Capítulo 2 Implementación de tipos abstractos de datos

La especificación de un tipo abstracto es el paso previo a la escritura de cualquier programa que lo manipule, porque permite describir inequívocamente su comportamiento; una vez establecida, se pueden construir diversas implementaciones usando un lenguaje de programación convencional. En este capítulo se introducen los mecanismos que proporciona la notación Merlí para codificar los tipos de datos, que son los habituales en los lenguajes de programación imperativos más utilizados (excepto algún aspecto puntual como la genericidad, ausente en los lenguajes más antiguos), se estudian las relaciones que se pueden establecer entre la implementación y la especificación de un TAD y, por último, se definen diversas estrategias de medida de la eficiencia de las implementaciones que permiten compararlas y determinar bajo qué circunstancias se puede considerar una implementación más adecuada que otra.

A lo largo del capítulo se supone que el lector conoce los constructores habituales que ofrecen los lenguajes de programación imperativos más habituales, como Modula-2, Pascal o C y, en general, todos aquellos que se pueden considerar derivados del Algol; por ello, no se explicarán en detalle los esquemas adoptados en Merlí, sino que tan sólo se mostrará su sintaxis, fácilmente traducible a cualquiera de ellos. Eso sí, se insistirá en los aspectos propios de la metodología de uso de tipos abstractos, derivados básicamente de la distribución en universos de las aplicaciones.

2.1 El lenguaje de implementación

En el primer capítulo se ha destacado la diferencia entre la especificación y la implementación de un TAD, que resulta en su visión a dos niveles diferentes. En concreto, dada la especificación (única) de un tipo se pueden construir diversas implementaciones y así escoger la más adecuada en cada contexto de uso. Cada una de estas implementaciones se encierra en un universo diferente, llamado universo de implementación, en contraposición a los universos de especificación y de caracterización explicados detalladamente en el capítulo anterior. Notemos que un universo de implementación no puede existir de forma independiente, sino que siempre se referirá a una especificación ya existente; para

establecer claramente el nexo entre un universo de implementación y el universo de especificación correspondiente, se consignará siempre en la cabecera del primero el nombre del segundo precedido por la palabra clave "implementa".

La construcción de un universo de implementación consta de dos fases:

- Elección de una representación para los diferentes géneros definidos en la especificación. Es habitual el uso de tipos auxiliares para estructurar el resultado, los cuales son invisibles a los usuarios del universo[1]. También se pueden definir constantes, implícitamente privadas.
- Codificación de todas y cada una de las diversas operaciones visibles de la especificación, en términos de las representaciones que se acaban de escribir. Igualmente puede ser necesario, e incluso recomendable, usar una o más operaciones auxiliares que surjan de la implementación de las operaciones usando la técnica de diseño descendente[1]; las operaciones auxiliares de la implementación no han de ser forzosamente las mismas de la especificación, y en general no lo son.

```
universo U_IMP implementa U
    usa U1, ..., Un
    const c1 vale v1, ... fconst
    tipo T1 es ... ftipo
    ...
    tipo privado Taux1 es ... ftipo
    ...
    función F1 ...
    ...
    función / acción privada Faux1 ...
    ...
funiverso
```

Fig. 2.1: esquema general de los universos de implementación.

Tanto en la representación como en la codificación de las operaciones pueden usarse identificadores de tipo y de operaciones visibles, definidos en otros universos cuyos nombres se escriben en una cláusula de usos al inicio del universo. En principio, los nombres se referirán a universos de especificación, ya que los programas dependen del comportamiento de los tipos que usen y no de su implementación; ahora bien, ya veremos a lo largo del texto que, en algunas situaciones y por razones de eficiencia, se exigirá que uno o más tipos usados en un universo estén implementados según una estrategia

[1] Aunque no sea estrictamente necesario, se usará la palabra clave "privado" para etiquetarlos.

determinada[2], en cuyo caso se consigna la implementación escogida mediante la cláusula "implementado con". Además de los usos, en un universo de implementación pueden aparecer ocultaciones, renombramientos e instancias; por lo que respecta a estas últimas, pueden implementar alguno de los tipos definidos en la especificación, en cuyo caso se determina la implementación elegida mediante la cláusula "implementado con".

Si el universo de especificación define un TAD parametrizado, los universos de implementación correspondientes también lo serán y en la implementación aparecerán, como mínimo, todos los parámetros formales de la especificación. En ocasiones, alguna técnica de implementación puede requerir parámetros formales adicionales que no afecten la especificación del tipo (por ejemplo, las técnicas de dispersión vistas en el capítulo 4).

2.1.1 Representación de tipos

Para representar un TAD, Merlí ofrece diversos tipos predefinidos, y también algunos constructores de tipo que permiten al usuario construir sus propias estructuras. Los tipos predefinidos del lenguaje son los booleanos, los naturales, los enteros, los caracteres y los reales, todos ellos con las operaciones habituales. Por lo que respecta a los constructores, se dispone de los siguientes:

- Escalares, o por enumeración de sus valores: se enumeran todos los valores del tipo uno detrás del otro. Se dispone de operaciones de comparación = y $\neq$.
- Tuplas: permiten la definición de un tipo como producto cartesiano. Cada componente o campo (ing., field) de la tupla se identifica mediante un nombre y se declara de un tipo determinado; dado un objeto v de tipo tupla, la referencia a su campo c se denota por v.c. El lenguaje permite la escritura de constantes tuplas: dada una tupla de n campos c_1, ..., c_n, el literal $<v_1, ..., v_n>$ representa la tupla tal que su campo c_i vale v_i, $1 \leq i \leq n$; evidentemente, los tipos de los v_i han de concordar con los tipos de los c_i. Las tuplas pueden ser tuplas variantes, es decir, tuplas tales que el nombre y el tipo de campos componentes varía en función del valor de uno de ellos, llamado discriminante.
- Vectores (ing., array): permiten la definición de tipos como funciones f: I $\rightarrow$ V; los elementos de I se llaman índices y los de V simplemente valores. Gráficamente se puede considerar un vector como una tabla que muestra el valor asociado a cada índice y cada celda se llama componente o posición. Dado un objeto A de tipo vector, la referencia al valor asociado al índice i se hace mediante A[i]. Se exige que el tipo de los índices sea natural, entero, carácter o bien un tipo escalar; generalmente no se usará todo el tipo para indexar el vector, sino que se especificará una cota inferior y una cota superior de manera que la referencia a un índice fuera del intervalo definido por estas cotas da error. El lenguaje permite la escritura de vectores literales: dado un vector de

[2] Incluso en este caso, la representación del tipo usado quedará inaccesible en virtud del principio de la transparencia de la representación propia del diseño modular con TAD.

n componentes, el literal $[v_1, ..., v_n]$ representa el vector tal que su posición i-ésima vale v_i, $1 \leq i \leq n$; a veces, en la constante tan sólo se quiere destacar el valor de una posición i concreta, y para ello pueden escribirse los literales [..., v_i, ...] o [..., $i \rightarrow x$, ...].

```
tipo semana es (lu, ma, mi, ju, vi, sa, do) ftipo
```

(a) tipo por enumeración de sus valores

```
tipo fecha es tupla
                día_s es semana
                día_m, mes, año son natural
              ftupla
ftipo
```

(b.1) tipo tupla

```
tipo trabajador es
    tupla
        nombre, apellido son cadena
        caso qué de tipo (jefe, analista, programador) igual a
            jefe entonces #subordinados es natural
            analista entonces #proyectos, #programadores son natural
            programador entonces #líneas_código es natural; senior? es bool
        fcaso
    ftupla
ftipo
```

(b.2) tipo tupla con partes variantes (el campo qué es el discriminante)

```
tipo estadística es vector [semana] de natural ftipo
```

(c.1) tipo vector; su índice es un tipo por enumeración

```
tipo cadena es vector [de 1 a máx_cadena] de carácter ftipo
```

(c.2) tipo vector; su índice es un intervalo de un tipo predefinido

```
tipo empresa es vector [de 1 a máx_empresa] de tupla
                                                  tr es trabajador
                                                  sueldo es real
                                               ftupla
ftipo
```

(c.3) tipo vector; sus valores usan otro tipo definido por el usuario

Fig. 2.2: ejemplos de uso de los constructores de tipo de Merlí.

2.1.2 Sentencias

Se describe a continuación el conjunto de sentencias o instrucciones que ofrece Merlí, cuya sintaxis se puede consultar en la fig. 2.3.

- Asignación: modificación del valor de un objeto de cualquier tipo.
- Secuencia: ejecución incondicional de las instrucciones que la forman, una tras otra.
- Sentencias condicionales: por un lado, se define la sentencia condicional simple, que especifica una instrucción que se ha de ejecutar cuando se cumple una condición booleana, y otra, opcional, para el caso de que no se cumpla. Por otro, se dispone de la sentencia condicional compuesta, que generaliza la anterior especificando una serie de condiciones y, para cada una, una sentencia para ejecutar en caso de que se cumpla. Las condiciones han de ser mutuamente exclusivas; también se puede incluir una sentencia para ejecutar en caso en que ninguna condición se cumpla.
- Sentencias iterativas o bucles. Existen diversos tipos:
  - ◊ Simple: se especifica el número de veces que se debe repetir una sentencia.
  - ◊ Gobernada por una expresión booleana: se repite la ejecución de una sentencia mientras se cumpla una condición de entrada al bucle, o bien hasta que se cumpla una condición de salida del bucle.
  - ◊ Gobernada por una variable: se define una variable, llamada variable de control, que toma valores consecutivos dentro de los naturales, de los enteros, de un tipo escalar o bien de un intervalo de cualquiera de ellos; el número de iteraciones queda fijado inicialmente y no cambia. En el caso de tipos escalares, los valores que toma la variable siguen el orden de escritura dentro de la declaración del tipo.

  Cada ejecución de las instrucciones que aparecen dentro de la sentencia iterativa se llama vuelta o también iteración.

  Al hablar de sentencias iterativas son básicos los conceptos de invariante y de función de acotamiento, que abreviaremos por I y F, respectivamente. El invariante es un predicado que se cumple tanto antes como después de cada vuelta y que describe la misión del bucle. La función de acotamiento implementa el concepto de terminación del bucle, por lo que se define como una función estrictamente decreciente que tiene como codominio los naturales positivos y que, generalmente, se formula fácilmente a partir de la condición del bucle. Tanto el uno como la otra se escriben usando las variables que intervienen en la sentencia iterativa. Es conveniente que ambos se determinen previamente a la escritura del código, porque definen inequívocamente el comportamiento del bucle. En este texto, no obstante, sólo se presentarán en aquellos algoritmos que, por su dificultad, se considere que aportan información relevante para su comprensión (por ello, se puede preferir una explicación informal a su expresión enteramente formal). Para profundizar en el tema, v. por ejemplo [Bal93] o [Peñ93], que proporcionan varias fuentes bibliográficas adicionales.

El lenguaje no admite ningún tipo de instrucción para romper la estructura dada por el uso de estas construcciones; es decir, no existen las sentencias goto, skip, break o similares que, a cambio de pequeñas reducciones del código resultante, complican la comprensibilidad, la depuración y el mantenimiento de los programas. Tampoco se definen las instrucciones habituales de entrada/salida, ya que no se necesitarán en el texto.

objeto := expresión

(a) asignación

$sent_1$; $sent_2$; ... ; $sent_n$

(b) secuencia; en lugar de ';', se pueden separar por saltos de línea

si expresión booleana entonces $sent_1$
si no $sent_2$ {opcional}
fsi

(c.1) sentencia alternativa simple

opción
caso expresión $booleana_1$ hacer $sent_1$
...
caso expresión $booleana_n$ hacer $sent_n$
en cualquier otro caso $sent_{n+1}$ {opcional}
fopción

(c.2) sentencia alternativa compuesta

repetir expresión natural veces sent frepetir

(d.1) sentencia iterativa simple

mientras expresión booleana hacer sent fmientras
repetir sent hasta que expresión booleana

(d.2) sentencias iterativas gobernadas por una expresión

para todo variable dentro de tipo hacer sent fpara todo
para todo variable desde cota inf hasta cota sup hacer sent fpara todo
para todo variable desde cota sup bajando hasta cota inf hacer sent fpara todo

(d.3) sentencias iterativas gobernadas por una variable

Fig. 2.3: forma general de las sentencias de Merlí.

2.1.3 Funciones y acciones

Son los dos mecanismos de encapsulamiento de las instrucciones que codifican las operaciones de un tipo. Las funciones son la correspondencia directa del concepto de operación de un TAD dentro del lenguaje: a partir de unos parámetros de entrada se devuelve un valor de salida y los parámetros, que tienen un nombre que los identifica y se declaran de un tipo dado, mantienen el mismo valor después de la ejecución de la función, aunque se modifiquen. En caso de que se devuelva más de un valor, se pueden encerrar entre '<' y '>' los nombres de los tipos de los valores que se devuelvan, e indicar así que en realidad se está devolviendo una tupla de valores (v. fig. 2.4, b). La función sólo puede devolver su valor una sola vez, al final del cuerpo. La recursividad está permitida e incluso recomendada como herramienta de diseño e implementación de algoritmos.

Aparte de los parámetros, dentro de la función pueden declararse variables locales, cuya existencia está limitada al ámbito de la función. Notemos que en los programas Merlí no existe el concepto de variable global, ya que una función sólo tiene acceso a sus parámetros y puede usar variables locales para cálculos intermedios. Este resultado se deriva del hecho de que no pueden imbricarse funciones ni declararse objetos compartidos entre diferentes módulos[3]. También es interesante notar que no hay un programa principal y unas funciones subordinadas, sino que una aplicación consiste en un conjunto de funciones de las cuales el usuario escogerá una, que podrá invocar las otras en tiempo de ejecución; el esquema resultante es altamente simétrico (no da preferencia a ningún componente) y favorece el desarrollo independiente y la integración posterior de diferentes partes de la aplicación.

