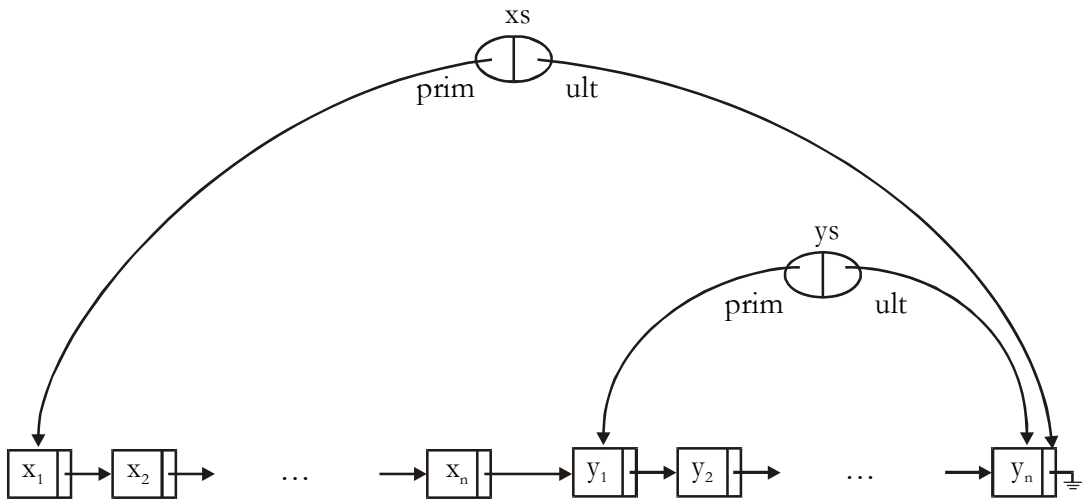
Implementación procedimental de las operaciones

Todas las operaciones de LISTA que poseen una análoga en DCOLA se pueden implementar con el mismo algoritmo ya estudiado en DCOLA. El tiempo de ejecución es $O(1)$ en todos los casos, excepto para *inicio* (análoga a *quitaUlt*) que necesita tiempo $O(n)$ –este tiempo podría descender a $O(1)$ usando una representación doblemente enlazada–.

Excepto la operación (++) el resto de operaciones que no tienen análoga en DCOLA las implementaremos como funciones, y es por eso que nos ocuparemos de ellas en el siguiente apartado.

Para (++) se puede plantear una implementación procedimental que modifica su primer argumento y consume tiempo $O(1)$:






```
proc conc( es xs : Lista[Elem]; e ys : Lista[Elem] )
{ P₀ : R(xs) ∧ R(ys) ∧ xs = XS ∧ ys = YS }
inicio
  si nula?(xs)
    entonces
      xs := ys

    sino
      si nula?(ys)
        entonces
          seguir
        sino
          xs.ult^.sig := ys.prim;
          xs.ult := ys.ult
      fsi
  fsi
{ Q₀ : R(xs) ∧ R(ys) ∧ A(xs) =LISTA[ELEM] A(XS) ++ A(ys) }
fproc
```

Esta implementación da lugar a compartición de estructura, por lo que:

— o las operaciones que eliminan elementos de una lista no los anulan

— o la lista usada como parámetro de entrada en la llamada no se vuelve a utilizar después de ésta.

En cuanto a las operaciones *copiar* y *anular* también las implementamos como procedimientos, pudiendo optar, como en los demás TADs, por realizar de forma *simple* o *profunda*. La operación de copia sería exactamente igual a la que ya implementamos para las colas. En cuanto a la anulación, es muy similar a la que implementamos para las pilas:

```
proc anula( es Lista : Lista[Elem] );
{ P₀ : R(xs) ∧ xs = XS }
```

```
var
  p, q : Enlace;
inicio
  si nula?(xs)
    entonces
      seguir
    sino
      p := xs.prim;
      it p /= nil →
        q := p^.sig;
        ELEM.anular(p^.elem);   % sólo si se hace anulación profunda
        liberar(p);
        p := q
      fit;
      xs.prim := nil;
      xs.ult := nil
  fsi
{ Q₀ : R(xs) ∧ A(xs) =LISTA[ELEM] Nula ∧ los nodos accesibles desde XS
       a través de punteros de tipo Enlace, se han liberado       }
fproc
```

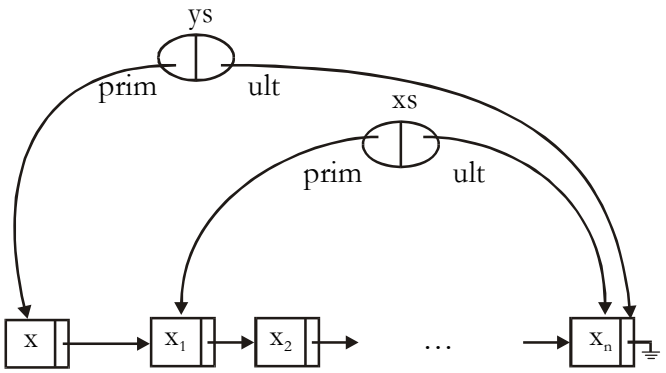
Implementación funcional de las operaciones

Las operaciones observadoras de LISTA que tienen una análoga en DCOLA se pueden implementar con el mismo algoritmo ya estudiado en DCOLA, con tiempo de ejecución $O(1)$.

A continuación, estudiamos las operaciones generadoras y modificadoras, así como las observadoras sin análoga en DCOLA. Vamos a permitir la compartir de estructura.

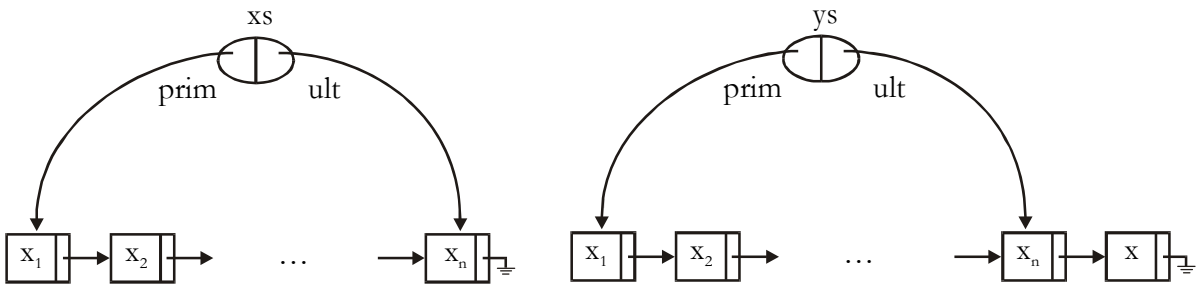
```
func Nula () dev xs : Lista[Elem];      /* O(1) */
{ P₀ : cierto }
inicio
  xs.prim := nil;
  xs.ult := nil
{ Q₀ : R(xs) ∧ A(xs) =LISTA[ELEM] Nula }
  dev xs
ffunc

func Cons ( x : Elem; xs : Lista[Elem] ) dev ys : Lista[Elem]; /* O(1)
{ P₀ : R(x) ∧ R(xs) }
var
  nuevo : Enlace;
inicio
  ubicar(nuevo);
  ys.prim := nuevo;
  nuevo^.elem := x; % compartición de estructura
```

```
  si nula?(xs)
    entonces
      ys.ult := nuevo;
      nuevo^.sig := nil
    sino
      ys.ult := xs.ult;
      nuevo^.sig := xs.prim
  fsi
{ Q₀ : R(ys) ∧ A(ys) = LISTA[ELEM] Cons(A(x), A(xs)) }
  dev ys
ffunc
```

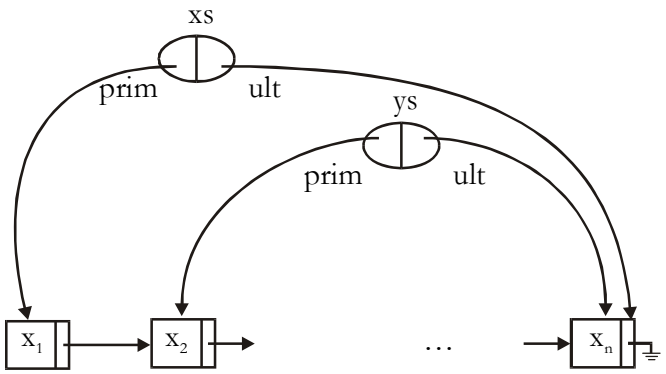



```
func ponDr( xs : Lista[elem]; x : Elem ) dev ys : Lista[Elem] /* O(n) por
copia */
{ P₀ : R(x) ∧ R(xs) }
var
  nuevo : Enlace;
inicio
  ys := copia(xs);
  ubicar(nuevo);
  nuevo^.elem := x; % compartición de estructura
  nuevo^.sig := nil;
  si nula?(ys)
    entonces
      ys.prim := nuevo
    sino
      ys.ult^.sig := nuevo
  fsi;
  ys.ult := nuevo;
{ Q₀ : R(ys) ∧ A(ys) =LISTA[ELEM] ponDr( A(xs), A(x) ) }
  dev ys
ffunc
```

Nótese que en este caso es necesaria la copia para que se mantenga el invariante de la representación de *xs* –el valor *nil* del campo *sig* del último nodo–.



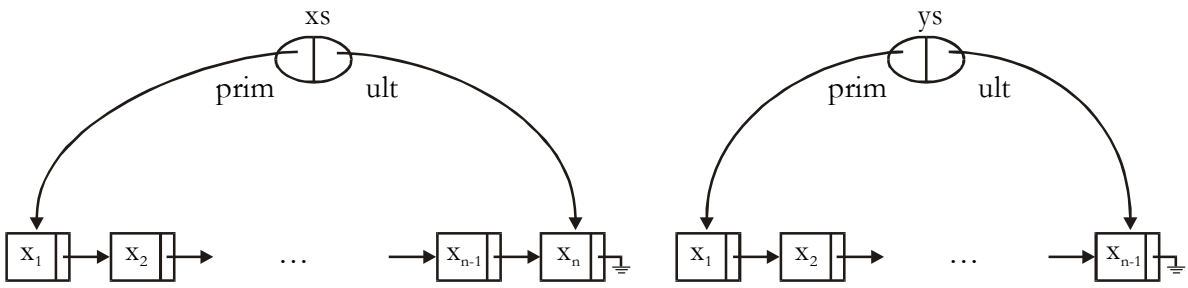

```
func resto ( xs : Lista[Elem] ) dev ys : Lista[Elem]  /* O(1) */
{ P₀ : R(xs) ∧ NOT nula?(A(xs)) }
inicio
  si nula?(xs)
    entonces
      error("Lista vacía")
    sino
      si xs.prim == xs.ult
        entonces
          ys.prim := nil;
          ys.ult := nil


        sino
          ys.prim := xs.prim^.sig;
          ys.ult := xs.ult
      fsi
  fsi
{ Q₀ : R(ys) ∧ A(ys) =LISTA[ELEM] resto( A(xs) ) }
  dev ys
ffunc
```



El siguiente es el primer ejemplo de operación que sólo implementamos parcialmente; los detalles de la función privada auxiliar *copiaInicio* se quedan como ejercicio.

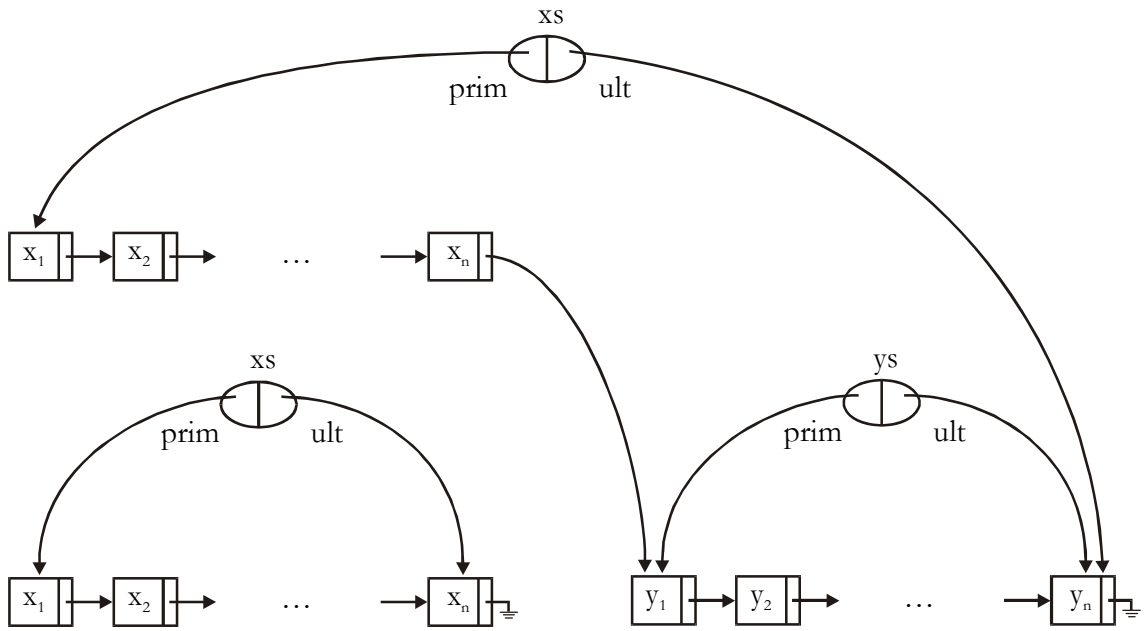
```
func inicio (xs : Lista[Elem] ) dev ys : Lista[Elem]  /* O(n)  por copia */
{ P0 : R(xs) ∧ NOT nula?(A(xs)) }
inicio
  si nula?(xs)
    entonces
      error("Lista vacía")
    sino
      ys := copiaInicio( xs )   % función privada O(n)
  fsi
{ Q0 : R/ys) ∧ A(ys) =LISTA[ELEM] inicio( A(xs) ) }
  dev ys
ffunc
```

En este caso la copia es necesaria para preservar la representación correcta de A(xs).




```
func conc ( xs, ys : Lista[Elem] ) dev zs : Lista[Elem];
% tiempo O(n) debido a la copia de xs, siendo n = # A(xs)
{ P0 : R(xs) ∧ R(ys) }
inicio
  si nula?(xs)
    entonces
      zs := ys
    sino
      zs := copia(xs);
      xs.ult^.sig := ys.prim;
      zs.ult := ys.ult
  fsi
{ Q0 : R(zs) ∧ A(zs) =LISTA[ELEM] A(xs) ++ A(ys) }
  dev zs
ffunc
```

La copia es necesaria para preservar la representación correcta de A(xs).




```
func longitud ( xs : Lista[Elem] ) dev n : Nat;   /* O(n)  */
{ P₀ : R(xs) }
var
  aux : Enlace;
inicio
  n := 0;
  aux := xs.prim;
  it aux /= nil →
    n := n+1;
    aux := aux^.sig
  fit
{ Q₀ : n = LISTA[ELEM] #( A(xs) ) }
  dev n
ffunc
```

El tiempo de ejecución de *longitud* se puede reducir a $O(1)$ modificando el tipo representante Lista[Elem]. La idea es añadir a la cabecera un tercer campo que almacene la longitud.

```
func consultar( xs : Lista[Elem]; i : Nat ) dev x : Elem;   /* O(i) */
{ P₀ : R(xs) ∧ 1 ≤ i ≤ #( A(xs) ) }
var
  j : Nat;
  aux : Enlace;
inicio
  j := i;
  aux := xs.prim;
  it j > 1 and aux /= nil →
    j := j-1;
```

```
    aux := aux^.sig
  fit;
  si j == 1 and aux /= nil
    entonces
      x := aux^.elem    % compartición de estructura
    sino
      error("Posición inexistente")
  fsi
{ Q₀ : A(x) =LISTA[ELEM] A(xs) !! i }
  dev x
ffunc
```

Como conclusión del apartado de implementación mostramos una tabla con la complejidad de las distintas operaciones:

Operación	Procedimiento	Función
Nula	O(1)	O(1)
Cons	O(1)	O(1) †
[_]	O(1)	O(1)
ponDr	O(1)	O(n) ©
primero	—	O(1)
resto	O(1)	O(1) †
último	—	O(1)
inicio	O(n)	O(n) ©
(++)	O(1) †	O(n) † ©
nula?	—	O(1)
(#)	—	O(n)
(!!)	—	O(i)

4.4.3 Programación recursiva con listas

Como se indica en los ejercicios, en los ejemplos que siguen:

— Suponemos dada una implementación funcional del TAD LISTA. Usamos las operaciones públicas, sin programar con punteros.

— Ignoramos los problemas de gestión de memoria: estructura compartida, copia, anula, …

Programación recursiva de #, !! y ++ usando funciones más básicas

```
func longitud ( xs : Lista[Elem] ) dev n : Nat;
{ P₀ : R(xs) }
inicio
  si nula?(xs)
```

```
    entonces
      n := 0
    sino
      n := 1 + longitud( resto(xs) )
  fsi
{ Q₀ : n = LISTA[ELEM] #( A(xs) ) }
  dev n
ffunc
```

Complejidad: disminución por sustracción de 1, O(n), siendo n = # xs

```
func consultar( xs : Lista[Elem]; i : Nat ) dev x : Elem;   /* O(i) */
{ P₀ : R(xs) ∧ 1 ≤ i ≤ #( A(xs) ) }
inicio
  si
    nula?(xs) OR i == 0      → Error("elemento inexistente")
  □ NOT nula?(xs) AND i == 1 → x := primero(xs)     % compartición de
estructura
  □ NOT nula?(xs) AND i > 1  → x := consultar( resto(xs), i-1 )
  fsi
{ Q₀ : A(x) =LISTA[ELEM] A(xs) !! i }
  dev x
ffunc
```

Complejidad: disminución por sustracción de 1, O(i)

```
func conc ( xs, ys : Lista[Elem] ) dev zs : Lista[Elem];
{ P₀ : R(xs) ∧ R(ys) }
inicio
  si nula?(xs)
    entonces
      zs := ys
    sino
      zs := Cons( primero(xs), conc( resto(xs), ys ) )
  fsi
{ Q₀ : R(zs) ∧ A(zs) =LISTA[ELEM] A(xs) ++ A(ys) }
  dev zs
ffunc
```

Complejidad: disminución por sustracción de 1, O(n), siendo n = # xs

Las llamadas a *Cons* van creando nuevos nodos para los elementos de *xs*. Al final, *zs* comparte estructura con *ys* –por la asignación del caso base–, y con los elementos de *xs*. El efecto es el mismo que se producía en la implementación funcional de (++) con punteros, presentada más arriba.

Funciones *coge* y *tira*

La función *coge* devuelve una lista con los *n* primeros elementos de *xs*. *Tira* devuelve la lista que resulta de quitar los primeros *n* elementos de *xs*. Consideramos el caso de que *n* sea mayor que el número de elementos de *xs*.

```
func coge( n : Nat; xs : Lista[Elem] ) dev us : Lista[Elem];
{ P₀ : cierto }
  si
    n == 0                  → us := Nula
  □ n > 0 AND nula?(xs)     → us := Nula
  □ n > 0 AND NOT nula?(xs) → us := Cons( primero(xs), coge( n-1, resto(xs)
) )
  fsi
{ Q₀ : #us = min(n, #xs) ∧ ∀ i : 1 ≤ i ≤ #us : us !! i = xs !! i }
  dev us
ffunc
```

Cuando escribimos funciones que usan un TAD ya no aparecen los representantes válidos ni las funciones de abstracción; usamos las operaciones del TAD como predicados que pueden aparecer en nuestros asertos.

```
func tira( n : Nat; xs : Lista[Elem] ) dev vs : Lista[Elem];
{ P₀ : cierto }
  si
    n == 0                  → vs := xs
  □ n > 0 AND nula?(xs)     → vs := Nula
  □ n > 0 AND NOT nula?(xs) → vs := tira( n-1, resto(xs) )
  fsi
{ Q₀ : #vs = max(0, #xs-n) ∧ ∀ i : 1 ≤ i ≤ #vs : vs !! i = xs !! (n+i) }
  dev us
ffunc
```

En ambos casos tenemos disminución del tamaño del problema por sustracción de 1, con una única llamada recursiva. La combinación de los resultados consume tiempo constante. Por lo tanto la complejidad es $O(n)$. Podríamos pensar que esa *n* es el *min* del parámetro *n* y #*xs*, sin embargo para un valor de *n* y *xs* dados el caso peor se da cuando $\#xs \geq n$.

Inversa de una lista

La idea naive para implementar la operación de inversión es una función recursiva de la forma:

```
(CD) inv([]) = []
(CR) inv([x/xs]) = inv(xs) ++ [x]
```

Que es la idea implementada en el enunciado del ejercicio 212:

```
func inv( xs : Lista[Elem] ) dev ys : Lista[Elem];
{ P₀ : cierto }
inicio
  si nula?(xs)
    entonces ys := Nula
    sino     ys := inv(resto(xs)) ++ [primero(xs)]
  fsi
{ Q₀ : ∀ : 1 ≤ i ≤ (# xs) : xs !! i = ys !! (# xs)-i+1 }
ffunc
```

Donde se han utilizado las operaciones de las listas como operadores, porque en el enunciado de los ejercicios todavía no habíamos escrito sus implementaciones y no les habíamos dado nombre. En realidad, habría que reescribirla utilizando la notación habitual –algunos lenguajes, como C++, permiten definir funciones como operadores–.

Esta implementación tiene complejidad $O(n^2)$, siendo $n = \#xs$, porque la concatenación implementada funcionalmente tiene complejidad $O(n)$ debido a la copia del primer argumento:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \le n < b \\ a \cdot T(n-b) + c \cdot n^k & \text{si } n \ge b \end{cases}$$

donde $b = 1, a = 1, k = 1$

$$T(n) \in \begin{cases} O(n^{k+1}) & \text{si } a = 1 \\ O(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

por lo tanto $T(n) \in O(n^2)$

Podemos utilizar la técnica de plegado-desplegado, para obtener un algoritmo recursivo final para la función más general *invSobre*, especificada como sigue:

```
func invSobre( as, xs : Lista[Elem] ) dev ys : Lista[Elem];
{ P₀ : cierto }
{ Q₀ : ys = inv(xs) ++ as }
ffunc
```

de forma que este algoritmo tenga complejidad $O(n)$

Efectivamente *invSobre* es una generalización de *inv*

```
invSobre(as, xs) = inv(xs) ++ as
invSobre([], xs) = inv(xs) ++ [] = inv(xs)
```

Aplicando la técnica de plegado-desplegado

```
(CD) invSobre( as, [] ) = inv([]) ++ as = as
(CR) invSobre(as, [x/xs]) = inv([x/xs]) ++ as
                          = inv(xs) ++ [x] ++ as
                          = inv(xs) ++ [x/as]
                          = invSobre( [x/as], xs )
```

Cuya implementación queda:

```
func invSobre( as, xs : Lista[Elem] ) dev ys : Lista[Elem];
{ P₀ : cierto }
  si nula?(xs)
    entonces
      ys := as
    sino
      ys := invSobre( Cons( primero(xs), as ), resto(xs) )
  fsi
{ Q₀ : ys = inv(xs) ++ as }
  dev ys
ffunc
```

Donde tenemos ahora que la preparación de la llamada es $O(1)$, y, por lo tanto, la complejidad total es $O(n)$

Mezcla de listas ordenadas

La solución directa a este problema según está especificado en el ejercicio 215:

```
func mezcla( xs, ys : Lista[Elem] ) dev zs : Lista[Elem]
{ P₀ : ordenada(xs) ∧ ordenada(ys) }
{ Q₀ : ordenada(zs) ∧ permutación( zs, xs++ys ) }
```

Sería

```
(CD)     mezcla( [],     ys ) = ys
       mezcla( [x/xs], [] ) = [x/xs]
(CR)     mezcla( [x/xs], [y/ys] ) = [x / mezcla(xs, [y/ys])]  si x ≤ y
       mezcla( [x/xs], [y/ys] ) = [y / mezcla([x/xs], ys)]  si x > y
```

La complejidad es $O(n)$, siendo $n = \#xs + \#ys$

Se puede convertir en una función recursiva final encontrando una generalización con un parámetro acumulador, y obteniendo el código de esta generalización mediante plegado-desplegado:

```
mezcla'(as, xs, ys) = as ++ mezcla(xs, ys)
```

Efectivamente es una generalización:

```
mezcla'([], xs, ys) = mezcla(xs, ys)
```

Aplicando plegado-desplegado resulta

```
(CD)
                        mezcla'(as, [], ys)
                    =   as ++ mezcla( [], ys )
                    =   as ++ ys

                        mezcla'(as, [x/xs], [])
                    =   as ++ mezcla( [x/xs], [] )
                    =   as ++ [x/xs]

(CR)
                        mezcla'(as, [x/xs], [y/ys])
                    =   as ++ mezcla( [x/xs], [y/ys] )
              x ≤ y =   as ++ [x / mezcla(xs, [y/ys])]
                    =   as ++ [x] ++ mezcla(xs, [y/ys])
                    =   mezcla'( ponDr(as,x), xs, [y/ys] )

                        mezcla'(as, [x/xs], [y/ys])
              x > y =   mezcla'( ponDr(as, y), [x/xs], ys )
```

Esta implementación tendrá complejidad $O(n)$, con $n = \#xs + \#ys$, siempre y cuando la concatenación –de los casos base– y *ponDr* –de la descomposición recursiva– tengan complejidad $O(1)$, lo cual sólo se verifica si utilizamos una implementación procedimental de las operaciones.

El predicado *permutación* que aparece en la postcondición se puede especificar de la siguiente manera:

```
      permutación(us, vs)
⇔def
      ∀ x : Elem : (#i : 1 ≤ i ≤ #us : us!!i = x) =
                   (#i : 1 ≤ i ≤ #vs : vs!!i = x)
```

4.5 Secuencias

Son colecciones de elementos con una posición distinguida dentro de esa colección. Es el TAD que utilizamos para poder hacer recorridos sobre una colección de elementos. Insistir aquí en la idea de que los TAD no son "objetos monolíticos" y que si, por ejemplo, me interesase una colección de elementos con acceso por posición y que además se pudiese recorrer, podría escribir la especificación de un TAD que mezclase incluyese operaciones de LISTA y SECUENCIA. Nótese también que si quisiese recorrer una *Lista[Elem]*, tendría que hacerlo con la operación (!!), lo que llevaría a un recorrido de complejidad cuadrática; mientras que con las operaciones de las secuencias podemos conseguir complejidad lineal.

4.5.1 Especificación

La especificación se basa en la idea de que podemos representar las secuencias como dos listas, siendo el punto de interés el punto divisorio entre las dos partes. Nótese que esto es ligeramente distinto de la primera idea que hemos presentado, donde el punto de interés es un elemento de la secuencia, que, en este planteamiento es el primer elemento de la parte derecha de la secuencia.

Aparecen en esta especificación dos elementos nuevos relacionados: el uso privado de un TAD y la definición de una generadora privada. El uso privado indica que los clientes de este TAD no tiene que conocer el uso que éste hace de LISTA[ELEM]. La generadora privada se introduce para llevar a cabo la idea de representar las secuencias como dos listas; entonces se expresa el comportamiento de las generadoras públicas, que son libres, en términos de la generadora privada. Para expresar comportamiento de las operaciones no generadoras no se utilizan las generadoras públicas sino la privada.

```
tad SEC[E :: ANY]
    usa
      BOOL
    usa privadamente
      LISTA[E]
    tipo
      Sec[Elem]
    operaciones
      Crea: → Sec[Elem]                                    /* gen */
```

Generadora que inserta un elemento a la izquierda del punto de interés –como el último de la parte izquierda–

```
      Inserta: (Sec[Elem], Elem) → Sec[Elem]               /* gen */
```

Modificadora que elimina el elemento a la derecha del punto de interés –el primero de la parte derecha–. Es parcial porque la parte derecha puede estar vacía; esta es también la razón que explica la parcialidad de las dos siguientes operaciones

```
      borra: Sec[Elem] - → Sec[Elem]                       /* mod */
```

Devuelve el elemento situado a la derecha del punto de interés

```
      actual: Sec[Elem] - → Elem                           /* obs */
```

Desplaza un lugar a la derecha el punto de interés. Es una generadora porque dos secuencias con el mismo contenido pero distinto punto de interés son secuencias diferentes

```
      Avanza: Sec[Elem] - → Sec[Elem]                      /* gen */
```

Generadora que posiciona el punto de interés a la izquierda del todo.

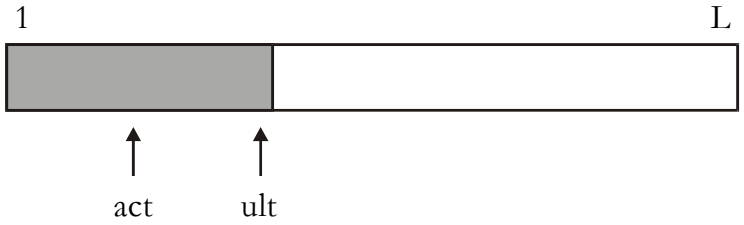
```
      Reinicia: Sec[Elem] → Sec[Elem]                      /* gen */

      vacía?: Sec[Elem] → Bool                             /* obs */
```

```
    fin?: Sec[Elem] → Bool                                    /* obs */
  operaciones privadas
    S : (Lista[Elem], Lista[Elem]) → Sec[Elem]                /* gen */
    piz, pdr, cont: Sec[Elem] → Lista[Elem]                   /* obs */

  ecuaciones
    ∀ s : Sec[Elem] : ∀ iz, dr : Lista[Elem] : ∀ x : Elem :
    Crea                        = S(Nula, Nula)
    Inserta(S(iz, dr), x)       = S(iz ++ [x], dr)
    fin?(S(iz, dr))             = nula?(dr)
    def borra(s)  si NOT fin?(s)
    borra(S(iz, Cons(x, dr)))   = S(iz, dr)
    def actual(s) si NOT fin?(s)
    actual(S(iz, Cons(x, dr)))  = x
    def Avanza(s) si NOT fin?(s)
    Avanza(S(iz, Cons(x, dr)))  = S(iz ++ [x], dr)
    Reinicia(S(iz, dr))         = S(Nula, iz ++ dr)
    vacía?(S(iz, dr))           = nula?(iz) AND nula?(dr)
    piz(S(iz, dr))              = iz
    pdr(S(iz, dr))              = dr
    cont(S(iz, dr))             = iz ++ dr
  errores
    borra(s)  si fin?(s)
    actual(s) si fin?(s)
    avanza(s) si fin?(s)
ftad
```

4.5.2 Implementación

Implementación estática

Basada en un vector, siguiendo una idea similar a la estudiada para la representación de las pilas. La idea gráfica:



Nótese que *act* apunta al primero de la parte derecha. Cuando el punto de interés esté a la derecha del todo, su valor será *ult+1*

Tipo representante

```
  const
```

```
  l = 100;
tipo
  Sec[Elem] = reg
    act, ult : Nat;
    esp : Vector [1..l] de Elem;
  freg;
```

Invariante de la representación

Dada xs : Sec[Elem]:

$$
\begin{array}{l}
\quad \texttt{R(xs)} \\
\Leftrightarrow_{def} \\
\quad \texttt{0} \leq \texttt{x.ult} \leq \texttt{l} \wedge \texttt{1} \leq \texttt{xs.act} \leq \texttt{xs.ult + 1} \wedge \\
\quad \forall\ \texttt{i} : \texttt{1} \leq \texttt{i} \leq \texttt{xs.ult} : \texttt{R(xs.esp(i))}
\end{array}
$$

Nótese que:

— ult = 0 ⇒ act = 1

En este caso cont(A(xs)) = [ ]

— ult = l ⇒ 1 ≤ act ≤ l+1

— act = ult + 1 ⇒ pdr(A(xs)) = [ ]

Función de abstracción

Dada xs : Sec[Elem] tal que R(xs)

$$\texttt{A(xs)} =_{def} \texttt{S(hazLista(x.esp, 1, xs.act-1), hazLista(xs.esp, xs.act, xs.ult))}$$

$$\texttt{hazLista(e, c, f)} =_{def} \begin{cases} \texttt{[]} & \textbf{si } \texttt{c = f+1} \\ \texttt{[A(e(c)) / hazLista(e, c+1, f)]} & \textbf{si } \texttt{c} \leq \texttt{f} \end{cases}$$

Nótese que R(xs) garantiza que las dos llamadas a lista cumplen c ≤ f+1, es decir, que se hacen con segmentos de xs.esp de longitud ≥ 0

Implementación procedimental de las operaciones

Con esta representación, la eficiencia que se puede conseguir para las operaciones con una implementación procedimental:

Operación	Complejidad
Crea	O(1)
Inserta	O(n)

borra	O(n)
actual	O(1)
Avanza	O(1)
Reinicia	O(1)
vacía?	O(1)
fin?	O(1)

La razón de la complejidad lineal de *Inserta* y *borra* es que suponen desplazamientos. Consideramos que esta implementación es ineficiente, ya que nuestro objetivo es conseguir que todas las operaciones tengan complejidad *O*(1).

Implementación basada en una pareja de listas

Seguimos directamente la idea de la especificación del TAD

Tipo representante

```
tipo
  Sec[Elem] = reg
    piz : Lista[Elem];
    pdr : Lista[Elem]
  freg;
```

Donde el tipo *Lista[Elem]* se importa del módulo LISTA[ELEM].

Invariante de la representación

Es trivial. Dada xs : Sec[Elem]

$$R(xs) \Leftrightarrow_{def} R(xs.piz) \wedge R(xs.pdr)$$

que está garantizado automáticamente por la correcta implementación de LISTA[ELEM].

Función de abstracción

Dado xs : Sec[Elem]

$$A(xs) =_{def} S(A(xs.piz), A(xs.pdr))$$

Implementación procedimental de las operaciones

Esta implementación haría uso de los procedimientos exportados por una implementación procedimental de LISTA[ELEM]. Se podrían obtener los siguientes tiempos de ejecución:

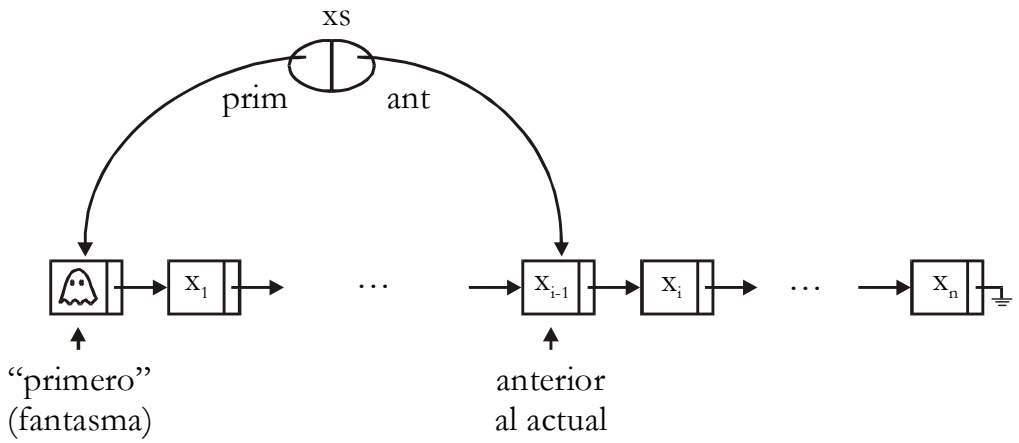
Operación	Complejidad	Idea
Crea	proc O(1)	
Inserta	proc O(1)	*ponDr* en piz

borra	proc O(1)	*resto* en pdr
actual	func O(1)	*primero* en pdr
Avanza	proc O(1)	*primero* y *resto* en pdr; *ponDr* en piz
Reinicia	proc $O(n^2)$	reiterar; *último* e *inicio* –O(n)– en piz; *Cons* en pdr
vacía?	func O(1)	*nula?* en piz y pdr
fin?	func O(1)	*nula?* en pdr

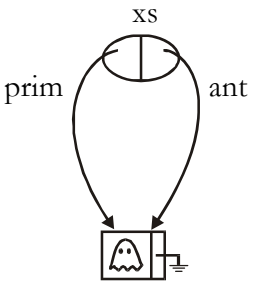
Reinicia resulta demasiado ineficiente.

Implementación dinámica eficiente

La idea gráfica de esta representación:



Elegimos apuntar al anterior al actual –al último de la parte izquierda– porque así podemos realizar todas las operaciones en tiempo $O(1)$: insertar delante del actual, eliminar el actual, consultar el actual, …



Aparece aquí una idea nueva que es el uso de un "nodo fantasma". La razón de utilizar este nodo es para simplificar los algoritmos. Como *ant* apunta al anterior al actual, ¿adónde apuntará en una secuencia vacía? Con el fantasma, nos evitamos el tratamiento de los casos en los que la secuencia está vacía.

Tipo representante

```
tipo
  Nodo = reg
```

```
    elem : Elem;
    sig : Enlace
  freg;
  Enlace = puntero a Nodo;
  Sec[Elem] = reg
    pri, ant : Enlace
  freg;
```

Invariante de la representación

Dada xs : Sec[Elem]:

$$
\begin{aligned}
& \texttt{R(xs)} \\
\Leftrightarrow_{\texttt{def}} \; & \\
& \texttt{xs.pri} \neq \texttt{nil} \wedge \texttt{ubicado(xs.pri)} \wedge \\
& \texttt{xs.ant} \neq \texttt{nil} \wedge \texttt{ubicado(xs.ant)} \wedge \\
& \texttt{xs.ant} \in \texttt{cadena(xs.pri)} \wedge \\
& \texttt{xs.pri} \notin \texttt{cadena(xs.pri\^{}.sig)} \wedge \\
& \texttt{cadenaCorrecta(xs.pri\^{}.sig)}
\end{aligned}
$$

Donde, como siempre, *cadena(p)* da el conjunto de enlace que parten de *p* hasta llegar a nil, exclusive.

$$
\texttt{cadena(p)} =_{\texttt{def}} \begin{cases} \emptyset & \textbf{si } \texttt{p} = \texttt{nil} \\ \{\texttt{p}\} \cup \texttt{cadena(p\^{}.sig)} & \textbf{si } \texttt{p} \neq \texttt{nil} \end{cases}
$$

Y donde, dado *p* : *Enlace*, *cadenaCorrecta(p)* se cumple SYSS todos los punteros de *cadena(p)* apuntan a nodos diferentes correctamente construidos:

$$
\begin{aligned}
& \texttt{cadenaCorrecta(p)} \\
\Leftrightarrow_{\texttt{def}} \; & \\
& \texttt{p} = \texttt{nil} \vee \\
& (\ \texttt{p} \neq \texttt{nil} \wedge \texttt{ubicado(p)} \wedge \texttt{R(p\^{}.elem)} \wedge \\
& \texttt{p} \notin \texttt{cadena(p\^{}.sig)} \wedge \texttt{cadenaCorrecta(p\^{}.sig)}\ )
\end{aligned}
$$

Función de abstracción

Dada xs : Sec[Elem] tal que R(xs):

```
A(xs) =def S( hazPiz( xs.pri^.sig, xs.ant^.sig ), hazPdr( xs.ant^.sig ) )
```

Parece más intuitivo invocar a *hazPiz* con *xs.ant* en lugar de *xs.ant^.sig*; sin embargo, esto complicaría la definición de *hazPiz*, porque habría que incluir un caso base más: *p=q=nil*; y lo que es peor, habría que distinguir si *p = q = xs.prim*. Es decir, facilita la definición de la función en los casos especiales de: piz vacía y secuencia vacía.

hazPiz(p, q) construye una lista abstracta con los elementos localizados en los nodos señalados por los punteros de *cadena(p)*, hasta *q* exclusive.

```
                ⎧ Nula                                  si p = q     (esto incluye p = q =
nil)
hazPiz =def     ⎨
                ⎩ Cons( A(p^.elem), hazPiz(p^.sig, q) )  si p ≠ q
```

Nótese que R(xs) garantiza que p^.elem está bien definido en el caso recursivo de *hazPiz*. Y, efectivamente, la función se comporta correctamente cuando la parte izquierda está vacía:

```
    xs.pri = xs.ant
⇒   xs.pri^.sig = xs.ant^.sig      (que puede ser nil si la secuencia está
vacía )
⇒   hazPiz(xs.pri^.sig, xs.ant^.sig) = Nula
```

hazPdr(p) construye una lista abstracta con los elementos localizados en los nodos señalados por los punteros de *cadena(p)*:

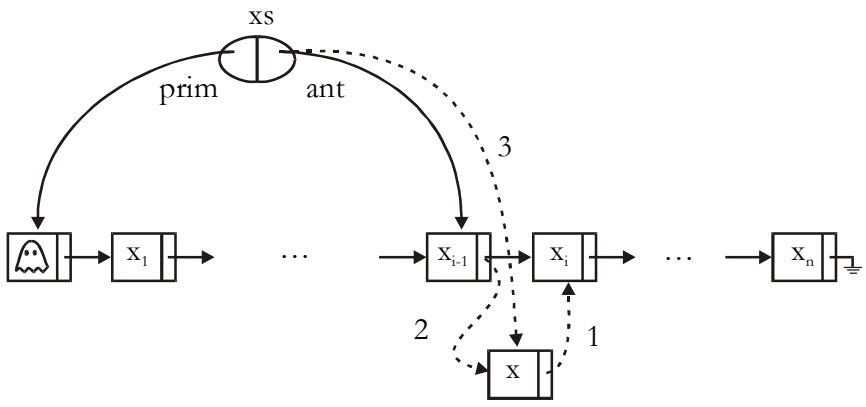
```
                ⎧ Nula                                  si p = nil
hazPdr(p) =def  ⎨
                ⎩ Cons( A(p^.elem), hazPdr(p^.sig) )     si p ≠ nil
```

Nótese que R(xs) garantiza que p^.elem está bien definido en el caso recursivo de *hazPdr*. Y, efectivamente, la función se comporta correctamente cuando la parte derecha está vacía: si *xs.ant^.sig = nil* resulta *hazPdr(xs.ant^.sig) = Nula*.

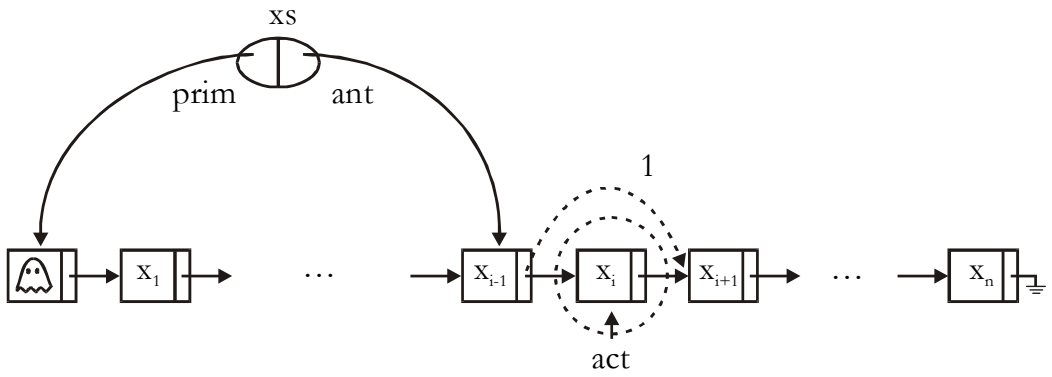
Implementación procedimental de las operaciones

```
proc Crea ( s xs : Sec[Elem] );
{ P0 : cierto }
var
  fantasma : Enlace;
inicio
  ubicar(fantasma);
  fantasma^.sig := nil;
  xs.pri := fantasma;
  xs.ant := fantasma
{ Q0 : R(xs) ∧ A(xs) =SEC[ELEM] Crea }
fproc
```

```
proc inserta ( es xs : Sec[Elem]; e x : Elem );
{ P₀ : xs = XS ∧ R(xs) ∧ R(x) }
var
  nuevo : Enlace;
inicio
  ubicar(nuevo);
  nuevo^.elem := x;
  nuevo^.sig := xs.ant^.sig;  /* 1 */
  xs.ant^.sig := nuevo;       /* 2 */
  xs.ant := nuevo             /* 3 */
{ Q₀ : R(xs) ∧ A(xs) =SEC[ELEM] inserta( A(XS), A(x) ) }
fproc
```




```
proc borra ( es xs : Sec[Elem] );
{ P₀ : xs = XS ∧ R(xs) ∧ NOT fin?(A(xs)) }
var
  act : Enlace;
inicio
  act := x.ant^.sig;
  si act == nil
    entonces
      error("Parte derecha vacía")
    sino
      xs.ant^.sig := act^.sig;    /* 1 */
      ELEM.anular(act^.elem);
      liberar(act)
  fsi
{Q₀ : R(xs) ∧ A(xs) =SEC[ELEM] borra( A(XS) ) }
fproc
```




```
func actual ( xs : Sec[Elem] ) dev x : Elem;
{ P₀ : R(xs) ∧ NOF fin?(A(xs)) }
var
  act : Enlace;
inicio
  act := xs.ant^.sig;
  si act == nil
    entonces
      error("Parte derecha vacía")
    sino
      x := act^.elem;    % x := ELEM.copia(act^.elem)
  fsi
{ Q₀ : A(x) =SEC[ELEM] actual(A(xs)) }
  dev x
ffunc

proc avanza ( es xs : Sec[Elem] );
{ P₀ : xs = XS ∧ R(xs) ∧ NOT fin?(A(xs)) }
var
  act : Enlace;
inicio
  act := xs.ant^.sig;
  si act == nil
    entonces
      error("Parte derecha vacía")
    sino
      xs.ant := act
  fsi
{ Q₀ : R(xs) ∧ A(xs) =SEC[ELEM] avanza( A(XS) ) }
Fproc



proc reinicia ( es xs : Sec[Elem] );
```

```
{ P₀ : R(xs) }
inicio
  xs.ant := xs.pri
{ Q₀ : R(xs) ∧ A(xs) =SEC[ELEM] reinicia( A(XS) ) }
fproc

func vacía? ( xs : Sec[Elem] ) dev r : Bool;
{ P₀ : R(xs) }
inicio
  r := xs.pri^.sig == nil
{ Q₀ : r =SEC[ELEM] vacía?( A(xs) ) }
  dev r
ffunc

func fin? ( xs : Sec[Elem] ) dev r : Bool;
{ P₀ : R(xs) }
inicio
  r := xs.ant^.sig == nil
{ Q₀ : r =SEC[ELEM] fin?( A(xs) ) }
  dev r
ffunc
```

4.5.3 Recorrido y búsqueda en una secuencia

Como hemos indicado anteriormente, las operaciones de las secuencias están pensadas para representar colecciones de datos que se pueden recorrer.

Esquema de recorrido de una secuencia

Un procedimiento genérico de recorrido:

```
proc recorre ( es xs : Sec[elem] );
{ P₀ : xs = XS ∧ piz(xs) = [] }
var
  x : Elem;
inicio
  % reinicia(xs);   si no supusiésemos que piz(xs) = []
{ I : ∀ i : 1 ≤ i ≤ #piz(xs) : tratado( piz(xs) !! i );
  C : #pdr(xs)            }
  it NOT fin?(xs) →
    x := actual(xs);
    tratar(x);
    avanza(xs)
  fit

{ Q₀ : cont(xs) = cont(XS) ∧ fin?(xs) ∧
```

```
        ∀ i : 1 ≤ i ≤ #piz(xs) : tratado( piz(xs) !! i )
fproc
```

Como variante de este esquema, podríamos no pedir *piz(xs)=[ ]* en P_0, y no comenzar con *reinicia(xs)*. De esta forma se recorre la secuencia desde el punto de interés hasta el final.

Nótese que en este ejemplo no hemos cualificado las operaciones con el nombre del TAD. La razón es que estamos usando un único TAD y, por lo tanto, no hay posibilidad de colisiones.

Nótese también que aunque *piz* es una operación privada del TAD, sí la utilizamos en las especificaciones de los procedimientos que usan el TAD.

Esquema de búsqueda de una secuencia

Un procedimiento genérico de recorrido:

```
proc busca ( es xs : Sec[elem]; s encontrado : Bool );
{ P₀ : xs = XS ∧ piz(xs) = [] }
var
  x : Elem;
inicio
  % reinicia(xs);   si no supusiésemos que piz(xs) = []
  encontrado := falso;
{ I : ∀ i : 1 ≤ i ≤ #piz(xs) : ¬prop( piz(xs) !! i ) ∧
      encontrado → NOT fin?(xs) ∧ prop(actual(xs)) );
  C : #pdr(xs)                                          }
```

En el invariante no ponemos *encontrado* ↔ … porque todavía no hemos comprobado si el elemento actual lo cumple o no.

```
  it NOT fin?(xs) AND NOT encontrado →
    x := actual(xs);
    si prop(x)
      entonces encontrado := cierto
      sino     avanza(xs)
    fsi
  fit
{ Q₀ : cont(xs) = cont(XS) ∧
       encontrado ↔ ∃ i : 1 ≤ i ≤ #cont(xs) : prop( cont(xs) !! i ) ∧
       ∀ i : 1 ≤ i ≤ #piz(xs) : ¬prop( piz(xs) !! i ) ∧
       encontrado → prop(act(xs))                                    }
fproc
```

En la postcondición podríamos expresar también que si *encontrado* es falso, entonces estamos al final de la secuencia.

Existe un problema en esta especificación, y es que el último aserto puede estar indefinido si *encontrado* es falso, ya que entonces estamos al final y *act(xs)* no está definido. Esto se enmarca dentro de un problema más general, y es que hasta ahora hemos evitado siempre escribir asertos que pudiesen estar indefinidos. Sin embargo, esta restricción nos obliga en algunos casos –como éste– a escribir especificaciones poco naturales. La cuestión es que tenemos que determinar las características de la lógica con la que estamos trabajando: ha de incluir el valor *indefinido* y no ha

de ser estricta. Si la lógica es estricta, entonces cuando una parte de una fórmula está indefinida, lo está la fórmula entera. Operacionalmente podemos interpretar la implicación del ejemplo como que cuando la parte izquierda de la implicación es falso, entonces afirmo que la implicación es cierta, y no me preocupo por el lado derecho.

Me corrijo, en realidad en este caso es inevitable utilizar un aserto que puede estar indefinido, porque la alternativa podría ser:

```
(¬encontrado) ∨ (encontrado ∧ prop(act(xs)))
```

que adolece del mismo problema.

Como variante de este esquema, podríamos no pedir *piz(xs)=[ ]* en P_0, y no comenzar con *reinicia(xs)*. De esta forma se busca en la secuencia desde el punto de interés hacia la derecha.

Si *prop(x)* $\Leftrightarrow x == u$, con *u* dado como parámetro, entonces se trata de la búsqueda de un valor dentro de una secuencia.

Ejemplos de aplicación de los esquemas de recorrido y búsqueda

Conteo en una secuencia de enteros

Dada *xs : Sec[Ent]*, queremos contar cuántas posiciones hay en ella que contengan un entero igual a la suma de los enteros en todas las posiciones anteriores. Este es un ejemplo de aplicación del esquema de recorrido:

```
proc conteo ( es xs : Sec[Ent]; s r : Ent );
{ P₀ : xs = XS ∧ piz(xs) = [] }
var
  x : Elem;
  s : Ent;  % lleva la suma hasta ese momento
inicio
  s := 0;
  { I : cont(xs) = cont(XS) ∧
        r = # i : 1 ≤ i ≤ #piz(xs) :
                  cont(xs) !! i = Σ j : 1 ≤ j < i : cont(xs) ¡!! j ∧
        s = Σ i : 1 ≤ i ≤ #piz(xs) : cont(xs) !! i ;
    C : #pdr(xs)                                                    }
  it NOT fin?(xs) →
    x := actual(xs);
    si x == s                    ⎫
      entonces                   ⎪
        <r, s> := <r+1, s+x>     ⎪
      sino                       ⎬  tratar(x)
        s := s+x                 ⎪
    fsi;                         ⎭
    avanza(xs)
  fit;
```

```
{ Q₀ : cont(xs) = cont(XS) ∧ fin?(xs) ∧
       r = # i : 1 ≤ i ≤ #cont(xs) :
                 cont(xs) !! i = Σ j : 1 ≤ j < i : cont(xs) !! j }
fproc
```

Hay que implementarlo como un procedimiento porque se modifica el punto de interés de *xs*.

Búsqueda en una secuencia de enteros

Dada *xs : Sec[Ent]*, queremos buscar la primera posición donde aparezca un entero que no sea mayor o igual que todos los anteriores, es decir, el primero que sea menor que el situado a su izquierda. Gráficamente:

$x_1 \leq x_2 \leq \ldots \leq x_p > x_{p+1}$

No consideramos al primero como candidato. Por ello, eso lo implementamos como una modificación del esquema de búsqueda, donde tratamos por separado al primer elemento, lo cual nos obliga a verificar que efectivamente hay un primer elemento –i.e., la secuencia no está vacía–.

```
proc busca( es xs : Sec[Ent]; s encontrado : Bool );
{ P₀ : piz(xs) = [] }
var
  x : Elem;
  m : Ent;    % para guardar el anterior
inicio
  encontrado := Falso;
  si vacía?(xs)
    entonces
      seguir
    sino
      m := actual(xs);
      avanza(xs);
{ I : noDecreciente(piz(xs)) ∧ m = ultimo(piz(xs)) ∧
      encontrado → NOT fin?(xs) ∧ actual(xs) < m;
  C : #pdr(xs)                                          }
      it NOT fin?(xs) AND NOT encontrado →
        x := actual(xs);
        si x < m
          entonces
            encontrado := cierto
          sino
            m := x;
            avanza(xs)
        fsi
      fit
  fsi
{ Q₀ : cont(xs) = cont(XS) ∧ noDecreciene( piz(xs) ) ∧
```

```
        encontrado ↔ ∃ i : 2 ≤ i ≤ #cont(xs) : cont(xs) !! i < cont(xs) !! i
- 1 ∧
        encontrado → actual(xs) < último(piz(xs))
}
```

Donde el predicado auxiliar *noDecreciente* indica que la lista está ordenada en orden no decreciente:

```
noDecreciente( ls ) ⇔def ∀ i, j : 1 ≤ i < j ≤ #ls : (ls !! i) ≤ (ls !! j)
```

Mezcla ordenada de secuencias

El último ejemplo que vamos a estudiar es el de la mezcla de dos secuencias ordenadas.

```
proc mezcla( es xs, ys : Sec[Elem] ; s zs : Sec[Elem] )
{ P₀ : xs = XS ∧ ys = YS ∧ ordenada(cont(xs)) ∧ piz(xs) = [] ∧
      ordenada(cont(ys)) ∧ piz(ys) = []                          }
var
  x, y : Elem;
inicio
  Crea(zs);
{ I : cont(xs) = cont(XS) ∧ cont(ys) = cont(YS) ∧ fin?(zs) ∧
ordenada(cont(zs)) ∧
     permutación(cont(zs) , piz(xs) ++ piz(ys);
  C : #pdr(xs) + #pdr(ys)
}
  it NOT fin?(xs) AND NOT fin?(ys) →
    x := actual(xs);
    y := actual(ys);
    si x ≤ y
      entonces
        inserta(x, zs);
        avanza(xs)
      sino
        inserta(y, zs);
        avanza(ys)
    fsi
  fit;
{ I; C }  % la misma cota e invariante del bucle anterior
  it NOT fin?(xs) →
    inserta(actual(xs), zs);
    avanza(xs)
  fit;
{ I; C }  % la misma cota e invariante del bucle anterior
  it NOT fin?(ys) →
    inserta(actual(ys), zs);
    avanza(ys)
  fit;
```

```
{ Q₀ : cont(xs) = cont(XS) ∧ cont(ys) = cont(YS) ∧ ordenada(cont(zs)) ∧
       permutación(cont(zs), cont(xs) ++ cont(ys))                    }
fproc
```

4.6 Ejercicios

Pilas

177. Las pilas formadas por números naturales del intervalo [0..N–1] (para cierto N ≥ 2 fijado de antemano) se pueden representar por medio de números, entendiendo que un número natural P cuya representación en base N tenga "1" como dígito de mayor peso representa la pila formada por los restantes dígitos de la representación de P en base N (excepto el de mayor peso), siendo la cima el dígito de menor peso. Por ejemplo, si N = 10, el número 1073 representa la pila que contiene los números 0, 7, 3, con 0 en la base y 3 en la cima. Formaliza el invariante de la representación y la función de abstracción siguiendo esta idea. Comprueba que esta representación permite implementar todas las operaciones de las pilas como funciones de coste *O*(1).

178. Define el tipo representante, el invariante de la representación y la función de abstracción adecuados para una representación dinámica de las pilas.

179. Desarrolla una implementación de las operaciones del TAD PILA mediante funciones, basándote en la representación del ejercicio anterior.

