

Prólogo

El estudio de las estructuras de datos es casi tan antiguo como el nacimiento de la programación, y se convirtió en un tema capital en este ámbito desde finales de la década de los 60. Como es lógico, una de las consecuencias de este estudio es la aparición de una serie de libros de gran interés sobre el tema, algunos de ellos ciertamente excelentes y que se han convertido en piedras angulares dentro de la ciencia de la programación (citemos, por ejemplo, los textos de D.E. Knuth; de A.V. Aho, J. Hopcroft y J.D. Ullman; de E. Horowitz y D. Sahni; de N. Wirth; y, recientemente, de T.H. Cormen, C.E. Leiserson i R.L. Rivest).

Ahora bien, el progreso en el campo de la programación ha dado como resultado la aparición de nuevos conceptos, algunos de los cuales no se han consolidado hasta la segunda mitad de la década de los 80. Muchos de estos conceptos están íntimamente interrelacionados con el ámbito de las estructuras de datos, y ésta es la razón por la cual los libros antes citados han quedado actualmente un poco desfasados en lo que respecta al método de desarrollo de programas que siguen, incluso en sus reediciones más recientes.

En este contexto, he confeccionado el libro "Estructuras de datos. Especificación, diseño e implementación", que trata el estudio de las estructuras de datos dentro del marco de los tipos abstractos de datos. La adopción de este enfoque se inscribe en una metodología de desarrollo modular de programas, que abunda en diferentes propiedades interesantes en la producción industrial de aplicaciones (corrección, mantenimiento, etc.), y permite enfatizar diversos aspectos importantes hoy en día: la necesidad de especificar el *software*, la separación entre la especificación y la implementación, la construcción de bibliotecas de componentes, la reusabilidad del *software*, etc. Diversos autores han explorado esta metodología (sobre todo, desde las aportaciones de B. Liskov y J.V. Guttag), pero sin aplicarla en el contexto de las estructuras de datos.

Destinatario

El libro ha sido concebido sobre todo como un texto de ayuda para alumnos de una asignatura típica de estructura de datos en un primer ciclo de ingeniería en informática; también se puede considerar adecuado para cualquier otra titulación técnica superior o media con contenido informático. A tal efecto, cubre el temario habitual de esta asignatura en tono autoexplicativo, y se ilustra con numerosas figuras, especificaciones y programas.

Dependiendo de los objetivos de la asignatura, el formalismo asociado al estudio de estos temas puede ser más o menos acusado; sea como sea, el libro puede usarse como texto básico de consulta.

Ahora bien, los temas que aparecen en el libro se han desarrollado con más profundidad que la estrictamente requerida por el alumno y, por ello, hay más posibles destinatarios. Por un lado, el mismo profesor de la asignatura, porque puede encontrar en un único volumen los aspectos de especificación y de diseño que no acostumbran a aparecer en los libros de estructuras de datos; además, la inclusión de especificaciones y de programas libera al docente de la necesidad de detallarlos en sus clases. Por otro lado, cualquier informático que quiera profundizar en el estudio de las estructuras de datos más allá de su aspecto puramente de programación, puede encontrar aquí una primera referencia.

Contenido

En el primer capítulo se introduce el concepto de tipo abstracto de datos. Después de analizar su repercusión en el diseño de programas, nos centramos en el estudio de su especificación formal, que es la descripción exacta de su comportamiento. De entre las diferentes opciones existentes de especificación formal, se sigue la llamada especificación ecuacional interpretada con semántica inicial. El capítulo muestra un método general para construir especificaciones para los tipos, les otorga un significado matemático (como álgebras heterogéneas) y también estudia su estructuración, y aquí destaca la posibilidad de definir tipos genéricos, profusamente utilizados a lo largo del libro.

En el segundo capítulo se estudian diversos aspectos sobre la implementación de los tipos de datos. El proceso de implementación se lleva a cabo cuando existe una especificación para el tipo; la segunda sección insiste precisamente en la relación formal entre los dos conceptos, especificación e implementación. También se introduce un punto clave en el análisis de los algoritmos y las estructuras de datos que se desarrollarán posteriormente: el estudio de su eficiencia a través de las denominadas notaciones asintóticas.

Las diversas familias de estructuras de datos se introducen en los cuatro capítulos siguientes: se estudian las secuencias, las tablas y los conjuntos, los árboles, y las relaciones binarias y los grafos. Para todas ellas se sigue el mismo método: descripción informal, formulación de un modelo, especificación algebraica del tipo e implementaciones más habituales. Por lo que se refiere a estas últimas, se detalla la representación del tipo y la codificación de las operaciones (hasta el último detalle y buscando la máxima legibilidad posible mediante el uso de funciones auxiliares, diseño descendente, comentarios, etc.), siempre en el caso de implementación en memoria interna; a continuación, se estudia su eficiencia tanto temporal como espacial y se proponen varios ejercicios.

Por último, el capítulo final muestra la integración del concepto de tipo abstracto de datos dentro del desarrollo modular de programas, y lo hace bajo dos vertientes: el uso de los tipos

abstractos previamente introducidos y el diseño de nuevos tipos de datos. El estudio se hace a partir de seis ejemplos escogidos cuidadosamente, que muestran la confrontación de los criterios de modularidad y eficiencia en el diseño de programas.

Para leer el texto, son necesarios unos conocimientos fundamentales en los campos de las matemáticas, de la lógica y de la programación. De las matemáticas, los conceptos básicos de conjunto, producto cartesiano, relación, función y otros similares. De la lógica, el concepto de predicado, los operadores booleanos y las cuantificaciones universal y existencial. De la programación, la habilidad de codificar usando un lenguaje imperativo cualquiera (Pascal, C, Ada o similares) que conlleva el conocimiento de los constructores de tipos de datos (tuplas y vectores), de las estructuras de control de flujo (asignaciones, secuencias, alternativas y bucles) y de los mecanismos de encapsulamiento de código (acciones y funciones).

Es importante destacar algunos puntos que el libro no trata, si bien por su temática se podría haber considerado la posibilidad de incluirlos. Primero, no aparecen algunas estructuras de datos especialmente eficientes que, por su complejidad, superan el nivel de una asignatura de primer ciclo de ingeniería; por ejemplo, diversas variantes de montículos y de árboles de búsqueda (*Fibonacci Heaps*, *Red-Black Trees*, *Splay Trees*, etc.) y de dispersión (*Perfect Hashing*, principalmente). También se excluyen algunas otras estructuras que se aplican principalmente a la memoria secundaria, como pueden ser las diversas variantes de árboles B y también los esquemas de dispersión incremental (*Extendible Hashing*, *Linear Hashing*, etc.). Tampoco se tratan en el libro algunos temas característicos de la programación, como pueden ser el estudio de diversas familias de algoritmos (*Greedy Algorithms*, *Dynamic Programming*, etc.) de los cuales constan algunos casos particulares en el capítulo de grafos; o como las técnicas de derivación y de verificación formal de programas, si bien se usan algunos elementos (invariantes de bucles, precondiciones y postcondiciones de funciones, etc.). Hay diversos libros de gran interés que sí tratan en profundidad estos temas, cuyas referencias aparecen convenientemente en este texto. Por último, no se utilizan los conceptos propios de la programación orientada a objetos (básicamente, herencia y vinculación dinámica) para estructurar los tipos de datos formando jerarquías; se ha preferido el enfoque tradicional para simplificar el volumen de la obra y no vernos obligados a introducir la problemática inherente a este paradigma de la programación.

Bibliografía

Las referencias bibliográficas del libro se pueden dividir en dos grandes apartados. Por un lado se citan todos aquellos artículos que son de utilidad para temas muy concretos, cuya referencia aparece integrada en el texto en el mismo lugar en que se aplican. Por el otro, hay diversos textos de interés general que cubren uno o más capítulos del libro y que aparecen dentro del apartado de bibliografía; estos libros han de considerarse como los más destacables en la confección de esta obra y no excluye que haya otros, igualmente buenos, que no se citan, bien porque su temática es muy similar a alguno de los que sí aparecen, bien porque el desarrollo de los temas es diferente al que se sigue aquí.