La existencia de acciones dentro del lenguaje proviene de la necesidad de encapsular operaciones auxiliares que modifican diversos objetos; la definición de estas operaciones como funciones daría como resultado cabeceras e invocaciones engorrosas. Las acciones también presentan el mecanismo de variables locales. La incorporación de acciones en el lenguaje obliga a definir diversos tipos de parámetros: los parámetros sólo de entrada (precedidos por la palabra clave "ent" en la cabecera), que se comportan como los de las funciones; los parámetros sólo de salida (precedidos por "sal"), correspondientes a objetos que no tienen un valor significativo en la entrada y almacenan un resultado en la salida; y los parámetros de entrada y de salida (precedidos por "ent/sal"), que tienen un valor significativo en la entrada que puede cambiar en la salida[4]. En este texto no nos preocuparemos por los posibles problemas de eficiencia en el uso de funciones en vez de acciones (v. sección 2.3), sino que usaremos siempre la notación que clarifique el código al máximo; se supone que el programador podrá adaptar sin problemas la notación funcional a las características de su instalación concreta.

[3] En realidad, el mecanismo de memoria dinámica del lenguaje (v. apartado 3.3.3) rompe esta regla general, porque actúa como una variable global implícita.

[4] La mayoría de los lenguajes imperativos comerciales no distinguen entre parámetros sólo de salida y parámetros de entrada y de salida, clasificándolos ambos como parámetros por variable (también por referencia), mientras que denominan parámetros por valor a los parámetros sólo de entrada.

En la fig. 2.4 aparecen tres implementaciones posibles de una operación para el cálculo del máximo común divisor de dos números siguiendo el algoritmo de Euclides, las dos primeras como funciones, en versiones recursiva e iterativa, y la tercera como una acción que devuelve el resultado en el primero de sus parámetros. También se muestra una función que, dado un vector de naturales y un elemento, devuelve una tupla de dos valores: un booleano que indica si el elemento está dentro del vector o no y, en caso de que esté, el índice de la primera aparición.

```
función MCD (a, b son nat) devuelve nat es
var res es nat fvar
    si a = b entonces res := a
    si no si a > b entonces res := MCD(a - b, b) si no res := MCD(a, b - a) fsi
    fsi
devuelve res
```

(a.1) función recursiva para el cálculo del máximo común divisor de dos naturales

```
función MCD (a, b son nat) devuelve nat es
    mientras a ≠ b hacer
        si a > b entonces a := a - b si no b := b - a fsi
    fmientras
devuelve a
```

(a.2) función no recursiva para el cálculo del máximo común divisor de dos naturales

```
acción MCD (ent / sal a es nat; ent b es nat) es
    mientras a ≠ b hacer
        si a > b entonces a := a - b si no b := b - a fsi
    fmientras
facción
```

(a.3) acción no recursiva para el cálculo del máximo común divisor de dos naturales

```
función está? (A es vector [de 1 a máx] de nat; x es nat) devuelve <bool, nat> es
var i es nat; encontrado es bool fvar
    i := 1; encontrado := falso
    mientras (i ≤ máx) ∧ ¬encontrado hacer
        si A[i] = x entonces encontrado := cierto si no i := i + 1 fsi
    fmientras
devuelve <encontrado, i>
```

(b) función para la búsqueda de un elemento dentro de un vector

Fig. 2.4: ejemplos de funciones y acciones.

Por último, introducimos los conceptos de precondición y de postcondición de una operación. La precondición, abreviadamente P, es un predicado lógico que debe satisfacerse al comenzar la ejecución de una operación; la postcondición, abreviadamente Q, es otro predicado lógico que debe satisfacerse al acabar la ejecución de una operación siempre que se cumpliera previamente la precondición correspondiente. Ambos predicados dependen exclusivamente de los parámetros de la operación. La invocación de una función o acción sin que se cumpla su precondición es incorrecta y da como resultado un valor aleatorio. Así como la especificación algebraica es un paso previo a la implementación de un tipo de datos, el establecimiento de la precondición y la postcondición (llamada especificación pre-post) ha de preceder, sin duda, a la codificación de cualquier operación.

Debe distinguirse el uso de las especificaciones algebraicas de los TAD y de las especificaciones pre-post de las operaciones. Por un lado, la especificación algebraica da una visión global del tipo de datos y define el comportamiento de las operaciones para todos los valores; por el otro, la especificación pre-post describe individualmente el comportamiento de cada operación en términos de la representación escogida para el tipo, y es el punto de partida de una verificación posterior del código de la operación.

En este texto se escriben sólo las precondiciones y postcondiciones de las funciones y acciones que no implementan operaciones visibles de los TAD (es decir, funciones auxiliares y algoritmos que usan los TAD), que aparecen juntamente con el código mismo, a menudo en estilo informal para mayor claridad en su lectura y comprensión, o bien simplemente como comentario si se considera que la misión de la operación es suficientemente clara. En lo que respecta a las funciones de los TAD, se formula el modelo matemático correspondiente al tipo previamente a su especificación algebraica y se describe, aunque sólo sea informalmente, el comportamiento de las operaciones en términos de este modelo, lo que puede considerarse como una especificación pre-post de alto nivel. Para profundizar en el tema de las especificaciones pre-post y su uso dentro de la verificación de programas, se puede consultar [Bal93] y [Peñ93].

2.1.4 Ejemplo: una implementación para los conjuntos

Como aplicación de los diversos mecanismos introducidos en esta sección, se quiere implementar el TAD de los conjuntos con operación de pertenencia y un número máximo de elementos, para los cuales se ha presentado una especificación en la fig. 1.32. Las representaciones eficientes de los conjuntos serán estudiadas a lo largo del texto como aplicación de diferentes estructuras de datos que en él se presentan; de momento, no nos preocupa que la implementación escogida sea eficiente, sino que simplemente se quiere ilustrar la aplicación de los constructores del lenguaje, por lo que la estrategia consiste en usar un vector dentro del cual se almacenan los elementos en posiciones consecutivas, y se añade un entero que apunta a la posición que se ocupará cuando se inserte un nuevo

elemento. A un objeto (entero, natural, puntero -v. apartado 3.3.3-, etc.) que apunta a otro se lo denomina apuntador (también cursor, si el objeto apuntado es una posición de vector), y se empleará bajo diversas variantes en la práctica totalidad de las estructuras de datos estudiadas en el libro; en este caso, lo llamamos apuntador de sitio libre, por razones obvias.

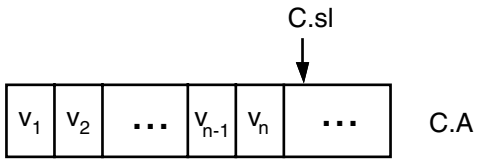


Fig. 2.5: representación del conjunto $C = \{v_1, v_2, ..., v_n\}$.

El universo parametrizado de la fig. 2.6 sigue esta estrategia. En primer lugar se establece cuál es la especificación que se está implementando. A continuación se escriben los universos de especificación que definen símbolos usados en la implementación. La representación del tipo no presenta ningún problema. Por último, se codifican las operaciones; notemos el uso de una operación auxiliar, índice, que se introduce para simplificar la escritura del universo y que se especifica pre-post según se ha explicado anteriormente. Destacamos que los errores que se establecen en la especificación han de tratarse en la implementación, ya sea incluyendo su negación en la precondición o bien detectándolos en el código de la función, como ocurre en la operación de añadir. En el segundo caso no se detalla el comportamiento del programa y se deja indicado por la palabra "error", porque su tratamiento depende del lenguaje de programación en que finalmente se implementará el programa, que puede ofrecer o no un mecanismo de excepciones.

2.2 Corrección de una implementación

Los elementos que ofrece el lenguaje para escribir universos de implementación permiten obtener programas ejecutables de la manera tradicional (mediante un compilador que genere código o bien a través de un intérprete); además, el programador puede verificar si el código de cada operación individual cumple su especificación pre-post. Ahora bien, es necesario definir algun método para comprobar si la implementación de un TAD es correcta respecto a su especificación ecuacional formulada previamente y éste es el objetivo de la sección[5]. Así, al igual que la programación de un módulo que use un TAD es independiente de la implementación del mismo, también su verificación podrá realizarse sin necesidad de recurrir a la representación del tipo, confiriendo un alto nivel de abstracción a las demostraciones.

[5] Una estrategia diferente consiste en derivar programas a partir de la especificación; si la derivación se hace aplicando técnicas formales, el resultado es correcto y no es preciso verificar el código. Esta técnica no se introduce en el presente texto; puede consultarse, por ejemplo, en [Bal93].

```
universo CJT_∈_ACOTADO_POR_VECT (ELEM_=, VAL_NAT)
    implementa CJT_∈_ACOTADO (ELEM_=, VAL_NAT)
    usa NAT, BOOL
    tipo cjt es
        tupla
            A es vector [de 0 a val-1] de elem
            sl es nat
        ftupla
    ftipo
    función Ø devuelve cjt es
    var C es cjt fvar
        C.sl := 0
    devuelve C
    función _∪{_} (C es cjt; v es elem) devuelve cjt es
        si índice(C, v) = C.sl entonces {se evita la inserción de repetidos}
            si C.sl = val entonces error   {conjunto lleno}
            si no C.A[C.sl] := v; C.sl := C.sl + 1
            fsi
        fsi
    devuelve C
    función _∈_ (v es elem; C es cjt) devuelve bool es
    devuelve índice(C, v) <  C.sl
    función lleno? (C es cjt) devuelve bool es
    devuelve C.sl = val
    {Función auxiliar índice(C, v) → k : devuelve la posición k que ocupa v en
     C.A; si v no aparece entre las posiciones de la 0 a la C.sl -1, devuelve C.sl.
        P ≅ { C.sl ≤ val } ∧ { ∀i, j: 0 ≤ i, j ≤C.sl-1: i ≠ j ⇒ C.A[i] ≠ C.A[j] }
        Q ≅ { está?(C, v) ⇒ C.A[k] = v } ∧ { ¬está?(C, v) ⇒ k = C.sl }
      siendo está? el predicado: está?(C, v) ≅ ∃i: 0 ≤ i ≤C.sl-1: C.A[i] = v   }
    función privada índice (C es cjt; v es elem) devuelve nat es
    var k es nat; encontrado es bool fvar
        k := 0; encontrado := falso
        mientras (k <  C.sl) ∧ ¬encontrado hacer
            {I ≅ ∀j: 0 ≤ j ≤ k-1: C.A[j] ≠ v ∧ (encontrado ⇒ C.A[k] = v). F ≅ C.sl - k}
            si C.A[k] = v entonces encontrado := cierto si no k := k + 1 fsi
        fmientras
    devuelve k
funiverso
```

Fig. 2.6: implementación para los conjuntos con pertenencia.

El punto clave consiste en considerar las implementaciones de los tipos abstractos también como álgebras, y entonces se podrá definir una implementación como la representación de los valores y operaciones del TAD en términos de los valores y las operaciones de otros TAD más simples. Para ello, se determina la función de abstracción de una implementación (ing., abstraction function, también conocida como función de representación), que transforma un valor v de la implementación del TAD en la clase correspondiente del álgebra cociente de términos; el estudio de esta función permite demostrar la corrección de la implementación.

La función de abstracción fue definida por primera vez por C.A.R. Hoare en "Proofs of Correctness of Data Representation", Acta Informatica, 1(1), 1972. Presenta una serie de propiedades que son fundamentales para entender su significado:

- Es parcial: puede haber valores de la implementación que no correspondan a ningún valor del TAD. La condición I_t que ha de cumplir una representación para denotar un valor válido del tipo t se llama invariante de la representación (ing., representation invariant) y la han de respetar todas las operaciones del tipo.
- Es exhaustiva: todas las clases de equivalencia del álgebra cociente de términos (que, recordemos, denotan los diferentes valores del TAD) se pueden representar con la implementación escogida.
- No es forzosamente inyectiva: valores diferentes de la implementación pueden denotar el mismo valor del TAD. Consecuentemente, la función de abstracción induce una relación de equivalencia R_t que llamamos relación de igualdad (representation equivalence en el artículo de Hoare) que agrupa en una clase los valores de la implementación que se corresponden con el mismo valor del TAD t.
- Es un homomorfismo respecto a las operaciones de la signatura del TAD; así, para toda operación $op \in OP_{t_1 \ldots t_n \to t}$ se cumple: $abs_t(op(v_1, \ldots, v_n)) = op_t(abs_{t_1}(v_1), \ldots, abs_{t_n}(v_n))$, donde abs_t es la función de abstracción del tipo t, op_t la interpretación de op dentro del álgebra del tipo, y la función de abstracción que se aplica sobre cada v_i es la correspondiente a su tipo.

