Tema 2

Estructuras de Datos y Algoritmos

Tipos abstractos de datos

Prof. Dr. P. Javier Herrera

Universidad Francisco de Vitoria
UFV Madrid

Grado en Ingeniería Informática
Escuela Politécnica Superior

Contenido

ESPECIFICACIÓN DE TAD

- Concepto de TAD, terminología y ejemplos
- Especificación algebraica de TAD's
- Construcción de especificaciones

IMPLEMENTACIÓN DE TAD

- Pasos en la implementación de un TAD
- Implementación de conjuntos finitos mediante vectores
- Implementación de los conjuntos finitos con elementos en 1..N

ESPECIFICACIÓN DE TAD

Tipos Abstractos de Datos

- Un **tipo abstracto de datos (TAD)** es un conjunto de *valores* junto con las *operaciones* que sobre él se pueden aplicar, las cuales cumplirán diversas propiedades que determinarán su comportamiento.

- Es necesario utilizar una notación *formal* para **describir el comportamiento de las operaciones**.

- El calificativo "*abstracto*" responde al hecho de que los valores de un tipo pueden ser manipulados **solamente** mediante sus operaciones, conociendo sobre ellas únicamente las propiedades que cumplen, sin que sea necesario ningún conocimiento adicional sobre la representación del tipo o la implementación de dichas operaciones.

Tipos Abstractos de Datos

- La manipulación de los objetos de un tipo solo depende del comportamiento descrito en su *especificación* y es **independiente** de su *implementación*.

Especificación

- La especificación de un TAD consiste en establecer las propiedades que lo definen.
- Una especificación ha de ser **precisa**, **general**, **legible** y **no ambigua**.
- La especificación de un tipo define totalmente su comportamiento a cualquier usuario que lo necesite.

Implementación

- La implementación de un TAD consiste en determinar una representación para los valores del tipo y en codificar sus operaciones a partir de esta representación, todo ello utilizando un lenguaje de programación.
- La implementación ha de ser estructurada, eficiente y legible.
- Una implementación del TAD es totalmente transparente a los usuarios del tipo y no se puede escribir hasta haber determinado claramente su especificación.

Especificación algebraica de TADs

- Entre diversas propuestas que existen para especificar el comportamiento de las operaciones sobre un tipo de datos, vamos a considerar la conocida como ***especificación algebraica*** o ***ecuacional***, que se basa en describir el comportamiento mediante ecuaciones, lo cual facilita el estilo habitual de razonamiento ecuacional, basado en sustituir iguales por iguales.

- Una especificación algebraica consta fundamentalmente de tres componentes:

  - **Tipos**: son nombres de conjuntos de valores. Entre ellos está el *tipo principal* del TAD, aunque puede haber también otros que se relacionen con este.

  - **Operaciones**: son funciones con un perfil asociado que indica el tipo de cada uno de los argumentos y el tipo del resultado. En una especificación algebraica no se permiten funciones que devuelvan varios valores, ni tampoco procedimientos no funcionales.

  - **Ecuaciones**: son igualdades entre términos formados con las operaciones y variables, y definen el comportamiento de las operaciones.

Signatura de un TAD

- Definimos la **signatura** de un TAD como los tipos que utiliza junto con los nombres y perfiles de las operaciones.

- Por ejemplo, para especificar el TAD de los booleanos utilizamos la siguiente signatura:

tipos *bool*

operaciones

cierto	:		→	*bool*
falso	:		→	*bool*
_ ∧ _	:	*bool bool*	→	*bool*
_ ∨ _	:	*bool bool*	→	*bool*
¬ _	:	*bool*	→	*bool*

Signatura de un TAD

- Ejemplo: TAD de los números naturales:

tipos *nat*

operaciones

cero	:		→	*nat*
suc	:	*nat*	→	*nat*
suma	:	*nat nat*	→	*nat*
ig	:	*nat nat*	→	*bool*

Clasificación de las operaciones

- Para escribir las ecuaciones es necesario clasificar las operaciones según el papel que queremos que jueguen en relación con el tipo principal *s*:

  – **Constructoras** (o generadoras): devuelven un valor de tipo *s*. Pensadas para construir todos los valores de tipo *s*. Puede haber más de un subconjunto de operaciones constructoras, del que habrá que elegir uno.