180. Supongamos disponible un módulo que exporte pilas de caracteres, implementadas con la técnica de los ejercicios 178 y 179. Representa gráficamente la estructura resultante de ejecutar lo siguiente:

```
var
  p, p1, p2, p3, p4 : Pila[Car];
  p := PilaVacía();
  p1 := Apilar( 'b', Apilar('a', p) );
  p2 := Apilar( 'c', p1 );
  p3 := Apilar( 'd', p1 );
  p4 := Apilar( 'e', p3 );
  p3 := Apilar( 'f', p3 );
  p1 := desapilar( p4 );
```

Observa que en esta implementación funcional de las pilas, la ejecución de una operación nunca destruye estructuras creadas previamente por la ejecución de otras operaciones. ¿Qué sucederá con el espacio ocupado por la estructura, si continuamos ejecutando lo que sigue?

```
p  := PilaVacía();
p1 := PilaVacía(); p2 := PilaVacía();
p3 := PilaVacía(); p4 := PilaVacía();
```

181. Usando la representación del ejercicio 178, especifica y programa un procedimiento con cabecera

```
proc anular( es xs : Pila[Elem] )
```

cuyo efecto sea liberar todo el espacio ocupado por *xs* antes de la llamada, dejando como nuevo valor de *xs* la representación de la pila vacía.

182. Usando la representación del ejercicio 178, completa el desarrollo de una implementación del TAD PILA que realice las operaciones generadoras y modificadoras como procedimientos. Comenta las ventajas e inconvenientes frente a la implementación estática de las pilas, estudiada en los ejercicios 165, 166 y 172.

183. Especifica un TAD parametrizado DOS_PILAS[E :: ANY], con un tipo *Pilas[Elem]* y operaciones adecuadas. La idea es que un dato de tipo *Pilas* representa una pareja de pilas, que pueden manejarse independientemente del modo usual. Desarrolla una implementación estática de este TAD, representando la pareja de pilas con ayuda de un único vector de almacenamiento. Organiza la implementación de modo que las dos pilas crezcan en sentidos opuestos, cada una desde uno de los dos extremos del vector.

184. Especifica un enriquecimiento SUPER_PILA del TAD PILA, añadiendo una nueva operación con la siguiente especificación informal:

`desapilarK: (Nat, Pila[Elem]) - → Pila[Elem]`

Modificadora. *desapilarK*(k, xs) desapila los *k* elementos de *xs* más próximos a la cima, si hay suficientes elementos.

†185. Aplicando la técnica de *análisis amortizado*, razona que el tiempo de ejecución de m_1 llamadas a *apilar* y m_2 llamadas a *desapilarK* es $O(m_1)$, siempre que todas las llamadas se refieran a una misma pila, inicialmente vacía.

Eliminación de la recursión lineal no final con ayuda de una pila

186. Recordemos que el esquema general de definición de una función recursiva lineal no final es de la forma:

```
func f( x : T ) dev y : S;
{ P₀ : P(x); Cota : t(x) }
var
  x' : T; y' : S;
% Otras posibles declaraciones locales
inicio
  si d(x)
    entonces
    { P(x) ∧ d(x) }
      y ;= r(x)
    { Q(x, y) }
    sino
    { P(x) ∧ ¬d(x) }
      x' := s(x);
    { x' = s(x) ∧ P(x') }
      y' := f(x');
    { x' = s(x) ∧ Q(x', y') }
      y := c(x, y');
    { Q(x, y) }
  fsi
{ Q₀ : Q(x, y) }
```

```
  dev y
ffunc
```

Supongamos que la función *s* encargada de calcular los parámetros de la siguiente llamada recursiva posea una inversa s^{-1}. En este caso, es posible transformar *f* en una definición iterativa equivalente, de la forma siguiente:

```
func f_it( x : T ) dev y : S;
{ P_0 : P(x) }
var
  x' : T;
% Otras posibles declaraciones locales
inicio
  x' := x;
{ Inv. I: R(x', x); Cota: t(x') }
  it ¬d(x') →
  { I ∧ ¬d(x') }
    x' := s(x')
  { I }
  fit;
{ I ∧ d(x') }
  y := r(x');
{ Inv. J: R(x', x) ∧ y = f(x'); Cota: m(x', x) }
  it x' /= x →
  { J ∧ x' ≠ x }
    x' := s^-1(x');
    y := c(x', y)
  { J }
  fit
{ J ∧ x' = x }
{ y = f(x) }
{ Q_0 : Q(x, y) }
  dev y
ffunc
```

siendo

$$R(x', x)$$

$\Leftrightarrow_{def}$

$$\exists\, n : Nat : (x' = s^n(x) \wedge P(x') \wedge \forall\, i : 0 \le i < n : (P(s^i(x)) \wedge \neg d(s^i(x))))$$

(*Idea*: expresa que *x'* desciende de *x* después de un cierto número de llamadas recursivas)

$$m(x', x)$$

$=_{def}$

$$\min n : Nat : x' = s^n(x)$$

(*Idea*: expresa el número de llamadas recursivas necesarias para alcanzar *x'* desde *x*)

Aplica esta transformación a las siguientes funciones recursivas lineales:

*(a) Función *fact* (ejercicio 79).
(b) Función *dosFib* (ejercicio 99, 128)

187. Pensemos de nuevo en una función recursiva lineal *f* definida según el esquema del ejercicio 186. si la función inversa s^{-1} no existe o es muy costosa de calcular, se puede aplicar otra transformación a forma iterativa, usando una pila para almacenar los parámetros de las sucesivas llamadas recursivas. La versión iterativa de *f* queda ahora:

```
func f_it( x : T ) dev y : S;
{ P_0 : P(x) }
var
  x' : T;
  xs : Pila[T];
% Otras posibles declaraciones locales
inicio
  x' := x;
  PilaVacía(xs);
{ Inv. I: R(xs, x', x); Cota: t(x') }
  it ¬d(x') →
  { I ∧ ¬d(x') }
    Apilar(x', xs);
    x' := s(x')
  { I }
  fit;
{ I ∧ d(x') }
  y := r(x');
{ Inv. J: R(xs, x', x) ∧ y = f(x'); Cota: tamaño(xs) }
  it NOT esVacía(xs) →
  { J ∧ ¬esVacía(xs) }
    x' := cima(xs);
    y := c(x', y);
    desapilar(xs)
  { J }
  fit
{ J ∧ esVacía(xs) }
{ y = f(x) }
{ Q_0 : Q(x, y) }
  dev y
ffunc
```

siendo

$$R(xs, x', x)$$
$$\Leftrightarrow_{def}$$
$$\exists n : Nat : (x' = s^n(x) \land P(x') \land \forall i: 0 \le i < n: (P(s^i(x)) \land \neg d(s^i(x)) \land s^i(x) = elem(i, xs)))$$

donde *elem*(i, xs) indica el elemento que ocupa el lugar *i* en la pila *xs*, contando el elemento del fondo como *elem*(0, xs)

(*Idea*: expresa que *x'* desciende de *x* después de un cierto número de llamadas recursivas, cuyos parámetros reales, hasta *x'* exclusive, están apilados en *xs*)

```
        tamaño(xs)
=def
        número de elementos apilados en xs
```

(*Idea*: expresa el número de llamadas recursivas necesarias para alcanzar *x'* desde *x*)

Aplica la transformación que acabamos de definir a las siguientes funciones recursivas lineales:

(a) Funciones *pot* y *pot'* (ejercicio 85).
(b) Función *mult* (ejercicio 90)
(c) Función *log* (ejercicio 91).
*(d) Función *bin* (ejercicio 92)

188. Enriquece la especificación del TAD PILA[E :: ANY], añadiendo las ecuaciones que definan el comportamiento de las operaciones *elem* y *tamaño* que hemos definido informalmente en el ejercicio anterior.

189. En la práctica, cuando se aplica la transformación de recursión lineal a iteración que hemos descrito en el ejercicio 187, no es necesario apilar toda la información correspondiente a una tupla x : T. En ciertos casos, es suficiente apilar un dato u : Z, más simple que *x*, elegido de modo que *x* sea fácil de calcular a partir de *u* y *s(x)*. Más exactamente, deben estar disponibles dos funciones *g* y *h* que verifiquen:

¬d(x) ⇒ u = g(x) está definido y cumple que x = h(u, s(x))

Modificando el esquema del ejercicio 187 según esta idea, se obtiene la siguiente versión iterativa de *f*:

```
func f_it( x : T ) dev y : S;
{ P_0 : P(x) }
var
  x' : T; u : Z;
  us : Pila[Z];
% Otras posibles declaraciones locales
inicio
  x' := x;
  PilaVacía(us);
{ Inv. I: R(us, x', x); Cota: t(x') }
  it ¬d(x') →
  { I ∧ ¬d(x') }
    u := g(x');
    Apilar(u, us);
    x' := s(x')
  { I }
```

```
  fit;
{ I ∧ d(x') }
  y := r(x');
{ Inv. J: R(us, x', x) ∧ y = f(x'); Cota: tamaño(us) }
  it NOT esVacía(us) →
  { J ∧ ¬esVacía(us) }
    u := cima(us);
    x' := h(u,x');
    y := c(x', y);
    desapilar(us)
  { J }
  fit
{ J ∧ esVacía(us) }
{ y = f(x) }
{ Q₀ : Q(x, y) }
  dev y
ffunc
```

siendo

$$R(us, x', x)$$

$\Leftrightarrow_{def}$

$$\exists n: Nat: (x' = s^n(x) \wedge P(x') \wedge \forall i: 0 \leq i < n: (P(s^i(x)) \wedge \neg d(s^i(x)) \wedge g(s^i(x)) = elem(i, us)))$$

donde *elem*(i, us) indica el elemento que ocupa el lugar *i* en la pila *us*, contando el elemento del fondo como *elem*(0, us)

(*Idea*: expresa que *x'* desciende de *x* después de un cierto número de llamadas recursivas, y que en *us* está apilada información suficiente para recuperar los parámetros reales de dichas llamadas)

$$tamaño(us)$$

$=_{def}$

número de elementos apilados en us

(*Idea*: expresa el número de llamadas recursivas necesarias para alcanzar *x'* desde *x*)

Comprueba que esta transformación optimizada se puede aplicar a las funciones recursivas lineales del ejercicio 187, utilizando en todos los casos una pila de valores booleanos. Precisa quiénes son en cada caso las funciones denominadas *g* y *h* en el esquema.

Colas

190. Especifica algebraicamente un TAD genérico COLA[e :: ANY], partiendo de la siguiente especificación informal de las operaciones deseadas:

- `ColaVacía: → Cola[Elem]`
  Generadora. Crea una cola vacía.

- `Añadir: (Elem, Cola[Elem]) → Cola[Elem]`
  Generadora. Añade un nuevo elemento al final de una cola.
- `avanzar: Cola[Elem] - → Cola[Elem]`
  Modificadora. Retira el primer elemento de una cola no vacía.
- `primero: Cola[Elem] - → Elem`
  Observadora. Consulta el primer elemento de una cola no vacía.
- `esVacía: Cola[Elem] → Bool`
  Observadora. Reconoce si una cola es vacía o no.

191. Plantea una implementación estática del TAD COLA basada en un vector, similar a la que ya conocemos para el caso de las pilas (ejercicios 165 y 166). Observa cómo evoluciona la zona ocupada del vector al ir ejecutando llamadas a las operaciones *Añadir* y *avanzar*. ¿Cómo afecta este problema a la eficiencia de la implementación?

192. Desarrolla una implementación estática del TAD COLA basada en un vector circular, realizando las operaciones generadoras y modificadoras como procedimientos. Observa que el tiempo de ejecución es $O(1)$ para todas las operaciones.

193. Desarrolla una implementación dinámica del TAD COLA, realizando las operaciones generadoras y modificadoras como procedimientos.

194. Usando la representación del ejercicio 193, especifica y programa una función con cabecera

```
func copia( xs : Cola[elem] ) dev ys : Cola[Elem]
```

que devuelva en *ys* un puntero a una estructura que represente a la misma cola que *xs*, pero ocupando celdas de memoria diferentes de las que ocupa la estructura apuntada por *xs*. Observa que la asignación `xs := ys` no tendría este efecto.

195. Una frase se llama palíndroma si la sucesión de caracteres obtenida al recorrerla de izquierda a derecha (ignorando los blancos) es la misma que si el recorrido se hace de derecha a izquierda. Esto sucede, por ejemplo, con la socorrida frase "dábale arroz a la zorra el abad". Construye una función iterativa ejecutable en tiempo lineal, que decida si una frase dada en forma de *cola de caracteres* es o no palíndroma. El algoritmo utilizará una *pila de caracteres*, declarada localmente.

Idea: Primeramente, el algoritmo recorre la cola y va apilando los caracteres no blancos. A continuación, se vuelve a recorrer la cola, comparando los caracteres no blancos con los caracteres almacenados en la pila. El algoritmo debe usar la función *copia* del ejercicio anterior, para garantizar que al final de la ejecución la estructura representante de la cola (parámetro de entrada) no se haya alterado.

Colas dobles

196. El TAD parametrizado DCOLA[E :: ANY] especifica el comportamiento de las *colas dobles*, similar al de las colas, pero con operaciones que permiten el acceso a los dos extremos de la sucesión de los elementos de la cola. La especificación informal de las operaciones de las colas dobles es como sigue:

- `DColaVacía: → DCola[Elem]`

Generadora. Crea una cola vacía.

- `PonDetrás: (Elem, DCola[Elem]) → DCola[Elem]`
  Generadora. Añade un nuevo elemento al final de una cola.
- `ponDelante: (Elem, DCola[Elem]) → DCola[Elem]`
  Modificadora. Añade un nuevo elemento al principio de una cola.
- `quitaUlt: DCola[Elem] - → DCola[Elem]`
  Modificadora. Retira el último elemento de una cola no vacía.
- `último: DCola[Elem] - → Elem`
  Observadora. Consulta el último elemento de una cola no vacía.
- `quitaPrim: DCola[Elem] - → DCola[Elem]`
  Modificadora. Retira el primer elemento de una cola no vacía.
- `primero: DCola[Elem] - → Elem`
  Observadora. Consulta el primer elemento de una cola no vacía.
- `esVacía: DCola[Elem] → Bool`
  Observadora. Reconoce si una cola es vacía o no.

Completa la especificación algebraica de este TAD.

197. Modifica la implementación estática del ejercicio 192 para adaptarla a las colas dobles.

198. Modifica la implementación dinámica del ejercicio 193 para adaptarla a las colas dobles.

199. Observa que el tiempo de ejecución de *quitaUlt* en la implementación del ejercicio anterior es *O*(n), debido a que el penúltimo nodo sólo puede localizarse recorriendo toda la estructura desde el primer nodo. Modifica la implementación, de manera que la estructura representante de una cola esté *doblemente enlazada*; cada nodo deberá incorporar un puntero al siguiente y un puntero al anterior. Implementa todas las operaciones en base a la nueva representación, obteniendo procedimientos ejecutables en tiempo *O*(1).

200. El agente 0069 ha inventado un nuevo método de codificación de mensajes secretos. El mensaje original X se codifica en dos etapas:

- En primer lugar, X se transforma en X' reemplazando cada sucesión de caracteres consecutivos que no sean vocales por su imagen especular.
- En segundo lugar, X' se transforma en la sucesión de caracteres X" obtenida al ir tomando sucesivamente: el primer carácter de X'; luego el último; luego el segundo; luego el penúltimo; etc.

Ejemplo: para X = "*Bond, James Bond*", resultan:

X' = "*BoJ ,dnameB sodn*" y X" = "*BnodJo s, dBneam*"

Construye los algoritmos de codificación y descodificación de mensajes y analiza su complejidad, utilizando pilas y colas. Supón en particular que el mensaje inicial viene dado como una cola de caracteres.

Especificación del TAD LISTA

201. Especifica algebraicamente un TAD genérico LISTA[E :: ANY], partiendo de la siguiente especificación informal de las operaciones deseadas:

- `Nula:` → `Lista[Elem]`
  Generadora. Crea una lista vacía.
- `Cons: (Elem, Lista[Elem])` → `Lista[Elem]`
  Generadora. *Cons(x,xs)* genera una nueva lista añadiendo *x* como primer elemento, por delante de los elementos de *xs*.
- `nula?: Lista[Elem]` → `Bool`
  Observadora. Reconoce si una lista es vacía o no.
- `primero: Lista[Elem] -` → `Elem`
  Observadora. Consulta el primer elemento de una lista no vacía.
- `resto: Lista[Elem] -` → `Lista[Elem]`
  Modificadora. Quita el primer elemento de una lista no vacía.

202. Enriquece el TAD parametrizado de las listas con nuevas operaciones, partiendo de las siguientes especificaciones informales:

- `( ++ ): (Lista[Elem], Lista[Elem])` → `Lista[Elem]`
  Modificadora. *xs* ++ *ys* construye la concatenación de *xs* con *ys*, dando una nueva lista formada por los elementos de *xs* seguidos de los de *ys*.
- `[ _ ]: Elem` → `Lista[Elem]`
  Modificadora. [*x*] es la lista formada por el único elemento *x*.
- `(# ): Lista[Elem]` → `Nat`
  Observadora. # *xs* devuelve la longitud de la lista *xs*.
- `( !! ): (Lista[Elem], Nat) -` → `Elem`
  Observadora. *xs* !! *i* está definido si 1 ≤ i ≤ n, siendo *n* = # *xs*, y devuelve el elemento de lugar *i* de *xs*.
- `miembro: (Elem, Lista[Elem])` → `Bool`
  Observadora. Reconoce si un elemento es miembro de una lista.
- `ponDr: (Lista[Elem], Elem)` → `Lista[Elem]`
  Modificadora. Añade un elemento a la derecha de una lista.
- `último: Lista[Elem] -` → `Elem`
  Observadora. Devuelve el último elemento de una lista no vacía.
- `inicio: Lista[Elem] -` → `Lista[Elem]`
  Modificadora. Quita el último elemento de una lista no vacía.

Implementación del TAD LISTA

203. Discute una posible implementación estática del TAD LISTA, basada en una representación similar a las que ya conocemos para implementaciones estáticas de pilas y colas. ¿Qué operaciones resultan problemáticas, y por qué?

204. Adapta la representación dinámica de las colas utilizada en el ejercicio 198 al TAD LISTA. Usando esta representación, programa una función *copia* que copie la estructura representante de una lista, y un procedimiento *anula* que libere todo el espacio ocupado por la estructura representante de una lista.

205. Localiza aquellas operaciones del TAD LISTA que tienen una análoga en el TAD DCOLA, y comprueba que los algoritmos estudiados en los ejercicios 198 y 199 se pueden adaptar para implementarlas, usando la representación del ejercicio 204.

206. Usando la misma representación del ejercicio anterior, desarrolla una implementación procedimental de la operación (++), que satisfaga la siguiente especificación Pre/Post:

```
proc conc( es xs : Lista[Elem]; e ys : Lista[Elem] )
{ P₀ : R(xs) ∧ R(ys) ∧ xs = XS ∧ ys = YS }
{ Q₀ : R(xs) ∧ R(ys) ∧ A(xs) =LISTA[ELEM] A(XS) ++ A(ys) }
fproc
```

Estudia el tiempo de ejecución de *conc*.

207. Desarrolla una implementación funcional del TAD LISTA usando la misma representación de los dos ejercicios anteriores. Analiza los tiempos de ejecución de las operaciones Estudia si se necesitan estructuras compartidas (para mejorar la eficiencia), o copias de estructuras (para proteger parámetros de entrada). Plantea posibles modificaciones del tipo representante, que sirvan para mejorar la eficiencia de alguna operación.

Programación recursiva con listas

En los ejercicios de este apartado, suponemos disponible un módulo que exporte una implementación funcional del TAD LISTA. Debemos programar usando solamente las operaciones exportadas, sin punteros. para simplificar, ignoraremos cuestión de gestión de memoria (*copiar*, *anular*).

208. Programa funciones recursivas que realicen el comportamiento correcto de las operaciones (#), (!!) y (++), usando otras operaciones más básicas del TAD LISTA. Analiza el tiempo de ejecución de los algoritmos obtenidos.

209. Usando la especificación del TAD LISTA y razonando por inducción sobre la estructura de los términos generados, demuestra que son válidas las siguientes ecuaciones. *Atención*: cada igualdad debe entenderse en el sentido de que sus dos miembros representan la misma lista abstracta.

```
∀ xs, ys, zs : Lista[Elem] : ∀ x, y : Elem :

 (a)  último( [xs \ x] ) = x
 (b)  inicio( [xs \ x] ) = xs
*(c)            xs ++ [] = xs
*(d)    xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
 (e)      xs ++ [ys \ y] = [(xs ++ ys) \ y]
```

210. Formaliza la especificación Pre/Post de las dos funciones que siguen, y construye algoritmos recursivos que las satisfagan

```
(a) func coge( n : Nat; xs : Lista[Elem] ) dev us : Lista[Elem]
    { P₀ : cierto }
    { Q₀ : us es la lista formada por los n primeros elementos de xs }
```

```
     ffunc
(b) func tira( n : Nat; xs : Lista[Elem] ) dev vs : Lista[Elem]
     { P₀ : cierto }
     { Q₀ : vs es la lista formada quitando los n primeros elementos de xs }
     ffunc
```

211. Demuestra usando inducción que la igualdad

```
coge(n, xs) ++ tira(n, xs) = xs
```

vale para *xs, ys : Lista[Elem]* y *n : Nat* cualesquiera. *Atención*: la igualdad debe entenderse en el sentido de que sus dos miembros representan la misma lista abstracta.

212. La siguiente función recursiva calcula la inversa de una lista dada:

```
func inv( xs : Lista[Elem] ) dev ys : Lista[Elem];
{ P₀ : cierto }
inicio
  si nula?(xs)
    entonces ys := Nula
    sino     ys := inv(resto(xs)) ++ [primero(xs)]
  fsi
{ Q₀ : ∀ : 1 ≤ i ≤ (# xs) : xs !! i = ys !! (# xs)-i+1 }
ffunc
```

Usando la técnica de plegado-desplegado, deriva un algoritmo recursivo final para la función más general *invSobre*, especificada como sigue:

```
func invSobre( as, xs : Lista[Elem] ) dev ys : Lista[Elem];
{ P₀ : cierto }
{ Q₀ : ys = inv(xs) ++ as }
ffunc
```

Analiza los tiempos de ejecución de *inv* e *invSobre*. Compara.

213. Especifica un TAD parametrizado LISTA-ORD[E :: ORD] que enriquezca el TAD LISTA[E :: ANY] con nuevas operaciones especificadas informalmente como sigue:

- `( == ): (Lista[Elem], Lista[Elem]) → Bool`
  Observadora. Operación de igualdad entre listas.
- `( ≤ ): (Lista[Elem], Lista[Elem]) → Bool`
  Observadora. Orden entre listas.
- `ordenada: Lista[Elem] → Bool`
  Observadora. Decide si una lista está ordenada.
- `insertaOrd: (Elem, Lista[Elem]) → Lista[Elem]`
  Modificadora. Inserta un elemento de manera que la lista resultante queda ordenada si la lista original lo estaba.

214. Plantea una implementación del TAD LISTA-ORD especificado en el ejercicio anterior, extendiendo la implementación de LISTA con funciones recursivas que realicen las operaciones extra de LISTA-ORD. Observa que no es necesario definir un tipo representante para Lista[Elem], ya que Lista[Elem] se puede importar del módulo que implementa a LISTA. Tampoco es necesario usar punteros.

215. Construye una función recursiva simple que satisfaga la especificación que sigue, y aplica el método de plegado-desplegado para transformarla en una función recursiva final más general. Suponemos que se trata de listas ordenas de tipo *Lista[Elem]*.

```
func mezcla( xs, ys : Lista[Elem] ) dev zs : Lista[Elem]
{ P₀ : ordenada(xs) ∧ ordenada(ys) }
{ Q₀ : ordenada(zs) ∧ permutación( zs, xs++ys ) }
```

216. Transforma el algoritmo recursivo final obtenido en el ejercicio anterior en un algoritmo iterativo.

217. Construye especificaciones Pre/Post y funciones recursivas que resuelvan los dos problemas que siguen:

(a) Dada una lista de enteros *xs*, construir otra lista *ys* formada por los números pares que aparezcan en *xs*, tomados en el mismo orden.

(b) Dada una lista de enteros *xs*, construir dos listas *us,vs* formadas respectivamente por los números pares e impares que aparezcan en *xs*, tomados en el mismo orden.

218. Usa las técnicas de plegado-desplegado y eliminación de la recursión final para transformar los algoritmos recursivos obtenidos en el ejercicio anterior en algoritmos iterativos ejecutables en tiempo $O(n)$, siendo n la longitud de *xs*.

Especificación e implementación de las secuencias

†219. El TAD parametrizado SEC[E :: ANY] especifica el comportamiento de las *secuencias*, también llamadas *listas con punto de interés*. Una secuencia se comporta como una lista dividida en una parte *derecha* y una parte *izquierda*. El *punto de interés* se imagina como el punto divisorio entre estas dos partes. La especificación informal de las operaciones de las secuencias es como sigue:

- `Crea:` → `Sec[Elem]`
  Generadora. Crea una secuencia vacía.
- `Inserta: (Sec[Elem], Elem)` → `Sec[Elem]`

Generadora. Inserta un nuevo elemento a la izquierda del punto de interés.

- `borra: Sec[Elem] -` → `Sec[Elem]`
  Modificadora. Elimina el elemento situado a la derecha del punto de interés, si lo hay.
- `actual: Sec[Elem] -` → `Elem`
  Observadora. Consulta el elemento situado a la derecha del punto de interés, si lo hay.
- `Avanza: Sec[Elem] -` → `Sec[Elem]`
  Generadora. Desplaza un lugar a la derecha el punto de interés, si es posible.

- `Reinicia: Sec[Elem] → Sec[Elem]`
  Generadora. Posiciona el punto de interés a la izquierda del todo.
- `vacía?: Sec[Elem] → Bool`
  Observadora. Reconoce si la secuencia está vacía.
- `fin?: Sec[Elem] → Bool`
  Observadora. Reconoce si el punto de interés se encuentra a la derecha del todo.

Completa la especificación algebraica de este TAD.

220. Plantea una posible implementación del TAD secuencia utilizando una representación estática. ¿Qué tiempo de ejecución se obtiene para las operaciones *inserta* y *borra*?

†221. Desarrolla una implementación dinámica del TAD secuencia realizando las operaciones generadoras y modificadoras como procedimientos, de manera que todas las operaciones sean ejecutables en tiempo $O(1)$.

Aplicaciones de las secuencias y otras estructuras lineales

222. Las especificaciones que siguen corresponden a *esquemas de procesamiento de secuencias* de uso común en muchas aplicaciones. Construye en cada caso un algoritmo que satisfaga la especificación, suponiendo disponible un módulo que implemente el TAD de las secuencias.

(a) Esquema de *recorrido secuencial*:

```
proc recorre( es xs : Sec[Elem] );
{ P₀ : xs = XS ∧ piz(xs) = [] }
{ Q₀ : cont(xs) = cont(XS) ∧ fin?(xs) = cierto ∧
      tratar se ha aplicado a todos los elementos de xs }
fproc
```

(b) Esquema de *búsqueda secuencial*:

```
proc busca( es xs : Sec[Elem]; s encontrado : Bool );
{ P₀ : xs = XS ∧ piz(xs) = [] }
{ Q₀ : cont(xs) = cont(XS) ∧
      (encontrado ↔ existe en xs algún elemento que cumpla prop) ∧
      (encontrado → actual(xs) cumple prop ∧
                    los elementos de piz(xs) no cumplen prop)      }
fproc
```

NOTA: Si el esquema de búsqueda se usa para buscar un elemento dado z (es decir, si se tiene prop(x) $\Leftrightarrow$ x = z), conviene modificar el planteamiento del esquema introduciendo z como parámetro.

223. Resuelve los problemas que siguen aplicando el esquema de recorrido de secuencias. En cada caso, el contenido de la secuencia debe quedar inalterado.

(a) Contar el número de apariciones de 'a' en una secuencia de caracteres dada.

(b) Contar el número de apariciones de vocales en una secuencia de caracteres dada.

(c) Dada una secuencia de enteros, contar cuantas posiciones hay en ella tales que el entero que aparece en esa posición es igual a la suma de todos los precedentes.

224. Resuelve los problemas que siguen aplicando el esquema de búsqueda secuencial.

(a) Buscar la primera aparición de 'b' en una secuencia de caracteres dada.

(b) Buscar la primera aparición de una consonante en una secuencia de caracteres dada.

(c) Dada una secuencia de enteros , buscar la primera posición ocupada por un número que no sea mayor o igual que todos los anteriores.

225. Algunos problemas de procesamiento de secuencias requieren modificar y/o combinar los esquemas de recorrido y búsqueda secuencial. Resuelve los casos siguientes:

(a) Dada una secuencia de caracteres, copiarla en otra eliminando los blancos múltiples.

(b) Contar el número de apariciones de 'a' posteriores a la primera aparición de 'b' en una secuencia de caracteres dada.

(c) Contar el número de caracteres anteriores y posteriores a la primera aparición de 'a' en una secuencia de caracteres dada.

(d) Contar el número de parejas de vocales consecutivas que aparecen en una secuencia de caracteres dada.

226. Construye un procedimiento iterativo que satisfaga la especificación que sigue, suponiendo que disponemos de un orden ≤ para el tipo *Elem*.

```
proc mezcla( es xs, ys : Sec[Elem] ; s zs : Sec[Elem] )
{ P₀ : xs = XS ∧ ys = YS ∧ ordenada(cont(xs)) ∧ piz(xs) = [] ∧
       ordenada(cont(ys)) ∧ piz(ys) = []                          }
{ Q₀ : cont(xs) = cont(XS) ∧ cont(ys) = cont(YS) ∧ ordenada(cont(zs)) ∧
        permutación(cont(zs), cont(xs) ++ cont(ys))                    }
fproc
```

227. Una expresión aritmética construida con los operadores binarios '+', '–', '*', '/' y operandos (representados cada uno por un solo carácter) se dice que está en forma *postfija* si es o bien un solo operando o dos expresiones en forma postfija una tras otra, seguidas inmediatamente de un operador. Lo que sigue es un ejemplo de una expresión escrita en la notación infija habitual, junto con su forma postfija:

Forma infija: `(A/(B-C))*(D+E))` Forma postfija: `ABC-/DE+*`

Diseña un algoritmo iterativo que calcule el valor de una expresión dada en forma postfija por el siguiente método: se inicializa una pila vacía de números y se van recorriendo de izquierda a derecha los caracteres de la expresión. Cada vez que se pasa por un operando, se apila su valor. Cada vez que se pasa por un operador, se desapilan los dos números más altos de la pila, se componen con el operador, y se apila el resultado. Al acabar el proceso, la pila contiene un solo número, que es el valor de la expresión. Representa la expresión dada como secuencia de caracteres, y supón disponible una función *valor* que asocie a cada operando su valor numérico.

228. Dado un número natural N ≥ 2, se llaman *números afortunados* a los que resultan de ejecutar el siguiente proceso: se comienza generando una *cola* que contiene los números desde 1 hasta *N*, en este orden; se elimina de la cola un número de cada 2 (es decir, los números 1, 3, 5, etc.); de la nueva cola, se elimina ahora un número de cada 3; etc. El proceso termina cuando se va a eliminar un número de cada *m* y el tamaño de la cola es menor que *m*. Los números que queden en la cola en este momento son los afortunados. Diseña un procedi-

miento que reciba *N* como parámetro y produzca una secuencia formada por los números afortunados resultantes.

(*Indicación*: para eliminar de una cola de *n* números un número de cada *m*, hay que reiterar *n* veces el siguiente proceso: extraer el primer número de la cola, y añadirlo al final de la misma, salvo si le tocaba ser eliminado.)

Más aplicaciones de los tipos de datos con estructura lineal

229. Estudia posibles implementaciones del TAD CJTO (cfr. ejercicios 154 y 156) usando listas como representantes de los conjuntos. Debes suponer que las listas se importan de un módulo separado, sin tener acceso a su representación interna. Compara con las implementaciones de conjuntos planteadas en el ejercicio 167.

230. Estudia una implementación del TAD de los polinomios (cfr. ejercicio 151) usando listas ordenadas (importadas de un módulo separado) como representantes de polinomios. Discute las ventajas e inconvenientes con respecto a las implementaciones basadas en vectores que se plantearon en el ejercicio 170.

231. Severino del Pino, profesor de arameo de la Universidad Imponente, ha detectado problemas de aburrimiento entre su numeroso alumnado. Su colega Tadeo de la Tecla, del departamento de informática, ha ofrecido ayudarle diseñando un sistema informático de control de bostezos. Tadeo propone una especificación de un TAD parametrizado BOSTEZOS[E :: ORD], donde el parámetro E nos da un tipo *Elem* equipado con operaciones de igualdad y orden, que representa a los alumnos. Tadeo propone que BOSTEZOS disponga de las siguientes operaciones:

- `Crea:` → `Bostezos[Elem]`
  Generadora. Crea un sistema de bostezos vacío.
- `Otro: (Elem, Bostezos[Elem])` → `Bostezos[Elem]`
  Generadora. Registra un nuevo bostezo en el sistema.
- `borra: (Elem, Bostezos[Elem])` → `Bostezos[Elem]`
  Modificadora. Borra del sistema todos los bostezos registrados de un elemento dado.
- `cuántos?: (Elem, Bostezos[Elem])` → `Nat`
  Observadora. Consulta el número de bostezos de un elemento dado que estén registrados en el sistema.
- `listaNegra: Bostezos[Elem]` → `Sec[Elem]`
  Observadora. Devuelve la secuencia ordenada de todos los elementos que tengan tres o más bostezos registrados.

Formaliza la especificación algebraica del TAD BOSTEZOS y estudia dos implementaciones alternativas: Una basada en vectores ordenados, y otra basada en secuencias ordenadas, importadas de un módulo separado.

232. En este ejercicio se trata de desarrollar un TAD CONSULTORIO que modelice el comportamiento de un *consultorio médico*. La especificación de CONSULTORIO usará (entre otros) los TADs MEDICO (con tipo principal *Médico*) y PACIENTE (con tipo principal *Paciente*), que se suponen ya conocidos. Suponemos además que MEDICO pertenece a la

clase de tipos ORD. Se desea que CONSULTORIO ofrezca a sus usuarios un tipo principal *Consultorio* junto con las operaciones que se describen informalmente a continuación:

- `Crea: → Consultorio`
  Genera un consultorio vacío sin ninguna información.
- `NuevoMédico: (Consultorio, Médico) - → Consultorio`

Altera un consultorio, dando de alta a un nuevo médico que antes no figuraba en el consultorio.

- `PideConsulta: (Consultorio, Médico, Paciente) - → Consultorio`
  Altera un consultorio, haciendo que un paciente se ponga a la espera para ser atendido por un médico, el cual debe estar de alta en el consultorio.
- `siguientePaciente: (Consultorio, Médico) - → Paciente`
  Consulta el paciente a quien le toca el turno para ser atendido por un médico; éste debe estar dado de alta, y debe tener algún paciente que le haya pedido consulta.
- `atiendeConsulta: (Consultorio, Médico) - → Consultorio`

Modifica un consultorio, eliminando el paciente al que le toque el turno para ser atendido por un médico; éste debe estar dado de alta, y debe tener algún paciente que le haya pedido consulta.

- `tienePacientes: (Consultorio, Médico) - → Bool`

Reconoce si hay o no pacientes a la espera de ser atendidos por un médico, el cual debe estar de alta.

(a) Construye una especificación algebraica completa del TAD CONSULTORIO, incluyendo todas las ecuaciones necesarias.

(b) Plantea un módulo de implementación CONSULTORIO del TAD CONSULTORIO, indicando claramente:

- Los módulos que se importan
- La definición del tipo representante de *Consultorio*
- El invariante de la representación y la función de abstracción para *Consultorio*
- Las cabeceras, pre- y postcondiciones de los procedimientos y funciones que se exportan.

Elige el tipo representante de *Consultorio* de manera que puedas realizarlo utilizando TADs conocidos con estructura lineal, importándolos de módulos que los implementen, y sin acceder a la representación interna.

(c) Completa el módulo de implementación del apartado (b) desarrollando procedimientos y funciones que implementen correctamente las operaciones de CONSULTORIO, y analizando el tiempo de ejecución de cada una de ellas en el caso peor, con respecto a las siguientes medidas del tamaño de un consultorio: *M*, número de médicos dados de alta en el consultorio; y *P*, máximo de los tamaños de las colas de espera de los diferentes médicos.

233. En este ejercicio se trata de desarrollar un TAD MERCADO que modelice el comportamiento de un *mercado de trabajo* simplificado, donde las personas pueden ser contratadas y despedidas por empresas. La especificación de MERCADO usará (entre otros) los TADs PERSONA (con tipo principal *Persona*) y EMPRESA (con tipo principal *Empresa*), que se suponen ya conocidos. Suponemos además que PERSONA y EMPRESA pertenecen a la

clase de tipos ORD. Se desea que MERCADO ofrezca a sus usuarios un tipo principal *Mercado* junto con las operaciones que se describen informalmente a continuación:

– `Crea`

Genera un mercado vacío, sin ninguna información.

– `Contrata`

Altera un mercado, efectuando la contratación de cierta persona como empleado de cierta empresa.

– `despide`

Altera un mercado, efectuando el despido de cierta persona que era antes empleado de cierta empresa.

– `empleados`

Consulta los empleados de una empresa, devolviendo el resultado como secuencia ordenada de personas.

– `empleado?`

Averigua si es cierto o no que una persona dada es empleado de una empresa dada.

– `pluriempleado?`

Averigua si es cierto o no que una persona es empleado de más de una empresa.

`(a)` Construye una especificación algebraica del TAD MERCADO, indicando los otros TADs que se necesite usar, los perfiles de las operaciones, la clasificación de las operaciones en generadoras, modificadoras y observadoras, y los dominios de definición de las operaciones parciales (si las hay).

`(b)` Plantea un módulo de implementación para el TAD MERCADO, del modo indicado en el enunciado del ejercicio anterior. Elige el tipo representante de *Mercado* de manera que puedas realizarlo utilizando TADs conocidos con estructura lineal, importándolos de módulos que los implementen, y sin acceder a la representación interna.

`(c)` Completa el módulo de implementación del apartado (b) desarrollando los procedimientos y funciones que implementan las operaciones de MERCADO, y analizando sus tiempos de ejecución en función de las siguientes medidas del tamaño de un mercado: *NE*, el número de empresas; y *NP*, el número máximo de personas contratadas por una misma empresa.

234. Especifica un TAD BANCO adecuado para modelizar el comportamiento de un banco simplificado, incluyendo operaciones que sirvan para crear un banco vacío (i.e., sin ninguna cuenta), abrir y cerrar cuentas, efectuar ingresos y extracciones en una cuenta, preguntar si un cliente tiene cuenta, consultar el saldo de una cuenta, etc. Desarrolla una implementación de este TAD, usando listas o secuencias, importadas de un módulo separado, para construir la representación de los bancos.

235. Supongamos disponible un módulo que implemente el TAD BANCO del ejercicio anterior. Diseña un procedimiento que procese una *cola de solicitudes de servicios* de clientes de un banco, dando como resultados un nuevo estado del banco y *una secuencia de incidencias*. Cada solicitud debe representar una petición de *ingreso* o *extracción* de una cantidad por parte de un cliente, y cada *incidencia* debe representar un cliente que ha solicitado realizar una operación indefinida. Además del módulo de datos BANCO, debes suponer disponibles otros dos módulos SOLICITUDES e INCIDENCIAS, que exporten los tipos *Solicitud* e *Incidencia*, respectivamente, junto con operaciones adecuadas. Especifica los TADs correspondientes al comportamiento abstracto de estos dos módulos, e indica claramente qué otros módulos de datos necesita utilizar el procedimiento que diseñes.

236. En este ejercicio se trata de automatizar algunos aspectos de la gestión de una biblioteca simplificada. Suponemos que la biblioteca dispone de uno o más ejemplares de cada libro. Un *libro* debe tener *autor* y *título*, mientras que un *ejemplar* debe tener además un *número de registro*. Los ejemplares se registran en la biblioteca al ser adquiridos, y posteriormente pueden ser prestados a *usuarios*. Por consiguiente, la *biblioteca* debe mantener la información de los ejemplares existentes y del *estado* de cada uno, que puede consistir en estar disponible o prestado a un determinado usuario. Especifica un TAD BIBLIOTECA que modelice el comportamiento de la biblioteca, proporcionando operaciones adecuadas para crear una biblioteca vacía, registrar un nuevo ejemplar, efectuar préstamos y devoluciones, generar un listado de autores disponibles, generar un listado de títulos disponibles para un autor dado, etc. Plantea una implementación de BIBLIOTECA, usando listas o secuencias importadas de módulos separados para construir la representación de las bibliotecas.

NOTA: Una representación razonable de una biblioteca puede ser una lista de listas, donde cada una de las lista-elemento contenga todos los ejemplares disponibles de un mismo autor (cada uno acompañado por la información de si está prestado o no, y en caso afirmativo, a qué usuario). Conviene que el invariante de la representación exija mantener la lista que representa la biblioteca ordenada por autores, y la lista de ejemplares de cada autor ordenada por títulos, con los diferentes ejemplares de un mismo título ordenados a su vez por número de registro.

237. Supongamos disponible un módulo que implemente el TAD BIBLIOTECA del ejercicio anterior. Diseña un procedimiento que procese una *cola de solicitudes de préstamo* de usuarios de una biblioteca, dando como resultados un nuevo estado de la biblioteca y una *secuencia de incidencias*. Cada solicitud debe identificar a un usuario y a un libro solicitado en préstamo por éste, y cada incidencia debe identificar una operación de préstamo que no ha podido ejecutarse, indicando el usuario solicitante y el motivo de la incidencia (libro no existente, ejemplares no disponibles, etc.). Además del módulo de datos BIBLIOTECA, debes suponer disponibles otros dos módulos SOLICITUDES e INCIDENCIAS, lo mismo que en el ejercicio 235.

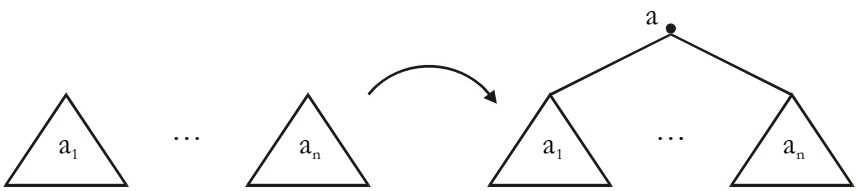
CAPÍTULO 5
ÁRBOLES

5.1 Modelo matemático y especificación

Los árboles son estructuras jerárquicas formadas por *nodos*, de acuerdo con la siguiente construcción inductiva:

— Un solo nodo forma un árbol *a*; se dice que el nodo es *raíz* del árbol.

— Dados *n* árboles ya construidos a_1, … , a_n, se puede construir un nuevo árbol *a* añadiendo un nuevo nodo como raíz y conectándolo con las raíces de los a_i. Se dice que los a_i son los *hijos* de *a*.



Ejemplos de uso de los árboles

Los árboles tienen muchos usos para representar jerarquías y clasificaciones, dentro y fuera de la Informática:

— Arboles genealógicos

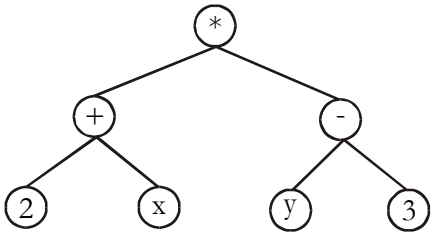
— Arboles taxonómicos

— Organización de un libro en capítulos, secciones, etc.

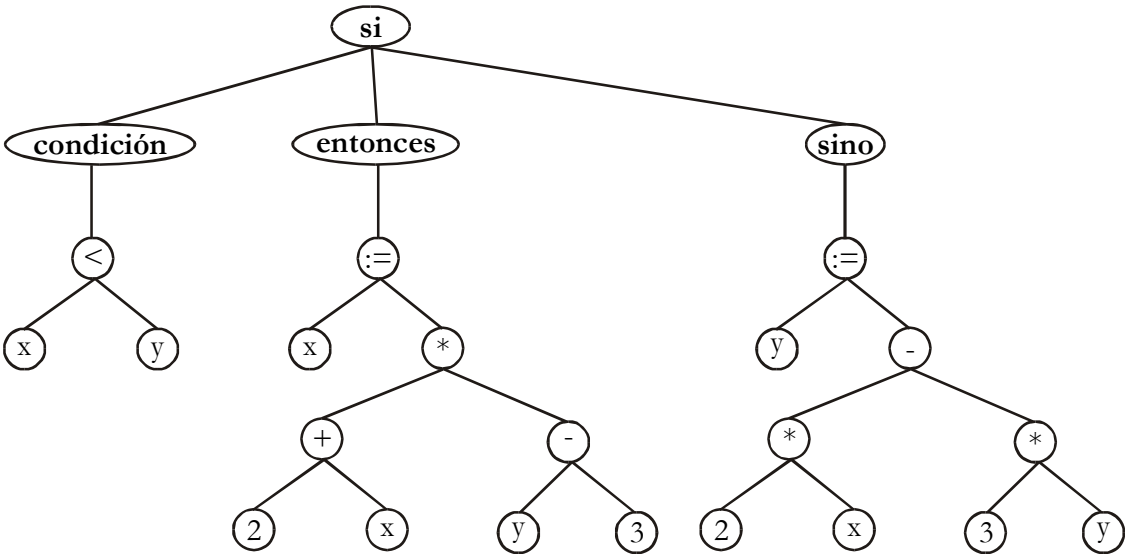
— Estructura de directorios y archivos de una computadora.

— Arbol de llamadas de una función recursiva.

— Arboles de análisis, por ejemplo para una expresión aritmética:



o para una acción condicional



Clases de árboles

— Ordenados o no ordenados. Un árbol es ordenado si el orden de los hijos de cada nodo es relevante. Nosotros vamos a considerar casi siempre árboles ordenados.

— Etiquetados o no etiquetados. Un árbol está etiquetado si hay informaciones asociadas a sus nodos. Nosotros vamos a utilizar árboles etiquetados.

— Generales o n-arios. Un árbol se llama general si no hay una limitación fijada al número de hijos de cada nodo. Si el número de hijos está limitado a un valor fijo *n*, se dice que el árbol es de *grado n*.

— Con o sin punto de interés. En un árbol con punto de interés hay un nodo distinguido (aparte de la raíz, que es distinguida en cualquier árbol). Nosotros vamos a estudiar fundamentalmente árboles sin punto de interés.

5.1.1 Arboles generales

Modelo matemático

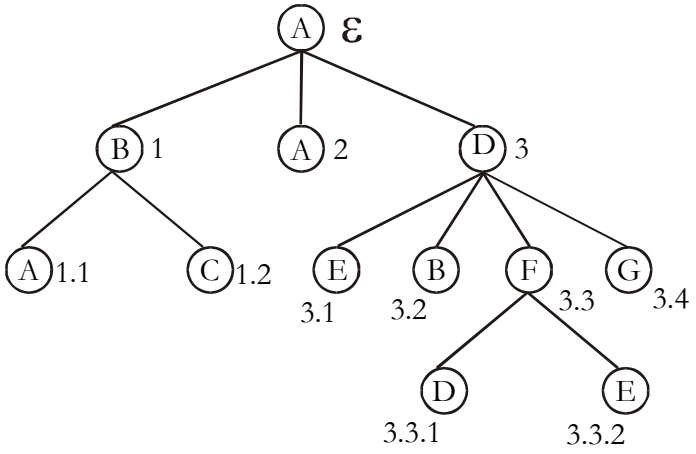
En Matemática discreta se suele definir el concepto de árbol como grafo conexo acíclico con un nodo distinguido como raíz. Sin embargo, esta definición no refleja el orden entre los hijos, ni tampoco las informaciones asociadas a los nodos.

Nosotros vamos a adoptar un modelo de árbol basado en la idea de representar las posiciones de los nodos como cadenas de números naturales positivos:

— La raíz de un árbol tiene como posición la *cadena vacía* ε.

— Si un cierto nodo de un árbol tiene posición $\alpha \in \aleph_+^*$, el hijo número *i* de ese nodo tendrá posición α.i (que indica α seguida de *i*).

Ejemplo:



El árbol puede modelizarse como una aplicación:

$a : N \rightarrow V$

donde $N \subseteq \aleph_+^*$ es el conjunto de posiciones de los nodos, y V es el conjunto de valores posibles para las informaciones que se asocian a los nodos.

Volviendo al ejemplo:

$N = \{ \varepsilon, 1, 2, 3, 1.1, 1.2, 3.1, 3.2, 3.3, 3.4, 3.3.1, 3.3.2 \}$

$a(\varepsilon) =$ 'A'
$a(1) =$ 'B' $\quad a(2) =$ 'C' $\quad a(3) =$ 'D' $\quad$ etc.

En general, un conjunto $N \subseteq \aleph_+^*$ de cadenas de números naturales positivos, debe cumplir las siguientes condiciones para ser válido como conjunto de posiciones de los nodos de un árbol general:

— N es finito.

— $\varepsilon \in N$ posición de la raíz.

— $\alpha.i \in N \Rightarrow \alpha \in N$ cerrado bajo prefijos.

— $\alpha.i \in N \wedge 1 \leq j < i \Rightarrow \alpha.j \in N$ hijos consecutivos sin huecos.

Terminología y conceptos básicos

Dado un árbol $a : N \rightarrow V$

— *Nodo* es cada posición, junto con la información asociada: $(\alpha, a(\alpha))$, siendo $\alpha \in N$.

— *Raíz* es el nodo de posición ε.

Hojas son los nodos de posición α tal que no existe *i* tal que $\alpha.i \in N$.

Nodos internos son los nodos que no son hojas.

— Un nodo $\alpha.i$ tiene como *padre* a α, y se dice que es *hijo* de α.

— Dos nodos de posiciones $\alpha.i$, $\alpha.j$ ($i \neq j$) se llaman *hermanos*.

— *Camino* es una sucesión de nodos tal que cada uno es padre del siguiente:

$$\alpha, \alpha.i_1, \ldots, \alpha.i_1. \ldots .i_n$$

n es la *longitud* del camino.

— *Rama* es cualquier camino que comience en la raíz y termine en una hoja.

— El *nivel* o *profundidad* de un nodo de posición α es $|\alpha|+1$. Es decir: $n+1$, siendo *n* la longitud del camino (único) que va de la raíz al nodo. En particular, el nivel de la raíz es 1. El árbol del ejemplo tiene nodos a 4 niveles. El nivel de un nodo es igual al número de nodos del camino que va desde la raíz al nodo.

— La *talla*, *altura* o *profundidad* de un árbol es el máximo de todos los niveles de nodos del árbol. Equivalentemente: $1+n$, siendo *n* el máximo de las longitudes de las ramas. el árbol del ejemplo es de talla 4.

— El *grado* o *aridad* de un nodo interno es su número de hijos. La aridad de un árbol es el máximo de las aridades de todos sus nodos internos.

— Si hay un camino del nodo de posición α al nodo de posición β (i.e., si α es prefijo de β), se dice que α es *antepasado* de β y que β es *descendiente* de α.

— Cada nodo de un árbol *a* determina un *subárbol* a_0 con raíz en ese nodo. Formalmente, si el nodo tiene posición α, entonces:

$$a_0 : N_0 \rightarrow V$$

siendo

$$N_0 =_{def} \{ \beta \in \aleph_+^* \mid \alpha\beta \in N \}$$
$$a_0(\beta) =_{def} a(\alpha\beta) \quad \text{para cada } \beta \in N_0$$

— Dado un árbol *a*, los subárboles de *a* (si existen), se llaman árboles *hijos* de *a*. El árbol del ejemplo tiene 3 árboles hijos.

Especificación algebraica de los árboles generales

Optamos por incluir operaciones que permiten:

— Construir árboles.

— Consultar la información de la raíz y los árboles hijos.

— Contar el número de hijos.

— Reconocer las hojas.

Para construir un árbol a partir de su raíz y sus hijos, se plantea el problema de que el número de hijos es variable. Es por ello que el TAD, además del tipo principal *Arbol[Elem]*, incluye el tipo *Bosque[Elem]*, que representan sucesiones finitas de árboles. Las operaciones generadoras de bosques siguen la misma idea que las generadoras de listas.

```
tad ARBOL[E :: ANY]
    usa
      BOOL, NAT
    tipos
      Arbol[Elem], Bosque[Elem]
    operaciones
      []: → Bosque[Elem]                                          /* gen */
      [ _ / _ ]: (Arbol[Elem], Bosque[Elem]) → Bosque[Elem]       /* gen */
      árbol: (Nat, Bosque[Elem]) - → Arbol[Elem]                  /* obs */
      nrArboles: Bosque[Elem] → Nat                               /* obs */
      esVacío: Bosque[Elem] → Bool                                /* obs */
      Cons: (Elem, Bosque[Elem]) → Arbol[Elem]                    /* gen */
      hijo: (Nat, Arbol[Elem]) - → Arbol[Elem]                    /* mod */
      nrHijos: Arbol[Elem] → Nat                                  /* obs */
      hijos: Arbol[Elem] → Bosque[Elem]                           /* obs */
      raíz: Arbol[Elem] → Elem                                    /* obs */
      esHoja: Arbol[Elem] → Bool                                  /* obs */
    ecuaciones
      ∀ i : Nat : ∀ as : Bosque[Elem] : ∀ a : Arbol[Elem] : ∀ x : Elem :
      def árbol(i, as) si Suc(Cero) ≤ i ≤ nrArboles(as)
      árbol(i, [a/as])       = a  si i = Suc(Cero)
      árbol(i, [a/as])       =f árbol(i-1, as)si i > Suc(Cero)
      nrArboles([])          = Cero
      nrArboles([a/as])      = Suc(nrArboles(as))
      esVacío(as)            = nrArboles(as) == Cero
      def hijo(i, a) si Suc(Cero) ≤ i ≤ nrHijos(a)
      hijo(i, a)          =f árbol(i, hijos(a))
      nrHijos(Cons(x, as)) = nrArboles(as)
      hijos(Cons(x, as))   = as
      raíz(Cons(x, as))      = x
      esHoja(a)           = nrHijos(a) == 0
    errores
      ∀ i : Nat : ∀ as : Bosque[Elem] : ∀ a : Arbol[Elem] :
      árbol(i, as) si NOT(Suc(Cero) ≤ i ≤ nrArboles(as))
      hijo(i, a)   si NOT(Suc(Cero) ≤ i ≤ nrHijos(a))
ftad
```

5.1.2 Arboles binarios

Modelo matemático

Los árboles binarios se definen de tal modo que cada nodo interno tiene como máximo dos hijos. Las posiciones de los nodos de estos árboles pueden representarse como cadenas $\alpha \in \{1,2\}^*$. En caso de que un nodo interno (con posición α) tenga un solo hijo, se distingue si éste es *hijo izquierdo* (con posición $\alpha.1$) o *hijo derecho* (con posición $\alpha.2$).

Estas ideas conducen a modelar un árbol binario etiquetado con informaciones de V como una aplicación:

```
a : N → V
```

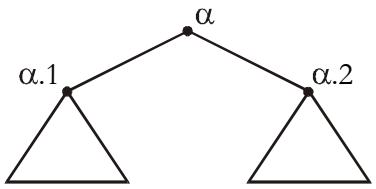
donde el conjunto N de posiciones de los nodos de *a* debe cumplir las siguientes condiciones:

— $N \subseteq \{1,2\}^*$, finito.

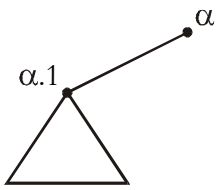
— $\alpha.i \in N \Rightarrow \alpha \in N$ cerrado bajo prefijos.

— $\alpha \in N \Rightarrow$ se dan uno de los 4 casos siguientes:

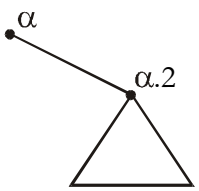
  — $\alpha.1 \in N$ y $\alpha.2 \in N$ dos hijos



  — $\alpha.1 \in N$ y $\alpha.2 \notin N$ sólo hijo izquierdo



  — $\alpha.1 \notin N$ y $\alpha.2 \in N$ sólo hijo derecho



  — $\alpha.1 \notin N$ y $\alpha.2 \notin N$ ningún hijo; hoja

Vemos que en los árboles binarios no se exige que "no haya huecos" en la serie de hijos de un nodo. Por esto, algunos árboles binarios pueden no ser válidos como árboles generales.

Cuando falta alguno de los hijos de un nodo interno, se dice también que el hijo inexistente es *vacío*. En particular, podemos decir que las hojas tienen dos hijos vacíos.

Así, aceptamos la idea de *árbol vacío*, cuyo conjunto de posiciones es ∅ (el subconjunto vacío de $\{1, 2\}^*$.

Toda la terminología y conceptos básicos que hemos estudiado para los árboles generales, pueden adaptarse sin dificultad a los árboles binarios.

Especificación algebraica de los árboles binarios

Gracias al concepto de *árbol vacío*, podemos suponer que cualquier árbol binario no vacío se construye a partir de un elemento raíz y dos árboles hijos (que pueden a su vez ser o no vacíos). Esto conduce a la signatura del TAD, con operaciones para:

— Generar el árbol vacío.

