Diseño jerárquico de tipos abstractos de datos

Algoritmos y Estructuras de Datos II, DC, UBA.

Primer cuatrimestre de 2016

1. Diseño jerárquico de tipos abstractos de datos

En la etapa de especificación de problemas, lo único que hemos hecho es detallar *qué* debemos hacer, pero no nos hemos preocupado por *cómo* hacerlo. El objetivo fue describir el comportamiento del problema a resolver, pero no interesaba determinar cómo lo resolveríamos. Al *especificar* estamos describiendo el problema, recién al *diseñar* comenzamos a resolverlo.

Al diseñar, centraremos nuestro interés tanto en el ámbito en el que será usado el *tipo abstracto de datos* –de ahora en más sólo *tipo*–, como en los aspectos que se necesitan optimizar de este tipo (espacio, tiempo de ejecución). Del estudio de estos temas podremos determinar, por ejemplo, en qué orden llegarán los datos, cómo se los consultará, o las operaciones más frecuentemente usadas. A su vez, ocasionalmente, puede haber requerimientos explícitos de eficiencia temporal o espacial. Sobre la base de esta información, a la que llamaremos *contexto de uso*, diseñaremos nuestro tipo aprovechando las ventajas que nos ofrezca y cuidando de responder a los requisitos que nos plantea.

Un *tipo* se define por sus *funciones*, antes que por sus *valores*. La forma en que los valores se representan es menos importante que las funciones que se proveen para manipularlos. Los generadores de los tipos describen la forma abstracta de construir elementos, nunca la forma de construirlos o representarlos físicamente.

Con el propósito de implementar un tipo, deberemos:

- proveer una representación para sus valores,
- definir las funciones del tipo en función de las de su representación,
- demostrar que las funciones implementadas satisfacen las relaciones especificadas en los axiomas.

Para cumplir con estas condiciones, el diseñador es libre de elegir entre diferentes representaciones, ponderando eficiencia y simplicidad en el marco del *contexto de uso*.

En la etapa de diseño es en donde realmente comenzaremos a aprovechar el nivel de abstracción del modelo especificado, al buscarle representaciones menos abstractas. Cuanto más abstracto sea éste, más opciones de diseño tendremos disponibles en cada paso de diseño. Es decir, debemos tener presente la necesidad de, evitar cualquier decisión de especificación que restrinja innecesariamente nuestra libertad de acción en la etapa de diseño.

Básicamente, nuestra metodología de diseño partirá entonces de un modelo abstracto (nuestra especificación) no implementable directamente en un lenguaje imperativo de programación y aplicará iterativamente sobre dicho modelo sucesivos pasos de refinamiento (desabstracciones) hasta llegar a estructuras que sí son implementables.

Cada iteración de este proceso definirá un *nivel* de nuestro diseño. Por su parte, cada uno de estos niveles tendrá asociado uno o más *módulos de abstracción*, que básicamente indicarán cómo se resuelven las operaciones de un módulo utilizando otras de módulos del nivel inmediato inferior.

Cada uno de los módulos de abstracción resultantes de cada iteración *será implementable en un lenguaje de programación*, obteniendo de tal forma un diseño estratificado en niveles, donde los módulos de un cierto nivel son usuarios de los servicios que les brindan los del nivel inmediato inferior, y *no conocen* (ni usan) a los módulos de otros niveles. Un módulo dará a conocer los servicios que provee a través de una declaración de las operaciones que exporte donde, para cada una de ellas, se indica cuál es el estado esperado de la máquina antes de ejecutarse la operación (a lo que llamaremos *precondición*) y cómo incidirá esta ejecución en ese estado (a lo que llamaremos *postcondición*). Esta información se incluye en una sección llamada *interfaz del módulo*.

Para utilizar un tipo no podrá accederse directamente a la estructura que lo represente, sino que se accederá al mismo a través de la interfaz que se le defina.

Cualquier cambio de implementación del nivel n será *transparente* al nivel superior $n+1$, siempre que el nivel n mantenga su interfaz. El módulo exporta al menos las mismas funciones que se exportaban antes, y la funcionalidad provista por las mismas no cambió, aunque puede haber mejorado su performance. Podremos verificar la validez del cambio de diseño viendo que la veracidad de las precondiciones y postcondiciones del nivel rediseñado se mantiene con respecto a la versión anterior.

La cantidad de niveles en los que se descompone el diseño de un tipo dependerá del criterio del diseñador. Por ejemplo, podríamos implementar el tipo conjunto directamente sobre la conocida lista de nodos con punteros, pero aquí deberíamos plantearnos: ¿no es acaso esta lista una *implementación* posible del tipo secuencia? Es decir, ¿no estamos en definitiva usando una secuencia para implementar el conjunto? Si es así, ¿por qué no dejamos explícita esta decisión en nuestro diseño? Entre otras cosas, no podemos asegurar que siempre sea la mejor implementación de secuencia, y en un futuro será quizás menos costoso sustituir la secuencia completa que la lista. En general, agregar niveles intermedios al diseño simplifica la resolución de los problemas, así como la legibilidad de las soluciones.

Nivel $n+1$: CONJUNTO_DE_NATURALES
↓
representado por
↓
Nivel n: SECUENCIA
↓
⋮
↓
representado por:
↓
Nivel 0: LISTA_DE_NODOS_CON_PUNTEROS

2. Paradigma imperativo

Para especificar formalmente el problema a resolver, escribimos el tipo abstracto de datos siguiendo el paradigma funcional. Sin embargo, al diseñar, debemos realizar un cambio de paradigma para poder expresar nuestra representación del modelo en un lenguaje imperativo. En esta sección discutiremos los principales aspectos que deberemos tener en cuenta al afrontar tal cambio.

2.1. Valores vs. objetos

Las aridades de las operaciones que definimos en la especificación para los tipos están en una notación funcional: se supone que construyen un objeto nuevo y lo devuelven al llamador. Una característica de esta notación es la *transparencia referencial*, esto es que una expresión siempre da el mismo resultado sin importar su contexto.

Nuestro método está orientado hacia lenguajes imperativos, y en éstos la situación es muy distinta. En primer lugar no toda interfaz funcional es la adecuada para usar o para ser implementada.

En el paradigma funcional los datos sólo tienen sentido en cuanto sean argumentos o resultados de funciones, sin embargo, en el paradigma imperativo, los datos son tratados como entidades independientes de las funciones que los utilizan. Es usual que se trabaje con una instancia de un objeto que se va modificando y cuyos valores anteriores no interesen. Por lo tanto, por cuestiones de optimización y uso, no tiene sentido construir cada vez un objeto nuevo para devolverlo como resultado de una función. Por ejemplo, si tenemos una secuencia y se le inserta un elemento, probablemente no se devuelva una copia de la secuencia con el nuevo elemento sino que se modificará la secuencia original.

2.2. Parámetros que se modifican

El mapeo de los parámetros de las funciones del tipo en operaciones del módulo no siempre es uno a uno. Por ejemplo, puede ser que la interfaz haya agregado requerimientos en el contexto de uso para que la operación Agregar devuelva además si el elemento ya existía, para poder, en el caso que así fuese, exhibir un mensaje de error. En tal caso, la operación quedaría declarada de la siguiente manera:

$$\text{AGREGAR}(\textbf{in/out}\ C : \texttt{conj(nat)}, \textbf{in}\ n : \texttt{nat}) \rightarrow res\ : \texttt{bool}$$

donde *resultado* devuelve *true* si el elemento ya estaba antes de efectuar el agregar.

Vimos cómo declarar las operaciones de los módulos, veamos ahora cómo invocarlas. Por ejemplo, si C es una variable de género `conj(nat)`, podemos agregarle el 4 con AGREGAR(C, 4).

