Introducción a la especificación de software mediante tipos abstractos de datos

Cátedra de Algoritmos y Estructuras de Datos II
Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Resumen

En este documento buscamos introducir al lector a la especificación formal de software, en particular mediante el uso de tipos abstractos de datos. En él presentaremos motivación, fundamentos, estructura, bibliografía, etc.

Este texto está dirigido a estudiantes de la materia Algoritmos y Estructuras de Datos II, del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales de la UBA.

⚠ Este apunte no se encuentra aún en su forma final. Lo consideramos correcto en cuanto a contenido, pero imperfecto en cuanto a presentación, por lo que desde ya pedimos disculpas. Por ese motivo, agradeceremos avisarnos de cualquier error que en él encontrasen. Gracias.

Índice

1. Introducción a los *tipos abstractos de datos*

1.1. Especificaciones formales

No podemos resolver un problema sin comprenderlo cabalmente. Si el problema tiene aspectos sutiles o complejos, más difícil nos resultará comprenderlo y una vez que lo hayamos hecho, desearemos no olvidar ninguno de estos detalles. Además, probablemente deseemos intercambiar opiniones al respecto con otras personas.

Para lograr estos objetivos, es necesario tener una descripción precisa del problema, una que pueda interpretarse sin lugar a dudas, sin necesidad de un interlocutor presente. El lenguaje natural (el Castellano en nuestro caso) no es buen medio para satisfacer este tipo de necesidades. Es por naturaleza ambiguo, y cuanto más preciso intentamos que sea, menos entendibles y naturales son las oraciones que tenemos que utilizar. Veamos un ejemplo:

> Juan se levanta todas las mañanas y va a la casa de José, quien toma las pertenencias de aquél y las carga en su auto. Luego parte, sin saludarlo, hacia su destino.

Parece claro que las pertenencias tomadas son de Juan, y que José es quien las toma, pero ¿de quién es el auto donde son colocadas? ¿Quién parte? Si intentamos lograr algo de precisión, el texto se vuelve peor:

> Juan se levanta todas las mañanas y va a la casa de José. José toma las pertenencias de Juan y las carga en el auto que le pertenece (el de José) y, sin saludarlo, él parte hacia su propio destino.

El texto del ejemplo parece no tener mayor importancia e interpretarlo de forma incorrecta no parece tener mayores consecuencias. Sin embargo, si buscásemos describir la forma de operación del software que controla el enfriador de una central nuclear, o el del piloto automático de un avión –entre tantos otros ejemplos–, pequeños errores podrían tener consecuencias muy graves.

En ese contexto, nace la idea de utilizar un lenguaje formal para especificar el comportamiento del software. Los lenguajes formales tienen una serie de ventajas que los hacen muy aptos para la tarea. No las mencionaremos todas aquí, pero basta con decir que no son ambiguos, es decir, tienen una interpretación única, inequívoca. Esa exactitud nos obliga a ser más precisos al escribir, pero eso es bueno: si a la hora de expresar determinada idea como una fórmula, no encontramos una buena forma de hacerlo, tal vez lo que esté sucediendo es que haya una parte del problema que no comprendamos en suficiente detalle.

Otra característica, que mencionamos aquí de manera anecdótica, aunque será (parcialmente) explorada más adelante en la materia, es la posibilidad de verificar propiedades. Una especificación formal nos puede ayudar a verificar si el comportamiento descripto es correcto o no.

Existen varios lenguajes formales para especificar software. Sin embargo, nos concentraremos en sólo uno de ellos, los *tipos abstractos de datos*. Los tipos abstractos de datos, también conocidos por su abreviatura TADs, son modelos matemáticos que se construyen con el fin de exponer los aspectos relevantes del problema bajo análisis.

1.2. ¿Por qué utilizar TADs?

Dado un objeto de estudio la cantidad de detalles, situaciones e interacciones que se pueden encontrar en él hacen imposible un análisis exhaustivo del mismo que permita una comprensión cabal sobre su funcionamiento.

En estos casos (que abarcan a casi todos los objetos de estudio, incluso aquellos que nos pueden parecer simples a primera vista), utilizar la *abstracción* como herramienta para la comprensión es fundamental.

Los TADs buscan precisamente eso. Y junto al concepto de encapsulamiento (ocultamiento de información), conforman una herramienta muy útil para resaltar las cualidades relevantes de lo que queremos analizar.

Con respecto a otros métodos, los TADs tienen la ventaja de que pueden ser entendidos en el segundo año de nuestra carrera (es decir, no requieren de alta sofisticación matemática para poder ser usados), son muy flexibles (es decir, permiten modelar una amplia variedad de situaciones), son generales (en el sentido de que no fueron ideados para un tipo en particular de sistemas, como otros formalismos). Además, desde un punto de vista histórico, son piedra angular de un gran número de conceptos, lo que les da más valor aún.

1.3. Formalmente, ¿qué es un TAD?

Hay más de una respuesta a esta pregunta. En general la más aceptada define un TAD como el "conjunto de modelos de una especificación". Es decir, sus posibles interpretaciones. Más informalmente, es lo que se entiende al leer una especificación; es aquéllo que la especificación describe. Un modelo determina qué elementos del mundo real estarán reflejados en la especificación, y qué operaciones se permitirá realizar sobre ellos. El concepto de *modelo* se verá en más detalle en la sección 1.5, pero hay un recorrido que debemos completar primero.

Debemos entender que todo lenguaje tiene una *gramática* o *sintaxis*, que es un conjunto de reglas que indican cómo se escriben las oraciones de ese lenguaje. Además, hay otras reglas, llamadas *semántica*, que indican cómo deben entenderse (interpretarse) las oraciones válidas del lenguaje. Vamos de a pasos; comencemos por la sintaxis.

1.4. ¿Cómo se escribe un tipo abstracto de datos? (sintáxis)

Para determinar un tipo abstracto de datos, en AEDII utilizamos *especificaciones* de TADs (de la misma forma que se usan axiomas para caracterizar estructuras matemáticas).

Para especificar un TAD es necesario definir:

1. La signatura (sólo dice qué operaciones y con qué aridad deben tener los modelos).
2. Los axiomas (que determinan el comportamiento de las operaciones).

Los axiomas son *fórmulas bien formadas* según ciertas reglas (gramática). Elaboraremos brevemente sobre esto.

Para definir qué es una fórmula necesitamos previamente definir qué son los *términos*. En nuestro caso los términos son las *variables*, los símbolos de *constantes* y los símbolos de *función* aplicados a términos.

Una *fórmula bien formada* es o bien un símbolo de predicado aplicado a términos (por ejemplo: $P(t_1, t_2)$), o bien fórmulas bien formadas unidas por conectivos lógicos ($\vee$, $\wedge$, $\Rightarrow$) o cuantificadas ($\forall$ y $\exists$ son cuantificadores). Una fórmula bien formada es *cerrada* si todas sus variables están cuantificadas. Las fórmulas cerradas son las que son susceptibles de ser verdaderas o falsas.