Lenguaje

En cualquier texto sobre programación, es fundamental la elección del lenguaje utilizado como vehículo para codificar (y, en este libro, también para especificar) los esquemas que se introducen. En vez de especificar y programar usando algún lenguaje existente, he preferido emplear la notación *Merlí*, diseñada por diversos miembros del Departament de Llenguatges i Sistemes Informàtics (antiguamente, Departament de Programació) de la Universitat Politècnica de Catalunya. Esta notación ha sido utilizada desde principios de los años 80 por los profesores del departamento en la impartición de las asignaturas de programación de los primeros niveles de las titulaciones en informática y ha demostrado su validez como herramienta para el aprendizaje de la programación. Las razones de esta elección son básicamente dos: por un lado, disponer de una notación abstracta que permita expresar fácilmente los diferentes esquemas que se introducen sin ningún tipo de restricción impuesta por el lenguaje; por otro, usar una sintaxis muy parecida tanto para especificar como para implementar los tipos de datos (el hecho de que el mismo lenguaje se pueda usar desde estos dos niveles diferentes refuerza la relación entre la especificación y la implementación de los tipos de datos, que es uno de los objetivos del texto). El inconveniente principal es la necesidad de traducir las especificaciones y los programas que aparecen en este texto a los lenguajes que el lector tenga a su disposición; ahora bien, este inconveniente no parece muy importante, dado que *Merlí* es fácilmente traducible a cualquier lenguaje comercial (a algunos mejor que a otros, eso sí), y que podría haber aparecido el mismo problema fuera cual fuera el lenguaje de trabajo elegido.

Terminología

Dado que, hoy en día, el idioma dominante en el ámbito de la informática es el inglés, he hecho constar las acepciones inglesas junto a aquellos vocablos que denotan conceptos básicos y universalmente aceptados; de esta manera, el lector puede relacionar rápidamente estos conceptos dentro de su conocimiento de la materia o, en el caso de que sea el primer libro que lee sobre estructuras de datos, adquirir el vocabulario básico para la lectura posterior de textos ingleses. Los términos ingleses se escriben siempre en singular independientemente del género con el que se usen en castellano.

Por el mismo motivo, se utilizan de manera consciente varios anglicismos usuales en el ámbito de la programación para traducir algunos términos ingleses. Dichos anglicismos se limitan a lo estrictamente imprescindible, pero he creído conveniente seguir la terminología técnica habitual en vez de introducir vocablos más correctos desde el punto de vista lingüístico pero no tan profusamente usados. Así, aparecen los términos "reusabilidad" en vez de "reutilización", "eficiencia" en vez de "eficacia", etc.

Agradecimientos

Este libro es el resultado de una experiencia personal de varios años de docencia en las asignaturas de estructuras de datos en los planes de licenciatura e ingeniería de la Facultat d'Informàtica de Barcelona de la Universitat Politècnica de Catalunya, por lo que refleja un

gran número de comentarios y aportaciones de todos los profesores que, a lo largo de este período, han sido compañeros de asignatura. Quizás el ejemplo más paradigmático sea la colección de ejercicios propuestos en el texto, muchos de ellos provenientes de las listas de ejercicios y exámenes de las asignaturas citadas. Para ellos mi más sincero agradecimiento. En particular, quiero citar al profesor Ricardo Peña por su ayuda durante el primer año que impartí la asignatura "Estructuras de la Información"; a los profesores y profesoras M.T. Abad, J.L. Balcázar, J. Larrosa, C. Martínez, P. Meseguer, T. Moreno, P. Nivelá, R. Nieuwenhuis y F. Orejas por la revisión de secciones, versiones preliminares y capítulos enteros del texto y por la detección de errores; y, sobre todo, al profesor Xavier Burgués por todos los años de continuos intercambios de opinión, sugerencias y críticas. A todos ellos, gracias.

Contacto

El lector interesado puede contactar con el autor en la dirección electrónica *franch@lsi.upc.es*, o bien dirigiéndose al departamento de Llenguatges i Sistemes Informàtics de la Universitat Politècnica de Catalunya. En especial, el autor agradecerá la notificación de cualquier errata detectada en el texto, así como toda sugerencia o crítica a la obra.

Barcelona, 10 de Junio de 1996

Capítulo 1 Especificación de tipos abstractos de datos

El concepto de tipo abstracto de datos será el marco de estudio de las estructuras de datos que se presentan en el libro. Por ello, dedicamos el primer capítulo a estudiar su significado a partir de lo que se denomina una especificación algebraica, que es la descripción precisa de su comportamiento. También se introducen en profundidad los mecanismos que ofrece la notación Merlí para escribirlas y que serán usados a lo largo del texto en la descripción preliminar de las diferentes estructuras de datos que en él aparecen.

1.1 Introducción a los tipos abstractos de datos

Con la aparición de los lenguajes de programación estructurados en la década de los 60, surge el concepto de tipo de datos (ing., *data type*), definido como un conjunto de valores que sirve de dominio de ciertas operaciones. En estos lenguajes (C, Pascal y similares, derivados todos ellos -de forma más o menos directa- de Algol), los tipos de datos sirven sobre todo para clasificar los objetos de los programas (variables, parámetros y constantes) y determinar qué valores pueden tomar y qué operaciones se les pueden aplicar.

Esta noción, no obstante, se reveló insuficiente en el desarrollo de software a gran escala, dado que el uso de los datos dentro de los programas no conocía más restricciones que las impuestas por el compilador, lo cual era muy inconveniente en los nuevos tipos de datos definidos por el usuario, sobre todo porque no se restringía de ninguna manera su ámbito de manipulación. Para solucionar esta carencia, resumida por J.B. Morris en "Types are not Sets" (Proceedings ACM POPL, 1973), diversos investigadores (citamos como pioneros a S.N. Zilles, a J.V. Guttag y al grupo ADJ, formado por J.A. Goguen, J.W. Thatcher, E.G. Wagner y J.B. Wright [ADJ78]) introdujeron a mediados de la década de los 70 el concepto de tipo abstracto de datos (ing., *abstract data type*; abreviadamente, TAD), que considera un tipo de datos no sólo como el conjunto de valores que lo caracteriza sino también como las operaciones que sobre él se pueden aplicar, juntamente con las diversas propiedades que determinan inequívocamente su comportamiento. Todos estos autores coincidieron en la necesidad de emplear una notación formal para describir el comportamiento de las operaciones, no sólo para impedir cualquier interpretación ambigua sino para identificar claramente el modelo matemático denotado por el TAD.

En realidad, el concepto de TAD ya existe en los lenguajes de programación estructurados bajo la forma de los tipos predefinidos, que se pueden considerar como tipos abstractos con poco esfuerzo adicional. Por ejemplo, consideremos el tipo de datos de los enteros que ofrece el lenguaje Pascal; la definición del TAD correspondiente consiste en determinar:

- cuáles son sus valores: los números enteros dentro del intervalo [minint, maxint];
- cuáles son sus operaciones: la suma, la resta, el producto, y el cociente y el resto de la división, y
- cuáles son las propiedades que cumplen estas operaciones: hay muchas; por ejemplo: $a+b = b+a$, $a*0 = 0$, etc.

Resumiendo, se puede definir un tipo abstracto de datos como un conjunto de valores sobre los que se aplica un conjunto dado de operaciones que cumplen determinadas propiedades. ¿Por qué "abstracto"? Éste es un punto clave en la metodología que se presentará y se aplicará en todo el libro. El calificativo "abstracto" no significa "surrealista" sino que proviene de "abstracción", y responde al hecho de que los valores de un tipo pueden ser manipulados mediante sus operaciones si se saben las propiedades que éstas cumplen, sin que sea necesario ningún conocimiento ulterior sobre el tipo; en concreto, su implementación en la máquina es absolutamente irrelevante. En el caso de los enteros de Pascal, cualquier programa escrito en este lenguaje puede efectuar la operación $x+y$ (siendo x e y dos variables enteras) con la certeza de que siempre calculará la suma de los enteros x e y , independientemente de su representación interna en la máquina que está ejecutando el programa (complemento a 2, signo y magnitud, etc.) porque, sea ésta cual sea, la definición de los enteros de Pascal asegura que la suma se comporta de una manera determinada. En otras palabras, la manipulación de los objetos de un tipo sólo depende del comportamiento descrito en su especificación (ing., specification) y es independiente de su implementación (ing., implementation):

- La especificación de un TAD consiste en establecer las propiedades que lo definen. Para que sea útil, una especificación ha de ser precisa (sólo tiene que decir aquello realmente imprescindible), general (adaptable a diferentes contextos), legible (que sirva como instrumento de comunicación entre el especificador y los usuarios del tipo, por un lado, y entre el especificador y el implementador, por el otro) y no ambigua (que evite posteriores problemas de interpretación). La especificación del tipo, que es única, define totalmente su comportamiento a cualquier usuario que lo necesite. Según su grado de formalismo, será más o menos fácil de escribir y de leer y más o menos propensa a ser ambigua o incompleta.
- La implementación de un TAD consiste en determinar una representación para los valores del tipo y en codificar sus operaciones a partir de esta representación, todo ello usando un lenguaje de programación convencional. Para que sea útil, una implementación ha de ser estructurada (para facilitar su desarrollo), eficiente (para optimizar el uso de recursos del computador) y legible (para facilitar su modificación y

mantenimiento). Una implementación del TAD (puede haber muchas, cada una de ellas pensada para un contexto de uso diferente) es totalmente transparente a los usuarios del tipo y no se puede escribir hasta haber determinado claramente su especificación; el cambio de una implementación por otra que respete el comportamiento deseado del tipo no ha de cambiar en absoluto la especificación ni, por consiguiente, la visión que de él tienen sus usuarios, que se limitarán a recompilar la aplicación correspondiente.

