incluyendo el anterior comportamiento como aserto invariante del tipo Pila, el resultado final de la expresión sería: "La pila está vacía".

2.4. Especificaciones formales constructivas

En una especificación formal constructiva, u operacional, el significado de las operaciones se establece mediante las cláusulas de **precondición** y **postcondición**. Al contrario de las especificaciones algebraicas –donde las operaciones se definen a través de las relaciones entre ellas– en las especificaciones constructivas cada operación es descrita independientemente de las demás. Por este motivo, se dice que en el método algebraico el significado de las operaciones es *implícito*, mientras que en el constructivo el significado es *explícito*.

El formato de notación utilizado en las especificaciones constructivas coincide con el ya visto para las algebraicas en las partes Nombre, Conjuntos y Sintaxis. La diferencia está en la parte Semántica, que contendrá la pre- y postcondición de cada operación.

2.4.1. Precondiciones y postcondiciones

La precondición y la postcondición son condiciones lógicas que indican, respectivamente, los requisitos de una operación y su efecto.

- **Precondición.** Relación lógica que deben de cumplir los valores de entrada para que la operación se pueda aplicar con garantías de producir una ejecución correcta.
- **Postcondición.** Relación lógica que se cumple con los valores de salida después de ejecutar la operación, siempre que se cumpliera la precondición.

Las pre- y postcondiciones son también llamadas **asertos**: afirmaciones que deben ser ciertas antes y después de ejecutar una operación, respectivamente. Para cada operación <nombre> definida en la parte Sintaxis, dentro de la parte Semántica deben aparecer las dos siguientes cláusulas:

pre-<**nombre**>(<param_entrada>)::= <condición_lógica>
post-<**nombre**>(<param_entrada>;<param_salida>)::= <condición_lógica>

Ejemplo 2.14 Especificación constructiva de una operación maximo, que tiene como entrada dos números reales y devuelve como resultado el mayor de los dos.

maximo: $\mathbf{R} \times \mathbf{R} \rightarrow \mathbf{R}$

pre-maximo(x, y)::= verdadero

post-maximo$(x, y; r)$::= $(r \geq x) \wedge (r \geq y) \wedge (r = x \vee r = y)$

Ejemplo 2.15 Especificación constructiva de una operación max_restring, máximo restringido, que calcula el máximo de dos números, pero sólo se puede aplicar a enteros positivos.

max_restring: $\mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$

pre-max_restring(x, y)::= $(x > 0) \wedge (y > 0)$

post-max_restring$(x, y; r)$::= $(r \geq x) \wedge (r \geq y) \wedge (r = x \vee r = y)$

Como se ve en los ejemplos anteriores, la postcondición no debe indicar necesariamente un valor concreto para el resultado. Simplemente expresa una condición lógica que es cierta. Igual que en las especificaciones algebraicas, es posible usar condicionales de la forma *SI* <*condición*> ⇒ <*valor_si_cierto*> | <*valor_si_falso*>. Además, también se pueden usar los cuantificadores ∀ y ∃.

Ejemplo 2.16 Especificación constructiva de una operación BusquedaBinaria, que realiza una búsqueda binaria dentro de un array de elementos de un tipo parametrizado T.

BusquedaBinaria: Array[T] × T → N

pre-BusquedaBinaria(a,t)::= $\forall i = 11,...,\text{Tamaño}(a)-1; a[i] \leq a[i+1]$

post-BusquedaBinaria$(a,t;n)$::= SI $t < a[1] \Rightarrow n = 0$ | SI $t > a[\text{Tamaño}(a)] \Rightarrow n = \text{Tamaño}(a)+1$ | $(a[n] = t) \vee (a[n] < t < a[n+1])$

Se supone que la operación Tamaño, definida sobre arrays, devuelve el número de elementos del array, y que el tipo parametrizado T tiene definidas operaciones de comparación entre valores del tipo.

En este caso, la precondición expresa el requisito de que el array de entrada debe estar ordenado de menor a mayor. En tal caso, la operación devuelve la posición donde se encuentra, o se debería encontrar, el valor buscado. Este resultado es lo que se indica en la postcondición. Pero se puede ver que la postcondición no dice nada de cómo se busca el elemento. Es un detalle de implementación que no debe aparecer en la especificación.

Error frecuente 2.1 Es muy importante no confundir *especificación* e *implementación*: la especificación describe el resultado esperado de una operación; la implementación es un trozo de código que calcula ese resultado. Por lo tanto, la especificación debe ser independiente de la implementación. Sin embargo, el método constructivo permite definir especificaciones que son muy próximas a implementaciones. Por ejemplo, una posible postcondición para la operación máximo podría ser:

post-maximo$(x,y;r)$::= SI $x > y \Rightarrow r = x$ | $r = y$

En este ejemplo sencillo, se ve más claramente el significado de la operación. Sin embargo, en general, hay que evitar este tipo de especificaciones porque mezclan especificación con implementación, ofreciendo al usuario de la abstracción detalles innecesarios de programación.

2.4.2. Especificación como contrato de una operación

El significado de una especificación constructiva es: si se cumple la precondición y se ejecuta la operación, entonces se puede asegurar que se cumple la postcondición. Una buena analogía para una especificación de este tipo es un **contrato** –que conlleva ciertos derechos y obligaciones– que se establece entre el que implementa la operación y el que la usa. La idea de la especificación como un contrato es mostrada en la figura 2.3.

Figura 2.3: La especificación de un producto software como contrato entre el que lo programa y el que lo usa.

En concreto, el contrato en la especificación constructiva estipula lo siguiente:

IMPLEMENTADOR

Derechos	**Obligaciones**
Supone que se cumple la precondición, sin preocuparse de qué hacer en caso contrario. Esto le evita tener que comprobarla.	Debe hacer los cálculos necesarios para conseguir que se cumpla la postcondición al acabar la ejecución de la operación.

USUARIO

Obligaciones	**Derechos**
Es el responsable exclusivo de garantizar que se cumpla la precondición, antes de ejecutar la operación.	Obtiene los resultados que se indican en la postcondición, siempre que haya cumplido sus obligaciones.