Dada una especificación de un TAD, podemos utilizar reglas de inferencia para razonar sobre la misma. Un *sistema deductivo* es un conjunto de axiomas y reglas de inferencias. Al decir *de primer orden*, queremos decir que incluye los axiomas de la lógica de primer orden (por ejemplo, $\forall x : P(x) \vdash P(x)$).

En nuestro caso los axiomas son los que damos explícitamente (más los de la lógica de primer orden que están implícitos) y las reglas de inferencia son:

- Modus Ponens ($(p \Rightarrow q, p) \vdash q$, es decir, si p implica q y vale p, entonces vale q),

- generalización ($P(x) \vdash (\forall x : P(x))$ si x no aparece en ninguna premisa),
- y las reglas de la lógica ecuacional (reemplazo de iguales por iguales).

Los axiomas y aquellas cosas que se infieren a partir de ellos y las reglas de inferencia son *teoremas*. Los teoremas son fórmulas bien formadas del sistema deductivo que tienen una demostración, es decir, una secuencia de axiomas y reglas de inferencias aplicadas a axiomas o líneas anteriores de la demostración, en la cuál el teorema es la última línea.

Veamos un ejemplo sencillo para entender el concepto anterior.

Nuestro teorema será long(a • b • <>) >0.

Los axiomas son

$$\begin{array}{lll} \text{ax}_1) & \text{long}(<>) & \equiv \quad 0 \\ \text{ax}_2) & \text{long}(\text{a} \bullet S) & \equiv \quad 1 + long(S) \end{array}$$

Luego, $\text{long}(\text{a} \bullet \text{b} \bullet <>) \underset{ax_2}{\equiv} 1 + \text{long}(\text{b} \bullet <>) \underset{ax_2}{\equiv} 1 + 1 + \text{long}(<>) \underset{ax_1}{\equiv} 1 + 1 + 0 \equiv 2$

También sabemos por la axiomatización de los naturales que $2 > 0$. Finalmente, reemplazando términos que demostramos que son equivalentes, obtenemos que vale nuestro teorema long(a • b • <>) > 0, es decir, se infiere de nuestra teoría.

Un sistema deductivo denota una *teoría*: el conjunto (infinito) de teoremas del sistema deductivo[1]. Hasta aquí, conceptos meramente sintácticos.

Esta sección puede resultar un poco difícil de digerir para el lector no familiarizado con estos temas. Aconsejamos releerla para fijar las ideas, pero no obsesionarse: somos conscientes de que a esta altura de la carrera no se han visto aún algunos conceptos necesarios para entender en profundidad lo que acabamos de decir. Sin embargo, esto no es fundamental para entender el resto, aunque creemos que puede servir de ayuda.

1.5. ¿Qué *significa* un tipo abstracto de datos? (semántica)

Hablemos ahora del significado de un TAD. Dada nuestra teoría de primer orden denotada por la especificación del TAD, debemos asociarle una *semántica*, es decir, tenemos que ponernos de acuerdo en qué significa cada cosa que se puede escribir de acuerdo a las reglas sintácticas. Dicho un poco más formalmente, tenemos que declarar con precisión qué cosas son modelos de nuestra teoría.

¿Qué es un modelo de una teoría? Un ente matemático es modelo de nuestra teoría si hace corresponder a cada género del TAD un conjunto y a cada operación una función (total, pero dejemos eso temporalmente de lado). Hagamos un paso más, una teoría es *consistente* cuando en ella no es cierto que verdadero es igual a falso. En las teorías inconsistentes, todo es cierto y por ende no tienen modelos (con lo cual no modelan en particular aquello que pretendíamos modelar).

¿Cualquier teoría es consistente? No. Veamos un ejemplo. Tomemos la especificación del TAD NAT, y agreguemos el constructor "cerapio : $\longrightarrow$ nat". Tomemos los axiomas de NAT que hablan de 0, y agreguémoslos reemplazando 0 por "cerapio". ¿Eso tiene modelo? Sí, los números naturales, donde "cerapio" y 0 se interpretan como el cero de los naturales. Ahora bien, agreguemos otro axioma que diga "$0 \neq$ cerapio". Eso que escribimos no tiene modelos, ya que podemos demostrar trivialmente que en los naturales hay un único neutro aditivo.

El videoclub de la vuelta de mi casa, ¿es modelo de la especificación del TAD VIDEOCLUB? Claramente no, ya que el videoclub de la vuelta de mi casa es una instancia y cualquier modelo

[1] Tenemos que hacer una aclaración, sólo para decir las cosas correctamente: nuestras teorías son en realidad teorías de primer orden "multi-sorted" es decir, una extensión de las teorías de primer orden para incorporar el concepto de género.

del TAD Videoclub representa a todas las instancias del TAD.

Los videoclubes del planeta, ¿son un modelo del TAD Videoclub? Formalmente, no: los modelos son entes matemáticos. Sin embargo, si hacemos una abstracción, en la que dejamos de lado el hecho de que los videoclubes del planeta son finitos mientras que las instancias del TAD Videoclub son infinitas y además, miramos a cada videoclub real (como el de la vuelta de mi casa) como un ente matemático, tal vez sí lo sean. Cuando decimos que buscamos que un TAD modele cierto aspecto de la vida real, a eso nos referimos.

Haciendo la misma abstracción, podemos considerar a la sucesión de videoclubes que se obtienen como la evolución temporal del videoclub de la vuelta de mi casa. Es decir, ahora, el videoclub representa una cosa, pero cuando el próximo cliente devuelva "La venganza de los Nerds, VIII", va a representar otra. En ese contexto, ya que tengo un conjunto infinito, puedo abstraer cada momento del videoclub a un ente matemático y buscar que eso sea modelo de mi TAD.

Además, cuando implementamos, lo que queremos es que una clase asociada a una especificación de un TAD satisfaga que las implementaciones de las operaciones computen funciones con las que puedo armar un modelo de la especificación. Difícil decirlo de corrido, ¿no? Releerlo hasta lograrlo.

Volvamos a los modelos. Acotación: gracias a que se trata de una teoría de primer orden, sabemos que todo lo que podamos demostrar (mediante procedimientos sintácticos) es verdad desde un punto de vista semántico.

Debemos notar que hay varios modelos posibles para una teoría, muchos de ellos isomorfos (iguales excepto por renombre de constantes, funciones, etc.) y algunos no. Entonces, cuando se presenta un lenguaje, no sólo debe darse su sintaxis, sino también las reglas para saber qué cosas son modelos (su semántica) ya que si no, sabemos escribir pero no entendemos qué significa lo que escribimos.

1.6. El metalenguaje

Muchas veces para describir propiedades acerca de un lenguaje formal no nos alcanza con ese lenguaje, y es por eso que necesitamos de un *metalenguaje* para lograr dicho fin. En lugar de intentar una complicada definición –que excedería por mucho el marco de este apunte–, trataremos de que el concepto quede claro para nuestra aplicación particular.