— Construir árboles no vacíos.

— Consultar la raíz y los hijos.

— Recorrer el árbol vacío.

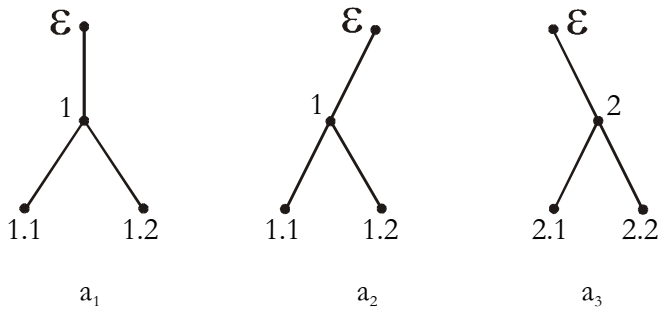
```
tad ARBIN[E :: ANY]
    usa
      BOOL
    tipos
      Arbin[Elem]
    operaciones
      Vacío: → Arbin[Elem]  /* gen */
      Cons: (Arbin[Elem], Elem, Arbin[Elem]) → Arbin[Elem] /* gen */
      hijoIz, hijoDr: Arbin[Elem] – → Arbin[Elem]/* mod */
      raíz: Arbin[Elem] – → Elem  /* obs */
      esVacío: Arbin[Elem] → Bool /* obs */
    ecuaciones
      ∀ iz, dr : Arbin[Elem] : ∀ x : Elem :
      def hijoIz(Cons(iz, x, dr))
      hijoIz(Cons(iz, x, dr))   = iz
      def hijoDr(Cons(iz, x, dr))
      hijoDr(Cons(iz, x, dr))   = dr
      def raíz(Cons(iz, x, dr))
      raíz(Cons(iz, x, dr))  = x
      esVacío(Vacío)          = cierto
      esVacío(Cons(iz, x, dr))  = falso
    errores
      hijoIz(Vacío)
      hijoDr(Vacío)
      raíz(Vacío)
ftad
```

5.1.3 Arboles n-arios

Se definen como generalización de los árboles binarios, de tal manera que cada nodo interno tiene exactamente *n* hijos ordenados, cualquiera de los cuales puede ser vacío (i.e., se admiten "huecos" en la sucesión de hijos de un nodo). En particular, las hojas tienen *n* hijos vacíos.

Observemos que "árbol n-ario" no es lo mismo que "árbol de grado *n*", de acuerdo con la definición de *grado* o *aridad* de un árbol que dimos anteriormente. En un árbol de grado *n*, cada nodo tiene como mucho *n* hijos, pero distintos nodos pueden tener distinto número de hijos, y los hijos existentes ocupan posiciones consecutivas.

Ejemplo. a_1 es un árbol general de grado 2, pero no es un árbol binario. Los árboles a_2 y a_3 sí son binarios:



a_1 a_2 a_3

5.1.4 Arboles con punto de interés

Como representación matemática de un árbol con punto de interés podemos adoptar una pareja de la forma:

$$(a, \alpha_0)$$

donde *a* es un árbol, modelizado matemáticamente del modo que ya hemos indicado:

$$a : N \rightarrow V$$
$$N \subseteq \aleph_+^*$$

y $\alpha_0 \in N$ indica la posición del punto de interés.

Para especificar TADs basados en árboles con punto de interés hay que considerar algunas operaciones que saquen partido del punto de interés. hay varias posibilidades:

— Modificarla información asociada al punto de interés.

— Insertar un nuevo nodo, o todo un árbol, como hijo del punto de interés.

— Desplazar el punto de interés (e.j., al padre, al hermano izquierdo, al hermano derecho.).

5.2 Técnicas de implementación

Vamos a centrarnos en la implementación de los árboles binarios, y trataremos más brevemente el caso de los árboles generales.

5.2.1 Implementación dinámica de los árboles binarios

Tipo representante

```
tipo
  Nodo = reg
    ra : Elem;
    iz, dr : Enlace;
  freg;
  Enlace = puntero a Nodo;
  Arbin[Elem] = Enlace;
```

Invariante de la representación

Sea *p : Enlace*. Consideramos dos posibilidades:

— Representación que prohibe ciclos, pero no nodos compartidos:

$$
\begin{aligned}
& R(p) \\
\Leftrightarrow_{def} \\
& p = nil \vee \\
& (p \neq nil \wedge ubicado(p) \wedge R(p^{\wedge}.elem) \wedge \\
& R(p^{\wedge}.iz) \wedge R(p^{\wedge}.dr) \wedge \\
& p \notin enlaces(p^{\wedge}.iz) \cup enlaces(p^{\wedge}.der))
\end{aligned}
$$

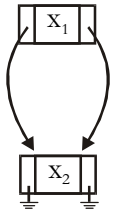
siendo

$$
enlaces(p) =_{def} \begin{cases} \varnothing & \textbf{si } p = nil \\ \{p\} \cup enlaces(p^{\wedge}.iz) \cup enlaces(p^{\wedge}.dr) & \textbf{si } p \neq nil \end{cases}
$$

— Representación que prohibe ciclos y nodos compartidos

$$
\begin{aligned}
& R^{NC}(p) \\
\Leftrightarrow_{def} \\
& p = nil \vee \\
& (p \neq nil \wedge ubicado(p) \wedge R(p^{\wedge}.elem) \wedge \\
& R^{NC}(p^{\wedge}.iz) \wedge R^{NC}(p^{\wedge}.dr) \wedge \\
& p \notin enlaces(p^{\wedge}.iz) \cup enlaces(p^{\wedge}.der) \wedge \\
& enlaces(p^{\wedge}.iz) \cap enlaces(p^{\wedge}.dr) = \varnothing)
\end{aligned}
$$

Esta representación prohibe situaciones como:



Función de abstracción

Sea *a : Arbin[Elem]* tal que *R*(a)

$$A(a) =_{def} \begin{cases} \texttt{Vacío} & \textbf{si } a = nil \\ \texttt{Cons(A(a^.iz), A(a^.elem), A(a^.dr))} & \textbf{si } a \neq nil \end{cases}$$

Implementación de las operaciones

Debido a la forma de las generadoras de este TAD, resulta natural realizar todas las operaciones como funciones:

```
func Vacío ( ) dev a : Arbin[Elem];   /* O(1) */
{ P0 : cierto }
inicio
  a := nil
{ Q0 : R(a) ∧ A =ARBIN[ELEM] Vacío }
  dev a
ffunc

func Cons( iz : Arbin[Elem]; x : Elem; dr : Arbin[Elem] ) dev a :
Arbin[Elem];
{ P0 : R(iz) ∧ R(x) ∧ R(dr) }    /* O(1) */
inicio
  ubicar(a);
  a.ra := x;
  a.iz := iz;
  a.dr := dr
{ Q0 . R(a) ∧ A(a) =ARBIN[ELEM] Cons(A(iz), A(x), A(dr)) }
  dev a
ffunc
```

Nótese que esta implementación no garantiza $R^{NC}(a)$, ya que, por ejemplo:

```
a := Cons( a1, x, a1 )
```

```
func hijoIz( a : Arbin[Elem] ) dev iz : Arbin[Elem];    /* O(1) */
{ P₀ : R(a) ∧ NOT esVacío(A(a)) }
inicio
  si esVacío(a)
    entonces
      error("El árbol vacío no tiene hijo izquierdo")
    sino
      iz := a^.iz
  fsi
{ Q₀ : R(iz) ∧ A(iz) =ARBIN[ELEM] hijoIz( A(a) ) }
  dev iz
ffunc
```

La función *hijoDr* es dual a ésta. También $O(1)$.

```
func raíz( a : Arbin[Elem] ) dev x : Elem;    /* O(1) */
{ P₀ : R(a) ∧ NOT esVacío(A(a)) }
inicio
  si esVacío(a)
    entonces
      error("El árbol vacío no tiene raíz")
    sino
      x := a^.ra
  fsi
{ Q₀ : R(x) ∧ A(x) =ARBIN[ELEM] raíz(A(a)) }
  dev x
ffunc

func esVacío ( a : Arbin[Elem] ) dev r : Bool;
{ P₀ : R(a) }
inicio
  r := a == nil
{ Q₀ : A(r) =ARBIN[ELEM] esVacío(A(a)) }
  dev r
ffunc
```

Compartición de estructura

Como ya hemos comentado, esta implementación de los árboles da lugar a compartición de estructura. El problema, ya conocido, de la compartición de estructura está en que al liberar el espacio ocupado por una variable puede destruirse el valor de otra. La solución ideal radica en que el programador no tenga que preocuparse por la anulación de las estructuras, sino que sea el sistema de gestión de memoria quien se encargue de ello: *recolección automática de basura.*

Con esta implementación, el cliente del TAD debe indicar expresamente cuándo quiere evitar la compartición de estructura, realizando copias de los parámetros:

```
a := Cons( copia(a1), x, copia(a2) )
```

Si no se dispone de recolección automática de basura, entonces si el cliente crea estructuras compartidas, es responsabilidad suya evitar que se libere el espacio de una estructura mientras haya alguna variable activa que tenga acceso a dicho espacio.

En resumen, el TAD deberá exportar las operaciones *copia* y *anula*.

```
proc anula ( es a : Arbin[Elem] );
{ P₀ : R^NC(a) ∧ a = A }
var
  aux : Arbin[Elem];
inicio
  si a == nil
    entonces
      seguir
    sino
      aux := a^.iz;
      anula(aux);
      aux := a^.dr;
      anula(aux);
      % ELEM.anula(a^.ra)       si la anulación es profunda
       liberar(a);
      a := nil
  fsi
{ Q₀ : a = nil ∧
      el espacio que ocupaba la estructura representante de A se ha liberado
}
fproc
func copia ( a : Arbin[Elem] ) dev b : Arbin[Elem];
{ P₀ : R(a) }
var
  iz, dr : Arbin[Elem];
inicio
  si a == nil
    entonces
      b := nil
    sino
      iz := copia(a^.iz);
      dr := copia(a^.dr);
      ubicar(b);
      b^.ra := a^.ra;    % ELEM.copia(a^.ra)    si la copia es profunda
      b^.iz := iz;
      b^.dr := dr
  fsi
{ Q₀ : R^NC(b) ∧ A(a) = A(b) ∧
      la estructura representante de b está ubicado en espacio nuevo }
  dev b
ffunc
```

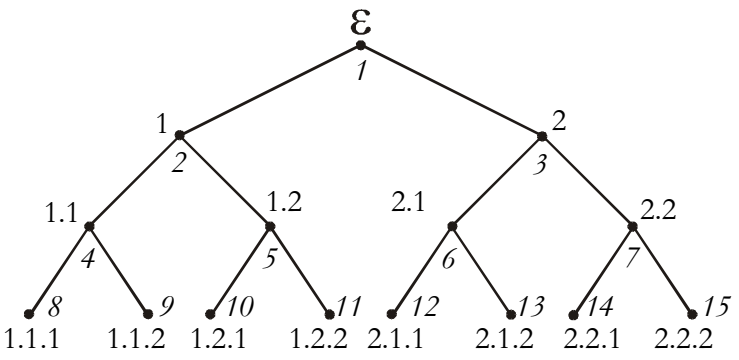
También podría venir bien que el módulo de los árboles exportase una operación de igualdad.

5.2.2 Implementación estática encadenada de los árboles binarios

Esta implementación se basa en simular la memoria dinámica con un vector, de forma que los punteros sean índices dentro de dicho vector. Esta implementación se describe en la subsección 5.2.1, pp. 229-232, del libro de Franch.

5.2.3 Representación estática secuencial para árboles binarios semicompletos

En este apartado estudiamos una técnica válida para representar un árbol binario en un vector sin ayuda de encadenamientos. La idea consiste en calcular, en función de la posición de cada nodo del árbol, el índice del vector donde vamos a almacenar la información asociada a ese nodo. Para hacer esto necesitamos establecer una biyección entre posiciones y números positivos. Podemos conseguir una numeración de las posiciones por niveles, y de izquierda a derecha dentro de cada nivel. Si comenzamos asociando el número 1 a la posición ε, resulta:



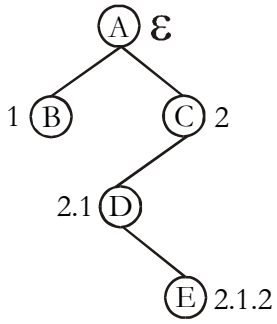
Esta numeración de posiciones corresponde a una biyección

$$\text{índice}: \{1, 2\}^* \rightarrow \aleph_+$$

que admite la siguiente definición recursiva:

$$\begin{aligned} &\text{índice}(\ \varepsilon\ ) &&= 1 \\ &\text{índice}(\ \alpha.1\ ) &&= 2 * \text{índice}(\alpha) \\ &\text{índice}(\ \alpha.2\ ) &&= 2 * \text{índice}(\alpha) + 1 \end{aligned}$$

Con ayuda de *índice*, podemos representar un árbol binario en un vector, almacenando la información del nodo α en la posición *índice*(α). Por ejemplo:



donde tenemos

```
índice(ε) = 1
índice(1) = 2 · índice(ε) = 2
índice(2) = 2 · índice(ε) + 1 = 3
índice(2.1) = 2 · índice(2) = 6
índice(2.1.2) = 2 · índice(2.1) + 1 = 13
```

que daría lugar al vector:

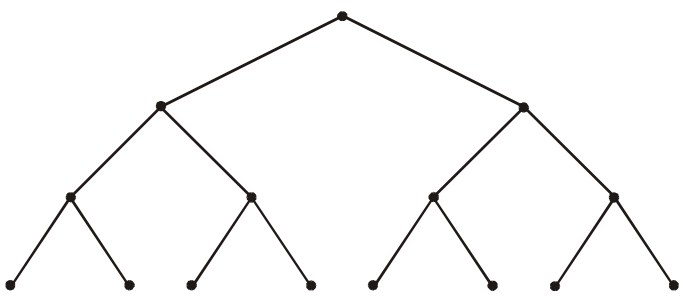
A	B	C			D							E		⋯
1	2	3	4	5	6	7	8	9	10	11	12	13	14	

Como vemos, pueden quedar muchos espacios desocupados si el árbol tiene pocos nodos en relación con su número de niveles. Por este motivo, esta representación sólo se suele aplicar a una clase especial de árboles binarios, que definimos a continuación.

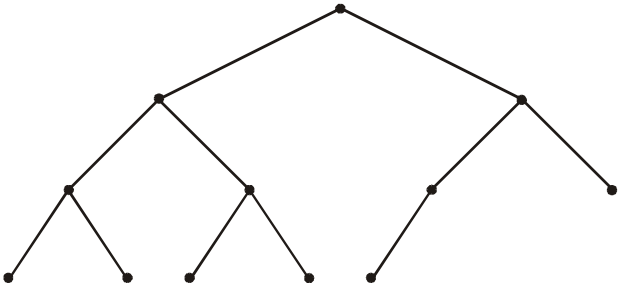
— Un árbol binario de talla *n* se llama *completo* si y sólo si todos sus nodos internos tienen dos hijos no vacíos, y todas sus hojas están en el nivel *n*.

— Un árbol binario de talla *n* se llama *semicompleto* si y sólo si es completo o tiene vacantes una serie de posiciones consecutivas del nivel *n*, de manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.

Por ejemplo, el siguiente es un árbol completo



Y este es semicompleto:



Un árbol binario completo de talla *n* tiene el máximo número de nodos que puede tener un árbol de esa talla. En concreto se verifica:

1. El número de nodos de cualquier nivel *i* en un árbol binario completo es $m_i = 2^{i-1}$
2. El número total de nodos de un árbol binario completo de talla *n* es $M_n = 2^n - 1$. Y, por lo tanto, la talla de un árbol binario completo con *M* nodos es *log(M+1)*.

Como se puede demostrar fácilmente:

1. Inducción sobre *i* :

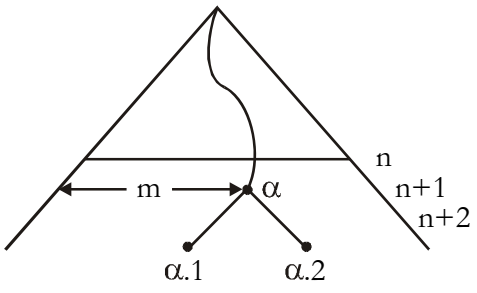
$$i = 1 \quad m_1 = 1 = 2^{1-1}$$
$$i > 1 \quad m_i = 2 \cdot m_{i-1} =_{H.I.} 2 \cdot 2^{i-2} = 2^{i-1}$$

2. Aplicando el resultado de la suma de una progresión geométrica:

$$M_n = \sum_{i=1}^{n} m_i = \sum_{i=1}^{n} 2^{i-1} = 2^n - 1$$

Usando estos dos resultados podemos demostrar que la definición recursiva de *índice* presentada anteriormente es correcta. La definición no recursiva es como sigue: si α es un posición de nivel *n+1* ($n \geq 0$)

índice(α) = número de posiciones de niveles 1..n +
número de posiciones de nivel n+1 hasta α inclusive
= $(2^n - 1) + m$



índice(ε) = 1

índice(α.1) = número de posiciones de niveles 1..(n+1) +
número de posiciones de nivel n+2 hasta α.1 inclusive
= $(2^{n+1} - 1) + 2 \cdot (m - 1) + 1 = 2^{n+1} + 2m - 2$
= 2 · índice(α)

índice(α.2) = número de posiciones de niveles 1..(n+1) +
número de posiciones de nivel n+2 hasta α.2 inclusive
= $(2^{n+1} - 1) + 2 \cdot (m - 1) + 2 = 2^{n+1} + 2m - 1$
= 2 · índice(α) + 1

Fórmulas para la representación indexada de árboles binarios

Recapitulando lo expuesto hasta ahora, tenemos que:

— Un árbol binario completo de talla *n* tiene $2^n - 1$ nodos.

— Por tanto, declarando

```
const
  max = 2^n - 1;
tipo
  esp = Vector[1..max] de Elem;
```

tendremos un vector donde podemos almacenar cualquier árbol binario de talla ≤ *n*

— Dado un nodo α almacenado en la posición *índice(α) = i*, tal que $1 \le i \le max$, valen las siguientes fórmulas para el cálculo de otro nodos relacionados con *i*:

  — Padre: *i div 2*, si $i > 1$

  — Hijo izquierdo: *2i*, si $2i \le max$

  — Hijo derecho: *2i + 1*, si $2i+1 \le max$

— Esta representación es útil para árboles completos y semicompletos, cuando estos se generan y procesan mediante operaciones que hagan crecer al árbol por niveles –habría que cambiar la especificación del TAD–.

— No es una representación útil para implementar árboles binarios cualesquiera, con las operaciones del TAD ARBIN[ELEM].

5.2.4 Implementación de los árboles generales

Se consigue con modificaciones sencillas de las técnicas que hemos estudiado para los árboles binarios.

Implementación dinámica

La idea consiste en representar los nodos como registros con tres campos:

— Información asociada.

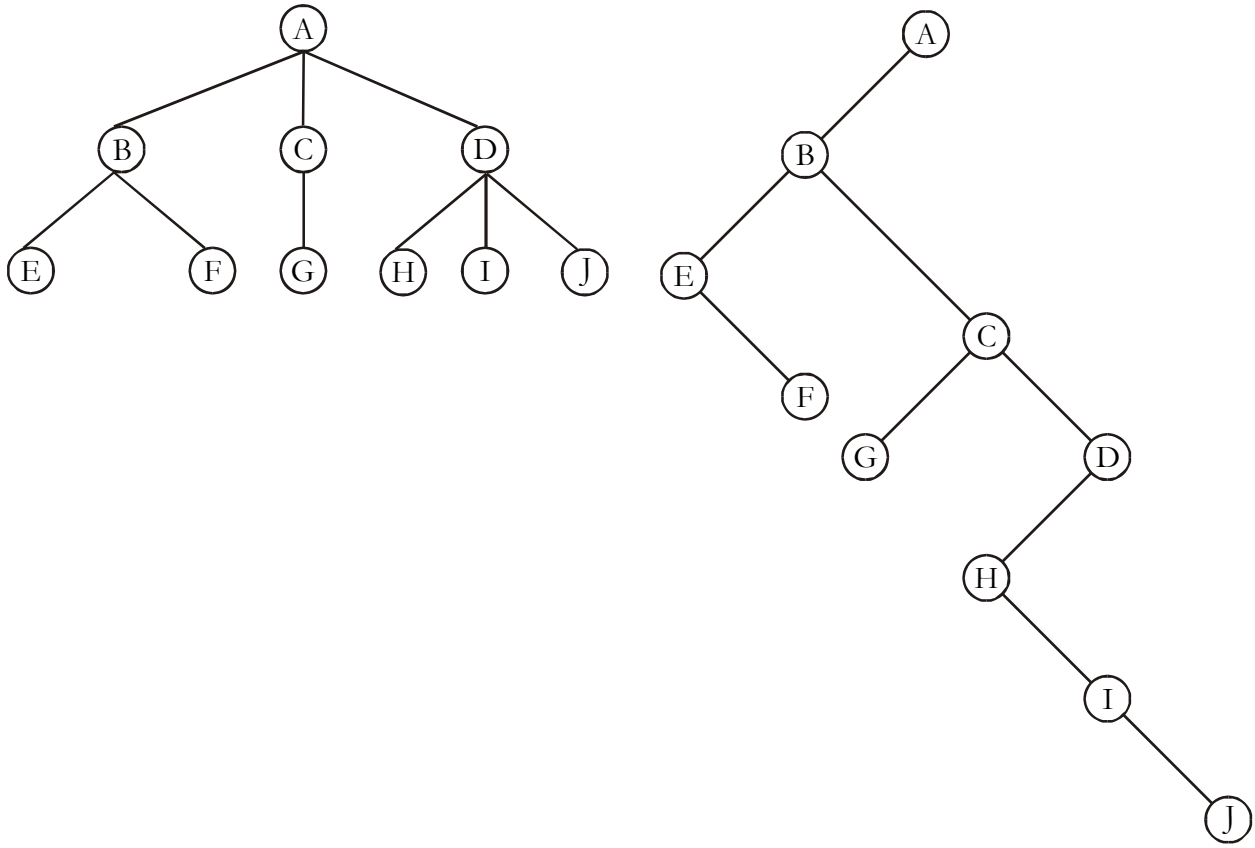
— Puntero al primer hijo.

— Puntero al hermano derecho

En realidad, esta idea se basa en la posibilidad de representa run árbol general cualquiera como árbol binario, a través de la siguiente conversión:

ARBOL GENERAL	ARBOL BINARIO
Primer hijo	Hijo izquierdo
Hermano derecho	Hijo derecho

Por ejemplo, el árbol general de la izquierda se representa como el árbol binario de la derecha:



La idea de esta transformación se puede formalizar especificando las operaciones:

```
hazArbin: Bosque[Elem] → Arbin[Elem]
hazBosque: Arbin[Elem] → Bosque[Elem]
```

de manera que resulten dos biyecciones inversas una de la otra. La especificación ecuacional de estas operaciones:

```
hazArbin([]) = ARBIN.Vacío
```

```
hazArbin([ARBOL.Cons(x,hs) / as]) = ARBIN.Cons( hazArbin(hs), x,
hazArbin(as) )

hazBosque(ARBIN.Vacío) = []
hazBosque(ARBIN.Cons(iz, x, dr)) = [ARBOL.Cons(x, hazBosque(iz)) /
hazBosque(dr)]
```

Se puede demostrar por inducción estructural que efectivamente una operación es la inversa de la otra:

```
∀ as : Bosque[Elem] : hazBosque(hazArbin(as)) = as
∀ b  : Arbin[Elem]  : hazArbin(hazBosque(b))  = b
```

En el libro de Franch se pueden encontrar más detalles sobre esta implementación.

5.2.5 Implementación de otras variantes de árboles

Arboles n-arios

- Se pueden adaptar todos los métodos conocidos para los árboles binarios.
- En las representaciones dinámicas, se puede optar por usar *n* punteros a los *n* hijos, o un puntero al primero hijo y otro al hermano derecho. ¿Cuál es mejor en términos de espacio?
- La representación estática indexada en un vector se basa en fórmulas de cálculo de índices que generalizan las del caso binario.
  - En un árbol completo de talla *t*
    - el nivel *i* ($1 \le i \le t$) tiene n^{i-1} nodos
    - el árbol completo tiene $max = (n^t - 1) / (n - 1)$ nodos
  - La biyección $índice : \{1 .. n\}^* \rightarrow \aleph_+$ que numera las posiciones, admite la definición recursiva:

    ```
    num(ε) = 1
    num(α.i) = n * num(α) + (i-1)        (1 ≤ i ≤ n)
    ```

  - Dado un nodo α con posición $índice(\alpha)=m$, $1 \le m \le max$, se tiene:
    - Hijo *i* de α: $n*m + i-1$, si $\le max$
    - Padre de α: $m \ div \ n$, si $m > 1$.

Arboles con punto de interés

- Las representaciones dinámicas deberán equiparse con punteros adicionales para permitir una realización eficiente de las operaciones de desplazamiento del punto de interés. en particular, resultará útil un puntero al padre.

5.2.6 Análisis de complejidad espacial para las representaciones de los árboles

— En las representaciones dinámicas hay que contar con el número de campos de enlace.

— Si hay n nodos con k campos de enlace cada uno, la estructura ocupará espacio $(k+x) \cdot n$, suponiendo que x sea el espacio ocupado por un elemento.

— Empleando el esquema "primer hijo-hermano derecho", podemos reducirnos a 2 campos de enlace por nodo, y el espacio ocupado descenderá a $(2+x) \cdot n$.

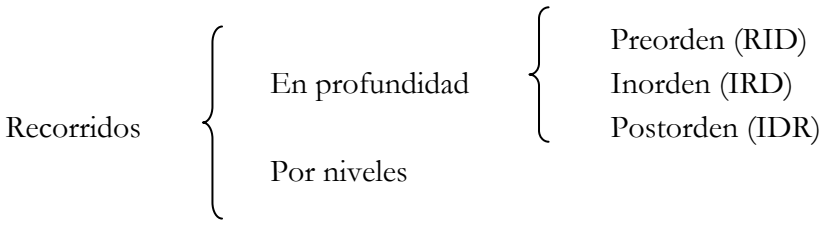
5.3 Recorridos

Recorrer un árbol consiste en visitar todos sus nodos en un cierto nodo, ya sea simplemente para escribirlos, o bien para aplicar a cada uno de ellos un cierto tratamiento.

En este tema vamos a estudiar los recorridos de los árboles binarios, representando un recorrido como una función que transforma un árbol en la lista de elementos de los nodos visitados – en el orden de la visita.

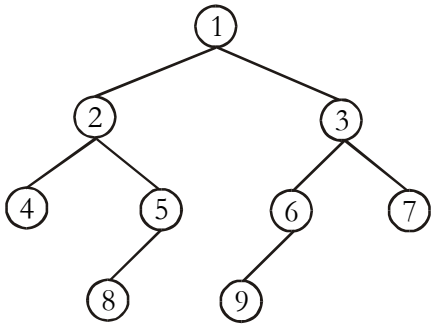
Clases de recorridos

Los principales recorridos de árboles binarios se clasifican como sigue:



Los recorridos en profundidad se basan en la relación padre-hijos y se clasifican según el orden en que se consideren la raíz (R), el hijo izquierdo (I) y el hijo derecho (D).

Ejemplo:



Preorden: 1, 2, 4, 5, 8, 3, 6, 9, 7
Inorden: 4, 2, 8, 5, 1, 9, 6, 3, 7
Postorden: 4, 8, 5, 2, 9, 6, 7, 3, 1
Niveles: 1, 2, 3, 4, 5, 6, 7, 8, 9

Los recorridos de árboles tienen aplicaciones muy diversas:

— Preorden: puede aplicarse al cálculo de atributos heredados en una gramática de atributos (los hijos heredan, y quizá modifican, atributos del padre).

— Inorden: en árboles ordenados –que estudiaremos más adelante– este recorrido produce una lista ordenada.

— Postorden: en árboles que representan expresiones formales, este recurrido corresponde a la evaluación de la expresión, y genera la forma postfija de ésta.

— Niveles: para árboles que representen espacios de búsqueda, este recorrido encuentra caminos de longitud mínima.

5.3.1 Recorridos en profundidad

Especificación algebraica

Según aparecen en las hojas con las especificaciones de los TAD:

```
tad REC-PROF-ARBIN[E :: ANY]
    usa
      ARBIN[E], LISTA[E]
    operaciones
      preOrd, inOrd, postOrd: Arbin[Elem] → Lista[Elem]          /* obs */
    ecuaciones
      ∀ iz, dr : Arbin[Elem] : ∀ x : Elem :
      preOrd(Vacío)               = []
      preOrd(Cons(iz, x, dr))     = [x] ++ preOrd(iz) ++ preOrd(dr)
      inOrd(Vacío)                = []
      inOrd(Cons(iz, x, dr))      = inOrd(iz) ++ [x] ++ inOrd(dr)
      postOrd(Vacío)              = []
      postOrd(Cons(iz, x, dr))    = postOrd(iz) ++ postOrd(dr) ++ [x]
ftad
```

Las operaciones de recorrido no necesitan acceder a la representación interna de los árboles; nótese cómo las hemos especificado como un enriquecimiento del TAD ARBIN.

Implementación recursiva

La implementación recursiva es directa a partir de la especificación. Veamos como ejemplo la implementación del *preOrden*:

```
func preOrd ( a : Arbin[Elem] ) dev xs : Lista[Elem];
{ P₀ : cierto }
var
  iz, dr : Lista[Elem];
inicio
  si ARBIN.esVacío(a)
    entonces
```

```
      LISTA.Nula(xs)
    sino
      iz := preOrd( hijoIz(a) );
      dr := preOrd( hijoDr(a) );
      LISTA.Cons( ARBIN.raíz(a), iz );
      LISTA.conc( iz, dr );
      xs := iz          % no se anulan iz, dr porque comparten estructura
con xs
  fsi
{ Q0 : xs = preOrd(a) }
  dev xs
ffunc
```

En cuanto a la complejidad de esta operación, suponemos una implementación de las listas donde *Nula*, *conc*, *ponDr* y *Cons* tengan coste $O(1)$ –lo que nos obliga a utilizar implementaciones procedimentales de *conc* y *ponDr*–. En ese caso obtenemos una complejidad para los tres recorridos de $O(n)$, como se puede ver aplicando los resultados teóricos. Se trata de funciones donde el tamaño del problema disminuye por división:

$$T(n) = \begin{cases} c_1 & \text{si } 0 \le n < b \\ a \cdot T(n/b) + c \cdot n^k & \text{si } n \ge b \end{cases}$$

$$T(n) \in \begin{cases} O(n^k) & \text{si } a < b^k \\ O(n^k \cdot \log n) & \text{si } a = b^k \\ O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

si suponemos que se trata de un árbol completo, entonces tenemos:

```
a = 2    número de llamadas recursivas
b = 2    si el árbol es completo, el tamaño se divide por 2 en cada llamada
k = 0
```

$a > b^k \Rightarrow$ coste $O(n^{\log_b a}) = O(n)$

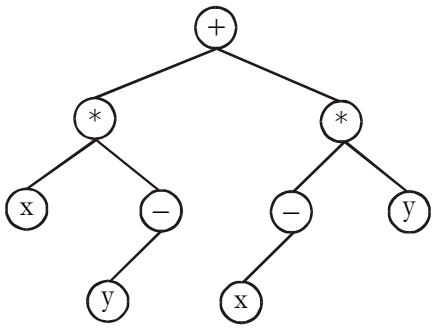
Para que este análisis sea válido ($k = 0$) es preciso que las operaciones *hijoIz*, *hihoDr* y *raíz* sean $O(1)$, lo cual se cumple en las implementaciones habituales de los árboles binarios. Concretamente, si la implementación es dinámica, *hijoIz* e *hijoDr* devolverán un puntero sin crear ningún nodo nuevo.

Implementación iterativa con ayuda de una pila

De todos es sabido que la ejecución de algoritmos recursivos es más costosa que la de algoritmos iterativos de la misma complejidad. Es por ello que nos plateamos la realización iterativa de los recorridos en profundidad.

Es posible obtener algoritmos de recorrido iterativos con ayuda de pilas de árboles. La idea básica es: en vez de hacer llamadas recursivas para recorrer los árboles hijos, apilamos estos en un orden tal que cuando luego los desapilemos, sean procesados en el orden correcto.

Veamos gráficamente cómo funciona esta idea en un ejemplo. Vamos a recorrer en preorden el árbol que representa a la expresión *x*(–y)+(–x)*y*



Mostramos la pila de árboles indicando las posiciones de las raíces de los árboles apilados en ella, cuando estos se consideran como subárboles de *a*:

Pila (cima a la derecha)	Recorrido
ε	
	+
2, 1	
	+ *
2, 1.2, 1.1	
	+ * x
2, 1.2	
	+ * x –
2, 1.2.1	
	+ * x – y
2	
	etc.

Desarrollemos ahora formalmente esta idea.

Dada cualquier operación de recorrido de árboles:

```
R : Arbin[Elem] → Lista[Elem]
```

podemos generalizarla obteniendo otra operación que acumula el efecto de *R* sobre una pila *as* de árboles, concatenando las listas resultantes. Formalmente:

```
acumula_R: Pila[Arbin[Elem]] → Lista[Elem]
acumula_R(PilaVacía) = []
acumula_R(Apilar(a, as)) = R(a) ++ acumula_R(as)
```

Usaremos las operaciones $acumula_R$ para formular los invariantes de los algoritmos que siguen. También usaremos el concepto de *tamaño* de un árbol binario, definido como la suma de nodos y el número de arcos. Formalmente:

```
tamaño: Arbin[Elem] → Nat
tamaño(Vacío)         = 0
tamaño(Cons(iz, x, dr))  = 1              si     esVacío(ix) AND
esVacío(dr)
                      = 2 + tamaño(iz)    si NOT esVacío(ix) AND
esVacío(dr)
                      = 2 + tamaño(dr)    si     esVacío(ix) AND NOT
esVacío(dr)
                      = 3 + tamaño(iz)
                          + tamaño(dr)    si NOT esVacío(ix) AND NOT
esVacío(dr)
```

Para no perjudicar la eficiencia, utilizamos la implementación procedimental de *ponDr*:

```
proc ponDr( es xs : Lista[Elem]; e x : Elem );
```

Recorrido en preorden:

```
func preOrd( a : Arbin[Elem] ) dev xs : Lista[Elem];
{ P₀ : cierto }
var
  as : Pila[Arbin[Elem]];
  aux, iz, dr : Arbin[Elem];
  x : Elem;
inicio
  LISTA.Nula(xs);
  si ARBIN.esVacío(a)
    entonces
      seguir
    sino
      PILA.PilaVacía(as);
      PILA.Apilar(a, as);
      { I: preOrd(a) = xs ++ acumula_preOrd(as);
        C: suma de los tamaños de los árboles de as }
      it NOT PILA.esVacía(as) →
        aux := PILA.cima(as);        /* aux no es vacío */
        PILA.desapilar(as);
        x := ARBIN.raíz(aux);
        iz := ARBIN.hijoIz(aux);
        dr := ARBIN.hijoDr(aux);
        LISTA.ponDr(xs, x);        /* visitar x */
        si ARBIN.esVacío(dr)         /* apilar dr, iz */
          entonces
            seguir
```

```
          sino
            PILA.apilar(dr, as)
        fsi;
        si ARBIN.esVacío(iz)
          entonces
            seguir
          sino
            PILA.apilar(iz, as)
        fsi
      fit
  fsi
{ Q₀ : xs = preOrd(a) }
  dev xs
ffunc
```

Recorrido en postorden:

```
func postOrd( a : Arbin[Elem] ) dev xs : Lista[Elem];
{ P₀ : cierto }
var
  as : Pila[Arbin[Elem]];
  aux, iz, dr : Arbin[Elem];
  x : Elem;
inicio
  LISTA.Nula(xs);
  si ARBIN.esVacío(a)
    entonces
      seguir
    sino
      PILA.PilaVacía(as);
      PILA.Apilar(a, as);
      { I: postOrd(a) = xs ++ acumula_postOrd(as);
        C: suma de los tamaños de los árboles de as }
      it NOT PILA.esVacía(as) →
        aux := PILA.cima(as);        /* aux no es vacío */
        PILA.desapilar(as);
        x := ARBIN.raíz(aux);
        iz := ARBIN.hijoIz(aux);
        dr := ARBIN.hijoDr(aux);
        si ARBIN.esVacío(iz) AND ARBIN.esVacío(dr)
          entonces
            LISTA.ponDr(xs, x);         /* visitar x */
          sino                          /* apilar x, dr, iz */
            PILA.Apilar(ARBIN.Cons(ARBIN.Vacío, x, ARBIN.Vacío), as);
            si ARBIN.esVacío(dr)
              entonces
                seguir
              sino
```

```
                PILA.apilar(dr, as)
            fsi;
            si ARBIN.esVacío(iz)
              entonces
                seguir
              sino
                PILA.apilar(iz, as)
            fsi
        fsi
      fit
  fsi
{ Q₀ : xs = postOrd(a) }
  dev xs
ffunc
```

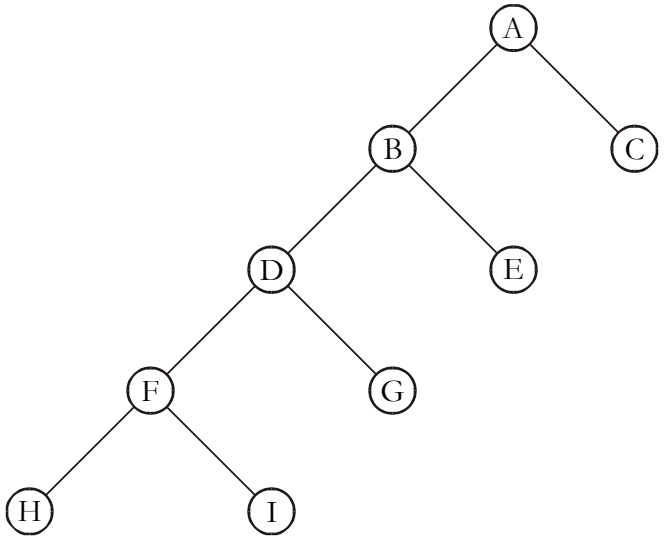
Recorrido en inorden:

```
func inOrd( a : Arbin[Elem] ) dev xs : Lista[Elem];
{ P₀ : cierto }
var
  as : Pila[Arbin[Elem]];
  aux, iz, dr : Arbin[Elem];
  x : Elem;
inicio
  LISTA.Nula(xs);
  si ARBIN.esVacío(a)
    entonces
      seguir
    sino
      PILA.PilaVacía(as);
      PILA.Apilar(a, as);
      { I: inOrd(a) = xs ++ acumula_inOrd(as);
        C: suma de los tamaños de los árboles de as }
      it NOT PILA.esVacía(as) →
        aux := PILA.cima(as);        /* aux no es vacío */
        PILA.desapilar(as);
        x := ARBIN.raíz(aux);
        iz := ARBIN.hijoIz(aux);
        dr := ARBIN.hijoDr(aux);
        si ARBIN.esVacío(iz) AND ARBIN.esVacío(dr)
          entonces
            LISTA.ponDr(xs, x);         /* visitar x */
          sino                          /* apilar dr, x, iz */
            si ARBIN.esVacío(dr)
              entonces
                seguir
              sino
                PILA.apilar(dr, as)
```

```
                fsi;
                PILA.Apilar(ARBIN.Cons(ARBIN.Vacío, x, ARBIN.Vacío), as);
                si ARBIN.esVacío(iz)
                  entonces
                    seguir
                  sino
                    PILA.apilar(iz, as)
                fsi
            fsi
          fit
      fsi
  { Q₀ : xs = inOrd(a) }
      dev xs
  ffunc
```

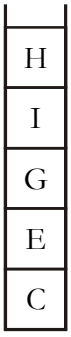
Si todas las operaciones involucradas de pilas, árboles y listas tienen complejidad $O(1)$ entonces los tres recorridos tienen complejidad $O(n)$, siendo n el número de nodos. En el recorrido en preorden cada nodo pasa una sola vez por la cima de la pila, por lo tanto se realizan n iteraciones. En los recorridos en *inorden* y *postorden* los nodos internos pasan 2 veces y las hojas una sola vez, por lo tanto se realizan menos de $2 \cdot n$ pasadas.

En cuanto a la complejidad en espacio, debemos preguntarnos cuál es el máximo número de elementos que debe almacenar la pila; considerando que en una implementación dinámica cada nodo de la pila ocupa espacio 2: 1 por el puntero al árbol y 1 por el puntero al siguiente nodo. El caso peor se da para árboles de la forma:



Para un árbol de esta forma con n nodos:

— *preOrd* llega a crea una pila con $n/2$ nodos:



— *inOrd* y *postOrd* llegan a crear una pila con *n* nodos.

El caso mejor para *preOrd* se da cuando cada nodo interno tiene exactamente un hijo. En ese caso se mantiene una pila con un solo nodo.

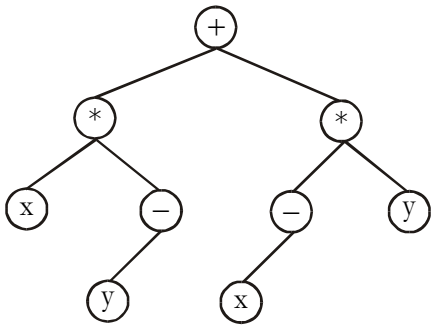
Con un árbol totalmente degenerado hacia la izquierda, *inOrd* y *postOrd* también están en el caso peor de construir pilas con *n* elementos.

Con un árbol totalmente degenerado hacia la derecha tenemos el caso mejor para *inOrd* porque la pila alcanza un tamaño máximo de 2. En cambio para *postOrd*, sigue siendo el caso peor con una pila de tamaño *n*.

Cuando tenemos un árbol completo con *n* nodos, tenemos el caso intermedio para *preOrd* con un tamaño máximo para la pila de *log n*. Este es el caso mejor para *postOrd* con tamaño *2·log n*. Y también caso intermedio para *inOrd* con tamaño *2·log n*.

5.3.2 Recorrido por niveles

En este caso, el orden recorrido no está tan relacionado con la estructura recursiva del árbol y por lo tanto no es natural utilizar un algoritmo recursivo. Se utiliza una cola de árboles, de forma que para recorrer un árbol con hijos, se visita la raíz y se ponen en la cola los dos hijos. Veamos gráficamente cómo funciona esta idea el recorrido por niveles del ejemplo anterior, *x*(–y)+(–x)*y*



Mostramos la cola de árboles indicando las posiciones de las raíces de los árboles almacenados en ella, cuando estos se consideran como subárboles de *a*:

Cola (último a la derecha)	Recorrido
ε	
	+
1, 2	
	+ *
2, 1.1, 1.2	
	+ * *
1.1, 1.2, 2.1, 2.2	
	+ * * x
1.2, 2.1, 2.2	
	+ * * x –
2.1, 2.2, 1.2.1	
	etc. …

Especificación algebraica

Necesitamos utilizar una cola de árboles auxiliar. La cola se inicializa con el árbol a recorrer. El recorrido de la cola de árboles tiene como caso base la cola vacía. Si de la cola se extrae un árbol vacío, se avanza sin más. Y si de la cola se extrae un árbol no vacío, se visita la raíz, y se insertan en la cola el hijo izquierdo y el derecho, por este orden

```
tad REC-NIVELES-ARBIN[E :: ANY]
    usa
      ARBIN[E], LISTA[E]
    usa privadamente
      COLA[ARBIN[E]]
    operaciones
      niveles: Arbin[Elem] → Lista[Elem]                        /* obs */
    operaciones privadas
      nivelesCola: Cola[Arbin[Elem]] → Lista[Elem]              /* obs */
    ecuaciones
      ∀ a : Arbin[Elem] : ∀ as : Cola[Arbin[Elem]] :
      niveles(a)      = nivelesCola(Añadir(a, ColaVacía))
      nivelesCola(as) = []    si esVacía(as)
      nivelesCola(as) = nivelesCola(Avanzar(as))
                           si NOT esVacía(as) AND esVacío(primero(as))
      nivelesCola(as) = [raíz(primero(as))] ++
                        nivelesCola(Añadir(hijoDr(primero(as)),
                                           Añadir(hijoIz(primero(as)),
                                                  avanzar(as))))
                           si NOT esVacía(as) AND NOT esVacío(primero(as))
ftad
```

Implementación iterativa con ayuda de una cola

A partir de la especificación algebraica es sencillo llegar a una implementación iterativa. el invariante del bucle utiliza las dos operaciones de la especificación:

```
niveles(a) = xs ++ nivelesCola(as)
```

donde *a* es el árbol dado para recorrer, *xs* es la lista con la parte ya construida del recorrido y *as* es la cola de árboles pendientes de ser recorridos.

```
func niveles ( a : Arbin[Elem] ) dev xs : Lista[Elem];
{ P₀ : cierto }
var
  as : Cola[Arbin[Elem]];
  aux, iz, dr : Arbin[Elem];
  x : Elem:
inicio
  LISTA.Nula(xs);
  si ARBIN.esVacío(a)
```

```
      entonces
        seguir
      sino
        COLA.ColaVacía(as);
        COLA.Añadir(a, as);
        { I : niveles(a) = xs ++ nivelesCola(as) ∧
              todos los árboles de la cola as son no vacíos ;
          C : suma de los tamaños de los árboles de as         }
        it NOT COLA.esVacía(as) →
          aux := COLA.primero(as);
          COLA.avanzar(as);
          x := ARBIN.raíz(aux);
          iz := ARBIN.hijoIz(aux);
          dr := ARBIN.hijoDr(aux);
          LISTA.ponDr(xs, x);        /* visitar x */
          si ARBIN.esVacío(iz)
            entonces
              seguir
            sino                     /* iz a la cola */
              COLA.Añadir(iz, as)
          fsi;
          si ARBIN.esVacío(dr)
            entonces
              seguir
            sino                     /* dr a la cola */
              COLA.Añadir(dr, as)
          fsi
        fit
    fsi
  { Q₀ : xs = niveles(a) }
    dev xs
  ffun
```

En cuanto a la complejidad, si todas las operaciones involucradas de pilas, colas y árboles tienen complejidad $O(1)$, entonces el recorrido por niveles resulta de $O(n)$, siendo n el número de nodos. La razón es que cada nodo pasa una única vez por la primera posición de la cola.

En cuanto al espacio, si suponemos una implementación dinámica para ARBIN y COLA, cada nodo de la cola ocupará espacio 2 (1 puntero al árbol + 1 puntero al siguiente).

Cuando la cola está encabezada por un subárbol a_0 del árbol inicial a, tal que la raíz de a_0 esté a nivel n, puede asegurarse que el resto de la cola sólo contiene subárboles de nivel n (más a la derecha que a_0) o $n+1$ (hijos de subárboles de nivel n a la izquierda de a_0). El caso peor se dará para un árbol completo, justo después de visitar el último nodo del penúltimo nivel del árbol, cuando la cola contendrá todos los nodos del último nivel, es decir, para un árbol de talla t tendremos 2^{t-1} nodos.

5.3.3 Arboles binarios hilvanados

Los árboles hilvanados son una representación de los árboles binarios que permite algoritmos *O*(n) de recorrido en profundidad, sin ayuda de una pila auxiliar. La idea fundamental es aprovechar los muchos punteros nulos que quedan desaprovechados en la representación dinámica del árbol binario. En particular, se puede demostrar por inducción que un árbol binario con *n* nodos tiene *2n+1* subárboles, de los cuales *n+1* son vacíos; por lo tanto, su representación dinámica tendrá *n+1* punteros nulos.

Los punteros nulos se sustituyen por punteros que facilitan un tipo particular de recorridos. Se necesita algo más de espacio para cada nodo pues es necesario indicar si los punteros son normales o *hilvanes*.

5.3.4 Transformación de la recursión doble a iteración

El esquema de la transformación está en los apuntes. La idea es que la ejecución de un algoritmo doblemente recursivo se corresponde con un recorrido en postorden del árbol de llamadas determinado por la llamada inicial. La pila representa en cada momento el camino desde la raíz hasta el nodo que se está visitando en un momento dado.

5.4 Arboles de búsqueda

El almacenamiento ordenado de elementos es útil es muchas aplicaciones, como por ejemplo en la implementación eficiente de conjuntos y tablas, o de algoritmos eficientes de ordenación.

Sabemos que la búsqueda en un vector ordenado que contenga *n* elementos es posible en tiempo *O*(log n). Pero otras operaciones en vectores ordenados, tales como *inserción* y *borrado*, exigen tiempo *O*(n) en el caso peor.

Las representaciones enlazadas de colecciones de datos que hemos estudiado hasta hora – pilas, listas, colas, … – permiten obtener implementaciones de las inserciones y supresiones en tiempo *O*(1), sin embargo las búsquedas son de complejidad *O*(n) en el caso peor. La razón es que no tenemos una operación de acceso directo con coste *O*(1), lo cual impide realizar una búsqueda binaria en colecciones implementadas con una representación enlazada. Lo más que podemos conseguir, si tenemos los elementos ordenados, es una complejidad *O*(n/2) en el caso promedio.

Los árboles de búsqueda son una solución a este problema porque permiten realizar las tres operaciones –inserción, borrado y búsqueda– en tiempo *O*(log n). Y además permiten obtener una lista ordenada de todos los elementos en tiempo *O*(n).

Un requisito básico para este tipo de árboles es que debe ser posible establecer un orden entre los elementos. Esto puede definirse de distintas formas, ya sea porque exista una función aplicable sobre los elementos que de resultados en un dominio ordenado, o porque los datos estén formados por parejas de valores: la información y el valor en el dominio ordenado. Para presentar los conceptos básicos nos ocuparemos de una simplificación de los árboles de búsqueda donde son los propios elementos los que pertenecen a un dominio ordenado, para luego generalizar esa restricción a otros tipos de elementos.

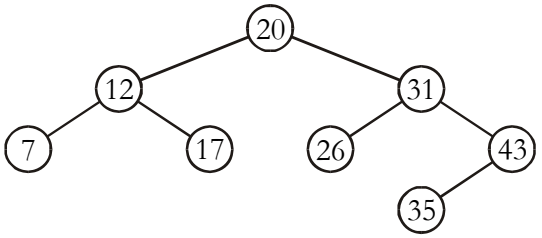
5.4.1 Arboles ordenados

Un árbol ordenado es un árbol binario que almacena elementos de un tipo de la clase ORD. Se dice que el árbol *a* está ordenado si se da alguno de los dos casos siguientes:

— *a* es vacío.

— *a* no es vacío, sus dos hijos están ordenados, todos los elementos del hijo izquierdo son estrictamente menores que el elemento de la raíz, y el elemento de la raíz es estrictamente menor que todos los elementos del hijo derecho.

Por ejemplo, el siguiente es un árbol ordenado:



Nótese que en la anterior definición no se admiten elementos repetidos.

Podemos especificar algebraicamente una operación que reconozca si un árbol binario está ordenado:

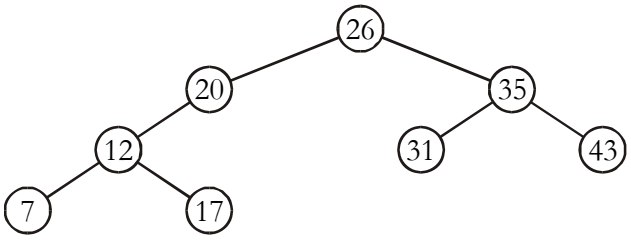
```
ordenado?(Vacío)  = cierto
ordenado?(Cons(iz, x, dr))  = ordenado?(iz) AND menor(iz, x) AND
                  ordenado?(dr) AND mayor(dr, x)
menor(Vacío, y) = cierto
menor(Cons(iz, x, dr), y) = x < y AND menor(iz, y) AND menor(dr, y)
mayor(Vacío, y) = cierto
mayor(Cons(iz, x, dr), y) = x > y AND mayor(iz, y) AND mayor(dr, y)
```

Una cualidad muy interesante de los árboles ordenados es que su recorrido en inorden produce una lista ordenada de elementos. De hecho, esta es una forma de caracterizar a los árboles ordenados:

— Un árbol binario *a* es un árbol ordenado si y sólo si *xs* = *inOrd(a)* está ordenada, es decir.

$$\forall\ i, j: 1 \leq i < j \leq \#xs : xs\ !!\ i < xs\ !!\ j$$

Es posible construir distintos árboles ordenados con la misma información, o, dicho de otro modo, dos árboles de búsqueda distintos pueden dar el mismo resultado al recorrerlos en inorden. Por ejemplo, el siguiente árbol produce el mismo recorrido que el presentado anteriormente:



Las operaciones que nos interesan sobre árboles ordenados son: la inserción, la búsqueda y el borrado. En esencia, las tres operaciones tienen que realizar una búsqueda en el árbol ordenado, donde se saca partido de dicha característica. Resulta además interesante, equipar a los árboles ordenados con una operación de recorrido y otra que nos indique si un determinado elemento está o no en el árbol.

Inserción en un árbol ordenado

En principio, suponemos que si intentamos insertar un elemento que ya está en el árbol, el resultado es el mismo árbol. Al presentar los árboles de búsqueda refinaremos esta idea.

La inserción se especifica de forma que si el árbol está ordenado, siga estándolo después de la inserción. La inserción de un dato *y* en un árbol ordenado *a*:

- Si *a* es vacío, entonces el resultado es un árbol con *y* en la raíz e hijos vacíos.
- Si *a* no es vacío:
  - Si *y* coincide con la raíz de *a* entonces el resultado es *a*.
  - Si *y* es menor que la raíz de *a*, entonces se inserta *y* en el hijo izquierdo de *a*.
  - Si *y* es mayor que la raíz de *a*, entonces se inserta *y* en el hijo derecho de *a*.

Algebraicamente se puede especificar la inserción de la siguiente forma:

```
inserta(y, Vacío)    = Cons(Vacío, y, Vacío)
inserta(y, Cons(iz, x, dr)) = Cons(iz, x, dr)              si y == x
inserta(y, Cons(iz, x, dr)) = Cons(inserta(y, iz), x, dr) si y < x
inserta(y, Cons(iz, x, dr)) = Cons(iz, x, inserta(y, dr)) si y > x
```

Se puede observar que el primer ejemplo que presentamos de árbol ordenado es el resultado de insertar sucesivamente: 20, 12, 17, 31, 26, 43, 7 y 35, en un árbol inicialmente vacío.

Búsqueda en un árbol ordenado

Especificamos esta operación de forma que devuelva como resultado el subárbol obtenido tomando como raíz el nodo que contenga el árbol buscado; así, a continuación, podríamos obtener la información de ese nodo mediante la operación que obtiene la raíz del árbol. Si ningún nodo del árbol, contiene el elemento buscado, se devuelve el árbol vacío.

La idea de la búsqueda es similar a la inserción: si el elemento buscado está en la raíz, ya hemos terminado; si es menor que la raíz buscamos en el hijo izquierdo; y si es mayor que la raíz, buscamos en el hijo derecho. Algebraicamente, podemos especificarla como:

```
busca(y, Vacío) = Vacío
busca(y, Cons(iz, x, dr)) = Cons(iz, x, dr)  si y == x
busca(y, Cons(iz, x, dr)) = busca(y, iz)     si y < x
busca(y, Cons(iz, x, dr)) = busca(y, dr)     si y > x
```

Borrado en un árbol ordenado

La operación de borrado es la más complicada, porque tenemos que reconstruir el árbol resultado de la supresión.

Se busca el valor *y* en el árbol. Si la búsqueda fracasa, la operación termina sin modificar el árbol. Si la búsqueda tiene éxito y localiza un nodo de posición α, el comportamiento de la operación depende del número de hijos de α:

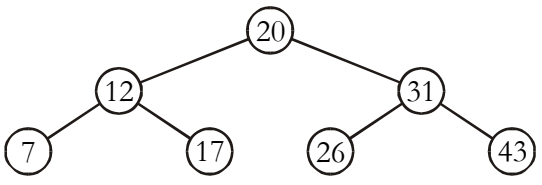
- Si α es una hoja, se elimina el nodo α
- Si α tiene un solo hijo, se elimina el nodo α y se coloca en su lugar el árbol hijo, cuya raíz quedará en la posición α
- Si α tiene dos hijos se procede del siguiente modo:
  - Se busca el nodo con el valor mínimo en el hijo derecho de α. Sea α' la posición de éste. (Alternativamente, se podría buscar el mayor nodo del hijo izquierdo de α.)
  - El elemento del nodo α se reemplaza por el elemento del nodo α'.
  - Se borra el nodo α'. Nótese que, por ser el mínimo del hijo derecho de α, α' no puede tener hijo izquierdo, por lo tanto, estaremos en la situación de eliminar una hoja o un nodo con un solo hijo –el derecho–.