Así que el responsable de comprobar la precondición es el usuario de la operación, y el responsable de garantizar la postcondición es el implementador de la operación. Este reparto de responsabilidades ayuda a simplificar la construcción de programas, puesto que se evitan comprobaciones redundantes e innecesarias.

Pero, ¿qué ocurre si no se cumple una precondición? Simplemente no se garantiza que se cumpla la postcondición. En tales casos, el creador de la especificación puede elegir básicamente entre dos opciones:

- No especificar nada. En caso de error en los parámetros de entrada, el efecto será imprevisible; puede que en algunas situaciones se cumpla la postcondición, en otras no, o incluso puede que el programa se cuelgue.

- Especificar un comportamiento estándar en caso de fallo de la precondición. Este comportamiento puede ser, por ejemplo, interrumpir la ejecución del programa, seguir como si no se hubiera ejecutado la operación, indicar la situación en una variable de error o provocar una *excepción*.

En la práctica, el uso de **excepciones** suele ser la solución más utilizada. La ejecución de una excepción implica que la operación interrumpe su ejecución. El control pasa al procedimiento que ha ejecutado la llamada, que tendrá definido un código para el tratamiento de excepciones. En caso contrario, la excepción pasará al que lo ha llamado y así sucesivamente.

Una tercera opción sería incluir el tratamiento del error dentro de la operación que la implementa y también dentro de la propia especificación formal. Igual que en la especificación algebraica, se debería indicar en la sintaxis que la operación puede devolver un mensaje de error. Por ejemplo, vamos a suponer que en la especificación constructiva de la operación máximo restringido incluimos el tratamiento de errores, en caso de que alguno de los parámetros no sea positivo. La especificación sería:

max_restring2: Z $\times$ **Z** $\rightarrow$ **Z** $\cup$ {**"Error de precondición"**}

pre-max_restring2(x, y)::= verdadero

post-max_restring2$(x, y; r)$::= SI $(x > 0) \wedge (y > 0)$
$\Rightarrow (r \geq y) \wedge (r \geq y) \wedge (r = x \vee r = y)$
| "Error de precondición"

Está claro que esta elección rompe la filosofía de los contratos: el implementador comprueba y trata las condiciones de error en los parámetros de entrada, lo cual no debería ser su responsabilidad. Al definirlo de esta manera, se puede decir que la especificación "oculta" los verdaderos requisitos de la operación, puesto que la precondición siempre es cierta. En cierto sentido, el cliente es "engañado": cree que puede ejecutar la operación sin problemas, pero se encuentra con mensajes de error inesperados.

2.4.3. Necesidad de un modelo subyacente

Si el TAD que se está definiendo es un tipo complejo, expresar las precondiciones y las postcondiciones usando únicamente operaciones lógicas puede ser difícil o inviable. En esos casos es posible utilizar otro tipo de datos como **tipo subyacente**, en el cual se basa la especificación del tipo que está siendo definido. Este tipo o modelo subyacente debería estar especificado formalmente, ya sea por el método algebraico o por el constructivo.

Un ejemplo de TAD donde resulta necesario un modelo subyacente son los tipos colección o contenedores. Para definir de forma constructiva cualquier contenedor es necesario usar como modelo subyacente algún otro tipo contenedor. En última instancia necesitaremos algún TAD contenedor cuya definición no esté basada en otro contenedor; este tipo subyacente será especificado de forma algebraica.

Vamos a ver como ejemplo la especificación formal constructiva de los TAD genéricos Pila[T] y Cola[T]. La definición de estos tipos se basará en el modelo subyacente Lista[T], el cual será definido de forma algebraica.

Ejemplo 2.17 Especificación formal algebraica del TAD parametrizado Lista[T], que define las listas de un tipo cualquiera T. Las listas se crean con las operaciones: crearLista, que devuelve una lista vacía; formarLista, que devuelve una lista con un sólo elemento; y concatenar que une dos listas en una.

NOMBRE

Lista[T]

CONJUNTOS

L Conjunto de listas de tipo T
T Conjunto de elementos que pueden ser almacenados
B Conjunto de valores booleanos
N Conjunto de naturales
M Conjunto de mensajes {"La lista está vacía"}

SINTAXIS

crearLista : $\rightarrow$ L
formarLista : T $\rightarrow$ L
concatenar : L $\times$ L $\rightarrow$ L
primero : L $\rightarrow$ T $\cup$ M
último : L $\rightarrow$ T $\cup$ M
cabecera : L $\rightarrow$ L
cola : L $\rightarrow$ L
longitud : L $\rightarrow$ N
esVacía : L $\rightarrow$ B

SEMÁNTICA

$\forall$ t $\in$ T ;$\forall$ a, b $\in$ L
1. primero(crearLista) = "La lista está vacía"
2. primero(formarLista(t)) = t
3. primero(concatenar(a, b)) = SI esVacía(a) $\Rightarrow$ primero(b) | primero(a)
4. último(crearLista) = "La lista está vacía"
5. último(formarLista(t)) = t
6. último(concatenar(a, b)) = SI esVacía(b) $\Rightarrow$ último(a) | último(b)
7. cabecera(crearLista) = crearLista
8. cabecera(formarLista(t)) = crearLista
9. cabecera(concatenar(a, b)) = SI esVacía(b) $\Rightarrow$ cabecera(a) | concatenar(a, cabecera(b))
10. cola(crearLista) = crearLista
11. cola(formarLista(t)) = crearLista
12. cola(concatenar(a, b)) = SI esVacía(a) $\Rightarrow$ cola(b) | concatenar(cola(a), b)
13. longitud(crearLista) = cero
14. longitud(formarLista(t)) = sucesor (cero)
15. longitud(concatenar(a, b)) = suma(longitud(a), longitud(b))
16. esVacía(crearLista) = verdadero
17. esVacía(formarLista(t)) = falso
18. esVacía(concatenar(a, b)) = esVacía(a) AND esVacía(b)

Ejemplo 2.18 Especificación formal constructiva del TAD parametrizado Cola[T], usando como modelo subyacente el tipo Lista[T].