Debemos notar que existen dos tipos de valores booleanos. Por un lado tenemos el género bool del TAD Bool, que es un género más, como nat, o como videoclub. Así como con el TAD Nat pretendemos capturar el comportamiento de $\mathbb{N}$, con el TAD Bool pretendemos capturar el comportamiento de los valores de verdad *verdadero* y *falso*.

Veamos una diferencia: **true** es una constante del lenguaje, mientras que *verdadero* es el "resultado" que se obtiene de evaluar una fórmula cerrada. En este caso, *verdadero* y *falso* pertenecen al metalenguaje, ya que no forman parte del "mundo" de los TADs.

En las especificaciones de TADs, el metalenguaje es utilizado para escribir las restricciones de las funciones y para describir la igualdad observacional, conceptos que veremos más adelante.

Aclaración importante: si bien en este apunte se presentan ambos tipos de valores booleanos, se hace a título informativo. Esto quiere decir que en el uso cotidiano haremos un abuso de notación y no tendremos en cuenta esta separación: utilizaremos un único bool, indistintamente.

1.7. Empezando a especificar

Debemos recordar por qué estamos especificando: se trata de capturar lo más fielmente posible, y con precisión matemática, un problema, para el que luego encontraremos una solución. Es importante tener en cuenta que en esta etapa debe preocuparnos el problema que se intenta

resolver, y no sus eventuales soluciones.

Nuestra primera tarea será comprender el enunciado. Difícilmente podamos expresar formalmente algo que no entendemos. Una vez que hemos entendido de qué se trata el problema con el que nos enfrentamos –y no antes– podemos empezar a darle la forma rigurosa que acompaña a los TADs.

Siempre hay excepciones. En algunos casos complicados tal vez no terminemos de entender cabalmente el problema porque nos parece que hay inconsistencias o errores en la forma en la que está enunciado. En esos casos lo más conveniente es preguntar al *experto de dominio*[2] (o en el caso de ejercicios de la práctica, a los docentes) a fin de clarificar la situación.

Si esto no fuera posible, o aún así la dificultad no se disipara, una buena idea es ir realizando aproximaciones sucesivas, capturando en un primer borrador la parte escencial e intentando luego introducir los aspectos más difíciles. Tal vez nos demos cuenta que el esquema que habíamos planteado originalmente no es el apropiado para el problema en estudio. Entonces deberemos empezar la tarea nuevamente, pero con seguridad nos habremos acercado a entender mejor la parte complicada. En estos borradores, no necesitamos realizar una axiomatización completa. Muchas veces basta para este fin con plantear la signatura y escribir informalmente una idea de la forma general de los axiomas.

Un punto que suele ser engañoso está relacionado con nuestros propios prejuicios y experiencias de vida. A la hora de especificar, nuestra noción de la "realidad" debe limitarse a aquellos aspectos dados por el enunciado y no debemos complementarlos con nuestras propias experiencias. No debemos olvidarnos que cuando se realiza una especificación de un problema, en realidad estamos trabajando sobre una *abstracción* del mismo. Es decir, lo más probable es que lo que reflejemos en nuestro modelo, sean los aspectos más importantes de aquél ente (material o conceptual) que queremos capturar.

Por ejemplo, si nuestra misión fuese modelar el proceso de manufactura de una fábrica para establecer el volúmen de producción en función de la materia prima, seguramente no sea de interés para nadie que incluyamos en el modelo detalles sobre los horarios de los empleados que operan la máquina, o dónde viven ellos y cuantos hijos tienen. De la misma forma, si estuviésemos especificando el sistema de sueldos, sí nos interesarían esos datos y no otros, como a qué distancia de la máquina se sienta cada operario.

Exactamente de eso se trata la abstracción: de dejar de lado los detalles que no son importantes a los fines que perseguimos, para concentrarnos en aquellos aspectos que son fundamentales.

1.8. Todo en su justa medida

A la hora de escribir los axiomas para nuestras operaciones, debemos tener en cuenta que siempre son funciones totales[3]. En ese sentido, siempre que utilicemos una restricción[4], estaremos *subespecificando*, en el sentido de que estaremos dejando constancia de que no vamos a decir qué valores toma la función cuando sus parámetros no cumplen con la restricción.

Ése es un uso lícito de la subespecificación, ya que no siempre podemos caracterizar los resultados de una operación para todo su dominio, o bien porque no tenga sentido en el marco de nuestro problema, o bien porque sea imposible por alguna limitación técnica (por ejemplo, dividir por cero, o encontrar el mínimo elemento de un conjunto vacío).

Sin embargo, debemos ser cuidadosos de dotar al resto de los valores del dominio de su

[2] Expertos del dominio es el nombre que se le da al grupo de personas que conoce con mayor precisión el problema que se intenta resolver. En el caso del desarrollo comercial de un software a medida, este rol seguramente lo cumplirá algún empleado del cliente.

[3] Una función es total cuando está definida para todo valor de su dominio. Por ejemplo, $\bullet/\bullet : \mathbb{N}_0 \to \mathbb{N}_0$ no lo es ya que no está definida para el 0.

[4] Debemos notar que no es correcto llamar "precondiciones" a las restricciones. Si bien son conceptos similares, el primero requiere una noción de estado que el segundo no.

correspondiente imagen. Dicho en otras palabras, si determinado valor de los parámetros de una operación es válido de acuerdo a las restricciones, entonces deberíamos indicar, mediante los axiomas, cuál es el resultado de la función para esos valores. En algunos casos no diremos exactamente qué resultado dá la función, si no que indicaremos qué características tiene ese resultado (como lo hacemos, por ejemplo, en la función dameUno de conjuntos).

Es importante no caer en el otro extremo: tampoco debemos *sobreespecificar*. Decimos que una operación está sobreespecificada cuando hay varias formas de saber cuál es su resultado para unos valores dados de sus parámetros. Veamos un ejemplo:

$$\begin{array}{lll} \text{ax}_1) & \text{long}(<>) & \equiv \quad 0 \\ \text{ax}_2) & \text{long}(\text{a} \bullet S) & \equiv \quad 1 + \text{long}(S) \\ \text{ax}_3) & \text{long}(\text{a} \bullet \text{b} \bullet S) & \equiv \quad 2 + \text{long}(S) \end{array}$$

En este caso, para la secuencia a $\bullet$ b $\bullet$ c $\bullet$ $<>$, tenemos dos formas de obtener su longitud. O bien aplicamos los axiomas 3 y 1, o bien aplicamos dos veces el axioma 2 y luego el 1. En este caso, el procedimiento puede resultar un poco confuso, pero no hay ningún inconveniente serio, ya que por cualquiera de los dos caminos llegamos al mismo resultado. El problema se presenta cuando dependiendo de nuestra elección obtenemos valores distintos (¡recordemos que son funciones!).