  – **Modificadoras**: devuelven también un valor de tipo *s*. Pero están pensadas para hacer cálculos que produzcan resultados de tipo *s*.

  – **Observadoras**: devuelven un valor de un tipo diferente a *s*. Pensadas para obtener valores de otros tipos a partir de valores de tipo *s*.

- Conjuntos de constructoras para *bool* y *nat*:

  constr1(*bool*) = {cierto, falso} constr2(*bool*) = {cierto, ¬ }

  constr(*nat*) = {cero, suc}

Términos

- Dada la signatura de un TAD y un conjunto de variables *X* con tipo, es posible construir el conjunto (generalmente infinito) de **términos** de tipo *s* mediante la aplicación de las operaciones del TAD. Cada termino representa una aplicación sucesiva de operaciones del TAD, y puede contener variables.

T_{bool} = {cierto, falso, (¬cierto) ∨ falso, ig(cero, suc(cero)), …}

$T_{bool}(X)$ = {cierto, falso, ¬b, ig(n,m), …}

siendo X = {b : *bool*, n : *nat*, m : *nat*, …}

T_{nat} = {cero, suc(cero), suc(suc(cero)), suma(suc(cero), cero), …}

- Un tipo especial de términos son aquellos que sólo contienen operaciones constructoras: son los **términos construidos**. Es necesario que las constructoras permitan generar al menos un término construido distinto para cada posible valor del tipo que se especifica.

TC^1_{bool} = {cierto, falso}

TC^2_{bool} = {cierto, ¬cierto, ¬¬cierto, …}

TC_{nat} = {cero, suc^n(cero)}, n ≥ 1

Ecuaciones

- Son de la forma

  $$t = t'$$

  con t y t' términos con variables $T_s(X)$ para cierto tipo s.

- Las ecuaciones deben **reflejar el comportamiento de las operaciones** para cualquier aplicación correcta de las mismas. Una operación está definida si las ecuaciones determinan su comportamiento respecto a todas las posibles combinaciones de valores que pueden tomar sus parámetros.

  - **Las ecuaciones deben permitir convertir cualquier término en un término construido**: el resultado de la secuencia de operaciones que representa el término.

  - Mediante las ecuaciones ha de ser posible deducir todas las equivalencias que son válidas entre los términos, es decir, identificar las secuencias de operaciones que producen el mismo resultado. Conviene evitar las ecuaciones redundantes.

Ecuaciones

- Ejemplo: Ecuaciones para booleanos:

variables

b : *bool*

ecuaciones

cierto $\wedge$ b = b

falso $\wedge$ b = falso

cierto $\vee$ b = cierto

falso $\vee$ b = b

$\neg$cierto = falso

$\neg$falso = cierto

- Usando las ecuaciones podemos convertir cualquier término en un término construido.

($\neg$cierto) $\vee$ falso = falso $\vee$ falso = falso

cierto $\wedge$ (cierto $\vee$ falso) = cierto $\wedge$ cierto = cierto

cierto $\wedge$ (cierto $\vee$ falso) = cierto $\vee$ falso = cierto

Especificación de los booleanos

especificación *BOOLEANOS*
tipos *bool*
operaciones
cierto : → *bool* { Constructora }
falso : → *bool* { Constructora }
_ ∧ _ : *bool bool* → *bool*
_ ∨ _ : *bool bool* → *bool*
¬ _ : *bool* → *bool*
variables
b : *bool*
ecuaciones
cierto ∧ *b* = *b*
falso ∧ *b* = falso
cierto ∨ *b* = cierto
falso ∨ *b* = *b*
¬cierto = falso
¬falso = cierto
fespecificación

Especificación de los naturales

especificación *NATURALES*
usa *BOOLEANOS*
tipos *nat*
operaciones

cero	:		→	*nat*	{ Constructora }
suc	:	*nat*	→	*nat*	{ Constructora }
_ + _, _ * _, _ - _	:	*nat nat*	→	*nat*	
exp	:	*nat nat*	→	*nat*	
_ == _, _ ≠ _	:	*nat nat*	→	*bool*	

variables
n,m : *nat*

Especificación de los naturales (cont.)