En la fig. 2.7 se muestra la función de abstracción abs_{cjt} para el tipo de los conjuntos, abs_{cjt}: $A_{cjt} \rightarrow T_{cjt}$, siendo A_{cjt} el álgebra de la implementación de los conjuntos y T_{cjt} el álgebra cociente de términos (en realidad, la función devuelve un término que identifica la clase resultado); se puede observar que realmente se cumplen las propiedades enumeradas. Así, pueden establecerse el invariante de la representación I_{cjt} y la relación de igualdad R_{cjt} de la representación de los conjuntos; I_{cjt} acota el número de posiciones ocupadas del vector y prohíbe la aparición de elementos repetidos, mientras que R_{cjt} define la igualdad de dos estructuras si los sitios libres valen igual y hay los mismos elementos en las posiciones ocupadas de los vectores:

I_{cjt} (C es cjt) $\cong$ C.sl $\leq$ val $\wedge \forall$i, j: 0 $\leq$ i, j $\leq$ C.sl-1: i $\neq$ j $\Rightarrow$ C.A[i] $\neq$ C.A[j]

R_{cjt} (C_1,C_2 son cjt) $\cong$ C_1.sl = C_2.sl $\wedge$ ($\forall$i: 0 $\leq$ i $\leq$ C_1.sl-1: $\exists$ j: 0 $\leq$ j $\leq$ C_2.sl-1: C_1.A[i] = C_2.A[j])

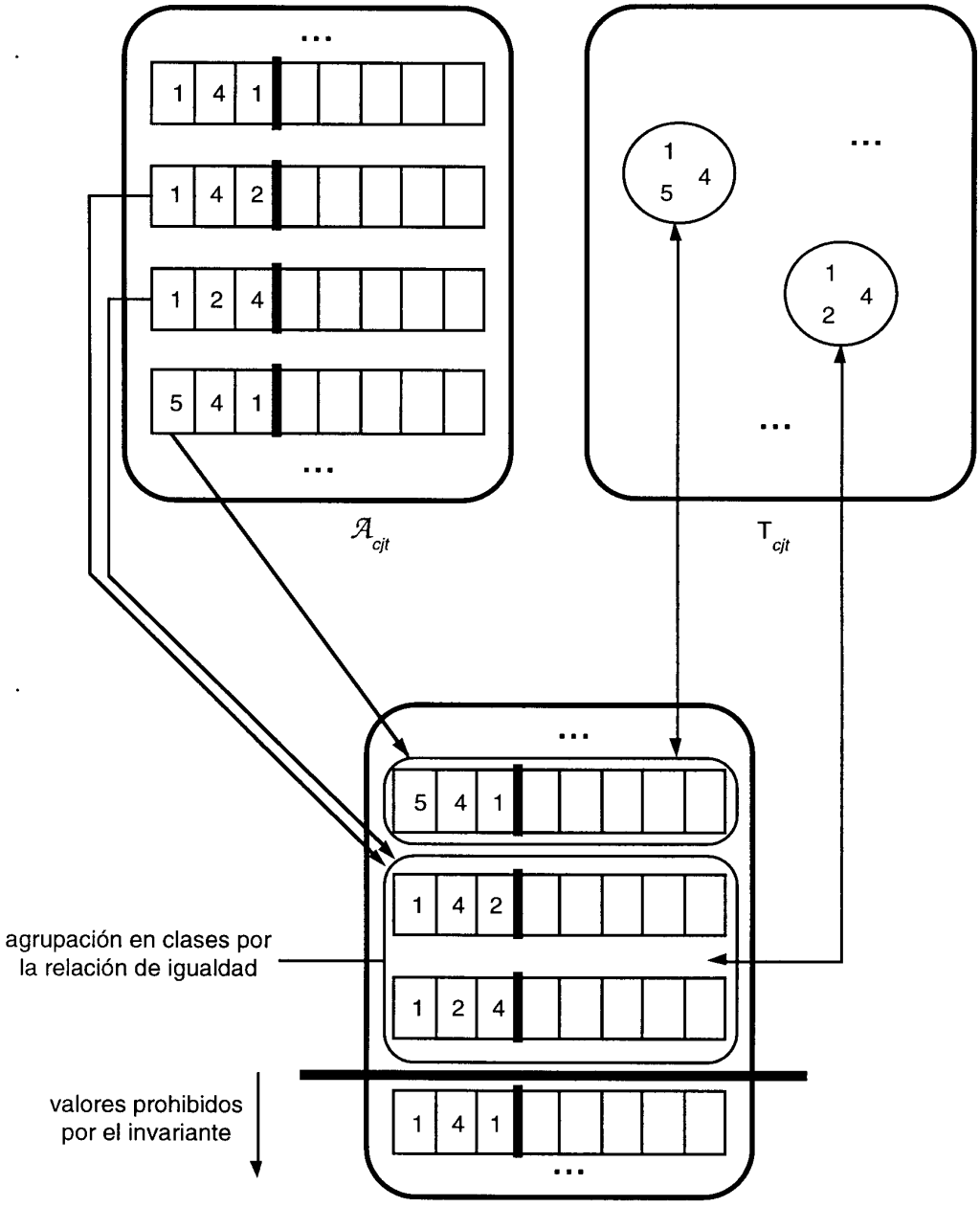


Fig. 2.7: función de abstracción aplicada sobre algunos valores de la representación de los conjuntos (las clases del álgebra cociente se han dibujado directamente como conjuntos).

Notemos que, al establecer la relación de igualdad sobre la representación, se trabaja directamente sobre el dominio de la función de abstracción, es decir, sólo se deben estudiar aquellos valores de la implementación que cumplan el invariante de la representación. Por este motivo en R_{cjt} no se tiene en cuenta la posibilidad de que haya elementos repetidos porque es una situación prohibida por I_{cjt}. Destaquemos también que la relación exacta entre estos dos predicados y la función de abstracción es (en el ejemplo de los conjuntos):

$$I_{cjt}(<A, sl>) \equiv <A, sl> \in dom(abs_{cjt}), \text{ y}$$
$$R_{cjt}(<A_1, sl_1>, <A_2, sl_2>) \equiv abs_{cjt}(<A_1, sl_1>) = abs_{cjt}(<A_2, sl_2>).$$

Es decir, la relación de igualdad sobre la representación es equivalente a la deducción ecuacional en el marco de la semántica inicial.

A continuación, ha de establecerse claramente cuál es el método de verificación de una implementación con respecto a su especificación, y es aquí donde aparecen varias alternativas propuestas por diversos autores, entre las que destacamos tres que estudiamos a continuación. Los formalismos que presentamos también son válidos si la especificación define más de un género, en cuyo caso cada género del universo de implementación dispondrá de sus propios predicados y, eventualmente, algunas demostraciones implicarán invariantes y relaciones de igualdad de diversos géneros.

Cronológicamente, el primer enfoque propuesto se basa en el uso de las pre y postcondiciones de las operaciones implementadas, y por ello se denomina comúnmente método axiomático. Este enfoque, formulado por el mismo C.A.R. Hoare en el artículo ya mencionado, requiere la escritura explícita de la función de abstracción, además de los predicados R_t e I_t; a continuación, puede establecerse de manera casi automática la especificación pre-post de las operaciones mediante las siguientes reglas:

- Toda operación tiene como precondición el invariante de la representación, así como la negación de todas aquellas condiciones de error que no se detecten en el cuerpo de la propia función; si la operación es una constante, la precondición es cierto.
- Las postcondiciones consisten en igualar la función de abstracción aplicada sobre la representación resultante con la aplicación de la operación sobre los parámetros de entrada, amén del propio invariante de la representación.

Así, por ejemplo, la operación de añadido de un elemento queda con la especificación:

$$\{I_{cjt}(s_1)\} \quad s_2 := s_1 \cup \{v\} \quad \{abs_{cjt}(s_2) = abs_{cjt}(s_1) \cup \{v\} \wedge I_{cjt}(s_2)\}^1$$

donde abs_{cjt} se puede escribir recursivamente como:

$$abs_{cjt}(<A, 0>) = \emptyset$$
$$abs_{cjt}(<A, x+1>) = abs_{cjt}(<A, x>) \cup \{A[x]\}$$

[1] Notemos que el símbolo $\cup$ está sobrecargado; por ello, algunos autores distinguen notacionalmente las referencias a la operación del TAD de las referencias a su implementación.

Una vez se dispone de la especificación pre-post, basta con verificar una a una todas las operaciones del tipo para concluir la corrección de la implementación. En dicha verificación, puede ser necesario utilizar la relación de igualdad. Como ejemplo ilustrativo, en el apartado 3.1 se presenta la verificación de la operación de añadir un elemento a una pila.

El segundo método, propuesto por J.V. Guttag, E. Horowitz y D.R. Muser en "Abstract Data Types and Software Validation", Communications ACM, 21(12), 1978, es un enfoque básicamente algebraico. Una vez escritos los predicados I_t y R_t, basta con restringir los valores del álgebra de la implementación mediante I_t, agrupar los valores resultantes en clases de equivalencia según R_t y, finalmente, demostrar la validez del modelo en la implementación. Es decir, es preciso efectuar cuatro pasos:

- Reescribir la implementación del tipo en forma de ecuaciones.
- Demostrar que R_t es efectivamente una relación de equivalencia y que se mantiene bajo transformaciones idénticas que involucren las operaciones de la signatura.
- Demostrar que I_t es efectivamente un invariante de la representación respetado por todas las operaciones constructoras del tipo.
- Demostrar que las ecuaciones de la especificación se satisfacen en la implementación.

Estudiemos el ejemplo de los conjuntos. Como resultado del primer paso, habrá un conjunto de ecuaciones por cada operación visible y, en ellas, toda variable de tipo cjt se habrá sustituido por la aplicación de la función de abstracción sobre la representación del tipo. Es necesario controlar que el invariante de la representación proteja de alguna manera las ecuaciones (ya sea explícitamente como premisa en una ecuación condicional, ya sea implícitamente, por construcción). En la fig. 2.8 se muestra el universo resultante, donde se aprecia el uso de operaciones sobre vectores, as y cons, cuyo significado es asignar un valor a una posición y consultarlo, y que se suponen especificadas en el universo VECTOR; además, se instancian los pares para poder representar los conjuntos mediante el vector y el apuntador. Notemos que las condiciones en el código se han traducido trivialmente a ecuaciones condicionales, mientras que los bucles se han transformado en ecuaciones recursivas. El universo resultante es utilizado en las tres fases siguientes de la verificación.

Para demostrar que la relación de igualdad se mantiene, basta con estudiar la igualdad de todas las parejas de términos posibles cuya operación más externa sea un símbolo de operación de la signatura, aplicadas sobre dos representaciones iguales según R_{cjt}; así, siendo $abs_{cjt}(<A_1, sl_1>) = abs_{cjt}(<A_2, sl_2>)$ por la relación de igualdad, debe cumplirse que $abs_{cjt}(<A_1, sl_1>) \cup \{v\} = abs_{cjt}(<A_2, sl_2>) \cup \{v\}$, $v \in abs_{cjt}(<A_1, sl_1>) = v \in abs_{cjt}(<A_2, sl_2>)$ y $lleno?(abs_{cjt}(<A_1, sl_1>)) = lleno?(abs_{cjt}(<A_2, sl_2>))$. Por lo que respecta a la conservación del invariante, se debe demostrar que, para toda operación constructora f, el resultado proteje el invariante, suponiendo que los parámetros de tipo cjt lo cumplen inicialmente. En el caso de que f no tenga parámetros de tipo cjt, basta con comprobar que la estructura se crea cumpliendo el invariante. En el último paso, se comprueba la validez de las ecuaciones del

tipo implementado una por una, previa sustitución de las operaciones que en éllas aparecen por su implementación, usando R_{cjt}, I_{cjt}, las ecuaciones de la implementación y las de los tipos que aparecen en la implementación (vectores, tuplas, y similares, incluyendo posibles TAD definidos por el usuario); así, por ejemplo, debe demostrarse que s∪{v}∪{v} = s∪{v}, utilizando, entre otras, las ecuaciones de los vectores tales como cons(as(A, k, v), k) = v.

universo CJT_∈_ACOTADO_POR_VECT (ELEM_=, VAL_NAT)
implementa CJT_∈_ACOTADO (ELEM_=, VAL_NAT)
usa NAT, BOOL
instancia VECTOR(ÍNDICE es ELEM_=, VALOR es ELEM) donde
ÍNDICE.elem es nat, ÍNDICE.= es NAT.=, VALOR.elem es ELEM.elem
instancia PAR (A, B son ELEM) donde A.elem es vector, B.elem es nat
tipo cjt és par
error ∀A∈vector; ∀sl∈nat: [NAT.ig(sl, val) ∧ ¬ v∈ abs_{cjt}(<A, sl>)] ⇒ abs_{cjt}(<A, sl>)∪{v}
ecns ∀A∈vector; ∀sl∈nat; ∀v∈elem
Ø = abs_{cjt}(<VECTOR.crea, 0>)
[v∈ abs_{cjt}(<A, sl>)] ⇒ abs_{cjt}(<A, sl>)∪{v} = abs_{cjt}(<A, sl>)
[¬ v∈ abs_{cjt}(<A, sl>)] ⇒ abs_{cjt}(<A, sl>)∪{v} = abs_{cjt}(<VECTOR.as(A, sl, v), sl+1>)
v∈ abs_{cjt}(<A, 0>) = falso
[sl ≠ 0] ⇒ v ∈ abs_{cjt}(<A, sl>) = (v = VECTOR.cons(A, sl-1)) ∨ (v∈ abs_{cjt}(<A, sl-1>))
lleno?(abs_{cjt}(<A, sl>)) = NAT.ig(sl, val)
funiverso

Fig. 2.8: implementación por ecuaciones para los conjuntos usando vectores.

El tercer método que se presenta fue propuesto por H. Ehrig, H.-J. Kreowski, B. Mahr y P. Padawitz en "Algebraic Implementation of Abstract Data Types", Theoretical Computer Science, 20, 1982, y se basa en el uso de la especificación algebraica del tipo que se utiliza para implementar. Concretamente, siendo t_{spec} el tipo que se ha de implementar definido en el universo SPEC = ({t_{spec}}, OP_{spec}, E_{spec}), y siendo t_{impl} el tipo implementador definido en el universo IMPL = ({t_{impl}}, OP_{impl}, E_{impl}), se trata de establecer un isomorfismo entre el álgebra cociente de términos T_{SPEC} y otro objeto algebraico construido a partir de la implementación de t_{spec} por t_{impl}. Sea ABS = (S_{abs}, OP_{abs}, E_{abs}) el universo que define esta nueva álgebra; su construcción sigue las fases siguientes:

- Síntesis: inicialmente, se define ABS = ({t_{spec}, t_{impl}}, OP_{spec} ∪ OP_{impl}, E_{impl}). A continuación se efectúa la implementación de las operaciones de OP_{spec} en términos de las de OP_{impl}. Al escribirlas, se necesitan, además, una o más operaciones para convertir un objeto de género t_{impl} a género t_{spec}.