La verdadera utilidad de los TAD aparece en el diseño de nuevos tipos de datos. Imaginemos que se quiere construir un programa Pascal que calcule la suma de una secuencia de números complejos introducida por el terminal, acabada por el valor $0 + 0i$ (para simplificar la escritura de algunos detalles irrelevantes, supondremos que tanto la parte real como la parte imaginaria de los números complejos son enteros en vez de reales), escribiendo el resultado en la pantalla. Fieles a la metodología que acabamos de esbozar, enfocamos el caso como un ejercicio resoluble a partir de la especificación de un TAD para los números complejos, definible de la siguiente forma:

- cuáles son sus valores: todos aquellos números complejos de partes real e imaginaria enteras y dentro del intervalo $[\text{minint}, \text{maxint}]$;
- cuáles son sus operaciones: como mínimo, y dada la funcionalidad del programa, se necesita una operación para sumar complejos, otra para crear un complejo a partir de dos enteros y dos más para obtener las partes real e imaginaria, y
- cuáles son las propiedades que cumplen las operaciones: las típicas de los complejos.

Una vez definido el TAD para los complejos es posible utilizarlo desde un programa Pascal: se pueden declarar variables del tipo, usar objetos del tipo como parámetros de funciones, utilizar el tipo para construir otros más complicados, etc.; dicho de otra forma, el (nuevo) TAD de los complejos tiene las mismas características de uso que el TAD (predefinido) de los enteros y desde un programa Pascal sus diferencias son exclusivamente notacionales. Como consecuencia, se puede escribir el programa principal que calcula la suma de los complejos sin implementar el TAD.

```
program suma_complejos;  
var res: complejo; a, b: integer;  
begin  
  res := crear(0, 0); read(a, b);  
  while (a <> 0) or (b <> 0) do begin  
    res := sumar(res, crear(a, b)); read(a, b)  
  end;  
  writeln('El resultado es: ', real(res), ' + ', imaginaria(res), 'i.')  
end.
```

Fig. 1.1: programa Pascal para sumar una secuencia de números complejos.

Es decir, el programa resultante es independiente de la implementación del tipo de los complejos en Pascal. Una vez determinadas las operaciones, para completar la aplicación se escoge una representación para el TAD y se implementa (en la fig. 1.2 se da una representación que mantiene los complejos en notación binómica).

```

type complejo = record re, im: integer end;

function crear (a, b: integer): complejo;
var c: complejo;
begin
    c.re := a; c.im := b; crear := c
end;

function sumar (a, b: complejo): complejo;
begin
    a.re := a.re + b.re; a.im := a.im + b.im; sumar := a
end;

function real (c: complejo): integer;
begin
    real := c.re
end;

function imaginaria (c: complejo): integer;
begin
    imaginaria := c.im
end;

```

Fig. 1.2: codificación en Pascal de una representación binómica para los números complejos.

La extrapolación de esta técnica a sistemas de gran tamaño conduce al llamado diseño modular de las aplicaciones (ing., modular design, formulado por D.L. Parnas en 1972 en el artículo "On the Criteria to be Used in Decomposing Systems into Modules", CACM, 15(12); también lo llamaremos diseño con TAD; v. [LiG86] para un estudio en profundidad). El diseño modular es una generalización del diseño descendente de programas (ing., stepwise design; llamado también diseño por refinamientos sucesivos) introducido a finales de los años 60 por diversos autores, entre los que destacan O.-J. Dahl, E.W. Dijkstra y C.A.R. Hoare (v. Structured Programming, Academic Pres Inc., 1972), y se caracteriza por el hecho de dividir el problema original en varios subproblemas más pequeños, cada uno de ellos con una misión bien determinada dentro del marco general del proyecto, que interaccionan de manera clara y mínima y de tal forma que todos ellos juntos solucionan el problema inicial; si algunos subproblemas siguen siendo demasiado complicados, se les aplica el mismo proceso, y así sucesivamente hasta llegar al estado en que todos los subproblemas son lo

bastante sencillos como para detener el proceso. El resultado es una estructura jerárquica que refleja las descomposiciones efectuadas; cada descomposición es el resultado de abstraer las características más relevantes del problema que se está tratando de los detalles irrelevantes en el nivel de razonamiento actual, los cuales adquieren importancia en descomposiciones sucesivas. Desde el punto de vista de su gestión, cada subproblema es un TAD que se encapsula en lo que se denomina un módulo¹ (ing., module); precisamente, la mejora respecto al diseño descendente proviene de la ocultación de la representación de los datos y de la limitación de su manipulación al ámbito del módulo que define el tipo.

A primera vista, puede parecer costoso, e incluso absurdo, dividir una aplicación en módulos y escribir procedimientos y funciones para controlar el acceso a la estructura que implementa un TAD; es decir, ¿por qué no escribir directamente la fórmula de la suma de complejos allí donde se necesite, en vez de encapsular el código dentro de una función? La respuesta es que esta metodología abunda en diversas propiedades interesantes:

- Abstracción. Los usuarios de un TAD no necesitan conocer detalles de implementación (tanto en lo que se refiere a la representación del tipo como a los algoritmos y a las técnicas de codificación de las operaciones), por lo que pueden trabajar en un grado muy alto de abstracción. Como resultado, la complejidad de un programa queda diluida entre sus diversos componentes.
- Corrección. Un TAD puede servir como unidad indivisible en las pruebas de programas, de manera que en una aplicación que conste de diversos tipos abstractos no tengan que probarse todos a la vez, sino que es factible y recomendable probarlos por separado e integrarlos más adelante. Evidentemente, es mucho más fácil detectar los errores de esta segunda manera, porque las entidades a probar son más pequeñas y las pruebas pueden ser más exhaustivas. Por otro lado, la adopción de una técnica formal de especificación como la que se explica en el resto del capítulo posibilita la verificación formal de la aplicación de manera que, eventualmente, se puede demostrar la corrección de un programa; ésta es una mejora considerable, porque la prueba empírica muestra la ausencia de errores en determinados contextos, pero no asegura la corrección absoluta. Hay que decir, no obstante, que la complejidad intrínseca de los métodos formales, junto con el volumen de los TAD que aparecen en aplicaciones reales, y también la inexistencia de herramientas totalmente automáticas de verificación, dificultan (y, hoy en día, casi imposibilitan) la verificación formal completa.
- Eficiencia. La separación clara entre un programa y los TAD que usa favorece la eficiencia, ya que la implementación de un tipo se retrasa hasta conocer las restricciones de eficiencia sobre sus operaciones, y así se pueden elegir los algoritmos óptimos².

¹ En realidad, el diseño modular identifica no sólo TAD (encapsulados en los llamados módulos de datos) sino también los llamados módulos funcionales, que se pueden catalogar como algoritmos no triviales que operan sobre diversos TAD (v. [LiG86] para más detalles).

² Aunque, como veremos a lo largo del texto, la inaccesibilidad de la implementación fuera de los módulos de definición de los TAD comporta a menudo problemas de eficiencia.