NOMBRE

Cola[T]

CONJUNTOS

Q Conjunto de colas de tipo T
L Conjunto de listas de tipo T
T Conjunto de elementos que pueden ser almacenados

B Conjunto de valores booleanos

SINTAXIS

crearCola : $\rightarrow$ Q
frente : Q $\rightarrow$ T
inserta : T $\times$ Q $\rightarrow$ Q
resto : Q $\rightarrow$ Q
esVacíaCola: Q $\rightarrow$ B

SEMÁNTICA

$\forall$ t $\in$ T; $\forall$ p, q $\in$ Q; $\forall$ b $\in$ B
pre-crearCola()::= verdadero
post-crearCola(; q)::= q=crearLista
pre-frente(q)::= not esVacía(q)
post-frente(q; t)::= t=primero(q)
pre-inserta(t, q)::= verdadero
post-inserta(t, q; p)::= p=concatenar(q, formarLista(t))
pre-resto(q)::= not esVacía(q)
post-resto(q; p)::= p=cola(q)
pre-esVacíaCola(q)::= verdadero
post-esVacíaCola(q; b)::= b=esVacía(q)

Ejemplo 2.19 Especificación formal constructiva del TAD parametrizado Pila[T], usando como modelo subyacente el tipo Lista[T]. Se puede ver que la única diferencia con la especificación del TAD Cola[T] es el orden de inserción de los elementos en la operación push.

NOMBRE

Pila[T]

CONJUNTOS

S Conjunto de pilas de tipo T
L Conjunto de listas de tipo T
T Conjunto de elementos que pueden ser almacenados
B Conjunto de valores booleanos

SINTAXIS

crearPila : $\rightarrow$ S
tope : S $\rightarrow$ T
push : T $\times$ S $\rightarrow$ S
pop : S $\rightarrow$ S
esVacíaPila : S $\rightarrow$ B

SEMÁNTICA

$\forall$ t $\in$ T; $\forall$ s, r $\in$ S; $\forall$ b $\in$ B
pre-crearPila()::= verdadero
post-crearPila(; s)::= s=crearLista
pre-tope(s)::= not esVacía(s)
post-tope(s; t)::= t=primero(s)
pre-push(t, s)::= verdadero
post-push(t, s; r)::= r=concatenar(formarLista(t), s)

pre-pop(s)::= not esVacía(s)
post-pop(s; r)::= r=cola(s)
pre-esVacíaPila(s)::= verdadero
post-esVacíaPila(s; b)::= b=esVacía(s)

2.4.4. Ejecución de especificaciones constructivas

Como vimos en la introducción del apartado 2.3.2, una de las ventajas de las especificaciones formales, respecto a las informales, es la posibilidad de ejecutar o, más exactamente, simular las especificaciones. La ejecución de una expresión constructiva consiste en comprobar las precondiciones y postcondiciones. Para cada llamada, se comprueba la precondición; si es cierta, entonces se puede dar por válida la postcondición; si no se cumple la precondición, el mecanismo de ejecución de la especificación indicaría el error y el punto donde se ha producido.

Ejemplo 2.20 Vamos a comprobar el resultado obtenido con la especificación constructiva del tipo Pila[Natural] para la siguiente expresión:
tope(push(4, pop(push(2, crearPila))))

Operación	Precondición	Postcondición
1. crearPila	verdadero	o1= crearLista
2. push(2, o1)	verdadero	o2= concatenar(formarLista(2), o1)
3. pop(o2)	not esVacía(o2) = [Ax. 18] not (esVacía(formarLista(2)) AND esVacía(crearLista)) = [Ax. 16,17] not (falso AND verdadero) = not falso = verdadero	o3 = cola(o2) = [Ax. 12] SI esVacía(formarLista(2)) ⇒ cola(crearLista) | concatenar(cola(formarLista(2)),crearLista) = [Ax. 17] concatenar(cola(formarLista(2), crearLista) = [Ax. 11] concatenar(crearLista, crearLista)
4. push(4, o3)	verdadero	o4= concatenar(formarLista(4), o3)
5. tope(o4)	not esVacía(o4) = [Ax. 18] not (esVacía(formarLista(4)) AND esVacía(o3)) = [Ax. 17] not (falso AND esVacía(o3)) = not falso = verdadero	o5 = primero(o4) = [Ax. 3] SI esVacía(formarLista(4)) ⇒ primero(o3) | primero(formarLista(4)) [Ax. 17] primero(formarLista(4)) = [Ax. 2] 4

Ejemplo 2.21 Vamos a comprobar ahora el resultado obtenido usando la especificación del TAD Cola[Natural] para la expresión:
esVacíaCola(inserta(1, resto(crearCola)))

Operación	Precondición	Postcondición
1. crearCola	verdadero	o1= crearLista
2. resto(o1)	not esVacía(o1) = not esVacía(crearLista) = [Ax. 16] not verdadero = falso	**ERROR en la expresión resto(crearCola)**. No se cumple la precondición. Ejecución interrumpida.

Otra interesante aplicación de las especificaciones constructivas es comprobar la corrección de programas. El concepto de **corrección de un programa** es un concepto relativo: un programa se dice que es correcto si cumple su especificación. Usando especificaciones constructivas es posible comprobar la ejecución correcta de un programa. Para ello, la precondición y la postcondición son añadidas como comprobaciones al principio

y al final de cada procedimiento, respectivamente. Un fallo de la precondición indica un error en el procedimiento que hace la llamada; un fallo en la postcondición indica un error de programación en la función correspondiente.

Por ejemplo, utilizando C++ las pre- y postcondiciones pueden ser comprobadas usando la función `assert(condicion)`. Si en el momento de ejecutarse la función `assert` la condición es cierta, se sigue la ejecución normalmente. Si la condición es falsa, se genera una excepción, indicando el punto donde se ha producido.

Ejemplo 2.22 Implementación de precondiciones y postcondiciones en C++. Especificación formal constructiva de una operación RaizCuarta que, dado un número real positivo x, calcula $\sqrt[4]{x}$.