ecuaciones

cero + *m* = *m*

suc(*n*) + *m* = suc(*n* + *m*)

cero * *m* = cero

suc(*n*) * *m* = (*n* * *m*) + *m*

cero − *m* = cero { Al restar a un número otro mayor el resultado que se obtiene es cero }

suc(*n*) − cero = suc(*n*)

suc(*n*) − suc(*m*) = *n* − *m*

exp(*n*, cero) = suc(cero)

exp(*n*, suc(*m*)) = *n* ∗ exp(*n*,*m*)

cero == cero = cierto

cero == suc(*m*) = falso

suc(*n*) == cero = falso

suc(*n*) == suc(*m*) = *n* == *m*

n ≠ *m* = ¬(*n* == *m*)

fespecificación

Metodología de constructoras

- Elección de un conjunto de operaciones como **constructoras**: **operaciones que son suficientes para generar todos los valores del tipo y tales que la eliminación de cualquiera de ellas del conjunto impide construir alguno de los valores del tipo**. Puede haber más de una.

- Aserción de las relaciones entre constructoras.
  - El conjunto de operaciones constructoras puede o no ser *libre*. Se dice que las operaciones constructoras de un TAD son **no libres** si existen términos construidos diferentes que sean equivalentes entre sí. En caso contrario, se dice que las operaciones constructoras son **libres**.
  - Si las constructoras son libres entonces no escribimos ninguna ecuación que las relacione; por el contrario, si las constructoras no son libres es necesario escribir ecuaciones que permitan determinar las equivalencias que nos interesen entre términos construidos.

    Por ejemplo: $\neg\neg b = b$ siendo $b : bool$

Metodología de constructoras

- Especificación del resto de operaciones, una a una, respecto a las constructoras.
  - Definición de los efectos de aplicar las operaciones sobre términos formados exclusivamente por constructoras.
  - Al especificar operaciones observadoras asegurar dos propiedades: **consistencia** y **completitud suficiente**. Si se ponen ecuaciones de más, se pueden igualar términos que son diferentes en el tipo correspondiente, mientras que si se ponen de menos, se puede generar un número indeterminado (posiblemente infinito) de nuevos valores, diferentes a los ya existentes.

Especificación de los conjuntos de naturales

especificación *CONJUNTOS-NATURALES*
usa *BOOLEANOS, NATURALES*
tipos *conjunto*
operaciones

cjto-vacío : → *conjunto*
añadir : *nat conjunto* → *conjunto*
unit : *nat* → *conjunto*
unión : *conjunto conjunto* → *conjunto*
es-vacío? : *conjunto* → *bool*
está? : *nat conjunto* → *bool*

- Lo primero que hay que decidir es el conjunto de constructoras. Podemos tomar cjto-vacío y añadir. Otra posibilidad es tomar cjto-vacío, unit y unión.

Especificación de los conjuntos de naturales

- Con constructoras cjto-vacío y añadir

variables

n,m : *nat*

x,y : *conjunto*

ecuaciones

{ constructoras no libres}

añadir(*n*, añadir(*m*, *x*)) = añadir(*m*, añadir(*n*, *x*)) { conmutatividad }

añadir(*n*, añadir(*n*, *x*)) = añadir(*n*, *x*) { idempotencia }

unit(*n*) = añadir(*n*, cjto-vacío)

unión(cjto-vacío, *y*) = *y*

unión(añadir(*n*, *x*), *y*) = añadir(*n*, unión(*x*, *y*))

es-vacío?(cjto-vacío) = cierto

es-vacío?(añadir(*n*, *x*)) = falso

está?(*n*, cjto-vacío) = falso

está?(*n*, añadir(*m*, *x*)) = (*n* == *m*) ∨ está?(*n*, *x*)

fespecificación

} Ecuaciones de equivalencia

Especificación de los conjuntos de naturales

- Con constructoras cjto-vacío, unit y unión

variables

n,m : *nat*

x,y,z : *conjunto*

ecuaciones

{ constructoras no libres}

unión(x, y) = unión(y, x) { conmutatividad }

unión(x, unión(y, z)) = unión(unión(x, y), z) { asociatividad }

unión(x, x) = x { idempotencia }

unión(x, cjto-vacío) = x { elemento neutro }

añadir(n, x) = unión(unit(n), x)

es-vacío?(cjto-vacío) = cierto

es-vacío?(unit(n)) = falso

es-vacío?(unión(x, y)) = es-vacío?(x) ∧ es-vacío?(y)

está?(n, cjto-vacío) = falso

está?(n, unit(m)) = $n == m$

está?(n, unión(x, y)) = está?(n, x) ∨ está?(n, y)

fespecificación

Ecuaciones de equivalencia

Ecuaciones condicionales

- Hasta ahora, las ecuaciones expresan propiedades que se cumplen incondicionalmente, pero a veces un axioma se cumple sólo en determinadas condiciones.