- Legibilidad. Por un lado, la estructuración de la información en varios TAD permite estudiar los programas como un conjunto de subprogramas con significado propio y de más fácil comprensión, porque la especificación de un TAD es suficiente para entender su significado. Por lo que respecta a la implementación, los programas que usan los diferentes TAD resultan más fáciles de entender, dado que no manipulan directamente estructuras de datos sino que llaman a las operaciones definidas para el tipo, que ya se encargarán de gestionar las estructuras subyacentes de manera transparente³.
- Modificabilidad y mantenimiento. Cualquier modificación que se tenga que efectuar sobre un programa provocada por cambios en sus requerimientos, por ampliaciones si se desarrollan versiones sucesivas (prototipos), o por su funcionamiento incorrecto no requiere normalmente examinar y modificar el programa entero sino sólo algunas partes. La identificación de estas partes queda facilitada por la estructuración lógica en TAD. Una vez más, es importante destacar que los cambios en la codificación de un TAD no afectan a la implementación de los módulos que lo usan, siempre que el comportamiento externo de las operaciones no cambie.
- Organización. La visión de una aplicación como un conjunto de TAD con significado propio permite una óptima repartición de tareas entre los diferentes componentes de un equipo de trabajo, que pueden desarrollar cada TAD independientemente y comunicarse sólo en los puntos en que necesiten interaccionar; esta comunicación, además, es sencilla, ya que consiste simplemente en saber qué operaciones ofrecen los TAD y qué propiedades cumplen (es decir, en conocer su especificación).
- Reusabilidad. Los TAD diseñados en una especificación pueden ser reutilizables a veces en otros contextos con pocos cambios (lo ideal es que no haya ninguno). En este sentido es importante, por un lado, disponer de un soporte para acceder rápidamente a los TAD (mediante bibliotecas de módulos reusables) y, por otro, escoger las operaciones adecuadas para el tipo en el momento de su definición, incluso añadiendo algunas operaciones sin utilidad inmediata, pero que puedan ser usadas en otros contextos futuros.
- Seguridad. La imposibilidad de manipular directamente la representación evita el mal uso de los objetos del tipo y, en particular, la generación de valores incorrectos. Idealmente los lenguajes de programación deberían reforzar esta prohibición limitando el ámbito de manipulación de la representación. A pesar de que algunos de ellos lo hacen (Ada, Modula-2 y la familia de lenguajes orientados a objetos, incluyendo C++, Eiffel y algunas versiones no estándares de Pascal), hay muchos que no, y es necesario un sobreesfuerzo y autodisciplina por parte del programador para adaptar los conceptos del diseño con TAD a las carencias del lenguaje de programación.

³ Evidentemente, sin olvidar la adopción de técnicas clásicas para la legibilidad, como por ejemplo el uso de diseño descendente, la inserción de comentarios y la aserción de predicados que especifiquen la misión de las funciones, bucles, etc.

Una vez vistos los conceptos de tipo de datos y tipo abstracto de datos, queda claro que el primero de ellos es una limitación respecto al segundo y por ello lo rechazamos; en el resto del texto, cualquier referencia al término "tipo de datos" se ha de interpretar como una abreviatura de "tipo abstracto de datos". Por otro lado, notemos que todavía no ha sido definida la noción de estructura de datos que da título al libro. A pesar de que no se puede decir que haya una definición estándar, de ahora en adelante consideraremos que una estructura de datos (ing., *data structure*) es la representación de un tipo abstracto de datos que combina los constructores de tipo y los tipos predefinidos habituales en los lenguajes de programación imperativos (por lo que respecta a los primeros, tuplas, vectores y punteros principalmente; en lo referente a los segundos, booleanos, enteros, reales y caracteres normalmente). Determinadas combinaciones presentan propiedades que las hacen interesantes para implementar ciertos TAD, y dan lugar a unas familias ya clásicas en el campo de la programación: estructuras lineales para implementar secuencias, tablas de dispersión para implementar funciones y conjuntos, árboles para implementar jerarquías y multilistas para implementar relaciones binarias. La distinción entre el modelo de un TAD y su implementación mediante una estructura de datos es fundamental, y se refleja a lo largo del texto en la especificación del TAD previa al estudio de la implementación.

1.2 Modelo de un tipo abstracto de datos

Para describir el comportamiento de un TAD, es obvio que el lenguaje natural no es una buena opción dada su falta de precisión. Se requiere, pues, una notación formal que permita expresar sin ningún atisbo de ambigüedad las propiedades que cumplen las operaciones de un tipo. Desde que apareció esta necesidad se han desarrollado diversos estilos de especificación formal, cada uno de ellos con sus peculiaridades propias, que determinan su contexto de uso. A lo largo del texto seguiremos la llamada especificación algebraica (también denominada ecuacional), que establece las propiedades del TAD mediante ecuaciones con variables cuantificadas universalmente, de manera que las propiedades dadas se cumplen para cualquier valor que tomen las variables.

El estudio del significado de estas especificaciones se hará dentro del ámbito de unos objetos matemáticos llamados álgebras (ing., *algebra*). El grupo ADJ fue el pionero y máximo impulsor en la búsqueda del modelo matemático de un tipo abstracto (ya desde [ADJ78], que recoge los resultados obtenidos en la primera mitad de la década de los 70, la mayoría de ellos publicados como reports de investigación de los laboratorios IBM Watson Research Center), y son incontables las aportaciones posteriores de otros autores. En el año 1985 se publicó el texto [EhM85], que constituye una compilación de todos los conceptos formales sobre el tema, y al que se ha de considerar como la referencia principal de esta sección; en él se formulan una serie de teoremas y propiedades de gran interés, que aquí se introducen sin demostrar. Posteriormente, los mismos autores presentaron [EhM90], que estudia el modelo formal de los TAD respetando su estructura interna, aspecto éste no tratado aquí.

1.2.1 Signaturas y términos

El primer paso al especificar un TAD consiste en identificar claramente sus objetos y operaciones. En Merlí se encapsula la especificación dentro de una estructura llamada universo (equivalente al concepto tradicional de módulo) donde, para empezar, se escribe el nombre del TAD en definición tras la palabra clave "tipo" y se establecen las operaciones detrás de la palabra clave "ops"; para cada operación se indica su nombre, el número y el tipo de sus parámetros y el tipo de su resultado; es lo que se llama la signatura (ing., signature) de la operación. Al universo se le da un nombre que se usará para referirse a él.

En la fig. 1.3 se muestra un universo para el TAD de los booleanos. Por defecto, las operaciones de una signatura se invocarán con los parámetros (si los hay) separados por comas y encerrados entre paréntesis; para indicar una sintaxis diferente, se usa el carácter de subrayado para determinar la ubicación de todos los parámetros (por ejemplo $\neg_$, $_ \vee _$ y $_ \wedge _$). Por otro lado, diversos operadores con la misma signatura se pueden introducir en la misma línea (como cierto y falso, o $_ \vee _$ y $_ \wedge _$).

```

universo BOOL es
  tipo bool
  ops
    cierto, falso:  $\rightarrow$  bool
     $\neg\_$ : bool  $\rightarrow$  bool
     $\_ \vee \_$ ,  $\_ \wedge \_$ : bool bool  $\rightarrow$  bool
  funiverso

```

Fig. 1.3: signatura de un TAD para los booleanos.

En la fig. 1.4 se ofrece un TAD para los naturales; es necesario, no obstante, introducir también el tipo de los booleanos porque, aunque el objetivo principal es especificar el tipo nat, hay un símbolo de operación, ig , que involucra bool ⁴; las operaciones sobre bool son las estrictamente imprescindibles para especificar más adelante los naturales. La signatura se puede presentar gráficamente encerrando en círculos los nombres de los tipos y representando las operaciones como flechas que salen de los tipos de los parámetros y van a parar al tipo del resultado (v. fig. 1.5).

Resumiendo, en un universo se establece, para empezar, qué objetos y qué operaciones intervienen en la definición del TAD; es lo que se conoce como signatura de un tipo abstracto de datos⁵. Al escribir la signatura, todavía no se da ninguna propiedad sobre los símbolos de operación; además, tampoco se les proporciona ningún significado, aparte de la información puramente subjetiva de sus nombres (que son totalmente arbitrarios).

⁴ En el apartado 1.3.1 estudiaremos cómo aprovechar especificaciones ya construidas.

⁵ Es decir, la palabra "signatura" puede usarse refiriéndose a operaciones individuales o a todo un TAD.

universo NAT es
tipo nat, bool
ops
 cero: $\rightarrow \text{nat}$
 suc: $\text{nat} \rightarrow \text{nat}$
 suma: $\text{nat nat} \rightarrow \text{nat}$
 ig: $\text{nat nat} \rightarrow \text{bool}$
 cierto, falso: $\rightarrow \text{bool}$
funiverso

Fig. 1.4: signatura de un TAD para los naturales.

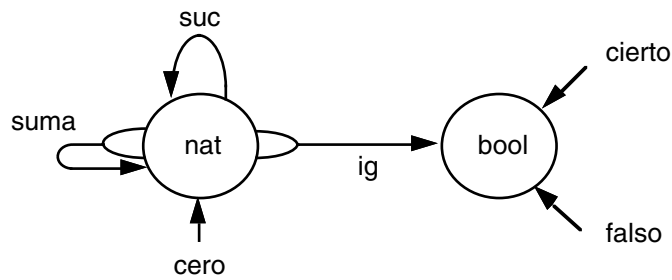


Fig. 1.5: representación pictórica de la signatura de la fig. 1.4.