En general, para invocar una operación que devuelve un valor se escribe:

valor_retornado ← operación (*parametros_necesarios*)

3. Lenguaje de diseño

Éste es el lenguaje con el que contamos para diseñar nuestros módulos.

3.1. Tipos disponibles

Los tipos de datos del lenguaje de diseño pueden definirse de la siguiente forma:

tipo_dato::=

`bool`	o
`nat`	o
`int`	o
`real`	o
`char`	o
`string`	o
`género`	o
`puntero`(tipo_dato)	o
`arreglo`[*nat*] de tipo_dato	o
`tupla`⟨$campo_1$: tipo_dato × ... × $campo_n$: tipo_dato⟩	o
`arreglo_dimensionable`[1] de tipo_dato	

donde `género` es un género que será definido en un módulo de abstracción, n es un número natural (constante) y los $campo_i$ son nombres de campos.

Los tipos incluidos en la lista anterior en general no serán diseñados (siempre que cumplan con los requerimientos de eficiencia planteados), aunque sí lo serán los tipos que los utilicen.

3.2. Declaración de operaciones

Para declarar operaciones utilizaremos el siguiente esquema:

$$\text{FUNCIÓN}(\textbf{PasajeArg}_1\ arg_1 : \texttt{TipoArg}_1, \ldots, \textbf{PasajeArg}_n\ arg_n : \texttt{TipoArg}_n) \rightarrow res\ : \texttt{TipoRes}$$

donde	indica:
FUNCIÓN	el nombre de la operación.
PasajeArg$_i$	el tipo de pasaje de parámetro (ver explicación más abajo) para el i-ésimo parámetro.
arg_i	el nombre del i-ésimo parámetro.
`TipoArg`$_i$	el tipo de dato del i-ésimo parámetro.
res	el nombre de la variable que guardará el resultado de la operación.
`TipoRes`	el tipo de dato de la variable que guardará el resultado de la operación.

3.3. Pasaje de parámetros

Los parámetros de una operación pueden ser

[1] Un arreglo dimensionable es aquél en el que el tamaño se especifica en momento de ejecución, pero no puede ser cambiado luego.

- *de entrada.* El valor es usado para efectuar cálculos, pero no es posible modificarlo (tiene una semántica similar al *const* de C++). Se denota anteponiendo al nombre de la variable en el tipo de la operación el símbolo "**in**".
- *de salida.* El valor se genera en la operación, y se almacena en el parámetro formal con el que se invocó a la función. Se lo denota con "**out**". La variable resultado también es de salida.
- *de entrada-salida.* Combina los dos conceptos anteriores. Se denota con "**in/out**".

Así, por ejemplo, la operación llamada SUMAR que toma dos naturales en dos variables, llamadas a y b, y devuelve el resultado en una variable de tipo natural llamada res, se debería declarar de la siguiente manera:

SUMAR(**in** a : `nat`, **in** b : `nat`) $\rightarrow$ res : `nat`

En cualquier caso se deberá tener en cuenta que los parámetros de tipos **primitivos** (`bool`, `nat`, `int`, `real`, `char`, `puntero`) y sólo éstos, siempre se pasan **por valor** y los de tipos **no primitivos** siempre se pasan por **referencia**. Notar que los arreglos dimensionables y estáticos se pasan por referencia a pesar de considerarse que ya vienen diseñados como parte del lenguaje (ver apéndice A.3). En el caso de las tuplas, cada una de sus componentes se pasa por referencia o por copia según sea un tipo primitivo o no. Si fuera necesario construir una copia de un parámetro de tipo no primitivo, dicha copia debe ser explícita: SUMAR(COPIAR(a), COPIAR(b)). Todo módulo que se diseñe y cuyas instancias se desee copiar deberá proveer una función a tal efecto.

3.4. Asignación

La expresión $A \leftarrow B$ (siendo A y B variables de un mismo tipo), denota la asignación del valor de B a la variable A. Esto funciona del mismo modo que el pasaje de parámetros. Es decir, si A y B pertenecen a un tipo primitivo, A pasará a ser una copia de B; y en caso contrario, después de la asignación las variables A y B harán referencia a la misma estructura física (generando *aliasing*).

4. El método

Presentemos los distintos pasos del método.

Vimos que nuestro objetivo es obtener un diseño jerárquico y modular. Por ello, el método que veremos tiene las nociones de los distintos niveles en la jerarquía. Cada uno de los niveles tendrá asociado un módulo de abstracción. Para ser más precisos, habrá distintos tipos abstractos de datos que deberemos diseñar. A cada uno de ellos le corresponderá un módulo de abstracción.

A grandes rasgos, el método se compone de los siguientes pasos:

- Elección del tipo abstracto de datos a diseñar.
- Módulo de abstracción para el tipo abstracto de datos elegido.
- Iteración o finalización.

Aplicaremos una iteración del método para un ejemplo. El ejemplo en cuestión es conjunto de naturales. Para poder dar el ejemplo, incluimos una especificación del tipo que diseñaremos (que sería el resultado obtenido en la etapa de especificación de un proyecto).

TAD CONJ(NAT)

usa	BOOL, NAT
géneros	conj(nat)
exporta	conj(nat), $\bullet \in \bullet$, $\emptyset$, Ag, $\emptyset?$, mínimo
observadores básicos	

• $\in$ • : nat $\times$ conj(nat) $\longrightarrow$ bool

generadores

$\emptyset$: $\longrightarrow$ conj(nat)

Ag : nat $\times$ conj(nat) $\longrightarrow$ conj(nat)

otras operaciones

• $-$ {•} : conj(nat) $\times$ nat $\longrightarrow$ conj(nat)

$\emptyset$? : conj(nat) $\longrightarrow$ bool

mínimo : conj(nat) c $\longrightarrow$ nat $\{\neg\emptyset?(c)\}$

axiomas los tradicionales

Fin TAD

4.1. Elección del tipo a diseñar

El orden en el cual se diseñan los tipos es arbitrario. Es, sin embargo, una buena práctica comenzar por los tipos *más importantes*, pues éstos serán los principales generadores de requerimientos de eficiencia para los tipos menos importantes (modalidad *top-down*).

Es importante notar que el proceso de diseño posee una natural ida y vuelta. Por ejemplo, la redefinición de las funciones de un tipo puede obligarnos a rever la sección representación de un tipo que basa su diseño en éste.

Cuando estemos diseñando un módulo, *no necesariamente* debemos diseñar todos los tipos que usamos en la especificación. Supongamos el siguiente ejemplo:

Cardinal : conj(α) $\longrightarrow$ nat

Donde *Cardinal* devuelve la cantidad de elementos de un conjunto. Una especificación posible de esta operación es:

Cardinal(c) $\equiv$ long(VolcarElementosEnSecuencia(c))

Donde *VolcarElementosEnSecuencia* es una operación que responde a la siguiente signatura y devuelve una secuencia con los elementos del conjunto, en algún orden.

VolcarElementosEnSecuencia : conj(α) $\longrightarrow$ secu(α)

Esta especificación, si bien no demasiado descriptiva, es válida. Ahora bien, en este caso nos debemos plantear si tiene sentido considerar en el diseño al tipo secuencia. Si la función *VolcarElementosEnSecuencia* se exportara, indicaría que *realmente* nos interesa ofrecer una función que vuelque el conjunto en una secuencia. En tal caso, no habrá más remedio que diseñar, en algún momento, el tipo secuencia. Pero si no se exporta, y la secuencia *no aparece visible* en ningún otro lado de la especificación, en principio no sería necesario diseñarla. Quizás la función Cardinal pueda implementarse de otra manera, sin pasar por la secuencia. Esto es una expresión de que la manera en la que se axiomatizó *Cardinal* no es importante, sino que solamente es importante lo que esos axiomas significan.