Especificado algebraicamente:

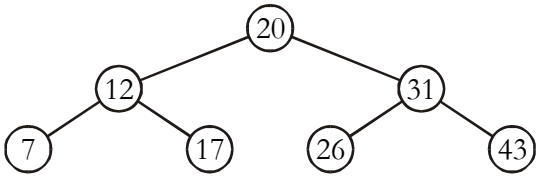
```
borra(y, Vacío)   = Vacío
borra(y, Cons(iz, x, dr)) = dr      si y == x AND esVacío(iz)
borra(y, Cons(iz, x, dr)) = iz      si y == x AND esVacío(dr)
borra(y, Cons(iz, x, dr)) = Cons(iz, z, borra(z, dr))
                            si y == x AND NOT esVacío(iz) AND
                               NOT esVacío(dr) AND z = min(dr)
borra(y, Cons(iz, x, dr)) = Cons(borra(y, iz), x, dr)   si y < x
borra(y, Cons(iz, x, dr)) = Cons(iz, x, borra(y, dr))   si y > x
```

Por ejemplo, aplicamos algunas operaciones de borrado al primer ejemplo que presentamos de árbol ordenado:

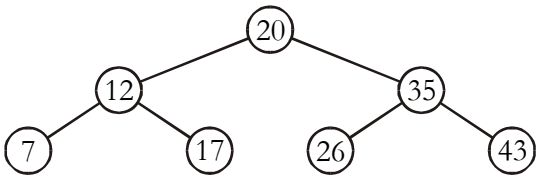
- *borra(35, a)*



- *borra(43, a)*



- *borra(31, a)*



Especificación algebraica del TAD ARB-ORD

Con todo lo anterior podemos presentar la especificación algebraica del TAD ARB-ORD. Lo especificamos como un enriquecimiento del TAD REC-PROF-ARBIN porque queremos que incluya una operación de recorrido que es precisamente el recorrido en inorden especificado en dicho TAD, el cual a su vez, es un enriquecimiento del TAD ARBIN, de donde importamos el tipo Arbin[Elem], junto con sus operaciones. Para hacer más legible la especificación, renombramos el tipo importado de *Arbin[Elem]* a *Arbus[Elem]* y la operación *inOrden* a *recorrido*; este es un mecanismo que ya habíamos utilizado anteriormente. Aparece así mismo un mecanismo nuevo: la ocultación de identificadores importados. De esta forma limitamos la interfaz eliminando de ella operaciones importadas que no tienen sentido sobre el TAD que estamos especificando. El resultado es que este TAD exporta el tipo *Arbus[Elem]* equipado con las operaciones: Vacío, raíz, esVacío, recorre, inserta, busca, borra y está?.

Nótese también que la operación privada *ordenado?* no se utiliza en la especificación de las operaciones exportadas. La razón de incluir esta operación es poder luego utilizarla en los razonamientos formales con árboles ordenados: asertos, invariante de la representación, …

```
tad ARB-ORD[E :: ORD]
    usa
      REC-PROF-ARBIN[E]    renombrando  inOrd a recorre
                              Arbin[Elem] a Arbus[Elem]
                ocultando    preOrd, postOrd,
                              Cons, hijoIz, hijoDr
    tipo
      Arbus[Elem]
    operaciones
      inserta: (Elem, Arbus[Elem]) → Arbus[Elem] /* gen */
      busca: (Elem, Arbus[Elem]) → Arbus[Elem] /* mod */
      borra: (Elem, Arbus[Elem]) → Arbus[Elem] /* mod */
      está?: (Elem, Arbus[Elem]) → Bool  /* obs */
    operaciones privadas
      ordenado?: Arbus[Elem] → Bool /* obs */
      min: Arbus[Elem] – → Elem  /* obs */
      mayor, menor: (Arbus[Elem], Elem) → Bool /* obs */
    ecuaciones
      ∀ x, y, z : Elem : ∀ a, iz, dr : Arbus[Elem] :
      def min(a) si NOT esVacío(a)
      min(Cons(iz, x, dr)) = x  si     esVacío(iz)
      min(Cons(iz, x, dr)) = min(iz)  si NOT esVacío(iz)
      menor(Vacío, y) = cierto
      menor(Cons(iz, x, dr), y) = x < y AND menor(iz, y) AND menor(dr, y)
```

```
        mayor(Vacío, y) = cierto
        mayor(Cons(iz, x, dr), y) = x > y AND mayor(iz, y) AND mayor(dr, y)
        ordenado?(Vacío)  = cierto
        ordenado?(Cons(iz, x, dr))  = ordenado?(iz) AND menor(iz, x) AND
            ordenado?(dr) AND mayor(dr, x)
        inserta(y, Vacío) = Cons(Vacío, y, Vacío)
        inserta(y, Cons(iz, x, dr)) = Cons(iz, x, dr)             si y == x
        inserta(y, Cons(iz, x, dr)) = Cons(inserta(y, iz), x, dr) si y < x
        inserta(y, Cons(iz, x, dr)) = Cons(iz, x, inserta(y, dr)) si y > x
        busca(y, Vacío) = Vacío
        busca(y, Cons(iz, x, dr)) = Cons(iz, x, dr)  si y == x
        busca(y, Cons(iz, x, dr)) = busca(y, iz)     si y < x
        busca(y, Cons(iz, x, dr)) = busca(y, dr)     si y > x
        borra(y, Vacío) = Vacío
        borra(y, Cons(iz, x, dr)) = dr      si y == x AND esVacío(iz)
        borra(y, Cons(iz, x, dr)) = iz      si y == x AND esVacío(dr)
        borra(y, Cons(iz, x, dr)) = Cons(iz, z, borra(z, dr))
                    si y == x AND NOT esVacío(iz) AND
                       NOT esVacío(dr) AND z = min(dr)
        borra(y, Cons(iz, x, dr)) = Cons(borra(y, iz), x, dr)   si y < x
        borra(y, Cons(iz, x, dr)) = Cons(iz, x, borra(y, dr))   si y > x
        está?(y, a)  = NOT esVacío(busca(y, a))
    errores
        min(Vacío)
ftad
```

Complejidad de las operaciones

Claramente, la complejidad de todas las operaciones está determinada por la complejidad de la búsqueda. El tiempo de una búsqueda en el caso peor es $O(t)$, siendo t la talla del árbol.

— El caso peor se da en un árbol *degenerado* reducido a una sola rama, en cuyo caso la talla de un árbol de búsqueda con n nodos en n. Tales árboles pueden producirse a partir del árbol vacío por la inserción consecutiva de n elementos ordenados –en orden creciente o decreciente–.

— Se puede demostrar que el promedio de las longitudes de los caminos en un árbol de búsqueda generado por la inserción de una sucesión aleatoria de n elementos con claves distintas es asintóticamente

$$t_n = 2 \cdot (\ln n + \gamma + 1)$$

siendo $\gamma = 0{,}577\ldots$ la constante de Euler, y suponiendo que las n! permutaciones posibles de los nodos que se insertan son equiprobables (véase N. Wirth, *Algorithms and Data Structures*, Prentice Hall, 1986, pp. 214-217).

La intuición que hay detrás de este resultado es que en promedio un árbol de búsqueda se comporta como un árbol completo, para el que ya vimos que su talla venía dada por la expresión $\log(n+1)$ siendo n el número de nodos.

Este análisis es cualitativamente similar al análisis de la complejidad del *quicksort*, donde el caso peor se tiene cuando el vector ya está ordenado, porque en ese caso la partición del subvector sólo reduce en 1 el tamaño del problema. Este caso se corresponde con el árbol degenerado.

Si se quiere garantizar complejidad de $O(\log n)$ en el caso peor, es necesario restringirse a trabajar con alguna subclase de los árboles ordenados en la cual la talla se mantenga logarítmica con respecto al número de nodos. De esto nos ocuparemos en el siguiente apartado del tema, dedicado a los árboles AVL.

No nos ocupamos de la implementación de los árboles ordenados, ya que sólo los hemos introducido para presentar los conceptos básicos de los árboles de búsqueda, que describimos a continuación, y de cuya implementación sí nos ocuparemos.

5.4.2 Arboles de búsqueda

La diferencia entre los árboles de búsqueda y los árboles ordenados radica en las condiciones que se exigen a los elementos. En los árboles ordenados simplemente se exige que el tipo de los elementos pertenezca a la clase ORD, es decir esté equipado con operaciones de igualdad y comparación. Sin embargo, en muchas aplicaciones las comparaciones entre elementos se puede establecer en base a una parte de la información total que almacenan dichos elementos: un campo *clave*. Esta idea proviene de las bases de datos.

Supongamos por ejemplo que tuviésemos que manejar una colección de datos de personas donde uno de los campo almacenados fuese su DNI. Errores administrativos al margen, podemos suponer que el DNI es único y que, por lo tanto podemos identificar a una persona simplemente a partir de su DNI. Si almacenásemos la colección de datos en un árbol ordenado, podríamos utilizar simplemente el DNI como *clave de acceso*.

En esta situación, lo que deberíamos exigirle a los elementos de un árbol de búsqueda es que dispusieran de una operación observadora cuyo resultado fuese de un tipo ordenado:

```
clave: Elem → Clave
```

Siendo *Clave* un tipo que pertenece a la clase ORD. Definiríamos así la clase de los tipos que tienen un operación de la forma *clave*, y especificaríamos e implementaríamos los árboles de búsqueda en términos de esta clase. Nótese que con este planteamiento los árboles ordenados son un caso particular de los árboles de búsqueda donde la operación *clave* es la identidad.

Sin embargo, podemos generalizar aún más el planteamiento si consideramos que la clave y el valor son datos separados, de forma que en cada nodo del árbol se almacenen dos datos. Nótese que este planteamiento engloba al anterior aunque tiene el inconveniente de que las claves se almacenarían dos veces. Así, el TAD ARBUS tendrá dos parámetros: el TAD de las claves y el TAD de los valores.

Al separar las claves y los valores, se nos plantea la cuestión de qué hacer cuando intentamos insertar un elemento asociado con una clave que ya está en el árbol. Dependiendo del tipo de los elementos tendrá sentido realizar distintas operaciones.

Con el objetivo de que el TAD sea lo más general posible, la solución que adoptamos es exigir que el tipo de los valores tenga una operación modificadora que *combine* elementos, con la signatura:

```
( ⊕ ) : (Elem, Elem) → Elem
```

De forma que cuando insertemos un elemento asociado a una clave que ya existe, utilicemos esta operación para *combinar* los valores y obtener el nuevo dato que se debe almacenar en el árbol. Así, en cada ejemplar del TAD tendremos un comportamiento específico del tipo de los valores, según la operación que haya sido designada para realizar la combinación. Por ejemplo, si queremos que el nuevo elemento reemplace al antiguo entonces definiremos esta función como:

```
x ⊕ y = y
```

Definimos entonces la siguiente clase de tipos, a la que deben pertenecer los valores de un árbol de búsqueda

```
clase COMB
    hereda
       ANY
    operaciones
       ( ⊕ ): (Elem, Elem) → Elem
    % Operación de combinación de elementos.
fclase
```

Es decir, estamos exigiendo que los valores de un árbol de búsqueda tengan una operación con esa signatura que se llame *combina*. Sería más flexible si en el lenguaje de implementación pudiésemos, al declarar una variable concreta de tipo *Arbus*, indicar explícitamente cuál es la operación que se corresponde con *combina*.

La idea de separar las claves de los datos, e indicar cuál es la forma de combinar datos, se puede aplicar a todos los TAD que manejan colecciones de datos. Aunque tiene más sentido cuando las claves están ordenadas y nos ayudan a construir una implementación más eficiente.

Con todo esto, la especificación de los árboles de búsqueda queda igual que la de los árboles ordenados, pero sustituyendo los elementos por parejas de (clave, valor), y utilizando la operación de combinación cuando se inserta un elemento asociado con una clave que ya existe.

Especificación algebraica

```
tad ARBUSCA[C :: ORD, V :: COMB]
 renombra C.Elem a Cla
          V.Elem a Val
 usa
    REC-PROF-ARBIN[PAREJA[C,V]] renombrando  inOrd a recorre
                                   Arbin[Pareja[Cla,Val]] a Arbus[Cla, Val]
                     ocultando     preOrd, postOrd,
                                   Cons, hijoIz, hijoDr
 tipo
    Arbus[Cla, Val]
 operaciones
    inserta: (Cla, Val, Arbus[Cla, Val]) → Arbus[Cla, Val]  /* gen */
    busca: (Cla, Arbus[Cla, Val]) → Arbus[Cla, Val] /* mod */
    borra: (Cla, Arbus[Cla, Val]) → Arbus[Cla, Val] /* mod */
    está?: (Cla, Arbus[Cla, Val]) → Bool  /* obs */
 operaciones privadas
    ordenado?: Arbus[Cla, Val] → Bool  /* obs */
    min: Arbus[Cla, Val] – → Pareja[Cla, Val]  /* obs */
    mayor, menor: (Arbus[Cla, Val], Cla) → Bool/* obs */
 ecuaciones
    ∀ c, c', d : Cla : ∀ x, x', y : Val : ∀ a, iz, dr : Arbus[Cla, Val] :
```

```
    def min(a) si NOT esVacío(a)
    min(Cons(iz, Par(c,x), dr))    = Par(c, x)  si     esVacío(iz)
    min(Cons(iz, Par(c,x), dr))    = min(iz) si NOT esVacío(iz)
    menor(Vacío, c')     = cierto
    menor(Cons(iz, Par(c,x), dr), c')   = c < c' AND menor(iz, c') AND
menor(dr, c')
    mayor(Vacío, c')     = cierto
    mayor(Cons(iz, Par(c,x), dr), c')   = c > c' AND mayor(iz, c') AND
mayor(dr, c')
    ordenado?(Vacío)     = cierto
    ordenado?(Cons(iz, Par(c,x), dr))   = ordenado?(iz) AND menor(iz, c) AND
              ordenado?(dr) AND mayor(dr, c)
    inserta(c', x', Vacío)    = Cons(Vacío, Par(c', x'), Vacío)
    inserta(c', x', Cons(iz, Par(c,x), dr))  = Cons(iz, Par(c, x⊕x'), dr)
                             si c' == c
    inserta(c', x', Cons(iz, Par(c,x), dr))  = Cons(inserta(c', x', iz),
Par(c,x), dr)
                             si c' < c
    inserta(c', x', Cons(iz, Par(c,x), dr))  = Cons(iz, Par(c,x), inserta(c',
x', dr))
                             si c' > c
    busca(c', Vacío)     = Vacío
    busca(c', Cons(iz, Par(c,x), dr))   = Cons(iz, Par(c,x), dr)  si c' == c
    busca(c', Cons(iz, Par(c,x), dr))   = busca(c', iz)           si c' < c
    busca(c', Cons(iz, Par(c,x), dr))   = busca(c', dr)           si c' > c
    borra(c', Vacío)                = Vacío
    borra(c', Cons(iz, Par(c,x), dr))   = dr      si c' == c AND esVacío(iz)
    borra(c', Cons(iz, Par(c,x), dr))   = iz      si c' == c AND esVacío(dr)
    borra(c', Cons(iz, Par(c,x), dr))   = Cons(iz, Par(d,y), borra(d, dr))
                    si c' == c AND NOT esVacío(iz) AND
                     NOT esVacío(dr) AND Par(d,y) = min(dr)
    borra(c', Cons(iz, Par(c,x), dr))   = Cons(borra(c', iz), Par(c,x), dr) si
c' < c
    borra(c', Cons(iz, Par(c,x), dr))   = Cons(iz, Par(c,x), borra(c', dr)) si
c' > c
    está?(c', a)    = NOT esVacío(busca(c', a))
 errores
    min(Vacío)
ftad
```

Implementación de los árboles de búsqueda

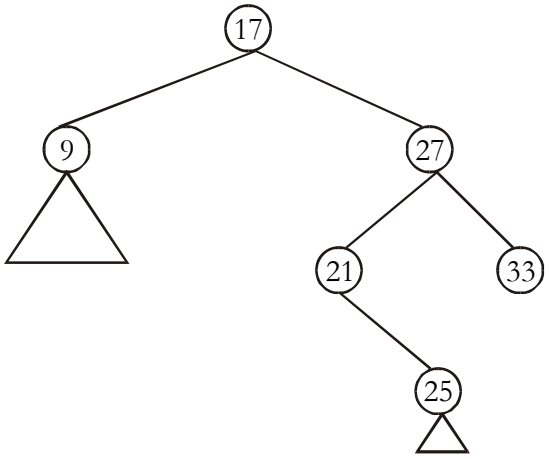
Siguiendo las indicaciones de la especificación, nos podríamos plantear la implementación de los árboles de búsqueda mediante un módulo cliente del módulo de los recorridos y el módulo de los árboles binarios que realizase las operaciones como funciones sin acceder a la representación interna de los árboles. Veamos cómo sería la operación de inserción en árboles ordenados, siguiendo directamente la idea de la especificación algebraica:

```
func inserta( x : Elem; a : Arbus[Elem] ) dev b : Arbus[Elem];
{ P₀ : R(x) ∧ R(a) }
inicio
  si esVacío(a)
    entonces
      b := Cons( Vacío, x, Vacío )
    sino
```

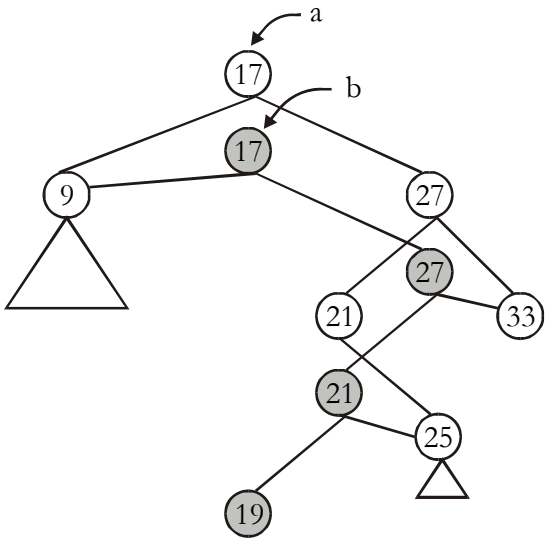
```
        si
          ORD.igual(x, raíz(a)) → b := a
        □ ORD.menor(x, raíz(a)) → b := Cons(inserta(x, hijoIz(a)), raíz(a),
                                           hijoDr(a))
        □ ORD.mayor(x, raíz(a)) → b := Cons(hijoIz(a), raíz(a),
                                           inserta(x, hijoDr(a)))
        fsi
    fsi
  { Q₀ : R(b) ∧ A(b) =ARB-ORD[ELEM] inserta(x, A(a)) }
    dev b
  ffunc
```

En el árbol resultado *b* tendremos que los nodos recorridos en la búsqueda se habrán copiado, como efecto de la operación *Cons*, mientras que el resto de los nodos se compartirán. Un problema similar se plantea con la operación de borrado

Veámoslo con un ejemplo. Dado el árbol *a*



si ejecutamos la operación *b := inserta(19, a)*, el resultado será



A diferencia de la compartición de estructura que producen las funciones *Cons, hijoIz* o *hijoDr*, esta otra es arbitraria y en la práctica impediría cualquier intento de anulación de los árboles invo-

lucrados, a no ser que decidiésemos dejar de usar al mismo tiempo todos los árboles que sabemos que pueden compartir algún nodo. Una solución a este problema sería hacer que el resultado no compartiese ningún nodo con el argumento, para lo cual deberíamos realizar copias al hacer las llamadas recursivas:

```
b := Cons(inserta(x, hijoIz(a)), raíz(a), copia(hijoDr(a)))
b := Cons(copia(hijoIz(a)), raíz(a), inserta(x, hijoDr(a)))
```

Pero de esta forma tenemos que el coste de la operación pasa a $O(n)$, con lo cual perdemos la ventaja que estamos pretendiendo obtener con los árboles ordenados.

La solución es por tanto implementar *inserta* y *borra* como procedimientos que acceden a la estructura interna de la representación; es decir, no podemos implementarlo como un módulo cliente de los árboles binarios, sino como un módulo donde se vuelva a definir el tipo.

Tipo representante

La definición del tipo es similar a la de los árboles binarios, aunque cada nodo contiene dos campos de información en lugar de uno. Lo escribimos tal y como aparecería en el módulo de implementación, para indicar que el tipo de los valores y las claves se importa de los módulos abstractos COMB y ORD.

```
módulo impl ARBUSCA[CLA,VAL]
  importa
    COMB, ORD
  privado
    tipo
      Val = COMB.Elem;
      Cla = ORD.Elem;
      Nodo = reg
        cla : Cla;
        val : Val;
        hi, hd : Enlace
      freg;
      Enlace = puntero a Nodo;
      Arbus[Cla, Val] = Enlace;

% Implementación de las operaciones

fmódulo
```

Invariante de la representación

Exigimos que cumplan el invariante de la representación de los árboles binarios sin compartición de estructura, y que el resultado del recorrido en inorden esté ordenado.

Dado *p : Enlace*

$$R_{Arbus[Cla,Val]}$$

$$\Leftrightarrow_{def}$$

```
R^NC_Arbin[Elem](p) ∧
∀ i, j : 1 ≤ i < j ≤ #inOrd(p) : inOrd(p) !! i < inOrd(p) !! j
```

Nótese que podemos exigir el invariante sin compartición de estructura porque en ARB-BUS no está disponible la operación *Cons*.

Función de abstracción

La misma que se utiliza con árboles binarios:

Implementación de las operaciones

```
func busca( d : Cla; a Arbus[Cla, Val]) dev b : Arbus[Cla, Val];
{ P₀ : R(d) ∧ R(a) }
inicio
  si a == nil
    entonces
      b := nil
    sino
      si
        ORD.igual(d, a^.cla) → b := a
      □ ORD.menor(d, a^.cla) → b := busca(d, a^.hi)
      □ ORD.mayor(d, a^.cla) → b := busca(d, a^.hd)
      fsi
  fsi
{ Q₀ : R(b) ∧ A(b) =ARBUSCA[CLA,VAL] busca(A(d), A(a)) }
  dev b
ffunc


proc inserta( e d : Cla; e y : Val; es a : Arbus[Cla, Val] );
{ P₀ : R(d) ∧ R(y) ∧ R(a) ∧ a = A }
inicio
  si a == nil

    entonces
      ubicar(a);
      a^.cla := d;
      a^.val := y;
      a^.hi := nil;
      a^.hd := nil;
    sino
      si
        ORD.igual(d, a^.cla) → a^.val := COMB.combina( a^.val, y )
      □ ORD.menor(d, a^.cla) → inserta(d, y, a^.hi)    /* ref */
      □ ORD.mayor(d, a^.cla) → inserta(d, y, a^.hd)    /* ref */
      fsi
```

```
  fsi
{ Q₀ : R(a) ∧ A(a) =ARBUSCA[CLA,VAL] inserta(A(d), A(y), A(A)) }
fproc
```

En las dos instrucciones marcadas como *ref* es donde se muestra la necesidad de que esta operación tenga acceso a la representación interna de los árboles. En las siguientes operaciones también ocurre esto cada vez que pasamos como parámetro actual uno de los hijos del nodo en cuestión.

Para la operación de borrado vamos a utilizar dos procedimientos auxiliares privados.

```
proc borra( e d : Cla; es a : Arbus[Cla, Val] );
{ P₀ : R(d) ∧ R(a) ∧ a = A }
inicio
  si a == nil
    entonces
      seguir
    sino
      si
        ORD.igual(d, a^.cla) → borraRaíz(a)
      □ ORD.menor(d, a^.cla) → borra(d, a^.hi)
      □ ORD.mayor(d, a^.cla) → borra(d, a^.hd)
      fsi
  fsi
{ Q₀ : R(a) ∧ A(a) =ARBUSCA[CLA,VAL] borra(A(d), A(A)) }
fproc
```

Las operaciones auxiliares de borrado:

```
proc borraRaíz( es a : Arbus[cla, Val] );
{ P₀ : R(a) ∧ a = A ∧ a /= nil }
var
  vacíoIz, vacíoDr : Bool;
  aux : Arbus[Cla, Val];
inicio
  vacíoIz := a^.hi == nil;
  vacíoDr := a^.hd == nil;
  si
    vacíoIz → aux := a;
              % COMB.anula(a^.val);
              a := a^.hd;
              liberar(aux)
  □ vacíoDr → aux := a;
              % COMB.anula(a^.val);
              a := a^.hi;
              liberar(aux)
```

```
  □ NOT vacíoIz AND NOT vacíoDr → borraConMin(a, a^.hd)
  fsi
{ Q₀ : R(a) ∧ A(a) =ARBUSCA[CLA,VAL] borra(A(A^.cla), A(A)) }
fproc
```

Dejando fijo el puntero *a*, descendemos por sus hijos izquierdos con el parámetro *b* hasta llegar al menor, y en ese punto realizamos el borrado

```
proc borraConMin ( es a, b : Arbus[cla, Val] );
{ P₀ : a = A ∧ R(a) ∧ a /= nil ∧ a^.hi /= nil ∧ a^.hd /= nil ∧
       b es un descendiente de a^.hd vía una cadena de hijos izquierdos }
var
  aux : Arbus[Cla, Val];
inicio
  si b^.hi /= nil
    entonces
      borraConMin(a, b^.hi)
    sino
      a^.cla := b^.cla;
      % COMB.anula(a^.val);
      a^.val := b^.val;
      aux := b;
      b := b^.hd;
      liberar(aux)
  fsi
{ Q₀ : R(a) ∧ A(a) =ARBUSCA[CLA,VAL] borra(A(A^.cla), A(A)) }
Fproc
```

5.5 Arboles AVL

5.5.1 Arboles equilibrados

En el tema anterior vimos cómo en los árboles de búsqueda se consigue una complejidad logarítmica para las operaciones de búsqueda, inserción y borrado en el caso promedio, aunque en le caso peor llega a ser O(n). Si se quiere garantizar complejidad logarítmica en el caso peor, es necesario restringirse a trabajar con alguna subclase de la clase de los árboles ordenados en la cual la talla se mantenga logarítmica con respecto al número de nodos.

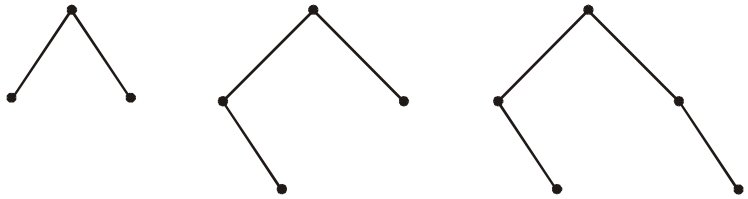
Una familia $\mathcal{F}$ de árboles binarios se llama *equilibrada* si existen constantes $n_0 \in \aleph$, $c \in \Re_+$ tales que para todo $n \geq n_0$ y todo árbol $a \in \mathcal{F}$ con n nodos se tenga $talla(a) \leq c \cdot \log n$.

Efectivamente, para árboles pertenecientes a una familia equilibrada la complejidad de las operaciones antes citadas en el caso peor es O(t) siendo t la talla, y por lo tanto resulta O(log n).

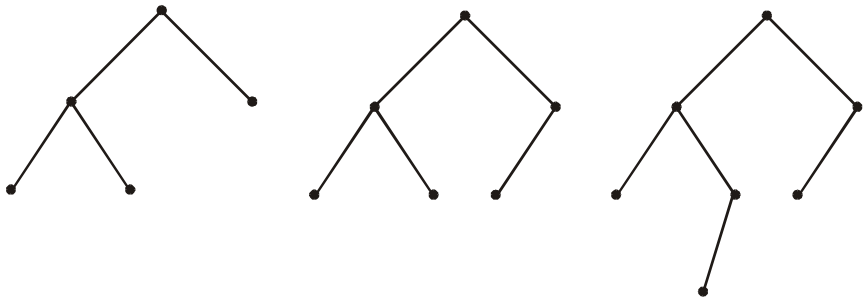
Algunos ejemplos de familias equilibradas son:

— La familia de los árboles semicompletos

— La familia de los árboles equilibrados en número de nodos. Un árbol pertenece a esta familia si cualquier subárbol suyo cumple que el número de nodos del hijo izquierdo y el número de nodos del hijo derecho difieren a lo sumo en 1:



— La familia de los árboles equilibrados en altura. Un árbol pertenece a esta familia si cualquier subárbol suyo cumple que la talla del hijo izquierdo y la talla del hijo derecho difieren a lo sumo en 1:



Nótese que, como se puede observar en los anteriores ejemplos, todos los árboles semicompletos son equilibrados en altura, pero que existen árboles equilibrados en altura que no son semicompletos. La idea es que para conseguir el coste logarítmico no es necesario exigir una condición tan fuerte como la semicompletitud sino que basta con una condición más débil: el equilibrio en altura.

En 1962, G. M. Adel'son-Vel'skii y E. M. Landis demostraron que la familia de los árboles equilibrados en altura son equilibrados en el sentido de la definición dada más arriba. En honor suyo, la familia suele llamarse familia de los árboles AVL. En efecto, los inventores de esta familia demostraron que un árbol AVL con n nodos tiene una talla t acotada como sigue:

$$\log(n+1) \leq t < 1'4404 \cdot \log(n+2) - 0'328$$

El caso peor, es decir, los árboles AVL con el mayor desequilibrio posible, se alcanza para los llamados *árboles de Fibonacci*. Para cada talla $t \geq 0$, el árbol de Fibonacci a_t y su número de nodos n_t se construyen recursivamente como sigue:

— $t = 0$ $\quad a_0 = Vacío$ $\quad n_0 = 0$

— $t = 1$ $\quad a_1 = Cons(a_0, 1, a_0)$ $\quad n_1 = 1$

— $t \geq 2$ $\quad a_t = Cons(a_{t-2}, i, a_{t-1}[+ i])$ $\quad n_t = n_{t-2} + 1 + n_{t-1}$

donde

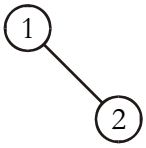
i es el mínimo valor ≥ 1 que no aparece en a_{t-2}

$a_{t-1}[+ i]$ es el árbol que se obtiene sumando i a cada nodo de a_{t-1}

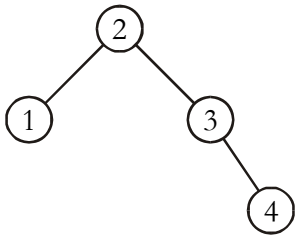
Por ejemplo:

— a_1

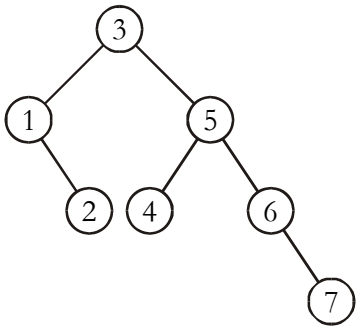
— a_2



— a_3



— a_4



Los números n_t se llaman números de Leonardo. Para los árboles de Fibonacci a_t se alcanza el caso peor de la estimación anterior

$$t \approx 1'4404 \cdot \log(n+2) - 0'328 \approx 1'5 \log n$$

Caracterización de los árboles AVL

Podemos caracterizar formalmente las condiciones que deben cumplir los árboles AVL:

```
avl?(a)   = ordenado?(a) AND equilibrado?(a)

ordenado?(Vacío)     = cierto
ordenado?(Cons(iz, Par(u,x), dr))   = ordenado?(iz) AND menor(iz, u) AND
            ordenado?(dr) AND mayor(dr, u)

menor(Vacío, u)   = cierto
```

```
    menor(Cons(iz, Par(v,x), dr), u)    = v < u AND menor(iz, u) AND menor(dr,
u)

    mayor(Vacío, u)   = cierto
    mayor(Cons(iz, Par(v,x), dr), u)    = v > u AND mayor(iz, u) AND mayor(dr,
u)

    equilibrado?(Vacío)    = cierto
    equilibrado?(Cons(iz, ux, dr))   = equilibrado?(iz) AND equilibrado?(dr)
AND
                | talla(iz) - talla(dr)| ≤ 1

    talla(Vacío)   = 0
    talla(Cons(iz, us, dr))   = 1 + max(talla(iz), talla(dr))
```

5.5.2 Operaciones de inserción y borrado

Según los resultados del apartado anterior, los árboles AVL constituyen una *familia equilibrada*, y, por lo tanto, las operaciones de búsqueda, inserción y borrado van a tener coste *O*(log n) en el caso peor, siempre que *Inserta* y *borra* se modifiquen de manera que al aplicarse a un árbol AVL devuelvan otro árbol AVL.

Las operaciones de inserción y borrado en árboles AVL funcionan básicamente en dos pasos:

— Inserción o borrado como en un árbol de búsqueda ordinario.

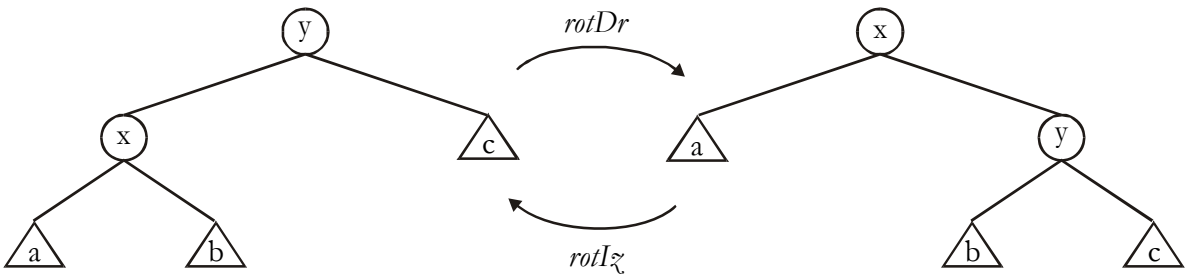
— Reequilibrado del nuevo árbol, si éste h dejado de ser AVL.

El reequilibrado se consigue con ayuda de dos operaciones básicas, llamadas *rotaciones*.

Rotaciones

Las *rotaciones* son operaciones que modifican la estructura de un árbol ordenado, llevando nodos de un subárbol a otro para resolver así un desequilibrio en altura.

La *rotación a la derecha* lleva nodos del subárbol izquierdo al derecho, mientras que la *rotación a la izquierda* lleva nodos del subárbol derecho al izquierdo. Gráficamente:



Una propiedad interesante de este proceso es que si cualquiera de las operaciones de rotación se aplica a un árbol ordenado, el resultado es un nuevo árbol ordenado con el mismo recorrido en inorden que el árbol original.

Formalmente podemos expresar las rotaciones como:

```
def rotIz(a) si NOT esVacío(a) AND NOT esVacío(hijoDr(a))
rotIz(Cons(a, ux, Cons(b, vy, c)))   = Cons(Cons(a, ux, b), vy, c)

def rotDr(a) si NOT esVacío(a) AND NOT esVacío(hijoIz(a))
rotDr(Cons(Cons(a, ux, b), vy, c))   = Cons(a, ux, Cons(b, vy, c))
```

Factor de equilibrio

Los algoritmos de inserción y borrado tienen que averiguar cuándo un subárbol se ha desequilibrado para proceder a reequilibrarlo con ayuda de rotaciones. Para caracterizar las distintas situaciones que pueden darse introducimos el concepto de *factor de equilibrio*, que toma uno entre tres valores posibles: equilibrado, desequilibrado a la izquierda y desequilibrado a la derecha.

Formalmente, introducimos en la especificación el tipo privado *FactEq*, con las siguientes generadoras:

```
tipo privado
   FactEq
operaciones privadas
   DI: → FactEq              /* gen */
   EQ: → FactEq              /* gen */
   DD: → FactEq              /* gen */
```

Y una operación que obtiene el factor de equilibrio de un árbol dado:

```
factEq: Arbus[Cla, Val] → FactEq /* obs */
factEq(Vacío)   = EQ
factEq(Cons(iz, ux, dr))  = DI      si talla(iz) > talla(dr)
factEq(Cons(iz, ux, dr))  = EQ      si talla(iz) = talla(dr)
factEq(Cons(iz, ux, dr))  = DD      si talla(iz) < talla(dr)
```

Como luego veremos al tratar la implementación, el factor de equilibrio se almacenará explícitamente en cada nodo. La razón es que aunque sería posible averiguar dicho factor calculando las tallas de los subárboles involucrados, este método encarecería demasiado los algoritmos.

Inserción

La idea es que este proceso, como ya indicamos antes, consiste en una inserción ordinaria seguida de reequilibrado, si éste es necesario.

Algebraicamente, se corresponde con la siguiente especificación:

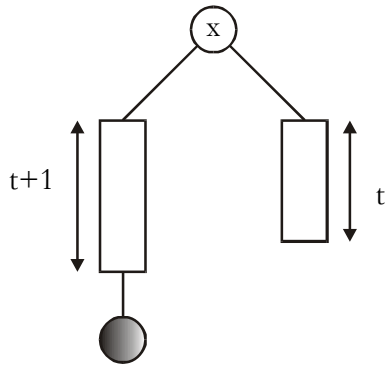
```
inserta(v, y, Vacío)      = Cons(Vacío, Par(v, y), Vacío)

inserta(v, y, Cons(iz, Par(u,x), dr)) = Cons(iz, Par(u, x⊕y), dr)
   si v == u

inserta(v, y, Cons(iz, Par(u,x), dr)) = Cons(iz', Par(u,x), dr)
```

```
        si v < u AND iz' = inserta(v, y, iz) AND talla(iz') == talla(iz)

    inserta(v, y, Cons(iz, Par(u,x), dr)) = Cons(iz', Par(u,x), dr)
        si v < u AND iz' = inserta(v, y, iz) AND talla(iz') == talla(iz) + 1
AND
           factEq(Cons(iz, Par(u,x), dr) /= DI

    inserta(v, y, Cons(iz, Par(u,x), dr)) = reeqIz(Cons(iz', Par(u,x), dr))
        si v < u AND iz' = inserta(v, y, iz) AND talla(iz') == talla(iz) + 1
AND
           factEq(Cons(iz, Par(u,x), dr) == DI   /* Iz */

    inserta(v, y, Cons(iz, Par(u,x), dr)) = Cons(iz, Par(u,x), dr')
        si v > u AND dr' = inserta(v, y, dr) AND talla(dr') == talla(dr)

    inserta(v, y, Cons(iz, Par(u,x), dr)) = Cons(iz, Par(u,x), dr')
        si v > u AND dr' = inserta(v, y, dr) AND talla(dr') == talla(dr) + 1
AND
           factEq(Cons(iz, Par(u,x), dr) /= DD

    inserta(v, y, Cons(iz, Par(u,x), dr)) = reeqDr(Cons(iz, Par(u,x), dr'))
        si v > u AND dr' = inserta(v, y, dr) AND talla(dr') == talla(dr) + 1
AND
           factEq(Cons(iz, Par(u,x), dr) == DD   /* Dr */
```

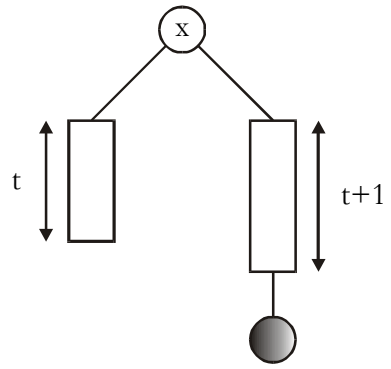
El reequilibrado se hace necesario cuando la inserción modifica la talla del subárbol donde se realiza, y el árbol original estaba desequilibrado hacia ese subárbol. Nótese que en este caso basta con reequilibrar el subárbol más próximo a la nueva hoja que se haya desequilibrado y corregir el desequilibrio con ayuda de rotaciones; ya que, por efecto de las rotaciones, el subárbol reequilibrado conserva la misma talla que tenía antes de desequilibrarse, por lo que el árbol total queda equilibrado.

Se produce desequilibrio cuando algún árbol *a* que era equilibrado antes de la inserción se convierte después de la inserción en un árbol no equilibrado *a'*. Suponemos que *a* es el subárbol más profundo en el cual la inserción causa desequilibrio. Esto sólo es posible en dos casos:

— Caso [Iz]: *a* tenía factor de equilibrio DI. La inserción se ha producido en el hijo izquierdo de *a*, y *a'* ha quedado:



— Caso [Dr]: *a* tenía factor de equilibrio DD. La inserción se ha producido en el hijo derecho de *a*, y *a'* ha quedado:



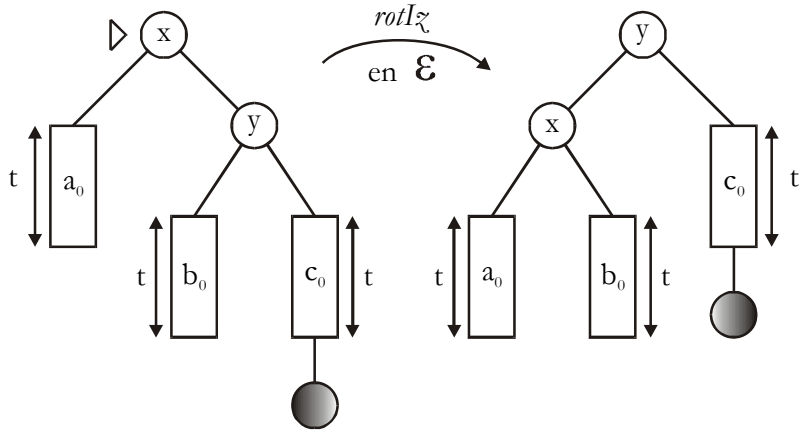
Nótese que el subárbol izquierdo, en el caso [Iz], y el subárbol derecho, en el caso [Dr], no pueden ser vacíos, pues de lo contrario la inserción no podría haber causado desequilibrio.

Vamos a tratar sólo el caso [Dr], dejando [Iz], que es simétrico, como ejercicio.

Hemos de distinguir dos situaciones, según en qué hijo del subárbol derecho se ha producido la inserción:

— [DrDr]: se ha insertado en el subárbol derecho del subárbol derecho de *a*.

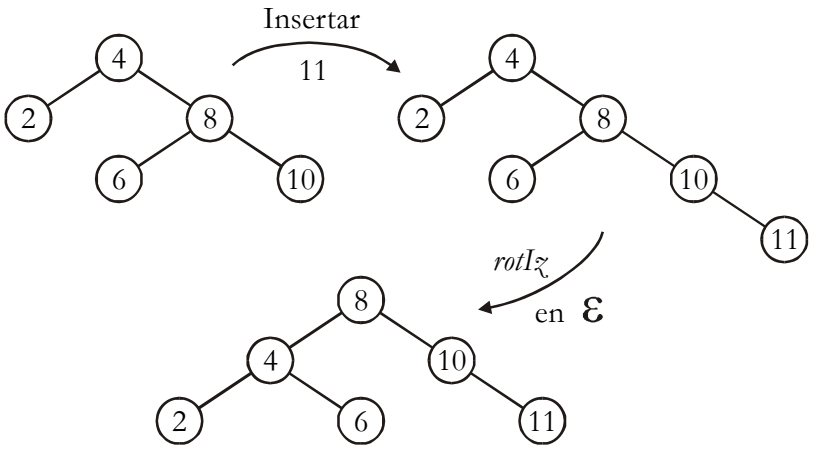
Se reequilibra realizando una rotación a la izquierda del árbol.



Nótese que b_0 debe tener talla *t*, porque el árbol considerado es el subárbol más profundo que se ha desequilibrado al insertar.

Observamos que el resultado final queda AVL con factor de equilibrio EQ y la misma talla *t+2* que el árbol original *a*. Por lo tanto, si *a* era un subárbol de un árbol mayor, también éste ha quedado equilibrado.

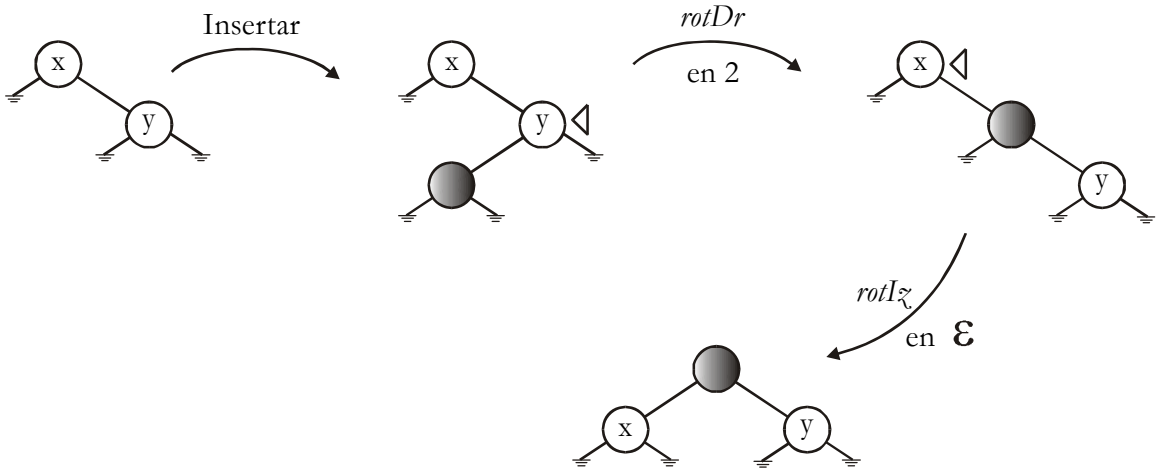
Podemos verlo aplicado a un ejemplo:



— [DrIz]: se ha insertado en el subárbol izquierdo del subárbol derecho de *a*.

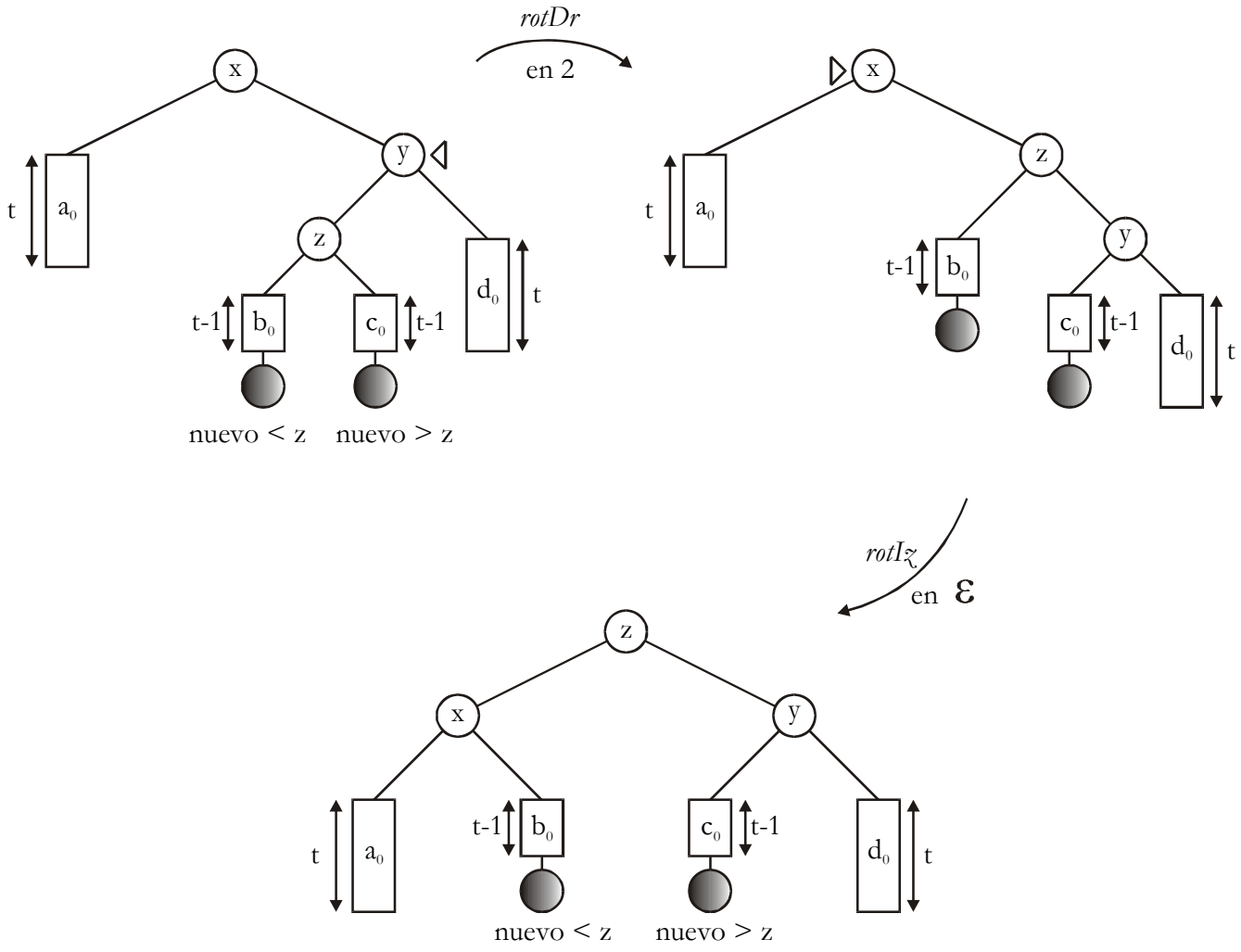
Se reequilibra haciendo una rotación a la derecha del subárbol derecho y una rotación a izquierda del resultado.

Dentro de este subcaso hay un caso trivial, cuando el hijo izquierdo del hijo derecho de *a* es vacío (como *a* estaba equilibrado, esta condición equivale a que el hijo derecho de *a* sea una hoja):



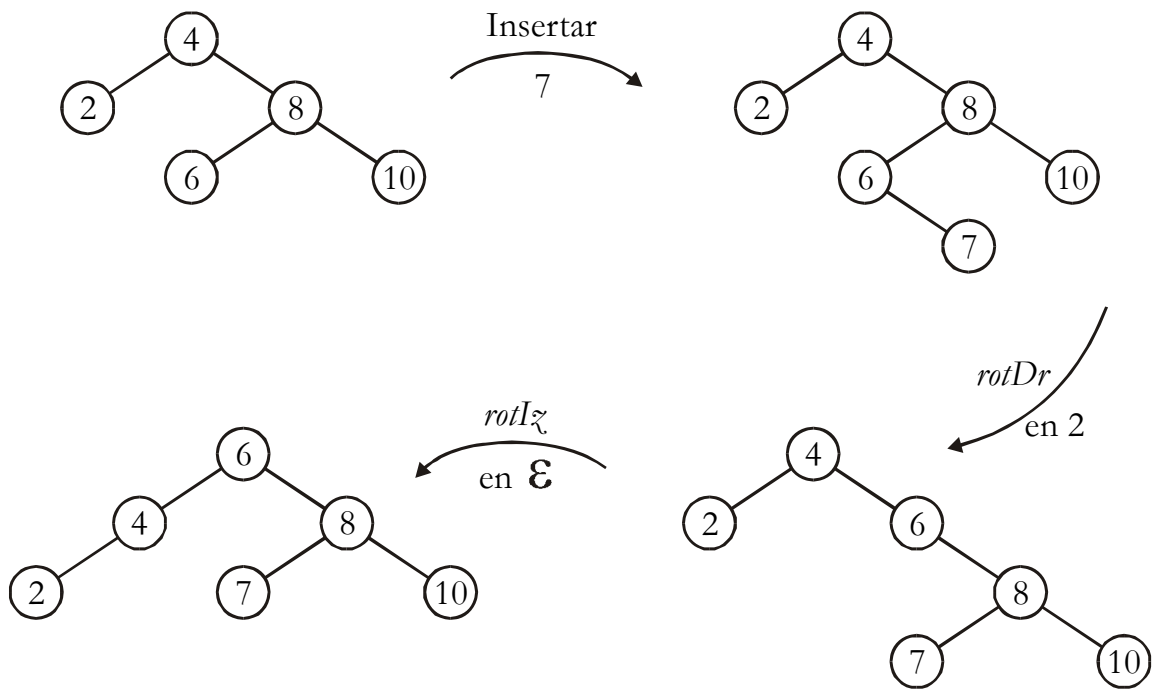
Aunque hemos hecho la transformación utilizando las dos rotaciones de la solución general, para mostrar que efectivamente dicha solución funciona, en la práctica esto se puede implementar como un caso especial que se realiza en un solo paso.

En el caso no trivial se tiene:



Tanto si el nuevo nodo ha quedado en b_0 como si ha quedado en c_0, el árbol resultante queda AVL con factor de equilibrio EQ, y la misma talla *t+2* que el árbol original *a*. Si *a* era subárbol de un árbol mayor, éste habrá quedado equilibrado.

Veamos cómo funciona en un ejemplo:



Con todo esto la especificación de la operación de reequilibrado a la derecha:

```
def reeqDr(Cons(iz, ux, dr)) si talla(dr) == talla(iz) + 2

   % Caso [DrDr]:
   reeqDr(Cons(iz, ux, dr))  =d rotIz(Cons(iz, ux, dr))  si factEq(dr) /= DI

   % Caso [DrIz]:
   reeqDr(Cons(iz, ux, dr))  =d rotIz(Cons(iz, ux, rotDr(dr)))    si
factEq(dr) == DI
```

Aunque la primera ecuación cubre los casos *factEq(dr) == DD* y *factEq(dr) == EQ*, éste último no se producirá nunca en el reequilibrado provocado por inserción, como podemos observar en la discusión anterior. Sin embargo, lo incluimos aquí porque en el caso de reequilibrado por supresión sí que puede presentarse.

El algoritmo de inserción AVL se puede programar recursivamente siguiendo este método. Conviene utilizar un procedimiento auxiliar con un parámetro ***s*** *crece? : Bool*, que informa de si una inserción ha aumentado la talla. Esta información permite reequilibrar y actualizar los factores de equilibrio. La corrección del desequilibrio debe aplicarse al primer subárbol desequilibrado que se encuentre yendo desde la nueva hoja hacia la raíz.

En cuanto a la complejidad del algoritmo de inserción, el análisis matemático aún no está resuelto del todo, pero las pruebas empíricas indican que:

— La talla media que puede esperarse para un árbol AVL generado por la inserción aleatoria de *n* claves diferentes es *0'25 + log n*.

— Un promedio de 1 de cada 2 inserciones requiere reequilibrio. Todos los tipos de desequilibrio son equiprobables, y, como acabamos de ver, un desequilibrio siempre se puede corregir con 1 o 2 rotaciones.

5.5.3 Implementación

Al igual que ocurre con los árboles de búsqueda es necesario implementar los árboles AVL con acceso al tipo representante de los árboles binarios.

Tipo representante

El tipo representante es similar al de los árboles de búsqueda, pero añadiendo el campo donde se almacena el factor de equilibrio.

```
módulo impl AVL[CLA,VAL]
  importa
    COMB, ORD
  privado
    tipo
      FactEq = DI | EQ | DD;
      Val = COMB.Elem;
      Cla = ORD.Elem;
      Nodo = reg
        feq : FactEq;
        cla : Cla;
        val : Val;
        hi, hd : Enlace
      freg;
      Enlace = puntero a Nodo;
      Arbus[Cla, Val] = Enlace;

% Implementación de las operaciones
fmódulo;
```

Invariante de la representación

Exigimos que cumplan el invariante de la representación de los árboles binarios sin compartición de estructura, y que el resultado del recorrido en inorden esté ordenado.

Dado *p : Enlace*

$$
\begin{aligned}
& R^{AVL}_{Arbus[Cla,Val]} \\
\Leftrightarrow_{def} \quad & \\
& R^{NC}_{Arbin[Elem]}(p) \wedge \\
& \forall\ i, j : 1 \leq i < j \leq \#inOrd(p) : inOrd(p)\ !!\ i < inOrd(p)\ !!\ j \wedge
\end{aligned}
$$

```
∀ p : Enlaces(p) : p^.feq representa correctamente
                   el factor de equilibrio de p
```

Implícitamente en la última condición estamos exigiendo que sea equilibrado en altura, por la forma cómo están definidos los factores de equilibrio.

Nótese que podemos exigir el invariante sin compartición de estructura porque en AVL no está disponible la operación *Cons*.

