Carreras de “Ingeniería y Licenciatura en Sistemas”

Cátedra de “Algoritmos y estructuras de datos II”

Apunte “Especificación de tipos abstractos de datos”

por Sergio Amitrano

Indice

Sobre la especificación de TADs

Objetivo

En este apunte se introduce el concepto de especificación de un tipo abstracto de datos (TAD), para diferenciarla de su implementación en un lenguaje imperativo clásico. La naturaleza de este tipo de especificación es declarativa funcional, basada en axiomas.

Si bien hay mucha información relacionada con este tema, algunas cuestiones como la sintaxis y el alcance de la semántica (significado) de algunos de sus componentes no logran estar unificados en la bibliografía. Es por eso que en este apunte se intentará describir cuáles son las particularidades que tomaremos en cuenta sobre este tipo de especificación, y cuáles son las cuestiones particulares que las diferencian con el resto de las especificaciones.

Introducción

La aparición del concepto de tipos abstractos de datos (TADs) fue un gran avance dentro del ámbito de la evolución de los lenguajes de programación porque aportó la idea de encapsulamiento (ocultamiento de la información) a los tipos de datos definidos por el usuario, además del ya conocido concepto de abstracción de tipos. Este ocultamiento de la información aplicaba a la estructura del tipo de datos concreto con el que se implementaba el tipo abstracto, así como el código de sus operaciones que podían ser utilizadas por otros TADs, y algunas operaciones de uso interno.

Sin embargo, el hecho de manejar en un TAD nuevos conceptos que implicaban no solamente modelarlos con una estructura (el tipo de dato concreto de su representación) y un comportamiento (las operaciones que lo manipulaban) reflejaba que en la implementación se mezclaban cuestiones referidas a la esencia o naturaleza de lo que se quería representar con cuestiones puras de la implementación en sí misma. Algo así como un alto nivel en relación al **qué** era lo que se quería representar (la esencia/naturaleza) y un bajo nivel en relación al **cómo** se lo quería implementar. Se ahí surge la idea de dividir el proceso de construcción de un TAD en su representación abstracta o especificación, y en su implementación en sí misma a partir de su especificación.

En lo que llamamos la especificación de un TAD tratamos de representar la esencia de los datos propios del TAD, con su estructrura y comportamiento. Esta representación deberá ser lo suficientemente clara, elegante y completa para comprender sus características centrales, aunque sin caer en detalles de implementación. En esta etapa no importa si la especificación es óptima en tiempo/espacio o simple/compleja, porque esos no son conceptos que se miden durante una especificación. Lo importante es la claridad y elegancia de la representación y de la funcionalidad de su operatoria.

En el momento de la implementación a partir de la especificación de un TAD, ahí se podrá optimizar el espacio de representación del tipo sobre el que se define el TAD, y el tiempo de ejecución de sus operaciones implementadas cuando éstos sean necesarios, con la posibilidad de perder elegancia y claridad, aunque sin pérdida de consistencia en el manejo de la información ni de coherencia en la operatoria de sus acciones definidas.

Sintaxis de la especificación

Hay muchas formas de especificar un TAD, y una conocida es una especificación funcional basada en axiomas o reglas funcionales. En ella, el tipo sobre el que se definen los elementos básicos del TAD se establece a partir de constructores, que pueden ser, o bien *constantes constructoras*, o *funciones constructoras*. Ambos generan expresiones básicas de ese tipo, pero las funciones constructoras además reciben como parámetros elementos de otros tipos. Las reglas funcionales o axiomas se utilizan para definir el comportamiento abstracto de las operaciones que aplican sobre los elementos del TAD definidos. Las operaciones representan principalmente funciones (y nunca procedimientos) que se definen con la esencia de las definiciones declarativas funcionales: con transparencia referencial (el resultado de la evaluación de una función depende exclusivamente de su definición propiamente dicha y de sus parámetros, siendo independiente del contexto en donde esta función se evalúa) y con falta de efectos colaterales (el resultado de la evaluación de la función solamente afecta al resultado que retorna, sin alterar el resto del contexto). Si se quisiera definir una operación que no representa una función porque altera el contexto (por ejemplo, una operación que modifica un valor de un atributo de una expresión del TAD), se deberá definir como si fuera una función -sin efectos colaterales-, que en lugar del modificar el atributo, retorne la misma expresión original, salvo con el valor del atributo cambiado (para "simular" esta acción).
Los axiomas de una función determinan cómo se evalúa esa función a partir del parámetro-subexpresión recibida, y cuál será su resultado. Este resultado, a su vez se podrá expresar como una expresión que representa una evaluación de otra función a su expresión-parámetro, la cual se evaluará. Este proceso puede continuar hasta llegar a una expresión irreducible (que ya no se puede evaluar).