4.2. Módulo de abstracción para el tipo elegido

Cada módulo de abstracción está compuesto por dos secciones: la definición de la interfaz y la definición de la representación. En la interfaz se describe básicamente la funcionalidad del módulo y en qué contexto puede ser usada. En la representación se elige, bajo algún criterio, una forma de representación utilizando otros módulos y se resuelven las operaciones del módulo en función de su representación. Luego veremos con mucho más detalle cada una de estas secciones.

4.3. Iteración o finalización

En este punto tenemos un diseño que puede contener tipos para los que no tenemos una propuesta de diseño. Son, en realidad, otros problemas a resolver de nivel de abstracción menor al original. Por lo tanto, debemos volver a repetir el método con los nuevos tipos a diseñar.

La iteración prosigue hasta llegar a tipos que tengamos diseñados en nuestra biblioteca o sean primitivos. La reutilización de tipos ahorra tiempo de diseño. Es posible reutilizar tipos que fueron diseñados con criterios distintos a los que deseamos, con lo cual podríamos perder parte de la eficiencia buscada, lo que será tolerable siempre y cuando no rompa las restricciones planteadas por el contexto de uso.

5. Aspectos de la interfaz

En este paso tomamos las decisiones concernientes al *cambio de paradigma*. El método exige que redefinamos las aridades de las funciones adaptándolas a un lenguaje imperativo. También pide que explicitemos los requerimientos (precondiciones), para la aplicación de cada operación y los efectos que tiene sobre el estado de la máquina (postcondiciones). Para escribir las precondiciones y las postcondiciones usaremos un lenguaje de descripción de estados (ya conocido de Algoritmos y Estructuras de Datos I) aprovechando la especificación del tipo a diseñar.

El cambio de aridades no se limita al cambio de una función por un procedimiento o de la cantidad y tipo de parámetros que recibe. Podemos también decidir usar más de una operación para reemplazar alguna del tipo a diseñar o unir varias funciones en una sola. Por ejemplo, se podrían diseñar las operaciones Tope y Desapilar de una pila por una sola que saque el tope y lo devuelva. También pueden incluirse funciones que no tengan sentido desde el punto de vista abstracto pero sean útiles dentro del paradigma imperativo, por ejemplo funciones para copiar instancias del tipo o funciones como "comprimir" cuyos efectos sean visibles solo desde la eficiencia temporal y espacial de las operaciones, pero no signifiquen un cambio en el valor representado. Estas decisiones dependen del contexto de uso.

5.1. Servicios exportados

En esta sección deben estar expresados los detalles acerca de nuestro módulo que resulten indispensables a sus usuarios. Es imprescindible tocar temas como complejidad temporal de los algoritmos, y cuestiones de aliasing y efectos secundarios de las operaciones. Además, pueden exhibirse comentarios con respecto a la implementación del módulo que, aunque tengan menor importancia, sean de interés para el usuario.

No puede faltar la complejidad temporal de cada operación ya que, si faltase, nuestro módulo no podría ser utilizado por ningún otro módulo cuyo contexto de uso restringiera tal aspecto. Dicho de otro modo, si un usuario de nuestro módulo debe responder a requerimientos de eficiencia temporal, no podrá saber si realmente está cumpliendo con ellos a menos que sepa cuál es el orden de nuestras operaciones.

Por otro lado, el conocimiento pleno de los detalles de aliasing es de vital importancia para el uso correcto y eficiente del módulo. A modo de ejemplo, supongamos que estuviéramos implementando árboles binarios y necesitásemos una operación Podar, cuya declaración fuera la siguiente:

$$\text{PODAR}(\textbf{in } A : \texttt{ab}(\alpha), \textbf{out } I : \texttt{ab}(\alpha), \textbf{out } r : \alpha, \textbf{out } D : \texttt{ab}(\alpha))$$

que dado un árbol binario (no vacío) nos devolviera dos árboles (el hijo izquierdo y el derecho del anterior) y un elemento (su raíz).

Una forma posible de implementarla sería armar copias de los subárboles izquierdo y derecho de A, y devolverlas como I y D. Otra forma, más rápida, sería devolver en I y D referencias a los subárboles de A. En este último caso estaríamos provocando *aliasing* entre los árboles, causando que cualquier modificación que se realice sobre I o D, luego de llamar a la operación PODAR, repercuta en A.

Si no informásemos las cuestiones de aliasing referentes a esta operación podría pasar, por ejemplo, que, si hiciéramos lo primero, nuestro usuario, al no saber si I y D son estructuras físicas separadas de A, por si acaso no lo fueran hiciera copias de estos dos subárboles. De este modo, sólo por falta de información estaríamos deteriorando la eficiencia temporal del algoritmo del usuario. Por otro lado, si hiciésemos lo segundo y nuestro usuario supusiera lo contrario, si modificase I o D se modificaría A, causando desazón y odio hacia nuestra persona.

Es por ello que siempre debemos aclarar para cada operación si se produce o no aliasing al invocarla, como un efecto colateral.

También es importante, en las operaciones que quitan elementos de la estructura (como *fin* de Secuencia, *borrar* de Diccionario, $\bullet - \{\bullet\}$ de Conjunto, etc.), indicar si dichos elementos seguirán existiendo o serán eliminados. Esto afecta a los usuarios que tengan referencias a dichos elementos. Si se los elimina, las referencias a ellos dejarán de ser válidas. En caso contrario, seguirán vigentes pero ya no estarán en la estructura, por lo que será el usuario el encargado de liberarlas al momento de implementar su módulo.

5.1.1. Contexto de uso y requerimientos de eficiencia de los servicios exportados

¿En qué ámbito será usado el tipo? ¿Qué se necesita optimizar? En parte, dependerá de las operaciones más frecuentemente usadas, pero puede haber requerimientos explícitos (por ejemplo, "la operación mínimo debe tener $O(1)$"). ¿Cómo se construirán los objetos (por ejemplo, "los datos se ingresarán ordenados")? ¿Cómo se los consultará?

Ejemplo cotidiano: Debo armar una estantería. Puedo armarla de caña con estantes de vidrio, de madera lustrada (quebracho o algarrobo) o usar una de esas estanterías modulares de chapa. ¿Con qué la armo? Y, depende del *contexto de uso*, esto es: ¿la usaré para poner adornos de poco peso, la usaré como biblioteca o para apoyar herramientas? ¿La usaré para las tres cosas? Si es para libros, ¿cuántos serán y cuánto pesarán en el peor caso? ¿Qué necesito optimizar? ¿El costo de los materiales, la facilidad de armado, la estética o la resistencia del mueble?

Es decir, no se puede saber si un diseño es adecuado si no se aclara precisamente el contexto de uso. La idea es que la *principal justificación* para el resultado obtenido en cada iteración de diseño es el contexto de uso que se le impuso al diseñador.

En nuestro ejemplo, se necesita contar con un conjunto de naturales, finito pero en principio no acotado. Sabemos que el mencionado conjunto será utilizado como parámetro en dos algoritmos, uno de los cuales requerirá con frecuencia la obtención del mínimo elemento (se desea resolver esto en tiempo constante) y el otro preguntará frecuentemente por la pertenencia de los naturales que están en el intervalo $[n, n + 100]$, donde n será conocido antes de la carga de los datos. Los datos se incorporarán al conjunto en forma aleatoria, y el borrado de elementos se dará en situaciones excepcionales.