- Restricción: en este paso se olvidan todos aquellos valores de la implementación que no son alcanzables y, además, se esconden los géneros y las operaciones de OP_{impl} para que el usuario de la implementación no pueda utilizarlas.
- Identificación: finalmente, se incorporan las ecuaciones de la especificación a E_{abs}, para asegurar que se igualen todas aquellas representaciones realmente equivalentes.

(Notemos la similitud de los pasos de restricción y de identificación con el invariante de la representación y la relación de equivalencia de los enfoques anteriores.) Finalmente, se considera que la implementación es correcta si el álgebra cociente de términos T_{ABS} de este universo es isomorfa a T_{SPEC}.

Si aplicamos este método al ejemplo de los conjuntos, se podría pensar en tomar como punto de partida la implementación por ecuaciones del universo de la fig. 2.8. Ahora bien, hay diversas situaciones anómalas; por ejemplo, dados dos elementos v_1 y v_2, las tuplas resultantes de añadir primero v_1 y después v_2 y viceversa son diferentes y, al particionar el álgebra de la implementación, van a parar a clases diferentes e impiden establecer un isomorfismo con el álgebra cociente de la especificación de los conjuntos. Estas situaciones eran irrelevantes en el método de J.V. Guttag et al., donde no se necesitaba definir ningún isomorfismo, sino que simplemente se comprobaba que el comportamiento determinado por la implementación "simulaba" correctamente el comportamiento establecido en la especificación. Estos y otros inconvenientes se solucionan en la fig. 2.10, que sí sigue el método y que se explica a continuación.

En el paso de síntesis se define el nuevo género cjt y una operación, abs, que convierte un valor de la implementación en otro de la especificación y que es la constructora generadora (una especie de función de abstracción que se limita a "copiar" el valor de la implementación en la especificación, y que es necesaria para no tener problemas con los tipos de los parámetros y los resultados de las operaciones). A continuación, se implementan por ecuaciones las operaciones de los conjuntos en términos de las tuplas. En la fase de restricción, se prohíben repeticiones en los vectores, se limita el valor máximo permitido del sitio libre (en realidad, la última condición no es estrictamente necesaria tal como se ha sintetizado previamente el tipo) y se olvidan todos aquellos símbolos invisibles para el usuario del TAD. Por último, se aplican las ecuaciones de los conjuntos para asegurar la equivalencia de los valores. En la fig. 2.9 se muestra todo el proceso aplicado sobre diversas representaciones que denotan el mismo valor del tipo.

Una vez presentados estos tres métodos, cabe preguntarse en qué contexto puede ser más adecuado uno u otro. En el marco de desarrollo de aplicaciones modulares, una implementación puede constar de diversas etapas; por ejemplo, un conjunto se podría representar mediante una secuencia y, a su vez, una secuencia se podría representar mediante el esquema de vector y apuntador que acabamos de ver. Esta situación se habría dado en caso de haber existido la implementación para las secuencias y de que se hubiera

considerado que su uso facilitaba la implementación de los conjuntos. La corrección de esta implementación se puede verificar usando el enfoque de H. Ehrig et al. para la primera etapa y el enfoque de J.V. Guttag et al. o de C.A.R. Hoare para la segunda, dado que la implementación de los conjuntos se basa en la especificación algebraica que definen las secuencias, mientras que la implementación de las secuencias se hace directamente usando los constructores del lenguaje.

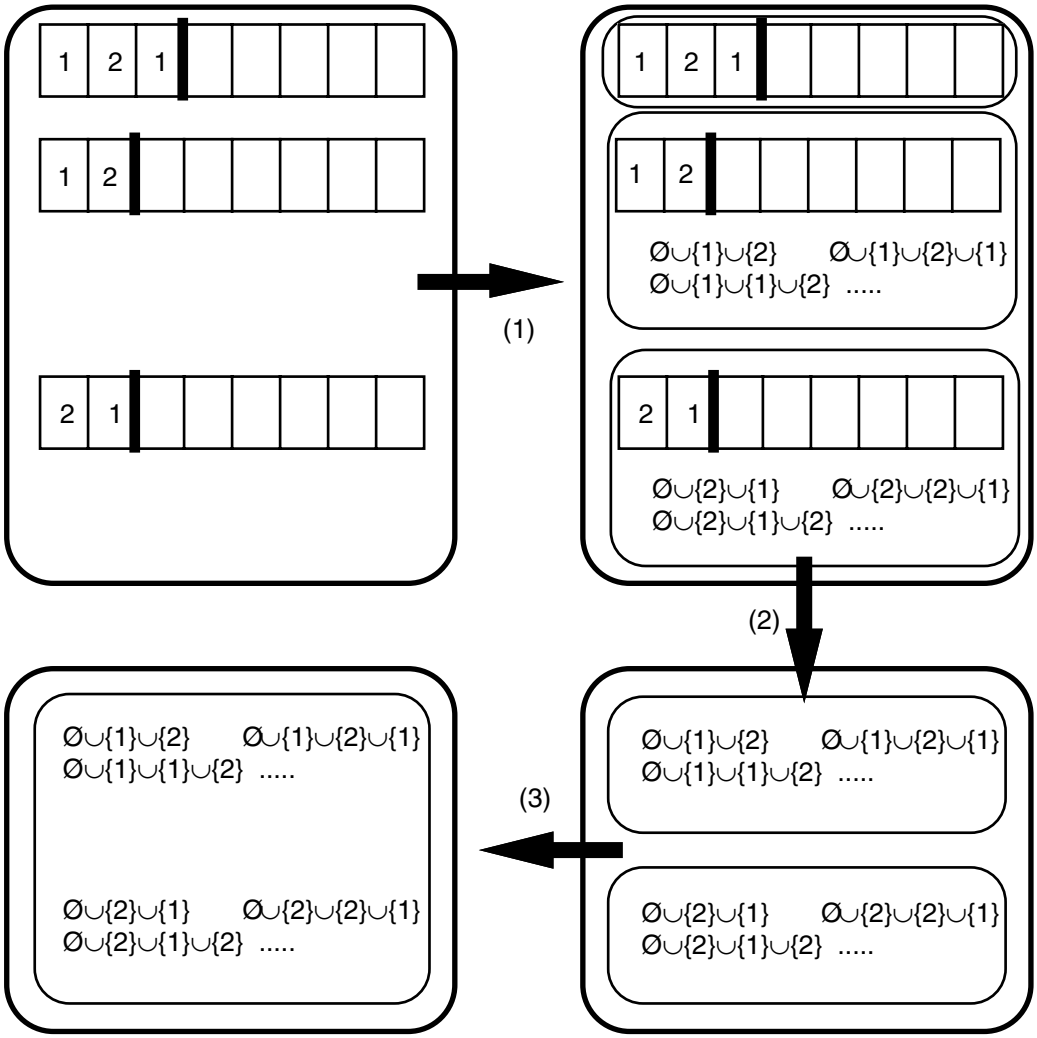


Fig. 2.9: implementación de los conjuntos: (1) síntesis; (2) restricción; (3) identificación.

universo CJT_∈_ACOTADO_POR_VECT (ELEM_=, VAL_NAT) es

usa NAT, BOOL

instancia VECTOR(ÍNDICE es ELEM_=, VALOR es ELEM) donde
ÍNDICE.elem es nat, ÍNDICE.= es NAT.=, VALOR.elem es ELEM.elem
instancia PAR(A, B son ELEM) donde A.elem es vector, B.elem es nat

{Paso 1: síntesis}

tipo cjt

ops abs: par → cjt
Ø: → cjt
_∪{_}: cjt ELEM.elem → cjt
_∈_: ELEM.elem cjt → bool
lleno?: cjt → bool

errores [lleno?(abs(<A, sl>)) ∧ ¬v ∈ abs(<A, sl>)] ⇒ abs(<A, sl>)∪{v}

ecns Ø = abs(<VECTOR.crea, 0>)
[v∈abs(<A, sl>)] ⇒ abs(<A, sl>)∪{v} = abs(<A, sl>)
[¬v∈abs(<A, sl>)] ⇒ abs(<A, sl>)∪{v} = abs(<VECTOR.as(A, sl, v), sl+1>)
v∈abs(<A, NAT.0>) = falso
[sl ≠ 0] ⇒ v∈abs(<A, sl>) = (v = VECTOR.cons(A, sl-1)) ∨ (v∈abs(<A, sl-1>))
lleno?(abs(<A, sl>)) = NAT.ig(sl, val)

{Paso 2: restricción}

errores as(as(A, i, v), i, v); [sl > val] ⇒ <A, sl>

esconde VECTOR.vector, VECTOR.crea, VECTOR.as, VECTOR.cons
PAR.par, PAR._.c1, PAR._.c2

{Paso 3: identificación}

errores [lleno?(C) ∧ ¬v∈C] ⇒ C∪{v}

ecns C∪{v}∪{v} = C∪{v}
C∪{v_1}∪{v_2} = C∪{v_2}∪{v_1}
...etc. {v. fig. 1.32}

funiverso

Fig. 2.10: aplicación de la técnica de tres pasos: síntesis, restricción e identificación.

A partir de la introducción dada aquí, queda claro que las demostraciones de corrección de las implementaciones son ciertamente complicadas sobrepasando los objetivos de este texto, y por ello no se adjuntan. Ahora bien, para mayor legibilidad incluiremos siempre los invariantes de la representación en forma de predicados booleanos que acompañarán a las representaciones del tipo; su escritura complementa las explicaciones que aparecen.

2.3 Estudio de la eficiencia de las implementaciones[2]

Como ya se ha dicho, un tipo de datos descrito mediante una especificación algebraica se puede implementar de varias maneras, que diferirán en la representación del tipo y/o en los algoritmos y detalles de codificación de algunas de sus funciones. La razón de que existan múltiples implementaciones de un único TAD es la adaptación a un contexto determinado de uso dado que, generalmente, diversas aplicaciones exigirán diferentes requisitos de eficiencia y, por ello, no se puede afirmar que una implementación concreta es "la mejor implementación posible" de un TAD (a pesar de que muchas veces es posible rechazar completamente una estrategia concreta de implementación, porque no es la mejor en ningún contexto). En el ejemplo de los conjuntos, una aplicación con pocas inserciones y muchas consultas reclamará la optimización de la operación de pertenencia por encima de la operación de inserción, pero hay diversos algoritmos en este libro que exigen que ambas operaciones sean igualmente rápidas, y entonces es necesario aplicar estrategias que consuman espacio adicional. Idealmente, el conjunto de todas las implementaciones existentes de un tipo de datos ha de ser lo suficientemente versátil como para ofrecer la máxima eficiencia al integrarlo dentro de cualquier entorno[3].

Para estudiar la integración correcta de una implementación dentro de una aplicación primero es necesario determinar los criterios que definen su eficiencia y después formular una estrategia de medida de los recursos consumidos; de esta manera, será posible clasificar los algoritmos y, también, compararlos cuando son funcionalmente equivalentes. Por lo que respecta al primer punto, los dos factores más obvios ya se han citado intuitivamente en el párrafo anterior y son los que usaremos a lo largo del texto: el tiempo de ejecución de las diversas funciones del tipo y el espacio necesario para representar los valores. Normalmente son criterios confrontados (siempre habrá que sacrificar la velocidad de una o más operaciones del tipo y/o espacio para favorecer otras o bien ahorrar espacio) y por ello su estudio tendrá que ser cuidadoso. Hay otros factores que pueden llegar a ser igualmente importantes, sobre todo en un entorno industrial de desarrollo de programas a gran escala, pero que raramente citaremos en este texto a causa de la dificultad para medirlos; destacan el tiempo de desarrollo de las aplicaciones (que es un criterio profusamente empleado en la industria, medido en hombres/año), el dinero invertido en escribirlas, el posible hardware adicional que exijan, la facilidad de mantenimiento, etc.

[2] Se podría argumentar que el estudio de la eficiencia de los programas resulta irrelevante desde el momento en que la potencia y la memoria de los computadores crece y crece sin fin. Ahora bien, este crecimiento, en realidad, incita a la resolución de problemas cada vez más complejos que exigen administrar cuidadosamente los recursos del computador y, por ello, el estudio de la eficiencia sigue vigente. Además, existe una clase de problemas (que aquí llamaremos "de orden de magnitud exponencial", v. 2.3.2) cuya complejidad intrínseca no parece que pueda ser solventada por las mejoras del hardware.

[3] Ahora bien, no es necesario construir *a priori* todas las implementaciones que se crean oportunas para un tipo, sino que se han de escribir a medida que se precisen (a no ser que se quiera organizar una biblioteca de módulos de interés general).

Centrémonos en la elección de las estrategias de medida de la eficiencia. Se podría considerar la posibilidad de medir el tiempo de ejecución del programa en segundos (o fracciones de segundo) y el espacio que ocupa en bytes o similares[4], pero estas opciones ocasionan una serie de problemas que las hacen claramente impracticables:

- Son poco precisas: un programa que ocupa 1 Mbyte y tarda 10 segundos en ejecutarse, ¿es eficiente o no? Es más, comparado con otro programa que resuelva el mismo enunciado, que ocupe 0.9 Mbytes y tarde 9 segundos en ejecutarse, ¿es significativamente mejor el primero que el segundo o la diferencia es desdeñable y fácilmente subsanable?
- Son a posteriori : hasta que no se dispone de código ejecutable no se puede estudiar la eficiencia.
- Dependen de parámetros diversos:
  - ◊ De bajo nivel: del hardware subyacente, del sistema operativo de la máquina, del compilador del lenguaje, etc. Esta dependencia es nefasta, porque evita la extrapolación de los resultados a otros entornos de ejecución.
  - ◊ De los datos de entrada: el comportamiento del programa puede depender y, generalmente, dependerá del volumen de datos para procesar y posiblemente de su configuración.