Una especificación de un TAD tiene la siguiente sintaxis:

TAD *idtipo*[**(***vtipo*$_1$**,** ... **,** *vtipo*$_n$**)**]
exporta *idop*$_1$**,** ... **,** *idop*$_n$
[**usa** *idtipo*$_1$[**(***exptipo*$_{11}$**,** ... **,** *exptipo*$_{1k}$**)**]**,** ... **,** *idtipo*$_n$[**(***exptipo*$_{n1}$**,** ... **,** *exptipo*$_{nr}$**)**]]
constructores
idcons$_1$**:** [*exptipo*$_{11}$ [**×** ... **×** *exptipo*$_{1k}$] **→**] *idtipo*[**(***vartipo*$_1$**,** ... **,** *vartipo*$_n$**)**]
...
idcons$_g$**:** [*exptipo*$_{g1}$ [**×** ... **×** *exptipo*$_{gk}$] **→**] *idtipo*[**(***vartipo*$_1$**,** ... **,** *vartipo*$_n$**)**]
proyectores
idproy$_1$**:** *exptipo*$_{11}$ [**×** ... **×** *exptipo*$_{1k}$] **→** *exptipo*$_1$
...
idproy$_u$**:** *exptipo*$_{u1}$ [**×** ... **×** *exptipo*$_{us}$] **→** *exptipo*$_u$
operaciones
idop$_1$**:** [*exptipo*$_{11}$ [**×** ... **×** *exptipo*$_{1k}$] **→**] *exptipo*$_1$
...
idop$_t$**:** [*exptipo*$_{t1}$ [**×** ... **×** *exptipo*$_{tw}$] **→**] *exptipo*$_t$
axiomas
idproy$_{11}$**(***pat*$_{111}$**,** ... **,** *pat*$_{11n}$**)** **=** *exp*$_{11}$
...
idproy$_{1h}$**(***pat*$_{1h1}$**,** ... **,** *pat*$_{1hn}$**)** **=** *exp*$_{1h}$
...
idproy$_{u1}$**(***pat*$_{u11}$**,** ... **,** *pat*$_{u1s}$**)** **=** *exp*$_{11}$
...
idproy$_{uh}$**(***pat*$_{uh1}$**,** ... **,** *pat*$_{uhs}$**)** **=** *exp*$_{1h}$

idop$_{11}$**(***pat*$_{111}$**,** ... **,** *pat*$_{11k}$**)** **=** *exp*$_{11}$
...
idop$_{1g}$**(***pat*$_{1g1}$**,** ... **,** *pat*$_{1gk}$**)** **=** *exp*$_{1g}$
...
idop$_{t1}$**(***pat*$_{t11}$**,** ... **,** *pat*$_{t1w}$**)** **=** *exp*$_{t1}$
...
idop$_{th}$**(***pat*$_{th1}$**,** ... **,** *pat*$_{thw}$**)** **=** *exp*$_{th}$

El texto subrayado representa texto literal, y los corchetes ([,]) reflejan la opcionalidad del texto contenido entre ellos.

Los identificadores *idcons*, *idproy* y *idop* representan identificadores de constructores, proyectores y operaciones respectivamente.

El identificador *idtipo* representa un identificador de tipo de datos. Si este identificador representara un tipo estructurado con *n* variables de tipo, luego de él se detallarán las *n* expresiones de tipo *exptipo*$_1$ a *exptipo*$_n$ entre paréntesis, y con comas separadoras entre ellas.

El formato de una expresión de tipos *exptipo* es la siguiente:

exptipo ::= *idtipo*[**(** *exptipo*$_1$ **,** ... **,** *exptipo*$_k$ **)**]
::= *exptipo*$_1$ **→** *exptipo*
::= *exptipo*$_1$ **×** ... **×** *exptipo*$_k$ **→** *exptipo*
::= *vtipo*

vtipo ::= *vartipo* [**con operaciones**
idop$_1$**:** [*exptipo*$_{11}$ [**×** ... **×** *exptipo*$_{1k}$] **→**] *exptipo*$_1$ **,**
...
idop$_s$**:** [*exptipo*$_{s1}$ [**×** ... **×** *exptipo*$_{st}$] **→**] *exptipo*$_s$]

donde el primer caso representa un tipo básico o estructurado, el segundo representa un tipo función (→) con una expresión de tipos en el dominio, y el tercero representa un tipo función con *n* expresiones de tipo en el dominio. En realidad, el dominio está conformado por el producto cartesiano (×) entre las *n* expresiones de tipo. Para ambas situaciones de las expresiones de tipo que representan una función, se especifica también una expresión de tipos en su imagen. En el cuarto caso, representa una variable de tipos. Esta variable de tipos puede tener restricciones, que consisten en que sobre esa variable de tipos deben estar definidas ciertas operaciones reconocibles (exportadas) en las que la misma aparece como el tipo de alguno de sus parámetros de los dominios o de las imágenes, aunque sin importar la definición de dichas operaciones (estas operaciones estarían sobrecargadas). Una variable de tipos hace referencia a cualquier tipo existente (básico o estructurado). Si la variable de tipos tuviera alguna restricción sobre la existencia de determinadas operaciones indicadas, ésta solamente hará referencia a cualquier tipo que tenga definidas las operaciones indicadas, sustituyendo en su declaración la variable de tipos por ese tipo concreto. Estas restricciones sobre las variables de tipo aplican para la variable de tipos en cuestión y para todas las apariciones de esa misma variable dentro del mismo alcance, sin necesidad de escribir la misma restricción para las otras apariciones de la misma variable en el mismo alcance (la declaración completa de una operación -proyector u operación restante- o la definición del tipo en el primer renglón -en este último caso el alcance de esta restricción sería "global"-).