Función de abstracción

La misma que se utiliza con árboles binarios:

Implementación de las operaciones

A excepción de la inserción y el borrado, todas las operaciones se implementan de la misma forma que en los árboles de búsqueda. Para no extendernos sólo presentamos la implementación de algunas de las operaciones. En el texto de Franch se pueden encontrar el resto.

```
proc inserta( e d : Cla; e y : Val; es a : Arbus[Cla, Val] );
{ P₀ : R(d) ∧ R(y) ∧ R(a) ∧ a = A }
var
  crece? : Boolean;
inicio
  insertaAVL( d, y, a, crece? )
{ Q₀ R(a) ∧ A(a) =AVL[CLA,VAL] inserta( A(d), A(y), A(A) ) }
fproc
```

Procedimiento privado auxiliar de inserción

```
proc insertaAVL( e d : Cla; e y : Val; es a : Arbus[Cla, Val]; s crece? :
Bool );
{ P₀ : R(d) ∧ R(y) ∧ R(a) ∧ a = A }
inicio
  si a == nil
    entonces
      ubicar(a);
      a^.cla := d;
      a^.val := y;
      a^.hi := nil;
      a^.hd := nil;
      a^.feq := EQ;
      crece? := cierto
    sino
      si
        ORD.igual(d, a^.cla) → a^.val := COMB.combina( a^.val, y );
                               crece? := falso
      □ ORD.menor(d, a^.cla) → insertaAVL(d, y, a^.hi, crece?)
```

```
                                  si NOT crece?
                                    entonces
                                      seguir
                                  sino
                                    caso a^.feq de
                                      DD → a^.feq := EQ; crece? := falso
                                    □ EQ → a^.feq := DI
                                    □ DI → reeqIz(d, a); crece? := falso
                                    fcaso
                                  fsi
        □ ORD.mayor(d, a^.cla) → insertaAVL(d, y, a^.hd, crece?)
                                  si NOT crece?
                                    entonces
                                      seguir
                                  sino
                                    caso a^.feq de
                                      DI → a^.feq := EQ; crece? := falso
                                    □ EQ → a^.feq := DD
                                    □ DD → reeqDr(d, a); crece? := falso
                                    fcaso
                                  fsi
      fsi
    fsi
{ Q₀ : R(a) ∧ A(a) =AVL[CLA,VAL] inserta(A(d), A(y), A(A)) ∧
       crece? =AVL[CLA,VAL] (talla(A(a)) == 1 + talla(A(A))) }
fproc
```

Vemos que en este procedimiento, cuando se realiza un reequilibrado se pone la variable *crece?* a falso, con lo que ya no se producirán más reequilibrados a la vuelta de las llamadas recursivas. en la implementación de *borra* el reequilibrado debe tener otro parámetro adicional que nos indique si dicha operación ha hecho decrecer la talla del árbol, con lo que se debería propagar dicha información hacia las llamadas anteriores.

Procedimiento auxiliar privado de reequilibrado por la derecha.

```
proc reeqDr( e d : Cla; es a : Arbus[Cla, Val] );
{ P₀ : a = A ∧ RARBUSCA[CLA,VAL](a) ∧
       talla(hijoDr(A(a))) =ARBUSCA[CLA, VAL] talla(hijoIz(A(a)))+2 ∧
       d es la clave del nodo que produce el desequilibrio      }
inicio
  si a^.hi == nil AND (a^.hd)^.hd == nil
    entonces          /* desequilibrio trivial */
      reorgDr(a);
      a^.feq := EQ;
      a^.hi^.feq := EQ;
      a^.hD^.feq := EQ
```

```
      sino
        si a^.hd^.feq == DD
          entonces
            rotIz(a);
            a^.feq := EQ;
            a^.hi^.feq := EQ
          sino
            rotDr(a^.hd);
            rotIz(a);
            a^.feq := EQ;
            si
            □ ORD.menor(d, a^.cla) → a^.hi^.feq := EQ;
                                     a^.hd^.feq := DD
            □ ORD.mayor(d, a^.cla) → a^.hi^.feq := DI;
                                     a^.hd^.feq := EQ
            fsi
        fsi
    fsi
  { Q₀ : R_AVL[CLA,VAL](a) ∧ recorre(A(a)) =_ARBUSCA[CLA,VAL] recorre(A(A)) }
  fproc
```

Procedimiento auxiliar privado de reorganización a la derecha:

```
  proc reorgDr ( es a : Arbus[Cla, Val] );
  { P₀ : a = A ∧ R_ARBUS[CLA,VAL](a) ∧ a^.hi == nil ∧ a^.hd /= nil ∧
         a^.hd^.hd == nil ∧ a^.hd^.hi /= nil ∧
         a^.hd^.hi^.hi == nil ∧ a^.hd^.hi^.hd == nil }
  var
    aux : Enlace;
  inicio
    aux := a^.hd^.hi;
    aux^.hi := a;
    aux^.hd := a^.hd;
    aux^.hi^.hd := nil;
    aux^.hd^.hi := nil;
    a := aux
  { Q₀ : R_AVL[CLA,VAL] (a) ∧ recorre(A(a)) =_ARBUS[CLA,VAL] recorre(A(A)) }
  fproc
```

Procedimiento auxiliar privado de rotación a la derecha

```
  proc rotDr( es arb : Arbus[Cla, Val] );
  { P₀ : a representa un árbol de la forma Cons(Cons(a, ux, b), vy, c) }
  var
    aux : Enlace;
  inicio
    aux := arb;
```

```
    arb := arb^.hi;
    arb^.hi := arb^.hd;
    arb^.hd := aux
{ Q₀ : arb representa el árbol Cons(a, ux, Cons(b, vy, c)),
       siendo a, ux, b, vy, c los mismo de P₀                }
fproc
```

Borrado

La idea es que este proceso, como ya indicamos antes, consiste en un borrado ordinario seguida de reequilibrado, si éste es necesario.

Mediante ecuaciones, podemos especificar esta operación de la siguiente forma:

```
borra(v, Vacío)   = Vacío

borra(v, Cons(iz, Par(u,x), dr))    = quitaRaíz(Cons(iz, Par(u,x), dr))
   si v == u

borra(v, Cons(iz, Par(u,x), dr))    = Cons(iz', Par(u,x), dr)
   si v < u AND iz' = borra(v, iz) AND talla(iz') == talla(iz)

borra(v, Cons(iz, Par(u,x), dr))    = Cons(iz', Par(u,x), dr)
   si v < u AND iz' = borra(v, iz) AND talla(iz') == talla(iz) - 1 AND
      factEq(Cons(iz, Par(u,x), dr) /= DD

borra(v, Cons(iz, Par(u,x), dr))    = reeqDr(Cons(iz', Par(u,x), dr))
   si v < u AND iz' = borra(v, iz) AND talla(iz') == talla(iz) - 1 AND
      factEq(Cons(iz, Par(u,x), dr) == DD   /* Dr */

borra(v, Cons(iz, Par(u,x), dr))    = Cons(iz, Par(u,x), dr')
   si v > u AND dr' = borra(v, dr) AND talla(dr') == talla(dr)

borra(v, Cons(iz, Par(u,x), dr))    = Cons(iz, Par(u,x), dr')
   si v > u AND dr' = borra(v, dr) AND talla(dr') == talla(dr) - 1 AND
      factEq(Cons(iz, Par(u,x), dr) /= DI

borra(v, Cons(iz, Par(u,x), dr))    = reeqIz(Cons(iz, Par(u,x), dr'))
   si v > u AND dr' = borra(v, dr) AND talla(dr') == talla(dr) - 1 AND
      factEq(Cons(iz, Par(u,x), dr) == DI   /* Iz */
```

La operación que se encarga de eliminar la raíz de un subárbol:

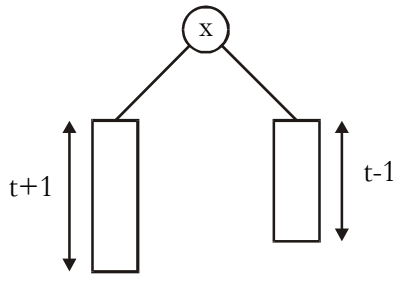
```
def quitaRaíz(Cons(iz, ux, dr))
quitaRaíz(Cons(iz, ux, dr))    = dr
   si esVacío(iz)

quitaRaíz(Cons(iz, ux, dr))    = iz
```

```
   si esVacío(dr)

quitaRaíz(Cons(iz, ux, dr))    = Cons(iz, Par(v,y), dr')
   si NOT esVacío(iz) AND NOT esVacío(dr) AND Par(v,y) = min(dr) AND
      dr' = borra(v, dr) AND
      talla(dr') == talla(dr)

quitaRaíz(Cons(iz, ux, dr))    = Cons(iz, Par(v,y), dr')
   si NOT esVacío(iz) AND NOT esVacío(dr) AND Par(v,y) = min(dr) AND
      dr' = borra(v, dr) AND
      talla(dr') == talla(dr) - 1 AND factEq(Cons(iz, ux, dr)) /= DI

quitaRaíz(Cons(iz, ux, dr))    = reeqIz(Cons(iz, Par(v,y), dr'))
   si NOT esVacío(iz) AND NOT esVacío(dr) AND Par(v,y) = min(dr) AND
      dr' = borra(v, dr) AND
      talla(dr') == talla(dr) - 1 AND factEq(Cons(iz, ux, dr)) == DI
```

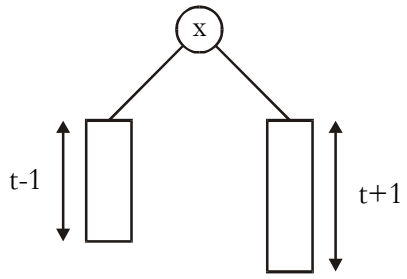
La idea del reequilibrado consiste ahora en localizar el subárbol más profundo a lo largo del camino de búsqueda que se haya desequilibrado, y corregir el desequilibrio con ayuda de rotaciones. Puede suceder ahora que el reequilibrio produzca un subárbol cuya talla sea una unidad menor que la que había antes del borrado, causándose un nuevo desequilibrio. En este caso el proceso de reequilibrio tiene que reiterarse.

Se produce desequilibrio cuando algún árbol *a* que era equilibrado antes del borrado se convierte después de la inserción en un árbol no equilibrado *a'*. Suponemos que *a* es el subárbol más profundo en el cual el borrado causa desequilibrio. Esto sólo es posible en dos casos:

— Caso [Iz]: *a* tenía factor de equilibrio DI. El borrado se ha producido en el hijo derecho de *a*, y *a'* ha quedado:

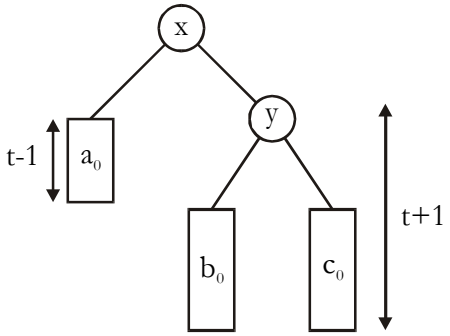


— Caso [Dr]: *a* tenía factor de equilibrio DD. El borrado se ha producido en el hijo izquierdo de *a*, y *a'* ha quedado:



Estudiamos el caso [Dr], dejando [Iz] que es simétrico como ejercicio.

En el caso [Dr], el hijo derecho del árbol inicial *a* tiene que ser no vacío y con dos o más nodos, ya que en caso contrario un borrado en el hijo izquierdo, no podría haber causado desequilibrio. Tenemos pues que *a'* es de la forma:

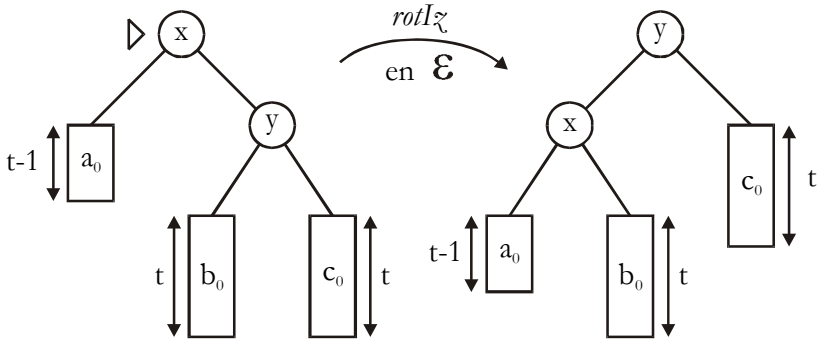


Hacemos la distinción de casos dependiendo de la talla de los subárboles b_0 y c_0.

— $talla(b_0) = t$, $talla(c_0) = t$

Es decir, tenemos que el factor de equilibrio del hijo derecho de *a'* es EQ.

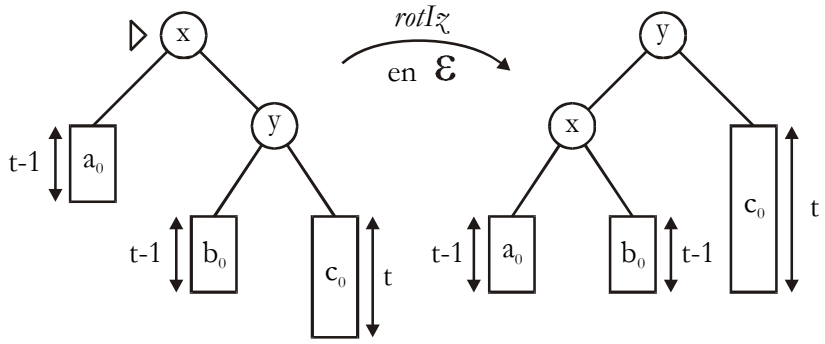
El árbol *a'* puede reequilibrarse como en el caso [DrDr] de la inserción AVL:



El árbol resultante es AVL con factor de equilibrio DI y la misma talla t+2 del árbol *a* original. Si *a* era un subárbol de un árbol mayor, éste también ha quedado reequilibrado.

— *talla(b_0) = t–1, talla(c_0) = t*

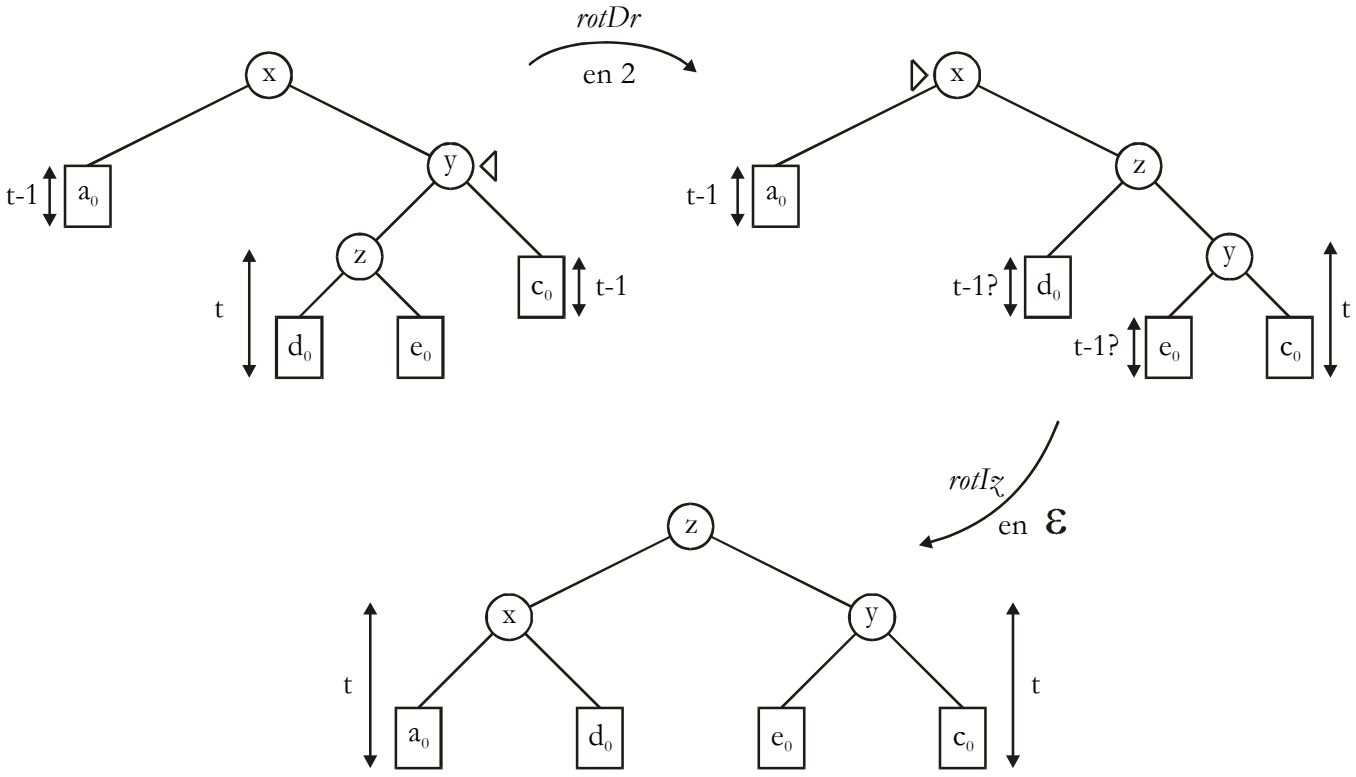
Es decir, tenemos que el factor de equilibrio del hijo derecho de *a'* es DD.
El reequilibrio se logra con la misma rotación del caso anterior.



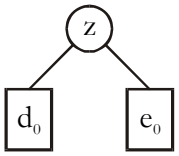
En este caso, el árbol AVL resultante tiene factor de equilibrio EQ y talla *t+1*, una unidad menos que *a*. Por lo tanto, si *a* era subárbol de un árbol mayor, éste puede necesitar aún ser reequilibrado.

— *talla(b_0) = t, talla(c_0) = t–1*

Es decir, tenemos que el factor de equilibrio del hijo derecho de *a'* es DI. El reequilibrio puede lograrse con dos rotaciones, como en el caso [DrIz] de la inserción AVL



Donde hay combinaciones posibles para las tallas de d_0 y e_0. Si *f* es el factor de equilibrio de



se tiene

f	*talla*(d_0)	*talla*(e_0)
EQ	*t–1*	*t–1*
DI	*t–1*	*t–2*
DD	*t–2*	*t–1*

Nótese que al menos uno de los dos árboles d_0, e_0 debe tener talla *t–1*, con lo cual es seguro que el árbol resultante de las dos rotaciones es AVL con factor de equilibrio EQ y talla *t+1*, una unidad menor que la del árbol original *a*. Si *a* era un subárbol de otro árbol mayor, éste puede seguir necesitando reequilibrado.

Observamos, por lo tanto, que la operación de reequilibrado es prácticamente igual que en el caso del reequilibrado provocado por una inserción. La única diferencia radica en que ahora hay un caso más: *fatEq(dr)* = *EQ*. A nivel de implementación también podemos establecer diferencias porque en el reequilibrado por inserción no hace falta considerar el caso adicional, y porque en el reequilibrado por borrado nos hace falta un parámetro adicional que indique si la talla del árbol ha disminuido. Aunque, también existe la posibilidad de hacer una única implementación.

El algoritmo de borrado AVL se puede programar recursivamente siguiendo este método. conviene utilizar un procedimiento auxiliar con un parámetro ***s*** *encoge? : Bool*, que informa de si un borrado ha disminuido la talla. Esta información permite reequilibrar y actualizar los factores de equilibrio.

Para terminar con este apartado dedicado a las operaciones de inserción y borrado en árboles AVL, indicar que existe una diferencia fundamental entre ellas:

— En cada inserción AVL, hay que reequilibrar a los sumo 1 subárbol (1 o 2 rotaciones).

— En un borrado AVL, puede llegar a ser necesario reequilibrar todos los subárboles encontrados a lo largo de la trayectoria de búsqueda, que tiene una longitud *O*(log n), si *n* s el número de nodos. Cada reequilibrado requiere 1 o 2 rotaciones. Los árboles de Fibonacci dan lugar a este caso pésimo en el comportamiento del borrado.

Sorprendentemente, las pruebas empíricas indican que en promedio se efectúan:

— 1 rotación cada 2 inserciones

— 1 rotación cada 5 borrados.

5.6 Ejercicios

Arboles: modelo matemático y especificación

238. Dibuja los árboles siguientes:

*(a) Tu árbol genealógico.

†(b) El árbol taxonómico de los seres vivos.

(c) Un árbol que represente la organización del texto de Xavier Franch *Estructuras de datos: especificación, diseño e implementación*, con sus divisiones en capítulos, secciones y subsecciones.

†(d) Un árbol que represente la organización jerárquica del sistema de directorios y archivos en el selvático PC del profesor Tadeo de la Tecla.

(e) El árbol de llamadas inducido por la llamada inicial *fib(4)* (cfr. ejercicio 83).

(f) El árbol de análisis sintáctico de la expresión aritmética (2+x)(y–3)

*(g) El árbol de análisis sintáctico de la siguiente expresión condicional:

```
si x < y
  entonces
    x := (2+x)*(y-3)
  sino
    y := 2*x-3*y
fsi
```

†239. Los *árboles generales* se caracterizan porque cada nodo tiene un número arbitrario de hijos, ordenados en secuencia. Construye un TAD parametrizado ARBOL[E :: ANY] que represente el comportamiento de los árboles generales, con operaciones adecuadas para construirlos y acceder a los datos almacenados en sus nodos.

240. Muestra mediante un ejemplo que los dos conceptos siguientes no son equivalentes:

(a) *Arbol general de grado dos*: Arbol general con la propiedad de que cada nodo tiene como máximo dos hijos.

(b) *Arbol binario*: Arbol con la propiedad de que cada nodo tiene exactamente dos hijos, *izquierdo* y *derecho*. Se admite el *árbol vacío*, sin ningún nodo.

†241. Construye un TAD parametrizado ARBIN[E :: ANY] que especifique formalmente el comportamiento de los árboles binarios.

242. Partiendo de la especificación del ejercicio anterior, añade ecuaciones que formalicen el comportamiento de las operaciones siguientes:

- `talla: Arbin[Elem] → Nat`
  Calcula la talla (altura, profundidad) de un árbol binario, definida como el número de nodos de la rama más larga.
- `numNodos: Arbin[Elem] → Nat`
  Calcula el número de nodos de un árbol binario.
- `numHojas: Arbin[Elem] → Nat`
  Calcula el número de hojas de un árbol binario.

243. Sea un árbol binario de talla *n*. Se define:

(C) *a* es *completo* syss todos sus nodos internos tienen dos hijos no vacíos, y todas sus hojas están en el nivel *n*.

(S) *a* es *semicompleto* syss *a* es completo o tiene vacantes una serie de posiciones consecutivas del nivel *n*, de manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.

Se pide:

(a) Dibuja ejemplos de árboles binarios completos, semicompletos y no semicompletos.

(b) Demuestra que un árbol binario completo de talla $t \geq 1$ tiene 2^{t-1} hojas y un total de 2^t-1 nodos.

†244. Suponiendo conocidas las operaciones del TAD ARBIN y la operación *talla* del ejercicio 242, construye ecuaciones que especifiquen formalmente el comportamiento de las operaciones siguientes:

- `esCompleto: Arbin[Elem] → Bool`
  Reconoce si un árbol binario dado es completo.
- `esSemiCompleto: Arbin[Elem] → Bool`
  Reconoce si un árbol binario dado es semicompleto.

245. Un árbol general *a* siempre se puede representar por medio de un árbol binario *b*, construido según la idea siguiente

- *a* y *b* tienen el mismo número de nodos. Cada nodo α de *a* está representado por un nodo β de *b*.
- El paso de un nodo α a su primer hijo en *a* se corresponde con el paso de β a su hijo izquierdo en *b*.
- El paso de un nodo α a su hermano derecho en *a* se corresponde con el paso de β a su hijo derecho en *b*.

Dibuja un árbol general con 10 nodos, que no sea un árbol binario, y dibuja su representación como árbol binario, siguiendo la idea anterior.

246. La idea del ejercicio anterior puede extenderse a la representación de un *bosque as* de árboles generales, por medio de un árbol binario *b*. Basta construir uno por uno los árboles binarios que representan los diferentes árboles del bosque *as*, y "enganchar" cada uno de ellos como hijo derecho del precedente. Dibuja un ejemplo de esta construcción.

†247. Formaliza las ideas de los dos ejercicios anteriores, especificando por medio de ecuaciones las dos operaciones siguientes:

- `hazArbin: Bosque[Elem] → Arbin[Elem]`
  Construye el árbol binario que representa un bosque dado.
- `hazBosque: Arbin[Elem] → Bosque[Elem]`
  Construye el bosque representado por un árbol binario dado.

†248. Las operaciones *hazArbin* y *hazBosque* son inversas una de otra. Verifícalo demostrando por inducción lo que sigue:

(a) $\forall$ `as : Bosque[Elem] : hazBosque(hazArbin(as)) = as`

```
(b) ∀ b  : Arbin[Elem]  : hazArbin(hazBosque(b))  = b
```

Arboles. Técnicas de implementación

249. Diseña una representación dinámica para el TAD ARBIN. Formaliza el invariante de la representación considerando dos variantes: R^{NC}, que prohibe que diferentes subárboles de un mismo árbol compartan estructura; y *R*, que no lo prohibe. Formaliza también la función de abstracción.

250. Desarrolla una implementación de ARBIN basada en la representación del ejercicio anterior. Comprueba que el coste temporal de todas las operaciones es $O(1)$ y que el espacio ocupado por la representación es $O(n)$, siendo n el número de nodos del árbol.

251. La implementación de ARBIN discutida en el ejercicio anterior puede generar estructuras compartidas. compruébalo dibujando un gráfico que represente las estructuras representantes de *a1*, *a2*, *a3* al finalizar la ejecución del siguiente fragmento de programa:

```
var
  a1, a2, a3 : Arbin[Ent];
inicio
  a1 := Cons(Vacío(), 1, Vacío());
  a2 := Cons(Vacío(), 2, Vacío());
  a3 := Cons(a1, 3, a2);
  a1 := Cons(a1, 4, a3)
```

252. Programa el procedimiento y la función que se especifican a continuación:

```
(a) proc anula ( es a : Arbin[Elem] );
    { P₀ : R^NC(a) ∧ a = A }
    { Q₀ : a = nil ∧
           el espacio que ocupaba la estructura representante
           de A se ha liberado }
    fproc

(b) func copia ( a : Arbin[Elem] ) dev b : Arbin[Elem];
    { P₀ : R(a) }
    { Q₀ : R^NC(b) ∧ A(a) = A(b) ∧
           la estructura representante de b está ubicado en espacio nuevo }
    ffunc
```

Nota: Para programa *anula* y *copia* es necesario tener acceso a la representación dinámica de los árboles. En la práctica, convendría que el módulo de implementación de ARBIN exportase estos procedimientos.

253. Enriquece la especificación del TAD ARBIN con una operación de igualdad

```
( == ): (Arbin[Elem], Arbin[Elem]) → Bool
```

Extiende la implementación del ejercicio 250 añadiendo una función que implemente (==) y analiza su coste temporal.

†254. La implementación dinámica de ARBIN abordada en el ejercicio 250 se puede realizar reemplazando la memoria dinámica por un vector de tipo adecuado que la simule. Estudia los detalles en el texto de Xavier Franch (subsección 5.2.1, pp. 229-232) y desarrolla la implementación resultante.

†255. Demuestra que la definición recursiva dada a continuación establece una aplicación biyectiva *índice* ; $\{1,2\}^* \to \aleph_+$ entre el conjunto de todas las posiciones posibles de los árboles binarios y el conjunto de los números naturales positivos.

```
índice( ε )     = 1
índice( α.1 )   = 2 * índice(α)
índice( α.2 )   = 2 * índice(α) + 1
```

256. Para representar un árbol binario de tipo *Arbin[Elem]* puede usarse un vector de tipo `Vector[1..max] de Elem`, siguiendo el criterio de que el elemento que ocupa la posición α del árbol se almacene en el lugar *índice(*α*)* del vector. Esta representación resulta útil para algunos algoritmos que operan con árboles semicompletos. Según el resultado del ejercicio 243(b), un valor max = 2^t –1 basta para representar cualquier árbol semicompleto de talla menor o igual que *t*. Plantea las fórmulas numéricas que habría que utilizar para calcular el padre y los hijos de un nodo dado, usando esta representación.

†257. Para desarrollar una implementación dinámica de los árboles generales, suele utilizarse la representación de los árboles generales como árboles binarios estudiada más arriba en el ejercicio 245. Estudia los detalles de esta implementación en el texto de Xavier Franch, subsección 5.2.2., pp. 234-237.

258. Para desarrollar una implementación dinámica de los árboles n-arios hay dos opciones posibles:

(a) Usar nodos que incluyan *n* punteros a los *n* hijos.

(b) Usar nodos que incluyan dos punteros señalando al primer hijo y al hermano derecho, respectivamente.

La opción (b) se basaría en utilizar una representación de los árboles n-arios como árboles binarios, al estilo estudiado en el ejercicio 245. compara las dos opciones desde el punto de vista del espacio ocupado por la estructura representante del árbol.

259. Define los conceptos de *árbol n-ario completo* y *árbol n-ario semicompleto*. Enuncia y demuestra para esta clase de árboles los resultados análogos a los obtenidos en el ejercicio 243(b).

260. Extiende los resultados de los ejercicios 255 y 256 al caso de los árboles n-arios.

Arboles binarios. Ejemplos elementales de algoritmos

Para resolver los ejercicios 261-264 nos situamos como clientes de un módulo que implemente el TAD ARBIN, y no tenemos acceso a la representación interna de los árboles.

261. Programa funciones recursivas que implementen las operaciones del ejercicio 242.

262. Especifica y programa una función recursiva *espejo* que construya la imagen especular de un árbol binario dado como parámetro.

263. Especifica y programa una función recursiva que calcule la *frontera* de un árbol dado como parámetro. Por frontera se entiende la lista formada por los elementos almacenados en las hojas del árbol, tomados de izquierda a derecha. Se supone disponible un módulo que implementa el TAD LISTA.

264. Programa un función recursiva que calcule el valor numérico de una expresión aritmética a partir de un árbol dado que represente la estructura sintáctica de la expresión.

Recorridos de árboles

265. Construye las listas resultantes de aplicar los diferentes recorridos posibles al árbol binario que representa la estructura de la expresión aritmética x*(–y)+(–x)*y.

266. Especifica algebraicamente los tres tipos de recorridos en profundidad de árboles binarios, planteados como operaciones que convierten un árbol en una lista, con perfil Arbin[Elem] → Lista[Elem]. Presenta la especificación resultante como enriquecimiento del TAD ARBIN.

267. Suponiendo disponibles dos módulos ARBIN y LISTA que implementen los TADs del mismo nombre, y sin acceder a la representación, programa funciones recursivas que realicen las operaciones de recorrido especificadas en el ejercicio anterior.

268. El recorrido de un árbol binario puede realizarse también con ayuda de un procedimiento recursivo del estilo

```
proc recorre ( e a : Arbin[Elem]; es xs : Sec[Elem] );
{ P₀ : xs = XS ∧ fin?(xs) = cierto }
{ Q₀ : fin?(xs) = Cierto ∧ cont(xs) = cont(XS) ++ recorrido(a) }
fproc
```

Desarrolla esta idea para los tres tipos de recorrido en profundidad.

269. Los recorridos en preorden y postorden también tienen sentido para árboles generales. Especifícalos algebraicamente por medio de una extensión del TAD ARBOL que contenga ecuaciones para dos nuevas operaciones:

```
preAG, postAG : Arbol[Elem] → Lista[Elem]
```

Nota: Debido a la dependencia mutua entre árboles generales y bosques, deberás especificar también dos operaciones auxiliares que describen el recorrido de bosques:

```
preBos, postBos: Bosque[Elem] → Lista[Elem]
```

270. Supongamos que R : Arbin[Elem] → Lista[Elem] sea una cualquiera de las tres operaciones de recorrido en profundidad de árboles binarios. Especifica por medio de ecuaciones una nueva operación

```
acumula_R : Pila[Arbin[Elem]] → Lista[Elem]
```

tal que $acumula_R(as)$ construya la concatenación de todos los recorridos $R(a)$ correspondientes a los árboles *a* apilados en *as*, empezando por la cima.

†271. Los recorridos en profundidad de árboles binarios pueden realizarse con algoritmos iterativos, manteniendo para el bucle principal un invariante de la forma:

$$R(a) = xs \mathbin{+\!\!+} acumula_R(as) \wedge \text{todos los árboles de as son no vacíos}$$

donde *R* es la operación de recorrido de que se trate, *a* es el árbol dado para recorrer, *xs* es la lista que debe contener al final el recorrido, y *as* es una *pila de árboles* auxiliar. Construye funciones iterativas que realicen esta idea para los tres tipos de recorrido. ¡Busca una inicialización adecuada para *xs* y *as*!

†272. Sea R : Arbin[Elem] → Lista[Elem] una cualquiera de las tres operaciones de recorrido en profundidad de árboles binarios. Considera la operación más general R' : (Pila[Arbin[Elem]], Lista[Elem]) → Lista[Elem] especificada por la ecuación

$$R'(as, xs) = xs \mathbin{+\!\!+} acumula_R(as)$$

siendo $acumula_R$ la operación definida en el ejercicio 270. Usa la técnica de plegado-desplegado para derivar un algoritmo recursivo final para *R'*, apoyándote en las especificaciones de $acumula_R$ y *R*. Comprueba que la transformación de los algoritmos recursivos finales así obtenidos a forma iterativa conduce a los algoritmos iterativos del ejercicio 271.

†273. Especifica algebraicamente el recorrido por niveles de un árbol binario, planteado como operación *niveles* : Arbin[Elem] → Lista[Elem]. Presenta la especificación resultante como enriqueci-miento del TAD ARBIN.

Sugerencia: Usa el TAD COLA[ARBIN[Elem]] y una operación auxiliar privada más general *nivelesCola*: Cola[Arbin[Elem]] → Lista[Elem], especificada de manera que se tenga:

$$niveles(a) = nivelesCola(as)$$

en el caso particular de que *as* sea la cola unitaria formada por *a*.

274. Programa una función iterativa que realice la operación de recorrido por niveles de un árbol binario, manteniendo para el bucle un invariante de la forma

$$niveles(a) = xs \mathbin{+\!\!+} nivelesCola(as) \wedge \text{todos los árboles de as son no vacíos}$$

donde *a* es el árbol dado para recorrer, *xs* es la lista que debe contener al final el recorrido, y *as* es una *cola de árboles* auxiliar. ¡Busca una inicialización adecuada para *xs* y *as*!

275. Modifica los algoritmos iterativos de recorrido de los ejercicios 271 y 274, convirtiéndolos en procedimientos iterativos con una especificación del estilo indicado en el ejercicio 268, de manera que el resultado del recorrido quede en una secuencia.

Arboles binarios hilvanados

276. Usando inducción sobre n ≥ 0, demuestra que un árbol binario con *n* nodos tiene 2*n*+1 subárboles, de los cuáles *n*+1 son vacíos.

Nota: los subárboles de un árbol se corresponden con los punteros que aparecen en la estructura que lo representa usando memoria dinámica. Por lo tanto, una representación dinámica del árbol *sin estructura compartida* incluirá *n*+1 punteros vacíos que "se desaprovechan".

†277. Estudia el tema relativo a los *árboles binarios hilvanados* en la sección 5.3.2 del texto de Xavier Franch. Se trata de una representación dinámica más sofisticada de los árboles binarios que "aprovecha" los punteros que quedan vacíos en la representación dinámica ordinaria (cfr. ejercicio 249), de modo que los algoritmos iterativos de recorrido pueden programarse eficientemente sin ayuda de una pila auxiliar.

Aplicaciones de los recorridos de los árboles

278. Programa un procedimiento iterativo que convierta una expresión aritmética representada como *árbol de símbolos*, en su forma postfija representada como *secuencia de símbolos*. Se supone que todo símbolo es o bien un *operador* o bien un *operando*. Puedes suponer disponible una operación *esOperador*: Símbolo → Bool que reconoce si un símbolo es un operador. El algoritmo empleado por el procedimiento que se pide, ¿corresponde a alguno de los tipos de recorrido de árboles binarios? ¿A cuál?

Nota: Combinándolo con el ejercicio 227, este ejercicio proporciona un método útil para la *evaluación* de expresiones aritméticas.

†279. Programa una función que convierta una expresión aritmética dada como *secuencia de símbolos* en un *árbol de símbolos* que represente la estructura sintáctica de la expresión. Supón que los símbolos componentes de la expresión dada son *operadores*, *operandos* y *paréntesis*, considerando para los operadores las prioridades habituales y asociatividad por la izquierda (excepto el operador de exponenciación, que asociará por la derecha).

Sugerencia: consulta la sección 7.1.1 del texto de Xavier Franch, donde se resuelve un problema similar a éste.

280. Demuestra por medio de un ejemplo que dos árboles binarios diferentes pueden tener el mismo recorrido en preorden. Sin embargo, sí es posible reconstruir un árbol binario *a* si se conoce una lista de parejas

$$ps = [(x_1, t_1), \ldots, (x_n, t_n)]$$

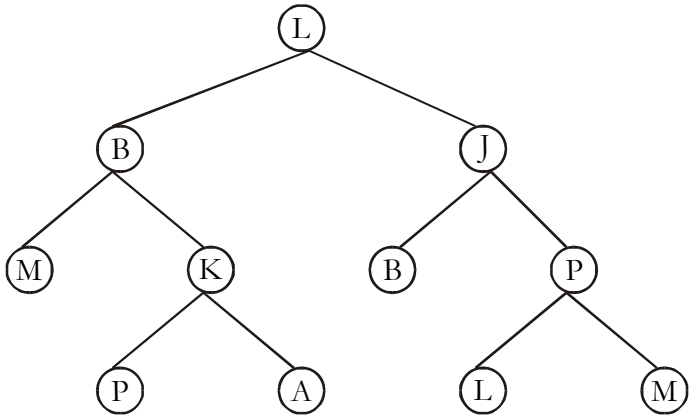
donde x_i es el elemento del nodo visitado en i-ésimo lugar durante el recorrido en preorden, y t_i es el número de nodos del hijo izquierdo de dicho nodo. Construye algoritmos que realicen la reconstrucción del árbol *a* partiendo de *ps*, en los dos supuestos siguientes:

(a) Que *ps* venga dada como lista, como se acaba de indicar.

(b) Que *ps* venga dada como secuencia (i.e. lista con punto de interés).

281. El problema de *búsqueda en profundidad* de un árbol binario consiste en lo siguiente: Dados *a : Arbin[Elem]* y *x* : *Elem*, queremos calcular la posición de *a* en la que se encuentra por primera vez el dato *x* al efectuar un recorrido de *a* en preorden. Especifica y programa una función iterativa *buscaProf* que resuelva este problema, usando una lista de valores enteros (1

y 2) para representar la posición que se devuelve como resultado. Por ejemplo, si *a* es el árbol de caracteres mostrado en la figura que sigue, y *x* es el carácter 'P', la función *buscaProf* deberá devolver la lista [1,2,1], representando la posición 1.2.1. Para resolver este problema, debes suponer disponible una operación de igualdad entre datos de tipo *Elem*.



282. El problema de *búsqueda por niveles* de un árbol binario es análogo al problema de búsqueda en profundidad, con la diferencia de que en este caso se desea localizar la primera posición en la que aparece *x* al efectuar el recorrido por niveles de *a*. Por ejemplo, si *a* es el árbol de caracteres mostrado en el ejercicio anterior y *x* es el carácter 'P', el resultado de la búsqueda por niveles debe ser la lista [1,1], que representa la posición 1.1. Especifica y programa una función iterativa *buscaNiv* que resuelva el problema de la búsqueda por niveles.

283. Un *árbol de codificación* es un árbol binario *a* que almacena en cada una de sus hojas un carácter diferente. La información almacenada en los nodos internos se considera irrelevante. Si un cierto carácter *c* se encuentra almacenado en la hoja de posición α, se considera que α es el *código* asignado a *c* por el árbol de codificación *a*. Más en general, el código de cualquier cadena de caracteres dada se puede construir concatenando los códigos de los caracteres que la forman, respetando su orden de aparición.

(a) Dibuja el árbol de codificación correspondiente al código siguiente:

Carácter	Código
'A'	1.1
'T'	1.2
'G'	2.1.1
'R'	2.1.2
'E'	2.2

(b) Construye el resultado de codificar la cadena de caracteres "RETA" utilizando el código representado por el árbol de codificación anterior.

(c) Descifra 1.2.1.1.2.1.2.1.2.1.1 usando el código que estamos utilizando en estos ejemplos, construyendo la cadena de caracteres correspondiente.

284. Suponemos disponibles módulos que exportan árboles binarios de caracteres, listas de enteros y listas de caracteres, con las operaciones básicas adecuadas. Convenimos en llamar *texto* a una lista de caracteres, y *código* a una lista de enteros en la que sólo aparezcan los enteros 1 y 2. Construye funciones que resuelvan los problemas siguientes:

(a) *Decodificación de un carácter*: Dados un árbol de codificación *a* y un código *us*, calcular el primer carácter *x* de la cadena de caracteres codificada por *us*, así como el código *vs* que queda pendiente de descifrar.

(b) *Decodificación de un texto*: Dados un árbol de codificación *a* y un código *us*, calcular el texto *xs* resultante de descifrar *us*.
Sugerencia: Aplicar reiteradamente la función del apartado anterior.

(c) *Codificación de un carácter*: Dados un árbol de codificación *a* y un carácter *x*, calcular el código *us* de *x*.
Sugerencia: Usar el algoritmo de búsqueda en profundidad del ejercicio 283.

(d) *Codificación de un texto*: Dados un árbol de codificación *a* y un texto *xs*, construir el código de *xs*.
Sugerencia: Aplicar reiteradamente la función del apartado anterior.

285. Especifica un TAD HARBIN[E :: ANY] adecuado para representar árboles binarios que sólo almacenen datos en sus hojas, y desarrolla una implementación dinámica eficiente, representando los árboles por medio de estructuras con dos clases de nodos: nodos hoja, con un campo de tipo *Elem*; y nodos internos, con dos campos de tipo *Enlace*.
Sugerencia: Para la especificación, utiliza dos operaciones generadoras:

```
Hoja: Elem → HArbin[Elem]
Nodo: (HArbin[Elem], HArbin[Elem]) → HArbin[Elem]
```

Eliminación de la recursión doble

†286. Una función recursiva doble cuya definición se ajuste al esquema:

```
func f( x : T ) dev y : S;
{ P₀ : P(x); Cota t(x) }
var
  x₁', x₂' : T;
  y₁', y₂' : S;
% Otras posibles declaraciones locales
inicio
  si d(x)
    entonces
      { P(x) ∧ d(x) }
      y := r(x)
      { Q(x, y) }
    sino
      { P(x) ∧ ¬d(x) }
      x₁' := s₁(x); x₂' := s₂(x);
      { x₁' = s₁(x) ∧ x₂' = s₂(x) ∧ P(x₁') ∧ P(x₂') }
      y₁' := f(x₁'); y₂' := f(x₂');
      { x₁' = s₁(x) ∧ x₂' = s₂(x) ∧ Q(x₁', y₁') ∧ Q(x₂', y₂') }
      y := c(x, y₁', y₂')
      { Q(x, y) }
  fsi
```

```
{ Q₀ : Q(x, y) }
  dev y
ffunc
```

Se puede transformar en una función iterativa equivalente definida como sigue:

```
func f_it ( x : T ) dev y : S;
{ P₀ : P(x) }
tipo
  Marca = [0..2];
  Nodo = reg
    marca  : Marca;
    param  : T;
    result : S
  freg;
var
  pila : Pila[Nodo];
  nuevoNodo, nodo : Nodo;
  m : Marca;
  u : T;
  v : S;
inicio
  PilaVacía(pila);
  nuevoNodo.marca := 0;
  nuevoNodo.param := x;
  Apilar(nuevoNodo, pila);
  { Inv. I; Cota: C }
  it NOT esVacía(pila) →
    nodo := cima(pila);
    m := nodo.marca;
    u := nodo.param;
    si
      m = 0 AND d(u)     → y := r(u);
                           desapilar(pila)
    □ m = 0 AND NOT d(u) → nodo.marca := 1;
                           desapilar(pila);
                           Apilar(nodo, pila);
                           nuevoNodo.marca := 0;
                           nuevoNodo.param := s₁(u)
                           Apilar(nuevoNodo, pila)
    □ m = 1              → nodo.marca := 2;
                           nodo.resul := y;
                           desapilar(pila);
                           Apilar(nodo, pila);
                           nuevoNodo.marca := 0;
                           nuevoNodo.param := s₂(u)
```

```
                                Apilar(nuevoNodo, pila)
    □ m = 2                  → v := nodo.resul;
                                y := c(u, v, y);
                                desapila(pila)
    fsi
  fit;
{ y = f(x) }
{ Q₀ : Q(x, y) }
  dev y
ffunc
```

La idea de la transformación es la siguiente: la ejecución de un algoritmo doblemente recursivo se corresponde con un recorrido en postorden del árbol de llamadas determinado por la llamada inicial. En efecto: la raíz corresponde a la llamada inicial; las hojas corresponden a caso directos, y al visitarlas se obtiene un resultado con ayuda de la función *r*; y los nodos internos corresponden a casos recursivos. Para obtener el resultado, hay que calcular primero el resultado de la primera llamada, recorriendo el hijo izquierdo; a continuación, se calcula el resultado de la segunda llamada, recorriendo el hijo derecho; finalmente, al visitar el propio nodo, se componen los resultados de ambas llamadas mediante la función *c* y se obtiene el resultado de la llamada inicial.

Durante el bucle del algoritmo iterativo se van visitando nodos del árbol de llamadas, en el orden correspondiente a un recorrido en postorden. La pila representa en cada momento el camino desde la raíz del árbol (correspondiente a la llamada inicial) hasta el nodo que se está visitando en un momento dado. Las variables *y* guardan el último valor calculado para un nodo que es raíz de un subárbol ya completamente recorrido. Los nodos de la pila guardan también cierta información acerca de los resultados obtenidos en la parte de recorrido ya realizada. Más exactamente:

- El nodo cima siempre cumple *marca* ∈ [0..2]; los demás nodos verifican que *marca* ∈ [1..2].
- En el nodo del fondo de la pila se tiene *param* = *x* (parámetros de la llamada inicial)
- Si un nodo cumple *marca* = *1* y *param* = *u*, entonces el nodo apilado justo sobre él (si existe) cumple *param* = $s_1(u)$.
- Si un nodo cumple *marca* = *2* y *param* = *u*, entonces el nodo cumple *resul* = $f(s_1(u))$ y el nodo apilado justo sobre él (si existe) cumple *param* = $s_2(u)$.
- Siempre que el nodo cima cumple *marca* = *2* y *param* = *u*, se verifica también que *y* = $f(s_2(u))$.

El invariante, que no detallamos, debería formalizar estas condiciones. Cuando la pila queda vacía y el bucle termina, en *y* queda el resultado *f(x)* correspondiente a la llamada inicial. En cuanto a la expresión de acotación, es posible definirla como el número de visitas a nodos del árbol de llamadas que están aún pendientes de realizar en cada momento.

(a) Aplica la transformación descrita a la función recursiva doble *fib* del ejercicio 83. Estudia paso a paso la evolución de la pila de nodos para una llamada inicial concreta, tal como $fib_{it}(3)$ o $fib_{it}(4)$.

(b) Aplica la transformación a las funciones recursivas dobles consideradas en los ejercicios 261-264.

Arboles ordenados y de búsqueda: modelo matemático y especificación

287. Sea un árbol binario que almacena en sus nodos elementos de un tipo de la clase ORD. Se dice que *a* está ordenado si se da alguno de los dos casos siguientes:

(a) *a* es vacío.

(b) *a* no es vacío, sus dos hijos están ordenados, todos los elementos del hijo izquierdo son estrictamente menores que el elemento de la raíz, y el elemento de la raíz es estrictamente menor que todos los elementos del hijo derecho.

Observa que esta definición equivale a pedir que el recorrido en inorden de *a* produzca una lista de elementos ordenada en orden estrictamente creciente. Especifica mediante ecuaciones una operación booleana que reconozca los árboles ordenados.

288. La operación de *inserción* de un elemento en un árbol se especifica de manera que el árbol resultante quede ordenado si lo estaba el árbol de partida. Dibuja el árbol ordenado de enteros que se obtiene a partir del árbol vacío, insertando sucesivamente los números siguientes:

20, 12, 17, 31, 26, 43, 7, 35

289. La operación de *búsqueda* de un elemento en un árbol ordenado se especifica de manera que devuelva como resultado el subárbol obtenido tomando como raíz el nodo que contenga el elemento buscado. Si ningún nodo del árbol contiene el elemento buscado, se devuelve el árbol vacío. Indica los resultados de buscar los enteros 35, 43 y 31, respectivamente, en el árbol obtenido en el ejercicio 288.

290. La operación de *borrado* de un elemento en un árbol ordenado se especifica de manera que el resultado sea un nuevo árbol ordenado del cual se ha eliminado el nodo que contenía el elemento en cuestión, si existía. Si ningún nodo del árbol contenía dicho elemento, la operación de borrado deja el árbol inalterado. Dibuja los árboles resultantes de borrar los enteros que se indican a continuación en el árbol obtenido en el ejercicio 288:

(a) Borrar 35: Este elemento aparece en una hoja. Esta hoja se elimina.

(b) Borrar 43: Este elemento aparece en un nodo interno con un solo hijo. Este nodo se elimina y se sustituye por su hijo.

(c) Borrar 31: Este elemento aparece en un nodo interno con dos hijos. Se reemplaza el elemento de este nodo por el menor elemento de su hijo derecho. Mediante otro borrado, se quita del hijo derecho el nodo que contenía este elemento.

†291. Escribe una especificación algebraica de una TAD ARB-ORD[E :: ORD] que represente el comportamiento de los árboles ordenados, con operaciones para crear un árbol vacío, insertar, buscar, borrar y preguntar si un elemento aparece en un árbol.

Sugerencia: Usa REC-PROF-ARBIN[E], ocultando aquellas operaciones que no convenga poner a disposición de los clientes de ARB-ORD[E]. En particular, las operaciones públicas de ARB-ORD[E] deben ser tales que los clientes de este TAD sólo puedan construir árboles ordenados.

292. Sean *a :Arbin[elem], x : Elem*. Si *a* no es un árbol ordenado, algunas operaciones de ARB-ORD pueden comportarse extrañamente:

(a) En *inserta(x, a)* puede haber elementos repetidos, aunque en *a* no los haya.

(b) Puede ser *está?(x, a)* = *falso*, aunque *x* aparezca en algún nodo de *a*.

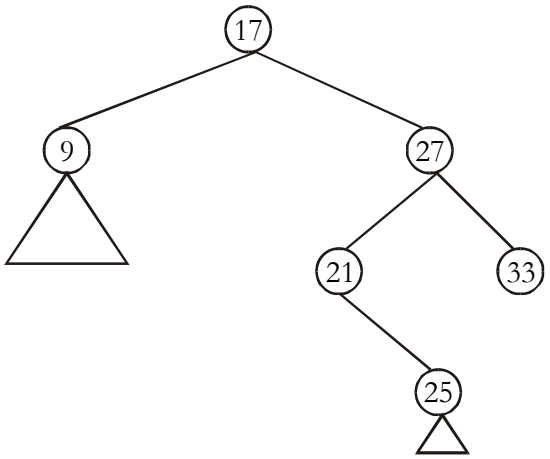
(c) Puede ser *borra(x, a)* = *a*, aunque *x* aparezca en algún nodo de *a*.

Busca ejemplos concretos en los que esto ocurra. Observa que los clientes del TAD ARB-ORD no tropiezan con este problema, debido a que las operaciones exportadas sólo permiten construir árboles ordenados.

†293. Los *árboles de búsqueda*, de tipo *Arbus[Cla, Val]*, son semejantes a los árboles ordenados; pero ahora cada nodo almacena dos informaciones: una *clave* de tipo *Cla*, y un valor de tipo *Val*. El tipo *Cla* debe poseer igualdad y orden, y el tipo *Val* debe estar dotado de una operación (⊕): (Val, Val) → Val que combine dos valores. Especifica un TAD ARBUSCA[C::ORD, V::COMB] que represente el comportamiento de los árboles de búsqueda, donde la clase de tipos COMB requiere que el tipo-parámetro *V* posea una operación de combinación (⊕). ARBUS[C, V] debe exportar operaciones para crear un árbol vacío, insertar un valor asociado a una clave dada, buscar una clave dada, y preguntar si una clave dada aparece en un árbol. La operación de inserción debe usar (⊕) para combinar el nuevo valor insertado con el antiguo, en el caso de que la clave de inserción ya aparezca en el árbol.

Arboles ordenados y de búsqueda: implementación

294. Plantea una implementación de la operación *inserta* de los árboles ordenados como función recursiva, basándote en las operaciones ya disponibles para ARBIN y siguiendo la pauta de la especificación ecuacional de ARB-ORD. Estudia la estructura compartida resultante de ejecutar *a' := inserta(19, a)*, suponiendo que *a* sea el árbol de enteros de la figura siguiente y que la implementación use la representación dinámica del ejercicio 249.



†295. Desarrolla una implementación de ARB-ORD basada en una representación dinámica similar a la del ejercicio 249, realizando las operaciones *busca* y *está?* como una función, y las operaciones *inserta* y *borra* como procedimientos con un parámetro ***es*** *a : Arbusca[Elem]*. *Advertencia*: Esta implementación no puede plantearse modularmente, importando el tipo representante de un módulo ARBIN[ELEM]. Para programar correctamente *inserta* y *borra* es necesario tener acceso a la estructura representante del árbol.

†296. Extiende el ejercicio anterior para obtener una implementación de ARBUSCA basada en una representación dinámica semejante a la del ejercicio 249. Ahora, cada nodo deberá contener 4 campos: una clave, un valor, y dos enlaces apuntando a los hijos.

297. Modifica las implementaciones desarrolladas en los dos ejercicios anteriores, de manera que la función *busca* y los procedimientos *inserta* y *borra* sean iterativos en lugar de recursivos.

298. Una familia $\mathcal{F}$ de árboles binarios se llama *equilibrada* si existen constantes $n_0 \in \aleph$, $c \in \Re_+$ tales que para todo $n \geq n_0$ y todo árbol $a \in \mathcal{F}$ con n nodos se tenga $talla(a) \leq c \cdot log\ n$. Razona que el tiempo de ejecución de las operaciones de búsqueda, inserción y borrado en árboles de una familia equilibrada es $O(\log n)$, siendo n el número de nodos. Para árboles arbitrarios, este tiempo aumenta a $O(n)$ en el caso peor.

Arboles ordenados y de búsqueda: aplicaciones

299. Supongamos que *Elem* viene dado por un tipo de datos de la clase ORD. Especifica mediante ecuaciones una operación

```
hazArbusca: Lista[Elem] → Arbus[Elem]
```

tal que *hazArbusca(xs)* sea el árbol ordenado resultante de insertar sucesivamente los datos de la lista *xs*, comenzando con un árbol vacío.

Sugerencia: Especifica *hazArbusca(xs) = reiteraInserta(xs, Vacío)*, y añade ecuaciones que especifiquen la operación más general *reiteraInserta*.

300. Sea *Elem* como en el ejercicio anterior. Sea *a : Arbus[elem]* un árbol ordenado.

(a) Demuestra que en general no es cierto que *a = hazArbus(inOrd(a))*. ¿Qué forma tiene el árbol *hazArbusca(inOrd(a))*?

(b) Demuestra que siempre se cumple que *a = hazArbusca(preOrd(a))*, razonando por inducción sobre el número de nodos de *a*.

301. Basándote en el apartado (b) del ejercicio anterior, construye un algoritmo que sea capaz de reconstruir un árbol ordenado a partir de su recorrido en preorden, sin ninguna información adicional. Compara con el ejercicio 280. Desarrolla dos variantes del algoritmo, adecuadas a dos posibles presentaciones del recorrido en preorden: como lista o como secuencia.

302. Programa un procedimiento recursivo que realice el recorrido en inorden de un árbol binario dejando el resultado en un vector, de acuerdo con la siguiente especificación pre/post:

```
proc inOrd( e a : Arbin[Elem]; es v : Vector[1..N] de Elem; es p : Ent );
{ P₀ : p = P ∧ v = V ∧ 1 ≤ p ≤ N+1 ∧ numNodos(a) ≤ N−p+1 }
{ Q₀ : P ≤ p+1 ≤ N+1 ∧ contenido(v, P, p) = inOrd(a) ∧
      v coincide con V fuera del intervalo [P..p]       }
fproc
```

Nota: Este ejercicio y el anterior se pueden generalizar fácilmente para extenderlos al caso de árboles de búsqueda de tipo *Arbus[Cla, Val]*.

303. El *algoritmo de ordenación mediante árbol de búsqueda* (*treeSort*) ordena un vector *v : Vector[1..N] de Elem* por medio del siguiente proceso:

(a) se construye un árbol ordenado mediante sucesivas inserciones de los elementos del vector a partir de un árbol vacío.

(b) se recorre el árbol obtenido en inorden, y durante el recorrido se van colocando los elementos en *v* (comenzando por la posición 1).

Diseña una función que realice este algoritmo, incluyendo en la precondición la condición de que el vector no tiene elementos repetidos. Razona que el algoritmo consume tiempo $O(N*t)$, siendo t la talla que llega a alcanzar el árbol de búsqueda durante el proceso.

304. Una variante del concepto de árbol ordenado consiste en permitir que distintos nodos del árbol almacenen un mismo elemento, pero con la condición siguiente: el elemento de cualquier nodo debe ser *estrictamente mayor* que los elementos de los nodos del hijo izquierdo, y *menor* o *igual* que los elementos del hijo derecho. Equivalentemente, el recorrido en inorden del árbol debe dar una lista de elementos ordenada en orden no decreciente. Especifica un TAD parametrizado ARB-ORD-REP[E :: ORD] que represente el comportamiento de este nuevo tipo de árboles ordenados.