En consecuencia, la medida de la eficiencia de los programas en general y de las estructuras de datos en particular sigue un enfoque diferente, que consiste en centrarse precisamente en el estudio de los datos de entrada del programa. Para conseguir que este método sea realmente previo al desarrollo total del código, se caracterizan las operaciones del TAD mediante el algoritmo subyacente y, a partir de éste, se deduce una función que proporciona la eficiencia parametrizada por el volumen de los datos que procesa; de forma similar, para el espacio se caracteriza la representación del tipo mediante la estructura de datos empleada. Para independizar el método de los parámetros de bajo nivel citados antes, la eficiencia se mide con las denominadas notaciones asintóticas que se introducen a continuación.

Queda un problema por resolver. Como ya se ha dicho, para un mismo volumen de datos de entrada el comportamiento del programa puede variar substancialmente según el dominio concreto que se maneje; por ejemplo, la búsqueda de un elemento dentro de un vector de n posiciones exige un número de comparaciones que oscila entre 1 y n, según donde se encuentre el elemento (si está). Por este motivo, parece adecuado formular un caso medio a partir de la frecuencia de aparición esperada de los datos. Ahora bien, normalmente es imposible caracterizar la distribución de probabilidad de los datos de entrada, ya sea porque se desconoce, o bien porque se conoce, pero su análisis se vuelve matemáticamente impracticable; debido a esto y a que, habitualmente, no se busca tanto una eficiencia determinada como una cota inferior que asegure que bajo ningún concepto el

[4] Por ejemplo, D.E. Knuth formula la eficiencia de los algoritmos a partir del número de instrucciones en un computador idealizado (v. [Knu68] y [Knu73]).

comportamiento del programa será inaceptable, se opta tradicionalmente por estudiar el caso peor en la distribución de los datos de entrada; eso sí, cuando se crea conveniente y sea posible, se hará referencia al caso medio (el caso mejor se descarta por ilusorio y sólo se referirá casualmente como anécdota).

2.3.1 Notaciones asintóticas

Las notaciones asintóticas (ing., asymptotic notation) son las estrategias de medida de la eficiencia seguidas en este texto: a partir de una función de dominio y codominio los naturales positivos, f : $N^+ \to N^+$, que caracteriza la eficiencia espacial de la representación de un tipo o bien la eficiencia temporal de una operación en función del volumen de los datos utilizados, una notación asintótica define un conjunto de funciones que acotan de alguna manera el crecimiento de f. El calificativo "asintótica" significa que la eficiencia se estudia para volúmenes de datos grandes por ser éste el caso en que la ineficiencia de un programa es realmente crítica. El estudio del caso asintótico permite desdeñar los factores de bajo nivel citados al principio de esta sección; la independencia del entorno de ejecución se traduce en un estudio, no tanto de los valores de f, como de su ritmo de crecimiento, y da como resultado unas definiciones de las notaciones asintóticas que desdeñan, por un lado, los valores más bajos del dominio por poco significativos y, por el otro, las constantes multiplicativas y aditivas que pueden efectivamente considerarse irrelevantes a medida que el volumen de datos aumenta.

Hay varias notaciones asintóticas, explicadas en detalle en diferentes libros y artículos. El artículo pionero en la aplicación de estas notaciones en el campo de la programación fue escrito por D.E. Knuth en el año 1976 ("Big Omicron and Big Omega and Big Theta", SIGACT News, 8(2)), que presenta, además, una descripción histórica (ampliada por P.M.B. Vitányi y L.G.L.T. Meertens en "Big Omega versus the Wild Functions", SIGACT News, 16(4), 1985). G. Brassard mejoró la propuesta de Knuth definiendo las notaciones como conjuntos de funciones (v. "Crusade for a Better Notation", SIGACT News, 17(1), 1985), que es el enfoque que se sigue actualmente; en [BrB87], el mismo Brassard, junto con P. Bratley, profundiza en el estudio de las notaciones asintóticas.

En este texto introducimos las tres notaciones asintóticas más habituales. Las dos primeras definen cotas superiores y inferiores de los programas: la notación O grande (ing., big Oh o big Omicron), o simplemente O, para la búsqueda de cotas superiores y la notación omega grande (ing., big Omega), o simplemente Ω, para la búsqueda de cotas inferiores. Concretamente, dada una función f : $N^+ \to N^+$, la O de f, denotada por O(f), es el conjunto de funciones que crecen como máximo con la misma rapidez que f y la omega de f, denotada por Ω(f), es el conjunto de funciones que crecen con mayor o igual rapidez que f. Dicho en otras palabras, f es una cota superior de todas las funciones que hay en O(f) y una cota inferior de todas las funciones que hay en Ω(f).

$$O(f) \cong \{g: \mathbb{N}^+ \to \mathbb{N}^+ / \exists c_0: c_0 \in \mathbb{R}^+: \exists n_0: n_0 \in \mathbb{N}^+: \forall n: n \geq n_0: g(n) \leq c_0 f(n)\}$$
$$\Omega(f) \cong \{g: \mathbb{N}^+ \to \mathbb{N}^+ / \exists c_0: c_0 \in \mathbb{R}^+: \exists n_0: n_0 \in \mathbb{N}^+: \forall n: n \geq n_0: g(n) \geq c_0 f(n)\}$$[5]

La constante n_0 identifica el punto a partir del cual f es cota de g, mientras que c_0 es la constante multiplicativa (v. fig. 2.11); n_0 permite olvidar los primeros valores de la función por poco significativos, y c_0 desprecia los cambios constantes dentro de un mismo orden debidos a variaciones en los factores de bajo nivel y también a la aparición reiterada en los algoritmos de construcciones que tienen la misma eficiencia.

Por ejemplo, dadas las funciones $f(n) = 4n$, $g(n) = n+5$ y $h(n) = n^2$, se puede comprobar aplicando la definición que f y g están dentro de O(f), O(g) y O(h) y también dentro de $\Omega(f)$ y $\Omega(g)$ (en concreto, $O(f) = O(g)$ y $\Omega(f) = \Omega(g)$, porque ambas funciones presentan el mismo ritmo de crecimiento despreciando la constante multiplicativa), mientras que h no está dentro de O(f) ni de O(g), pero sí dentro de $\Omega(f)$ y $\Omega(g)$. A partir de la definición, y como ilustra este ejemplo, es obvio que las dos notaciones están fuertemente relacionadas según la fórmula $f \in O(g) \Leftrightarrow g \in \Omega(f)$, cuya demostración queda como ejercicio para el lector.

Por lo que respecta a su utilidad como herramienta de medida de la eficiencia de los programas, las notaciones O y Ω presentan varias propiedades especialmente interesantes que se enumeran a continuación; su demostración queda como ejercicio para el lector. Así, el cálculo de la eficiencia de un algoritmo se basará en la aplicación de estas propiedades y no directamente en la definición, facilitando el proceso. Para abreviar se dan solamente las propiedades de O; las de Ω son idénticas.

1) La notación O es una relación reflexiva y transitiva, pero no simétrica:
$$\forall f,g,h: \mathbb{N}^+ \to \mathbb{N}^+: f \in O(f) \wedge \neg (f \in O(g) \Rightarrow g \in O(f)) \wedge (f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h))$$

2) La notación O es resistente a las multiplicaciones por, y adiciones de, constantes:
$$\forall f,g: \mathbb{N}^+ \to \mathbb{N}^+: \forall c \in \mathbb{R}^+: \{ g \in O(f) \Leftrightarrow c.g \in O(f) \} \wedge \{ g \in O(f) \Leftrightarrow c + g \in O(f) \}$$

3) Reglas de la notación O respecto a la suma y el producto de funciones:
$$\forall f_1,g_1,f_2,g_2: \mathbb{N}^+ \to \mathbb{N}^+: g_1 \in O(f_1) \wedge g_2 \in O(f_2):$$
3.a) $g_1+g_2 \in O(\text{máx}(f_1, f_2))$, siendo $(g_1+g_2)(n) \cong g_1(n)+g_2(n)$, y
$\text{máx}(f_1, f_2)(n) \cong \text{máx}\{ f_1(n), f_2(n) \}$
3.b) $g_1.g_2 \in O(f_1.f_2))$, siendo $(g_1.g_2)(n) \cong g_1(n).g_2(n)$

Otra manera de enunciar estas reglas es:
3.a) $O(f_1)+O(f_2) = O(f_1 + f_2) = O(\text{máx}(f_1, f_2))$, y
3.b) $O(f_1).O(f_2) = O(f_1.f_2))$

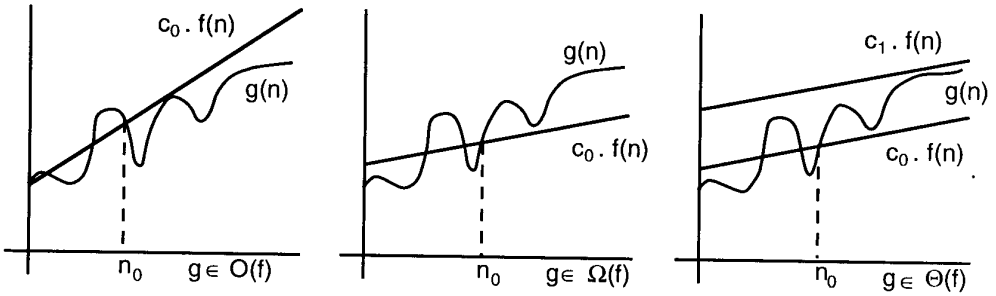
La caracterización de los conjuntos $O(f_1)+O(f_2)$ y $O(f_1).O(f_2)$ aparece en [Peñ93, p.8].

[5] Hay una definición diferente de la notación Ω (v. ejercicio 2.6) que amplía el conjunto de funciones que pertenecen a $\Omega(f)$; no obstante, estas funciones "especiales" no surgen demasiado a menudo en el análisis de la eficiencia de un programa y, por este motivo y también por su simplicidad, preferimos la definición aquí dada.

Notemos que la definición de las notaciones asintóticas es independiente de si se trata el caso peor o el caso medio; en el caso peor, no obstante, se produce una asimetría entre ellas: si $f \in O(g)$, entonces g es una cota superior del caso peor y por ello no puede haber ningún valor de la entrada que se comporte peor que g, mientras que si $f \in \Omega(g)$, entonces g es una cota inferior del caso peor, pero puede haber diversos valores de la entrada, potencialmente infinitos, que se comporten mejor que g, si el algoritmo trabaja en un caso que no sea el peor.

Las dos notaciones hasta ahora presentadas permiten buscar cotas superiores e inferiores de programas; ahora bien, la formulación de una de estas cotas no proporciona suficiente información como para determinar la eficiencia exacta de un programa. Por ello, introducimos una última notación que será la más usada en este texto, llamada teta grande (ing., big Theta) o, simplemente, *Θ*. La teta de f, notada mediante *Θ*(f), es el conjunto de funciones que crecen exactamente al mismo ritmo que f; dicho de otra manera, $f \in \Theta(g)$ si y sólo si g es a la vez cota inferior y superior de f; este conjunto es formulable a partir de O(f) y Ω(f) como $\Theta(f) = O(f) \cap \Omega(f)$ (v. fig. 2.11) y así la definición queda:

$$\Theta(f) \cong \{g: N^+ \to N^+ / \exists c_0, c_1: c_0, c_1 \in R^+: \exists n_0: n_0 \in N^+: \forall n: n \geq n_0: c_1 f(n) \geq g(n) \geq c_0 f(n)\}$$



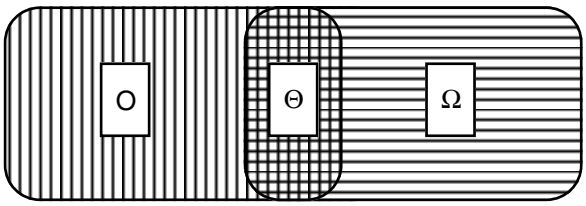


Fig. 2.11: notaciones O, *Ω* y *Θ*.

En el ejemplo anterior se cumple que $f \in \Theta(g)$, ya que $f \in O(g)$ y $f \in \Omega(g)$ y, simétricamente, $g \in \Theta(f)$, dado que $g \in O(f)$ y $g \in \Omega(f)$. Precisamente, la simetría de la notación Θ es un rasgo diferencial respecto a O y Ω, porque determina que Θ es una relación de equivalencia; el resto de propiedades antes enunciadas son, no obstante, idénticas. Destaca también una

caracterización de Θ por límites: $\lim_{n\to\infty} f(n)/g(n) \in R - \{0\} \Rightarrow f \in \Theta(g)$. Por lo que respecta a la nomenclatura, dadas dos funciones $f,g: N^+ \to N^+$ tales que $f \in \Theta(g)$, decimos que f es de orden (de magnitud) g o, simplemente, que f es g (siempre en el contexto del estudio de la eficiencia, claro); también se dice que f es de (o tiene) coste (o complejidad; ing., cost o complexity) g [6].