A efectos de la tipificación, tanto los identificadores *idcons*, *idproy* como *idop* se caracterizan por tener cada uno una expresión de tipo asociada, que expresa el tipo de cada uno de ellos. El lugar de la especificación donde esto se indica para estos identificadores, se dice que corresponde a la declaración de ellos.
Si alguno de ellos fuera de tipo función con dos expresiones de tipo en su dominio, y se quisiera declarar como infijo (con sus dos expresiones-parámetro a ambos lados de su nombre), se deberán declarar de la siguiente manera:

_ *idcons* _
_ *idproy* _
_ *idop* _

Los datos propiamente dichos corresponden a lo que serían las expresiones, aquí expresadas como *exp*. El formato de las expresiones *exp* es el siguiente:

exp	::=	*idcons*	(que no sea de tipo función)
	::=	*idcons* **(** exp_1 **,** ... **,** exp_n **)**	(con $n > 0$)
	::=	*idproy* **(** exp_1 **,** ... **,** exp_n **)**	
	::=	*idop* **(** exp_1 **,** ... **,** exp_n **)**	
	::=	exp_1 *idcons* exp_2	(*idcons* declarado infijo)
	::=	exp_1 *idproy* exp_2	(*idproy* declarado infijo)
	::=	exp_1 *idop* exp_2	(*idop* declarado infijo)

Así como a efectos de la tipificación, los identificadores *idcons*, *idproy* y *idop* tienen un tipo de datos asignado, también las expresiones *exp* tienen un tipo asignado referenciado como *exptipo*. En general, el tipo de *exp* corresponde a la expresión de tipos de la imagen del identificador que lo compone (o el tipo del identificador mismo, si éste no es de tipo función). Pero para que esto sea válido, y también a efectos de la tipificación, la cantidad de subexpresiones del identificador (n) debe corresponder a la cantidad declarada de componentes de su dominio, y el tipo de cada una de las subexpresiones que corresponden a los parámetros del identificador en cuestión deberá corresponder al tipo del parámetro correspondiente que aparece en la declaración de ese identificador. También el identificador deberá recibir sus parámetros en forma prefija (normal) o infija, de acuerdo a su declaración.

En el caso de utilizar variables de tipo en declaraciones de identificadores, a efectos de la tipificación, en las definiciones de los mismos esas variables de tipo no podrán corresponder a expresiones de las que por su uso de puede deducir que su tipo de datos es más específico.

Ahora se pasan a detallar cada una de las siguientes "secciones" de la especificación.

En la sección **`TAD`** se indica el nuevo tipo de dato a especificar en el resto del TAD, que se identificará como *idtipo*. Si el tipo correspondiera a un tipo estructurado, deberán indicarse luego de su nombre y entre paréntesis, tantas variables de tipo (*vtipo*) como las que sean necesarias. Si las variables de tipo tuvieran restricciones, éstas se aplican a todas las apariciones de la misma en todo el TAD. Las restricciones indicadas sobre estas variables de tipo no deberán indicarse en apariciones posteriores de la misma variable de tipos en el resto de la especificación del TAD.

En la sección **`exporta`** se indican aquellas operaciones y proyectores del TAD (que corresponden a las que tienen nombre $idop_i$) que serán reconocidas por otros TAD que lo usan. El resto de las operaciones o proyectores del TAD que no están indicadas ahí, no serán dados a conocer por los otros TADs que lo usen, lo mismo que la estructura interna de representación del TAD. Desde el punto de vista formal, los constructores no pueden ser exportados, aunque se podría asumir que sí pueden ser exportados, con la salvedad de que los TAD que los usan como operaciones deben desconocer su naturaleza real de constructores.

En la sección optativa **usa** se indican aquellos TADs externos que el TAD utiliza. El TAD solamente podrá utilizar de los TAD indicados (usados), aquellas operaciones de los mismos que están indicadas en las secciones **exporta** de cada uno de ellos. Los TAD usados son aquellos cuyos elementos aparecen como expresiones utilizadas en definiciones axiomáticas (sección **axiomas**) de operaciones o proyectores propios. Sobre estos elementos de los TADs usados que aparecen en las definiciones axiomáticas se le podrán aplicar operaciones o proyectores exportados por aquellos mismos TADs usados.
Si los TADs usados contuvieran tipos paramétricos (variables de tipo) en su nombre, ellos se podrán utilizar en su versión más general, utilizando esas variables tipo como parámetros de tipo, o versiones más particulares, sustituyendo las variables de tipo-parámetros por tipos más concretos, restringiendo el uso de los mismos a las sustituciones de esos tipos más particulares. Un mismo TAD paramétrico puede ser utilizado más de una vez por el mismo TAD (apareciendo más de una vez en su sección **usa**) con distintas sustituciones de sus variables de tipo.

En la sección **constructores** se indican los constructores del TAD. Los constructores permiten generar elementos "básicos e irreducibles" (no evaluables) como expresiones del tipo del TAD que se está especificando. Los constructores pueden ser de dos tipos: *constantes constructoras* o *funciones constructoras*. Las constantes constructoras son del tipo del TAD (incluso con sus tipos paraméticos variables, si los tuviera). Las funciones constructoras son de tipo función que reciben una expresión de un tipo dado (si recibiera más de un valor como parámetro estaría recibiendo un producto cartesiano de los tipos de sus parámetros) y retornan el tipo del TAD (también con sus tipos paraméticos variables, si los tuviera).
En esta sección se indican los nombres de todos los constructores del TAD -sean constantes o funciones- (de nombre $idcons_i$) con la declaración de los mismos luego del símbolo **:** que se ubican luego de cada uno de estos nombres.
Los constructores, al representar elementos "básicos e irreducibles", no tienen una definición axiomática de evaluación. Si una expresión representa una constante constructora, ésta no se puede evaluar. Si la expresión representara la evaluación de una función constructora que se aplica sobre sus distintas subexpresiones-parámetros, esta función constructora no se evaluará, evaluándose solamente sus subexpresiones-parámetros y manteniendo a la función.