305. Usando el TAD ARB-ORD-REP en lugar de ARB-ORD, modifica el algoritmo de ordenación del ejercicio 303 de modo que sea posible ordenar vectores que contengan repeticiones de elementos.

306. Construye algoritmos de ordenación basados en árboles de búsqueda que puedan aplicarse para ordenar *listas* o *secuencias* en lugar de vectores.

307. El problema de las *concordancias* consiste en lo siguiente: Dado un texto, se trata de contar el número de veces que aparece en él cada palabra, y producir un listado ordenado alfabéticamente por palabras, donde cada palabra aparece acompañada del número de veces que ha aparecido en el texto. Suponemos que el texto a analizar viene dado como secuencia de tipo *Sec[Palabra]*, siendo *Palabra* un tipo disponible de la clase ORD. Se pide construir un algoritmo que resuelva el problema con ayuda de un árbol de búsqueda de tipo *Arbus[Palabra, Nat]*, y analizar su complejidad. El listado pedido se dará como secuencia de parejas, de tipo *Sec[Pareja[Palabra, Nat]]*.

308. Dado un texto organizado por líneas, el problema de las *referencias cruzadas* pide producir un listado ordenado alfabéticamente por palabras, donde cada palabra del texto vaya acompañada de una *lista de referencias*, que contendrá los números de todas las líneas del texto en las que aparece la palabra en cuestión (con posibles repeticiones si la palabra aparece varias veces en una misma línea). Suponiendo que el texto a analizar venga dado como secuencia de tipo *Sec[Sec[Palabra]]*, construye un algoritmo que resuelve el problema con ayuda de un árbol de búsqueda de tipo *Arbus[Palabra, Lista[Nat]]*, y analiza su complejidad. El listado pedido se dará como secuencia de parejas, de tipo *Sec[Pareja[Palabra, Lista[Nat]]]*.

309. Estudia una implementación del TAD POLI de los polinomios con coeficientes enteros en una indeterminada (cfr. ejercicio 151) utilizando como representación de un polinomio un árbol de búsqueda de tipo *Arbus[Exp, Coef]*. La idea es que cada nodo del árbol representa un monomio, con el exponente como clave y el coeficiente como valor asociado a ésta. Compara la eficiencia de las operaciones resultantes con las otras implementaciones de los polinomios estudiadas anteriormente (cfr. ejercicios 170 y 230).

310. Desarrolla implementaciones modulares de los TADs siguientes, usando árboles de búsqueda importados de un módulo conveniente para la construcción del tipo representante. En cada caso, analiza el tiempo de ejecución de las operaciones y el espacio ocupado por la representación.

(a) El TAD BOSTEZOS del ejercicio 231.

(b) El TAD CONSULTORIO del ejercicio 232.

(c) El TAD MERCADO del ejercicio 233.
(d) El TAD BANCO del ejercicio 234.
(e) El TAD BIBLIOTECA del ejercicio 236.

Arboles AVL

311. Especifica mediante ecuaciones operaciones booleanas que reconozcan los árboles binarios equilibrados en talla y los árboles AVL, respectivamente.

312. Observa que todos los árboles completos son equilibrados. Dibuja un árbol AVL que no sea completo, y un árbol ordenado que no sea AVL.

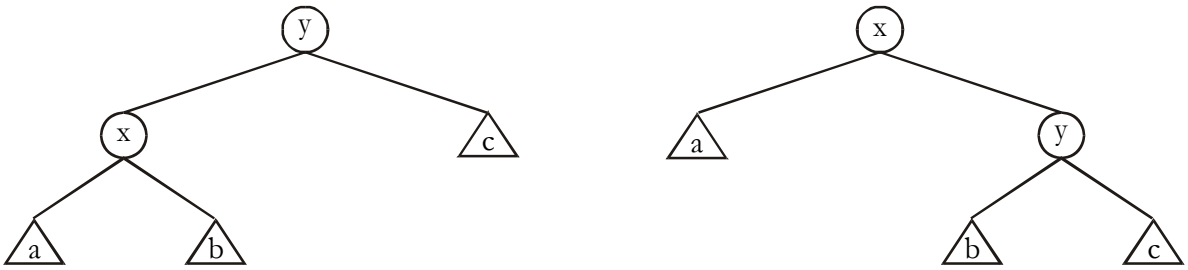
313. Construye todos los árboles ordenados posibles formados por cuatro nodos con elementos diferentes (e.g. 1, 2, 3, 4). ¿Cuántos de ellos son árboles AVL?

314. Busca una ley de recurrencia para la sucesión b_n, siendo b_n el número de árboles de búsqueda que se pueden formar con n nodos con elementos diferentes (e.g. 1, 2, … , n).

315. Define recursivamente dos sucesiones n_t *: Nat*, a_t *: Arbus[Nat]*, de manera que para todo $t \geq 0$, a_t sea un árbol AVL "lo más desequilibrado posible" con n_t nodos. Estos a_t se llaman *árboles de Fibonacci*, y los n_t se llaman *números de Leonardo*.

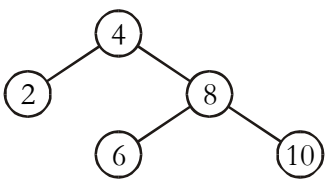
316. Especifica ecuacionalmente una operación que calcule el factor de equilibrio de un árbol binario dado.

317. Especifica ecuacionalmente las operaciones de *rotación a la derecha* y *rotación a la izquierda* que se definen gráficamente en la figura siguiente. Se supone que *x, y : Elem; a, b, c : Arbus[Elem]*.



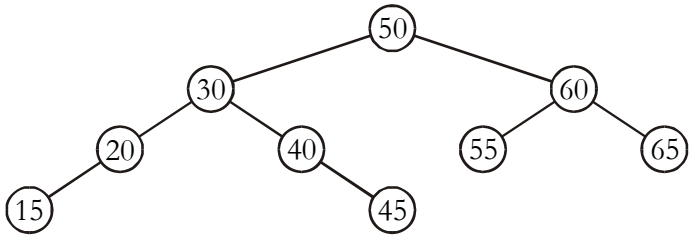
318. En cada uno de los casos siguientes, dibuja el árbol resultante de la inserción ordinaria, y a continuación efectúa las rotaciones necesarias para conseguir el efecto de una inserción AVL. Suponemos que los datos son enteros, y que la función *clave* es la identidad.

(a) *insertaAVL(11, a)*, siendo *a*:



(b) *insertaAVL(7, a)*, siendo *a* como en el apartado anterior:

(c) *insertaAVL(10, b)*, siendo *b*:



†319. Especifica ecuacionalmente la operación de inserción en árboles AVL.

320. Modifica el tipo representante utilizado en anteriores implementaciones de árboles binarios (ver ejercicios 249, 250 y 296) de manera que sea posible determinar eficientemente los factores de equilibrio sin calcular tallas. Formula un invariante de la representación y una función de abstracción adecuados para representar árboles AVL con ayuda del nuevo tipo.

†321. Desarrolla un algoritmo recursivo que implemente la inserción AVL. Puedes consultar la Sección 5.6.2 del texto de Xavier Franch.

322. Dibuja el árbol resultante de efectuar el borrado ordinario *borra(1, a_5)*, siendo a_5 el árbol de Fibonacci de lugar 5 (ver ejercicio 315). A continuación, efectúa las rotaciones necesarias para reequilibrar dos subárboles a lo largo de la trayectoria de búsqueda para que el árbol total quede reequilibrado.

323. Especifica ecuacionalmente la operación de borrado en árboles AVL.

†324. Desarrolla una implementación de la operación de borrado AVL siguiendo un planteamiento similar al del ejercicio 321, y usando el mismo tipo representante. Consulta la Sección 5.6.2 del texto de Franch.

†325. Adapta todo lo desarrollado en los ejercicios 316-324 para obtener una especificación e implementación del comportamiento de los árboles de búsqueda AVL de tipo *Arbus[Cla, Val]*.

Colas de prioridad

326. Una *cola de prioridad* tiene un comportamiento similar al de una cola, con la diferencia de que el tipo de los elementos almacenados dispone de operaciones de igualdad y orden. Si $x < y$, entenderemos que el elemento *x precede* a (o tiene *mayor prioridad* que) el elemento *y*. Si $x == y$, entenderemos que *x* e *y* tienen la misma prioridad, aunque no sean necesariamente idénticos. Construye un TAD parametrizado COLA-PRIO[E :: ORD] que formalice el comportamiento de las colas de prioridad, con operaciones para crear una cola de prioridad vacía, añadir un elemento, consultar el elemento de mayor prioridad, eliminar el elemento de mayor prioridad, y averiguar si la cola es vacía. Ten en cuenta que se considera como elemento de mayor prioridad al que sea mínimo con respecto al orden <. La operación que añade un nuevo elemento a una cola de prioridad sólo debe estar definida si la prioridad del nuevo elemento es diferente de las prioridades de todos los elementos ya presentes en la cola. Es decir, prohibimos en nuestra especificación que en una cola de prioridad se encuentren elementos con prioridades repetidas.

En los ejercicios siguientes suponemos que el tipo *Elem* dispone de operaciones de igualdad y orden, que se interpretan en el sentido de una comparación de prioridades.

327. Plantea una implementación secuencial de las colas de prioridad, utilizando listas o secuencias ordenadas como tipo representante. Estima los órdenes de magnitud de los tiempos de ejecución de las operaciones, en el caso peor.

Montículos

328. Sea *a : Arbin[Elem]*. Se dice que *a* es un *montículo* (de mínimos) syss *a* es semicompleto, la raíz de *a* (si existe) precede en prioridad a los elementos almacenados en los hijos, y los hijos (si existen) son a su vez montículos.

`*(a)` Dibuja un montículo de números naturales que contenga los números del intervalo [1..7].

`(b)` Especifica ecuacionalmente una operación booleana

```
montículo? : Arbin[Elem] → Bool
```

que reconozca si un árbol binario dado es o no un montículo.

329. Para *insertar* un nuevo elemento *x* en un montículo *a* se utiliza el siguiente algoritmo:

`(I.1)` Se añade *x* como nueva hoja en la primera posición libre del último nivel. El resultado es un árbol semicompleto, si *a* lo era.

`(I.2)` Se deja *flotar x*; i.e., mientras *x* no se encuentre en la raíz y sea menor que su padre, se intercambia *x* con su padre. El resultado es un montículo, suponiendo que *a* lo fuese y que *x* fuese diferente (en prioridad) de todos los elementos de *a*.

`*(a)` Construye un montículo de naturales por inserción sucesiva de los números 5, 3, 4, 7, 1, 6, 2, en este orden.

`†(b)` Especifica ecuacionalmente dos operaciones

```
ponerHoja: (Elem, Arbin[Elem]) → Arbin[Elem]
flotar: Arbin[Elem] – → Arbin[Elem]
```

de modo que la siguiente ecuación especifique correctamente la operación de inserción para montículos:

```
insertar: (Elem, Arbin[Elem]) → Arbin[Elem]
insertar(x, a) = flotar(ponerHoja(x, a))
```

(se trata de que *insertar(x, a)* sea un montículo, siempre que *a* lo sea y *x* sea diferente en prioridad de todos los elementos de *a*).

330. Para *eliminar* el elemento de la raíz de un montículo *a*, se utiliza el siguiente algoritmo:

`(E.1)` Si *a* es vacío, la operación no tiene sentido. Si *a* tiene un solo nodo, el resultado de la eliminación es el montículo vacío. Si *a* tiene dos o más nodos, se quita la última hoja y se pone el elemento *x* que ésta contenga en el lugar de la raíz, que queda eliminada. Esto da un árbol semicompleto, si *a* lo era. Se pasa a (E.2).

`(E.2)` Se deja *hundirse x*; i.e., mientras *x* ocupe una posición con hijos y sea mayor que alguno de sus hijos, se intercambia *x* con el *hijo elegido*, que es aquel que sea menor que *x* (si hay un solo hijo con esta prioridad) o el hijo menor (si ambos son menores que *x*). El resultado es un montículo, suponiendo que *a* lo fuese.

(a) Realiza sucesivas eliminaciones de la raíz en el montículo obtenido en el ejercicio 329(a), hasta llegar al montículo vacío. Observa el orden en el que van siendo extraídos los números.

†(b) Especifica ecuacionalmente dos operaciones

```
preparar: Arbin[Elem] – → Arbin[Elem]
hundir: Arbin[Elem] – → Arbin[Elem]
```

de modo que las siguientes ecuaciones especifiquen correctamente la operación de eliminación del mínimo para montículos:

```
eliminarRaíz: Arbin[Elem] – → Arbin[Elem]
eliminarRaíz(a) = Vacío                 si numNodos(a) == 1
eliminarRaíz(a) = hundir(preparar(a))    si numNodos(a)  > 1
```

Implementación de colas de prioridad usando montículos

331. Plantea una implementación modular del TAD COLA-PRIO usando montículos como tipo representante. Suponiendo que el módulo MONTICULO exporte operaciones *insertar* y *eliminarRaíz* ejecutables en tiempo logarítmico (con respecto al número de elementos almacenados), ¿qué puede asegurarse sobre los tiempos de ejecución de las operaciones de COLA-PRIO?

332. Define un tipo representante adecuado para una representación de los montículos basada en vectores. Formula el invariante de la representación y la función de abstracción.

Recuerda que un árbol completo de talla *t* tiene 2^t-1 nodos. Para los ejercicios que siguen presuponemos las declaraciones:

```
const
  max = 2^t–1;
tipo
  Vec = Vector[1..max] de Elem;
```

333. Programa los procedimientos especificados informalmente a continuación.

```
proc flota( es v : Vec; e i, n : Nat);
{ P₀ : v = V ∧ 1 ≤ i ≤ n ≤ max ∧
       dejando flotar v(i) puede lograrse que v[1..n] llegue a
       representar un montículo                                  }
{ Q₀ : v[1..n] representa un montículo ∧
      v se ha obtenido a partir de V dejando flotar V(i)         }
fproc

proc hunde( es v : Vec; e i, n : Nat);
{ P₀ : v = V ∧ 1 ≤ i ≤ n ≤ max ∧
       dejando hundirse v(i) puede lograrse que v[1..n] llegue a
       representar un montículo                                   }
```

```
{ Q₀ : v[1..n] representa un montículo ∧
       v se ha obtenido a partir de V dejando hundirse V(i)       }
fproc
```

Observa que estos procedimientos corresponde a generalizaciones de las operaciones *flotar* y *hundir* introducidas en los ejercicios 329 y 330, y que su tiempo de ejecución es $O(\log n)$ en el caso peor.

334. Usando los procedimientos del ejercicio 333 y el tipo representante del ejercicio 332, construye implementaciones de las operaciones de inserción y eliminación de la raíz en un montículo, que sean ejecutables en tiempo logarítmico.

335. Usando los procedimientos del ejercicio 333, programa un procedimiento ejecutable en tiempo $O(\log n)$ que satisfaga la especificación siguiente. Observa que este procedimiento puede servir para implementar una operación que modifique el montículo cambiando uno de sus datos por otro de diferente prioridad.

```
proc altera( es v : Vec; e i, n : Nat; e x : );
{ P₀ : v = V ∧ 1 ≤ i ≤ n ≤ max ∧
       v[1..n] representa un montículo ∧
       la prioridad de x es diferente de las prioridades de
       los datos de v[1..n]                                    }
{ Q₀ : v[1..n] representa un montículo ∧
       v se ha obtenido a partir de V asignando x a V(i) y
       dejándolo flotar o hundirse, según convenga             }
fproc
```

336. Desarrolla una implementación de las colas de prioridad basada en vectores, con un invariante de la representación que fuerce al vector a representar un montículo.

Observa: A diferencia del ejercicio 334, se exportan colas de prioridad en vez de montículos. A diferencia del ejercicio 331, no se importan montículos de otro módulo.

Algoritmos de ordenación con colas de prioridad y montículos

337. Una sucesión de n elementos sin prioridades repetidas puede ordenarse construyendo una cola de prioridad a partir de la cola vacía por inserción reiterada de los elementos, y reiterando seguidamente la operación de extraer el elemento mínimo (con prioridad máxima) hasta que la cola se vacíe. Suponiendo disponible un módulo que implemente el TAD COLA-PRIO, construye un procedimiento de ordenación para vectores de tipo *Vec* siguiendo este método, y razona que el tiempo de ejecución es $O(n \cdot \log n)$.

338. Una generalización del problema del ejercicio anterior consiste en obtener los k elementos menores de la sucesión dada, ordenados de menor a mayor, siendo $0 \leq k \leq n$. Modifica el procedimiento del ejercicio anterior para resolver este nuevo problema, y razona que el tiempo de ejecución sigue siendo $O(n \cdot \log n)$.

En los dos ejercicios anteriores, el algoritmo necesita espacio auxiliar $O(n)$ para la cola de prioridad, además del espacio ocupado por el vector a ordenar. En 1964, J.W.J. Williams y R.W. Floyd construyeron un algoritmo de ordenación de vectores basado en montículos, que no necesita espacio auxiliar. A cambio, el algoritmo no es modular, porque utiliza el espacio del mismo vector

que se está ordenando para representar el montículo. Los ejercicios que siguen desarrollan esta idea.

339. El siguiente procedimiento reorganiza un segmento de vector de manera que pase a representar un montículo:

```
proc hazMont( es v : Vec; e n : Nat );
{ P₀ : v = V ∧ 0 ≤ n ≤ max ∧
      ∀ i, j : 1 ≤ i < j ≤ n : v(i) ≠ v(j) }
var
  i : Nat;
inicio
  si n ≤ 1
    entonces
      seguir
    sino
      para i bajando desde n div 2 hasta 1 hacer
      { I : ∀ k : i+1 ≤ k ≤ n : v[k..n] representa un montículo }
        hunde(v, i, n)     /* ver ej. 334 */
      fpara
  fsi
{ Q₀ : v[1..n] representa un montículo ∧ v[n+1..max] = V[n+1..max] ∧
       v[1..n] es una permutación de V[1..n]  }
fproc
```

Se puede demostrar que este procedimiento consume tiempo $O(n)$; ver el texto de Xavier Franch, Sección 5.5.2. Dibuja el árbol semicompleto cuyo recorrido por niveles da la sucesión de números 10, 8, 2, 4, 7, 5, 9, 3, 6, 1 (en este orden) y conviértelo en un montículo siguiendo el algoritmo del procedimiento *hazMont*.

340. El siguiente procedimiento ordena un vector con ayuda de un montículo representado dentro del mismo vector:

```
proc ordenaMont ( v : Vec; n : Nat );
{ P₀ : v = V ∧ 0 ≤ n ≤ max ∧
      ∀ i, j : 1 ≤ i < j ≤ n : v(i) ≠ v(j) }
var
  i : Nat;
inicio
  hazMont(v, n);
  si n ≤ 1
    entonces
      seguir

    sino
      para i bajando desde n hasta 2 hacer
```

```
        { I : v[1..n] es una permutación de V[1..n] ∧
              v[n+1..max] = V[n+1..max] ∧
              v[i+1..n] contiene los n-i datos de V[1..n] de
              menor prioridad, ordenado en orden decreciente ∧
              v[1..i] representa un montículo                    }
        fpara
    fsi
{ Q₀ : v[1..n] es una permutación de V[1..n] ∧
       v[n+1..max] = V[n+1..max] ∧
       v[1..n] está ordenado en orden decreciente   }
fproc
```

Usando el ejercicio 339, razona que este algoritmo de ordenación requiere tiempo $O(n \cdot \log n)$. Ensaya el funcionamiento del algoritmo para ordenar un vector que contenga la sucesión de números 10, 8, 2, 4, 7, 5, 9, 3, 6, 1, en este orden.

341. Desarrolla modificaciones del algoritmo del ejercicio 340, de manera que:

(a) Se permitan prioridades repetidas y se obtenga al final un vector ordenado en orden no decreciente. Para esto, basta con modificar la propiedad de montículo, exigiendo que el elemento de cada nodo sea *menor o igual* que los elementos de sus dos hijos (si existen), y adaptar las operaciones de los montículos a este nuevo criterio.

(b) Se ordenen únicamente los k menores elementos del vector, como en el ejercicio 338.

Colas de prioridad con prioridades repetidas

342. Construye una especificación algebraica del TAD de las colas de prioridad con posibles repeticiones de prioridades. La especificación deberá determinar que los elementos de igual prioridad sean atendidos por orden de llegada.

343. El TAD especificado en el ejercicio anterior no se puede implementar directamente mediante montículos, ya que puede ocurrir que ciertos elementos con prioridades iguales no salgan del montículo en el mismo orden en que entraron. Construye un ejemplo que ilustre este fenómeno.

CAPÍTULO 6
TABLAS

6.1 Modelo matemático y especificación

Desde el punto de vista matemático, las tablas son aplicaciones que hacen corresponder a claves, $c \in C$ –C conjunto de claves–, valores, $v \in V$ –V conjunto de valores–. Podemos verlas como una generalización de los árboles de búsqueda, como colecciones de pares (clave, valor), donde el acceso se realiza por las claves y donde, a diferencia de los árboles de búsqueda, no se supone una estructura de árbol –aunque pueden ser implementadas como tales–.

Las aplicaciones de las tablas en Informática incluyen, entre otras: las tablas de símbolos de los compiladores, las tablas usadas en los sistemas de archivos, las tablas usadas en los sistemas de gestión de bases de datos.

Para dar un modelo matemático de las tablas hay tres posibilidades naturales (cfr. [Franch 93], pp. 160-162):

— Funciones totales $T : C \to V$, suponiendo en V un elemento distinguido *NoDef*.

— Funciones parciales $T : C \dashrightarrow V$.

— Conjuntos $T \subseteq V$, suponiendo en este caso que V es un conjunto de elementos con claves asociadas –mediante una operación *clave:* $V \to C$–, y conviniendo en que C no puede contener dos elementos distintos con la misma clave.

Las operaciones que nos interesan en las tablas son las que permiten: crear una tabla vacía, insertar una pareja (clave, valor), comprobar si la tabla está vacía, comprobar si una clave está en la tabla, consultar el valor asociado a una clave, y borrar un par (clave, valor).

6.1.1 Tablas como funciones parciales

La especificación de las tablas como funciones parciales

```
tad TABLA[C :: EQ, V :: ANY]
    renombra
      C.Elem a Clave
      V.Elem a Valor
    usa
      BOOL
    tipo
      Tabla[Clave, Valor]
    operaciones
      TablaVacía: → Tabla[Clave, Valor]                          /* gen */
      Inserta: (Tabla[Clave,Valor], Clave, Valor) → Tabla[Clave,Valor]/* gen */
      esVacía: Tabla[Clave, Valor] → Bool                        /* obs */
      está: (Tabla[Clave, Valor], Clave) → Bool                  /* obs */
      consulta: (Tabla[Clave, Valor], Clave) – → Valor           /* obs */
```

```
      borra: (Tabla[Clave, Valor], Clave) → Tabla[Clave, Valor]   /* mod */
    ecuaciones
      ∀ t : Tabla[Clave, Valor] : ∀ i, j : Clave : ∀ x, y : Valor :
      Inserta(Inserta(t, i, x), j, y)                             =
Inserta(t, j, y)                    si i == j
      Inserta(Inserta(t, i, x), j, y)                             =
Inserta(Inserta(t, j, y), i, x)     si i /= j
      esVacía(TablaVacía)           = cierto
      esVacía(Inserta(t, i, x))     = falso
      está(TablaVacía, j)           = falso
      está(Inserta(t, i, x), j)     = i == j OR está(t, j)
      def consulta(t, i) si está(t, i)
      consulta(Inserta(t, i, x), j) = x                           si i == j
      consulta(Inserta(t, i, x), j) =f consulta(t, j)             si i /= j
      borra(TablaVacía, j)          = TablaVacía
      borra(Inserta(t, i, x), j)    = borra(t, j)                 si i == j
      borra(Inserta(t, i, x), j)    = Inserta(borra(t, j), i, x)  si i /= j
    errores
      consulta(t, i) si NOT está(t, i)
ftad
```

6.1.2 Tablas como funciones totales

Para poder especificar las tablas como funciones totales –la representación en la que nos centraremos en el resto de este tema– debemos suponer a las claves equipadas con un valor especial *NoDef*, restricción que expresamos mediante una clase de tipos:

```
clase EQ-ND
    hereda
      EQ
    operaciones
      NoDef: → Elem
fclase
```

Ahora ya podemos definir las tablas como funciones totales

```
tad TABLA[C :: EQ, V :: EQ-ND]
    renombra
      C.Elem a Clave
      V.Elem a Valor
    usa
      BOOL
    tipo
      Tabla[Clave, Valor]
    operaciones
      TablaVacía: → Tabla[Clave, Valor]                           /* gen */
```

```
      Inserta: (Tabla[Clave,Valor], Clave, Valor) → Tabla[Clave,Valor]/* gen */
      esVacía: Tabla[Clave, Valor] → Bool                          /* obs */
      está: (Tabla[Clave, Valor], Clave) → Bool                    /* obs */
      consulta: (Tabla[Clave, Valor], Clave) → Valor               /* obs */
      borra: (Tabla[Clave, Valor], Clave) → Tabla[Clave, Valor]    /* mod */
   ecuaciones
      ∀ t : Tabla[Clave, Valor] : ∀ i, j : Clave : ∀ x, y : Valor :
      Inserta(Inserta(t, i, x), j, y)                              =
Inserta(t, j, y)                    si i == j
      Inserta(Inserta(t, i, x), j, y)                              =
Inserta(Inserta(t, j, y), i, x)     si i /= j
      esVacía(TablaVacía)           = cierto
      esVacía(Inserta(t, i, x))     = falso
      está(TablaVacía, j)           = falso
      está(Inserta(t, i, x), j)     = i == j OR está(t, j)
      consulta(TablaVacía, j)       = NoDef
      consulta(Inserta(t, i, x), j) = x                            si i == j
      consulta(Inserta(t, i, x), j) = consulta(t, j)               si i /= j
      borra(TablaVacía, j)          = TablaVacía
      borra(Inserta(t, i, x), j)    = borra(t, j)                  si i == j
      borra(Inserta(t, i, x), j)    = Inserta(borra(t, j), i, x)   si i /= j
ftad
```

6.1.3 Tablas ordenadas

Para ciertas aplicaciones de las tablas resulta conveniente que exista un orden entre las claves, y que se disponga de una operación que extraiga la información almacenada en la tabla en forma de lista (o secuencia) de parejas de tipo (Clave, Valor), ordenada por claves. En ocasiones, también resulta útil que una inserción con clave ya presente en la tabla combine el nuevo valor con el antiguo, en lugar de limitarse a reemplazar el valor antiguo por el nuevo. Llamaremos *tablas ordenadas* a las tablas que incorporen estas dos ideas. Este TAD es prácticamente igual al de los árboles de búsqueda, con la diferencia de que la consulta en un árbol de búsqueda da un árbol como resultado, mientras que la consulta en un tabla devuelve un valor. Es directo implementar las tablas ordenadas como árboles de búsqueda.

La especificación hace uso de una clase que caracteriza a los tipos combinables equipados con el valor *NoDef*:

```
clase EQ-ND-COMB
   hereda
      EQ-ND, COMB
   axiomas
      ∀ x : Elem :
      NoDef ⊕ x = x
fclase
```

La especificación de las tablas ordenadas

```
tad TABLA-ORD[C :: ORD, V :: EQ-ND-COMB]
    renombra
      C.Elem a Clave
      V.Elem a Valor
    usa
      BOOL
      LISTA[PAREJA[C, V]]
    tipo
      Tabla[Clave, Valor]
    operaciones
      TablaVacía: → Tabla[Clave, Valor]                              /* gen */
      Inserta: (Tabla[Clave,Valor], Clave, Valor) → Tabla[Clave,Valor]/* gen */
      esVacía: Tabla[Clave, Valor] → Bool                            /* obs */
      está: (Tabla[Clave, Valor], Clave) → Bool                      /* obs */
      consulta: (Tabla[Clave, Valor], Clave) → Valor                 /* obs */
      borra: (Tabla[Clave, Valor], Clave) → Tabla[Clave, Valor]      /* mod */
      lista: Tabla[Clave, Valor] → Lista[Pareja[Clave, Valor]]       /* obs */
    operaciones privadas
      minClave: Tabla[clave, Valor] – → Clave
    ecuaciones
      ∀ t, t' : Tabla[Clave, Valor] : ∀ i, j : Clave : ∀ x, y : Valor :
      Inserta(Inserta(t, i, x), j, y)                                =
Inserta(t, i, x ⊕ y)               si i == j
      Inserta(Inserta(t, i, x), j, y)                                =
Inserta(Inserta(t, j, y), i, x)    si i /= j
      esVacía(TablaVacía)          = cierto
      esVacía(Inserta(t, i, x))    = falso
      está(TablaVacía, j)          = falso
      está(Inserta(t, i, x), j)    = i == j OR está(t, j)
      consulta(TablaVacía, j)      = NoDef
      consulta(Inserta(t, i, x), j) = consulta(t, j) ⊕ x          si i == j
      consulta(Inserta(t, i, x), j) = consulta(t, j)              si i /= j
      borra(TablaVacía, j)         = TablaVacía
      borra(Inserta(t, i, x), j)   = borra(t, j)                  si i == j
      borra(Inserta(t, i, x), j)   = Inserta(borra(t, j), i, x)   si i /= j
      def minClave(t) si NOT esVacía(t)
      minClave(Inserta(TablaVacía, j, y))   = j
      minclave(Inserta(Inserta(t,i,x),j,y)) = minClave(Inserta(t,i,x)) si i
≤ j
      minclave(Inserta(Inserta(t,i,x),j,y)) = minClave(Inserta(t,j,y)) si i
> j
      lista(t)                     = []   si esVacía(t)
      lista(t)                     = [ Par(i, x) / lista(t') ]
        si NOT esVacía(t) AND i = minClave(t) AND
          x = consulta(t, i) AND t' = borra(t, i)
```

```
ftad
```

6.1.4 Casos particulares: conjuntos y vectores

Conjuntos

El comportamiento del TAD CJTO[E :: EQ] se puede considerar equivalente al del TAD TABLA[E :: EQ, V :: EQ-ND], suponiendo que *V* sea un tipo especial que sólo contenga los dos valores *Si* y *Nodef*. Se puede establecer la siguiente correspondencia entre las operaciones de ambos TADs:

CJTO[E :: EQ]	TABLA[E :: EQ, V :: EQ-ND]
Vacío( )	TablaVacía( )
Pon(x, xs)	Inserta(xs, x, Sí)
quita(x, xs)	borra(xs, x)
esVacío(xs)	esVacía(xs)
pertenece(x, xs)	está(xs, x)

Vectores

Los vectores también pueden considerarse como una clase especial de tablas, donde las claves (≡ índices) deben ser de un tipo *discreto*. Un tipo de datos es *discreto* si el tipo contiene una cantidad finita de valores y es isomorfo a un intervalo *[1..n]* de los enteros, disponiendo de igualdad, orden y operaciones para determinar los valores primero y último y los valores siguiente y anterior de un valor dado. La clase DIS de los tipos de datos discretos se puede especificar como subclase de la clase de tipos ORD.

```
clase DIS
    hereda
       ORD
    operaciones
       card: → Nat
       ord: Elem → Nat
       elem: Nat – → Elem
       prim, ult: → Elem
       suc, pred: Elem – → Elem
    axiomas
       ∀ x, y : Elem : ∀ i : Nat :
       card ≥ 1
       1 ≤ ord(x) ≤ card
       def elem(i) si 1 ≤ i ≤ card
       ord(elem(i)) =d i
       elem(ord(x)) = x
       prim = elem(1)
       ult = elem(card)
```

```
      def suc(x) si x /= ult
      suc(x)              =d elem(ord(x) – 1)
      x == y        = ord(x) == ord(y)
      x ≤ y         = ord(x) ≤ ord(y)
fclase
```

Equipados con la clase DIS, podemos construir una especificación algebraica del TAD VECTOR[I :: DIS, D :: ANY] pensado para representar el comportamiento de los vectores, con operaciones que permitan crear un vector vacío, modificar un vector asignando un nuevo dato en la posición de un índice dado, y consultar en un vector el dato asociado a un índice dado:

```
tad VECTOR[I :: DIS, D :: ANY]
    renombra
      I.Elem a Indice
      D.Elem a Dato
    usa
      BOOL
    tipo
      Vector[Indice, Dato]
    operaciones
      Crea: → Vector[Indice, Dato]                                  /* gen */
      Asigna: (Vector[Indice,Dato], Indice, Dato) → Vector[Indice,Dato]/* gen */
      valor: (Vector[Indice, Dato], Indice) – → Dato                /* obs */
    operaciones privadas
      def?: (Vector[Indice, Dato], Indice) → Bool /* obs */
    ecuaciones
      ∀ v : Vector[Indice, Dato] : ∀ i, j : Indice: ∀ x, y : Dato :
      Asigna(Asigna(v, i, x), j, y) = Asigna(v, j, y)               si i == j
      Asigna(Asigna(v, i, x), j, y) = Asigna(Asigna(v, j, y), i, x)    si i
/= j
      def?(Crea, j)                 = falso
      def?(Asigna(v, i, x), j)      = i == j OR def?(v, j)
      def valor(v, i) si def?(v, i)
      valor(Asigna(v, i, x), j)     = x                             si i == j
      valor(Asigna(v, i, x), j)     =f valor(v, j)                  si i /= j
    errores
      valor(v, i) si NOT def?(v, i)
ftad
```

Podemos establecer la siguiente correspondencia entre las operaciones de los vectores y las tablas, que nos sugiere cómo los vectores se pueden implementar de manera directa en forma de tablas

VECTOR[I :: DIS, D :: ANY]	TABLA[E :: EQ, V :: EQ-ND]
Crea( )	TablaVacía( )

Asigna(xs, i, x)	Inserta(xs, i, x)
valor(xs, i)	consulta(xs, i)

Al entender a los vectores como TAD se pueden plantear implementaciones diferentes de la predefinida, ganando en eficiencia en ciertos casos, como por ejemplo los vectores dispersos.

6.2 Implementación con acceso basado en búsqueda

Utilizando las estructuras de datos que ya conocemos, podemos plantear diversas implementaciones para el TAD TABLA como colecciones de parejas (Clave, Valor), donde el acceso por clave se implementa como una búsqueda en la colección:

— Vectores de parejas (clave, valor), desordenados u ordenados por clave. Las complejidades que se pueden obtener para las operaciones:

Operación	Vector desordenado	Vector ordenado
TablaVacía	O(1)	O(1)
Inserta	O(n) / O(1) **	O(n)
esVacía	O(1)	O(1)
está	O(n)	O(log n)
consulta	O(n)	O(log n)
borra	O(n)	O(n)

** La complejidad de la inserción en un vector desordenado es *O*(n) si no permitimos claves repetidas y *O*(1) si las permitimos, con el consiguiente desaprovechamiento de espacio y haciendo necesario que las búsquedas comiencen por el final de la zona ocupada del vector.

— Secuencias de parejas (clave, valor) implementadas con memoria dinámica, desordenadas u ordenadas por clave. Las complejidades que se pueden obtener para las operaciones:

Operación	Secuencia desordenada	Secuencia ordenada
TablaVacía	O(1)	O(1)
Inserta	O(n) / O(1) **	O(n)
esVacía	O(1)	O(1)
está	O(n)	O(n)
consulta	O(n)	O(n)
borra	O(n)	O(n)

** De nuevo, la complejidad de la inserción en un secuencia desordenada es *O*(n) si no permitimos claves repetidas y *O*(1) si las permitimos, con el consiguiente desaprovechamiento de espacio y haciendo necesario que las búsquedas tengan en cuenta esta circunstancia. En el caso de las secuencias ordenadas la complejidad de la búsqueda mejora en el caso promedio, aunque no así en el caso peor.

— Arboles de búsqueda. Suponiendo árboles equilibrados se puede conseguir:

Operación	Arbol de búsqueda
TablaVacía	O(1)
Inserta	O(log n)
esVacía	O(1)
está	O(log n)
consulta	O(log n)
borra	O(log n)

6.3 Implementación con acceso casi directo: tablas dispersas

En todas las implementaciones consideradas en el apartado anterior, las operaciones de inserción y consulta requieren búsqueda. En este apartado estudiamos técnicas que permiten realizar todas las operaciones en tiempo *O*(1) en promedio, aunque en el caso peor *Inserta*, *está*, *consulta* y *borra* son *O*(n).

En los vectores, que son estructuras típicas de acceso directo, todas las operaciones se ejecutan en tiempo constante. La idea es conseguir algo parecido para las tablas, tratando las claves como índices de un vector. Esta idea no puede aplicarse directamente cuando el conjunto de todas las claves posibles sea demasiado grande.

Por ejemplo, si las claves son cadenas de caracteres con un máximo de 8 caracteres elegidos de un conjunto de 52 caracteres, habría un total de

$$L = \Sigma\, i : 1 \leq i \leq 8 : 52^i$$

siendo 52^i el número de cadenas distintas con i caracteres. En una aplicación práctica, la cantidad de cadenas que lleguen a usarse como claves será mucho menor, y es absolutamente impensable reservar un vector de tamaño L para implementar la tabla.

Lo que sí tiene sentido es trata de representar una tabla como un vector de tipo

```
Vector [0..N-1] de Pareja[Clave, Valor]
```

siendo N suficientemente grande, aunque mucho menor que el número total de claves posibles.

Función de localización

Para operar con la tabla se necesitará interponer una función que asocie a cada clave un índice del vector:

```
h : Clave → [0..N-1]
```

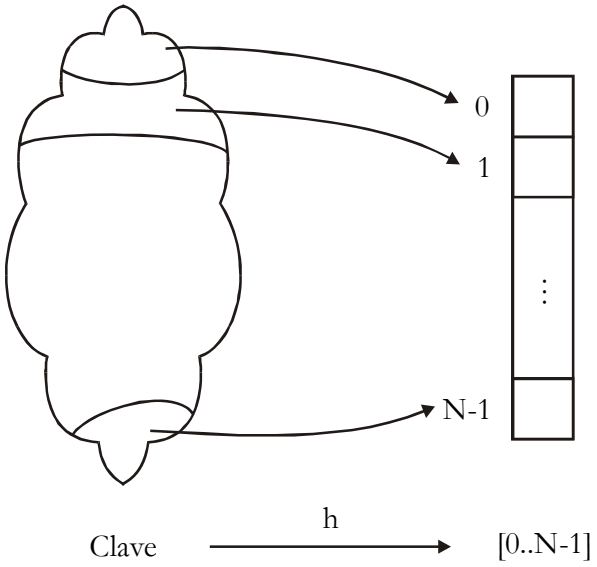
Dada una clave *c*, el índice *h(c)* es la posición del vector en la que en un principio intentaremos *localizar*la clave. La función *h* que usemos para esto se llamará *función de localización* (en inglés, *hashing function*).

Al ser el número total de claves posibles mayor que *N*, *h* no puede ser inyectiva. Se deberá procurar que se cumplen las dos condiciones siguientes:

— Eficiencia. El coste de calcular *h(c)* para una clave *c* dada debe ser bajo.

— Uniformidad. El reparto de claves entre posiciones debe ser lo más uniforme posible. Idealmente, para una clave *c* elegida al azar la probabilidad de que *h(c) = i* debe valer *1/N* para cada *i ∈ [0..N–1]*. Una función de localización que cumpla esta condición se llama uniforme.

Gráficamente, la situación es:



Por ejemplo, supongamos que *N = 16* y que las claves son cadenas de caracteres. Una posible función de localización es la definida como:

```
h(c) =def ord(ult(c)) mod 16
```

donde *ult(c)* es el último carácter de *c*, y *ord* es la función que devuelve el código ASCII de un carácter. Usando esta función de localización, obtenemos:

```
h("Fred") = ord('d') mod 16 = 100 mod 16 = 4
h("Joe") = ord('e') mod 16 = 101 mod 16 = 5
h("John") = ord('n') mod 16 = 110 mod 16 = 14
```

aunque esta función de localización no es muy buena, la seguiremos usando para algunos ejemplos debido a su sencillez. Más adelante presentaremos algunos métodos para definir buenas funciones de localización.

Colisiones

Como hemos visto, el dominio de una función de localización siempre suele tener un cardinal mucho mayor que su rango. Por lo tanto, una función de localización *h* no puede ser inyectiva.

Cuando se encuentran claves *c, c'* tales que:

$$c \neq c' \wedge h(c) = h(c')$$

se dice que se ha producido una *colisión*. Se dice también que *c* y *c'* son claves *sinónimas* con respecto a *h*.

Por ejemplo, para la anterior función de colisión:

h("Fred") = h("David") = h("Violet") = h("Roland") = 4

las cuatro cadenas colisionan en el índice 4 y son sinónimas con respecto a *h*.

La probabilidad de que no se produzcan colisiones es muy baja. Usando cálculo de probabilidades puede demostrarse la llamada "paradoja de los cumpleaños", que dice: en un grupo de 23 o más personas, la probabilidad de que al menos dos de ellas tengan su cumpleaños en el mismo día del año es mayor que 1/2.

Tablas dispersas

Se llaman así a las tablas implementadas de manera que:

— La representación concreta de una tabla es un vector *t : **Vector** [0..N–1] **de** Pareja[Clave, Valor]*

— Se utiliza una función de localización para el paso de claves a índices.

Las distintas implementaciones de tablas dispersas se diferencian en dos cosas:

— El método elegido para el tratamiento de colisiones, cuando éstas se presenten.

— La función de localización elegida

Vamos a centrarnos primero en estudiar la primera de las características, suponiendo fijada una función de localización *h*, y estudiaremos la segunda característica más adelante. Según la técnica que se utilice para el tratamiento de las colisiones, hablamos de:

— tablas abiertas, y

— tablas cerradas

6.3.1 Tablas dispersas abiertas

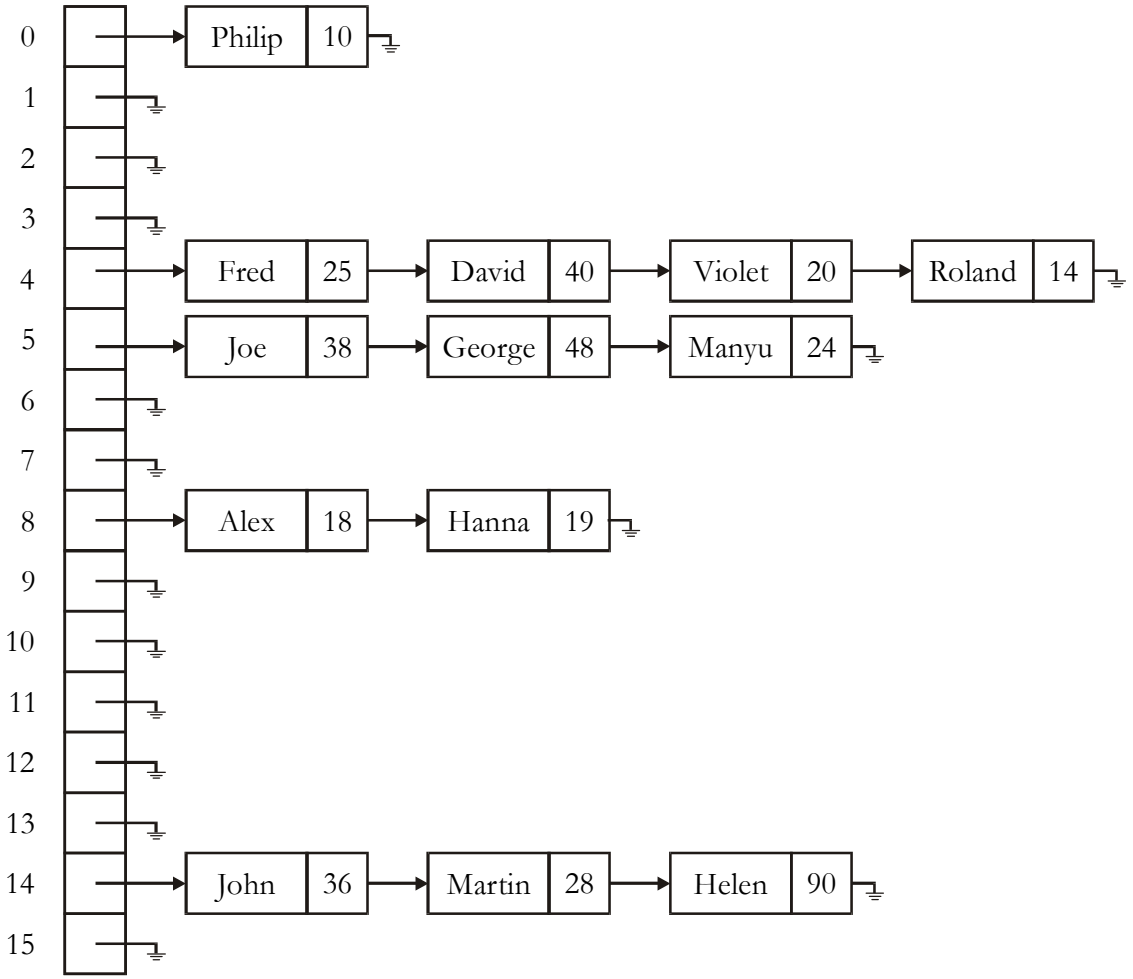
Para cada índice *i*, *t(i)* almacena una lista de parejas (Clave, Valor) con claves sinónimas *c*, tal que *h(c) = i*. Estas listas se llaman *agrupaciones* o, simplemente, *listas de colisiones*. Las colisiones que se producen durante una operación de inserción se resuelven fácilmente alargando las agrupaciones. Durante las operaciones de borrado, las agrupaciones se acortan.

Por ejemplo, una tabla dispersa abierta implementada sobre un vector con índices del intervalo *[0..15]* y donde se utilizada la función de localización presentada anteriormente, después de las siguientes operaciones:

```
TablaVacía(t);
inserta(t, "Fred", 25); inserta(t, "Alex", 18); inserta(t, "Philip", 10);
```

```
inserta(t, "Joe", 38); inserta(t, "John", 36); inserta(t, "Hanna", 19);
inserta(t, "David", 40); inserta(t, "Martin", 28); inserta(t, "Violet", 20);
inserta(t, "George", 48); inserta(t, "Helen", 90); inserta(t, "Manyu", 24);
inserta(t, "Roland", 14);
```

resulta:



Aunque hemos dibujado las listas con punteros por mayor claridad, no presuponemos nada sobre la implementación de las agrupaciones. Nótese que la inserción en las listas se realiza por el final porque es necesario comprobar en cada inserción si ya existe la clave.

Una característica interesante de un valor concreto de una tabla dispersa es su *tasa de ocupación*, que en el caso de las tablas abiertas de define como el cociente entre el número de parejas (Clave, Valor) almacenadas en la tabla, y el número total de posiciones del vector.

La situación final:

— Tamaño del vector $N = 16$

— Número de parejas almacenadas $n = 13$

— Tasa de ocupación $\alpha = n/N = 13/16 = 0'8125$

La realización de las operaciones de las tablas en una tabla abierta es sencilla, empleando técnicas básicas de procesamiento de listas. Las colisiones se manejan fácilmente. Otra "ventaja"

es que la tabla puede llegar a almacenar más de *N* parejas, aunque en este caso la rapidez de los accesos puede degenerar, haciéndose *O*(n) como en los métodos de búsqueda secuencial.

Implementación de las tablas dispersas abiertas

Tipo representante

```
módulo impl TABLA[CLAVE,VALOR]
  importa
    EQ, EQ-ND, SEC[ELEM]
  privado
    const
      N = ??;
    tipo
      Clave = EQ.Elem;
      Valor = EQ-ND.Elem;
      Indice = [0..N-1];
      Pareja = reg
        clave : Clave;
        valor : Valor
      freg;
      Agrupación = SEC.Sec[Pareja];
      Tabla[Clave, Valor] = Vector Indice de Agrupación

% implementación de las operaciones

fmódulo
```

Invariante de la representación

Para *t : Tabla[Clave, Valor]*:

```
      R(t)
⇔def
      ∀ i : 0 ≤ i < N :
        ( ( t(i) : Agrupación ) ∧
          ( ∀ j : 1 ≤ j ≤ #cont(t(i)) : h((cont(t(i)) !! j).clave) = i ) ∧
          ( ∀ j, k : 1 ≤ j < k ≤ #cont(t(i)) : (cont(t(i)) !! j).clave ≠
                                               (cont(t(i)) !! k).clave   )
∧
          ( ∀ j : 1 ≤ j ≤ #cont(t(i)) : (cont(t(i)) !! j).valor ≠ NoDef ) )
```

Donde se expresa que *t(i)* es tipo Agrupación, cada pareja de *t(i)* tiene una clave *c* tal que *h(c)* = *i*, parejas diferentes de t(i) tienen diferente clave y cada pareja de t(i) tiene un valor ≠ NoDef.

Función de abstracción

Para *t : Tabla[Clave, Valor]* tal que R(t):

$$A(t) =_{def} hazTabla(\ t,\ 0\ )$$

$$hazTabla(t,\ i) =_{def} \begin{cases} TablaVacía & \textbf{si } i = N \\ añadeAgrupación(\ cont(t(i)),\ hazTabla(t,\ i+1)\ ) & \textbf{si } i < N \end{cases}$$

$$añadeAgrupación([],\ t) =_{def} t$$

$$añadeAgrupación([x/xs],\ t) =_{def} añadeAgrupación(xs,\ Inserta(t,\ x.clave,\ x.valor))$$

Implementación de las operaciones

Suponemos disponible una función de localización, implementada como una función privada:

```
h: Clave → Indice
```

La construcción de una tabla vacía:

```
proc TablaVacía( s t : Tabla[clave, Valor] );
{ P₀ : cierto }
var
  i : Indice;
inicio
  para i desde 0 hasta N−1 hacer
    SEC.Crea(t(i))
  fpara
{ Q₀ : R(t) ∧ A(t) =TABLA[CLAVE, VALOR] TablaVacía }
```

Función que detecta si una tabla está vacía.

```
func esVacía( t : Tabla[Clave, Valor] ) dev vacía : Bool;
{ P₀ : R(t) }
var
  i : Indice;
inicio
  vacía := cierto;
  i := 0;
  it vacía AND i < N →
    vacía := SEC.vacía?( t(i) );
    i := i+1
  fit
{ Q₀ : vacía ↔ A(t) =TABLA[CLAVE,VALOR] TablaVacía }
  dev vacía
```

```
ffunc
```

El resto de las operaciones se realiza con ayuda de un procedimiento privado auxiliar que realiza una búsqueda en una agrupación, una versión del algoritmo de búsqueda en una secuencia:

```
proc busca ( e clave : Clave; es xs : Agrupación; s encontrado : Bool );
{ P₀ : xs = XS }
var
  x : Pareja;
inicio
  SEC.reinicia(xs);
  encontrado := falso;
  it NOT SEC.fin?(xs) AND NOT encontrado →
    x := SEC.actual(xs);
    si EQ.igual( x.clave, clave )
      entonces encontrado := cierto
      sino     SEC.avanza(xs)
    fsi
  fit
{ Q₀ : cont(xs) = cont(XS) ∧
       encontrado ↔ la clave de un elemento de xs coincide con clave ∧
       encontrado → el actual de xs es el primero cuya clave coincide con
clave }
fproc
```

Disponiendo de *busca* la realización de *Inserta*, *está, consulta* y *borra* resulta inmediata. Veamos cómo es la implementación de *Inserta*

```
proc Inserta( es t : Tabla[Clave, Valor]; e c : Clave; e v : Valor );
{ P₀ : t = T ∧ R(t) }
var
  i : Indice;
  encontrado : Bool;
  par : Pareja;
inicio
  i := h(clave);
  busca( clave, t(i), encontrado );
  si encontrado
    entonces
      SEC.borra(t(i))
    sino
      seguir
  fsi;
  par.clave := clave;
  par.valor := valor;
  SEC.Inserta(t(i), par)
{ Q₀ : R(t) ∧ A(t) =TABLA[CLAVE,VALOR] Inserta(A(T), A(c), A(v)) }
fproc
```

6.3.2 Tablas dispersas cerradas

La idea básica de esta representación es que la información almacenada en la tabla se representa por medio de un vector de parejas (Clave, Valor). Cualquier acceso a una tabla vía una clave dada *c* comienza calculando el llamado *índice primario*:

$i_0 = h(c)$

Si el índice primario produce colisión con otra clave sinónima, se ensaya un índice alternativo. Como la colisión puede reiterarse, es necesario ensayar una sucesión de índices:

$i_1 = prueba(1, c),\ i_2 = prueba(2, c),\ \ldots,\ i_m = prueba(m, c),\ \ldots$

llamada *sucesión de pruebas*. A esta técnica se la conoce como *relocalización*. Una técnica de relocalización concreta debe cumplir que para toda clave *c* se puede calcular la sucesión de pruebas:

$i_m = prueba(m, c) \quad 0 \le m < N$

que es una permutación de *[0..N–1]*. Por lo tanto, las sucesiones de pruebas de dos claves distintas sólo difieren en el orden, pero no en los elementos.

Por ejemplo, un método muy sencillo de relocalización consiste en definir la sucesión de pruebas como:

$i_0 = h(c)$

$i_m = (i_{m-1} + 1) \bmod N \qquad 1 \le m < N$

o lo que es lo mismo

$prueba(m, c) = (h(c) + m) \bmod N$

Este método es conocido como *relocalización lineal*.

Las tablas implementadas en un vector de parejas, con ayuda de una función de localización y alguna técnica de tratamiento de colisiones mediante sucesión de pruebas, se llaman *tablas dispersas cerradas*.

Claves ficticias

Como veremos a continuación, los algoritmos de acceso a tablas dispersas cerradas necesitan distinguir 3 clases de posiciones en el vector de parejas:

— Posiciones *vacías*: aún no se ha introducido información.

— Posiciones *ocupadas*: contienen una pareja (Clave, Valor).

— Posiciones *borradas*: han contenido información que posteriormente se borró. La razón de no marcar estas posiciones como vacías es que interrumpirían la sucesión de pruebas, u obligarían a reorganizaciones del vector después de cada borrado.

Para marcar los distintos tipos de posiciones del vector, podemos utilizar un campo adicional en cada uno de los registros almacenados en él, o suponer que el tipo de las claves está equipado con dos valores especiales que denominados *claves ficticias*

```
ClaveVacía, ClaveBorrada : Clave
```

diferentes entre sí. Llamaremos *claves ordinarias* a las que sean distintas de estas dos. La función *prueba* no está definida sobre las claves ficticias.

Si suponemos disponible una función

```
clave: (Tabla[Clave, Valor], [0..N-1]) → Clave
```

que calcula la clave de la posición i-ésima de la tabla, podemos definir predicados que caracterizen a los distintos tipos de posiciones de una tabla dispersa cerrada:

```
vacía(t, i)      ⇔def clave(t, i) == ClaveVacía
borrada(t, i)    ⇔def clave(t, i) == ClaveBorrada
ocupada(t, i)    ⇔def NOT vacía(t, i) AND NOT borrada(t, i)
usada(t, i)      ⇔def NOT vacía(t, i) ⇔ ocupada(t, i) OR borrada(t, i)
disponible(t, i) ⇔def NOT ocupada(t, i) ⇔ vacía(t, i) OR borrada(t, i)
```

Tamaño y tasa de ocupación

En la definición del tamaño de una tabla cerrada, debemos considerar todas las posiciones *usadas –ocupadas* o *borradas–*. Dada *t : Tabla[Clave, Valor],* definimos el *tamaño n* = |*t*| como el número de posiciones usadas:

```
|t| =def #i : 0 ≤ i < N : usada(t, i)
```

La tasa de ocupación en una tabla dispersa cerrada viene dada entonces por:

```
α = n/N
```

número de posiciones ocupadas entre número de posiciones del vector. Obviamente, en una tabla cerrada $0 \le n \le N$, y con ello $0 \le \alpha \le 1$.

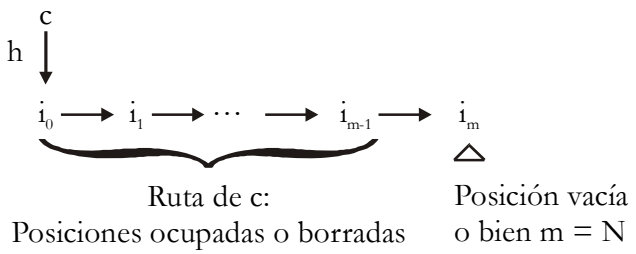
Idea de los algoritmos

Todos los algoritmos sobre tablas se basan en una característica que debe cumplir el invariante de la representación: cada clave ordinaria *c* que aparezca en la tabla aparece necesariamente en una posición de la *ruta* de c, definida de la siguiente forma:

Dadas *t :Tabla[clave, Valor]* y *c : Clave*, definimos la *ruta* de *c* como el segmento inicial de la sucesión de pruebas de *c* anterior a la primera posición vacía (si la hay):

```
ruta(c, t) =def [prueba(0, c), … , prueba(m-1, c)]
```

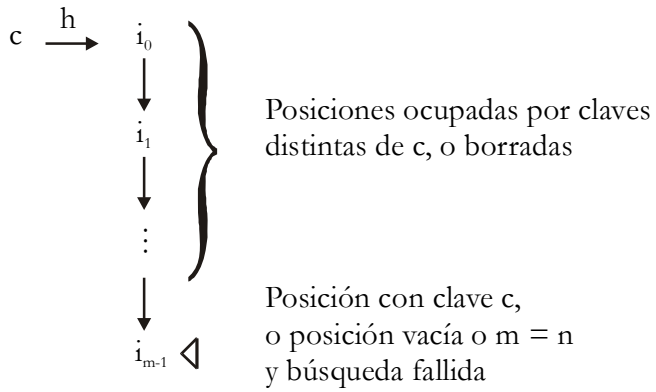
siendo *m* el menor *i* comprendido entre 0 y *N–1* tal que *vacía(t, prueba(i, c))*, si existe; si no existe, tomamos *m* = *N*.