Hasta ahora, las notaciones asintóticas se han definido para funciones de una sola variable. No obstante, en el caso general, la eficiencia de un algoritmo puede depender de diferentes magnitudes, cada una de las cuáles representa el tamaño de un dominio de datos determinado (por ejemplo, en una biblioteca, número de libros, de socios, de ejemplares por libro, etc.). La definición de O, Ω y Θ en esta nueva situación introduce tantas constantes como variables tenga la función para determinar los valores a partir de los que la cota dada es válida. Por ejemplo, la definición de Θ para una función f de k variables queda:

$$\Theta(f)^{7} \cong \{g: N^+ \times \ldots \times N^+ \to N^+ / \exists c_0, c_1 \in R^+: c_0, c_1 \in R^+: \exists x_1, \ldots, x_k: x_1, \ldots, x_k \in N^+:$$
$$\forall n_1, \ldots, n_k: n_1 \geq x_1 \wedge \ldots \wedge n_k \geq x_k: c_1 f(n_1, \ldots, n_k) \geq g(n_1, \ldots, n_k) \geq c_0 f(n_1, \ldots, n_k)\}$$

La existencia de diferentes variables genera órdenes de magnitud incomparables cuando no es posible relacionarlas completamente; así, $\Theta(a^2.n) \notin \Theta(a.n^2)$ y $\Theta(a.n^2) \notin \Theta(a^2.n)$. Esta situación dificulta el análisis de la eficiencia y provoca que algunos resultados queden pendientes del contexto de utilización del algoritmo, que será el que determinará todas las relaciones entre los diferentes parámetros de la eficiencia; concretamente, para asegurar que todos los órdenes se puedan comparar, esta relación debería ser un orden total que permitiera expresar todos los parámetros en función de uno solo. Así mismo, la regla de la suma de las notaciones asintóticas se ve afectada porque puede ser imposible determinar el máximo de dos funciones tal como exige su definición.

Por último, es conveniente constatar que el análisis asintótico de la eficiencia de los programas a veces esconde e, incluso, distorsiona otros hechos igualmente importantes, y puede conducir a conclusiones erróneas. Destacamos los puntos siguientes:

- Como ya se ha dicho, el comportamiento asintótico se manifiesta a medida que crece la cantidad de datos de entrada. Eventualmente, el contexto de utilización de un programa puede asegurar que el volumen de datos es siempre reducido, y entonces el análisis asintótico es engañoso: una función asintóticamente costosa puede llegar a ser más rápida que otra de orden inferior si las diversas constantes aditivas y, sobre todo, multiplicativas, olvidadas en el proceso de cálculo de la eficiencia, son mucho mayores en la segunda que en la primera. Por ejemplo, hay diversos algoritmos de productos de enteros o matrices que son asintóticamente más eficientes que los tradicionales, pero

[6] Algunos autores proponen la terminologia "orden exacto"; en el caso de que $f \in O(g)$, dicen que f tiene "como máximo" orden g, y si $f \in \Omega(g)$ dicen que f tiene "como mínimo" orden g.

[7] En este caso, y por motivos de claridad, puede ser conveniente explicitar las variables en la definición misma, y así escribir $\Theta(f(n_1, \ldots, n_k))$.

que no se empiezan a comportar mejor hasta dimensiones de los datos demasiado grandes para que sean realmente prácticos. En general, no obstante, las constantes que pueden surgir en el análisis de la eficiencia tienen normalmente un valor pequeño.

- En algunos contextos es necesario un análisis más depurado del comportamiento de un algoritmo si realmente se quieren extraer conclusiones pertinentes. Por ejemplo, en la familia de los algoritmos de ordenación (ing., sorting algorithm) hay varios con el mismo coste asintótico y, por ello, también se consideran las constantes y se estudia en qué situaciones son más apropiados unos u otros.

- Una versión muy eficiente de un programa puede dar como resultado un trozo de código excesivamente complicado y rígido que vulnere los principios de la programación modular enumerados en la sección 1.1, especialmente en lo que se refiere a su mantenimiento y reusabilidad.

- El estudio asintótico muestra claramente que el ahorro de una variable o instrucción que complique el código resultante no aporta ninguna ganancia considerable y reduce la legibilidad; por tanto, dicho ahorro es claramente desaconsejable. Sin embargo, no es justificable la ineficiencia causada por la programación descuidada, por mucho que no afecte al coste asintótico del programa; un ejemplo habitual es el cálculo reiterado de un valor que se podría haber guardado en una variable, si dicho cálculo tiene un coste apreciable.

- Si la vida de un programa se prevé breve priman otros factores, siendo especialmente importante el coste de desarrollo y depuración. En el mismo sentido, si se tiene la certeza de que un trozo de programa se usará sólo en situaciones excepcionales sin que sea necesario que cumpla requisitos de eficiencia exigentes, el estudio de la eficiencia del programa puede prescindir de él y centrarse en las partes realmente determinantes en el tiempo de ejecución. Cualquiera de estas dos suposiciones, evidentemente, ha de estar realmente fundada.

- Otros factores, como la precisión y la estabilidad en los algoritmos de cálculo numérico, o bien el número de procesadores en algoritmos paralelos, puede ser tan importantes (o más) como la eficiencia en el análisis de los programas.

2.3.2 Órdenes de magnitud más habituales

Toda función f: $N^+ \to N^+$ cae dentro de una de las clases de equivalencia determinadas por la relación Θ. Para poder hablar con claridad de la eficiencia de los programas se identifican a continuación las clases más usuales que surgen de su análisis; todas ellas se caracterizan mediante un representante lo más simple posible, se les da un nombre para referirse a ellas en el resto del libro y se muestran algunas funciones que son congruentes.

- $\Theta(1)$: constante. Incluye todas aquellas funciones independientes del volumen de los datos que se comportan siempre de la misma manera; por ejemplo, secuencias de instrucciones sin iteraciones ni llamadas a funciones o acciones, y tuplas de campos no estructurados. Dentro de estas clases se encuentran las funciones 4, 8+log2, etc.
- $\Theta(\log n)$: logarítmico[8]. Aparecerá en algoritmos que descarten muchos valores en un único paso (generalmente, la mitad) durante un proceso de búsqueda; las estructuras arborescentes del capítulo 5 son un ejemplo paradigmático. Se incluyen funciones como $\log n+k$, $\log n+k \log n$, etc., siendo k una constante no negativa cualquiera.
- $\Theta(n^k)$, para k constante: polinómico; serán habituales los casos $k = 1$, 2 y 3 llamados, respectivamente, lineal, cuadrático y cúbico. Notemos que el caso $k = 0$ es el coste constante. El análisis de una función o estructura de datos en la que haya vectores y la aparición de bucles comporta normalmente un factor polinómico dentro del coste. Son funciones congruentes $n^k+\log n$, n^k+7, n^k+n^{k-1}, etc.; consultar también el ejercicio 2.5.
- $\Theta(n \log n)$ y $\Theta(n^2 \log n)$: casi-lineal y casi-cuadrático. Generalmente aparecen en bucles tales que cada paso comporta un coste logarítmico o casi-lineal, respectivamente.
- $\Theta(k^n)$, para k constante: exponencial; es el caso peor de todos, asociado generalmente a problemas que se han de resolver mediante el ensayo reiterado de soluciones.

En la fig. 2.12 se muestra la evaluación de estas funciones para algunos valores representativos de n. Notemos la similitud de comportamiento de los casos constante y logarítmico, lineal y casi-lineal y cuadrático y casi-cuadrático, que en memoria interna se pueden considerar prácticamente idénticos, sobre todo teniendo en cuenta que en el cálculo de los costes se desprecian constantes y factores aditivos más pequeños, que pueden llegar a hacer un algoritmo asintóticamente logarítmico mejor que otro constante, como ya se ha dicho en el punto anterior.

n	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n^2 \log n)$	$\Theta(n^3)$	$\Theta(2^n)$
1	1	1[9]	1	1	1	1	1	2
10	1	2.3	10	23	100	230	1000	1024
100	1	4.6	100	460	10000	46000	10^6	1.26×10^{30}
1000	1	6.9	1000	6900	10^6	6.9×10^6	10^9	$\rightarrow \infty$
10000	1	9.2	10000	92000	10^8	9.2×10^8	10^{12}	$\rightarrow \infty$
100000	1	11.5	10^5	11.5×10^5	10^{10}	11.5×10^{10}	10^{15}	$\rightarrow \infty$

Fig. 2.12: algunos valores de los costes asintóticos más frecuentes.

[8] No importa la base (que habitualmente será 2), ya que un cambio de base se concreta en una multiplicación por una constante (v. ejercicio 2.5).

[9] El tiempo de ejecución de un programa no se puede considerar nunca 0.

En caso que la eficiencia dependa de más de un parámetro, será necesario explicitar éstos al catalogar la función; frecuentemente, para evitar ambigüedades, en vez de usar los nombres que se acaban de dar, leeremos directamente la función representante de la clase; así, un algoritmo que tenga un coste $2a^2.3\log n$ lo describiremos como "a cuadrado por el logaritmo de n".

2.3.3 Análisis asintótico de la eficiencia temporal

Una vez formuladas las diferentes notaciones asintóticas de interés e identificadas sus propiedades más relevantes, es indispensable determinar cómo se puede deducir la eficiencia de una función o representación a partir del algoritmo o la estructura de datos correspondiente. Estudiamos en este apartado la eficiencia temporal y en el siguiente la eficiencia espacial.

El cálculo de la eficiencia temporal de un programa se puede realizar aplicando las reglas que se enumeran a continuación, donde se muestra el tipo de construcción del lenguaje y también las construcciones Merlí concretas para así poder analizar más fácilmente los programas que aparecen en este texto completamente codificados. Destacamos que la mayoría de las reglas dan como resultado sumatorios que, aplicando las propiedades de las notaciones asintóticas, son equivalentes al orden de magnitud del sumando más costoso. Denotaremos por T(S) la eficiencia temporal del trozo S de código; a veces, cuando el trozo sea una función o acción entera f, denotaremos su eficiencia por T_f [10].

- Referencia a valores y operadores predefinidos: se toma como coste constante cualquier referencia a un objeto así como la aplicación de operaciones aritméticas, relacionales y lógicas sobre los tipos que lo permitan. También se consideran constantes las indexaciones de vectores (sin incluir el cálculo del índice mismo, que tendrá su propio coste) y las referencias a tuplas. Los vectores y las tuplas podrán compararse con = y $\neq$; en el caso de los vectores, el coste de la comparación es el producto de la dimensión del vector por el coste de comparar un elemento; en el caso de las tuplas, el resultado es igual al máximo del coste de las comparaciones de los campos individuales.
- Evaluación de una expresión: suma de los costes de las subexpresiones que la componen y de los operadores que las combinan:

  $$T(E_1\, op_1\, E_2\, op_2 \ldots E_n\, op_n\, E_{n+1}) \cong [\,\Sigma\, i: 1 \le i \le n+1: T(E_i)\,] + [\,\Sigma\, i: 1 \le i \le n: T(op_i)\,]$$

  Los costes $T(op_i)$ y $T(E_i)$ vienen dados por la regla anterior y, en el caso de que aparezcan invocaciones a funciones, por la regla siguiente.

[10] También se escribe $T_f(n_1, \ldots, n_k)$, siendo los n_i los diversos parámetros que determinan el coste.

- Invocación a una función: suma del tiempo empleado en evaluar y, posiblemente, duplicar los diversos parámetros reales más el tiempo empleado en ejecutar la función; si alguno de los parámetros es una expresión, se debe aplicar la regla anterior.

  $T(f(E_1, ..., E_n)) \cong [\ \Sigma\ i: 1 \le i \le n: T(E_i) + T(\text{duplicación } E_i)\ ] + T_f$

  Por mucho que los parámetros de la función sean implícitamente parámetros sólo de entrada, se considera que su paso sólo exige hacer una copia cuando la función los modifica; es decir, las categorías de los parámetros se consideran exclusivamente como información relativa a su uso, al contrario que los lenguajes imperativos tradicionales, que las tratan también como información orientada a la generación de código. Como resultado, el tiempo de duplicación de un parámetro será constante en muchos casos. No consideramos el posible tiempo de duplicar el resultado, ya que supondremos que éste será disponible implícitamente en el ámbito de la llamada.

  De momento, se puede considerar el tiempo de duplicación igual al espacio asintótico que ocupa el tipo correspondiente, calculado según las reglas que se dan en el apartado siguiente. En el apartado 3.3.4 introduciremos en los TAD (por otros motivos) una función de asignación de valores y veremos entonces que la duplicación de los parámetros consistirá en llamar a esta función, que tendrá su propio coste.

  Por otro lado, no consideramos la formulación de ecuaciones de recurrencia típicas del caso de las funciones recursivas (por ejemplo, consultar [Peñ93]) dado que, en este texto, el cálculo de la eficiencia de este tipo de funciones se hará usando otros razonamientos.

- Composición secuencial de instrucciones: suma de cada una de las instrucciones que componen la secuencia.

  $T(S_1; ...; S_n) \cong \Sigma\ i: 1 \le i \le n: T(S_i)$

- Asignación: suma de los tiempos de evaluación de la parte izquierda y de la parte derecha. Una vez más, no obstante, la asignación puede ser de vectores o tuplas completos, en cuyo caso ni la parte izquierda ni la parte derecha presentarán expresiones. En la asignación de vectores se multiplica el coste de la asignación a un componente por la dimensión, y en la de tuplas, al ser el coste la suma, domina la asignación de campos más costosa.

  $T(E_1 := E_2) \cong T(E_1) + T(E_2)$, si el tipo es predefinido o escalar

  $T(v_1 := v_2) \cong n.T(v_1[i] := v_2[i])$, para vectores de n componentes

  $T(v_1 := v_2) \cong \Sigma\ i: 1 \le i \le n: T(v_1.c_i := v_2.c_i)$, para tuplas de campos $c_1, ..., c_n$

- Sentencias alternativas: suma de las diferentes expresiones condicionales e instrucciones asociadas. A causa del cálculo del caso peor, se examinan todas las ramas posibles y predomina el coste de la condición o la expresión más costosa.