Veámoslo en un ejemplo: supongamos que queremos definir una función que indique si un número entero es no-negativo y por algún motivo no contamos con el operador $\geq$. Nuestro primer intento podría tener esta pinta:

$$\begin{array}{l} \text{ax}_1)\ \text{es\_nonegativo}(0) \equiv \textbf{true} \\ \text{ax}_2)\ \text{es\_nonegativo}(n) \equiv \textbf{if}\ n > 0\ \ \textbf{then}\ \ \textbf{true}\ \ \textbf{else}\ \ \textbf{false}\ \ \textbf{fi} \end{array}$$

Parece correcto: si el número es 0, diremos que es no negativo, gracias al axioma 1; si aún no sabemos la respuesta recurriremos al axioma 2, que nos dirá que es no negativo si es mayor que 0, o falso en otro caso. ¡ERROR! Los axiomas no funcionan de esa forma: no se evalúan en orden, terminando la evaluación con la primer coincidencia, como sucede en algunos lenguajes. Todos los axiomas valen a la vez. Por ese motivo, *es_nonegativo*(0) da **true** de acuerdo al primer axioma y **false** de acuerdo al segundo. Hemos introducido una inconsistencia, y por lo tanto privado a nuestra teoría de modelos.

La forma correcta de hacerlo sería:

$$\begin{array}{l} \text{ax}_1)\ \text{es\_nonegativo}(n) \equiv \textbf{if}\ n = 0\ \ \textbf{then} \\ \qquad\qquad\qquad\qquad\quad\ \ \textbf{true} \\ \qquad\qquad\qquad\qquad\ \ \textbf{else} \\ \qquad\qquad\qquad\qquad\quad\ \ \textbf{if}\ n > 0\ \ \textbf{then}\ \ \textbf{true}\ \ \textbf{else}\ \ \textbf{false}\ \ \textbf{fi} \\ \qquad\qquad\qquad\qquad\ \ \textbf{fi} \end{array}$$

Hay, sin embargo, un concepto más sutil y avanzado que a veces también recibe el nombre de subespecificación. Sólo comparte el nombre con el anterior, ya que no se trata de un error y suele ser bastante útil.

Supongamos que tenemos el siguiente enunciado: "[...] El ganador del torneo recibirá un premio de entre los disponibles. El mecanismo concreto de asignación del premio se definirá más adelante, probablemente durante la etapa de diseño. [...]". La idea de dejar algunos aspectos particulares sin una definición precisa es un recurso muy útil para manejar algunas incertidumbres a la hora de especificar. Pensemos en dos soluciones posibles para la función:

Premiar : torneo t $\times$ jugador j $\longrightarrow$ premio $\qquad\qquad$ {es_ganador?(t, j)}

Solución A: Premiar$(t, j) \equiv$ SeleccionarMejorPremio(PremiosDisponibles(t))

Solución B: Premiar$(t, j) \equiv$ DameUno(PremiosDisponibles(t))

Veamos que en la primera solución estamos dando explícitamente la estrategia de selección de premios. Eso hace que cualquier implementación que utilice otro criterio no satisfaga (es decir, no sea modelo) de nuestra especificación. La segunda, por el contrario, caracteriza de forma mucho más *débil* el resultado. Eso significa que cualquier forma de elección de premios que se decida al implementar va a ser apropiada, ya que todas ellas van a satisfacer la característica de elegir uno del conjunto de potenciales premios.

Debemos tener en cuenta que la debilidad de la solución B no la hace incorrecta: el resultado de la función sigue estando correctamente caracterizado para cualquier valor del dominio que satisfaga la restricción.

2. Especificación de *tipos abstractos de datos*

A continuación explicaremos en detalle cada uno de los componentes que conforman un tipo abstracto de datos, seguido de un esquema a modo de molde, que muestra la estructura y sintaxis necesarias para escribir un TAD.

2.1. Igualdad observacional

La igualdad observacional es un predicado que nos dice cuándo dos instancias del aspecto de la vida real que nuestro TAD está modelando se comportan de la misma manera. Claramente es un concepto semántico y no sintáctico, con lo cual es necesario utilizar el metalenguaje para describirla.

Para los TADs, una semántica posible es la semántica inicial, que burdamente descripta trata de partir el universo de acuerdo a las ecuaciones que aparecen en el TAD. Al hacer esto se obtiene que todos los términos se interpretan en el modelo como distintos, excepto que haya ecuaciones que permitan deducir que son iguales. Ventaja: la relación de igualdad que queda definida en el modelo es una *congruencia* (esto es, cosas que están en la misma clase de equivalencia, van a la misma clase de equivalencia sin importar qué función se les aplique). Desventaja: los modelos no son muy bonitos, salvo que uno agregue axiomas *ad-hoc* (ejemplo: el modelo de los conjuntos son las secuencias).

Debido a este problema se inventó la semántica observacional. ¿En qué consiste? Algunas funciones se etiquetan como observadores básicos. El universo se particiona de acuerdo a ellos. Ventaja: los modelos son más bonitos, legibles e intuitivos. Desventaja: en el caso general, la igualdad del modelo no es un congruencia (dicho con más formalidad, la igualdad de términos no se interpreta con una congruencia). Ejemplo: TAD Persona con DNI como observador básico y nombre como otra operación.

Para un matemático, eso no es un problema. Es perfectamente válido tener una teoría cuyo modelo no tenga a $=$ como congruencia. Para un profesional de la computación, sí es un problema. Eso significa que al implementar habrá instancias de su género de implementación para las que su *operator*$==$ dirá **true** y para las cuales alguna función dará valores distintos. Un horror. Lo importante a observar es que se trata de un problema de modelado pero no de un problema formal (es decir, sigue habiendo modelos consistentes). Como queremos que la igualdad observacional sea una congruencia, en el caso del TAD Persona la operación nombre debe ser también observador.

2.2. Observadores básicos

Los observadores básicos son un conjunto de funciones pertenecientes al TAD que permiten particionar el universo de sus instancias en clases de equivalencia, con la idea de agrupar en cada clase a las instancias que posean un comportamiento similar con respecto al estudio que

queremos realizar. En particular, deseamos que el TAD se convierta en una congruencia, es decir, una relación de equivalencia en la que si se aplica cualquier operación a dos instancias de la misma clase, los resultados obtenidos también formen parte de la misma clase (correspondiente al TAD al que pertenezca el resultado).

No hay una regla fija para identificar el conjunto de observadores básicos. De hecho, para cualquier TAD hay varios de tales conjuntos no *isomorfos* entre sí (es decir, más allá del obvio renombramiento de funciones). Es importante, a la hora de elegir los observadores, tener en claro cuándo vamos a considerar que dos instancias son iguales (i.e.: mediante la igualdad observacional). Si comparamos instancias observacionalmente iguales no debería pasar que al aplicar un observador a ambas obtengamos resultados observacionalmente distintos, porque en ese caso el TAD no se comportaría como una congruencia (propiamente dicho, sus funciones no lo harían). Si las instancias no eran observacionalmente iguales, entonces no podrá pasar que al aplicar todos los observadores a ambas obtengamos los mismos resultados. Otro dato a tener en cuenta es que el conjunto de observadores debería ser minimal, es decir, no deberían existir observadores que sólo identifiquen aspectos de la instancia que ya han sido identificado por los otros observadores.