A continuación, se quiere formalizar el concepto de signatura como primer paso hacia la búsqueda del modelo asociado a una especificación. Previamente hay que introducir una definición auxiliar: dado un conjunto S , definimos un S -conjunto A como una familia de conjuntos indexada por los elementos de S , $A = (A_s)_{s \in S}$; el calificativo "indexada" significa que cada uno de los elementos de S sirve como medio de referencia para un conjunto de A . Definimos el S -conjunto vacío \emptyset como aquel S -conjunto que cumple $\forall s: s \in S: \emptyset_s = \emptyset$. Sobre los S -conjuntos se definen operaciones de intersección, unión, pertenencia e inclusión, cuya definición es, para dos S -conjuntos $A = (A_s)_{s \in S}$ y $B = (B_s)_{s \in S}$:

- $A \cup B = (A_s \cup B_s)_{s \in S}$, $A \cap B = (A_s \cap B_s)_{s \in S}$
- $A \subseteq B \equiv \forall s: s \in S: A_s \subseteq B_s$
- $v \in A \equiv \exists s: s \in S: v \in A_s$

Ahora, se puede definir una signatura SIG como un par $SIG = (S_{SIG}, OP_{SIG})$ o, para reducir subíndices, $SIG = (S, OP)$, donde:

- S es un conjunto de géneros (ing., sort); cada género representa el conjunto de valores que caracteriza el tipo.

- OP es un conjunto de símbolos de operaciones (ing., operation symbol) o, más exactamente, una familia de conjuntos indexada por la signatura de las operaciones, es decir, un $\langle S^* \times S \rangle$ -conjunto⁶, $OP = (OP_{w \rightarrow s})_{w \in S^*, s \in S}$; cada uno de estos conjuntos agrupa los símbolos de operaciones que tienen la misma signatura (exceptuando el nombre). La longitud de w , denotada como $\|w\|$, recibe el nombre de aridad (ing., arity) de la operación; los símbolos de aridad 0 se llaman símbolos de constantes (ing., constant symbol); su tratamiento no difiere del que se da al resto de símbolos (a pesar de que, a veces, habrá que distinguirlos al formular ciertas definiciones recursivas).

Por ejemplo, la signatura $NAT = (S, OP)$ de la fig. 1.4 queda:

$$\begin{aligned} S &= \{\text{nat}, \text{bool}\} \\ OP_{\rightarrow \text{nat}} &= \{\text{cero}\} & OP_{\text{nat} \rightarrow \text{nat}} &= \{\text{suc}\} & OP_{\text{nat nat} \rightarrow \text{nat}} &= \{\text{suma}\} \\ OP_{\rightarrow \text{bool}} &= \{\text{cierto}, \text{falso}\} & OP_{\text{nat nat} \rightarrow \text{bool}} &= \{\text{ig}\} \end{aligned}$$

y el resto de conjuntos $OP_{w \rightarrow s}$ son vacíos.

Mediante la aplicación sucesiva y correcta de símbolos de operaciones de una signatura se pueden construir términos (ing., term) sobre ella; por ejemplo, $\text{suc}(\text{suc}(\text{suc}(\text{cero})))$ es un término sobre la signatura NAT . El conjunto de términos (generalmente infinito) que se puede construir sobre una signatura $SIG = (S, OP)$ es un S -conjunto denotado con T_{SIG} , $T_{SIG} = (T_{SIG,s})_{s \in S}$, definido recursivamente como:

- $\forall c: c \in OP_{\rightarrow s}: c \in T_{SIG,s}$
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow s} \wedge n > 0: \forall t_1: t_1 \in T_{SIG,s_1} : \dots \forall t_n: t_n \in T_{SIG,s_n} : op(t_1, \dots, t_n) \in T_{SIG,s}$
- No hay nada más en T_{SIG}

Es decir, el conjunto de términos contiene todas las combinaciones posibles de aplicaciones de operaciones, respetando aridades y tipos. Los términos que pertenecen a $T_{SIG,s}$ se llaman términos sobre SIG de género s . Por ejemplo, unos cuantos términos correctos para el conjunto de términos $T_{NAT} = (T_{NAT,\text{nat}}, T_{NAT,\text{bool}})$ son cero , $\text{suc}(\text{cero})$ y también $\text{suma}(\text{suc}(\text{cero}), \text{suma}(\text{cero}, \text{cero}))$, que están dentro de $T_{NAT,\text{nat}}$ y falso e $\text{ig}(\text{suc}(\text{cero}), \text{cero})$, que están dentro de $T_{NAT,\text{bool}}$; en cambio, no son términos las expresiones $\text{suc}(\text{cierto})$ y $\text{suc}(\text{cero}, \text{cero})$, porque violan las reglas de formación dadas. La estructura de los términos queda muy clara con una representación gráfica como la fig. 1.6.

La definición de término no considera la existencia de operaciones de invocación no funcional, como \neg o \vee . Este hecho es irrelevante al ser la diferencia puramente sintáctica; si se quieren incorporar, es necesario incluir un mecanismo de parentización en la definición.

⁶ S^* representa el monoide libre sobre S , es decir, secuencias de elementos de S (v. sección 1.5 para una introducción y el capítulo 3 para más detalles). Por su parte, $\langle S \times T \rangle$ representa el producto cartesiano de los conjuntos S y T .

$$\begin{aligned}
NAT &= (S_{NAT}, OP_{NAT}), \\
S_{NAT} &= \{\text{nat}_{NAT}, \text{bool}_{NAT}\}, \text{ donde } \text{nat}_{NAT} \equiv N, \text{ bool}_{NAT} \equiv B^7 \\
OP_{NAT} &= \{\text{cero}_{NAT}, \text{suc}_{NAT}, \text{suma}_{NAT}, \text{ig}_{NAT}, \text{cierto}_{NAT}, \text{falso}_{NAT}\}, \text{ donde} \\
&\quad \text{cero}_{NAT} \equiv 0, \text{ suc}_{NAT} \equiv +1, \text{ ig}_{NAT} \equiv =, \text{ suma}_{NAT} \equiv +, \text{ cierto}_{NAT} \equiv C, \text{ falso}_{NAT} \equiv F
\end{aligned}$$

Esta estructura se llama álgebra respecto de la signatura SIG o, para abreviar, SIG-álgebra (ing., SIG-algebra), y se caracteriza porque da una interpretación de cada uno de los símbolos de la signatura; en el ejemplo anterior, NAT es una NAT-álgebra y el subíndice " NAT " se puede leer como "interpretación dentro del modelo NAT ".

Más formalmente, dada una signatura $SIG = (S, OP)$, una SIG-álgebra A es un par ordenado, $A = (S_A, OP_A)$, siendo S_A un S-conjunto y OP_A un $\langle S^* \times S \rangle$ -conjunto, definida como:

- $\forall s: s \in S: s_A \in S_A$; s_A son los conjuntos de base (ing., carrier set) de A .
- $\forall c: c \in OP_{\rightarrow S}: c_A: \rightarrow s_A \in OP_A$; c_A son las constantes de A .
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow S} \wedge n > 0: op_A: s_{1A} \times \dots \times s_{nA} \rightarrow s_A \in OP_A$; op_A son las operaciones de A .

Así pues, un álgebra es un conjunto de valores sobre el que se aplican operaciones; esta definición es idéntica a la de TAD y por este motivo normalmente se estudian los TAD dentro del marco de las álgebras heterogéneas, es decir, álgebras que implican varios géneros.

Otras veces, las asociaciones a efectuar con los símbolos de la signatura no son tan obvias. Por ejemplo, consideremos la signatura de la fig. 1.7; supongamos que los símbolos nat, bool, cero, suc, ig, cierto y falso tienen el significado esperado dado su nombre, y planteémonos qué significa el género x y qué significan las operaciones crea, convierte, fusiona y está?. Una primera solución consiste en pensar que x representa los conjuntos de naturales y que las operaciones significan:

- crea $\equiv \emptyset$, el conjunto vacío.
- convierte(m) $\equiv \{m\}$, el conjunto que sólo contiene el elemento m .
- fusiona $\equiv \cup$, la unión de conjuntos.
- está? $\equiv \in$, la operación de pertenencia de un elemento a un conjunto.

Todo encaja. Ahora bien, es igualmente lícito conjeturar que x es el monoide libre sobre los naturales (secuencias de naturales, v. sección 1.5) y que las operaciones significan:

- crea $\equiv \lambda$, la secuencia vacía
- convierte(m) $\equiv m$, la secuencia que sólo contiene el elemento m .
- fusiona(r, s) $\equiv r.s$, la concatenación de dos secuencias.
- está? $\equiv \in$, la operación de pertenencia de un elemento a una secuencia.