En la sección **proyectores** se indican las declaraciones de aquellas operaciones que en particular se llaman proyectores. Los proyectores corresponden a las operaciones más básicas que se aplican sobre expresiones del TAD. Suelen ser operaciones que expresan cuál es el constructor principal del TAD sobre el que se define esa expresión, y para el caso en que la expresión se componga principalmente por una función constructora, retornan cada una de las subexpresiones parámetro de la función constructora principal. Estas operaciones sí tienen una definición, ya que las evaluaciones de funciones proyectoras están definidas según su definición axiomática posterior. Podemos decir que estas operaciones tal como fueron descriptas antes necesitan conocer la estructura interna (constructores) de las expresiones del TAD. Es por eso que las definiciones de estas

operaciones (en lo que veremos como axiomas) deberán acceder directamente a los constructores de la expresión del TAD en cuestión.
La bien la definición formal de función proyectora es la expresada arriba, podemos extenderla a todas aquellas operaciones tal que en su definición deberán indefectiblemente acceder a los constructores que determinan la estructura de la expresión del TAD para que su resultado sea posible. Con esto último, se podría extender la definición de proyectores que alcance a operaciones que corresponden a "operaciones constructoras" (que se definen en forma similar a los constructores, y que el TAD las exporta para que sean de uso por parte de otros TAD) y a operaciones relacionadas con la modificación de valores de atributos (que retornan el elemento completo modificado para simular la actualización imposible en un modelo axiomático funcional).
Es un error suponer que todos los proyectores de un TAD deban ser exportados por el mismo. En realidad, la naturaleza de los proyectores es muy dependiente de la definición del tipo concreto del TAD, y es posible que se quieran proveer a los otros TAD que lo usan, operaciones básicas que no "sugieran" la estructura del tipo concreto real.

En la sección **`operaciones`** se indican las declaraciones del resto de las operaciones, que también como los proyectores, tienen una definición axiomática posterior. La diferencia entre las operaciones aquí declaradas con los proyectores, es que al no ser operaciones "básicas", no es necesario que se definan axiomáticamente accediendo directamente a los constructores de la expresión, sino que se podrán definir utilizando (invocando) a las funcionales proyectoras para obtener las características esenciales de la expresión del TAD, sin necesidad de acceder a los constructores. También es necesario tener algún tipo de mecanismo para poder discriminar según lo que sería el formato de los constructores (al ser reemplazado por la invocación de proyectores). Esto se lograría con una pseudofunción, que es la función `if` (sería la versión funcional de la estructura de control `if..then..else..`). Está declarado y definido así (en forma infija de tres parámetros):

`if _ then _ else _ : Bool × a × a → a`	(declaración)
`if True then x else y = x`	(definición axiomática 1)
`if False then x else y = y`	(definición axiomática 2)

Esta definición suele estar presente en el TAD `Bool` (valores lógicos).
La definición de operaciones de esta forma puede ser elegante y más fácilmente mapeable a su procedimiento o función equivalente a nivel implementación imperativa, pero también es cierto que las definiciones axiomáticas accediendo directamente a los constructores suelen ser menos tediosas, especialmente si se desea acceder a los constructores de la expresión principal y a los constructores de sus subexpresiones parámetros en la misma definición. Se puede asumir que acceder a los constructores de las expresiones del TAD en las definiciones axiomáticas de operaciones no proyectoras no es una mala práctica.

En la sección **`axiomas`** se describen cada una de las definiciones axiomáticas de cada una de los proyectores y operaciones comunes. Cada una de las operaciones proyectoras y las comunes deberá tener una o más definiciones axiomáticas en este sector (cada

definición axiomática corresponde a un renglón de definición de la operación $idproy_{ij}$ o $idop_{ij}$). Para cada caso de definición axiomática, se establece el formato de sus *n* parámetros de entrada si el tipo del dominio de esa operación corresponde a un producto cartesiano de los *n* componentes, que representa lo que se llama un patrón (pat_i). El formato de los patrones *pat* es el siguiente:

pat	::=	*idcons*	(constante constructora)
	::=	*idcons* **(** pat_1 **,** ... **,** pat_n **)**	(imagen de función constructora)
	::=	*idvar*	(variable)

Cada uno de los patrones *pat*, como las expresiones *exp*, también tiene un tipo de dato asociado. Si el patrón corresponde a una constante constructora, deberá ser del tipo del TAD y además ser una constante constructora válida. Si el patrón corresponde a la imagen de una función constructora, su imagen deberá ser del tipo del TAD, deberá ser una función constructora válida del TAD, y el tipo de cada uno de sus subpatrones deberá ser del tipo de cada uno de sus parámetros correspondientes. Si el patrón no tiene ninguna de esas formas anteriores, se asume que es una variable, que será del tipo de ese parámetro. **Solamente se podrán usar patrones de la forma de constructores (no variables) en el TAD a quien representan esos constructores**.

En el caso de realizar una evaluación de la aplicación de una operación a sus parámetros, el sistema deberá controlar cuál regla (axioma) de esa operación será elegible para ser aplicada, en función de los parámetros respecto a la estructura de sus patrones respectivos en esa regla. Para que sea aplicable esa regla, cada uno de sus parámetros deberá "unificar" con sus patrones respectivos. Si algún parámetro no llegara a unificar con su patrón, la regla será descartada y buscará otra regla posible. Si ninguna regla fuera posible, la especificación es errónea, o no hay definición para esa operación con los parámetros indicados.