5.2. Interfaz

En esta sección se explicitan el tipo diseñado, cuáles son las operaciones exportadas y cuáles son las operaciones que existirán en el diseño, con sus aridades, sus precondiciones y sus postcondiciones.

5.2.1. Relación entre los objetos y la especificación

Al describir la interfaz de un módulo deberemos indicar, para cada una de las operaciones, cuáles son sus restricciones y qué efectos produce en el estado de la máquina. Para describir esto haremos uso de la especificación del tipo abstracto de datos asociado al módulo. Para ello veremos cuál es la relación que existe entre las variables del diseño y el tipo abstracto de datos especificado.

Retomemos el ejemplo del conjunto de naturales, y consideremos la operación AGREGAR.

AGREGAR(**in/out** c: `conj(nat)`, **in** n: `nat`)

Queremos poder describir en la interfaz que, una vez agregado el elemento, el conjunto resultante es el conjunto inicial más el nuevo elemento. Recordemos que en nuestra especificación teníamos declaradas las funciones $\bullet \in \bullet$ y Ag.

Quisiéramos decir que el estado inicial de la operación AGREGAR es

$\{c =_{\text{obs}} c_0\}$

y el estado final de la misma es

$\{c =_{\text{obs}} \text{Ag}(n, c_0)\}$

Tenemos un "problema de mundos" que solucionar. La variable c es del paradigma imperativo y por ende no puede ser comparada mediante $=_{\text{obs}}$ ya que su valor no está definido en base a axiomas. Lo mismo sucede con n, y por ende

no puede usarse en una operación de un tipo abstracto, como Ag. Sin embargo no hay problema con c_0, ya que es una *metavariable* que sí pertenece al paradigma funcional[2]. Para subsanar esta dificultad, presentaremos la función $\widehat{\ }$.

Llamaremos G_I al conjunto de géneros del paradigma imperativo y G_T al conjunto de géneros del paradigma funcional. Subindexaremos con I a los géneros de G_I y con T a los de G_T.

Disponemos de una función que dado un género del paradigma imperativo nos da su "equivalente" en el paradigma funcional[3]:

$$\widehat{\bullet} : G_I \to G_T$$

$\widehat{\ }$ define una familia de funciones –una para cada género en G_I–, que por abuso de notación llamaremos también $\widehat{\ }$ e identificaremos por contexto:

$$(\forall g \in G_I)(\widehat{\cdot} : g \to \widehat{g})$$

Es decir: $\widehat{conjunto_I} = conjunto_T$. Si c es de género $conjunto_I$ entonces $\widehat{c}$ es de género $conjunto_T$. Debemos tener en cuenta que en los géneros paramétricos, $\widehat{\ }$ se aplica recursivamente sobre el parámetro.

Ahora sí podemos declarar la interfaz de la operación Agregar:

Agregar(**in/out** c: `conj(nat)`, **in** n: `nat`)
$\{\widehat{c} =_{\text{obs}} c_0\}$
$\{\widehat{c} =_{\text{obs}} \text{Ag}(\widehat{n}, c_0\}$

Vemos que escribiendo las precondiciones y postcondiciones de cada operación del módulo podemos establecer de una forma clara el mapeo entre las operaciones del módulo y las funciones de la especificación.

Otro ejemplo. Supongamos que necesitamos implementar el tipo pila de naturales, y que en el contexto de uso nos han sugerido la necesidad de que la operación Desapilar no sólo desapile, sino que también devuelva el elemento desapilado. Esto se escribe:

Desapilar(**in/out** p: `pila_nat`) $\to$ res : `nat`
$\{\widehat{p} =_{\text{obs}} p_0 \wedge \neg \text{ vacia?}(p_0)\}$
$\{\widehat{p} =_{\text{obs}} \text{desapilar}(p_0) \wedge \widehat{res} =_{\text{obs}} \text{tope}(p_0)\}$

Haremos ahora una aclaración importante. Si bien lo presentado es formalmente correcto, nos permitiremos un abuso de notación que nos simplificará la escritura en nuestro uso cotidiano en la materia. Ésta consiste en no escribir el $\widehat{\ }$ y considerarlo implícito donde corresponda. De esta forma, los ejemplos anteriores se pueden escribir como:

Agregar(**in/out** c: `conj(nat)`, **in** n: `nat`)
$\{c =_{\text{obs}} c_0\}$
$\{c =_{\text{obs}} \text{Ag}(n, c_0)\}$

Desapilar(**in/out** p: `pila_nat`) $\to$ res : `nat`
$\{p =_{\text{obs}} p_0 \wedge \neg \text{ vacia?}(p_0)\}$
$\{p =_{\text{obs}} \text{desapilar}(p_0) \wedge res =_{\text{obs}} \text{tope}(p_0)\}$

NOTA: los tipos paramétricos se manejan de la misma manera que al especificar. Es decir, deben indicarse los géneros y operaciones que se utilizarán como parámetros formales, y lo que se requiera de ellos. En algunos casos es posible que se requieran más operaciones que en la especificación (por ejemplo, si la especificación no requería una relación de orden, pero los elementos se guardarán ordenados en la estructura).

Ahora sí, ya estamos en condiciones de presentar la interfaz para conjuntos genéricos.

Notar que la función Copiar no aparece en la especificación (no tendría sentido). Existen muchos casos en los que las funciones que se desea incluir en el diseño no coinciden exactamente con las especificadas, ya sea porque su funcionalidad carece de sentido en el mundo abstracto, porque se desea implementar no determinismo, o por otros motivos. En esos casos, se puede especificar funciones o predicados auxiliares para escribir las precondiciones y postcondiciones, o se pueden escribir las mismas recurriendo a una o más funciones de la especificación.

interfaz CONJ(α, $=_\alpha$, $<_\alpha$)

parámetros formales

[2] Al ser una metavariable también podríamos, por convención, asumir que pertenece al paradigma imperativo y tratarla de esa forma.
[3] La función se llama "sombrerito", y se escribe $\widehat{x}$.

géneros α

operaciones

$\bullet =_\alpha \bullet$	$: \alpha \times \alpha$	$\longrightarrow$	bool	Relación de equivalencia
$\bullet <_\alpha \bullet$	$: \alpha \times \alpha$	$\longrightarrow$	bool	Relación de orden total estricto

usa: BOOL[4]

se explica con: CONJUNTO(α)[5]

género: conj(α)

operaciones:[6]

VACÍO() $\rightarrow$ *res* : `conj`(α)
{true}
{$\widehat{res} =_{\text{obs}} \emptyset$}

AGREGAR(**in/out** C : `conj`(α), **in** n : α)
{$\widehat{C} =_{\text{obs}} C_0$}
{$\widehat{C} =_{\text{obs}} \text{Ag}(\widehat{n}, C_0)$}

BORRAR(**in/out** C : `conj`(α), **in** n : α)
{$\widehat{C} =_{\text{obs}} C_0$}
{$\widehat{C} =_{\text{obs}} C_0 - \{\widehat{n}\}$}

PERTENECE(**in** C : `conj`(α), **in** n : α) $\rightarrow$ *res* : `bool`
{true}
{$\widehat{res} =_{\text{obs}} \widehat{n} \in \widehat{C}$}

VACÍO?(**in** C : `conj`(α)) $\rightarrow$ *res* : `bool`
{true}
{$\widehat{res} =_{\text{obs}} \emptyset?(\widehat{C})$}

MÍNIMO(**in** C : `conj`(α)) $\rightarrow$ *res* : α
{$\neg\emptyset?(\widehat{C})$}
{$\widehat{res} =_{\text{obs}} \text{mín}(\widehat{C})$}

COPIAR(**in** S : `conj`(α)) $\rightarrow$ *res* : `conj`(α)
{true}
{$\widehat{res} =_{\text{obs}} \widehat{S}$}

fin interfaz

6. Representación

El objetivo de este paso es definir la forma en que representaremos el tipo que estamos diseñando en esta iteración. La elección de una forma de representación está dada por la elección de una o más estructuras, las cuales deberán estar debidamente justificadas. Además de elegir la estructura de representación, es necesario definir cuál es la relación entre la estructura de representación y el tipo representado. Por último, se deberán proveer los algoritmos que operan sobre la estructura y que resuelven cada una de las operaciones.