⁷ A lo largo de este texto, el símbolo \equiv se usa con el significado "se define como".

```

universo X es
  tipo x, nat, bool
  ops
    crea: → x
    convierte: nat → x
    fusiona: x x → x
    está?: nat x → bool
    cero, suc, ig, cierto y falso: v. fig. 1.4
funiverso

```

Fig. 1.7: una signatura misteriosa.

Incluso se puede interpretar alguna operación de manera atípica respecto a su nombre; en el primer modelo dado es válido asociar la diferencia entre conjuntos a fusiona, porque también se respeta la signatura de la operación.

Con la información disponible hasta el momento no se puede decidir cuál es el modelo asociado a la signatura, pues aún no se han especificado las propiedades de sus símbolos; es decir, existen infinitas SIG-álgebras asociadas a una signatura SIG, todas ellas igualmente válidas, que pueden variar en los conjuntos de base o, a conjuntos de base idénticos, en la interpretación de los símbolos de las operaciones; el conjunto de todas las SIG-álgebras se denota mediante Alg_{SIG} . Desde este punto de vista, se puede contemplar una signatura como una plantilla que define la forma que deben tener todos sus posibles modelos.

Más adelante se dan ciertos criterios para elegir un modelo como significado de una especificación; a tal efecto, se presenta a continuación una SIG-álgebra particular, fácil de construir: la SIG-álgebra resultante de considerar como álgebra el conjunto de términos. Sea $\text{SIG} = (S, \text{OP})$ una signatura y $V \in \text{Vars}_{\text{SIG}}$, el álgebra de términos (ing., term-álgebra) sobre SIG y V, denotada por $T_{\text{SIG}}(V)$, es el álgebra $T_{\text{SIG}}(V) = (S_T, \text{OP}_T)^8$, siendo el S-conjunto S_T , $S_T = (s_T)_{s \in S}$, los conjuntos de base y el $\langle S^* \times S \rangle$ -conjunto OP_T , $\text{OP}_T = ((\text{op}_T)_{w \rightarrow s})_{w \in S^*, s \in S}$, las operaciones, definidos como:

- $\forall s : s \in S : s_T \in S_T$, definido como: $s_T \equiv T_{\text{SIG},s}(V)$
- $\forall \text{op} : \text{op} \in \text{OP}_{s_1 \dots s_n \rightarrow s} : \text{op}_T : T_{\text{SIG},s_1}(V) \times \dots \times T_{\text{SIG},s_n}(V) \rightarrow T_{\text{SIG},s}(V) \in \text{OP}_T$, definida como:
 $\text{op}_T(t_1, \dots, t_n) \equiv \text{op}(t_1, \dots, t_n)$.

Resumiendo, los conjuntos de base son todos los términos que se pueden construir a partir de los símbolos de operaciones y de las variables, y las operaciones son las reglas de formación de términos considerando la signatura concreta. De manera análoga, se puede definir el álgebra de términos sin variables. Como ejemplo, en la fig. 1.8 se construye T_{NAT} .

⁸ Para mayor claridad, se omiten algunos subíndices.

$$\begin{aligned}
S_T &= (\text{nat}_T, \text{bool}_T) \\
\text{nat}_T &\equiv T_{\text{NAT}, \text{nat}} = \{\text{cero}\} \cup \{\text{suc}(x) / x \in T_{\text{NAT}, \text{nat}}\} \cup \{\text{suma}(x, y) / x, y \in T_{\text{NAT}, \text{nat}}\} \\
\text{bool}_T &\equiv T_{\text{NAT}, \text{bool}} = \{\text{cierto}\} \cup \{\text{falso}\} \cup \{\text{ig}(x, y) / x, y \in T_{\text{NAT}, \text{nat}}\} \\
\text{OP}_T: \text{cero}_T &\rightarrow T_{\text{NAT}, \text{nat}}; \text{cero}_T \equiv \text{cero} \\
\text{suc}_T: T_{\text{NAT}, \text{nat}} &\rightarrow T_{\text{NAT}, \text{nat}}; \text{suc}_T(x) \equiv \text{suc}(x) \\
\text{suma}_T: T_{\text{NAT}, \text{nat}} \times T_{\text{NAT}, \text{nat}} &\rightarrow T_{\text{NAT}, \text{nat}}; \text{suma}_T(x, y) \equiv \text{suma}(x, y) \\
\text{ig}_T: T_{\text{NAT}, \text{nat}} \times T_{\text{NAT}, \text{nat}} &\rightarrow T_{\text{NAT}, \text{bool}}; \text{ig}_T(x, y) \equiv \text{ig}(x, y) \\
\text{cierto}_T, \text{falso}_T &\rightarrow T_{\text{NAT}, \text{bool}}; \text{cierto}_T \equiv \text{cierto}, \text{falso}_T \equiv \text{falso}
\end{aligned}$$

Fig. 1.8: álgebra de términos $T_{\text{NAT}} = (S_T, \text{OP}_T)$ para la signatura NAT.

1.2.3 Evaluación de un término dentro de un álgebra

Una vez determinada la interpretación de cada símbolo de una signatura SIG dentro de una SIG-álgebra A , es inmediato definir un mecanismo para calcular el valor representado por un término de T_{SIG} en A . Previamente, es necesario introducir el concepto de homomorfismo entre álgebras.

Dados dos S-conjuntos A y B , definimos una S-aplicación $f: A \rightarrow B$ como una familia de aplicaciones indexada por los elementos de S , $f = (f_s: A_s \rightarrow B_s)_{s \in S}$. Clasificamos f como inyectiva, exhaustiva o biyectiva, si y sólo si todas las f_s lo son a la vez. Entonces, dadas la signatura $\text{SIG} = (S, \text{OP})$ y dos SIG-álgebras A_1 y A_2 , un (S-)homomorfismo (ing., homomorphism) de A_1 a A_2 es una S-aplicación $f: A_1 \rightarrow A_2$ que cumple:

$$\begin{aligned}
- \forall c: c \in \text{OP}_{\rightarrow s}: f_s(c_{A_1}) &= c_{A_2} \\
- \forall \text{op}: \text{op} \in \text{OP}_{s_1 \dots s_n \rightarrow s} \wedge n > 0: \forall t_1: t_1 \in s_1: \dots \forall t_n: t_n \in s_n: \\
&f_s(\text{op}_{A_1}(t_1, \dots, t_n)) = \text{op}_{A_2}(f_{s_1}(t_1), \dots, f_{s_n}(t_n))
\end{aligned}$$

Si el homomorfismo es biyectivo se llama isomorfismo; si entre dos álgebras A_1 y A_2 se puede establecer un isomorfismo⁹, entonces decimos que A_1 y A_2 son isomorfas (ing., isomorphic) y lo denotamos por $A_1 \approx A_2$. Remarquemos que si $A_1 \approx A_2$ y $A_2 \approx A_3$, entonces $A_1 \approx A_3$.

Por ejemplo, sean la signatura NAT y el álgebra NAT presentadas en el apartado anterior, y sea la NAT-álgebra $\text{NAT}^2 = (S_{\text{NAT}^2}, \text{OP}_{\text{NAT}^2})$ de los naturales módulo 2 definida como:

$$S_{\text{NAT}^2} = (\text{nat}_{\text{NAT}^2}, \text{bool}_{\text{NAT}^2}); \text{nat}_{\text{NAT}^2} \equiv \{0, 1\}, \text{bool}_{\text{NAT}^2} \equiv B$$

⁹ Notemos que en los isomorfismos es indiferente el sentido de la función: si existe un isomorfismo de A a A' existe también un isomorfismo de A' a A .

$$\begin{aligned}
OP_{NAT2} &= \{\text{cero}_{NAT2}, \text{suc}_{NAT2}, \text{suma}_{NAT2}, \text{ig}_{NAT2}, \text{cierto}_{NAT2}, \text{falso}_{NAT2}\} \\
\text{cero}_{NAT2} &\equiv 0 \\
\text{suc}_{NAT2}(0) &\equiv 1, \text{suc}_{NAT2}(1) \equiv 0 \\
\text{suma}_{NAT2}(0, 0) &= \text{suma}_{NAT2}(1, 1) \equiv 0, \text{suma}_{NAT2}(1, 0) = \text{suma}_{NAT2}(0, 1) \equiv 1 \\
\text{ig}_{NAT2} &\equiv = \\
\text{cierto}_{NAT2} &\equiv C, \text{falso}_{NAT2} \equiv F
\end{aligned}$$

Entonces, la aplicación $f: NAT \rightarrow NAT2$, definida por $f(n) = 0$, si n es par, y $f(n) = 1$, si n es impar, es un homomorfismo, dado que, por ejemplo:

$$\begin{aligned}
f(\text{cero}_{NAT}) &= f(0) = 0 = \text{cero}_{NAT2} \\
f(\text{suc}_{NAT}(2n)) &= f(2n+1) = 1 = \text{suc}_{NAT2}(0) = \text{suc}_{NAT2}(f(2n)) \\
f(\text{suc}_{NAT}(2n+1)) &= f(2(n+1)) = 0 = \text{suc}_{NAT2}(1) = \text{suc}_{NAT2}(f(2n+1))
\end{aligned}$$

y así para el resto de operaciones.