$$t_0 = t'_0 \Leftarrow t_1 = t'_1 \wedge \ldots \wedge t_k = t'_k$$

- La condición de una ecuación es una conjunción de ecuaciones. Dado un *predicado* P, es decir, una operación con perfil $\mathrm{P} : T_1\ T_2\ \ldots\ T_n \rightarrow bool$, abreviaremos una condición de la forma $\mathrm{P}(t_1, \ldots, t_n) = \text{cierto}$ como $\mathrm{P}(t_1, \ldots, t_n)$, y una condición de la forma $\mathrm{P}(t_1, \ldots, t_n) = \text{falso}$ como $\neg\mathrm{P}(t_1, \ldots, t_n)$.

- Más en general, una ecuación de la forma $t = \text{cierto}$, donde t es un término de tipo $bool$, se abrevia como t en las condiciones.

Borrar elementos de un conjunto

especificación *CONJUNTOS-NATURALES-BORRAR*
usa *CONJUNTOS-NATURALES*
operaciones
quitar : *nat conjunto* → *conjunto*
variables
n,m : *nat*
x : *conjunto*
ecuaciones
quitar(*n,* cjto-vacío) = cjto-vacío
quitar(*n,* añadir(*n, x*)) = quitar(*n*, *x*)
quitar(*n,* añadir(*m, x*)) = añadir(*m*, quitar(*n*, *x*)) ⇐ $n \neq m$
fespecificación

Operaciones parciales

- En ocasiones, las operaciones de un tipo de datos, incluyendo las constructoras, pueden ser **parciales**. Es decir, pueden **no estar definidas para todos los valores de los parámetros**.

- Esta situación se explicitaría en la especificación señalando por un lado las operaciones parciales (mediante un subíndice p en el perfil de la operación $\rightarrow_p$), y escribiendo por otro lado **ecuaciones de error que indiquen en qué situación la operación en cuestión no está definida**.

- El resto de ecuaciones solo serán válidas si ambos términos están bien definidos, es decir, no producen error.

- Sin embargo, no haremos un tratamiento explícito de errores, es decir, que vamos a suponer implícitamente que cualquier operación aplicada a un error devuelve un error, y no vamos a escribir ecuaciones que hagan explícita esta propagación de errores.

Enriquecimiento de los naturales

especificación *NATURALES+*
usa *NATURALES*
operaciones

$\leq , < , \geq , >$	: *nat nat*	$\rightarrow$	*bool*	
max, min	: *nat nat*	$\rightarrow$	*nat*	
div, mod	: *nat nat*	$\rightarrow_p$	*nat*	{ operaciones parciales }
es-par?, es-impar?	: *nat*	$\rightarrow$	*bool*	

variables

n, *m* : *nat*

Enriquecimiento de los naturales (cont.)

ecuaciones

$\text{cero} \leq m = \text{cierto}$

$\text{suc}(n) \leq \text{cero} = \text{falso}$

$\text{suc}(n) \leq \text{suc}(m) = n \leq m$

$n < m = (n \leq m) \wedge (n \neq m)$

$n \geq m = m \leq n$

$n > m = m < n$

$\max(\text{cero}, n) = n$

$\max(\text{suc}(n), \text{cero}) = \text{suc}(n)$

$\max(\text{suc}(n), \text{suc}(m)) = \text{suc}(\max(n,m))$

$\min(\text{cero}, n) = \text{cero}$

$\min(\text{suc}(n), \text{cero}) = \text{cero}$

$\min(\text{suc}(n), \text{suc}(m)) = \text{suc}(\min(n,m))$

Enriquecimiento de los naturales (cont.)