Observamos que *ruta(c, t)* = *[ ]* cuando se cumple *vacía(t, prueba(0, c))*; i.e., cuando el índice primario de *c* corresponde a una posición vacía.

Búsqueda

Partiendo del índice primario, se sigue la sucesión de pruebas hasta encontrar la clave buscada o una posición vacía. Las posiciones borradas no interrumpen la búsqueda:



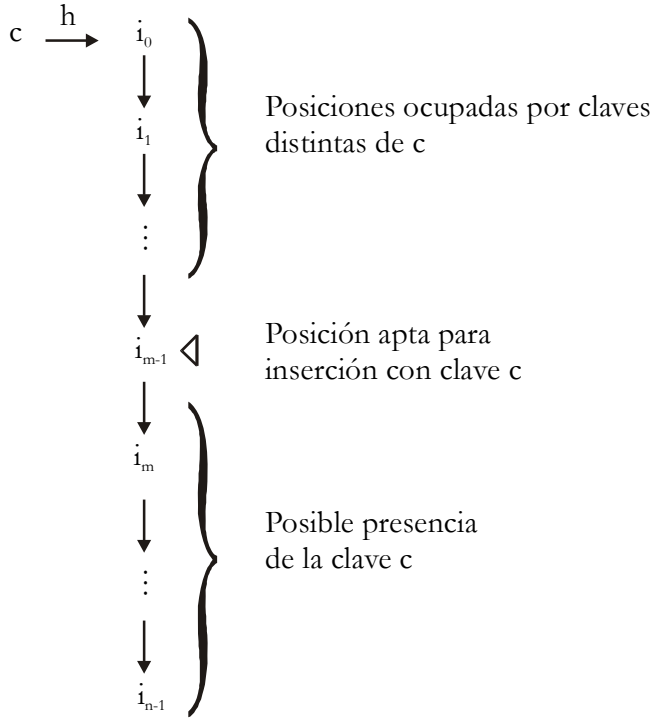
Inserción

Partiendo del índice primario, se sigue la sucesión de pruebas hasta agotar la tabla o encontrar una posición apta para la inserción. Si se agota la tabla, se produce error. Si se encuentra una posición apta, debe darse uno de los tres casos siguientes:

— La posición está vacía. Entonces se inserta en esa posición y se incrementa *n* en 1.

— La posición está ocupada por la clave de inserción. Entonces se cambia el valor asociado.

— La posición está borrada. Se colocan la clave y el valor asociado. Se sigue buscando en la serie de pruebas, hasta encontrar una posición vacía, o encontrar la clave de inserción, o completar *n* pruebas. Si se encuentra la clave de inserción, hay que borrar la posición correspondiente.



Consulta

Se realiza una búsqueda con la clave de consulta. Si se encuentra la clave, se devuelve el valor asociado. En otro caso, se devuelve *NoDef*.

Borrado

Se realiza una búsqueda con la clave de borrado. Si la búsqueda tiene éxito, la posición correspondiente se borra, i.e., se pone *ClaveBorrada*.

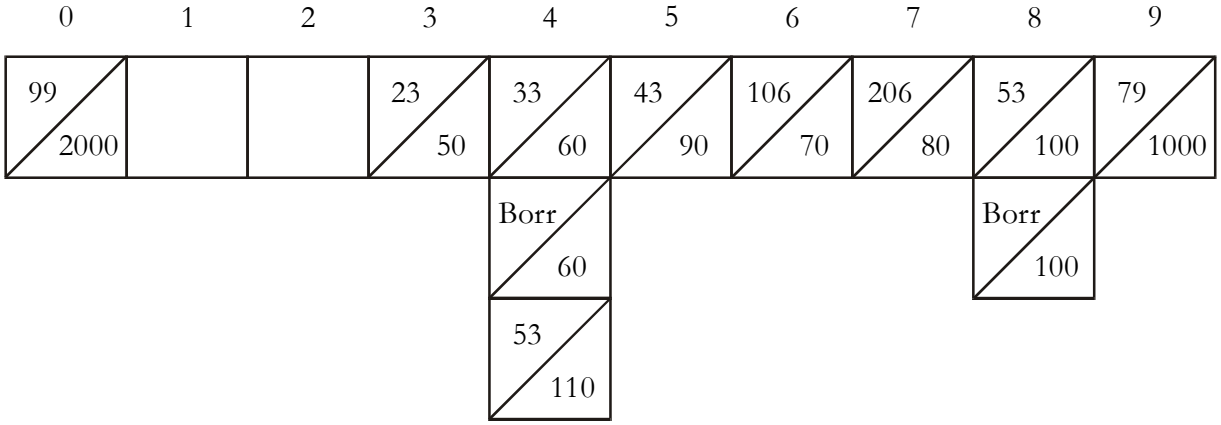
Ejemplo

Veamos cómo evoluciona una tabla dispersa cerrada, con índices en el intervalo [0..9], usando relocalización lineal y con una función *h* de localización definida como

```
h(c) = c mod 10
```

ante la serie de llamadas:

Operación	Serie de pruebas	resultado
`TablaVacía(t);`		
`Inserta(t, 23, 50);`	`3`	
`Inserta(t, 33, 60);`	`3, 4`	
`Inserta(t, 106, 70);`	`6`	
`x := consulta(t, 53);`	`3, 4, 5`	⇒ `NoDef`
`Inserta(t, 206, 80);`	`6, 7`	
`Inserta(t, 43, 90);`	`3, 4, 5`	
`y := consulta(t, 33);`	`3, 4`	⇒ `60`
`Inserta(t, 53, 100);`	`3, 4, 5, 6, 7, 8`	
`borra(t, 33);`	`3, 4`	
`Inserta(t, 53, 110);`	`3, 4, 5, 6, 7, 8`	
`Inserta(t, 79, 1000);`	`9`	
`Inserta(t, 99, 2000);`	`9, 0`	
`z := consulta(t, 53);`	`3, 4`	⇒ `110`
`u := consulta(t, 109);`	`9, 0, 1`	⇒ `NoDef`



Con lo que la tasa de ocupación resultante queda:

α = número de posiciones usadas / tamaño del vector = 8 / 10 = 0’8

Implementación de las tablas dispersas cerradas

Tipo representante

```
módulo impl TABLA[CLAVE,VALOR]
  importa
    EQ, EQ-ND
  privado
    const
      N = ??;            % Capacidad de la tabla; se recomienda primo > 20
    tipo
      Clave = EQ.Elem;
      Valor = EQ-ND.Elem;
      Indice = [0..N-1];
      IndiceN = [0..N];
      Pareja = reg
        clave : Clave;
        valor : Valor
      freg;
      Tabla[Clave, Valor] = reg
        tamaño : IndiceN;
        parejas : Vector Indice de Pareja
      freg;
```

Donde hemos optado por suponer que el tipo Clave dispone de dos *claves ficticias*

```
ClaveVacía, ClaveBorrada : Clave
```

Invariante de la representación

Dada *t : Tabla[Clave, Valor]*

$R(t) \Leftrightarrow_{def} I_0 \wedge I_1 \wedge I_2 \wedge I_3$

siendo:

$I_0 \Leftrightarrow_{def}$ *t.tamaño* = $|t|$ (número de posiciones usadas)

$I_1 \Leftrightarrow_{def}$ el valor ficticio *NoDef* no aparece en *t.parejas*

$I_2 \Leftrightarrow_{def}$ cada clave ordinaria *c* aparece en *t.parejas* a lo sumo en una posición

$I_3 \Leftrightarrow_{def}$ cada clave ordinaria *c* que aparezca en *t.parejas* lo hace en una posición perteneciente a *ruta(c, t)*

Función de abstracción

Dada *t : Tabla[Clave, Valor]* tal que R(t):

```
A(t) =def hazTabla( t, 0 )

                    ⎧ TablaVacía                          si i = N
hazTabla(t, i) =def ⎨ hazTabla(t, i+1)                    si i < N AND NOT
ocupada(t, i)       ⎩
                      Inserta(hazTabla(t, i+1), t.parejas(i).clave,
                              t.parejas(i).valor )     si i < N AND ocupada(t,
i)
```

Implementación de las operaciones

Suponemos que se ha fijado un método de tratamiento de colisiones y que se dispone de una función que calcula el índice de lugar *m* de la sucesión de pruebas de una clave dada *c*:

```
func prueba ( m : Indice; c : Clave ) dev i : Indice;
{ P0 : cierto }
{ Q0 : i es el m-ésimo índice de la sucesión de pruebas de c }
ffunc
```

Disponemos también de una función auxiliar privada que permite comparar la clave de una posición dada de una tabla con una clave dada:

```
func igualClave( t : Tabla[Clave, Valor]; i : Indice; c : Clave ) dev r :
Bool;
inicio
  r := EQ.igual( t.parejas(i).clave, c )
  dev r
ffunc
```

Y usando la anterior, funciones que nos permiten determinar el tipo de una posición: *vacía, borrada, ocupada, usada, disponible*.

Creación de una tabla vacía:

```
proc TablaVacía( s t : Tabla[Clave, Valor] );
{ P₀ : cierto }
var
  i : Indice;
inicio
  t.tamaño := 0;
  para i desde 0 hasta N-1 hacer
    t.parejas(i).clave := ClaveVacía
  fpara
{ Q₀ : R(t) ∧ A(t) = TablaVacía }
fproc
```

Función que detecta si una tabla está vacía.

```
func esVacía( t : Tabla[Clave, Valor] ) dev vacía : Bool;
{ P₀ : R(t) }
var
  i : Indice;
inicio
  vacía := cierto;
  i := 0;
  it vacía AND i < N →
    vacía := disponible(t, i);
    i := i+1
  fit
{ Q₀ : vacía ↔ A(t) =TABLA[CLAVE,VALOR] TablaVacía }
  dev vacía
ffunc
```

Las operaciones *está, consulta* y *borra* se programan con ayuda de una función auxiliar de búsqueda, que llamaremos *busca*. La idea es seguir la sucesión de pruebas hasta encontrar la clave buscada o una posición vacía. Las posiciones borradas no interrumpen la búsqueda.

```
func busca ( t : Tabla[Clave, Valor]; c : Clave ) dev i : IndiceN;
{ P₀ : R(t) ∧ c no es una clave ficticia }
var
  m, lm : Indice;
inicio
  m := 0;
  i := hash(c);                % i = prueba(0, c)
```

```
  lm := t.tamaño – 1;                 % lm acota la longitud de la ruta de
c
  it m /= lm AND NOT vacía(t, i) AND NOT igualClave(t, i, c) →
    m := m+1;
    i := prueba(m, c)
  fit;
  si igualClave(t, i, c)
    entonces                    % clave encontrada
      seguir
    sino
      i := N                    % clave no encontrada
  fsi
{ Q₀ : ( NOT está(A(t), c) ∧ i = N ) ∨
       ( está(A(t), c) ∧ t.parejas(i).clave == c ) }
  dev i
ffunc
```

Utilizando *busca* resulta inmediata la implementación de *está, consulta* y *borra.*
Finalmente, el algoritmo de inserción

```
proc Inserta( es t : Tabla[Clave, Valor]; e c : Clave; e v : Valor );
{ P₀ : t = T ∧ R(t) ∧ c no es una clave ficticia ∧
       ( está(A(t), c) ∨ ∃ i : 0 ≤ i ≤ N–1 : disponible(t, i) ) }
var
  m, lm, i : Indice;
  n : IndiceN;
inicio
  n := t.tamaño;
  m := 0; i := hash(c);   % i = prueba(0, c)
  lm := N–1;                    % acota la longitud de la sucesión de
pruebas
  it m /= lm AND ocupada(t, i) AND NOT igualClave(t, i, c) →
    m := m+1;
    i := prueba(m, c)
  fit;
  si
    ocupada(t, i) AND NOT igualClave(t, i, c)
    → error("Tabla llena")
  □ igualClave(t, i, c)
    → t.parejas(i).valor := v
  □ vacía(t, i)
    → t.parejas(i).clave := c;
      t.parejas(i).valor := v;
      t.tamaño := n+1
  □ borrada(t, i)
    → t.parejas(i).clave := c;
      t.parejas(i).valor := v;
      lm := n–1;        % lm acota la longitud de la ruta de c
```

```
        si m == lm
          entonces
            seguir
          sino
            m := m+1; i := prueba(m, c);
            it m /= lm AND NOT vacía(t, i) AND NOT igualClave(t, i, c) →
              m := m+1; i := prueba(m, c)
            fit;
            si igualclave(t, i, c)
              entonces
                t.parejas(i).clave := ClaveBorrada
              sino
                seguir
            fsi
        fsi
    fsi
{ Q₀ : R(t) ∧ A(t) =TABLA[CLAVE,VALOR] Inserta(A(T), A(c), A(v)) }
fproc
```

En la práctica, la relocalización no se implementa mediante llamadas a una función *prueba*, como hemos supuesto en las anteriores implementaciones, sino que el cálculo de los índices de la sucesión de pruebas se incorpora dentro del código de la función *busca* y el procedimiento *Inserta*, del modo adecuado a la técnica de relocalización que se esté utilizando. Presentamos a continuación las versiones de *busca* e *Inserta* que incorporan relocalización lineal sin llamadas a una función *prueba*.

```
func busca ( t : Tabla[Clave, Valor]; c : Clave ) dev i : IndiceN;
{ P₀ : R(t) ∧ c no es una clave ficticia }
var
  li : Indice;
inicio
  i := hash(c);
  li := (i + t.tamaño – 1) mod N;            % acota el último índice en la
ruta de c
  it i /= li AND NOT vacía(t, i) AND NOT igualClave(t, i, c) →
    i := (i + 1) mod N
  fit;
  si igualClave(t, i, c)
    entonces                    % clave encontrada
      seguir
    sino
      i := N                    % clave no encontrada
  fsi
{ Q₀ : ( NOT está(A(t), c) ∧ i = N ) ∨
       ( está(A(t), c) ∧ t.parejas(i).clave == c ) }
  dev i
ffunc
```

Y el algoritmo de inserción

```
proc Inserta( es t : Tabla[Clave, Valor]; e c : Clave; e v : Valor );
{ P₀ : t = T ∧ R(t) ∧ c no es una clave ficticia ∧
       ( está(A(t), c) ∨ ∃ i : 0 ≤ i ≤ N–1 : disponible(t, i) ) }
var
  i0, i, li : Indice;
  n : IndiceN;
inicio
  n := t.tamaño;
  i0 := hash(c);
  i := i0;
  li := (i0 + N – 1) mod N;          % acota el último índice de la
sucesión de pruebas
  it i /= li AND ocupada(t, i) AND NOT igualClave(t, i, c) →
    i := (i + 1) mod N
  fit;
  si
    ocupada(t, i) AND NOT igualClave(t, i, c)
    → error("Tabla llena")
  □ igualClave(t, i, c)
    → t.parejas(i).valor := v
  □ vacía(t, i)
    → t.parejas(i).clave := c;
       t.parejas(i).valor := v;
       t.tamaño := n+1
  □ borrada(t, i)
    → t.parejas(i).clave := c;
       t.parejas(i).valor := v;
       li := (i0 + n – 1) mod N;% acota el último índice en la ruta de c
       si i == li
         entonces
           seguir
         sino
           i := (i + 1) mod N
           it i /= li AND NOT vacía(t, i) AND NOT igualClave(t, i, c) →
             i := (i + 1) mod N
           fit;
           si igualclave(t, i, c)
             entonces
               t.parejas(i).clave := ClaveBorrada
             sino
               seguir
           fsi
       fsi
```

```
  fsi
{ Q₀ : R(t) ∧ A(t) =TABLA[CLAVE,VALOR] Inserta(A(T), A(c), A(v)) }
fproc
```

6.3.3 Funciones de localización y relocalización

Métodos de localización

Como se ha dicho más atrás, una buena función de localización debe ser uniforme y fácil de calcular eficientemente. En general, lo más recomendable es calcular el resto módulo el tamaño *N* de la tabla. Es decir:

— Si la clave es un número natural *c*

$$h(c) =_{def} c \bmod N$$

— Si la clave es una cadena de caracteres

$$c = c_1\ c_2\ \ldots\ c_k$$

la idea es parecida, primero se transforma *c* en un número interpretando sus caracteres como dígitos en base *B*:

$$h(c) =_{def} num(c) \bmod N$$

$$num(c) =_{def} \Sigma\ i : 0 \le i < k : B^i \cdot ord(c_{k-i})$$
$$= ord(c_k) + B \cdot ord(c_{k-1}) + \ldots + B^{k-1} \cdot ord(c_1)$$

Conviene que *N* sea un número primo mayor que 20. No conviene que *N* sea una potencia de la base de numeración *B*, pues esto tiende a hacer colisionar las claves cuyos códigos numéricos coinciden en los dígitos de menor peso.

El código de *h(c)* se puede hacer más eficiente expresándolo como

$$h(c) = (num(c) \bmod 2^w) \bmod N = ((\Sigma\ i : 0 \le i < k : B^i \cdot ord(c_{k-i})) \bmod 2^w) \bmod N$$

siendo *w* el número de bits de una palabra máquina. Así, el resto *mod* 2^w puede ser ejecutado muy rápidamente por la circuitería.

Se recomienda el valor $B = 131$ porque para este valor B^i tiene un ciclo máximo *mod* 2^k para $8 \le k \le 64$.

Métodos de relocalización

Cada uno se caracteriza por una técnica diferente para el cálculo de la sucesión de pruebas. Desde el punto de vista de la eficiencia, lo deseable es que un método de relocalización desacople lo más posible las sucesiones de pruebas.

Relocalización lineal

En inglés *Linear probing hashing*.

La serie de pruebas es:

$$prueba(m, c) =_{def} (h(c) + m) \bmod N \qquad 0 \le m < N$$

Se cumple:

$$i_0 = h(c)$$
$$i_m = (i_{m-1} + 1) \bmod N$$

El primer índice determina la serie de pruebas. Se dice por esto que hay *agrupamientos primarios*. Hay una única sucesión de pruebas, módulo permutaciones cíclicas.

En general, se dice que un método de relocalización produce agrupamientos *k-arios* si el número de series de pruebas, módulo permutaciones circulares es $\Theta(N^{k-1})$; o equivalentemente, si cada serie de pruebas está determinada por sus *k* primeros índices.

En la relocalización lineal, el hecho de que los agrupamientos correspondientes a diferentes claves pueden llegar a solaparse, junto con la presencia de posiciones borradas, degrada la eficiencia de los accesos.

Relocalización cuadrática

En inglés *Quadratic probing hashing*.

La serie de pruebas es

$$prueba(m, c) =_{def} (h(c) + m^2) \bmod N \qquad 0 \le m < N$$

Para un cálculo más eficiente, se aprovecha que los cuadrados se pueden obtener como suma de impares consecutivos:

$$i_0 = h(c) \qquad j_0 = 1$$
$$i_m = (i_{m-1} + j_{m-1}) \bmod N \qquad j_m = j_{m-1} + 2$$

Con esta construcción, la sucesión de pruebas en general no es una permutación de *[0..N–1]* y, por lo tanto, no recorre toda la tabla; pueden repetirse índices, de manera que la mitad del espacio de la tabla quede sin aprovechar. Se sabe que si *N* es un número primo de la forma *4k+3*, la siguiente sucesión cuadrática de pruebas sí es una permutación de *[0..N–1]*:

$$prueba(0, c) = h(c)$$
$$prueba(2j-1, c) = (h(c) + j^2) \bmod N \qquad 1 \le j \le (N-1)/2$$
$$prueba(2j, c) = (h(c) - j^2) \bmod N \qquad 1 \le j \le (N-1)/2$$

Algunos primos de la forma $N = 4k+3$

k	N	k	N
0	3	10	43
1	7	14	59
2	11	31	127
4	19	62	251
5	23	125	503
7	31	254	1019

En este método se dice que hay *agrupamientos secundarios*, porque los dos primeros índices de una sucesión de pruebas la determinan. Hay $\Theta(N)$ sucesiones de pruebas, módulo permutaciones circulares.

Relocalización doble

En inglés *Double hashing*.

Este método usa una segunda función de localización *k* para calcular un incremento que, junto con el índice primario, determina la sucesión de pruebas:

```
prueba(m, c) = (h(c) + m · k(c)) mod N 0 ≤ m < N
```

Se cumple:

$$i_0 = h(c)$$
$$i_m = (i_{m-1} + d) \bmod N \qquad d = k(c) \in [1..N-1]$$

Con una buena elección de *h, k* el comportamiento de este método es muy similar en la práctica al que resultaría del modelo teórico conocido como *relocalización uniforme* (en inglés, *uniform probing hashing*). Este modelo presupone que la sucesión de pruebas es una permutación aleatoria de *[0..N–1]*, dependiente de la clave de acceso, con lo cual no se producen agrupamientos.

Una buena elección de *N, h, k* es:

— *N* primo, tal que *N–2* primo

— $h(c) =_{def} num(c) \bmod N$

— $k(c) =_{def} (num(c) \bmod N–2) + 1$

Por ser *N* primo, *N* y *d* son primos entre sí, y, por lo tanto, la serie de pruebas recorre toda la tabla –i.e., es una permutación de *[0..N–1]*–.

6.3.4 Eficiencia

Los resultados sobre eficiencia vienen expresados en términos del tamaño *n* (número de parejas en las tablas abiertas y número de posiciones usadas en las tablas cerradas) y la tasa de ocupación $\alpha = n/N$.

En el caso peor, una búsqueda o inserción puede requerir tiempo $O(n)$, tanto en el caso de las tablas abiertas como en el caso de las tablas cerradas.

Las evaluaciones experimentales, sin embargo, muestran para las tablas dispersas un rendimiento superior al de los árboles de búsqueda equilibrados.

Los análisis probabilísticos permiten estimar la complejidad en promedio esperada para los distintos modelos de tabla dispersa. Para realizar esta clase de análisis se hacen los siguientes supuestos:

— La función de localización se supone *uniforme*; es decir, para cada $i \in [0..N–1]$, la probabilidad de que una clave aleatoria *c* cumpla $h(c) = i$ debe ser $1/N$. (En ciertas aplicaciones, por ejemplo en compiladores, no está claro si esta suposición es realista).

— Llamamos C_n al número promedio de accesos a posiciones de una tabla de tamaño *n* para localizar una clave no ficticia presente en la tabla.

— Llamamos C'_n al número promedio de accesos a posiciones de una tabla de tamaño n para insertar con una nueva clave no presente en la tabla.

Se cumple entonces:

(a) Para tablas abiertas

$$C_n \approx 1 + \alpha/2 \qquad C'_n \approx \alpha$$

(b) Para tablas cerradas con relocalización lineal

$$C_n \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \qquad C'_n \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

(c) Para tablas cerradas con relocalización cuadrática o doble:

$$C_n \approx -\left(\frac{1}{\alpha}\right)\ln(1-\alpha) \qquad C'_n \approx \frac{1}{1-\alpha}\alpha$$

Suponiendo $\alpha = 0'8$, las fórmulas aproximadas anteriores quedan:

(a) $C_n \approx 1'4$ $\qquad C'_n \approx 0'8$

(b) $C_n \approx 3$ $\qquad C'_n \approx 13$

(c) $C_n \approx 2'0118$ $\qquad C'_n \approx 5$

Observamos que, para una tasa de ocupación fijada, las tablas abiertas son más eficientes que las cerradas. Como contrapartida, las tablas abiertas consumen más memoria.

En ambos tipos de tablas la eficiencia puede degradarse si la tasa de ocupación aumenta en exceso. Algunas posibilidades para prevenir esta circunstancia son:

— En las tablas cerradas, se puede incluir una operación que "limpie" la tabla, vaciando las posiciones borradas y reestructurando las ocupadas, de forma que todas las informaciones con claves sinónimas queden en posiciones consecutivas de su serie de pruebas, a partir de la posición primaria que corresponda.

— En ambos tipos de tablas, se puede incluir una operación que duplique el tamaño del vector. Para ello, es necesario cambiar la función de localización h a otra con rango doble del anterior, y reorganizar el vector representante de la tabla.

Ambas operaciones son costosas, pero debido a la disminución que producen en la tasa de ocupación, son operaciones beneficiosas desde el punto de vista de un análisis amortizado del tiempo requerido por una serie de operaciones de acceso a la misma tabla.

6.4 Ejercicios

Tablas: Modelos matemáticos y especificación algebraica

344. Construye una especificación algebraica de un TAD TABLA[C :: EQ, V :: ANY] que represente el comportamiento de las tablas entendidas como *funciones parciales* que asignan valores a claves. La operación:

```
consulta : (Tabla[Clave, Valor], Clave) - → Valor
```

será parcial y estará definida solamente cuando la clave de consulta esté en la tabla.

345. Construye una especificación algebraica de un TAD TABLA[C :: EQ, V :: EQ-ND] que represente el comportamiento de las tablas entendidas como *funciones totales* que asignan valores a claves. La operación:

```
consulta : (Tabla[Clave, Valor], Clave) → Valor
```

ahora será total, y devolverá un valor ficticio *NoDef* cuando la clave de consulta no esté en la tabla. Especifica también la clase de tipos EQ-ND como subclase de EQ, de manera que a los tipos de esta clase se les exija disponer de un elemento distinguido *NoDef*.

Casos especiales: Conjuntos y vectores

346. El comportamiento del TAD CJTO[E :: EQ] se puede considerar equivalente al del TAD TABLA[E :: EQ, V :: EQ-ND], suponiendo que *V* sea un tipo especial que sólo contenga los dos valores *Si* y *Nodef*. Razona por qué, estudiando cómo se corresponden las operaciones disponibles de ambos TADs.

347. Diremos que un tipo de datos es *discreto* si el tipo contiene una cantidad finita de valores y es isomorfo a un intervalo *[1..n]* de los enteros, disponiendo de igualdad, orden y operaciones para determinar los valores primero y último y los valores siguiente y anterior de un valor dado. Especifica la clase DIS de los tipos de datos discretos como subclase de la clase de tipos ORD.

348. Construye una especificación algebraica del TAD VECTOR[I :: DIS, D :: ANY] pensado para representar el comportamiento de los vectores, con operaciones que permitan crear un vector vacío, modificar un vector asignando un nuevo dato en la posición de un índice dado, y consultar en un vector el dato asociado a un índice dado. A la vista de las especificaciones, ¿cuáles son las analogías y diferencias entre vectores y tablas?

349. Para la verificación formal de algoritmos en los que intervengan vectores se puede razonar usando la especificación algebraica de estos, teniendo en cuenta que cualquier asignación de la forma *v(i) := e* (donde tanto *i* como *e* pueden ser expresiones compuestas) debe entenderse como una abreviatura de la asignación *v := Asigna(v, i, e)*, que expresa una *modificación global* de *v*. Comprueba que la regla de verificación para asignaciones estudiada en el tema 1.2 es consistente con esta idea, y repasa los ejercicios 21 y 22.

Tablas: Implementaciones con acceso basado en búsqueda

350. Plantea una representación de las tablas basada en vectores que almacenen parejas de tipo (Clave, Valor), realizando las operaciones generadoras y modificadores como procedimientos con un parámetro ***es*** de tipo *Tabla[Clave, Valor]*. Estudia la eficiencia que podría obtenerse para una implementación del TAD TABLA basada en esta representación, considerando por separado los dos casos siguientes:

(a) El invariante de la representación exige que el vector representante se mantenga ordenado en orden creciente de claves (suponiendo que el tipo de las claves sea de la clase ORD).

(b) El invariante de la representación permite que el vector representante se mantenga desordenado.

351. Haz un estudio similar al del ejercicio anterior, considerando ahora una representación de las tablas como listas o secuencias de parejas de tipo (Clave, Valor).

352. Plantea ahora una representación de las tablas como árboles de búsqueda de tipo *Arbus[Clave, Valor]*. ¿Cómo conviene definir la operación ($\oplus$) de combinación de valores, que se usa en caso de inserción con una clave repetida? ¿Qué eficiencia puede esperarse para las operaciones de una implementación del TAD TABLA basada en esta representación, suponiendo que los árboles de búsqueda sean equilibrados?

†353. Para ciertas aplicaciones de las tablas resulta conveniente que exista un orden entre las claves, y que se disponga de una operación que extraiga la información almacenada en la tabla en forma de lista (o secuencia) de parejas de tipo (Clave, Valor), ordenada por claves. En ocasiones, también resulta útil que una inserción con clave ya presente en la tabla combine el nuevo valor con el antiguo, en lugar de limitarse a reemplazar el valor antiguo por el nuevo. Llamaremos *tablas ordenadas* a las tablas que incorporen estas dos ideas. Construye una especificación algebraica de las tablas ordenadas como TAD parametrizado, y desarrolla una implementación basada en árboles de búsqueda. ¿Qué diferencia de comportamiento hay entre este TAD y el TAD de los árboles de búsqueda?

Tablas: Implementaciones basadas en funciones de dispersión

354. Para implementar tablas dispersas que usen cadena de caracteres como claves, podría considerarse la siguiente función de localización:

```
h(c) =def ord(ult(c)) mod 16
```

donde *ord* es la función que hace corresponder a cada carácter el número de orden que le corresponde, según el código ASCII. Calcula el índice que *h* hace corresponder a cada una de las cadenas de caracteres que siguen, y observa qué colisiones se producen:

"Fred", "Alex", "Philip", "Joe", "John", "Hanna", "David",
"Martin", "Violet", "George", "Helen", "Manyu", "Roland"

355. Una *tabla dispersa abierta* es una tabla implementada utilizando una función de localización *h* y un vector que almacena listas de parejas de tipo (Clave, Valor). Las colisiones se resuelven almacenando todas las claves sinónimas *c* tales que $h(c) = i$ en la lista asociada al índice *i* del vector. Plantea una implementación de las tablas como tablas dispersas abiertas, y estudia la

eficiencia que puede lograrse para las operaciones (realizando las generadoras y modificadoras como procedimientos con un parámetro ***es*** de tipo *Tabla[Clave, Valor]*).

356. Sea *t : Tabla[Cadena, Nat]* una tabla que se está utilizando para almacenar valores de tipo *Nat* (que representan edades) asociados a claves de tipo *Cadena* (que representan nombres). Supongamos que *t* está implementada como *tabla dispersa abierta* usando un vector con índices del intervalo *[0..15]* y la función de localización *h* del ejercicio 354. Haz un dibujo que muestre el estado de la estructura representante de *t* después de ejecutar la siguiente serie de llamadas:

```
TablaVacía(t);
inserta(t, "Fred", 25); inserta(t, "Alex", 18); inserta(t, "Philip", 10);
inserta(t, "Joe", 38); inserta(t, "John", 36); inserta(t, "Hanna", 19);
inserta(t, "David", 40); inserta(t, "Martin", 28); inserta(t, "Violet", 20);
inserta(t, "George", 48); inserta(t, "Helen", 90); inserta(t, "Manyu", 24);
inserta(t, "Roland", 14);
```

¿Cuál es en estos momentos la tasa de ocupación de la tabla?

357. Una *tabla dispersa cerrada* es una tabla implementada utilizando una función de localización *h* y un vector que almacena parejas de tipo (Clave, Valor), y tratando las colisiones del siguiente modo: Si el *índice primario* $i_0 = h(c)$ asociado a una clave *c* produce colisión, se prueban otros índices $i_1 = prueba(1, c)$, $i_2 = prueba(2, c)$, … , $i_m = prueba(m, c)$, etc. Esta técnica se llama *relocalización*, y la sucesión de índices i_m que se van probando se llama *sucesión de pruebas*. Cuando la función *prueba* está definida como $prueba(m, c) =_{def} (h(c) + m) \bmod N$ (siendo N la dimensión del vector usado por la implementación), se dice que se tiene *relocalización lineal*.

Sea *t : Tabla[Nat, Nat]* una tabla que usa datos de tipo *Nat* tanto para las claves como para los valores. Supongamos que *t* está implementada como *tabla dispersa cerrada* usando un vector con índices del intervalo *[0..9]* (i.e., $N = 10$), usando relocalización lineal y la función de localización *h* definida como $h(c) =_{def} c \bmod 10$. Estudia cómo va variando el vector representante de la tabla al ejecutar la siguiente serie de llamadas:

```
TablaVacía(t);
Inserta(t, 23, 50); Inserta(t, 33, 60); Inserta(t, 106, 70);
x := consulta(t, 53);
Inserta(t, 206, 80); Inserta(t, 43, 90);
y := consulta(t, 33);
Inserta(t, 53, 100);
borra(t, 33);
Inserta(t, 53, 110); Inserta(t, 79, 1000); Inserta(t, 99, 2000);
z := consulta(t, 53); u := consulta(t, 109);
```

Observa cómo se resuelven las colisiones y cuál es en cada momento la tasa de ocupación de la tabla.

†358. Define un tipo representante adecuado para una implementación del TAD TABLA que use tablas dispersas cerradas. Formula el invariante de la representación y la función de abstracción.

359. Usando la representación del ejercicio anterior, programa el procedimiento que implementa la operación *TablaVacía* y la función que implementa la operación *esVacía*.

360. Usando la representación del ejercicio 358, programa una función auxiliar *busca* que satisfaga la siguiente especificación pre/post:

```
func busca( t : Tabla[Clave, Valor]; c : Clave ) dev i : Nat;
{ P₀ : R(t) ∧ c no es una clave ficticia }
{ Q₀ : ( NOT está(A(t), c) ∧ i = N ) ∨
       ( está(A(t), c) ∧ t.parejas(i).clave == c ) }
ffunc
```

Idea: Para buscar la clave *c*, se comienza con el índice primario $i_0 = h(c)$ y se sigue la sucesión de pruebas hasta encontrar la clave buscada o una posición vacía. Las posiciones borradas no interrumpen la búsqueda. El invariante de la representación garantiza que este algoritmo es correcto.

361. Usando la función auxiliar del ejercicio anterior y la representación del ejercicio 358, programa el procedimiento que implementa la operación *borra* y la función que implementa la operación *consulta*.

362. Usando la representación del ejercicio 358, programa el procedimiento que implementa la operación *Inserta*.

Idea: Si *c* es la clave de inserción, el algoritmo de inserción comienza considerando el índice primario $i_0 = h(c)$, y sigue la serie de pruebas hasta encontrar la clave de inserción o una posición que esté vacía o borrada. Si esta búsqueda tiene éxito, se inserta. Si se ha insertado en una posición borrada, se hace aún otra búsqueda para ver si la clave de inserción aparece más adelante en alguna otra posición de la serie de pruebas. En caso afirmativo, se borra esta posición, que corresponde a otra inserción anterior con la misma clave. Este procedimiento es correcto si el invariante de la representación se cumple inicialmente, y su ejecución preserva el invariante de la representación.

363. En la práctica, la relocalización no se implementa mediante llamadas a una función *prueba*, como hemos supuesto en los ejercicios 360 y 362, sino que el cálculo de los índices de la sucesión de pruebas se incorpora dentro del código de la función *busca* y el procedimiento *Inserta*, del modo adecuado a la técnica de relocalización que se esté utilizando. Modifica los algoritmos obtenidos en los ejercicios 360 y 362 para obtener versiones de *busca* e *Inserta* que incorporen relocalización lineal sin llamadas a una función *prueba*.

364. Como en el ejercicio 357, consideramos *t : Tabla[Nat, Nat]* implementada como tabla cerrada, pero ahora suponemos que $N = 100$ y que la función de localización está definida como $h(c) =_{def} c \bmod 100$. Estudia el efecto de la sucesión de llamadas.

```
TablaVacía(t);
Inserta(t, 10, 1010); Inserta(t, 210, 2110);
Inserta(t, 110, 1110); Inserta(t, 310, 3110);
x := consulta(t, 10); y := consulta(t, 310); z := consulta(t, 56);
Inserta(t, 109, 1109);
u := consulta(t, 9);
Inserta(t, 9, 9999); Inserta(t, 410, 4111); Inserta(t, 12, 1212);
Inserta(t, 99, 1099); Inserta(t, 1999, 1999); Inserta(t, 0, 0);
```

(a) utilizando *relocalización lineal*.
(b) utilizando *relocalización cuadrática*.

365. Diseña un procedimiento que "limpie" una tabla cerrada, de tal manera que las posiciones borradas se eliminen y las restantes informaciones se reubiquen en la tabla. Todas las in-

formaciones correspondientes a claves sinónimas deberán quedar situadas en posiciones consecutivas de la sucesión de pruebas, a partir de la posición primaria que corresponda. (Este procedimiento podría ser exportado por un módulo de implementación del TAD TABLA, junto con las funciones y procedimientos que realizan las operaciones del TAD).

Tablas: Aplicaciones

366. Se llaman *vectores dispersos* a los vectores implementados por medio de tablas dispersas. Esta técnica es recomendable cuando el conjunto total de índices posibles es muy grande, y la gran mayoría de los índices tiene asociado un *valor por defecto* (por ejemplo, cero). Diseña funciones que ejecuten el *producto escalar* y la suma de vectores dispersos de números reales con índices enteros, suponiendo como valor por defecto el cero. Para ello deberás suponer disponible un módulo que implemente los vectores dispersos por medio de tablas (con *NoDef = 0*). Las operaciones exportadas por este módulo serán las del TAD VECTOR, y la representación interna de los vectores no será visible para los usuarios del módulo.

367. Desarrolla una implementación del TAD POLI de los polinomios con coeficientes enteros en una indeterminada (cfr. ejercicio 151), representando los polinomios como vectores dispersos de coeficientes, indexados por exponentes y con el valor por defecto cero. El planteamiento de la implementación debe ser modular, importando los vectores dispersos de un módulo que los implemente.

368. Resuelve de nuevo el *problema de las concordancias* (ver ejercicio 307), utilizando en lugar de un árbol de búsqueda una tabla ordenada del tipo que hemos considerado en el ejercicio 353. Analiza el tiempo de ejecución del algoritmo que obtengas.

369. Una variante del problema de las concordancias consiste en pedir como resultado un listado de las k palabras que han aparecido más frecuentemente en el texto, ordenadas en orden decreciente de frecuencias de aparición. Diseña un algoritmo que resuelva este problema, combinando las ideas de los ejercicios 368 y 338

370. Desarrolla implementaciones modulares de los TADs siguientes, usando tablas con claves y valores de tipo conveniente para la construcción del tipo representante. Analiza en cada caso el tiempo de ejecución de las operaciones y el espacio ocupado por la representación, haciendo suposiciones adecuadas acerca del tipo de tabla utilizado y las características de su implementación.

(a) El TAD BOSTEZOS del ejercicio 231.
(b) El TAD CONSULTORIO del ejercicio 232.
(c) El TAD MERCADO del ejercicio 233.
(d) El TAD BANCO del ejercicio 234.
(e) El TAD BIBLIOTECA del ejercicio 236.

CAPÍTULO 7
GRAFOS

7.1 Modelo matemático y especificación

Un grafo está compuesto por un conjunto de *vértices* (o *nodos*) y un conjunto de *aristas* (*arcos* en los grafos dirigidos) que definen conexiones entre los vértices. A partir de esta idea básica, se definen distintos tipos de grafos:

— Dirigidos o no dirigidos. Según que las aristas estén o no orientadas.

— Valorados y no valorados. Según que las aristas tengan o no peso.

— Simples o multigrafos. Según que se permita una o más aristas entre dos vértices.

Los grafos son objeto de estudio en la asignatura *Matemática discreta*. Desde el punto de vista matemático, un grafo se puede modelar como una pareja formada por un conjunto de vértices y un conjunto de aristas:

$G = (V, A) \qquad A \subseteq V \times V$

(Según si el grafo está dirigido o no, A es una relación o un conjunto de conjuntos de dos elementos).

Otro posible modelo para los grafos consiste en representarlos como un aplicación:

$g : V \times V \to Bool$ para los grafos no valorados

$g : V \times V \to W$ para los grafos valorados

De Matemática discreta podemos recordar conceptos como:

— Adyacencia, incidencia. En un grafo no dirigido, un vértice es adyacente a otro si existe una arista que los una; en un grafo dirigido, un vértice u es incidente a otro v si existe una arista desde u a v.

— Grado –de entrada y de salida– de un vértice.

— Los árboles son un caso particular de los grafos

— Resultados sobre la relación entre el número de vértices NV y el de aristas NA.

  — Grafo completo:

    — dirigido: $NA = NV \cdot (NV - 1) = NV^2 - NV$

    — no dirigido: $NA = (NV^2 - NV) / 2$

  — Arbol: $NA = NV - 1$

  — Bosque con k árboles: $NA = NV - k$

— Un *camino* es una sucesión de vértices $v_0, v_1, \ldots, v_n$ tal que para $1 \le i \le n$ (v_{i-1}, v_i) es un arco (o una arista si el grafo no es dirigido). La longitud del camino es n (el número de arcos o aristas).

  — Camino abierto: $v_n \neq v_0$

  - Circuito, camino cerrado o circular: $v_n = v_0$
  - Camino simple: no repite vértices (excepto quizá $v_0 = v_n$)
  - Camino elemental: no repite arcos
  - Ciclo: camino circular simple
- Grafo conexo: grafo no dirigido tal que existe un camino entre cada 2 vértices
- Grafo fuertemente conexo: grafo dirigido tal que existe un camino entre cada 2 vértices.

Especificación algebraica

Los grafos se pueden especificar partiendo la idea de generarlos a partir del grafo vacío, usando operaciones que añadan vértices y aristas, e incluyendo otras operaciones para decidir si dos vértices dados son adyacentes, calcular los vértices vecinos de un vértice dado, etc. Los detalles de la especificación cambian según la clase de grafos que se quiera especificar.

En el caso más general de los grafos valorados, necesitamos especificar los grafos con dos parámetros: el tipo de los vértices y el tipo de los valores de los arcos.

A los vértices les exigimos que pertenezcan a la clase de los tipos discretos, una clase que ya presentamos al hablar del TAD VECTOR, y que sirve como generalización de los tipos que se pueden utilizar como índices de los vectores.

```
clase DIS
    hereda
      ORD
    operaciones
      card: → Nat
      ord: Elem → Nat
      elem: Nat – → Elem
      prim, ult: → Elem
      suc, pred: Elem – → Elem
    axiomas
      ∀ x, y : Elem : ∀ i : Nat :
      card ≥ 1
      1 ≤ ord(x) ≤ card
      def elem(i) si 1 ≤ i ≤ card
      ord(elem(i)) =d i
      elem(ord(x)) = x
      prim = elem(1)
      ult = elem(card)
      def suc(x) si x /= ult
      suc(x)                    =d elem(ord(x) – 1)
      x == y             = ord(x) == ord(y)
      x ≤ y              = ord(x) ≤ ord(y)
fclase
```

La razón de exigir esta condición a los vértices de los grafos está en que algunos algoritmos sobre grafos devuelven vectores –bidimensionales– indexados por vértices. Para que tuviese más sentido esta restricción, se deberían incluir dichas operaciones en la especificación del TAD.

En cuanto a las etiquetas de los arcos, les exigimos que pertenezcan a un tipo ordenado, que exporte una operación de suma, y que incluya el valor *NoDef*, que renombramos a ∞, y el valor 0. La razón de estas restricciones está otra vez en los algoritmos que queremos implementar sobre los grafos, donde necesitaremos sumar valores, comparar valores, y tener disponible un valor mínimo y un valor máximo. En la especificación de los grafos que presentamos a continuación sólo aparece el valor máximo, ∞, para conseguir que la operación que consulta sobre el coste de una arista sea una operación total, que de ∞ como coste de una arista que no está en el grafo.

```
clase VAL-ORD
    hereda
      EQ-ND, ORD
    renombra
      Elem a Valor
      Nodef a ∞
    operaciones
      0: → Valor
      ( + ): (Valor, Valor) → Valor
    axiomas
      ∀ x, y, z : Valor :
      0 ≤ x
      x ≤ ∞
      x + y              = y + x
      (x + y) + z  = x + (y + z)
      x + 0              = 0
      x + ∞              = ∞
fclase
```

Incluimos operaciones para: construir un grado vacío, *Vacío*; añadir una arista, indicando los vértices que une, y su coste, *PonArista*; quitar la arista que une a dos vértices dados, *quitaArista*; obtener el coste de la arista que une a dos vértices dados, *costeArista*; consultar si existe una arista que une a dos vértices dados, *hayArista?*; obtener los vértices sucesores de uno dado, *sucesores*; y obtener los vértices predecesores de uno dado, *predecesores*. No prohibimos que haya *bucles*, con origen y destino en el mismo vértice. Nótese que no incluimos ninguna operación para insertar un vértice, por lo que no puede existir ningún grafo –componente conexa– con un solo vértice, a no ser que tenga un bucle. Especificamos *PonArista* de forma que no pueda haber dos aristas entre los mismos vértices; en la especificación de los multigrafos habría que suprimir esta restricción.

Finalmente, la especificación de las grafos dirigidos valorados:

```
tad WDGRAFO[V :: DIS, W :: VAL-ORD]
   renombra
      V.Elem a Vértice
```

```
usa
  BOOL, SEC[PAREJA[V, W]]
usa privadamente
  CJTO[PAREJA[V, W]]
tipo
  DGrafo[Vértice, Valor]
operaciones
  Vacío: → DGrafo[Vértice, Valor]                                  /* gen */
  PonArista: (DGrafo[Vértice, Valor], Vértice, Vértice, Valor)
            → DGrafo[Vértice, Valor]                               /* gen */
  quitaArista: (DGrafo[Vértice, Valor], Vértice, Vértice)
            → DGrafo[Vértice, Valor]                               /* mod */
  costeArista: (DGrafo[Vértice, Valor], Vértice, Vértice) → Valor /* obs */
  hayArista?: (DGrafo[Vértice, Valor], Vértice, Vértice) → Bool   /* obs */
  sucesores: (DGrafo[Vértice, Valor], Vértice)
            → Sec[Pareja[Vértice, Valor]]                          /* obs */
  predecesores: (DGrafo[Vértice, Valor], Vértice)
            → Sec[Pareja[Vértice, Valor]]                          /* obs */
operaciones privadas
  cjtoSuc: (DGrafo[Vértice, Valor], Vértice)
            → Cjto[Pareja[Vértice, Valor]]                         /* obs */
  cjtoPred: (DGrafo[Vértice, Valor], Vértice)
            → Cjto[Pareja[Vértice, Valor]]                         /* obs */
  enumera: Cjto[Pareja[Vértice, Valor]]
            → Sec[Pareja[Vértice, Valor]]                          /* obs */
  insertaOrd: (Pareja[Vértice, Valor], Sec[Pareja[Vértice, Valor]]
            → Sec[Pareja[Vértice, Valor]]                          /* obs */
ecuaciones
  ∀ g : DGrafo[Vértice, Valor] : ∀ u, u1, u2, v, v1, v2 : Vértice :
  ∀ c, c1, c2 : Valor : ∀ ps : Sec[Pareja[Vértice, Valor]] :
  ∀ xs : Cjto[Pareja[Vértice, Valor]] :

  PonArista(PonArista(g, u1, v1, c1), u2, v2, c2 )  = PonArista(g, u2, v2, c2
)
       si u1 == u2 AND v1 == v2
  PonArista(PonArista(g, u1, v1, c1), u2, v2, c2 )
    = PonArista(PonArista(g, u2, v2, c2), u1, v1, c1 )
       si NOT (u1 == u2 AND v1 == v2 )
  quitaArista(Vacío, u, v)                          = Vacío
  quitaArista(PonArista(g, u1, v1, c), u2, v2)      = quitaArista(g, u2, v2)
       si u1 == u2 AND v1 == v2
  quitaArista(PonArista(g, u1, v1, c), u2, v2)
    = PonArista(quitaArista(g, u2, v2), u1, v1, c)
       si NOT (u1 == u2 AND v1 == v2 )
  costeArista(Vacío, u, v)                          = ∞
```

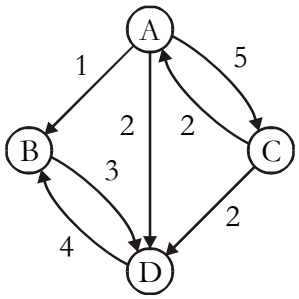
```
      costeArista(PonArista(g, u₁, v₁, c), u₂, v₂)      = c
           si u₁ == u₂ AND v₁ == v₂
      costeArista(PonArista(g, u₁, v₁, c), u₂, v₂)      = costeArista(g, u₂, v₂)
           si NOT (u₁ == u₂ AND v₁ == v₂ )
      hayArista?(g, u, v)                               = costeArista(g, u, v) /=
∞
      sucesores(g, u)                                   = enumera(cjtoSuc(g, u))
      predecesores(g, v)                                = enumera(cjtoPred(g, v))

   % enumera devuelve una secuencia con parte izquierda vacía
      enumera(CJTO.Vacío)                               = SEC.Crea
      enumera(Pon(Par(v, c)), xs))
        = insertaOrd(Par(v, c), enumera(quita(Par(v, c), xs)))

   % insertaOrd devuelve una secuencia con parte izquierda vacía
      insertaOrd(Par(v, c), ps)
        = reinicia(inserta(Par(v, c), ps))
           si fin?(ps)
      insertaOrd(Par(v, c), ps)
        = reinicia(inserta(Par(v, c), ps))
           si NOT fin?(ps) AND actual(ps) = Par(v₁, c₁) AND v < v₁
      insertaOrd(Par(v, c), ps)
        = insertaOrd(Par(v, c), avanza(ps))
           si NOT fin?(ps) AND actual(ps) = Par(v₁, c₁) AND v ≥ v₁
      cjtoSuc(Vacío, u)                                 = CJTO.Vacío
      cjtoSuc(PonArista(g, u₁, v, c), u)
        = Pon(Par(v, c), cjtoSuc(quitaArista(g, u, v), u))
           si u == u₁
      cjtoSuc(PonArista(g, u₁, v, c), u)                                =
cjtoSuc(g, u)
           si u /= u₁
      cjtoPred(Vacío, v)                                = CJTO.Vacío
      cjtoPred(PonArista(g, u, v₁, c), v)
        = Pon(Par(u, c), cjtoPred(quitaArista(g, u, v), v))
           si v == v₁
      cjtoPred(PonArista(g, u, v₁, c), v)               = cjtoPred(g, v)
           si v /= v₁
ftad
```

7.2 Técnicas de implementación

Matriz de adyacencia

El grafo se representa como una matriz bidimensional indexada por vértices. En los grafos valorados, en la posición *(i, j)* de la matriz se almacena el peso de la arista que va del vértice *i* al vértice *j*, ∞ si no existe tal arista.

Por ejemplo, el grafo:

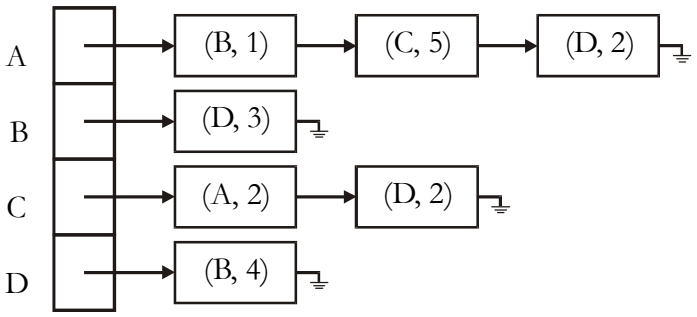


vendría representado por la matriz:

	A	B	C	D
A	∞	1	5	2
B	∞	∞	∞	3
C	2	∞	∞	2
D	∞	4	∞	∞

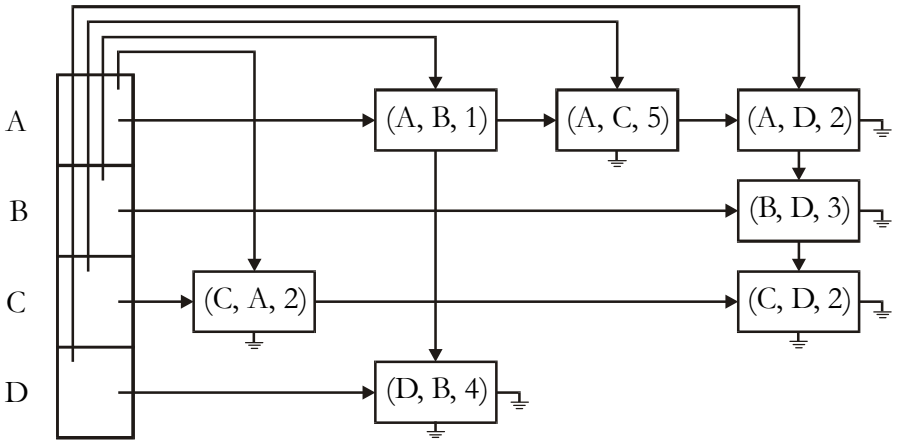
Vector de listas/secuencias de adyacencia

Se representa como un vector indexado por vértices, donde cada posición del vector contiene la lista de aristas que parten de ese vértice –la lista de sucesores–, representadas como el vértice de destino y la etiqueta de la arista. En el grafo del ejemplo anterior:



Vector de multilistas de adyacencia

Se representa el grafo como un vector indexado por vértices, donde cada posición del vector contiene dos listas: una con las aristas que inciden en ese vértice –lista de predecesores–, y otra con las aristas que parten de él –lista de sucesores–. La representación de cada aristas se compone de el vértice de partida, el de destino y el peso. En el grafo del ejemplo anterior:



En este caso no se puede realizar una implementación modular que importe las listas de otro módulo; hay que construir directamente un tipo representante usando registros y punteros.

Variantes

- Grafos no dirigidos.
  - La matriz de adyacencia es triangular y admite una representación optimizada.
  - En las listas de adyacencia puede ahorrarse espacio suponiendo un orden entre vértices y poniendo v en la lista de u sólo si $u < v$ –con lo que nos ahorramos representar dos veces la misma arista–.
- Grafos no valorados.
  - La matriz de adyacencia puede ser de booleanos
  - Las listas de adyacencia se reducen a listas de vértices.
- Multigrafos.
  - Las matrices de adyacencia no son una representación válida.
  - Las listas de adyacencia deben ser listas de ternas *(identificador, vértice, valor*), donde el *identificador* identifica unívocamente al arco.

Eficiencia de las operaciones

Dados los parámetros de tamaño:

- NV: número de vértices
- NA: número de aristas
- GE: máximo grado de entrada

— GS: máximo grado de salida

Operación	Matriz	Lista	Multilista
Vacío	O(NV^2) (1)	O(NV)	O(NV)
PonArista	O(1)	O(GS) (2)	O(GS+GE) (4)
quitaArista	O(1)	O(GS) (2)	O(GS+GE) (4)
costeArista	O(1)	O(GS) (2)	O(GS) (2)
hayArista	O(1)	O(GS) (2)	O(GS) (2)
sucesores	O(NV) (1)	O(GS) (2) (C)	O(GS) (2) (C)
predecesores	O(NV) (1)	O(NV+NA) (3)	O(GE) (5) (C)

(1) Recorrido de todos los vértices

(2) Recorrido de la lista de sucesores de un vértice. Nótese que $GS \leq NV-1$ y que por lo tanto $O(GS) \subseteq O(NV)$.

(3) Recorrido de todos los vértices, y para cada uno, recorrido de su lista de sucesores. El tiempo es $O(NV+NA)$ porque cada arista $u \rightarrow v$ se atraviesa una sola vez –en el recorrido de los sucesores de u–.

(4) Para poner o quitar la arista $u \rightarrow v$ hay que recorrer los sucesores de u y los predecesores de v –para así determinar la modificación oportuna de los enlaces en la multilista de adyacencia–. Nótese que $GE, GS \leq NV-1$ y que por lo tanto $O(GE+GS) \subseteq O(NV)$.

(5) Recorrido de los predecesores de un vértice.

Los tiempos marcados como *(C)* se reducen a $O(1)$ si no se hace una copia de la lista de sucesores/predecesores devuelta como resultado.

Implementación de las matrices de adyacencia

Tipo representante

```
módulo impl WDGRAFO[VERTICE, ARISTA]
  importa
    VAL-ORD, DIS, VECTOR[ELEM, ELEM, ELEM]
  privado
    tipo
      Vértice = DIS.Elem;
      Valor = VAL-ORD.Valor;
      DGrafo[Vértice, Valor] = VECTOR.Vector[Vértice, Vértice, Valor]
…
fmódulo
```

Estamos suponiendo implementado un módulo genérico para los vectores bidimensionales. El problema que se plantearía sería la utilización de este módulo con índices que fuesen un subrango

de un tipo predefinido, en cuyo caso estaríamos obligados a implementar también un módulo para representar a ese subrango.