Una vez introducido el concepto de homomorfismo, puede definirse la evaluación de un término sobre su signatura: dados una signatura $SIG = (S, OP)$, una SIG -álgebra A y un conjunto de variables V sobre SIG , definimos:

- La función de evaluación de términos sin variables dentro de A , $\text{eval}_A: T_{SIG} \rightarrow A$, que es única dada una interpretación de los símbolos de la signatura dentro del álgebra, como un S -homomorfismo:

$$\begin{aligned}
\Diamond \forall c: c \in T_{SIG} : \text{eval}_A(c) &\equiv c_A \\
\Diamond \forall \text{op}(t_1, \dots, t_n): \text{op}(t_1, \dots, t_n) \in T_{SIG} : \text{eval}_A(\text{op}(t_1, \dots, t_n)) &\equiv \text{op}_A(\text{eval}_A(t_1), \dots, \text{eval}_A(t_n))
\end{aligned}$$

eval_A da la interpretación de los términos sin variables dentro de una álgebra A .

- Una función de asignación de V dentro de A , $\text{as}_{V,A}: V \rightarrow A$, como una S -aplicación. $\text{as}_{V,A}$ da la interpretación de un conjunto de variables dentro del álgebra A .
- La función de evaluación de términos con variables de V dentro del álgebra A , $\text{eval}_{\text{as}_{V,A}}: T_{SIG}(V) \rightarrow A$, como un homomorfismo:

$$\begin{aligned}
\Diamond \forall v: v \in V : \text{eval}_{\text{as}_{V,A}}(v) &\equiv \text{as}_{V,A}(v) \\
\Diamond \forall c: c \in T_{SIG}(V) : \text{eval}_{\text{as}_{V,A}}(c) &\equiv \text{eval}_A(c) \\
\Diamond \forall \text{op}(t_1, \dots, t_n): \text{op}(t_1, \dots, t_n) \in T_{SIG}(V) : \\
&\text{eval}_{\text{as}_{V,A}}(\text{op}(t_1, \dots, t_n)) \equiv \text{op}_A(\text{eval}_{\text{as}_{V,A}}(t_1), \dots, \text{eval}_{\text{as}_{V,A}}(t_n))
\end{aligned}$$

Esta función es única, dada una interpretación eval_A de los símbolos de la signatura dentro del álgebra y dada una asignación $\text{as}_{V,A}$ de variables de V dentro del álgebra.

Para simplificar subíndices, abreviaremos $\text{eval}_{\text{as}_{V,A}}$ por $\text{eval}_{V,A}$.

Por ejemplo, dadas la signatura NAT y la NAT -álgebra NAT introducidas anteriormente, la función de evaluación correspondiente es $eval_{NAT} : T_{NAT} \rightarrow NAT$ definida como:

$$\begin{aligned} eval_{NAT}(cero_T) &\equiv 0, \quad eval_{NAT}(suc_T(x)) \equiv eval_{NAT}(x) + 1 \\ eval_{NAT}(suma_T(x, y)) &\equiv eval_{NAT}(x) + eval_{NAT}(y) \\ eval_{NAT}(ig_T(x, y)) &\equiv (eval_{NAT}(x) = eval_{NAT}(y)) \\ eval_{NAT}(cierto_T) &\equiv C, \quad eval_{NAT}(falso_T) \equiv F \end{aligned}$$

Ahora se puede evaluar dentro del álgebra NAT el término $suc(suma(cero, suc(cero)))$ construido a partir de la signatura NAT :

$$\begin{aligned} eval_{NAT}(suc(suma(cero, suc(cero)))) &= \\ eval_{NAT}(suma(cero, suc(cero))) + 1 &= \\ (eval_{NAT}(cero) + eval_{NAT}(suc(cero))) + 1 &= \\ (0 + (eval_{NAT}(cero) + 1)) + 1 &= (0 + (0 + 1)) + 1 = 2 \end{aligned}$$

Dado el conjunto de variables (de género nat) $V = \{x, y\}$, definimos una función de asignación dentro de NAT como $as_{V,NAT}(x) = 3$ y $as_{V,NAT}(y) = 4$. Dada esta función y la función de evaluación $eval_{NAT}$, la evaluación $eval_{V,NAT}$ dentro de NAT de $suc(suma(x, suc(y)))$ queda:

$$\begin{aligned} eval_{V,NAT}(suc(suma(x, suc(y)))) &= eval_{V,NAT}(suma(x, suc(y))) + 1 = \\ (eval_{V,NAT}(x) + eval_{V,NAT}(suc(y))) + 1 &= \\ (3 + (eval_{V,NAT}(y) + 1)) + 1 &= (3 + (4 + 1)) + 1 = 9 \end{aligned}$$

1.2.4 Ecuaciones y especificaciones algebraicas

Dados los géneros y los símbolos de operación que forman la signatura de un tipo, es necesario introducir a continuación las propiedades que cumplen, de manera que se pueda determinar posteriormente el significado del TAD; para ello, se añaden a la signatura unas ecuaciones o axiomas (ing., *equation* o *axiom*), que forman la llamada especificación algebraica o ecuacional del TAD (ing., *algebraic* o *equational specification*).

Actualmente, la utilidad de las especificaciones formales es indiscutible dentro de los métodos modernos de desarrollo de software. Una especificación, ya sea ecuacional o de otra índole, no sólo proporciona un significado preciso a un tipo de datos asociándole un modelo matemático a partir de su descripción formal, sino que, como ya se ha dicho en la primera sección, responde a cualquier cuestión sobre el comportamiento observable del tipo sin necesidad de consultar el código, y por ello clarifica la misión de un tipo dentro de una aplicación. Además, las especificaciones ecuacionales pueden usarse como un primer prototipo de la aplicación siempre que cumplan determinadas condiciones (v. sección 1.7).

Una especificación algebraica $SPEC = (SIG_{SPEC}, E_{SPEC})$ se compone de:

- Una signatura $SIG_{SPEC} = (S, OP)$.
- Un conjunto E_{SPEC} de ecuaciones que expresan relaciones entre los símbolos de la signatura. Cada ecuación tiene la forma sintáctica $t = t'$, siendo t y t' términos con variables sobre SIG_{SPEC} ; se dice que t es la parte izquierda de la ecuación y t' la parte derecha (ing., left-hand side y right-hand side, respectivamente); definimos $Vars_{t=t'}$ como la unión de las variables de los dos términos. La ecuación $t = t'$ representa la fórmula universal de primer orden $\forall x_1 \forall x_2 \dots \forall x_n: t = t'$, siendo $\{x_1, x_2, \dots, x_n\} = Vars_{t=t'}$.

Para simplificar subíndices, de ahora en adelante abreviaremos SIG_{SPEC} por SIG y E_{SPEC} por E . Asimismo, denotaremos $SPEC = (SIG, E)$ por $SPEC = (S, OP, E)$ cuando sea preferible.

Las ecuaciones se incluyen en los universos de Merlí precedidas por la palabra clave "ecns" y, opcionalmente, por la declaración de sus variables; cuando sea necesario, a estos universos los llamaremos universos de especificación o también universos de definición para distinguirlos de otras clases de universos. Las ecuaciones se escriben en líneas diferentes o, de lo contrario, se separan con el carácter ';':

```

universo BOOL es
  tipo bool
  ops cierto, falso: → bool
  ¬_: bool → bool
  _∨_, _∧_: bool bool → bool
  ecns ∀b∈bool
    ¬ cierto = falso; ¬ falso = cierto
    b ∨ cierto = cierto; b ∨ falso = b
    b ∧ cierto = b; b ∧ falso = falso
  funiverso

```

Fig. 1.9: especificación algebraica para el TAD de los booleanos.