RaizCuarta : R → R

pre-RaizCuarta(x)::= $x \geq 0$

post-RaizCuarta(x; r)::= $r^4 = x$

Implementación de la operación en C++, incluyendo la comprobación de precondición y postcondición.

```
float RaizCuarta (float x)
{
  assert (x>=0);        // PRECONDICIÓN
  float r;
  r= sqrt(x);
  if (r!=0.0) r= sqrt(r);
  assert (r*r*r*r==x);  // POSTCONDICIÓN
  return r;
}
```

En este caso concreto, hay que llevar cuidado con los redondeos de los números en punto flotante. Con toda probabilidad, multiplicar cuatro veces la raíz cuarta de x no dará exactamente x. Una postcondición más adecuada podría ser algo como:

```
assert (fabs(r*r*r*r-x)<= 0.0001); // POSTCONDICION
```

Ejercicios resueltos

Ejercicio 2.1 Desarrollar una especificación formal para el TAD genérico Conjunto[T], por el método axiomático o algebraico. La especificación debe ser completa, es decir, todas las operaciones deben estar definidas en la semántica. Incluye las operaciones: Vacío, EsVacío, Inserta, Suprime, Miembro, Unión, Intersección, Diferencia y Cardinalidad (número de elementos del conjunto).

Solución.

Los dos constructores del tipo son las operaciones Vacío e Inserta. Así pues, debemos relacionar las restantes operaciones con ambos constructores de la forma adecuada, es decir, de manera que podamos obtener el resultado de cualquier expresión en función de los constructores.

NOMBRE

Conjunto[T]

CONJUNTOS

C Conjunto de conjuntos de elementos de tipo T
T Conjunto de elementos que pueden ser almacenados
N Conjunto de los números naturales
B Conjunto de valores booleanos

SINTAXIS

Vacío : $\rightarrow$ C
Inserta : C $\times$ T $\rightarrow$ C
EsVacío : C $\rightarrow$ B
Suprime : C $\times$ T $\rightarrow$ C
Miembro : C $\times$ T $\rightarrow$ B
Unión : C $\times$ C $\rightarrow$ C
Intersección : C $\times$ C $\rightarrow$ C
Diferencia : C $\times$ C $\rightarrow$ C
Cardinalidad : C $\rightarrow$ N

SEMÁNTICA

$\forall$ c, d $\in$ C; $\forall$ t, p $\in$ T
1. EsVacío(Vacío) = verdadero
2. EsVacío(Inserta(c, t)) = falso
3. Suprime(Vacío, t) = Vacío
4. Suprime(Inserta(c, t), p) = SI (t = p) $\Rightarrow$ Suprime(c) | Inserta(Suprime(c, p), t)
5. Miembro(Vacío, t) = falso
6. Miembro(Inserta(c, t), p) = SI (t = p) $\Rightarrow$ verdadero | Miembro(c, p)
7. Unión(Vacío, c) = c
8. Unión(Inserta(c, t), d) = SI Miembro(d, t) $\Rightarrow$ Unión(c, d) | Inserta(Unión(c, d), t)
9. Intersección(Vacío, c) = Vacío
10. Intersección(Inserta (c, t), d) = SI Miembro(d, t) $\Rightarrow$ Inserta(Intersección(c, d), t)
    | Intersección(c, d)
11. Diferencia(Vacío, c) = Vacío
12. Diferencia(Inserta(c, t), d) = SI Miembro(d, t) $\Rightarrow$ Diferencia(c, d)
    | Inserta(Diferencia(c, d), t)
13. Cardinalidad(Vacío) = cero
14. Cardinalidad(Inserta(c, t)) = SI Miembro(c, t) $\Rightarrow$ Cardinalidad(c)
    | Sucesor(Cardinalidad(c))

Ejercicio 2.2 A la especificación del ejercicio 2.1 añade las operaciones máximo(c) y mínimo(c) (para calcular el mayor y el menor elemento del conjunto c) y sucesor(c, n) y predecesor(c, n) (para calcular el elemento de c más próximo a n, por arriba o por abajo, respectivamente). Se supondrá que existen las operaciones adecuadas de comparación entre valores de tipo T.

Solución.

Los axiomas deben relacionar cada una de las operaciones anteriores con los constructores del tipo, Vacío e Inserta. Vamos a suponer que el TAD T tiene definidas las operaciones min, max : T $\times$ T $\rightarrow$ T, y mayorQue : T $\times$ T $\rightarrow$ B. Además, suponemos que

el máximo y el mínimo valor representable del tipo T son masInfinitoTipoT y menosInfinitoTipoT, respectivamente.

Las partes que se ven modificadas en la sintaxis y en la semántica son las siguientes.

SINTAXIS

máximo: C → T
mínimo: C → T
sucesor: C × T → T
predec: C × T → T

SEMÁNTICA

∀ c ∈ C; ∀ t, p ∈ T
15. máximo(Vacío) = menosInfinitoTipoT
16. máximo(Inserta(c, t)) = max(t, máximo(c))
17. mínimo(Vacío) = masInfinitoTipoT
18. mínimo(Inserta(c, t)) = min(t, mínimo(c))
19. sucesor(Vacío, t) = masInfinitoTipoT
20. sucesor(Inserta(c, t), p) = SI mayorQue(t, p) ⇒ min(t, sucesor(c, p)) | sucesor(c, p)
21. predec(Vacío, t) = menosInfinitoTipoT
22. predec(Inserta(c, t), p) = SI mayorQue(p, t) ⇒ max(t, predec(c, p)) | predec(c, p)

Otra posibilidad sería decidir que se devuelva un mensaje de error en caso de aplicar las operaciones máximo y mínimo sobre un conjunto vacío, o en caso de que no exista ningún predecesor o sucesor en el conjunto. De esta forma, habría que añadir un conjunto M de mensajes de error, y cambiar la sintaxis y la semántica de las operaciones.