La estructura de representación de las instancias de los tipos sólo será accesible (modificable, consultable) a través de las operaciones que se hayan detallado en la interfaz del módulo de abstracción respectivo. Las operaciones no exportadas también tendrán acceso a esta información, pero sólo podrán ser invocadas desde operaciones del mismo módulo.

[4] Interfaces de módulos de abstracción que se usan.
[5] Tipos abstractos referidos en las precondiciones o en las postcondiciones de las operaciones.
[6] Todas las operaciones presentadas aquí se asumen exportadas por el módulo. Las precondiciones y postcondiciones de las funciones que no se exporten (auxiliares) deben escribirse junto a sus respectivos algoritmos en la sección de Algoritmos.

6.1. Estructura de representación

En esta sección se elegirá una estructura para representar el tipo y se indicará cómo dicha estructura será utilizada.

6.1.1. Elección de la estructura

Para esto, la idea es plantearnos formas de representación alternativas, y luego optar por una de ellas. Esta opción surge de tener en cuenta cuáles son las operaciones que nos interesan optimizar y en qué contexto de uso serán utilizadas. Además, es muy importante considerar criterios de optimización tales como: espacio de disco, espacio de memoria, tiempo de ejecución, reusabilidad, claridad y sencillez de la implementación, homogeneidad de los algoritmos, etc.

Las variables en un programa *referencian* valores. Será imposible el acceso a la representación interna de éstos, como veremos más adelante, y esto redundará en la *modularidad* de nuestro diseño y en el *ocultamiento de información*. El ocultamiento de información nos permite hacer invisibles algunos aspectos que serán encapsulados. Esto es útil para aumentar el nivel de abstracción y diseñar código que sea más fácilmente modificable, mantenible y extensible. Al acceder a los objetos sólo a través de su interfaz no nos atamos a su implementación, sólo a su funcionalidad. Cualquier cambio de implementación en un tipo que no altere la funcionalidad no nos obligará a rediseñar los tipos superiores.

En virtud del contexto de uso, se decide diseñar conjunto de naturales sobre la siguiente estructura:

> `conj(nat)` se representa con `tupla`⟨intervalo: `conj_acotado_nat`, los_demás: `conj_no_acotado_nat`⟩

Una nota sobre la sintaxis. Para representaciones más complicadas podemos usar "macros", declaradas mediante la palabra *es*:

> `ab_raro` se representa con `tupla`⟨punt: `puntero(nodo)`, lo_demás: `resto_estructura`⟩
>
> donde `nodo` es `tupla`⟨der: `puntero(nodo)`, izq: `puntero(nodo)`, valor: `real`⟩
> y
> `resto_estructura` es `tupla`⟨conj: `conjunto`, sec: `secuencia`⟩

6.1.2. Modalidad de uso

Pautas de por qué nos servirá la estructura, cómo aprovecharemos la información eventualmente redundante que mantendremos. "Tal componente de la estructura nos ayudará a optimizar tal operación, y deberemos tener cuidado de cuando se agregan datos hacer esto, esto y lo otro", ¿cómo usamos la estructura?, ¿por qué nos interesa mantener tal propiedad como invariante?, ¿qué haremos para preservar el invariante?, etc. No se trata de escribir los algoritmos ni el invariante en lenguaje natural, sino sólo de aclarar aquellas cosas que *facilitarán* su lectura. Es un resumen de para qué sirven y cómo se mantienen las estructuras.

El conjunto acotado mantiene los elementos que además pertenecen al intervalo. Aquellas operaciones que reciban un natural n como parámetro verificarán si este n está o no en el intervalo y, dependiendo de esto, resolverán la operación en función de los servicios brindados por el conjunto acotado o por el conjunto no acotado. Por ejemplo, al insertar verificaremos si el parámetro n está o no en el intervalo del conjunto acotado. Si es así, insertaremos allí, y si no, en el conjunto no acotado.

6.2. Justificación. Otras estructuras consideradas.

¿Por qué usamos esta estructura y no otra? ¿Pensamos en utilizar otras estructuras? ¿Por qué las descartamos? "Porque la otra ocupaba más memoria", "porque el tipo será necesario en más de un contexto y la estructura elegida fue una solución de compromiso para satisfacer aceptablemente ambas necesidades", "porque tiene una complejidad $O(n)$, mejor que tal otra estructura, que en el peor caso tiene $O(n^3)$", "porque es la única estructura, entre las analizadas, que resuelve la operación xxx con complejidad $O(n)$, tal como pide el contexto de uso", "porque será usado en un

sistema de tiempo real, y preferimos la predictibilidad del tiempo de respuesta antes que la mejor eficiencia de las otras estructuras", "porque la que elegimos es más fácil de implementar, y tiene un comportamiento levemente peor que la que descartamos", etc.

Por un lado tendremos almacenados los números del intervalo finito en un conjunto acotado (un nuevo tipo), y por el otro lado tendremos el resto de los números en un conjunto no acotado. La razón de esta separación es que le exigiremos al conjunto acotado un mejor comportamiento (mayor eficiencia) para resolver las operaciones de pertenencia y mínimo. Una instancia de conj(nat) sólo será un selector que, en virtud de los parámetros recibidos por sus operaciones, delegará la responsabilidad en alguno de estos dos tipos soporte.

6.3. Relación entre la representación y la abstracción

6.3.1. Invariante de representación

Es un predicado

$$\text{Rep} \ : \ \text{genero\_de\_}\widehat{\text{representacion}} \ \longrightarrow \ \textit{boolean}$$

que nos indica si una instancia del tipo de representación es válida para representar una instancia del tipo representado. Es el conocimiento sobre la estructura que necesitan las distintas operaciones para funcionar correctamente y que garantizan al finalizar. De alguna manera, es el concepto coordinador entre las mismas. Quedan expresados en él las restricciones de coherencia de la estructura, surgidas de la redundancia de información que pueda haber. Notemos que su dominio es la imagen funcional del tipo que estamos implementando. Esto es necesario para que podamos "tratar" los elementos del dominio en lógica de primer orden. Su imagen no es el género bool, sino los valores lógicos Verdadero y Falso. En general nos referiremos a ellos como *boolean*.