- Las operaciones div y mod son operaciones parciales pues no se puede dividir por cero. En este caso la distinción de casos no está basada en constructoras, sino en la relación de orden entre los argumentos.

n div cero = error { las operaciones parciales producen errores }
n div m = cero $\Leftarrow n < m$
n div m = suc(($n - m$) div m) $\Leftarrow m \neq$ cero $\wedge\ m \leq n$
n mod cero = error
n mod m = $n \Leftarrow n < m$
n mod m = ($n - m$) mod $m \Leftarrow m \neq$ cero $\wedge\ m \leq n$
es-par?(cero) = cierto
es-par?(suc(n)) = es-impar?(n)
es-impar?(cero) = falso
es-impar?(suc(n)) = es-par?(n)

fespecificación

Especificación de los conjuntos finitos

especificación *CONJUNTOS[ELEM=]*
usa *BOOLEANOS,NATURALES*
tipos *conjunto*
operaciones

cjto-vacío : → *conjunto*
añadir : *elemento conjunto* → *conjunto*
unit : *elemento* → *conjunto*
está? : *elemento conjunto* → *bool*
es-vacío? : *conjunto* → *bool*
quitar : *elemento conjunto* → *conjunto*
unión : *conjunto conjunto* → *conjunto*
intersección : *conjunto conjunto* → *conjunto*
diferencia : *conjunto conjunto* → *conjunto*
cardinal : *conjunto* → *nat*

- Lo primero que hay que decidir es el conjunto de constructoras. Podemos tomar cjto-vacío y añadir. Otra posibilidad es tomar cjto-vacío, unit y unión. Elegimos la primera opción.

Especificación de los conjuntos finitos

variables

e, f : *elemento*

x, y, z : *conjunto*

ecuaciones

{ constructoras no libres }

añadir(e, añadir(f, x)) = añadir(f, añadir(e, x))

añadir(e, añadir(e, x)) = añadir(e, x)

unit(e) = añadir(e, cjto-vacío)

es-vacío?(cjto-vacío) = cierto

es-vacío?(añadir(e, x)) = falso

está?(e, cjto-vacío) = falso

está?(e, añadir(f, x)) = ($e == f$) V está?(e, x)

quitar(e, cjto-vacío) = cjto-vacío { no es parcial }

quitar(e, añadir(e, x)) = quitar(e, x)

quitar(e, añadir(f, x)) = añadir(f, quitar(e, x)) $\Leftarrow e \neq f$

unión(cjto-vacío, y) = y

unión(añadir(e, x), y) = añadir(e, unión(x, y))

} Ecuaciones de equivalencia

Especificación de los conjuntos finitos

intersección(cjto-vacío, y) = cjto-vacío
intersección(añadir(e, x), y) = intersección(x, y) ⇐ ¬está?(e, y)
intersección(añadir(e, x), y) = añadir(e, intersección(x, y)) ⇐ está?(e, y)
diferencia(x, cjto-vacío) = x
diferencia(x, añadir(e, y)) = diferencia(quitar(e, x), y)
cardinal(cjto-vacío) = 0
cardinal(añadir(e, x)) = cardinal(quitar(e, x)) + 1

fespecificación

IMPLEMENTACIÓN DE TAD

Pasos en la implementación de un TAD

- Las implementaciones que vamos a realizar seguirán el **paradigma de programación imperativo**.

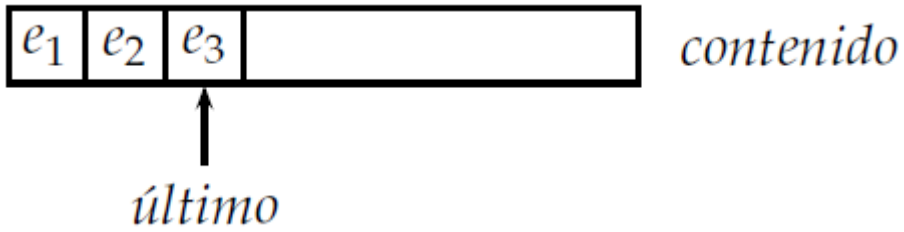
- **Implementar un TAD** consiste en:
  - Representar sus **valores** por medio de valores de tipos de datos más concretos del lenguaje de programación (tipo representante).
    - Esta representación se oculta al usuario del TAD.
    - Existe mas de una representación posible.
  - Simular sus **operaciones** por medio de funciones o procedimientos que actúan sobre dichos tipos más concretos.

Pasos en la implementación de un TAD

- Existen dos tipos de representaciones:
  - **Estáticas**: el tamaño de la estructura no cambia durante la ejecución.
    - Desaprovechamiento de la memoria.
    - Desbordamiento.
  - **Dinámicas**:
    - Utilizan memoria dinámica.
    - Proporcionan estructuras mas versátiles ↔ No tienen un tamaño fijo durante la ejecución.
    - Su programación es más compleja.