Hay una relación muy cercana entre el conjunto de observadores básicos y la igualdad observacional. Ambos nos permiten distinguir cuando dos instancias de un TAD se comportan de la misma manera a los efectos de nuestro estudio. La diferencia principal radica en que los observadores "viven" adentro de la especificación del TAD, forman parte de su signatura y son axiomatizados para determinar su comportamiento, mientras que la igualdad observacional es un concepto semántico, un predicado que "vive" fuera del TAD. Una manera muy común de describir la igualdad observacional entre dos instancias es con un predicado construido a partir de la igualdad entre sus observadores básicos. Notar que por ende las clases de equivalencia que generan los observadores básicos y la que genera la igualdad observacional son las mismas.

2.3. Generadores

Los generadores son un conjunto de funciones que retornan un resultado del género principal del TAD especificado, y que tienen la particularidad de que a partir de una *aplicación finita* de ellos se pueden construir (o generar, de ahí su nombre) *absolutamente todas* las instancias del TAD.

No puede existir una instancia de un TAD que no se pueda generar a partir de una sucesión de aplicaciones de sus generadores. Si nos encontramos en ese caso, seguramente nos falta definir algún generador que permita generar dicha instancia. Contrariamente, nada impide que definamos generadores de más, es decir, que solamente generen instancias que ya podían ser generadas a partir de los otros. Sin embargo, no es recomendable esta práctica, ya que dificultaría bastante la axiomatización de las funciones (habría que describir el comportamiento similar de cada función con respecto a instancias generadas por generadores distintos), lo que lleva a la posible aparición de errores e inconsistencias en la especificación.

El conjunto de generadores puede ser particionado de la siguiente manera:

- Los que no reciben como parámetro *ninguna* instancia del tipo que están generando, a los que llamaremos *generadores base* o *no recursivos*.
- Los que reciben como parámetro *al menos una* instancia del tipo que están generando, a los que llamaremos *generadores recursivos*.

Luego, todas las instancias de un TAD están generadas o bien a partir de un generador base, o bien a partir de la aplicación de un generador recursivo a otra/s instancia/s del TAD. Este punto es fundamental a la hora de realizar demostraciones de propiedades sobre tipos abstractos de datos, ya que nos ofrece un esquema de demostración dividido en dos partes. La primer

parte demuestra la propiedad para todas las instancias generadas por generadores base, y la segunda demuestra la propiedad para todas las instancias generadas por generadores recursivos, suponiendo que la propiedad es válida para las instancias del TAD que reciben como parámetros (que a su vez están generadas a partir de generadores base o recursivos). Este esquema de demostración es conocido como *inducción estructural* y se verá en detalle más adelante en la materia.

Es importante notar que al aplicar un generador recursivo a una instancia de un TAD (e.g.: "Ag$(1, \emptyset)$") *no se está modificando* la instancia que recibe como parámetro, ya que en nuestro lenguaje abstracto no existe la noción de "cambio de estado", sino que lo que se está haciendo es *generar una nueva instancia* basada en la anterior, y cuyo comportamiento podrá ser descripto cuando se apliquen los observadores básicos sobre ella. Recordar el concepto de *transparencia referencial* del paradigma funcional, que indica que los resultados de las funciones sólo dependen de sus argumentos.

2.4. Otras operaciones

En esta sección se agregan el resto de las operaciones que se necesiten declarar en un TAD, incluyendo las operaciones auxiliares que no se exportan. No debería ocurrir que una función que aparezca en esta sección devuelva valores distintos cuando se aplique sobre dos instancias observacionalmente iguales del TAD. Si esto ocurre, habrá que replantearse el conjunto elegido de observadores básicos, ya que no están cumpliendo con el fin requerido.

2.5. Restricciones

En el formalismo de los tipos abstractos de datos sólo se permite especificar funciones totales, es decir, aquellas que están definidas para todo su dominio. Técnicamente, lo que estamos haciendo no es restringir el dominio de las funciones sino definirlas sólo para la parte del dominio que nos interesa. La diferencia es demasiado sutil y probablemente no tenga sentido a esta altura de la carrera, pero ése es el motivo por el cuál utilizamos predicados del metalenguaje (ver sección 1.6) para restringirlas.

El ejemplo paradigmático de restricción está relacionado con la operación de división: no se puede dividir por cero. Por eso, la división tiene restringido su dominio de aplicación, no se puede tomar cualquier número y dividir por él.

Una restricción es, justamente, una declaración sobre las propiedades de los distintos argumentos de una función. Bien pueden hablar sobre una propiedad que tienen que cumplir los argumentos en sí mismos o bien, más generalmente, una relación que debe darse entre ellos.

De la misma manera, durante la materia encontraremos muchos casos donde una operación que estamos definiendo no tiene sentido en ciertos casos. Por ejemplo, si nos piden que axiomaticemos una función para calcular el número anterior a un natural dado *debemos* exigir que el número sea mayor a cero. La función que especificaremos no tiene sentido si el parámetro es cero. No importa que, eventualmente, podamos tomar la convención de devolver cero en ese caso: esa decisión es incorrecta.

Otro motivo, menos subjetivo, por el cual es necesario declarar restricciones es que en la axiomatización realizada de la función usemos otras funciones que tienen restricciones, y las usemos sin validar el uso efectuado. Por ejemplo, si para axiomatizar $f : \text{nat } x \longrightarrow \text{nat}$ usamos $g : \text{nat } x \longrightarrow \text{nat}$ que tiene como restricción $x > 2$ y la axiomatización para f es

$$f(x) \equiv g(x) + 3 \tag{1}$$

es necesario que f pida que su argumento sea mayor a dos, pero si la axiomatización es

$$f(x) \equiv \textbf{if } x > 2 \textbf{ then } g(x) + 3 \textbf{ else } g(2 + x) + 3 \textbf{ fi} \tag{2}$$

no es necesario hacerlo, pues f misma garantiza no llamar a g en un caso donde no está *definida* (una función esta definida únicamente para los valores de los argumentos que cumplen sus restricciones).

En vista de este último motivo nos surge una pregunta que siempre nos tenemos que hacer al realizar una axiomatización: ¿estamos llamando a las funciones que utilizamos correctamente?

Si la respuesta a esta pregunta es negativa, bien pueden suceder dos cosas: que simplemente baste con agregar las restricciones a la nueva función o que nos demos cuenta que la axiomatización realizada es incorrecta.

Las restricciones son una parte fundamental de un TAD, con ellas explicitamos los casos para los cuales ciertas operaciones no tienen sentido, aportan claridad y coherencia a una especificación. Aportan expresividad a nuestro lenguaje ya que nos permiten limitar (aún más que el tipado de los argumentos de las operaciones) el universo al cual aplican ciertas operaciones de nuestros TADs.