Por ejemplo, si implementamos conjunto de naturales como `conj(nat)` se representa con `estructura`, donde `estructura` es `tupla`⟨cantidad: `nat`, elementos: `secu(nat)`⟩, el invariante de representación es

$\text{Rep} \ : \ \text{estr}\widehat{\text{uctura}} \ \longrightarrow \ \textit{boolean}$

$(\forall e : \widehat{\text{estructura}})$

$\text{Rep}(e) \ \equiv \ ((e.\text{cantidad} = \text{long}(e.\text{elementos})) \equiv \text{true} \wedge \text{NoTieneDuplicados}(e.\text{elementos}) \equiv \text{true})$

Notemos algunos puntos importantes:

- *Rep* no toma un `conj(nat)`, toma la imagen "abstracta" (funcional, del mundo de los TADs) de su representación, es decir, una tupla abstracta.
- Cuando escribimos e.cantidad nos referimos al proyector ".cantidad" de la tupla abstracta.
- ¿Por qué escribimos $\equiv$ true? Porque e.cantidad = long(e.elementos) es una comparación utilizando la función $\bullet = \bullet \ : \ \text{nat} \times \text{nat} \ \longrightarrow \ \text{bool}$ del TAD Nat, que devuelve bool, no un valor de verdad. Lo mismo sucede con NoTieneDuplicados(e.elementos). Eso nos permite declarar NoTieneDuplicados axiomáticamente.
- También podríamos haber escrito: ((e.cantidad = long(e.elementos)) $\wedge$ NoTieneDuplicados(e.elementos)) $\equiv$ true. En ese caso el $\wedge$ hubiese sido la operación $\bullet \wedge \bullet \ : \ \text{bool} \times \text{bool} \ \longrightarrow \ \text{bool}$ del TAD Bool.
- SI SABEMOS LO QUE ESTAMOS HACIENDO, podemos cometer un *abuso de notación* y escribir:

  $\text{Rep}(e) \ \equiv \ ((e.\text{cantidad} = \text{long}(e.\text{elementos})) \wedge \text{NoTieneDuplicados}(e.\text{elementos}))$

  De hecho, ESTA FORMA SERÁ LA QUE PREFERIREMOS, para no recargar la notación.

El invariante de representación debe ser cierto tanto al comienzo de las operaciones exportadas del tipo como al final de las mismas. Por lo tanto, las operaciones del tipo deberán preservarlo, aunque quizás no sea cierto en algún estado intermedio del algoritmo que implemente alguna operación.

Continuando con el ejemplo de arriba, se da el caso de que:

```
Borrar(in/out C : conj(nat), in n : nat)
    //Aquí es cierto el invariante de representación.
    if Está?(C.elementos, n)
        C.cantidad ← (C.cantidad) - 1
        //Aquí ya no es cierto el invariante.
```

BORRAR(C.elementos, n)
endif
//Se reestableció el invariante, vuelve a ser verdadero.

Como antes, omitiremos el $\widehat{\ }$ del tipo de entrada, y supondremos *Rep* siempre toma un valor del tipo abstracto.

Volviendo al caso del conjunto, deberemos pedir que nuestros dos conjuntos (uno acotado, el otro no) no se solapen, ya que no hay ninguna necesidad de que así suceda y, aparte, se nos simplificarán algunas operaciones. Por ejemplo, para borrar un elemento sólo debemos borrarlo de una de las dos componentes. Formalmente:

Rep : tupla⟨intervalo: conj_acotado_nat, los_demas: conj_no_acotado_nat⟩ $\longrightarrow$ *boolean*

($\forall t$: tupla⟨intervalo: conj_acotado_nat, los_demas: conj_no_acotado_nat⟩)

$$\text{Rep}(t) \equiv (\forall n\text{: nat})(n \in t.\text{invervalo} \Rightarrow n \in \text{rango}(t.\text{intervalo})) \wedge (\forall n\text{: nat})(n \in t.\text{los\_demas} \Rightarrow n \notin \text{rango}(t.\text{intervalo}))$$

En el caso de que todas las instancias del tipo de representación sean válidas, simplemente se denota:

Rep : genero_de_representacion $\longrightarrow$ boolean

($\forall t$: genero_de_representacion) Rep(t) $\equiv$ true

Un invariante de representación muy difícil de escribir puede ser un síntoma de la falta de algún nivel intermedio en el diseño. En nuestro ejemplo, si en lugar del conjunto no acotado, habláramos directamente de una implementación del mismo, como puede ser una lista, sería complicado explicar la no-duplicación dentro de esa lista y del no-solapamiento entre esa lista y el conjunto acotado.

6.3.2. Función de abstracción

La función de abstracción tiene el siguiente tipo

Abs : genero_de_representacion $\widehat{\ }$ g $\longrightarrow$ genero_del_tipo_representado $\widehat{\ }$ {Rep(g)}

Es decir, tiene por dominio al conjunto de instancias que son la imagen abstracta del tipo de representación y que verifican el invariante de representación, y devuelve una imagen abstracta de la instancia del tipo representado (aquella instancia que estamos pretendiendo representar). Diremos que T *representa a* A si Abs(T) $=_{\text{obs}}$ A. Como usualmente, omitiremos el $\widehat{\ }$.

Por ejemplo, supongamos que decidimos implementar el tipo conjunto de naturales sobre secuencia de naturales. Deseamos que, por ejemplo,

Abs : secu(nat) s $\longrightarrow$ conj(nat) {Rep(s)}

Abs($1 \bullet 2 \bullet 3 \bullet <>$) $=_{\text{obs}}$ $\{1,2,3\}$ $=_{\text{obs}}$ Abs($2 \bullet 1 \bullet 3 \bullet <>$)

Algunas observaciones sobre la función de abstracción:

- No necesariamente es inyectiva (ver el ejemplo anterior).
- No necesariamente es suryectiva sobre el conjunto de términos de un TAD. Por la forma en la que Abs es construida no es posible diferenciar entre instancias de un TAD que son observacionalmente iguales y por lo tanto no es posible garantizar que todo término del TAD es imagen de Abs para alguna estructura de representación.
- Debe ser sobreyectiva sobre las clases de equivalencia definidas por la igualdad observacional, al menos con respecto al universo que nos ha restringido nuestro contexto de uso. Si no lo fuera, significaría que hay elementos del tipo que queremos representar que no podrán ser efectivamente representados. Por ejemplo, obviamente con un arreglo [1.,20] de nat no podemos representar conjuntos con más de 20 elementos. La estructura sería válida si el contexto de uso nos garantizara que efectivamente no manejaremos tales conjuntos.

Tendremos dos formas de describir la función de abstracción. La primera de ellas, en función de sus observadores básicos. Dado que éstos identifican de manera unívoca al objeto del cual hablan, al hablar del valor que tienen los observadores básicos aplicados al objeto, estamos describiendo sin ambigüedad el objeto representado.

Por ejemplo,

Abs : secu(nat) s $\longrightarrow$ conj(nat) {Rep(s)}

($\forall s$: secu(nat)) Abs(s) $=_{\text{obs}}$ c: conj(nat) | ($\forall n$: nat)($n \in c \iff$ Esta?(s, n))

Estamos diciendo que el conjunto representado por la secuencia tendrá como únicos elementos a aquéllos que estén en la secuencia.

Observar lo siguiente:

- Usamos los observadores básicos sólo para describir al tipo abstracto. Sobre el tipo de representación podemos aplicar cualquier función auxiliar (eventualmente podrían ser observadores, pero no es indispensable).
- Un $\Longrightarrow$ en lugar del $\Longleftrightarrow$, s representaría a cualquier subconjunto de c, con lo cual Abs no sería una función.
- Un $\Longleftarrow$ en lugar del $\Longleftrightarrow$, s representaría a cualquier superconjunto de c, con lo cual Abs no sería una función.

En nuestro ejemplo del conjunto, si la escribimos en función de los observadores básicos:

Abs : tupla⟨intervalo: conj_acotado_nat, los_demas: conj_no_acotado_nat⟩ t $\longrightarrow$ conj(nat) {Rep(t)}

($\forall t$: tupla⟨intervalo: conj_acotado_nat, los_demas: conj_no_acotado_nat⟩)

Abs(t) $=_{\text{obs}}$ c: conj(nat) | ($\forall n$: nat)($n \in c \Longleftrightarrow (n \in t.\text{intervalo} \vee n \in t.\text{los\_demas})$)

Otra forma de describir la función de abstracción es en función de los generadores del tipo de representación.