Una expresión unifica con el patrón constante constructora si esa expresión evalúa en esa constante constructora. Una expresión unifica con el patrón función constructora con subpatrones-parámetros si esa expresión evalúa en esa función constructora, y sus subexpresiones evaúan con los subpatrones correspondientes. Una expresión siempre unifica con el patrón variable. Junto con este proceso de unificación, en el caso de ser positivo, se van obteniendo sustituciones de las variables de los patrones que aparecen en la regla por sus expresiones correspondientes que intentan unificar con ellas.

Cuando una regla axiomática es elegida (cumpliéndose los procesos de unificación de los patrones de todos los parámetros de la regla con las expresiones-parámetro respectivas), entonces se aplica la regla, evaluando finalmente en la expresión de la parte derecha del símbolo **=**. En esta expresión pueden aparecer como subexpresiones, variables de los patrones o subpatrones de su parte derecha. Si contuviera esas variables como subexpresiones, el valor de ellas corresponde a la aplicación de la sustitución obtenida para esa variable.

Las definiciones axiomáticas de la misma operación no deben "solaparse" en relación a la estructura de los patrones de cada una de ellas. Eso quiere decir que ante cualquier combinación de parámetros de una operación, deberá existir una única regla axiomática

para esa operación cuyos patrones unifiquen con las subexpresiones parámetros-correspondientes. Se puede asumir que en el caso de que exista más de una definición axiomática en condiciones de ser elegida (por estructuras solapadas de sus patrones), la primera de ellas que apareczca en orden será finalmente elegida para evaluar.

Todas las variables que aparecen en cada una de las definiciones axiomáticas, no se consideran libres, sino ligadas cada una a un cuantificador lógico universal *para todo* (∀) que tiene como alcance la regla axiomática completa donde la variable tiene su definición.

Se puede dar la situación que se quieran describir operaciones de un TAD ya existente con anterioridad, que se agregarían al mismo. En ese caso, se utiliza un tipo de sintaxis similar, que corresponde a la siguiente:

extension de TAD *idtipo*[**(***vtipo*$_1$**,** ... **,** *vtipo*$_n$**)**]
exporta *idop*$_1$**,** ... **,** *idop*$_n$
[**usa** *idtipo*$_1$[**(***exptipo*$_{11}$**,** ... **,** *exptipo*$_{1k}$**)**]**,** ... **,** *idtipo*$_n$[**(***exptipo*$_{n1}$**,** ... **,** *exptipo*$_{nr}$**)**]]
operaciones
idop$_1$**:** [*exptipo*$_{11}$ [**×** ... **×** *exptipo*$_{1k}$] **→**] *exptipo*$_1$
...
idop$_t$**:** [*exptipo*$_{t1}$ [**×** ... **×** *exptipo*$_{tw}$] **→**] *exptipo*$_t$
axiomas
idop$_{11}$**(***pat*$_{111}$**,** ... **,** *pat*$_{11k}$**)** **=** *exp*$_{11}$
...
idop$_{1g}$**(***pat*$_{1g1}$**,** ... **,** *pat*$_{1gk}$**)** **=** *exp*$_{1g}$
...
idop$_{t1}$**(***pat*$_{t11}$**,** ... **,** *pat*$_{t1w}$**)** **=** *exp*$_{t1}$
...
idop$_{th}$**(***pat*$_{th1}$**,** ... **,** *pat*$_{thw}$**)** **=** *exp*$_{th}$

La sintaxis es similar a la definición central del TAD, aunque algo más acotada. Aquí se indican a qué TAD extendería (se incorporaría plenamente), la declaración de las operaciones agregadas, sus definiciones axiomáticas, cuáles se ellas se exportan (agregándose a las otras operaciones ya exportadas) y qué TADs adicionales habría que importar para la declaración y definición de estas nuevas operaciones. Los constructores del TAD se mantienen respecto a la definición original.

También existe una expresión especial que en el caso de evaluarse, retorna un error. El formato se esta expresión es la siguiente:

exp ::= **error** [“*texto de error*”]

Las expresiones de error se utilizan en las definiciones axiomáticas para indicar resultados erróneos o indefinidos de proyectores u operaciones a partir de parámetros que se ajustan a los patrones indicados. Se puede decir también que una expresión de error es la

retornada ante la evaluación de una operación en la cual por la estructura de sus subexpresiones parámetros no existe una regla axiomática aplicable para evaluar.
En general, nunca una expresión de tipo error será una subexpresión de otra. Estas expresiones de error pueden ser de cualquier tipo `a`, que se ajustará al que el contexto necesite.

Como apartado en relación al tema anterior, existe una forma de definir axiomáticamente una operación-expresión que sea de tipo `a` (que en su declaración se indique que sea de tipo `a`), donde el tipo `a` es genérico. La forma de definir axiomáticamente la siguente operación `any` (para darle un nombre) será la siguiente:

```
any = any
```

Como en el caso de la expresión de error, esta operación-expresión `any` es imposible de ser evaluada.