CONJUNTOS

...
M Conjunto de mensajes de error {"El conjunto está vacío", "No existe sucesor", "No existe predecesor"}

SINTAXIS

máximo: C → T ∪ M
mínimo: C → T ∪ M
sucesor: C × T → T ∪ M
predec: C × T → T ∪ M

SEMÁNTICA

∀ c ∈ C; ∀ t, p ∈ T
15'. máximo(Vacío) = "El conjunto está vacío"
16'. máximo(Inserta(c, t)) = SI EsVacío(c) ⇒ t | max(t, máximo(c))
17'. mínimo(Vacío) = "El conjunto está vacío"
18'. mínimo(Inserta(c, t)) = SI EsVacío(c) ⇒ t | min(t, mínimo(c))
19'. sucesor(c, t) = SI EsVacío(c) ⇒ "No existe sucesor"
| SI mayorQue(mínimo(c), t) ⇒ mínimo(c) | sucesor(Suprime(c, mínimo(c)), t)

20'. predec(c, t) = SI EsVacío(c) ⇒ "No existe predecesor"
| SI mayorQue(t, máximo(c)) ⇒ máximo(c) | predec(Suprime(c, máximo(c)), t)

Ejercicio 2.3 Partiendo de la especificación formal para el TAD Natural vista en el apartado 2.3.3, añade a la especificación las operaciones: predecesor, producto y poten-

cia. Ten en cuenta que pueden ocurrir casos donde el resultado no sea un número natural.
Solución.

Puesto que el resultado de estas operaciones puede ser un número no natural, debemos añadir un valor NoNatural a la especificación. Obviamente, no tiene sentido añadir ese valor como una operación constante del tipo NoNatural : $\rightarrow$ N, ya que entonces estaríamos diciendo que ¡no natural es un natural! La solución es añadirlo como un nuevo conjunto en la especificación formal:

CONJUNTOS

...

NoN Conjunto formado por el elemento NoNatural

SINTAXIS

predecesor : N $\rightarrow$ N $\cup$ NoN

producto : N $\times$ N $\rightarrow$ N

potencia : N $\times$ N $\rightarrow$ N $\cup$ NoN

Los axiomas de la semántica deben ser los necesarios y suficientes para definir de forma completa las tres operaciones. Para ello, debemos relacionarlos con los dos constructores del tipo: cero y sucesor.

SEMÁNTICA

$\forall$ n, m $\in$ N

9. predecesor(cero) = NoNatural
10. predecesor(sucesor (n)) = n
11. producto(cero, n) = cero
12. producto(sucesor(n), m) = suma(m, producto(n, m))
13. potencia(cero, cero) = NoNatural
14. potencia(cero, sucesor(n)) = cero
15. potencia(sucesor(n), cero) = sucesor (cero)
16. potencia(sucesor(n), sucesor(m)) = producto(sucesor(n), potencia(sucesor(n), m))

Para la operación potencia son necesarios cuatro axiomas para especificarla completamente. La regla 16, escrita con la notación habitual dice que $(n+1)^{m+1} = (n+1)(n+1)^m$, y la regla 12 diría que $(n+1)m = m+n\ m$. De esta forma, la operación potencia se define a través de producto, producto se define a través de suma y, finalmente, como ya vimos, suma se define a través del constructor sucesor.

Ejercicio 2.4 Con la especificación del ejercicio 2.3, comprueba el resultado que se obtiene para las siguientes expresiones:

a) igual(predecesor(sucesor(cero)), sucesor(predecesor(cero)))

b) producto(sucesor(sucesor(cero)), sucesor(sucesor(cero)))

Solución.

a) igual(predecesor(sucesor(cero)), sucesor(predecesor(cero)))

igual(predecesor(sucesor(cero)), sucesor(predecesor(cero))) = [Axioma 10, con n=cero]

igual(cero, sucesor(predecesor(cero)) = [Axioma 9]

igual(cero, sucesor(NoNatural))

ERROR en la expresión. La operación sucesor espera un parámetro de tipo Natural.

b) producto(sucesor(sucesor(cero)), sucesor(sucesor(cero)))

Para simplificar las expresiones, cambiaremos sucesor por s y cero por c.

producto(s(s(c)), s(s(c))) = [Axioma 12, con n= s(c), m= s(s(c))]
suma(s(s(c)), producto(s(c), s(s(c)))) = [Axioma 12, con n= c, m= s(s(c))]
suma(s(s(c)), suma(s(s(c)), producto(c, s(s(c))))) = [Axioma 11, con n= s(s(c))]
suma(s(s(c)), suma(s(s(c)), c)) = [Axioma 4, con n= s(c), m= c]
suma(s(s(c)), s(suma(s(c), c))) = [Axioma 4, con n= c, m= c]
suma(s(s(c)), s(s(suma(c, c)))) = [Axioma 3, con m= c]
suma(s(s(c)), s(s(c))) = [Axioma 4, con n= s(c), m= s(s(c))]
s(suma(s(c), s(s(c)))) = [Axioma 4, con n= c, m= s(s(c))]
s(s(suma(c, s(s(c))))) = [Axioma 3, con m= s(s(c))]
s(s(s(s(c))))

Ejercicio 2.5 Evaluar el resultado de las siguientes expresiones, usando la especificación formal por el método axiomático del TAD Pila[T], estudiada en el apartado 2.3.5.

a) esVacía(pop(crearPila))
b) top(pop(pop(push(a, push(b, push(c, crearPila))))))
c) esVacía(pop(push(a, push(x, crearPila))))

Solución.

a) esVacía(pop(crearPila))
esVacía(pop(crearPila) = [Axioma 3]
esVacía(crearPila) = [Axioma 5]
verdadero

b) top(pop(pop(push(a, push(b, push(c, crearPila))))))
top(pop(pop(push(a, push(b, push(c, crearPila))))))) = [Axioma 4, con t= a, s= push(b, push(c, crearPila))]
top(pop(push(b, push(c, crearPila)))) = [Axioma 4, con t= b, s= push(c, crearPila)]
top(push(c, crearPila)) = [Axioma 2, con t= c, s= crearPila]
c

c) esVacía(pop(push(a, push(x, crearPila))))
esVacía(pop(push(a, push(x, crearPila)))) = [Axioma 4, con t= a, s= push(x, crearPila)]
esVacía(push(x, crearPila)) = [Axioma 6, con t= x, s= crearPila]
falso

Ejercicio 2.6 Proponer algún modelo subyacente adecuado para escribir la especificación formal por el método constructivo del TAD Conjunto[T]. Escribir la parte de semántica para las operaciones: Vacío, EsVacío, Inserta, Suprime, Miembro, Unión.