En el ejemplo del conjunto implementado sobre una secuencia:

Abs : secu(nat) s $\longrightarrow$ conj(nat) {Rep(s)}

($\forall s$: secu(nat), $\forall n$: nat)

Abs(<>) $\equiv$ $\emptyset$

Abs(n $\bullet$ s) $\equiv$ Ag(n, Abs(s))

Usar una u otra forma de escribir el Abs es equivalente, la elección depende de la comodidad y declaratividad.

6.4. Algoritmos

En este paso se implementan las operaciones del tipo a diseñar en términos de operaciones de los tipos soporte. Deben aparecer los algoritmos de cada una de las operaciones, sean éstas de la interfaz o auxiliares. En el caso de las funciones auxiliares, es bueno incluir, junto a sus algoritmos, sus precondiciones y postcondiciones. Siempre que ayude a clarificar los algoritmos, es bueno agregar invariantes y variantes (al menos en castellano), así como también comentarios que ayuden a la comprensión.

En el diseño de los algoritmos hay que tener en cuenta que las operaciones deben satisfacer sus pre/postcondiciones, y además deben satisfacer los requerimientos de eficiencia surgidos del contexto de uso.

6.4.1. Manipulación de datos de la representación

Supongamos que representamos los conjuntos de naturales mediante esta tupla:

`conj(nat)` se representa con `tupla`⟨elementos: `secu(nat)`, minimo: `nat`⟩.

El invariante de representación es:

$$\begin{aligned}\text{Rep}(c) \equiv\ & (\text{Esta?}(c.\text{elementos}, c.\text{minimo}) \vee (c.\text{minimo}{=}0 \wedge \text{vacia?}(c.\text{elementos}))) \wedge \\ & (\forall m\text{: nat})(\text{Esta?}(c.\text{elementos}, m) \Rightarrow c.\text{minimo} \leq m) \wedge \text{sinRepetidos}(c)\end{aligned}$$

Aquí sinRepetidos es una función del mundo funcional de los TADs que dice true si y sólo si la secuencia no tiene elementos repetidos.

Escribamos la operación *Agregar*. Por más que en la interfaz su signatura es

AGREGAR(**in/out** c: `conj(nat)`, **in** n: `nat`)

internamente debe trabajar con una tupla. Para denotar esta conversión y que no sea algo mágico, prefijaremos al nombre de la función con una "i". Al ver esta "i" interpretaremos que el parámetro que recibe es del tipo de representación y por ende se puede acceder a su estructura. Es decir, un tipo que usa conjunto, como lo ve "desde

afuera" debe invocar a la función de la interfaz –AGREGAR– y asumimos que el lenguaje de implementación se encarga de hacer la conversión de tipos.

Entonces, la función queda:

```
IAGREGAR(in/out c: tupla⟨elementos: secu(nat), minimo: nat⟩, in n: nat)
    if (VACÍA?(c.elementos) ∨ c.minimo > n)
        c.minimo ← n
    endif
    if ¬ESTÁ?(c.elementos, n)
        INSERTAR(c.elementos, n)
    endif
```

La interfaz del módulo de abstracción de secuencias debe tener las operaciones VACÍA?, ESTÁ? e INSERTAR.

Veamos en detalle:

- c es de género `conj(nat)` para el exterior, pero para el cuerpo del procedimiento es una tupla.
- Luego el valor c es una tupla, con campos elementos y minimo. Entonces c.elementos es de género `secu(nat)`.
- De manera análoga, una vez obtenida la secuencia, operamos sobre ella para insertarle el nuevo elemento.
- Y además mantenemos el invariante del mínimo cierto.

Observemos los siguientes aspectos claves:

- No nos metimos dentro de la estructura de representación de la secuencia y, por lo tanto, no hemos hecho asunciones acerca de cómo está implementada. De esta manera, un cambio en la implementación de secuencia no tendrá impacto en nuestro diseño del tipo conjunto.
- Las operaciones VACÍA?, ESTÁ? e INSERTAR del tipo secuencia deben aparecer en la interfaz de su módulo.

Consideremos:

`agenda` se representa con `conj_acotado_gente`.

AGREGAR(**in/out** A: `agenda`, **in** G: `gente`)

```
IAGREGAR(in/out A: conj_acotado_gente, in G: gente)
    //Queremos agregar G a una agenda.
    //Supongamos que agenda se diseñó sobre conj_acotado_gente,
    //y éste sobre arreglo[1...10] de gente.
    var encontrado: bool
        n: nat

    encontrado ← false
    n ← 0
    while (¬encontrado)
        n ← n + 1
        encontrado ← A[n].libre
    endwhile
    A[n].gente ← G
```

Está **MAL**, pues estamos metiéndonos con la representación del conjunto acotado de gente, que no pertenece a nuestro módulo: A es un `conj_acotado_gente`.

En su lugar, correspondería:

```
IAGREGAR(inout A: conj_acotado_gente, in G: gente)
    AGREGAR(A, G)

IAGREGAR(inout C: arreglo [1..10] de gente, in G: gente)
    // Esta operación es del módulo conjunto acotado de gente.
    // El código está oculto a los demás módulos; lo único que se necesita para
```

```
// usar al conjunto acotado es conocer las operaciones exportadas.
var
    encontrado: bool
    n: nat

encontrado ← false
n ← 0
while (¬encontrado)
    n ← n + 1
    encontrado ← C[n].libre
endwhile
C[n].gente ← G
```

Ahora ya podemos presentar los algoritmos para todas las operaciones del módulo.

Para abreviar, llamaremos `estructura_interna` a la `tupla`⟨intervalo: `conj_acotado_nat`, los_demas: `conj_no_acotado_nat`⟩.

```
iVacío() → res : estructura_interna
    res.intervalo ← Vacío
    res.los_demas ← Vacío

iAgregar(in/out c: estructura_interna, in n: nat)
    if EnElRango(c.intervalo, n)
        Agregar(c.intervalo, n)
    else
        Agregar(c.los_demas, n)
    endif

iBorrar(in/out c: estructura_interna, in n: nat)
    if EnElRango(c.intervalo, n)
        Borrar(c.intervalo, n)
    else
        Borrar(c.los_demás, n)
    endif

iPertenece(in c: estructura_interna, in n: nat) → res : bool
    if EnElRango(c.intervalo, n)
        res ← Pertenece(c.intervalo, n)
    else
        res ← Pertenece(c.los_demas, n)
    endif

iVacío?(in c: estructura_interna) → res : bool
    res ← Vacío?(c.intervalo) ∧ Vacío?(C.los_demas)

iMínimo(in c: estructura_interna) → res : nat
    case
    [] Vacío?(c.intervalo)      // Por la precondición, los_demas no es vacío.
        res ← Mínimo(c.los_demas)
    [] Vacío?(c.los_demas)      // Por la precondición, intervalo no es vacío.
        res ← Mínimo(()c.intervalo)
    otherwise
        res ← mín(Mínimo(c.intervalo), Mínimo(c.los_demas))
    endcase
```

7. Servicios usados

Aquí es donde indicamos qué responsabilidades le dejamos a los tipos soporte que usamos. Son las pautas y requerimientos que se extraen del diseño de este tipo para el diseño de los tipos de la representación. Luego pasarán a

ser las *interfaces* y los *contextos de uso y requerimientos de eficiencia* para los módulos de soporte de los tipos usados en la representación.