Los tipos de datos × (producto cartesiano) y → (función)

El tipo de datos $T_1 \times ... \times T_n$ corresponde al producto cartesiano n-ario de los tipos de datos T_1 a T_n. Hay que considerar que este tipo de dato es "no asociativo" en relación a que no es equivalente al producto cartesiano agrupado en productos cartesianos contenidos de menor aridad.
Por ejemplo, para el producto cartesiano ternario:

$$T_1 \times T_2 \times T_3 \neq (T_1 \times T_2) \times T_3 \neq T_1 \times (T_2 \times T_3)$$

Se puede decir que el producto cartesiano tiene un constructor de su tipo, que representa a la `n`-upla (con cada una de sus componentes entre comas, y encerradas todas entre paréntesis) y que puede ser accedido libremente desde cualquier TAD. Este constructor, también hace de patrón en las definiciones axiomáticas. En realidad se puede decir que si una definición axiomática de una operación recibe un elemento de un producto cartesiano, lo que escribimos entre paréntesis y separados entre comas, no son los patrones de cada uno de sus parámetros, sino en realidad *todo representa el patrón de su único parámetro, que es de tipo producto cartesiano*. Dicho de otro modo, con un ejemplo:

```
f(x, l) = x > length(l)
```

En el ejemplo, no es que la función `f` (de tipo `Nat × Lista(a) → Bool`) reciba dos "parámetros" representados por los patrones `x` y `l` de tipo `Nat` y `Lista(a)` respectivamente, sino que realmente recibe un parámetro representado por el patrón `(x, l)` que es de tipo `Nat × Lista(a)`. En realidad, en las especificaciones axiomáticas, todas las funciones tienen realmente un parámetro, que es de tipo producto cartesiano (si tuviera más de un parámetro-componente) o el tipo de su dominio (si tuviera uno solo). Por convención, si el

dominio no fuera un producto cartesiano (tiene un solo parámetro-componente), se describen igualmente los paréntesis a ambos lados de su parámetro-componente.
En el caso de que sea necesario, se pueden acceder a las operaciones proyectoras del producto cartesiano (que obtienen cada una de sus componentes), que se llaman `#i`, y que son de tipo $T_1 \times ... \times T_i \times ... \times T_n \rightarrow T_i$ (este valor `i` puede ir desde `1` hasta `n`).

En resumen, las expresiones y patrones de tipo producto cartesiano y operaciones relacionadas, son las siguientes (debe valer que $n > 1$):

exp ::= **(***exp*$_1$**,** ... **,** *exp*$_n$**)**
::= **#**`i` **(***exp***)** $(1 \leq i \leq n)$

pat ::= **(***pat*$_1$**,** ... **,** *pat*$_n$**)**

Se puede llegar a permitir que los productos cartesianos puedan conformar un parámetro de los varios que conforman el dominio completo de una función, así como ser retornados por una función. Un ejemplo de definición axiomática donde se ve esto, sería el siguiente:

```
f((x, y), z) = (suma(x, z), not(y))
```

donde en este caso, el tipo de `f` sería $((Nat \times Bool) \times Nat) \rightarrow (Nat \times Bool)$.

Ejemplos análogos a la definición anterior serían los siguientes:

```
f((x, y), z) = (suma(x, z), not(y))

f(w, z) = (suma(#1(w), z), not(#2(w)))

f(n) = (suma(#1(#1(n)), #2(n)), not(#2(#1(n))))
```

El tipo de datos $T_1 \rightarrow T_2$ corresponde al tipo función con dominio en T_1 e imagen en T_2. El tipo función se caracterizan por no poseer constructores (con lo cual el único patrón posible aplicado a ellas es *idvar* -variable-). Tampoco poseen proyectores ni operaciones definidas, más alla de la "función aplicación" de una función a una expresión-parámetro, retornando la expresión-resultado correspondiente.

Si una función fuera capaz de recibir expresiones de tipo función como parte de sus parámetros, o de retornar expresiones de tipo función como resultado, se dice que la función es de *orden superior*, aumentando el nivel de abstracción de las mismas. Este tema está fuera del alcance de este apunte, y de la materia.

Tipos de datos versus TADs

Así como el método de especificación axiomática se aplica para especificar tipos abstractos de datos, también se puede aplicar para especificar tipos de datos no abstractos. La diferencia entre ambos es que los tipos de datos no abstractos (que llamaremos *tipos de datos* a secas), es que mientras los tipos abstratos de datos garantizan encapsulamiento bajo determinadas operaciones indicadas, los tipos de datos no poseen encapsulamiento.

Existen algunas diferencias sintácticas en las especificaciones de ambos tipos de datos, que son las siguientes:

- Los tipos de datos a secas en cabezan la especificación con las siguientes cláusulas (la primera para la especificación inicial del tipo, y la segunda para una extensión de ese tipo), que indican que es un tipo de datos a secas, y no es abstracto:

  Tipo *idtipo*[**(***vtipo*$_1$**,** ... **,** *vtipo*$_n$**)**]
  extension de Tipo *idtipo*[**(***vtipo*$_1$**,** ... **,** *vtipo*$_n$**)**]

- Los tipos de datos a secas, al no garantizar encapsulamiento, exporta absolutamente todas sus operaciones. Esto hace que la siguiente cláusula no exista en su definición (ya que exporta todas sus operaciones).

  exporta *idop*$_1$**,** ... **,** *idop*$_n$

- Así como exportan todas sus operaciones, los tipos de datos a secas también exportan todos sus constructores. Además permiten que sus constructores se puedan utilizar como patrones en otros tipos o TAD en los que el tipo de datos a secas es utilizado.

- La cláusula optativa **usa**, utilizada tanto para tipos de datos a secas como para TADs, permite utilizar tanto tipos de datos de secas como TADs.