Solución.

Un posible modelo subyacente podría ser el TAD Lista[T], definido por el método axiomático en el apartado 2.4.3. Este tipo tiene las operaciones: crearLista, esVacía, formarLista, concatenar, cabeza, cola, primero y último. La especificación por el método constructivo podría ser la siguiente.

NOMBRE

Conjunto[T]

CONJUNTOS

C Conjunto de conjuntos de elementos de tipo T
T Conjunto de elementos que pueden ser almacenados
N Conjunto de los números naturales

B Conjunto de valores booleanos

SINTAXIS

Vacío : → C
Inserta : C × T → C
EsVacío : C → B
Suprime : C × T → C
Miembro : C × T → B
Unión : C × C → C

SEMÁNTICA

∀ c, d ∈ C; ∀ t, p ∈ T
pre-Vacío()::= verdadero
post-Vacío(;c)::= c = crearLista
pre-Inserta(c, t)::= verdadero
post-Inserta(c, t; d)::= d= (SI Miembro(c, t) ⇒ c | concatenar(c, formarLista(t)))
pre-Suprime(c, t)::= verdadero
post-Suprime(c, t; d)::= d = (SI esVacía(c) ⇒ c
| SI (primero(c)=t) ⇒ cola(c) | Inserta(Suprime(cola(c), t), primero(t)))
pre-Miembro(c, t)::= verdadero
post-Miembro(c, t; b)::= b = (SI esVacía(c) ⇒ falso
| (primero(c)=t) OR Miembro(cola(c), t))
pre-Unión(c, d)::= verdadero
post-Union (c, d; e)::= e = (SI esVacía(d) ⇒ c
| SI Miembro(c, primero(d)) ⇒ Union(c, cola(d))
| concatenar(formarLista(primero(d)), Union(c, cola(d))))

Aun siendo correcta, el inconveniente de la especificación anterior se encuentra en que es muy próxima a lo que sería una implementación del tipo Conjunto, sugiriendo una estructura de representación mediante listas. No obstante, cualquier implementación válida de los conjuntos será coherente con la anterior especificación, use listas o no.

Ejercicio 2.7 A la definición formal axiomática del TAD Conjunto[T], del ejercicio 2.1, queremos añadir las operaciones: EsSubcjto, EsSubPropio y EsIgual, que dados dos conjuntos comprueban si uno es subconjunto del otro, si uno es subconjunto propio del otro o si ambos son iguales, respectivamente. Escribe la sintaxis y la semántica de las operaciones.

Solución.

Igual que en algunos de los ejemplos anteriores, es posible simplificar la semántica usando operaciones que ya estén definidas. De esta forma, bastaría con definir EsSubcjto en base a los constructores. Las operaciones EsSubPropio y EsIgual se pueden definir con un sólo axioma, usando la anterior operación.

...

SINTAXIS

EsSubcjto : C × C → B
EsSubPropio : C × C → B
EsVacío : C × C → B

SEMÁNTICA

∀ c1, c2 ∈ C; ∀ t ∈ T

EsSubcjto(Vacío, c1) = verdadero
EsSubcjto(Inserta(t, c1), c2) = Miembro(t, c2) AND EsSbcjto(c1, c2)
Igual(c1, c2) = EsSubcjto(c1, c2) AND EsSubcjto(c2, c1)
EsSubPropio(c1, c2) = EsSubcjto(c1, c2) AND NOT Igual(c1, c2)

Ejercicio 2.8 Comprueba qué resultado se obtiene para las siguientes expresiones, utilizando la especificación por el método constructivo del TAD Cola[T], estudiada en el apartado 2.4.3.

a) frente(inserta(4, inserta(5, crearCola)))
b) esVacíaCola(resto(inserta(a, crearCola)))
c) inserta(frente(crearCola), crearCola))

Solución.

a) frente(inserta(4, inserta(5, crearCola)))

Operación	**Precondición**	**Postcondición**
1. crearCola	verdadero	o1= crearLista
2. inserta(5, o1)	verdadero	o2= concatenar(o1, formarLista(5))
3. inserta(4, o2)	verdadero	o3= concatenar(o2, formarLista(4))
4. frente(o3)	not esVacía(o3) = [Ax. 18] not (esVacía(o2) AND esVacía(formarLista(4))) = [Ax. 17] not (esVacía(o2) AND falso) = not falso = verdadero	o4= primero(o3)= primero(concatenar(o2, formarLista(4))) = [Ax. 3] primero(o2) = primero(concatenar(crearLista, formarLista(5))) = [Ax. 3] primero(formarLista(5)) = [Ax. 2] 5

b) esVacíaCola(resto(inserta(a, crearCola)))

Operación	**Precondición**	**Postcondición**
1. crearCola	verdadero	o1= crearLista
2. inserta(a, o1)	verdadero	o2= concatenar(o1, formarLista(a))
3. resto(o2)	not esVacía(o2) = [Ax. 18] not (esVacía(o1) AND esVacía(formarLista(a))) = [Ax. 16,17] not (verdadero AND falso) = not falso = verdadero	o3 = cola(o2) = cola(concatenar(o1, formarLista(a))) = [Ax. 12] cola(formarLista(a)) = [Ax. 11] crearLista
4. esVacíaCola(o3)	verdadero	o4= esVacía(o3) = esVacía(crearLista) = [Ax. 12] verdadero

c) inserta(frente(crearCola), crearCola))

Operación	**Precondición**	**Postcondición**
1. crearCola	verdadero	o1= crearLista
2. frente(o1)	not esVacía(o1) = not esVacía(crearLista) = [Ax. 16] not verdadero = falso	**ERROR en la expresión frente(crearCola)**. No se cumple la precondición. Ejecución interrumpida.