- Una parte esencial de la programación de cualquier algoritmo es el estudio de su coste en tiempo y en memoria. En general, nos referiremos al **coste en tiempo en el caso peor**.

Implementación de conjuntos finitos mediante vectores



tipos

conjunto = **reg**

contenido[1..N] de elemento

último : 0..N

freg

ftipos

- Nótese que un conjunto es vacío si y sólo si el índice *último* vale cero, y que se ignora la información que pueda haber en el vector entre las posiciones *último* + 1 y *N*.

Implementación de conjuntos finitos mediante vectores

- Implementación de las operaciones cjto-vacío, unit y es-cjto-vacío?:

fun cjto-vacío() **dev** *x :* conjunto { $O(1)$ }
    x. último := 0
ffun

fun unit(*e :* elemento) **dev** *x :* conjunto { $O(1)$ }
    x. último := 1
    x.contenido[1] := *e*
ffun

fun es-cjto-vacío?(*x :* conjunto*)* **dev** *b :* bool { $O(1)$ }
    b := (x. último = 0*)*
ffun

Implementación de conjuntos finitos mediante vectores

- Implementación de la operación está?.

fun está?(*e :* elemento, *x :* conjunto*)* **dev** *b :* bool { *O*(*x.último*) }
    b := falso
    i := 1
    mientras $i \leq x.\ último \wedge \neg b$ **hacer**
      b := *(x.contenido*[*i*] = *e)*
      i := *i* + 1
    fmientras
ffun

- El coste es lineal con respecto al tamaño de la parte ocupada del vector.

Implementación de conjuntos finitos mediante vectores

- Implementación de la operación añadir (vector sin repeticiones): se añade el elemento en la primera posición libre, pero sólo si el elemento no está ya en el vector.

```
proc añadir(e e : elemento, x : conjunto) {  O(x.último)  }
    si ¬está?(e, x) entonces
       si x.último = N entonces error(Espacio insuficiente)
       si no
     x.último := x.último + 1 ;
     x.contenido[x.último] := e
       fsi
    fsi
fproc
```

- El coste es lineal debido a la búsqueda.

Implementación de conjuntos finitos mediante vectores

- Implementación de la operación quitar (vector sin repeticiones): podemos parar cuando se encuentra el elemento (o cuando se llega al final si no está). Para llenar el hueco que se deja al quitar un elemento, se coloca allí el último elemento.

proc quitar(**e** *e :* elemento, *x :* conjunto*)* { $O(x.último)$ }

$i := 1$

mientras $i \leq x.último \wedge_c x.contenido[i] \neq e$ **hacer**

$i := i + 1$

fmientras

si $i \leq x.último$ **entonces**

$x.contenido[i] := x.contenido[x.último]$;

$x.último := x.último - 1$

fsi

fproc

Implementación de conjuntos finitos mediante vectores

- Una posibilidad para implementar las operaciones unión, intersección y diferencia es hacerlo en términos de cjto-vacío, está? y añadir.

fun unión(x, y : conjunto) **dev** z : conjunto { $O\left((x.último + y.último)^2\right)$ }

z := cjto-vacío()

para i = 1 **hasta** $x.último$ **hacer**

añadir($x.contenido[i]$, z)

fpara

para i = 1 **hasta** $y.último$ **hacer**

añadir($y.contenido[i]$, z)

fpara

ffun

Implementación de conjuntos finitos mediante vectores

fun intersección(*x*, *y* : conjunto) **dev** *z* : conjunto { $O(x.último(y.último + x.último))$ }
    z := cjto-vacío()
    para *i* = 1 **hasta** *x.último* **hacer**
        si está?(*x.contenido*[*i*], *y*) **entonces** añadir(*x.contenido*[*i*], *z*) **fsi**
    fpara
ffun

fun diferencia(*x*, *y* : conjunto) **dev** *z* : conjunto { $O(x.último(y.último + x.último))$ }
    z := cjto-vacío()
    para *i* = 1 **hasta** *x.último* **hacer**
        si ¬está?(*x.contenido*[*i*], *y*) **entonces** añadir(*x.contenido*[*i*], *z*) **fsi**
    fpara
ffun