El criterio para definir un tipo de datos a secas en lugar de un TAD es cuando se quiere definir un tipo de datos muy simple y utilizado a gran escala, y donde difícilmente se pueda pensar en realizar cambios en su estructura.

En general, los TAD deberán poseer como proyectores u operaciones, operaciones que permitan "construir" elementos de ese TAD, ya que desde el exterior no se pueden invocar a los constructores. Contrariamente, en los tipos de datos a secas, no se necesario definir estas "operaciones constructoras" para diferenciarlos de los constructores, ya que se pueden utilizar directamente los constructores para construir elementos.

Algunos ejemplos

Se presentan aquí algunos ejemplos. El primero sería el del tipo de dato `Bool`, que es no recursivo. Notar que dentro del mismo se define la función `if..then..else...` Como es un tipo muy básico, muy utilizado, y donde los constructores nunca se modificarán, se define como tipo, y no como TAD.

```
Tipo Bool
constructores
   True : Bool
   False : Bool
proyectores
   not : Bool → Bool
   _ and _ : Bool × Bool → Bool
   _ = _ : Bool × Bool → Bool
   if _ then _ else _ : Bool × a × a → a
operaciones
   _ or _ : Bool × Bool → Bool
   _ implies _ : Bool × Bool → Bool
axiomas
   not(True) = False
   not(False) = True
   True and x = x
   False and x = False
   True = x = x
   False = x = not(x)
   if True then x else y = x
   if False then x else y = y
   x or y = not(not(x) and not(y))
   x implies y = not(x) or y
```

Aquí se muestra un TAD recursivo, como `Nat` (números naturales). Como este tipo de datos podría llegar a modificar su estructura interna, se lo define como TAD en lugar de cómo tipo de dato.

```
TAD Nat
exporta zero, suc, isZero, pred, _=_, suma
usa Bool
constructores
   Zero : Nat
   Suc : Nat → Nat
proyectores
   zero : Nat
   suc : Nat → Nat
   isZero : Nat → Bool
   pred : Nat → Nat
operaciones
   _ = _ : Nat × Nat → Bool
   suma : Nat × Nat → Nat
axiomas
   zero = Zero
   suc(n) = Suc(n)
   isZero(Zero) = True
   isZero(Suc(n)) = False
   pred(Zero) = error
   pred(Suc(n)) = n
   x = y = if isZero(x) then isZero(y)
                        else if isZero(y) then False
                                          else pred(x) = pred(y)
   suma(x, y) = if isZero(x) then y
                             else suc(suma(pred(x), y))
```

En la definición axiomática de `pred` se puede ver una definición de caso de error, cuando recibe un `Nat` con valor igual a `Zero`. Esa definición podría no realizarse, y la definición de este operador seguiría siendo correcta.

Las dos últimas operaciones no proyectoras (`_=_` y `suma`), fueron definidas axiomáticamente utilizando operaciones proyectoras y utilizando la función `if`. Definiciones axiomáticas equivalentes (y también válidas) para estas dos últimas operaciones definiéndolas sobre los constructores de sus parámetros sería la siguiente:

```
   Zero = Zero = True
   Zero = Suc(m) = False
   Suc(n) = Zero = False
   Suc(n) = Suc(m) = n = m
   suma(Zero, y) = y
   suma(Suc(n), y) = Suc(suma(n, y))
```

La definición de un tipo de dato recursivo y paramétrico, podría ser el tipo `List`, definido de la siguiente forma:

```
Tipo List(a)
usa Bool
constructores
   EmptyList : List(a)
   ConsList : a × List(a) → List(a)
proyectores
   isEmptyList : List(a) → Bool
   head : List(a) → a
   tail : List(a) → List(a)
operaciones
   length : List(a) → Nat
   belongs : a con operaciones _=_ × List(a) → Bool
   _ = _ : List(a con operaciones _=_) × List(a) → Bool
   concat : List(a) × List(a) → List(a)
   reverse : List(a) → List(a)
   add : List(a) × a → List(a)
axiomas
   isEmptyList(EmptyList) = True
   isEmptyList(ConsList(x, xs)) = False
   head(ConsList(x, xs)) = x
   tail(ConsList(x, xs)) = xs
   length(l) = if isEmptyList(l) then zero else suc(length(tail(l)))
   belongs(x, l) = if isEmptyList(l) then
                      False
                   else
                      (x = head(l)) or belongs(x, tail(l))
   l1 = l2 = if isEmptyList(l1) then
                isEmptyList(l2)
             else
                if isEmptyList(l2) then
                   False
                else
                   (head(l1) = head(l2)) and (tail(l1) = tail(l2))
   concat(l1, l2) = if isEmptyList(l1) then l2
                     else consList(head(l1), concat(tail(l1), l2))
   reverse(l) = if isEmptyList(l) then EmptyList
                else add(reverse(tail(l)), head(l))
   add(l, x) = if isEmptyList(l) then consList(x, emptyList)
               else consList(head(l), add(tail(l), x))
```

Aquí se pueden ver dos operaciones (`belongs` y `_=_`) que requieren que el tipo paramétrico del tipo `List(a)` tenga la operación `_=_` (igualdad). Si un tipo no tuviera esta operación, se puede usar como tipo paramétrico del tipo `List(a)` con sus operaciones sin restricciones sobre el tipo paramétrico, pero no se le podrán aplicar estas dos operaciones.