Por eso, es bueno valerse de esa expresividad adicional que ellas nos dan para dejar denotado formalmente todas las salvedades que han de hacerse en el uso de nuestras funciones. Sin querer presentar una "regla de oro", si tenemos una función que bien puede no indefinirse para ciertos argumentos pero nosotros, a priori, no queríamos considerar esos valores es preferible restringirlos a dejar la función sin restricción alguna.

2.6. Géneros

En esta sección se incluyen todos los géneros nuevos que se describen en el TAD. El género es el nombre colectivo con el que se va a hacer referencia a instancias del TAD que estamos definiendo. En algunos casos, podríamos definir varios géneros dentro de un mismo TAD, pero no nos detendremos en esto por ahora.

Para establecer una analogía, podríamos pensar que el conjunto $\mathbb{N}$ es el género de los números naturales (pensados éstos no sólo como un conjunto de valores, sino también como operaciones entre ellos).

2.7. Usa

En esta sección se incluyen los nombres de los TADs que necesitaremos para definir nuestro nuevo tipo. Nuestro TAD podrá utilizar todos los elementos que aparecen en la sección *Exporta* de estos TADs. Se pueden necesitar, por ejemplo, para declarar la aridad de las funciones, para axiomatizar a partir de funciones de los TADs usados, etc.

Desde el punto de vista formal, lo que estamos haciendo es incluir otros teorías en la que estamos definiendo.

2.8. Exporta

En esta sección se incluyen todos los elementos declarados en el TAD que querramos que puedan ser utilizados por otros TADs. Por lo general, en esta sección siempre se incluyen a los géneros, los observadores básicos y los generadores. Las otras operaciones también pueden incluirse, pero la idea es que queden afuera todas aquellas funciones auxiliares que fueron declaradas solamente para facilitar la descripción del comportamiento de otras más importantes.

2.9. Estructura de un TAD

TAD NOMBRE_DEL_TAD

igualdad observacional

$$(\forall A, B : \text{elGénero}) \left(A =_{\text{obs}} B \iff \begin{pmatrix} (\text{Obs}_1(A) =_{\text{obs}} \text{Obs}_1(B)) \wedge \\ (\forall\, q : \text{loquesea})\ (\text{Obs}_2(q, A) =_{\text{obs}} \text{Obs}_2(q, B) \wedge \\ (\text{Obs}_2(q, A)\ \Rightarrow\ (\text{Obs}_3(q, A) =_{\text{obs}} \text{Obs}_3(q, B))) \ \ldots \end{pmatrix} \right)$$

(Acá va la igualdad observacional, escrita en lógica de primer orden).

exporta *(Acá van los géneros, observadores básicos, generadores y todas las operaciones que puedan ser utilizadas desde afuera, separados por comas).*
Ej.: elGénero, observadores básicos, generadores, OO_1, OO_2

usa *(Los nombres de los TADs que se usan en esta especificación, separados por comas).*
Ej.: Nat, Bool, Secu(Nat), LoQueSea, ...

géneros *(Él o los géneros que se estén definiendo. Si sólo estamos extendiendo un TAD y no agregamos géneros nuevos, esta parte se omite).*

extiende *(Esto sólo se incluye cuando estamos extendiendo otro TAD ya existente; en ese caso, acá va el nombre de ese TAD).*

observadores básicos

La sintaxis es la que sigue:

NombreDelObservador : parámetro$_1$ × parámetro$_2$ × ... ⟶ género_de_salida
{Restricción, si la hubiera}

Ejemplos:

Obs_1 : elGénero ⟶ secu(nat)
Obs_2 : loquesea × elGénero ⟶ bool
Obs_3 : loquesea q × elGénero A ⟶ nat {$\text{Obs}_2(q, A)$}

Nota: las restricciones son predicados en lógica de primer orden. Sólo tiene sentido evaluar la función si la restricción se cumple. En una restricción, como en cualquier otro predicado, se pueden usar cuantificadores, operadores lógicos y funciones. Si una función se exporta, las funciones que se hayan utilizado para escribir su restricción deben exportarse también (si no, no se podría saber cuándo una función del TAD puede usarse).

generadores

La sintaxis similar a la de los observadores, con la diferencia de que un generador puede no recibir parámetros, en cuyo caso crea una nueva instancia "de la nada".

Ejemplos:

G_1 : ⟶ elGénero
G_2 : loquesea q × elGénero A ⟶ elGénero {$\neg\ \text{OO}_1(q, A)$}
...

otras operaciones

Acá van todas las operaciones necesarias (ya sea porque el problema las requiere o porque se usan en los axiomas de otras), que no sean observadores ni generadores. La sintaxis para estas funciones es igual a la de los observadores básicos.

OO_1 : loquesea q × elGénero A ⟶ bool
OO_2 : elGénero ⟶ nat
OO_{aux} : elGénero × nat ⟶ nat
...

Obviamente, los nombres deberían ser más declarativos que éstos. Esto es sólo un ejemplo para mostrar la sintaxis de un TAD.

axiomas

$\forall$ loquesea q, p, $\forall$ elGénero A, $\forall$ nat $n \leftarrow$ *(todas las variables que aparezcan en los axiomas)*

De acá en adelante van todos los axiomas que se necesiten para darle sentido a la signatura (es decir a lo que habíamos escrito hasta ahora).

Un axioma tiene la forma:

$T_1 \equiv T_2$

donde T_1 y T_2 son términos que representan instancias de un mismo género.

$\text{Obs}_1(G_1) \equiv <>$

$\text{Obs}_1(G_2(q, A)) \equiv \text{operaciónDeLoQueSea}(q)$ & $\text{Obs}_1(A)$

...

Fin TAD

También es posible definir renombres de tipos para que resulte más claro en el contexto en que se los usa.
Por ejemplo:

TAD Edad es Nat
TAD String es Secuencia(Char)
TAD Nombre es String

3. Conceptos importantes a la hora de escribir un TAD

3.1. Recursión

La recursión está íntimamente relacionada con el principio de inducción que se ha estudiado en Álgebra I. La utilizaremos habitualmente cuando axiomaticemos funciones en nuestros TADs.

Una definición es recursiva si hace referencia a sí misma. Por ejemplo, podemos definir un ramo de rosas como:

1. Una rosa.
2. Una rosa junto a otro ramo de rosas.

Vemos que en el segundo caso hacemos uso del mismo concepto que estamos definiendo:

> "...la recursividad parece aproximarse mucho a la paradoja. Por ejemplo, las *definiciones recursivas*, las cuales pueden dar la impresión de que se está definiendo algo en función de ello mismo. Esto implicaría una circularidad, y conduciría a una regresión infinita, si no a la paradoja misma. En verdad, una definición recursiva (si es adecuadamente formulada) jamás conduce a una regresión infinita ni a una paradoja. Y es así porque una definición recursiva nunca define una cosa en función de esa cosa sino, siempre, en función de las *interpretaciones más simples* de la misma."
> **D.R. Hofstadter** - *"Gödel, Escher, Bach"*

La idea es que al ir simplificando se llega a un punto donde no se puede simplificar más: el *caso base*, tal como existe en inducción (el primer caso del ejemplo de las rosas). Allí la definición (o axioma, para decirlo de acuerdo al contexto donde usaremos recursión) se resuelve directamente sin usar el concepto que se está definiendo, lo importante es que para resolver el caso base nos basta con saber qué tiene que devolver la función para ese caso particular, lo cual es relativamente sencillo.