Implementación de conjuntos finitos mediante vectores

- Calcular el cardinal es inmediato.

fun cardinal(x : conjunto) **dev** n : nat { $O(1)$ }
    $n := x.último$
ffun

Implementación de los conjuntos finitos con elementos en 1..N

- Al imponer requisitos muy exigentes sobre los elementos, podemos simplificar considerablemente la representación general de los conjuntos.

tipos

conjunto : **vector**[1..*N*] **de** bool

elemento: 1..*N*

ftipos

- Para crear el conjunto vacío, todas las posiciones del vector se rellenar con el valor falso.

fun cjto-vacio() **dev** *x*: conjunto { $O(N)$ }

x[1..*N*] := [falso]

ffun

Implementación de los conjuntos finitos con elementos en 1..N

- Para añadir un elemento, la posición del vector correspondiente a ese elemento se rellena con cierto (si el elemento ya estaba, la información no ha cambiado) y las demás posiciones se dejan igual.

```
proc añadir(e e: elemento, x: conjunto)    { O(1) }
    x[e] := cierto
fproc
```

- Para quitar un elemento se aplica la misma idea con falso.

```
proc quitar(e e: elemento, x: conjunto)    { O(1) }
    x[e] := falso
fproc
```

Implementación de los conjuntos finitos con elementos en 1..N

- Para construir un vector unitario con el elemento e, podemos hacer x := cjto-vacío();
  añadir (e,x). Expandiendo ambos algoritmos se obtiene el siguiente:

fun unit(e: elemento) **dev** x : conjunto { $O(N)$ }
  $x[1..N]$:= [falso]
  $x[e]$:= cierto
ffun

- En esta representación las búsquedas son inmediatas, pues basta acceder directamente a la posición correspondiente del vector.

fun está?(e: elemento, x: conjunto) **dev** b: bool { $O(1)$ }
  $b := x[e]$
ffun

Implementación de los conjuntos finitos con elementos en 1..N

- Para reconocer el conjunto vacío ahora es necesario recorrer todo el vector, puesto que la representación no contiene ningún contador que facilite la información relacionada con la cardinalidad del conjunto representado.

```
fun es-cjto-vacio?(x: conjunto) dev b: bool     {  O(N)  }
    i := 1
    b := cierto
    mientras i ≤ N ∧ b hacer
        b := ¬ x[i]
        i := i + 1
    fmientras
ffun
```

- En el caso mejor el coste es constante, pero en el caso peor el coste es lineal.

Implementación de los conjuntos finitos con elementos en 1..N

- Las operaciones de unión, intersección y diferencia tienen una traducción inmediata en términos de operaciones booleanas.

```
fun unión(x, y : conjunto) dev z : conjunto      { O(N) }
    para i = 1 hasta N hacer
        z[i] := x[i] ∨ y[i]
    fpara
ffun

fun intersección(x, y : conjunto) dev z : conjunto    { O(N) }
    para i = 1 hasta N hacer
        z[i] := x[i] ∧ y[i]
    fpara
ffun
```

Implementación de los conjuntos finitos con elementos en 1..N

```
fun diferencia(x, y : conjunto) dev z : conjunto      { O(N) }
    para i = 1 hasta N hacer
        z[i] := x[i] ∧ ¬ y[i]
    fpara
ffun
```

Implementación de los conjuntos finitos con elementos en 1..N

- Como la representación no contienen ningún contador explícito, para calcular el cardinal del conjunto representado hay que recorrer todo el vector, contando el número de posiciones cuyo valor es cierto.

```
fun cardinal(x : conjunto) dev n: nat {  O(N)  }
    n := 0
    para i = 1 hasta N hacer
       si x[i] entonces
          n := n + 1
       fsi
    fpara
ffun
```

Bibliografía

- Martí, N., Ortega, Y., Verdejo, J.A. *Estructuras de datos y métodos algorítmicos*. Ejercicios resueltos. Pearson/Prentice Hall, 2003. Capítulos 1 y 2

- Peña, R.; *Diseño de programas. Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005. Capítulo 5

- Franch, X. *Estructuras de Datos: Especificación, diseño e implementación*. Ediciones UPC, 1999. Capítulo 1

(Estas transparencias se han realizado a partir de aquéllas desarrolladas por los profesores Clara Segura y Alberto Verdejo de la UCM, y basadas en la bibliografía anterior)

Derechos de Autor