La operación `belongs` también podría definirse de la siguiente manera:

```
   belongs(x, l) =
      not(isEmptyList(l)) and ((x = head(l)) or belongs(x, tail(l)))
```

Si bien se puede ver que esta definición es equivalente a la anterior, la problemática de esta definición reside en la evaluación de la función `and` del tipo `Bool`, en el caso cuando la lista (`l`) es vacía. Cuando esto ocurre, el primer parámetro de `and` dará como resultado `false`, y se asume que con sólo la evaluación del primer parámetro (`false`), es suficiente para que el resultado de la evaluación de `and` sea `false`, sin importar cuál sea el resultado del segundo parámetro. El problema reside en que en este caso, si el segundo parámetro se evaluara, daría como resultado *`error`*, ya que la lista `l`, al ser vacía, no se le puede aplicar la operación proyectora `head` ni `tail`. Esta situación de evaluar sólo los parámetros que se necesitan al evaluar la aplicación de una función, es lo que se llama *evaluación por necesidad* (un caso particular de lo que se llama *evaluación perezosa*, o *lazy*). Se puede asumir que las definiciones axiomáticas aquí presentadas aceptan el criterio de *evaluación por necesidad*. Utilizando evaluación por necesidad puede hacer posible que una expresión de error (o una expresión-operación del estilo de `any`, ya presentada antes) sea una subexpresión de otra, ya que dependiendo del contexto en que se la utilice, esa expresión de error no sea necesario que sea evaluada, como por ejemplo en la siguiente expresión, que dará como resultado `False` sin necesidad de evaluar la expresión de error:

```
false and (error "situación errónea")
```

Otro caso para destacar en el TAD anterior reside en la definición de la operación `reverse`. Aquí es donde se puede observar el concepto de reusabilidad de operaciones. La definición de esta operación se apoya en la definición de otra operación, que es `add` (agregar un elemento a la lista, por detrás). Esta operación `add` tiene carácter de operación auxiliar (sólo usada por `reverse`), y podría no ser exportada por el tipo de datos si fuera un TAD.

Además de la reusabilidad de operaciones, también existe la reusabilidad de tipos. Este concepto aparece cuando se define un nuevo TAD o tipo a partir de otro TAD o tipo existente (basándose plenamente en él, o utilizándolo en parte), y no sólo a partir de sus constructores propios. Al utilizar un TAD o tipo existente, también utilizará las operaciones exportadas por éste.

En el siguiente ejemplo se observa una nueva definición del TAD `Nat`, esta vez definido a partir del tipo de datos de las listas (tal como se definió aquí).

```
TAD Nat
exporta zero, suc, isZero, pred, _=_, suma
usa Bool, Unit, List(Unit)
constructores
   FromList : List(Unit) → Nat
proyectores
   getList : Nat → List(Unit)
operaciones
   zero : Nat
   suc : Nat → Nat
   isZero : Nat → Bool
   pred : Nat → Nat
   _ = _ : Nat × Nat → Bool
   suma : Nat × Nat → Nat
axiomas
   getList(FromList(l)) = l
   zero = FromList(EmptyList)
   suc(n) = FromList(ConsList(unit, getList(n)))
   isZero(n) = isEmptyList(getList(n))
   pred(n) = FromList(tail(getList(n)))
   x = y = FromList(x) = FromList(y)
   suma(x, y) = FromList(concat(getList(x), getList(y)))
```

Lo que aquí se observa es que aunque cambió la representación interna del TAD `Nat` respecto a la definición anterior, no cambió las operaciones que exporta, y todas ellas mantienen el mismo comportamiento que en el de la definición anterior (se mantiene la *interfaz* del TAD). Este mantenimiento de la interfaz del TAD no obliga a cambiar las especificaciones de todos los TAD que lo usan. Lo curioso de la nueva definición es que los nuevos proyectores (que son diferentes al cambiar el tipo de representación del TAD) no son exportados.

Otra diferencia importante entre las dos representaciones del TAD `Nat` es que en la primera definición, la definición estructural del tipo estaba dada por la naturaleza de los propios constructores, mientras que en la segunda definición, la definición estructural del tipo está apoyada en una particularización de un TAD existente, como es el TAD `List(a)`. Si bien en esta última definición del TAD `Nat` también existen constructores, solamente existe un único constructor función (`FromList`) que hace de "conversor" entre el tipo sobre el cual se define y el TAD, donde la única utilidad del mismo es diferenciar el tipo estructural con el que se construye el TAD con el tipo propio del TAD.
Para efectivizar la última definición, fue necesario definir un tipo especial llamado `Unit` que se caracteriza por poseer un solo elemento, para ser usado como parámetro de tipo del tipo `List(a)`. La definición del tipo `Unit` (que por ser tan simple no es un TAD) es la siguiente:

```
Tipo Unit
usa Bool
constructores
   Unit : Unit
operaciones
   _ = _ : Unit × Unit → Bool
axiomas
   Unit = Unit = True
```

Bibliografía

- *Okasaki, Chris*. **Purely functional data structures**. 1998. Cambridge University Press. [capítulos 2 y 3]

- *Abelson, Harold / Sussman, Gerald Jay / Sussman, Julie*. **Structure and implementation of computer programs**. 1996. MIT Press. [capítulos 1 y 2]

- *Sommerville, Ian*. **Software engineering**. 2007. Addison Wesley. [capítulo 10]

- *O'Sullivan, Bryan / Stewart, Don / Goerzen, John*. **Real World Haskell**. 2008. O'Reilly.