La autoreferencia a la definición que se está dando se realiza en el *caso recursivo*, donde se descompone (se obtienen las interpretaciones más simples) el objeto sobre el cual se está definiendo y se aplica la definición a esas simplificaciones. Al ir simplificando en cada caso llegará un punto donde no se pueda simplificar más y lleguemos al caso base. Eso es una propiedad fundamental que tienen que cumplir nuestras definiciones recursivas: tenemos que garantizar

que eventualmente se llegará al caso base para todos los valores sobre los que se encuentra definida la operación (de acuerdo a la restricción). La manera más habitual de garantizar esto es que en cada caso recursivo se logre *disminuir la complejidad* de los parámetros involucrados.

Ahora bien, la autoreferencia que hemos dicho caracteriza a las definiciones bien puede darse de manera indirecta. Como en la figura 1, cada mano no se dibuja a sí misma sino que dibuja a la mano que la dibuja.

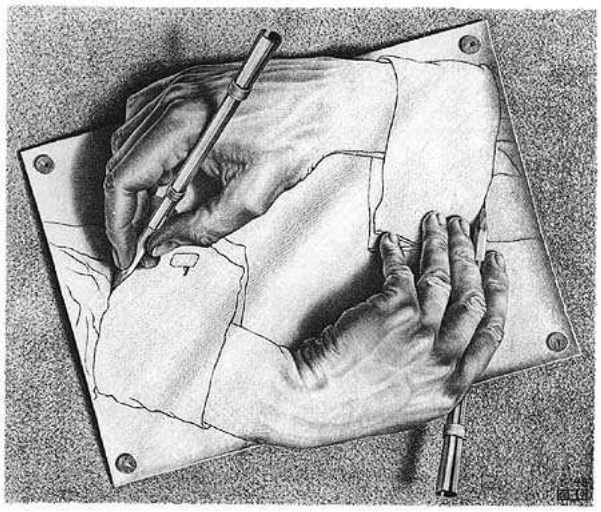

Figura 1: "Drawing Hands" — M. C. Escher

Lo que encontramos en estos casos es *recursión mutua*; donde una definición no hace autoreferencia directamente sino lo hace a través de otra definición. Claramente, la recursión mutua puede darse en más de dos niveles.

En estos casos, las consideraciones respecto de la disminución de la complejidad han de ser realizadas; lo único que cambia es que ahora es un poco más complicado poder garantizar esa disminución en los sucesivos pasos.

Un ejemplo elemental de recursión mutua podría plantearse entre dos funciones par : nat $\longrightarrow$ bool e impar : nat $\longrightarrow$ bool, que podrían definirse de la siguiente manera:

$$\begin{aligned} \text{par}(0) &\equiv \text{true} \\ \text{par}(x+1) &\equiv \text{impar}(x) \\ \text{impar}(0) &\equiv \text{false} \\ \text{impar}(x+1) &\equiv \text{par}(x) \end{aligned}$$

Vemos como ambas funciones se llaman mutuamente, cada una tiene un caso base y un caso recursivo. En cada caso recursivo nos acercamos al valor del caso base de ambas funciones (que coincide en este ejemplito).

Cuando planteamos una recursión debemos, fundamentalmente, concentrarnos en resolver cada caso particular correctamente sin preocuparnos por los otros. Luego si cada caso se resuelve bien, el conjunto también será correcto. Si uno está pensando en varios casos en lugar de enfocarse en cada uno en particular suelen presentarse confusiones; al axiomatizar $par(x+1)$ uno está seguro de que $x+1$ es par si y sólo si x es impar, ya luego veremos cómo resolver impar de x.

3.2. No axiomatizar sobre casos restringidos

Volviendo al tema de las restricciones, su existencia implica, justamente, que la función no se encuentra definida para ciertos valores de sus argumentos. Más allá de las consideraciones ya realizadas, esto implica que a la hora de axiomatizar una función con restricciones no se ha de realizar ningún tipo de consideración para "controlar" que los argumentos cumplan efectivamente la restricciones: al momento de axiomatizar todos los argumentos siempre cumplen las restricciones.

Luego puede darse el caso de que sobre ciertos generadores del tipo sobre el que se especifica la función ni siquiera se plantée un axioma. Si h : secu(nat) $S \longrightarrow$ bool tiene como restricción "$\neg$vacia?(S)" sería incorrecto plantear un axioma de la forma $h(<>) \equiv \ldots$.

La existencia de este tipo de restricciones hace que, eventualmente, una función tenga un caso base sobre una generador recursivo. Sin ir más lejos, h debería axiomatizarse como:

$$h(a \bullet S) \equiv \textbf{if } \text{vacia?}(S) \textbf{ then } \ldots \textbf{ else } \ldots \textbf{ fi} \tag{3}$$

Aquí el caso base se da para el generador $\bullet\bullet\bullet$: nat $\times$ secu(nat) $\longrightarrow$ secu(nat), que es un generador recursivo pues recibe un argumento del tipo que construye.

Tambíen estaría mal considerar en la parte derecha del axioma casos que no pueden darse; el siguiente axioma sería incorrecto:

$$h(S) \equiv \textbf{if } \text{vacia?}(S) \textbf{ then } \ldots \textbf{ else } \ldots \textbf{ fi}$$

Por la restricción de la función h, S nunca podría ser la secuencia vacía.

3.3. No axiomatizar sobre generadores de otros tipos

Resultará habitual a lo largo de nuestra práctica encontrarnos con situaciones donde querramos que un TAD que estamos especificando tenga una operación auxiliar que recibe argumentos de tipos definidos en otros TADs. Al realizar estas axiomatizaciones lo preferible es que ellas se efectuen en función de los observadores del tipo usado y no sobre los generadores. Formalmente son dos alternativas válidas pero basarse en generadores puede ocasionar problemas que con los observadores básicos nunca se presentan: descartaremos el uso de los generadores de otros tipos a la hora de realizar axiomatizaciones en estas condiciones.

El problema que puede ocasionarse es que al usar los generadores del otro tipo nuestra operación viole la igualdad observacional de este tipo usado; en cambio usando los observadores esto nunca podrá darse. De alguna manera, si axiomatizamos una operación con los generadores nos estamos atribuyendo la potestad de darles una cierta interpretación a la estructura sintática que ellos tienen: ésa es la tarea de los observadores del tipo en cuestión. Dentro de un mismo tipo el hecho de definir una "otra operación" en función de los generadores también deja la posiblidad de una violación de la igualdad observacional, pero en este caso lo permitimos ya que todo queda en el contexto de un único TAD.