$T(\underline{\text{si}}\ E\ \underline{\text{entonces}}\ S_1\ \underline{\text{sino}}\ S_2\ \underline{\text{fsi}}) \cong T(E) + T(S_1) + T(S_2)$[11]

$T(\underline{\text{opción}}\ \underline{\text{caso}}\ E_1\ \underline{\text{hacer}}\ S_1 \ldots \underline{\text{caso}}\ E_n\ \underline{\text{hacer}}\ S_n\ \underline{\text{en cualquier otro caso}}\ S_{n+1}\ \underline{\text{fopción}})$

$\cong [\ \Sigma\, i: 1 \le i \le n+1: T(S_i)\ ] + [\ \Sigma\, i: 1 \le i \le n: T(E_i)\ ]$

- Sentencias iterativas: suma de la condición del bucle y de las instrucciones que forman el cuerpo, multiplicadas por el número n de veces que se ejecuta el bucle; no obstante, hay diversas modalidades de bucle que sólo evalúan la condición una única vez, al inicio.

  $T(\underline{\text{hacer}}\ E\ \underline{\text{veces}}\ S\ \underline{\text{fhacer}}) \cong T(E) + (n.T(S))$

  $T(\underline{\text{mientras}}\ E\ \underline{\text{hacer}}\ S\ \underline{\text{fmientras}})$, $T(\underline{\text{repetir}}\ S\ \underline{\text{hasta que}}\ E) \cong n.(T(E) + T(S))$

  $T(\underline{\text{para todo}}\ v\ \underline{\text{desde}}\ E_1\ \underline{\text{hasta}}\ E_2\ \underline{\text{hacer}}\ S\ \underline{\text{fpara todo}}) \cong T(E_1) + T(E_2) + (n.T(S))$

  $T(\underline{\text{para todo}}\ v\ \underline{\text{dentro de}}\ T\ \underline{\text{hacer}}\ S\ \underline{\text{fpara todo}}) \cong T(E) + (n.T(S))$

  En los dos últimos casos, la referencia a v no afecta al coste porque la variable ha de ser de tipo natural, entero o escalar y su asignación, por tanto, constante.

  Hay que destacar que, a veces, n es una magnitud desconocida y entonces se asume el valor más grande que puede tomar (hipótesis de caso peor), a pesar de que también se puede conjeturar un valor fijo si se pretende comparar diversos algoritmos más que formular una eficiencia concreta. También es frecuente que el coste de ejecución de un bucle no sea siempre el mismo; si se dan los dos casos, se puede multiplicar el número máximo de vueltas por el coste más grande de un bucle y se obtiene una cota superior O. Por último, destacar que en ciertos contextos el análisis de un bucle puede no seguir las reglas semiautomáticas dadas aquí, sino que se emplea una estrategia diferente que exige un análisis más profundo del significado del código; así lo haremos en este mismo capítulo al analizar el algoritmo de ordenación por el método de la inserción, y también en el capítulo 6 al estudiar diversos algoritmos sobre grafos.

- Invocación a una acción: por los mismos razonamientos que la invocación a funciones:

  $T(P(E_1, \ldots, E_n)) \cong [\ \Sigma\, i: 1 \le i \le n: T(E_i) + T(\text{duplicación } E_i)] + T_P$

  Como en las funciones, sólo se duplicarán los parámetros de entrada que se modifiquen.

- Retorno de valor: simplemente el tiempo de evaluar la expresión: $T(\underline{\text{devuelve}}\ E) \cong T(E)$.

- Errores: supondremos que el tratamiento de errores es de tiempo constante.

[11] Para ser más exactos, el coste del algoritmo se puede precisar en términos de las notaciones O y Ω, y obtener $O(\text{máx}(T(E), T(S_1), T(S_2))$ como cota superior y $\Omega(\text{máx}(T(E), \text{mín}(T(S_1), T(S_2)))$ como cota inferior. Si ambos valores son iguales, se puede afirmar efectivamente que Θ tiene el coste dado; si no lo son, realmente se está tomando como coste la cota superior del caso peor, dentro del contexto pesimista en que nos movemos. Se pueden aplicar razonamientos similares en la sentencia alternativa múltiple.

2.3.4 Análisis asintótico de la eficiencia espacial

También el espacio que ocupa una estructura de datos se puede calcular automáticamente a partir de unas reglas formuladas sobre los diferentes constructores de tipo del lenguaje, tomando como coste $\Theta(1)$ el espacio que ocupa un objeto de tipo predefinido o escalar. Concretamente, si T es un tipo de datos representado en Merlí dentro de un universo de implementación, el espacio $E(T)$ que ocupa un objeto de tipo T se puede calcular según las reglas:

- Vectores: producto del espacio que ocupa cada componente por la dimensión n.

  $E(T) \cong E(\underline{\text{vector}}\ [\text{índice}]\ \underline{\text{de}}\ t) = n.E(t)$

- Tuplas: suma del espacio de los campos; las tuplas variantes no afectan al coste asintótico.

  $E(T) \cong E(\underline{\text{tupla}}\ c_1\ \underline{\text{es}}\ t_1; \ldots; c_n\ \underline{\text{es}}\ t_n\ \underline{\text{ftupla}}) = \Sigma\ i: 1 \leq i \leq n: E(t_i)$

Sin embargo, a menudo, cuando se estudia el espacio de una estructura de datos, no sólo interesa el coste asintótico sino también su coste "real" sin despreciar las constantes multiplicativas ni los factores aditivos de orden inferior. La razón es que no será habitual que dos estrategias de representación de un tipo de datos sean tan radicalmente diferentes como para resultar en costes asintóticos desiguales, sino que las diferencias estarán en el número de campos adicionales empleados para su gestión eficiente; estas variaciones no influyen en el coste asintótico, pero pueden ser suficientemente significativas como para considerarlas importantes en la elección de la representación concreta. Para el cálculo correcto de este coste "real" se debe determinar el espacio que ocupa un objeto de cada uno de los tipos predefinidos, un objeto de tipo escalar y un objeto de tipo puntero (v. apartado 3.3.3), y aplicar las mismas reglas en los vectores y en las tuplas, poniendo especial atención en el caso de las tuplas variantes: dada una parte variante de n conjuntos de campos $\{c_1\}, \ldots, \{c_n\}$, el espacio que ocupa es la suma del campo discriminante más el máximo del espacio de los $\{c_i\}$. El espacio se puede dar en términos absolutos o bien relativos, eligiendo el espacio de alguno de los tipos como unidad y expresando el resto en función suya.

Por último, cabe destacar que el estudio de la eficiencia espacial se aplica, no sólo a las representaciones de estructuras de datos, sino también a las variables auxiliares que declara una función o acción. Es decir, se considera que toda función o acción, aparte de sus parámetros (que necesitarán un espacio independiente del algoritmo usado dentro de la función o acción), utiliza un espacio adicional que puede depender de la estrategia concreta y que, eventualmente, puede llegar a cotas lineales o cuadráticas. En este libro, el espacio adicional se destacará cuando sea especialmente voluminoso o cuando sea un criterio de elección u optimización de una operación.

Para ilustrar las reglas definidas en estos dos últimos puntos, se determina la eficiencia de la implementación de los conjuntos de la fig. 2.6. La eficiencia espacial es muy sencilla de establecer:

- Asintótica: la representación de los conjuntos es una tupla y, por ello, el espacio E(cjt) queda E(cjt) = val.E(elem) + $\Theta(1) = \Theta(val.n)$, siendo E(elem) $\cong \Theta(n)$ y $\Theta(1)$ el coste del campo sl.
- Real: sea x el espacio que ocupa un natural y sea n el espacio que ocupa un objeto de tipo elem, el espacio resultante será val.n + x, muy parecido en este caso al espacio asintótico.

Es decir, el coste espacial depende de lo que ocupe un objeto de tipo elem; ésta será la situación habitual de los tipos parametrizados. Por otro lado, notemos que todas las funciones son lo más óptimas posible dado que sólo requieren un espacio adicional constante, incluso la operación auxiliar[12]. Por último, destaquemos que la eficiencia espacial de un conjunto es independiente del número de elementos que realmente contiene.

En lo que respecta al coste temporal, empezamos por calcular la eficiencia de índice, que sirve de base para el resto. El cuerpo de la función es una secuencia de cuatro instrucciones (dos asignaciones, un bucle y el retorno) y el coste total será su suma. Las asignaciones y la instrucción de retorno tienen un coste obviamente constante, porque son valores o referencias a variables de tipos predefinidos. Por lo que se refiere al bucle, el coste se calcula de la manera siguiente:

- La evaluación de la condición de entrada queda constante, porque consiste en aplicar una operación booleana (por definición, $\Theta(1)$) sobre dos operandos, siendo cada uno de ellos una operación de coste constante aplicada sobre unos operandos, que son valores o bien referencias a variables de tipos predefinidos.
- El cuerpo consiste en una sentencia condicional y su eficiencia es la suma de tres factores: la evaluación de la condición y dos asignaciones de coste constante igual que las examinadas anteriormente. El coste de la evaluación es igual al coste $T_{=}$ de la operación de igualdad, que es desconocido por depender del parámetro formal.
- El número de iteraciones es desconocido a priori. En el caso peor, el elemento que se busca no está dentro de la parte ocupada del vector y entonces el número de iteraciones para un conjunto de k elementos es igual a k.

Considerando todos estos factores, el coste total del bucle para un conjunto de k elementos es $k.(\Theta(1)+T_{=}) = \Theta(k.T_{=})$. El coste total de índice es, pues, $\Theta(1)+\Theta(1)+\Theta(k.T_{=})+\Theta(1) = \Theta(k.T_{=})$. El coste del resto de funciones se calcula aplicando razonamientos parecidos y quedan $\Theta(k.T_{=})$ igualmente, excepto crea, que no invoca índice y es constante.

[12] Por mucho que una función no declare ninguna variable, se considera un coste constante y no $\Theta(0)$ por factores de bajo nivel (para guardar la dirección de retorno, por ejemplo).

Para cerrar el estudio de la eficiencia asintótica de las implementaciones, consideremos un algoritmo algo más complicado, como es el algoritmo de ordenación de un vector A, usando el método de inserción (ing., insertion sorting). El método de inserción (v. fig. 2.13) sitúa los elementos en su lugar uno a uno, empezando por la posición 1 hasta el máximo: al inicio del paso i-ésimo del bucle principal, las posiciones entre la A[1] y la A[i-1] contienen las mismas posiciones que al empezar, pero ordenadas (como establece el invariante) y entonces el elemento A[i] se intercambia reiteradamente con elementos de este trozo ordenado hasta situarlo en su lugar. Para facilitar la codificación, se supone la existencia de una posición 0 adicional, que se emplea para almacenar el elemento más pequeño posible del tipo (valor que sería un parámetro formal del universo) y que simplifica el algoritmo; así mismo, supondremos que los elementos presentan dos operaciones de comparación, _<_ y _≤_, con el significado habitual.

```
{Acción ordena_inserción: dado un vector A, lo ordena por el método de la inserción.
        P ≅ A = A0 (se fija un valor de inicio para referirse a él en Q )
        Q ≅ elems(A, 1, máx) = elems(A0, 1, máx) ∧ ordenado(A, 1, máx), donde
            elems(A, i, j) ≅ {A[i], A[i+1], ..., A[j]}, como multiconjunto[13], y
            ordenado(A, i, j) ≅ ∀k: i ≤ k ≤ j-1: A[k] ≤ A[k+1] }
acción ordena_inserción (ent/sort A es vector [de 0 a máx] de elem) es
var i, j son nat; temp es elem fvar
    A[0] := el_menor_de_todos
    para todo i desde 2 hasta máx hacer
            { I ≅ elems(A, 1, i-1) = elems(A0, 1, i-1) ∧ ordenado(A, 1, i-1) ∧
                  ∧ ∀k: 1 ≤ k ≤ máx: A[k] > A[0]
              F ≅ máx-i }
        j := i
        mientras A[j] < A[j-1] hacer
                { I ≅ 1 ≤ j ≤ i ∧ ordenado(A, j, i) ∧ ∀k: 1 ≤ k ≤ máx: A[k] > A[0]. F ≅ j }
            temp := A[j]; A[j] := A[j-1]; A[j-1] := temp; j := j-1
        fmientras
    fpara todo
facción
```

Fig. 2.13: algoritmo de ordenación de un vector por el método de inserción.

Estudiemos el coste temporal del algoritmo (obviamente, el coste espacial adicional es $\Theta(1)$), que abreviamos por T_{ord}. El coste es igual a la suma del coste de la primera asignación y del bucle; dado que dentro del bucle aparecen más asignaciones del mismo tipo, se puede decir simplemente que T_{ord} es igual al coste del bucle principal, que se ejecuta máx-1 veces.

[13] Un multiconjunto es un conjunto que conserva las repeticiones. Así, los multiconjuntos {1} y {1,1} son diferentes.

Dentro de este bucle se encuentran las expresiones que determinan los valores que toma la variable de control, una asignación de naturales (ambas construcciones de coste constante) y un segundo bucle que se ejecuta un número indeterminado de veces, que es función del valor i y de la suerte que tengamos. Precisamente es esta indeterminación lo que dificulta el análisis de la eficiencia del algoritmo y conduce a un cambio de enfoque: en lugar de aplicar las reglas de cálculo de la eficiencia dadas, se calcula el número N de veces que se ejecutan las instrucciones del cuerpo del bucle interno a lo largo de todo el algoritmo, se suma el resultado al coste Θ(máx) de ejecutar la asignación j := i exactamente máx-1 veces y así se obtiene el coste total del bucle principal.

El valor N es la resolución de dos sumatorios imbricados; el más interno surge de la suposición del caso peor que lleva a i iteraciones del bucle interno:

$$N = \sum_{i=2}^{máx} \sum_{j=i}^{1} 1 = \sum_{i=2}^{máx} i = \frac{máx(máx+1)}{2} - 1$$

Ahora se puede determinar el coste total del bucle principal que queda parametrizado por los costes de comparar y de asignar elementos, $T_<$ y T_{as}, respectivamente, y da como resultado:

$$\Theta(\{\ máx(máx+1)/2 - 1\ \}.T_<.T_{as}) = \Theta(máx^2.T_<.T_{as}),$$

cuadrático sobre el número de elementos. No es un coste demasiado bueno, dado que hay algoritmos de ordenación de coste casi-lineal (en el capítulo 5 se presenta uno, conocido como heapsort).