Ejercicios propuestos

Ejercicio 2.9 Suponiendo la especificación formal algebraica vista en el apartado 2.3.3 para el TAD Natural (con las operaciones: cero, sucesor, esCero, igual y suma), escribe la sintaxis y la semántica correspondientes a las operaciones de comparación entre naturales: esMenor, esMenorIgual, esMayor, esMayorIgual; y las operaciones Máximo y Mínimo.

Ejercicio 2.10 Se define el TAD Bolsa[T] como un conjunto de elementos de tipo T donde un elemento puede ser insertado más de una vez. Escribe una especificación algebraica y otra constructiva para el tipo Bolsa[T], con las operaciones: bolsaVacía, inserta, suprime, esVacía, numVeces (devuelve el número de veces que un elemento ha sido insertado en la bolsa) e igual (comprueba si dos bolsas son iguales).

Ejercicio 2.11 Considera la siguiente especificación formal algebraica del TAD CMT (contador módulo tres). Las operaciones definidas son: Nuevo : $\rightarrow$ CMT; Inc : CMT $\rightarrow$ CMT; Dec : CMT $\rightarrow$ CMT. Los axiomas de la parte Semántica son los siguientes:

$\forall$ c $\in$ CMT

1. Inc(Dec(c)) = c
2. Dec(Inc(c)) = c
3. Inc(Inc(Inc(c))) = c
4. Dec(Dec(Dec(c))) = c

Demuestra, usando los axiomas de la especificación formal (e indicando el número de los que se usen), que se cumplen las dos siguientes propiedades:

a) Dec(Nuevo) = Inc(Inc(Nuevo))

b) Inc(c) = Dec(Dec(c)), $\forall$ c $\in$ CMT

Ejercicio 2.12 Usando la especificación de los ejercicios 2.1 y 2.2, aplicada sobre el tipo Conjunto[Natural], escribe la expresión correspondiente a crear un conjunto con los elementos {5, 2}. Sobre ese conjunto, aplica las operaciones: máximo, mínimo, sucesor(C, 22), predec(C, 6) y deduce el resultado usando los axiomas.

Ejercicio 2.13 Construye una especificación formal algebraica para el TAD genérico Lista[T] en la cual los constructores sean las operaciones: listaVacía (crea una lista vacía) e inserta (añade un elemento al principio de la lista). Añade las operaciones: primero, último, cola, concatenar y longitud.

Ejercicio 2.14 Queremos añadir a la especificación algebraica del tipo Lista[T] las operaciones: invertirLista(l) y elementoEn(l, n) (devuelve el elemento de la lista l insertado en la posición n-ésima). Escribe la sintaxis y la semántica de las operaciones anteriores, tomando como base las especificaciones algebraicas de listas del apartado 2.4.3 y la del ejercicio 2.13.

Ejercicio 2.15 Utilizando las especificaciones de los ejercicios 2.13 y 2.14, obtén el resultado de las siguientes expresiones. Se supone que l es una lista cualquiera.

a) elementoEn(cola(inserta(4, inserta(2, l))), 3)

b) primero(cola(invertirLista(inserta(7, listaVacía))))

c) invertirLista(invertirLista(inserta(a, inserta(b, listaVacía))))

Ejercicio 2.16 Construye la especificación formal algebraica del TAD Persona, que está formada por un registro con tres campos: nombre (de tipo cadena), dirección (de tipo cadena) y teléfono (de tipo entero). Incluye operaciones de consulta y modificación de cada uno de ellos. Muestra varias expresiones de ejemplo, donde se establezcan varios valores y luego se consulten algunos de ellos. ¿Cuáles son los constructores del tipo?

Ejercicio 2.17 En una especificación formal constructiva del tipo Array[T], tenemos una operación Ordena : A → A, que dado un array devuelve sus elementos ordenados de menor a mayor. Escribe la semántica para esta operación, indicando las precondiciones y postcondiciones adecuadas. Se supone que existen otras operaciones del tipo: Tamaño : Array → Entero; ObtenValor : Array × Entero → T; PonValor : Array × Entero × T → Array; MenorQue : T × T → B.

Ejercicio 2.18 Construye una especificación formal axiomática para el TAD Array[T]. Los índices del array son enteros y el rango de índices del array no está limitado. Las operaciones sobre el TAD son: Nuevo, PonValor, ObtenValor (para escribir o leer un valor en una posición del array, respectivamente) y Máximo (para obtener el índice máximo de array). Adopta las decisiones que creas adecuadas para los posibles casos de error. ¿Cuáles son los constructores del tipo y las operaciones de modificación y consulta?

Ejercicio 2.19 A la definición formal constructiva del TAD Conjunto[T], del ejercicio 2.6, queremos añadir las operaciones: EsSubcjto, EsSubPropio y EsIgual, que dados dos conjuntos comprueban si uno es subconjunto del otro, si uno es subconjunto propio del otro o si ambos son iguales, respectivamente. Escribe la sintaxis y la semántica de las operaciones, utilizando las pre- y postcondiciones adecuadas.

Ejercicio 2.20 Tenemos definido el TAD Pantano, por el método axiomático, con las operaciones:

Nuevo : N → Pantano. Devuelve un pantano con capacidad máxima N y cantidad actual de agua 0.

Llenar : Pantano → Pantano. Pone la cantidad actual de agua del pantano al valor de capacidad máxima.

Cantidad : Pantano → N. Devuelve la cantidad actual de agua del pantano

Transvasar : Pantano × N → Pantano ∪ Error. Decrementa la cantidad actual en N, siempre que sea posible.

Ocupación : Pantano → R. Devuelve el porcentaje actual de ocupación del pantano.

Escribe una especificación formal para este tipo abstracto. Se puede utilizar el método algebraico o el constructivo. Si lo crees conveniente puedes añadir otras operaciones.