Por ejemplo, supongamos que tenemos que implementar una operación AlgunaCosa con un parámetro r, y en el contexto de uso dice que su complejidad debe ser $O(n^3)$ en el peor caso, donde $n = \text{Tamaño}(r)$. Supongamos también que el algoritmo es más o menos así:

AlgunaCosa(**in/out** r : `género`, **in** p : `algún_género`)
for i from 1 to Tamaño(r)
    Algo(r.estructura_interna, otros_parámetros)
endfor

Claramente, la complejidad de Algo debe ser $O(n^2)$ en el peor caso[4]. Este requerimiento será indicado en esta sección, y será tenido en cuenta posteriormente al diseñar el tipo de estructura_interna.

Notar que si no aclaramos esto, los requerimientos de eficiencia expresados en el contexto de uso del tipo que estamos diseñando no necesariamente serían satisfechos. Es decir, estamos diciendo que los requerimientos de eficiencia del tipo diseñado serán satisfechos si los tipos que usamos a su vez satisfacen los que imponemos.

También deben incluirse aquí los requerimientos relacionados con aspectos de aliasing. Por ejemplo, si se tiene un diccionario d cuyos significados son conjuntos y se desea agregar un elemento e al conjunto asociado a la clave c, las siguientes líneas
$sig \leftarrow \text{Obtener}(c, d)$
$\text{Ag}(e, c)$
sólo modificarán el conjunto que se encuentra en el diccionario si obtener devuelve una referencia. En caso contrario, se deberá agregar a continuación la línea $\text{Definir}(c, sig, d)$ para que el cambio se registre en el diccionario.

A. Tipos de datos

A.1. Tuplas

Una *tupla* especifica objetos que están compuestos por un número finito de otros objetos, llamados *componentes*.

El tipo `tupla`$\langle campo_1$: `tipo_dato`$_1$, ..., $campo_n$: `tipo_dato`$_n\rangle$ tiene disponibles las siguientes operaciones:

- $\langle \bullet, \ldots, \bullet \rangle$(`tipo_dato`$_1$, ..., `tipo_dato`$_n$) $\rightarrow$ res : `tupla`
  Genera un objeto de tipo tupla.
- $\bullet.campo_i \leftarrow \bullet$(`tupla`, `tipo_dato`$_i$) $\rightarrow$ res : `tupla`
  Cambia el valor del i-ésimo campo de una tupla.
- $\bullet.campo_i$(`tupla`) $\rightarrow$ res : `tipo_dato`$_i$
  Retorna el valor del i-ésimo campo de una tupla.
- $\bullet = \bullet$(`tupla`, `tupla`) $\rightarrow$ res : `bool`
  Es la igualdad entre tuplas, componente a componente. Por eso se *exige* que los tipos que son componentes de una tupla tengan definida una operación de igualdad.

Por ejemplo:

`complejo` se representa con `tupla`$\langle$*parte_real*: `real`, *parte_imaginaria*: `real`$\rangle$

La operación de creación tiene complejidad $O(n)$ y las demás operaciones $O(1)$. Se genera aliasing con todos los parámetros de tipos no básicos.

Nota: tanto para la asignación como para el pasaje de parámetros, las tuplas se transfieren componente a componente, pasándose por valor las componentes pertenecientes a tipos primitivos, y por referencia aquéllas que no lo sean.

A.2. Punteros

Un *puntero* es una variable que almacena direcciones de memoria. Si la dirección de memoria tiene un contenido previamente creado por la ejecución del programa y este contenido pertenece a un tipo que coincide con el tipo del

[4] Si bien hay otras opciones correctas para posibles complejidades de Algo, obviaremos este detalle en este momento.

puntero, diremos que la dirección es válida. En caso contrario diremos que es inválida.

El tipo `puntero(tipo_dato)` tiene disponibles las siguientes operaciones:

&•(`tipo_dato`) → *res* : `puntero(tipo_dato)`
: Retorna la ubicación en memoria de la variable indicada. Para poder utilizar esta operación la variable pasada por parámetro debe haber sido inicializada en algún momento.

*•(`puntero(tipo_dato)`) → *res* : `tipo_dato`
: Permite acceder al dato ubicado en la dirección de memoria. Para poder utilizar esta operación el puntero pasado por parámetro debe contener una dirección de memoria válida.

NULL : `puntero(tipo_dato)`
: Es un puntero especial que no apunta a nada. No se puede aplicar la operación * a NULL.

Ejemplo: Si la variable p es de tipo `puntero(conj(nat))`, las siguientes operaciones son válidas:

$p \leftarrow$ &Vacío() //*p es un puntero a un* `conj(nat)` *vacío.*

if Pertenece(*p, 4) ... //*Verifica la pertenencia de* 4 *en* *p.

$p \leftarrow$ NULL //*p es un puntero a* NULL

if $p =$ NULL ... //*verifica si p es un puntero a* NULL

if $p = \&a$ ... //*verifica si p está apuntando a la variable a*

if $*p = a$ ... //*verifica si el lugar al que apunta p y la variable a tienen el mismo valor*

Notar que los dos últimos ejemplos no son equivalentes: el anteúltimo ejemplo exhibe una comparación estrictamente mas fuerte que el último,.

Por supuesto, un puntero tiene aliasing con la última variable a la que se haya apuntado. Todas las operaciones tienen complejidad $O(1)$.

Nota: los punteros se comportan como tipos primitivos del lenguaje.

A.3. Arreglos

Tanto el denominado arreglo estático como el arreglo dimensionable responden a la especificación de arreglo. La diferencia entre ambos en el lenguaje de diseño es la siguiente: la dimensión del arreglo estático se define al declararlo, mientras que la del areglo dimensionable se puede definir en tiempo de ejecución. Ninguno de los dos posee la capacidad de alterar su dimensión en tiempo de ejecución una vez establecida.

Ambos arreglos resuelven sus operaciones de asignación de elementos en posiciones, consulta de posiciones y consulta de definición de posiciones en tiempo $O(1)$. El tiempo de creación de ambos es $O(n)$, siendo n la cantidad de posiciones en el arreglo.

Un arreglo de n posiciones tiene índices válidos entre 0 y $n - 1$.

Ejemplos:

var AD: `arreglo_dimensionable de char`
    AE: `arreglo_estático[15] de char`
    n: `nat`
$n \leftarrow 15$
// Inicialización
$AD \leftarrow$ CrearArreglo(n)
//AE ya está inicializado, aunque no sus posiciones
// Asignación
$AD[2] \leftarrow$ 'h'
$AE[7] \leftarrow$ 'j'
// Consulta
if (Definido?(AD, 10) $\wedge_L$ $AD[10] = 8$) ...
if (Definido?(AE, 10) $\wedge_L$ $AE[10] = 8$) ...

Nota: el arreglo estático y el arreglo dimensionable se comportan como tipos no primitivos del lenguaje tanto para la asignación como para el pasaje de parámetros, si bien se considera que ya están diseñados.

B. Documentación de diseño

Un módulo de diseño debe tener la siguiente estructura:

1. Especificación (puede omitirse si es uno de los TADs provistos por la cátedra, o incluirse sólo los cambios si es una extensión de un TAD ya conocido).
2. Aspectos de la interfaz
   a) Servicios exportados: órdenes de complejidad, aspectos de aliasing, efectos secundarios, todo lo que el usuario necesite saber.
   b) Interfaz
3. Pautas de implementación
   a) Estructura de representación: estructura elegida, justificación, estructuras alternativas, etc.
   b) Invariante de Representación
   c) Función de Abstracción
   d) Algoritmos
4. Servicios usados
   a) Órdenes de complejidad, aspectos de aliasing, etc., requeridos de los tipos soporte.

En los casos en que se requiera justificar órdenes de complejidad, estas justificaciones pueden incluirse o bien al final del módulo, o junto a los algoritmos correspondientes.