El invariante de la representación es *cierto* pues cualquier matriz del tipo adecuado es un representante válido. En cuanto a la función de abstracción, se deja como ejercicio.

Algunas operaciones

Construir un grafo vacío:

```
proc Vacío( s g : DGrafo[Vértice, Valor] );
var
  u, v : Vértice;
inicio
  VECTOR.Crea(g);
  u := DIS.prim( );
  it DIS.menorIgual( u, DIS.ult( ) ) →
    v := DIS.prim( );
    it DIS.menorIgual( v, DIS.ult( ) ) →
      VECTOR.Asigna(g, u, v, VAL-ORD.infi( ));      % g(u,v) := ∞
      v := DIS.suc(v);
    fit;
    u := DIS.suc(u)
  fit
fproc
```

Hay un error porque el sucesor del último no está definido, y eso es lo que intentamos obtener en la última iteración. Se resolvería utilizando un bucle *repeat .. until* o incluyendo la obtención del sucesor dentro de una condición que protegiera la situación conflictiva.

Quitar una arista:

```
proc quitaArista( es g : DGrafo[Vértice, Valor]; e u, v : Vértice )
inicio
  VECTOR.Asigna(g, u, v, VAL-ORD.infi( ));   % g(u, v) := ∞
fproc
```

Obtener los predecesores de un vértice:

```
func predecesores( g : DGrafo[Vértice, Valor]; v : Vértice )
      dev ps : Sec[Pareja[Vértice, Valor]]
var
  u : Vértice;
  w : Valor;
inicio
  SEC.Crea(ps);
```

```
  u := DIS.prim( );
  it DIS.menorIgual( u, DIS.ult( ) ) →
    w := VECTOR.valor(g, u, v);        % w := g(u, v)
    si VAL-ORD.igual(w, VAL-ORD.infi( ))
      entonces
        seguir
      sino
        SEC.inserta(ps, PAREJA.Par(u, w))
    fsi;
    u := DIS.suc(u)
  fit;
  SEC.reinicia(ps);
  dev ps
ffunc;
```

Implementación de las listas de adyacencia

Tipo representante

```
módulo impl WDGRAFO[VERTICE, ARISTA]
  importa
    VAL-ORD, DIS, VECTOR[ELEM, ELEM], SEC[ELEM], PAREJA[ELEM,ELEM]
  privado
    tipo
      Vértice = DIS.Elem;
      Valor = VAL-ORD.Valor;
      DGrafo[Vértice, Valor] = VECTOR.Vector[Vértice, Sec[Pareja[Vértice,
Valor]]]
…
fmódulo
```

Invariante de la representación

Dado *g : DGrafo[Vértice, Valor]*

```
      R(g)
⇔def
      ∀ u : Vértice : las parejas de g(u) aparecen ordenadas en orden
creciente
                      con respecto a la componente Vértice
```

Mantenemos las secuencias de sucesores ordenadas, para mejorar la eficiencia de las operaciones. Para escribir formalmente este aserto es necesario definir un cuantificador sobre el dominio de los tipos de la clase DIS.

Algunas operaciones

Inserción de una arista:

```
proc PonArista( es g : DGrafo[Vértice, Valor]; e u, v : Vértice; e w : Valor
);
var
  xs : Sec[Pareja[Vértice, Valor]];
  encontrado : Bool;
inicio
  xs := VECTOR.valor(g, u);           % xs := g(u);
{ piz(xs) = [] }
  busca( xs, v, encontrado );   % proc. auxiliar de búsqueda en secuencia
  si encontrado
    entonces
      SEC.borra(xs)
    sino
      seguir
  fsi;
  SEC.inserta(xs, PAREJA.Par(v, w));
  SEC.reinicia(xs);
  VECTOR.asigna(g, u, xs)       % g(u) := xs
fproc
```

El procedimiento auxiliar *busca* recorre la secuencia, comparando el vértice buscado con el primer componente de cada pareja de la secuencia. Se para cuando encuentra el vértice buscado, o cuando se encuentra con uno mayor.

Todas las secuencias del vector están reiniciadas entre operación y operación. Podríamos añadirlo al invariante de la representación.

En cuanto a la gestión de la memoria dinámica, estamos suponiendo que *VECTOR.valor* no hace copia del valor devuelto. Si las secuencias están implementadas como un puntero a un nodo cabecera, no sería necesaria la reinserción.

Obtención del coste de una arista:

```
func costeArista( g : DGrafo[Vértice, Valor]; u, v : Vértice ) dev w :
Valor;
var
  xs : Sec[Pareja[Vértice, Valor]];
  encontrado : Bool;
inicio
  xs := VECTOR.valor(g, u);           % xs := g(u);
  busca( xs, v, encontrado );   % proc. auxiliar de búsqueda en secuencia
  si encontrado
    entonces
      w := PAREJA.sg( SEC.actual(xs) )
    sino
      w := VAL-ORD.infi( )
```

```
  fsi;
  dev w
ffunc
```

Estamos suponiendo que las secuencias se implementan con un nodo cabecera, un registro representado con memoria estática, y que por lo tanto los cambios en *xs* no afectan a *g(u)*. Si lo que tuviésemos fuese un puntero al nodo cabecera, entonces deberíamos reiniciar *xs*.

La función que obtiene los predecesores de un vértice

```
func predecesores( g : DGrafo[Vértice, Valor]; v : Vértice )
      dev ps : Sec[Pareja[Vértice, Valor]]
var
  u : Vértice;
  w : Valor;
  xs : Sec[Pareja[Vértice, Valor]];
inicio
  SEC.Crea(ps);
  u := DIS.prim( );
  it DIS.menorIgual( u, DIS.ult( ) ) →
    xs := VECTOR.valor(g, u);          % xs := g(u)
    busca(xs, v, encontrado);
    si encontrado
      entonces
        w := PAREJA.sg( SEC.actual(xs) );
        SEC.inserta(ps, PAREJA.Par(u, w))
      sino
        seguir
    fsi;
    u := DIS.suc(u)
  fit;
  SEC.reinicia(ps);
  dev ps
ffunc;
```

7.3 Recorridos de grafos

En analogía con los árboles, los grafos admiten recorridos en profundidad y en anchura, tanto si son dirigidos como si no. En el caso no dirigido, hay que considerar cada arista como una pareja de aristas orientadas. En muchos casos, los recorridos no dependen de los valores de los arcos, por lo que en este tema nos limitaremos a grafos no valorados. El TAD grafo no impone un orden determinado a los sucesores (o predecesores) de un vértice (aunque esto no es necesariamente cierto, pues los grafos se pueden especificar con un orden entre los sucesores/predecesores de cada vértice). Como consecuencia, no es posible especificar de manera unívoca una lista de vértices como resultado de un recorrido. Esta indeterminación no causa inconvenientes prácticos. Es suficiente que los algoritmos de recorrido devuelvan una de los recorridos en profundidad y anchura.

Los grafos dirigidos acíclicos admiten un tercer tipo de recorrido, denominado *recorrido de ordenación topológica*.

Vamos a hacer implementaciones modulares de las operaciones, sin acceder a la representación interna de los grafos.

7.3.1 Recorrido en profundidad

Se puede considerar como una generalización del recorrido en preorden de un árbol. La idea es:

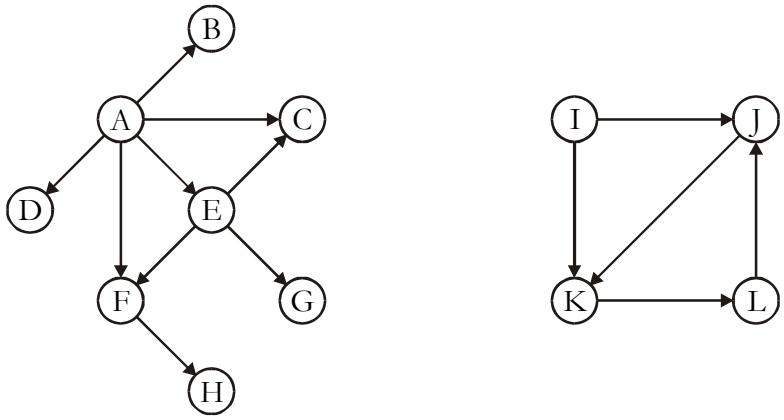
— Visitar el vértice inicial

— Si es posible, avanzar a un sucesor aún no visitado

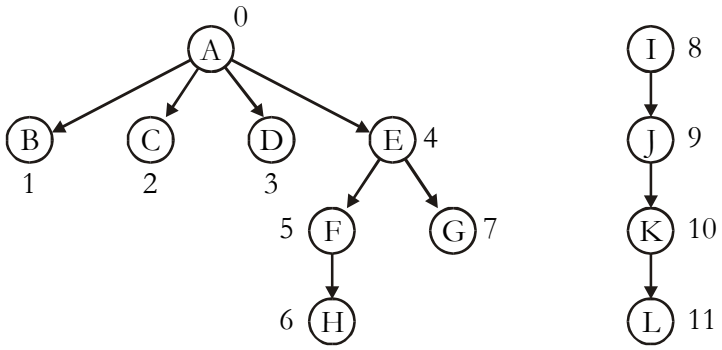
— En otro caso, retroceder al vértice visitado anteriormente, e intentar continuar el recorrido desde éste.

Una vez visitados todos los descendientes del vértice inicial, se quedan vértices no visitados hay que iniciar un recorrido de otra componente del grafo.

Por ejemplo, el recorrido del grafo:



Representado como un bosque:



Posibles aplicaciones del recorrido en profundidad:

— Determinar si un grafo es conexo (sirve en el caso de grafos no dirigidos)

— Determinar si un grafo es acíclico (sirve en el caso de grafos no dirigidos)

— Buscar un vértice con una propiedad determinada.

El algoritmo recursivo de recorrido en profundidad

— Devuelve una secuencia en la cual aparece cada vértice del grafo una sola vez.

— Usa un conjunto de vértices para llevar cuenta de los vértices visitados. Las operaciones de CJTO necesarias son $O(1)$, lo que se puede conseguir con una implementación basada en tablas dispersas (un vector de booleanos).

— Usa un procedimiento auxiliar privado encargado del recorrido de una sola componente.

```
func recorreProf( g : DGrafo[Vértice] ) dev xs : Sec[Vértice];
{ P₀ : cierto }
var
  v : Vértice;
  vs : Cjto[Vértice];
inicio
  CJTO.Vacío(vs);
  SEC.Crea(xs);
  v := DIS.prim( );
  it DIS.menorIgual( v, DIS.ult( ) ) →
    si CJTO.pertenece(v, vs)
      entonces
        seguir
      sino
        recorreProfComp(g, v, vs, xs)
    fsi;
    v := DIS.suc(v)
  fit
{ Q₀ : cont(xs) representa un recorrido en profundidad de g }
  dev xs
ffunc
```

El procedimiento auxiliar que recorre en profundidad una componente:

```
proc recorreProfComp( e g : DGrafo[Vértice]; e v : Vértice;
                      es vs : Cjto[Vértice]; es xs : Sec[Vértice]);
{ P₀ : vs = VS ∧ xs = XS ∧ NOT CJTO.pertenece(v, vs) ∧ SEC.fin?(xs) = cierto
∧
      vs contiene los vértices de xs
}
var
  ss : Sec[Vértice];
  u : Vértice;
inicio
  CJTO.Pon(v, vs);
  SEC.inserta(xs, v);
```

```
    ss := DGRAFO.sucesores(g, v);
    it NOT SEC.fin?(ss) →
      u := SEC.actual(ss);
      SEC.avanza(ss);
      si CJTO.pertenece(u, vs)
        entonces
          seguir
        sino
          recorreProfComp(g, u, vs, xs)
      fsi
    fit
  { Q₀ : SEC.fin?(xs) = cierto ∧
         cont(xs) es cont(XS) prolongado con el resultado de un recorrido
         en profundidad de los vértices de g accesibles desde v que no
         estaban en VS ∧ vs contiene los vértices de xs                    }
  fproc
```

La complejidad de la operación depende de la representación elegida para los grafos:

— Si se usan listas o multilistas de adyacencia, el tiempo es $O(NV+NA)$, ya que cada vértice se visita una sola vez, pero se exploran todas las aristas que salen de él.

— Si se usan matrices de adyacencia el tiempo es $O(NV^2)$, ya que cada vértice se visita una sola vez, pero el cálculo de sus sucesores requiere tiempo $O(NV)$.

7.3.2 Recorrido en anchura

El recorrido en anchura, o por niveles, generaliza el recorrido de árboles con igual denominación. La idea es:

— Visitar el vértice inicial.

— Si el último vértice visitado tiene sucesores aún no visitados, realizar sucesivamente un recorrido desde cada uno de estos. (Esta descripción informal no describe correctamente el recorrido en anchura).

— En otro caso, continuar con un recorrido iniciado en cualquier vértice no visitado aún.

El recorrido en anchura encuentra caminos de longitud mínima. Por este motivo, es típico aplicarlo a problemas de planificación, donde:

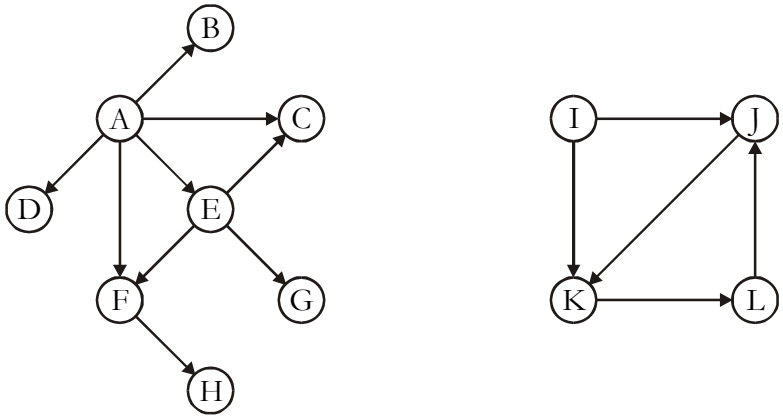
— los vértices representan estados

— los arcos representan transiciones entre estados

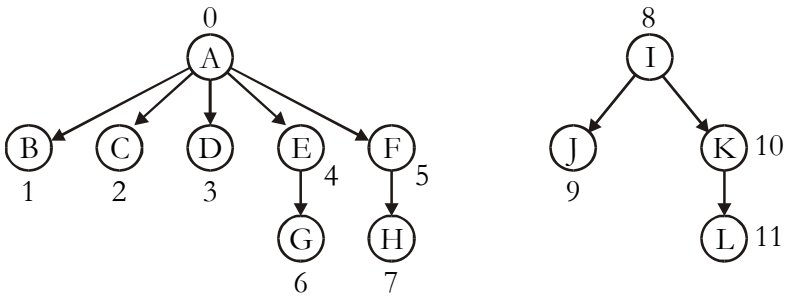
— los caminos representan cambios de estado causados por sucesiones de transiciones

Entre las aplicaciones de este estilo se incluyen juegos, laberinto, etc.

Por ejemplo, para el mismo grafo del ejemplo anterior:



representado como un bosque, resultado del recorrido por niveles



La implementación es similar a la del recorrido en profundidad, con ayuda de un procedimiento auxiliar que recorre una componente del grafo. La diferencia está en que en lugar de utilizar un procedimiento recursivo, lo hacemos iterativo con ayuda de una cola –a la manera del recorrido por niveles de un árbol–. si cambiásemos esa cola por una pila tendríamos la versión iterativa del recorrido en profundidad.

```
func recorreNiv( g : DGrafo[Vértice] ) dev xs : Sec[Vértice];
{ P₀ : cierto }
var
  v : Vértice;
  vs : Cjto[Vértice];
inicio
  CJTO.Vacío(vs);
  SEC.Crea(xs);
  v := DIS.prim( );
  it DIS.menorIgual( v, DIS.ult( ) ) →
    si CJTO.pertenece(v, vs)
      entonces
        seguir
```

```
      sino
        recorreNivComp(g, v, vs, xs)
    fsi;
    v := DIS.suc(v)
  fit
{ Q₀ : cont(xs) representa un recorrido por niveles de g }
  dev xs
ffunc
```

El procedimiento auxiliar que recorre por niveles una componente:

```
proc recorreNivComp( e g : DGrafo[Vértice]; e v : Vértice;
                     es vs : Cjto[Vértice]; es xs : Sec[Vértice]);
{ P₀ : vs = VS ∧ xs = XS ∧ NOT CJTO.pertenece(v, vs) ∧ SEC.fin?(xs) = cierto
∧
      vs contiene los vértices de xs
}
var
  us : Cola[Vértice];
  ss : Sec[Vértice];
  u, w : Vértice;
inicio
  CJTO.Pon(v, vs);
  COLA.ColaVacía(us);
  COLA.Añadir(v, us);
  it NOT COLA.esVacía(us) →
    u := COLA.primero(us);
    COLA.avanzar(us);
    SEC.inserta(xs, u);
    ss := DGRAFO.sucesores(g, u);
    it NOT SEC.fin?(ss) →
      w := SEC.actual(ss);
      SEC.avanza(ss);
      si CJTO.pertenece(w, vs)
        entonces
          seguir
        sino
          CJTO.Pon(w, vs);
          COLA.Añadir(w, us)
      fsi
    fit
  fit
{ Q₀ : SEC.fin?(xs) = cierto ∧
       cont(xs) es cont(XS) prolongado con el resultado de un recorrido
       por niveles de los vértices de g accesibles desde v que no
       estaban en VS ∧ vs contiene los vértices de xs                    }
```

```
fproc
```

El análisis de la complejidad es el mismo que hemos hecho para el recorrido en profundidad.

7.3.3 Recorrido de ordenación topológica

Este tipo de recorridos sólo es aplicable a grafos dirigidos acíclicos.

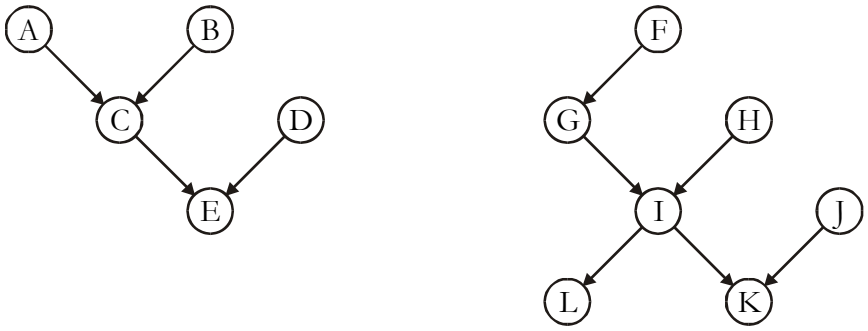
Dado un grafo dirigido acíclico *G*, la relación entre vértices definida como

$$u \prec_G v \Leftrightarrow_{\text{def}} \text{existe un camino de } u \text{ a } v \text{ en } G \text{ (i.e., } u \text{ es antepasado de } v)$$

es un orden parcial. Se llama *recorrido de ordenación topológica* de *G* a cualquier recorrido de *G* que visite cada vértice *v* solamente después de haber visitado todos los vértices de *u* tales que $u \prec_G v$. En general, son posibles varios recorridos de ordenación topológica para un mismo grafo *G*.

Este tipo de recorrido sirve para tener en cuenta relaciones de precedencia entre los vértices del grafo (por ejemplo, los prerrequisitos de un plan de estudios).

Por ejemplo, algunos recorridos en ordenación topológica del grafo:



xs : [A, B, C, D, E, F, G, H, I, J, K, L]

xs : [A, B, D, F, H, J, C, E, G, I, L, K] …

La idea del algoritmo es reiterar la elección de un vértice aún no visitado y tal que todos sus predecesores hayan sido ya visitados. El problema es que el algoritmo resultante de seguir directamente esta idea es poco eficiente.

Una forma de lograr un algoritmo más eficiente es:

- Mantener un vector *P* indexado por vértices, tal que *P(v)* es el número de predecesores de *v* aún no visitados.
- Mantener los vértices *v* tal que *P(v) = 0* en un conjunto *M*.
- Organizar un bucle que en cada vuelta:
  - añade un vértice de *M* al recorrido
  - actualiza *P* y *M*

Para la implementación de esta idea necesitamos suponer al módulo que implementa a los conjuntos equipado con una operación que permite extraer un elemento cualquiera del conjunto:

```
func elige( m : Cjto[Vértice] ) dev u : Vértice;
{ P₀ : NOT esVacío(m) }
{ Q₀ : pertenece(u, m) }
ffunc
proc ordenaTop( e g : DGrafo[Vértice]; s xs : Sec[Vértice] );
{ P₀ : g es acíclico }
var
  u, v : Vértice;
  p : Vector[Vértice, Nat];
  m : Cjto[Vértice];
  ss : Sec[Vértice];
inicio
% inicialización de p y m, primera fase O(NV)
  CJTO.Vacío(m);
  VECTOR.Crea(p);
  v := DIS.prim( );
  it DIS.menorIgual( v, DIS.ult( ) ) →
    VECTOR.Asigna(p, v, 0);           % p(v) := 0
    CJTO.Pon(v, m);
    v := DIS.suc(v)
  fit;
% inicialización de p y m, segunda fase, O(NV+NA) o O(NV²)
  u := DIS.prim( );
  it DIS.menorIgual( u, DIS.ult( ) ) →
    ss := DGRAFO.sucesores(g, u);
    it NOT SEC.fin?(ss) →
      v := SEC.actual(ss);
      SEC.avanza(ss);
      VECTOR.Asigna(p, v, VECTOR.valor(p, v) + 1); % p(v) := p(v) + 1
      CJTO.quita(v, m)
    fit
  fit;
  SEC.Crea(xs);
% bucle principal, O(NV+NA) o O(NV²)
  it NOT CJTO.esVacío(m) →
    u := CJTO.elige(m);
    CJTO.quita(u, m);
    SEC.inserta(u, xs);
    ss := DGRAFO.sucesores(u);
    it NOT SEC.fin?(ss) →
      v := SEC.actual(ss);
      SEC.avanza(ss);
      VECTOR.Asigna(p, v, VECTOR.valor(p, v) - 1); % p(v) := p(v) - 1
      si VECTOR.valor(p, v) == 0
        entonces
          CJTO.Pon(v, m)
        sino
          seguir
      fsi
```

```
      fit
    fit
  { Q₀ : cont(xs) representa un recorrido de ordenación topológica de g }
  fproc
```

En cuanto a la complejidad, el algoritmo opera en tiempo:

— $O(NV^2)$ si el grafo está representado como matriz de adyacencia.

— $O(NV+NA)$ si el grafo está representado como vector de listas de adyacencia.

Este algoritmo se puede modificar para detectar si el grafo es acíclico o no, en lugar de exigir aciclicidad en la precondición. Si el conjunto *M* es queda vacío antes de haber visitado NV vértices, el grafo no es acíclico.

7.4 Caminos de coste mínimo

En este apartado vamos a estudiar algoritmos que calculan caminos de coste mínimo en grafos dirigidos valorados.

El coste de un camino se calcula como la suma de los valores de sus arcos. Se presupone por tanto que existe una operación de suma entre valores. En las aplicaciones prácticas, los valores suelen ser números no negativos. Sin embargo, la corrección de los algoritmos que vamos a estudiar sólo exige que el tipo de los valores satisfaga los requisitos expresados en la especificación de la clase de tipos VAL-ORD que presentamos al principio del tema.

7.4.1 Caminos mínimos con origen fijo.

Dado un vértice *u* de un grafo dirigido valorado *g*, nos interesa calcular todos los caminos mínimos con origen *u* y sus correspondientes costes.

Para resolver este problema vamos a utilizar el *algoritmo de Dijkstra* (1959). El algoritmo mantiene un conjunto *M* de vértices *v* para los cuales ya se ha encontrado el camino mínimo desde *u*. La idea del algoritmo es:

— Se inicializa $M := \{u\}$. Para cada vértice *v* diferente de *u*, se inicializa un coste estimado $C(v) := costeArista(G, u, v)$.

— Se entra en un bucle. En cada vuelta se elige un vértice *w* que no esté todavía en *M* y cuyo coste estimado sea mínimo. Se añade *w* a *M*, y se actualizan los costes estimados de los restantes vértices *v* que no estén en *M*, haciendo $C(v) := min(C(v), C(w) + costeArista(G, w, v))$.

Al terminar, se han encontrado caminos de coste mínimo *C(v)* desde *u* hasta los restantes vértices *v*.

Un par de comentarios sobre la implementación del algoritmo:

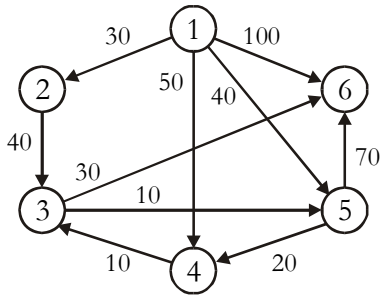
— Si en algún momento llega a cumplirse que para todos los vértices *v* que no están en *M* $C(v) = \infty$, el algoritmo puede terminar.

— Además de calcular *C*, calculamos otro vector *p*, indexado por los ordinales de los vértices, que representa los caminos mínimos desde *u* a los restantes vértices, según el siguiente criterio:

— *p(ord(v)) = 0* si *v* no es accesible desde *u*.

— *p(ord(v)) = ord(w)* si *v* es accesible desde *u* y *w* es el predecesor inmediato de *v* en el camino mínimo de *u* a *v* calculado por el algoritmo.

Como ejemplo del funcionamiento del algoritmo, vamos a ejecutarlo para el grafo dirigido de la figura siguiente, tomando como vértice inicial *u = 1*.



It.	M	w	c(1)/p(1)	c(2)/p(2)	c(3)/p(3)	c(4)/p(4)	c(5)/p(5)	c(6)/p(6)
0	{1}	–	0/1	30/1	∞/0	50/1	40/1	100/1
1	{1, 2}	2	0/1	30/1	70/2	50/1	40/1	100/1
2	{1, 2, 5}	5	0/1	30/1	70/2	50/1	40/1	100/1
3	{1, 2, 5, 4}	4	0/1	30/1	60/4	50/1	40/1	100/1
4	{1, 2, 5, 4, 3}	3	0/1	30/1	60/4	50/1	40/1	90/3
5	{1, 2, 5, 4, 3, 6}	6	0/1	30/1	60/4	50/1	40/1	90/3

De aquí se deduce, por ejemplo, que un camino mínimo de 1 a 6, con coste *c(v) = 90*, es [1, 4, 3, 6].

Este es un ejemplo típico de *algoritmo voraz*, porque sigue la estrategia de construir la solución haciendo en cada momento lo que parece *localmente óptimo* (elección de *w* en cada vuelta del bucle principal) sin reconsiderar nunca las decisiones tomadas.

En los apuntes de Mario se demuestra la corrección de este algoritmo.

```
proc Dijkstra( e g : DGrafo[Vértice, Valor]; e u : Vértice;
               s c : Vector[Vértice,Valor];
               s p : Vector [1..DIS.card()] de [0..DIS.card()] );
{ P₀ : cierto }
var
  m : Cjto[Vértice];            % Vértices para los que ya tenemos el
camino mínimo
  n : Nat;                % cardinal de m
  v, w : Vértice;
  fin : Bool;
  minCoste, nuevoCoste : Valor;
inicio
% inicialización de m, n, c, p y fin
  CJTO.Vacío(m);
  CJTO.Pon(u, m);
  n := 1;
```

```
  v := DIS.prim( );
  it menorIgual(v, DIS.ult( )) →                    % O(NV) con MA; O(NV+NA)
con VLA
    VECTOR.Asigna(c, v, WDGRAFO.costeArista(g, u, v));
    si VAL-ORD.igual( VECTOR.valor(c, v), VAL-ORD.infi() )
      entonces
        p(DIS.ord(v)) := 0            % v no tiene predecesor
      sino
        p(DIS.ord(v)) := DIS.ord(u)
    fsi;
    v := DIS.suc(v);
  fit;
  VECTOR.asigna(c, u, 0);
  p(DIS.ord(u)) := DIS.ord(u);
  fin := falso;
% bucle principal, O(NV^2)
  it n < DIS.card() AND NOT fin →           % O(NV) iteraciones
    % elegimos w ∉ m tal que c(w) sea mínimo
    minCoste := VAL-ORD.infi();
    v := DIS.prim();
    it DIS.menorIgual(v, DIS.ult()) → % O(NV)
      si NOT CJTO.pertenece(v, m) AND VAL-ORD.menor( VECTOR.valor(c, v),
minCoste )
        entonces
          minCoste := VECTOR.valor(c, v);
          w := v
        sino
          seguir
      fsi;
      v := DIS.suc(v);
    fit;
    si VAL-ORD.igual(minCoste, VAL-ORD.infi())
      entonces                  % No quedan más vértices accesibles desde
u. Terminamos
        fin := cierto
      sino               % añadimos w a m; actualizamos c y p
        CJTO.Pon(w, m); n := n + 1;
        v := DIS.prim();
        it DIS.menorIgual( v, DIS.ult() ) → % O(NV) con MA
          si NOT CJTO.pertenece(v, m)
            entonces
              nuevoCoste := VAL-ORD.suma(  VECTOR.valor(c, w),
                                           WDGRAFO.costeArista(g, w, v) );
              si
                entonces
                  VECTOR.Asigna( c, v, nuevoCoste );
                  p(DIS.ord(v)) := DIS.ord(w)
                sino
                  seguir
              fsi
            sino
              seguir
          fsi;
          v := DIS.suc(v)
        fit
    fsi
  fit
{ Q₀ : ∀ v : Vértice : c(v) es el coste de un camino mínimo de u a v ∧
```

```
        p(ord(v)) indica el número de orden del vértice predecesor
        inmediato de v en dicho camino                              }
fproc
```

Al definir el tipo del vector donde almacenamos los caminos ***Vector*** *[1..DIS.card( )]* ***de*** *[0..DIS.card( )]* estamos usando la información que aparece en la especificación de la clase DIS, donde se dice que los ordinales de los elementos están en el intervalo [1..card].

Complejidad del algoritmo de Dijkstra

La realización anterior está pensada para una representación de los grafos con matrices de adyacencia. El coste de las diferentes etapas:

- Incialización de *c* y *p*: $O(NV)$. NV iteraciones donde cada iteración sólo involucra operaciones con coste constante. Esto es cierto si los grafos se implementan con matrices de adyacencia, donde efectivamente *costeArista* es $O(1)$.
- El bucle principal se compone de $O(NV)$ iteraciones. En cada iteración:
  - La selección de *w* se realiza con un bucle de coste $O(NV)$. Siempre y cuando utilicemos una implementación de los conjuntos donde *pertenece* sea $O(1)$.
  - La actualización de *c* y *p* se realiza con un bucle que realiza $O(NV)$ iteraciones. Cada iteración tiene coste constante siempre y cuando *costeArista* tenga coste $O(1)$.

Por lo tanto, el coste total es $O(NV^2)$.

Si se utilizan grafos representados con listas de adyacencia, es necesario realizar algunos cambios en el algoritmo para obtener esta misma complejidad.. En lugar de recorrer todo el dominio de los vértices, consultado el coste de cada posible arista, se recorren exclusivamente los sucesores del vértice considerado en cada caso. La razón es que el coste de la operación *costeArista* es $O(1)$ en matrices de adyacencia y $O(NV)$ en listas de adyacencia. Por lo tanto los cambios son:

- Inicializaciones, para obtener tiempo $O(NV)$
  - *c* se inicializa con ∞ y *p* se inicializa con 0, excepto *c(u) := 0* y *p(ord(u)) := ord(u).* $O(NV)$
  - Para cada pareja *(c, v)* perteneciente a *sucesores(g, u)* se hace: *c(v) := c; p(v) := u.* $O(GS) \subseteq O(NV)$.
- Bucle principal, para obtener tiempo $O(NV^2)$

  Se cambia el bucle interno que actualiza *c* y *p*, escribiéndolo como bucle que recorre los sucesores de *w*. Si usamos una implementación de *sucesores* que no realice una copia de la secuencia, entonces el coste de esta operación es $O(1)$, si realizamos copia entonces es $O(GS)$. Con lo que la actualización de *c* y *p* se reduce a recorrer la secuencia de sucesores de *w*, que es una operación con coste $O(GS)$. Por lo tanto, tenemos $O(1) + O(GS)$ o $O(GS) + O(GS)$, en cualquier caso $O(GS)$. Como además se tiene $O(GS) \subseteq O(NV)$, tenemos que el coste de la actualización es $O(NV)$.

En algunos puntos del análisis con listas de adyacencia hemos utilizado $O(GS)$, indicando luego que este valor está acotado por $O(NV)$. Sin embargo, también se cumple que $O(GS) \subseteq O(NA)$, y esto nos hace pensar que aunque en el caso general NA es $O(NV^2)$, si tenemos un grafo disperso donde se cumple que $NA << NV^2$, podemos obtener mejores resultados utilizando listas de adyacencia. Siguiendo esta idea se pueden realizar una optimización del algoritmo de Djikstra que consiste en sustituir el conjunto *m* por una cola de prioridad formada por parejas *(c, v)*, donde *v* es

un vértice y $c = c(v)$. Para conseguir la eficiencia deseada, hay que implementar la cola de prioridad por medio de un montículo, y dotarla de operaciones algo diferentes a las operaciones habituales de las colas de prioridad. En [Fra94] pp. 323-325 se dan más detalles. Se consigue complejidad $O((NV+NA) \log NV)$, que es mejor que $O(NV^2)$ para grafos dispersos.

En cuanto al espacio, todas las versiones del algoritmo de Dijkstra que hemos estudiado requieren espacio adicional $O(NV)$ para el conjunto de vértices *m*, además del espacio ocupado por el propio grafo.

7.4.2 Caminos mínimos entre todo par de vértices

El problema es, dado un grafo dirigido valorado *g*, se quieren calcular todos los caminos mínimos entre todas las parejas de vértices de *g*, junto con sus costes.

Aplicando reiteradamente el algoritmo de Dijkstra, se obtiene una solución cuya complejidad depende de la variante del algoritmo de Dijkstra que se haya utilizado: $O(NV^3)$ o bien $O(NV \cdot NA \cdot \log NV)$.

La otra posibilidad es utilizar el *algoritmo de Floyd* (1962). Este algoritmo tiene la ventaja de ser más elegante y compacto. Además sólo necesita espacio adicional $O(1)$ (aparte del ocupado por el grafo y la solución calculada), mientras que cualquier versión del algoritmo de Dijkstra necesita espacio auxiliar $O(NV)$.

Mientras que el algoritmo de Dijkstra es *voraz*, el de Floyd es *dinámico*. Esto quiere decir que las iteraciones realizadas por el algoritmo se van aproximando a la solución, pero ninguna parte de ésta es definitiva hasta que no se termina.

La idea básica del algoritmo consiste en mejorar la estimación $c(u, v)$ del coste de un camino mínimo de *u* a *v* mediante la asignación:

```
c(v, w) := min( c(v, w), c(v, u) + c(u, w) )
```

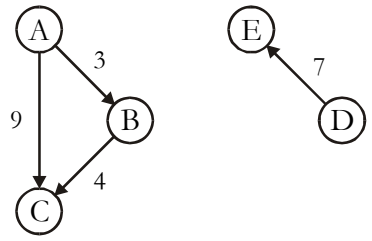
Diremos que el vértice *u* actúa como *pivote* en esta actualización de $c(v, w)$. El algoritmo comienza con una inicialización natural de *c*, y a continuación reitera la actualización de $c(v, w)$ para todas las parejas de vértices (v, w) y con todos los pivotes *u*.

Al igual que en el algoritmo de Dijkstra, construimos un resultado adicional que representa los caminos mínimos entre cada par de vértices. Este resultado viene representado por un vector bidimensional *s* indexado por parejas de ordinales de los vértices, según el siguiente criterio:

— $s(ord(v), ord(w)) = 0$ si *v* no hay caminos de *v* a *w*.

— $s(ord(v), ord(w)) = ord(u)$ si *w* es accesible desde *v* y *u* es el sucesor inmediato de *v* en el camino mínimo de *v* a *w* calculado por el algoritmo.

Apliquemos el algoritmo al siguiente grafo:



0: Estado inicial

Matriz de costes *c*

	A	**B**	**C**	**D**	**E**
A	0	3	9	∞	∞
B	∞	0	4	∞	∞
C	∞	∞	0	∞	∞
D	∞	∞	∞	0	7
E	∞	∞	∞	∞	0

Matriz de sucesores *s*

	1	**2**	**3**	**4**	**5**
1	1	2	3	0	0
2	0	2	3	0	0
3	0	0	3	0	0
4	0	0	0	4	5
5	0	0	0	0	5

1: Después de actualizar *c* y *s* usando *A* como vértice pivote.
Ningún arco entra en *A* por lo tanto usar *A* como pivote no mejora nada. *c* y *s* quedan inalterados.

2: Después de actualizar *c* y *s* usando *B* como vértice pivote.
Esto permite mejorar el coste del camino entre *A* y *C*. *c(A, C)* y *s(1, 3)* se modifican, el resto de las posiciones de las matrices no se modifican.

Matriz de costes *c*

	A	**B**	**C**	**D**	**E**
A	0	3	7	∞	∞
B	∞	0	4	∞	∞
C	∞	∞	0	∞	∞
D	∞	∞	∞	0	7
E	∞	∞	∞	∞	0

Matriz de sucesores *s*

	1	**2**	**3**	**4**	**5**
1	1	2	2	0	0
2	0	2	3	0	0
3	0	0	3	0	0
4	0	0	0	4	5
5	0	0	0	0	5

3, 4, 5: Actualizaciones de *c* y *s* usando como vértices pivote *C, D* y *E* respectivamente.
No mejoran nada, *c* y *s* quedan como en la figura anterior.

Finalmente la implementación del algoritmo:

```
proc Floyd( e g : DGrafo[Vértice, Valor];
            s c : Vector[Vértice, Vértice, Valor];
            s s : Vector [1..DIS.card(), 1..DIS.card()] de [0..DIS.card()]
);
{ P₀ : cierto }
var
  u, v, w : Vértice;
  c : Valor;
inicio
% inicialización de c y s. O(NV²) o O(NV+NA)
  v := DIS.prim( );
  it menorIgual(v, DIS.ult( )) →
    w := DIS.prim( );
    it menorIgual(w, DIS.ult( )) →
      VECTOR.Asigna(c, v, w, WDGRAFO.costeArista(g, v, w));
      si VAL-ORD.igual( VECTOR.valor(c, v, w), VAL-ORD.infi() )
        entonces
          s(DIS.ord(v), DIS.ord(w)) := 0
        sino
          s(DIS.ord(v), DIS.ord(w)) := DIS.ord(w)
      fsi;
      VECTOR.Asigna(c, v, v, VAL-ORD.Cero());
      s(DIS.ord(v), DIS.ord(v)) := DIS.ord(v)
      w := DIS.suc(w)
    fit;
    v := DIS.suc(v);
  fit;
% bucle principal, O(NV³)
  u := DIS.prim( );
  it menorIgual(u, DIS.ult( )) →
% actualizamos c y d usando u como pivote
    v := DIS.prim( );
    it menorIgual(v, DIS.ult( )) →
      w := DIS.prim( );
      it menorIgual(w, DIS.ult( )) →
        c := VAL-ORD.suma( VECTOR.valor(c, v, u), VECTOR.valor(c, u, w) );
        si VAL-ORD.menor( c, VECTOR.valor(c, v, w) )
          entonces              % podemos mejorar el camino pasando por u
            VECTOR.Asigna(c, v, w, c);
            s(DIS.ord(v), DIS.ord(w)) := s(DIS.ord(v), DIS.ord(u))
          sino
            seguir
        fsi;
        w := DIS.suc(w)
      fit;
      v := DIS.suc(v);
    fit;
    u := DIS.suc(u)
  fit
{ Q₀ : ∀ v, w : Vértice :
       c(v, w) es el coste de un camino mínimo de v a w ∧
       s(ord(v), ord(w)) indica el número de orden del vértice suecsor
       inmediato de v en dicho camino                                  }
fproc
```

La complejidad del algoritmo de Floyd es claramente $O(NV^3)$ si el grafo está representado como matriz de adyacencia. En el caso de que *g* esté representado como vector de listas de adyacen-

cia, conviene modificar el bucle doble de inicialización de *c* y *s*, cambiando el bucle interno que recorre todos los vértices *w* por un recorrido de los sucesores del vértice *v*. Así, la inicialización requiere tiempo $O(NV+NA)$, y el bucle principal sigue consumiendo tiempo $O(NV^3)$.

7.5 Ejercicios

Grafos: Modelos matemáticos y especificación algebraica

371. Especifica un TAD WDGRAFO[V :: DIS, W :: VAL-ORD] que represente el comportamiento de los grafos dirigidos valorados con vértices tomados del tipo discreto *V* y arcos etiquetados por valores tomados del tipo con valores ordenados W. Especifica convenientemente la clase de tipos VAL-ORD, e incluye en WDGRAFO operaciones para crear un grafo vacío (sin aristas), poner y quitar aristas, consultar el coste asociado a una arista, recorrer los sucesores inmediatos de un vértice, y recorrer los predecesores inmediatos de un vértice.

372. Modifica la especificación del ejercicio anterior para obtener TADs que describan el comportamiento de otras clases de grafos:

(a) WGRAFO[V :: DIS, W :: VAL-ORD] para los grafos valorados no dirigidos.
(b) DGRAFO[V :: DIS] para los grafos dirigidos no valorados.
(c) GRAFO[V :: DIS] para los grafos no dirigidos no valorados.
(d) WMGRAFO[V :: DIS, W :: VAL-ORD] para los multigrafos valorados.
(e) MGRAFO[V :: DIS] para los multigrafos no valorados.

373. Especifica un TAD REL[X, Y :: DIS] que describa el comportamiento de las relaciones binarias entre datos de tipo *X.Elem* y datos de tipo *Y.Elem* (siendo *X, Y* tipos discretos). REL deberá estar equipado con un tipo *Rel[X.Elem, Y.Elem]* y operaciones con los perfiles siguientes:

– `Vacía:` → `Rel[X.Elem, Y.Elem]`
Generadora. Construye una relación vacía.

– `Inserta: (Rel[X.Elem, Y.Elem], X.Elem, Y.Elem)` → `Rel[X.Elem, Y.Elem]`
Generadora. Añade una nueva pareja a la relación.

– `borra: (Rel[X.Elem, Y.Elem], X.Elem, Y.Elem)` → `Rel[X.Elem, Y.Elem]`
Modificadora. Quita una pareja de una relación.

– `está?: (Rel[X.Elem, Y.Elem], X.Elem, Y.Elem)` → `Bool`
Observadora. Reconoce si una pareja está en una relación.

– `fila: (Rel[X.Elem, Y.Elem], X.Elem)` → `Sec[Y.Elem]`
Observadora. Enumera una fila de una relación.

– `columna: (Rel[X.Elem, Y.Elem], Y.Elem)` → `Sec[X.Elem]`
Observadora. Enumera una columna de una relación.

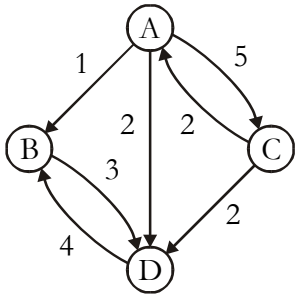
Observa la analogía entre el TAD DGRAFO[V :: DIS] del ejercicio 372(b) y los ejemplares REL[V :: DIS, V :: DIS] de REL.

374. Modifica la especificación del ejercicio anterior para obtener un TAD WREL[X, Y :: DIS, W :: VAL-ORD] que describa el comportamiento de las relaciones binarias valoradas.

Compara el TAD WDGRAFO[V :: DIS, W :: VAL-ORD] del ejercicio 371 con los ejemplares WREL[V :: DIS, V :: DIS, W :: VAL-ORD] de WREL.

Grafos: Técnicas de implementación

375. Dibuja la matriz de adyacencia y el vector de listas/secuencias de adyacencia que representen el grafo dirigido valorado de la figura siguiente.



376. Plantea dos implementaciones del TAD de los grafos dirigidos valorados, usando dos representaciones alternativas:

(a) Matriz de adyacencia indexada por pares de vértices.

(b) Vector de listas/secuencias de adyacencia, indexado por vértices.

Compara la eficiencia de las dos implementaciones, considerando los tiempos de ejecución de las operaciones y el espacio ocupado por la representación.

†377. Dibuja la representación del grafo del ejercicio 375 como vector de multilistas de adyacencia. Plantea una implementación del TAD de los grafos dirigidos valorados basada en esta representación, y analiza su eficiencia.

Sugerencia: Consulta el texto de Xavier Franch, sección 6.2.2, apartado c).

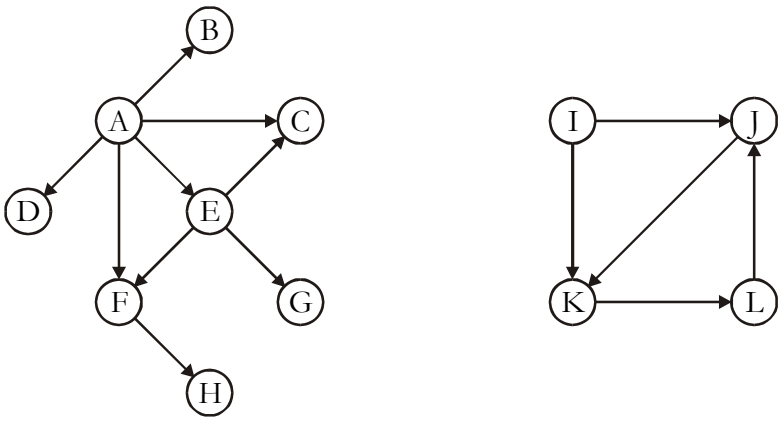
378. Estudia cómo modificar las técnicas de implementación del ejercicio 376 para adaptarlas a las otras variedades de grafos del ejercicio 372, así como a las relaciones de los ejercicios 382 y 383.

379. Especifica un TAD que describa las *matrices simétricas*. Diseña una implementación basada en la representación de una matriz simétrica $N \times N$ como vector, evitando representar dos veces los elementos que ocupen posiciones simétricas en la matriz. Plantea una implementación optimizada de los grafos no dirigidos (valorados o no) usando matrices de adyacencia simétricas como tipo representante.

380. En un *grafo completo* (i.e., grafo que posea todos los arcos posibles entre sus vértices) el número NA de arcos es del orden NV^2, siendo NV el número de vértices. Un grafo se llama *disperso* si NA es significativamente menor que NV^2. Estudia implementaciones de los grafos basadas en una representación de la matriz de adyacencia como *matriz dispersa* (cfr. ejercicio 366). Obviamente, esta técnica de implementación es recomendable para los grafos dispersos con un conjunto de vértices de cardinal grande.

Recorridos en grafos

381. Para un grafo dirigido, el *recorrido en profundidad* y el *recorrido por niveles* se efectúan de manera análoga al caso de los árboles, pero cuidando de que cada vértice se visite una sola vez aunque haya ciclos en el grafo. Para ello, se usa un conjunto de vértices auxiliar que contiene los vértices ya visitados. Dibuja en forma de bosques los resultados de recorrer en profundidad y por niveles el grafo dirigido de la figura siguiente, indicando el orden de visita de los vértices.



†382. Diseña un algoritmo de recorrido en profundidad para grafos dirigidos no valorados, construido como función recursiva que devuelva el recorrido en forma de secuencia de vértices. Usa un conjunto de vértices auxiliar para llevar cuenta de los vértices ya visitados. analiza la complejidad del algoritmo en función de la representación adoptada para el grafo.

†383. Diseña un algoritmo de recorrido por niveles para grafos dirigidos no valorados, construido como función iterativa que devuelva el recorrido en forma de secuencia de vértices. Usa una cola de vértices para controlar el recorrido, y un conjunto de vértices auxiliar para llevar la cuenta de los vértices ya visitados. Analiza la complejidad del algoritmo en función de la representación adoptada para el grafo.

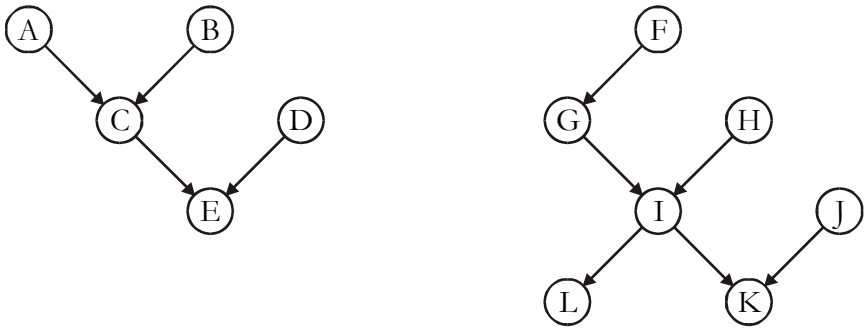
384. Modifica el algoritmo del ejercicio 383 cambiando la cola por una pila, de manera que resulte un algoritmo iterativo para el recorrido en profundidad de grafos dirigidos no valorados.

385. Desarrolla una implementación del TAD CJTO[E :: DIS] usando como tipo representante el tipo ***Vector*** *E* ***de*** *Bool*, de manera que las operaciones *Pon*, *quita* y *pertenece* sean ejecutables en tiempo constante, y las operaciones *Vacío* y *esVacío* sean ejecutables en tiempo lineal. Esta implementación de CJTO es recomendable para los conjuntos auxiliares utilizados en los algoritmos de recorrido de los ejercicios anteriores.

386. Dado un grafo dirigido acíclico G, la relación entre vértices definida como

$$u \prec_G v \Leftrightarrow_{\text{def}} \text{existe un camino de } u \text{ a } v \text{ en } G \text{ (i.e., } u \text{ es antepasado de } v)$$

es un orden parcial. Se llama *recorrido de ordenación topológica* de G a cualquier recorrido de G que visite cada vértice v solamente después de haber visitado todos los vértices de u tales que $u \prec_G v$. En general, son posibles varios recorridos de ordenación topológica para un mismo grafo G. Construye algunos de ellos para el grafo de la figura siguiente.



†387. Dado un grafo dirigido y acíclico *G*, se desea construir una secuencia de vértices *xs* que represente un recorrido de ordenación topológica de *G*. Diseña un procedimiento que resuelva este problema, y analiza su complejidad en función de la representación adoptada para el grado.

Idea: El algoritmo debe utilizar un bucle que elija en cada vuelta un vértice cuyos antepasados ya hayan sido visitados y que no haya sido visitado todavía, para añadirlo al recorrido. Conviene usar un conjunto de vértices auxiliar *M* que contiene en cada momento los vértices que tienen la propiedad de que todos sus antepasados han sido ya visitados.

388. Modifica el algoritmo del ejercicio anterior, obteniendo una variante que no exija en su precondición que *G* sea acíclico, y que detecte durante la ejecución si *G* es acíclico o no.

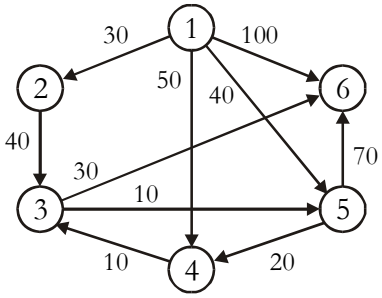
Idea: Si el conjunto *M* mencionado en el ejercicio anterior se queda vacío antes de haber visitado todos los vértices, entonces el grafo no es acíclico; en caso contrario, el grafo es acíclico, y el algoritmo termina completando un recorrido de ordenación topológica.

Búsqueda de caminos mínimos en grafos

389. Dados un grafo dirigido valorado *G* y un vértice *u* de *G*, el *algoritmo de Dijkstra* (1959) busca caminos de coste mínimo con origen en *u* y destinos en los demás vértices *v* de *G*. El algoritmo mantiene un conjunto *M* de vértices *v* para los cuales ya se ha encontrado el camino mínimo desde *u*. La idea del algoritmo es:

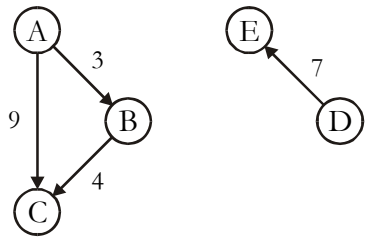
- Se inicializa *M* := {*u*}. Para cada vértice *v* diferente de *u*, se inicializa un coste estimado *C(v) := costeArista(G, u, v)*.
- Se entra en un bucle. En cada vuelta se elige un vértice *w* que no esté todavía en *M* y cuyo coste estimado sea mínimo. Se añade *w* a *M*, y se actualizan los costes estimados de los restantes vértices *v* que no estén en *M*, haciendo *C(v) := min(C(v), C(w) + costeArista(G, w, v))*.

Al terminar, se han encontrado caminos de coste mínimo *C(v)* desde *u* hasta los restantes vértices *v*. Ejecuta el algoritmo de Dijkstra para el grafo dirigido de la figura siguiente, tomando como vértice inicial *u = 1*.



†390. Diseña un procedimiento iterativo que implemente el algoritmo de Dijkstra, devolviendo un vector *C* indexado por vértices, tal que *C(v)* indique el coste del camino mínimo de *u* a *v*; y otro vector *P* indexado por ordinales de vértices, tal que *P(ord(v))* indique el ordinal del vértice predecesor de *v* en el camino mínimo desde *u* hasta *v* (por convenio, será 0 si no hay camino de *u* a *v*). Analiza la complejidad del algoritmo en función de la representación adoptada para el grafo, y estudia las optimizaciones que se describen en el texto de Xavier Franch, sección 6.4.1.

391. Aplica el algoritmo de Dijkstra al grafo dirigido y valorado de la figura siguiente, tomando *'A'* como vértice inicial. Observa que el algoritmo se detiene después de haber encontrado caminos mínimos para todos los vértices accesibles desde *'A'*.



392. Si se desea obtener caminos de coste mínimo entre todas las posibles parejas de vértices de un grafo dirigido y valorado dado, una posibilidad es aplicar reiteradamente el algoritmo de Dijkstra. Otro método más compacto y elegante es el *algoritmo de Floyd* (1962), que usa una matriz *C* indexada por parejas de vértices para calcular en *C(u, v)* el coste de un camino mínimo de *u* a *v*. La idea del algoritmo es:

- Se inicializan *C(u, u) := 0* y *C(u, v) := costeArista(G, u, v)* para *u* /= *v*.
- Se entra en un bucle anidado que recorre todas las ternas de vértices *u, v, w*, actualizando
  C(v, w) := min(C(v, w), C(v, u) + C(u, w)).

Al terminar, está garantizado que todos los valores *C(v, w)* corresponden a costes de caminos mínimos. ejecuta el algoritmo de Floyd para el grafo del ejercicio 391.

†393. Diseña un procedimiento iterativo que implemente el algoritmo de Floyd, devolviendo además de la matriz *C* mencionada en el ejercicio 392, otra matriz *S* indexada por parejas de ordinales de vértices, tal que *S(ord(v), ord(w))* indique el ordinal del vértice sucesor de *v* en un camino mínimo de *v* a *w* (por convenio, será 0 si no hay camino de *v* a *w*). Analiza la complejidad del algoritmo en función de la representación adoptada para el grafo.

BIBLIOGRAFÍA

[Bal93] Balcázar, J. L.: "Programación metódica". McGraw-Hill, 1993

[Brass08] Brass, P.: "Advanced Data Structures". Cambridge University Press, 2008

[BS04] Baldwin, D., Scragg, G.W.: "Algorithms and Data Structures. The Science of Computing". Charles River Media, 2004.

[CCMRSV93] Castro, J., Cucker, F., Messeguer, X., Rubio, A., Solano, L.,Valles, B.: "Curso de Programación". McGraw-Hill, 1993.

[Chang03] Chang, S.K. (Editor): "Data Structures and Algorithms". World Scientific, 2003.

[Fra94] Franch, X.: "Estructuras de datos. Especificación, diseño e implementación". Ediciones UPC. 1994

[HS94] Horowitz, E., Sahni, S.: "Fundamentals of Data Structures in Pascal", Computer Science Press, 1994

[HSM07] Horowitz, E., Sahni, S., Mehta, D.: "Fundamentals of Data Structures in C++", Segunda Edición. Silicon Press, 2007

[Kal90] Kaldewaij, A.: "Programming: the Derivation of Algorithms". Prentice Hall, 1990.

[Kingston90] Kingston, J.: "Algorithms and data structures: design, correctness, analysis". Addison-Wesley, 1990.

[MS08] Mehlhorn, K., Sanders, P.: "Algorithms and Data Structures. The Basic Toolbox". Springer Verlag. 2008

[MOV94] Martí Oliet, N., Ortega Mallén, Y., Verdejo López, J.A.: "Estructuras de datos y métodos algorítmicos. Ejercicios resueltos". Prentice-Hall, 2004

[Peñ05] Peña Marí, R.: "Diseño de programas. Formalismo y abstracción". Prentice Hall, 2005