Ejercicio 2.21 Evalúa el resultado de las siguientes expresiones sobre pilas, usando la especificaciones formales constructivas del apartado 2.4.3 y la especificación algebraica del apartado 2.3.5.

a) esVacía(pop(push(t1, pop(push(t2, crearPila)))))

b) tope(pop(pop(push(t3, crearPila))))

Ejercicio 2.22 Construye una especificación formal axiomática para el TAD ArbolBinario[T]. El tipo debe contener las operaciones: CrearArbol, Construir (dados dos árboles I y D, y un elemento x, devuelve un nuevo árbol donde x es la raíz y los subárboles izquierdo y derecho son I y D, respectivamente), Raíz (devuelve la raíz), HijoIzq (devuelve el hijo izquierdo), HijoDer y EsVacío.

Ejercicio 2.23 Partimos de las especificaciones formales algebraicas del TAD Lista[T], del apartado 2.4.3, y ArbolBinario[T], del ejercicio 2.22. Añade a este último las operaciones OrdenPrevio, OrdenSimétrico y OrdenPosterior que, dado un árbol como argumento, devuelven una lista que contiene todos los elementos del árbol ordenados según su recorrido en orden previo, simétrico y posterior, respectivamente.

Ejercicio 2.24 Definimos el TAD ColaCíclica[T] como una cola FIFO usual (primero en entrar, primero en salir), pero en la cual al sacar un elemento de la cabeza, se vuelve a meter en la cola en la última posición. Las operaciones del tipo son: Crear, Meter (mete un elemento en la última posición de la cola), Cabeza (devuelve el elemento que está en la cabeza de la cola) y Avanzar (el elemento de la cabeza pasa al final de la cola). Escribe una especificación formal axiomática del tipo ColaCíclica[T]. Si lo crees conveniente puedes incluir otras operaciones (en cuyo caso deberás especificarlas también).

Ejercicio 2.25 Queremos definir el TAD genérico GrafoDirigido[T] por el método axiomático, con las operaciones: GrafoVacío (crea un nuevo grafo sin vértices), InsArista (añade una arista al grafo, con los dos vértices pasados como parámetros), ExisteArista (comprueba si existe una arista en el grafo) y GradoEntrada (devuelve el grado de entrada de un vértice dado). Escribe la especificación formal axiomática del tipo. Tener en cuenta lo siguiente:

- Los vértices son del tipo genérico T.
- No existe una operación para insertar vértices, ya que se supone que son añadidos implícitamente en la operación de insertar aristas (si no existen ya).
- Si se inserta una arista que ya existe, no ocurre nada ya que sólo puede existir una arista entre un par de vértices.

Cuestiones de autoevaluación

Ejercicio 2.26 Uno de los resultados fundamentales de utilizar abstracciones es la distinción entre especificación e implementación. ¿Qué diferencias existen entre una especificación y una implementación? ¿Qué relaciones de dependencia existen entre ambas, es decir cuál va antes y por qué? Indica las ventajas de tener una buena especificación de un tipo o procedimiento, según se use una notación formal o informal.

Ejercicio 2.27 Para una aplicación que utiliza colas de enteros, representadas mediante listas enlazadas, definimos la especificación formal del TAD ColaEnteros por el método axiomático. ¿Qué partes de la especificación formal deberíamos cambiar si en otra aplicación queremos representar las colas mediante arrays? Responde la misma cuestión suponiendo que se usa el método constructivo de especificación.

Ejercicio 2.28 ¿Qué diferencias existen entre un TAD mutable y uno no mutable? ¿Es posible desarrollar una especificación formal algebraica de un TAD mutable? ¿Por qué? ¿Y si la especificación formal es constructiva u operacional?

Ejercicio 2.29 ¿Qué ocurre en una especificación formal constructiva si en una expresión no se cumple la precondición? ¿Puede ocurrir algo parecido en una especificación por el método algebraico? ¿Qué ocurre si no se cumple la postcondición? ¿De quién es la responsabilidad en cada caso?

Ejercicio 2.30 Sea un TAD cualquiera A, definido mediante una especificación formal algebraica. ¿Qué propiedad cumplen las operaciones que son los constructores del tipo? ¿Qué otros tipos de operaciones pueden existir? ¿Por qué es importante esta distinción?

Ejercicio 2.31 Supón que desarrollamos la especificación algebraica del TAD genérico Array[T], incluyendo una operación para ordenar los elementos de un array. ¿Cómo se podría expresar con los axiomas el hecho de que el resultado debe ser un array ordenado?

Ejercicio 2.32 La abstracción de iteradores aparece como un tipo de abstracción diferente a las abstracciones de datos y las funcionales. ¿Podrían considerarse como un caso especial de abstracción funcional? ¿Cómo se puede implementar abstracciones de iteradores usando lenguajes como Pascal o C?

Ejercicio 2.33 En el formato de notación para la especificación informal de abstracciones de datos, del apartado 2.2.2, la cabecera tiene la forma: **TAD** nombre **es** lista_operaciones. El uso de **es**, en lugar de **tiene** u **operaciones**, intenta resaltar el hecho de que el tipo *sólo* debe ser usado mediante las operaciones definidas con el mismo. Por ejemplo, si el tipo es un registro no se puede acceder a sus campos directamente. ¿Qué implicaciones tiene esto en la implementación de un TAD utilizando clases? ¿Qué partes del tipo deberían ir en la parte `public` y cuáles en la parte `private`?

Ejercicio 2.34 ¿Cómo podemos asegurar que una especificación formal algebraica es válida, es decir no contiene contradicciones y es completa? ¿Y en el caso de la especificación constructiva?

Referencias bibliográficas

El concepto de abstracción y los mecanismos y tipos de abstracciones se encuentran en la mayoría de los libros de algoritmos y estructuras de datos, como por ejemplo el capítulo 1 de [Aho88]. Una descripción más extensa puede consultarse en el capítulo 2 de [Collado87], donde aparece también el formato de especificación informal de abstracciones funcionales y de datos, y el método de especificación axiomático.

Para profundizar en las especificaciones algebraicas se puede utilizar [Franch94], donde se utiliza una aproximación bastante formal al estudio de los TAD. En cuanto a las especificaciones constructivas, la idea de las pre- y postcondiciones como contratos surge de [Meyer99], y es descrita en los capítulos 11 y 12. En este libro se encuentran las bases y los principios de la programación orientada a objetos, aunque en general resulta algo excesivo para el nivel aquí tratado.