3.4. Utilización correcta de los TADs básicos

Durante la materia los TADs más utilizados son aquellos que denominamos *TADs básicos*, entre los que se encuentran:

- Secuencia
- Conjunto
- Pila
- Cola

- Árbol
- Diccionario
- ...

Muchas veces nos enfrentaremos con la situación de tener que optar por alguno de ellos frente a los otros. En estos casos, siempre hemos de tomar la alternativa más económica. Debemos elegir aquél TAD que más se ajusta a nuestras necesidades; entre dos TADs que las satisfacen suele convenir elegir el que menos características presenta.

El ejemplo típico de estas situaciones es aquél donde se nos pide informar un grupo de elementos para conocer qué elementos aparecen en él; lo que nos interesa es la pertenencia, con lo cual debemos pensar en un conjunto y no en una secuencia, la cual además de responder lo que nos interesaba también nos aporta información sobre el orden de los elementos y, eventualmente, sus repeticiones. Menos aún se debería pensar en, por ejemplo, un árbol.

Para poder discernir qué TAD tenemos que usar debemos, primero, definir que es lo que se quiere "ver" de acuerdo al problema que nos ocupa. Luego, teniendo en claro las necesidades, debemos hacer un repaso mental de lo que cada TAD provee (de acuerdo a sus observadores básicos) y elegir de acuerdo al criterio de economía mencionado.

3.5. Parámetros formales

Muchas veces nos encontramos ante la necesidad de definir un TAD que, potencialmente, puede hacer uso de muchos otros para definir su comportamiento. Sin embargo, también es cierto que, en gran parte de esas ocasiones, no importa realmente de *qué* TADs se trata, sino más bien de *cuáles son sus características*. Tomemos al conjunto, por ejemplo. Nuestra intención, presumiblemente, es la de modelar una "bolsa" donde contener varias instancias de otro TAD. Sería extremadamente engorroso y repetitivo esbozar un TAD para cada posible contenido[5], razón por la cual escribimos

TAD Conj(elem)

Ahora bien, ¿qué significa ese *elem* en la definición del TAD? ¿Es *elem* un TAD?

Pues no. *elem* es lo que llamamos un *parámetro formal*. Así como dentro de la axiomatización de las funciones utilizamos cierto *identificador* para las *variables* que forman parte de la axiomatización, *elem* puede entenderse como una *variable de tipo*.

Así las cosas, podríamos seguir y definir, por ejemplo, la secuencia ordenada de *algo*. Escribiríamos entonces

TAD SecuenciaOrdenada(elem)

Sin embargo, podemos notar que el esquema de utilizar parámetros formales no alcanza en este caso. Las secuencias ordenadas dependen fuertemente de que el TAD subyacente posea una relación de *orden*. ¿Cómo salvamos esta situación? Si leemos un poco más arriba, recordaremos que nos importaba cuáles eran las características del TAD que utilizaríamos. En particular, el hecho de tener una relación de orden es una de tales características. Simplemente, tendremos que requerir ese hecho. Esto lo notamos de la siguiente manera:

TAD SecuenciaOrdenada($\text{elem}_{\leq}$)

De esta forma, indicamos que se pueden utilizar aquellos *elem* que poseen una relación de orden $\leq$. Es válido notar que podemos requerir no sólo una relación de orden; podríamos exigir que exista una relación de equivalencia $=$ (por ejemplo, en el caso de las claves de un diccionario), o cualquier otra cosa que se nos ocurra o necesitemos.

Claramente, también debemos tener cuidado de no abusar de esta facilidad (o mejor dicho, de no restringir de más los posibles TADs). Por ejemplo, definir el TAD Secu o Conjunto

[5] Incluso, podría haber infinitos TADs cuyo comportamiento nos sirviese.

exigiendo cierta particularidad a los TADs sobre los que los definimos, nos quita flexibilidad y no nos ayuda a modelar lo que realmente estamos buscando.

3.6. Interfaces gruesas

Para meternos de lleno dentro del tema de las interfaces gruesas y finas, nada mejor que un ejemplo: supongamos que vamos a realizar un trámite y nos piden que nos identifiquemos. A esto respondemos algo como "mi nombre es Juan Pérez, mido un metro ochenta, nací tal o cual día, mis padres son Fulano y Mengana de Tal", etc. Por un lado, cumplimos con el objetivo de "identificarnos", pero... ¿era necesaria tanta información?

Algo similar ocurre cuando definimos las funciones que describen nuestro TAD. Si dentro de él tenemos una función que devuelve cierto dato asociado a una persona, es innecesario que dicha función utilice una instancia del TAD Persona. Tal vez le alcance sólo con conocer el DNI, por ejemplo. A esta situación de proveer más datos que los necesarios es que nos referimos cuando decimos que un TAD provee una "*interfaz gruesa*". En general, las interfaces gruesas no son deseables ya que, al introducir información que no hace al objetivo de la función, "ensucian" y hacen más confusa la especificación. Pero entonces... ¿cómo evitamos las interfaces gruesas?

A veces, un indicador de que "algo anda mal" es el uso excesivo de los observadores dentro de la axiomatización de la función. Es decir, si no utilizamos toda la instancia que tenemos, sino que proyectamos sistemáticamente una de las caracterísitcas de la instancia, vale preguntarse si no correspondía tener sólo esa característica en primer lugar. Sin embargo, esto no es una "receta" y debe ser analizado en cada caso.

3.7. Buena práctica

A continuación se presentan algunos consejos para mejorar la calidad de los TADs que escribimos.

- Las cosas, por su nombre. Para mejor comprensión del TAD, se deben utilizar *nombres declarativos* al definir operaciones, géneros, etc., deben ser descriptivos y estar relacionados con el objetivo que deben cumplir. Se deben evitar denominaciones tales como "aux", "f", etc. Un nombre erróneo también puede traer problemas en la axiomatización de la función, por ejemplo, si se quiere obtener el anteúltimo elemento de una secuencia (de al menos dos elementos) y llamamos Segundo : $\text{secu}(\alpha) \longrightarrow \alpha$ a la operación que realiza dicha acción, por más que tengamos claro cual es el comportamiento de la operación, es posible que el nombre mal elegido nos haga confundir a la hora de axiomatizarla.
- Siempre es conveniente agregar un breve comentario en castellano describiendo el objetivo de cada operación, principalmente en aquellas cuyo comportamiento no es trivial.
- La especificación debe ser de fácil lectura. Muchas veces una especificación no es adecuada, no porque el comportamiento que describe sea incorrecto, sino porque su nivel de claridad y descriptividad es muy bajo.

Referencias

[BBK95] Gilles Bernot, Michel Bidoit, and Teodor Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Computer Science*, 139(1–2):275–314, 1995.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* Springer-Verlag, 1985.

[GHG+93] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. 1993.

[LG89] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1989.