Precisamente, el desconocimiento del coste T_{as} esconde una posible ineficiencia del algoritmo dado en la fig. 2.13, porque la colocación del elemento A[i] exige tres asignaciones a cada paso del bucle más interno para implementar el intercambio. En lugar de intercambiar físicamente los elementos se puede simplemente guardar A[i] en una variable auxiliar temp, hacer sitio desplazando elementos dentro del vector y copiar temp directamente en la posición que le corresponda. El resultado es una nueva versión asintóticamente equivalente a la anterior, pero que simplifica las constantes multiplicativas con poco esfuerzo y sin complicar el código, y que por tanto es preferible a la anterior.

2.3.5 Eficiencia y modularidad

La técnica de diseño modular de programas empleada en este texto presenta diversas ventajas ya comentadas en la sección 1.1, pero puede dar como resultado aplicaciones ineficientes precisamente a causa del principio de transparencia de la implementación, que impide la manipulación de la representación de un tipo desde fuera de sus universos de implementación. Lo comprobaremos con un ejemplo.

Sea la especificación habitual de los conjuntos y sea la implementación vista en el apartado anterior; dado que los conjuntos son un TAD que aparece en casi toda aplicación, parece una decisión acertada incluirlos en una biblioteca de módulos reusables que contenga tipos y algoritmos de interés general. Notemos, no obstante, que el estado actual del tipo incluye muy pocas operaciones, y para que sea realmente útil se deben añadir algunas nuevas; como mínimo, se necesita alguna operación para sacar elementos y seguramente la unión, la intersección y similares; en concreto, supongamos que hay una operación sacar_uno_cualquiera que, dado un conjunto, selecciona y borra un elemento cualquiera. Ahora, sea U un universo que precisa una nueva operación, cuántos, que cuenta el número de elementos que hay dentro de un conjunto y que no aparece en la especificación residente en la biblioteca. Usando los mecanismos de estructuración de especificaciones, se puede definir un enriquecimiento del universo de especificación de los conjuntos con esta nueva operación, ya sea en el mismo módulo U o bien en un universo aparte (v. fig. 2.14); su codificación dentro de un universo de implementación es inmediata, altamente fiable e independiente de la representación escogida, pero presenta un problema evidente: para un conjunto de n elementos, el coste temporal de la operación es $\Theta(n)$, a causa de la imposibilidad de manipular la representación del tipo.

```
universo CJT_∈_ACOTADO_CON_CUÁNTOS(ELEM_=, VAL_NAT) es
    usa CJT_∈_ACOTADO(ELEM_=, VAL_NAT), NAT, BOOL
    ops cuántos: cjt → nat
    ecns cuántos(Ø) = 0...
funiverso

universo IMPL_CJT_∈_ACOTADO_CON_CUÁNTOS(ELEM_=, VAL_NAT) es
    implementa CJT_∈_ACOTADO_CON_CUÁNTOS(ELEM_=, VAL_NAT)
    usa CJT_∈_ACOTADO(ELEM_=, VAL_NAT), NAT, BOOL
    función cuántos (c es conjunto) devuelve nat es
    var cnt es nat; v es elem fvar
        cnt := 0
        mientras ¬vacío?(c) hacer
            <c, v> := sacar_uno_cualquiera(c); cnt := cnt + 1
        fmientras
    devuelve cnt
funiverso
```

Fig. 2.14: especificación (arriba) e implementación (abajo) de un enriquecimiento de los conjuntos.

Notemos que no hay ninguna alternativa que permita reducir el coste de la operación cuántos (como resultado del bucle que forzosamente dará n vueltas) y que, además, la estrategia escogida es la óptima si no se viola el principio de transparencia de la representación. Si se quiere reducir el coste de la operación a orden constante, no queda más remedio que introducir la función cuántos dentro de la especificación básica de los conjuntos (v. fig. 2.15) y adquirir así el derecho de manipular la representación del tipo.

```
universo CJT_∈_ACOTADO(ELEM_=, VAL_NAT) es
    usa NAT, BOOL
    ops
        Ø, _∪{_}, ...
        cuántos: cjt → nat
    ecns
        ... las de Ø, _∪{_}, ...
        cuántos(Ø) = 0 ...
funiverso

universo CJT_∈_ACOTADO_POR_VECT(ELEM_=, VAL_NAT) es
    implementa CJT_∈_ACOTADO(ELEM_=, VAL_NAT)
    tipo cjt es ...{v. fig. 2.6}
    función cuántos (c es conjunto) devuelve nat es
    devuelve c.sl
funiverso
```

Fig. 2.15: especificación e implementación de la operación para contar los elementos de un conjunto como parte del TAD que los define.

Este enfoque, no obstante, presenta diversos inconvenientes:

- Se introduce en la definición de un tipo de interés general (que residirá dentro una biblioteca de universos) una operación que inicialmente no estaba prevista; si esta nueva operación no es utilizada por ninguna otra aplicación, se complica la definición y el uso del tipo sin obtener demasiado provecho a cambio.
- Si el universo ya existía previamente, hay que modificarlo y esto es problemático:
  - ◊ Es necesario implementar la nueva operación en todas las implementaciones existentes para el tipo. En cada una de ellas, si el implementador original (persona o equipo de trabajo) está disponible y puede encargarse de la tarea, ha de recordar el funcionamiento para poder programar la nueva operación (¡y puede hacer meses que se escribió!). En caso contrario, todavía peor, pues alguna otra persona ha de entender y modificar la implementación, siendo esta tarea más o menos complicada

dependiendo de la calidad del código (sobre todo por lo que respecta a la legibilidad); incluso es posible que el código fuente no esté disponible y sea imposible modificarlo.

◊ La sustitución de la versión anterior del universo (residente en una biblioteca de módulos) por la nueva puede provocar una catástrofe si se introduce inadvertidamente algún error; si, por prudencia, se prefiere conservar las dos versiones, se duplicarán módulos en la biblioteca.

Resumiendo, el desarrollo modular de aplicaciones permite construir de manera clara, rápida, fiable y elegante cualquier programa, pero, a veces, entra en conflicto con cuestiones de eficiencia espacial y/o temporal, y la política a seguir entra dentro de los requerimientos de uso de la aplicación y del criterio de los diseñadores y programadores. Notemos también la libertad de definir los tipos con las operaciones que más interesen, puesto que se pueden añadir operaciones para unir e intersectar conjuntos, para obtener el elemento más grande o más pequeño, etc., a discreción. Así, se pueden seguir dos enfoques diferentes al definir un tipo: pensar en los universos de definición como un suministro de las operaciones indispensables para construir otras más complicadas en universos de enriquecimiento, o bien incluir en el universo de definición del tipo todas aquellas operaciones que se puedan necesitar en el futuro, buscando mayor eficiencia en la implementación (en el capítulo 7 se insiste sobre este tema).

Ejercicios

2.1 Escribir una implementación para los conjuntos acotados con pertenencia usando el tipo V de los elementos como índice de un vector A de booleanos, tal que A[v] vale cierto si v está en el conjunto. Suponer que la cardinalidad de V es n y que se dispone de una función inyectiva h : V $\rightarrow$ [1, n]. Escribir la especificación pre-post de las operaciones. Verificar la corrección de la implementación respecto a la especificación usando las estrategias vistas en la sección 2.2.

2.2 a) Escribir una implementación para los polinomios $\mathbb{Z}$[X] especificados en el apartado 1.5.1, de modo que se almacenen en un vector todos los monomios de coeficiente diferente de cero sin requerir ninguna condición adicional. Escribir la especificación pre-post de las diferentes operaciones y también los invariantes y las funciones de acotamiento de los diversos bucles que en ella aparezcan. Verificar la corrección de la implementación respecto a la especificación usando las diferentes estrategias vistas en la sección 2.2. Calcular el coste asintótico de la implementación.

b) Repetir el apartado a) requeriendo que los pares del vector estén ordenados en forma creciente por exponente.

2.3 Sea el TAD de los multiconjuntos (conjuntos que admiten elementos repetidos; es decir, el multiconjunto {x, x} es válido y diferente del multiconjunto {x}) sobre un dominio V con operaciones de crear el multiconjunto vacío y añadir un elemento al multiconjunto. Proponer una representación (que incluya la función de abstracción, el invariante de la representación y la relación de igualdad) para las dos situaciones siguientes:

a) El número total de elementos del multiconjunto está acotado por n.

b) El número total de elementos del multiconjunto no está acotado, pero, en cambio, se sabe que la cardinalidad de V es n y que se dispone de una función inyectiva h: $V \rightarrow [1, n]$.

2.4 Sean las funciones $f_1(n) = n^3$, $f_2(n) = 90000n^2+70000n$, $f_3(n) = n^2\log n$, $f_4(n) = n^3\log n$, $f_5(n) = n^3+\log n$. Estudiar, para todo par de valores diferentes i, j entre 1 y 5, si $f_i \in O(f_j)$, si $f_i \in \Omega(f_j)$ y si $f_i \in \Theta(f_j)$. Representar gráficamente los conjuntos $O(f_j)$, $\Omega(f_j)$ y $\Theta(f_j)$.

2.5 **a)** Sea el polinomio $p(n) = c_k n^k + c_{k-1}n^{k-1} + ... + c_1 n + c_0$, con $\forall i: 0 \leq i \leq k\text{-}1: c_i \geq 0$, $c_k > 0$. Demostrar que $p(n) \in \Theta(n^k)$. **b)** Demostrar que $\Sigma i: 1 \leq i \leq n: i^k \in \Theta(n^{k+1})$, $k > 0$. **c)** Demostrar que $\forall a, b: a > 1 \wedge b > 1: \log_a n \in \Theta(\log_b n)$.

2.6 Las notaciones O y Ω tienen asociadas otras dos, o pequeña (ing., small o) y ω pequeña (ing., small omega), abreviadamente o y ω, que definen cotas significativamente más fuertes que las primeras y que se definen, para $f: N^+ \rightarrow N^+$:

$$o(f) = \{g: N^+ \rightarrow N^+ / \forall c_0: c_0 \in R^+: \exists n_0: n_0 \in N^+: \forall n: n \geq n_0 : g(n) \leq c_0 f(n)\}$$
$$\omega(f) = \{g: N^+ \rightarrow N^+ / \forall c_0: c_0 \in R^+: \exists n_0: n_0 \in N^+: \forall n: n \geq n_0 : g(n) \geq c_0 f(n)\}$$

Estudiar si estas notaciones son reflexivas, simétricas o transitivas, así como sus relaciones con las notaciones O y Ω. Mostrarlas gráficamente con un esquema similar al de la fig. 2.11.

2.7 Dados los siguientes algoritmos, formular su especificación pre-post, establecer los invariantes y las funciones de acotamiento de sus bucles y calcular su coste.

a) Producto de matrices.

```
acción producto (ent A, B son vectores [de 1 a n, de 1 a n] de enteros;
                 sal C es vector [de 1 a n, de 1 a n] de enteros) es
var fil, col, índice son enteros fvar
    para todo fil desde 1 hasta n hacer
        para todo col desde 1 hasta n hacer {cálculo de C[fil, col] }
            C[fil, col] := 0
            para todo índice desde 1 hasta n hacer
                C[fil, col] := C[fil, col] + (A[fil, índice]*B[índice, col])
            fpara todo
        fpara todo
    fpara todo
facción
```

b) Ordenación de un vector por el método de la burbuja.

```
acción burbuja (ent/sal A es vector [de 1 a n] de enteros) es
var i, j, aux son enteros fvar
    para todo i desde 1 hasta n-1 hacer {localización del i-ésimo menor elemento}
        para todo j bajando desde n hasta i+1 hacer
                {examen de la parte del vector todavía no ordenada}
            si A[j-1] > A[j] entonces {intercambio de los elementos}
                aux := A[j]; A[j] := A[j-1]; A[j-1] := aux
            fsi
        fpara todo
        {el i-ésimo menor elemento ya está en A[i] }
    fpara todo
facción
```

c) Búsqueda dicotómica en un vector.

```
función búsqueda_dicot (A es vector [de 1 a n] de enteros; x es entero) devuelve bool es
var izq, der, med son enteros; encontrado es bool fvar
        {se determinan los extremos de la porción del vector donde puede estar x}
    izq := 0; der := n+1; encontrado := falso
        {se busca mientras quede vector por explorar}
    mientras (izq < der-1) ∧ ¬ encontrado hacer
        med := (izq + der) / 2
        opción
            caso x = A[med] hacer encontrado := cierto
            caso x < A[med] hacer der := med
            caso x > A[med] hacer izq := med
        fopción
    fmientras
devuelve encontrado
```

2.8 En el artículo de Vitányi y Meertens citado en el inicio de la sección 2.3 se propone una definición alternativa de la notación Ω:

$$\Omega(f) = \{g: N^+ \to N^+ / \exists c_0: c_0 \in R^+: \forall n_0: n_0 \in N^+: \exists n: n \geq n_0: g(n) \geq c_0 f(n)\}$$

Esta definición sirve también para el caso en que el algoritmo a analizar presente oscilaciones en su comportamiento, porque no exige que g sea una cota superior de f a partir de un cierto punto sino sólo que g sobrepase o iguale f un número infinito de veces. La utilidad de este caso se hace evidente en algoritmos que tienen un coste para algunas entradas y otro diferente para el resto. Estudiar su reflexividad, simetría y transitividad y establecer la relación de esta Ω con la dada en la sección 2.3. Imaginar un algoritmo que tenga un coste diferente según las dos definiciones dadas de Ω.