Las ecuaciones definen el comportamiento de las operaciones de la signatura; consideraremos que una operación está definida si las ecuaciones determinan su comportamiento respecto a todas las combinaciones posibles de valores (de los géneros correctos) que pueden tomar sus parámetros. Por ejemplo, por lo que se refiere a la suma en la especificación de los naturales de la fig. 1.10, se escribe su comportamiento respecto a cualquier par de naturales con dos ecuaciones: 1 trata el caso de que el segundo operando sea el natural cero y 2 que sea un natural positivo; por lo que respecta a la especificación de la igualdad, se estudian los cuatro casos resultantes de considerar todas las posibles combinaciones de dos naturales, que pueden ser o bien cero o bien positivos. En la sección

siguiente se da un método de escritura de especificaciones basado en esta idea intuitiva.

```

universo NAT es
  tipo nat, bool
  ops cero: → nat
    suc: nat → nat
    suma: nat nat → nat
    ig: nat nat → bool
    cierto, falso: → bool
  ecns ∀n,m∈ nat
    1) suma(n, cero) = n
    2) suma(n, suc(m)) = suc(suma(n, m))
    3) ig(cero, cero) = cierto
    4) ig(cero, suc(m)) = falso
    5) ig(suc(n), cero) = falso
    6) ig(suc(n), suc(m)) = ig(n, m)
funiverso

```

Fig. 1.10: especificación algebraica para el TAD de los naturales.

También es posible demostrar formalmente que la suma está correctamente definida de la siguiente forma. Dado que todo natural n puede representarse mediante la aplicación n veces de suc sobre cero , abreviadamente $\text{suc}^n(\text{cero})$, basta con demostrar el enunciado: cualquier término de tipo nat que contenga un número arbitrario de sumas puede reducirse a un único término de la forma $\text{suc}^n(\text{cero})$, interpretando las operaciones dentro del modelo de los naturales. Previamente, introducimos un lema auxiliar.

Lema. Todo término t de la forma $t = \text{suc}^X(\text{cero}) + \text{suc}^Y(\text{cero})$ es equivalente (por las ecuaciones del tipo) a otro de la forma $\text{suc}^{X+Y}(\text{cero})$.

Demostración. Por inducción sobre y .

$y = 0$. El término queda $t = \text{suc}^X(\text{cero}) + \text{cero}$, que es igual a $\text{suc}^X(\text{cero})$ aplicando 1.

$y = k$. Hipótesis de inducción: $t = \text{suc}^X(\text{cero}) + \text{suc}^k(\text{cero})$ se transforma en $\text{suc}^{X+k}(\text{cero})$.

$y = k+1$. Debe demostrarse que $t = \text{suc}^X(\text{cero}) + \text{suc}^{k+1}(\text{cero})$ cumple el lema. t también puede escribirse como $t = \text{suc}^X(\text{cero}) + \text{suc}(\text{suc}^k(\text{cero}))$, y aplicando 2 se transforma en $t = \text{suc}(\text{suc}^X(\text{cero}) + \text{suc}^k(\text{cero}))$, que es igual a $\text{suc}(\text{suc}^{X+k}(\text{cero}))$ aplicando la hipótesis de inducción, con lo que t ha podido transformarse finalmente en el término $\text{suc}^{X+k+1}(\text{cero})$, que cumple el enunciado del lema.

Teorema. Todo término de tipo nat que contenga r operaciones de suma y s operaciones suc es equivalente por las ecuaciones a otro término de la forma $\text{suc}^s(\text{cero})$.

Demostración. Por inducción sobre r .

$r = 0$. El término es de la forma $\text{suc}^s(\text{cero})$ y cumple el enunciado.

$r = k$. Hipótesis de inducción: el término t de tipo nat con k operaciones de suma y s operaciones suc se transforma en $\text{suc}^s(\text{cero})$.

$r = k+1$. Sea $\alpha = \text{suc}^x(\text{cero}) + \text{suc}^y(\text{cero})$ un subtérmino de t que no contiene ninguna suma (siempre existirá al menos uno pues $r = k + 1 > 0$). Aplicando el lema sobre α se obtiene otro término $\beta = \text{suc}^{x+y}(\text{cero})$ que elimina la (única) suma y conserva el número de apariciones de suc , y sustituyendo α por β dentro de t , resulta en un término con k operaciones de suma y s operaciones suc , siendo posible aplicar la hipótesis de inducción y así obtener el término $\text{suc}^s(\text{cero})$.

En el apartado siguiente se estudia con mayor detalle el significado de esta demostración. Básicamente, se enuncia una biyección entre los términos de la forma $\text{suc}^n(\text{cero})$ y los naturales, definida por $\text{suc}^n(\text{cero}) \leftrightarrow n$, y a continuación se generaliza la biyección a isomorfismo considerando no el conjunto de términos sino el álgebra de términos. En el caso general, será necesario una manipulación adicional que dará lugar a la denominada álgebra cociente de términos que también se define más adelante.

Por último, se introduce la noción de satisfacción de una ecuación. Sea e la ecuación $t_1 = t_2$, siendo t_1 y t_2 términos con variables sobre una signatura SIG , y sea $V = \text{Vars}_e$; decimos que e es válida dentro de una SIG -álgebra A (también se dice que A satisface e) si, para toda función $\text{as}_{V,A}: V \rightarrow A$ de asignación, se cumple $\text{eval}_{V,A}(t_1) =_A \text{eval}_{V,A}(t_2)$, siendo $=_A$ la igualdad dentro del álgebra A y eval_A la función de evaluación correspondiente. Por extensión, una SIG -álgebra A satisface una especificación $\text{SPEC} = (\text{SIG}, E)$ (también se dice que A es una SPEC -álgebra) si A satisface todas las ecuaciones de E ; el conjunto de álgebras que satisfacen una especificación SPEC se denota mediante Alg_{SPEC} .

1.2.5 Modelo inicial de una especificación

Dada una especificación ecuacional es imprescindible determinar con exactitud cuál o cuáles son los modelos por ella representados; dado el universo NAT de la fig. 1.10, está claro que el álgebra NAT cumple sus ecuaciones (con la interpretación eval_{NAT} de los símbolos de la signatura), pero hay más modelos posibles: los enteros, o las matrices de naturales de dos dimensiones, donde se interpreta la operación suc como sumar el uno a todos sus elementos. Así, hay que establecer ciertos criterios que determinen claramente cuál o cuáles de estas álgebras son el modelo asociado a una especificación, y lo haremos con un ejemplo: investiguemos cuál de las álgebras siguientes:

$M_1 = (N, 0, +1, +)$, naturales con cero, incremento en uno y suma.
 $M_2 = (Z, 0, +1, +)$, enteros con cero, incremento en uno y suma.
 $M_3 = (M_{2 \times 2}(N), (0, 0; 0, 0), (+1, +1; +1, +1), +)$, matrices 2×2 de naturales con matriz cero, incremento en uno de todos los componentes y suma de matrices.
 $M_4 = (N, 0, +1, \cdot)$, naturales con cero, incremento en uno y producto.
 $M_5 = (N, 0, +1, +, -)$, naturales con cero, incremento en uno, suma y resta.
 $M_6 = (N, 0, +1, \text{mod } 2)$, naturales con cero, incremento en uno, suma y resto de la división por 2.
 $M_7 = (N, Z, 0, +1, | |)$ naturales y enteros con cero, incremento en uno y valor absoluto.
 $M_8 = (\{*\}, f, g, h)$, modelo con un único elemento en su conjunto base, con las operaciones definidas por $f \equiv *, g(*) \equiv * \text{ y } h(*, *) \equiv *$.

es el modelo de la especificación $Y = (SY, EY)$:

universo Y es
tipo y
ops $\text{cero}: \rightarrow y$
 $\text{suc}: y \rightarrow y$
 $\text{op}: y \text{ y } y \rightarrow y$
ecns $\forall n, m \in y$
 $1) \text{op}(n, \text{cero}) = n; 2) \text{op}(n, \text{suc}(m)) = \text{suc}(\text{op}(n, m))$
funiverso

Dado que buscamos el modelo asociado a una especificación, es imprescindible que este modelo presente unos conjuntos de base y unas operaciones que respondan a la plantilla determinada por la signatura. Así, se puede hacer una primera criba de las álgebras introducidas, porque hay algunas que no son SY-álgebras: M_5 tiene cuatro símbolos de operación (tendríamos que olvidar la suma o la resta para obtener una SY-álgebra), M_6 tiene tres, pero las aridades no coinciden (no hay ninguna operación que se pueda asociar a op), y M_7 define dos géneros (tendríamos que olvidar uno para obtener una SY-álgebra). El resto de álgebras son SY-álgebras y la interpretación de los símbolos de la signatura es inmediata en cada caso, como también la función de evaluación resultante (v. fig. 1.11).

Por lo que concierne a la satisfacción de las ecuaciones en las SY-álgebras, es fácil ver que M_1, M_2, M_3 y M_8 son Y-álgebras, pero que M_4 no lo es, porque no cumple la propiedad $m \cdot 0 = 0$. Por ello, puede descartarse M_4 como posible modelo, dado que ni siquiera cumple las ecuaciones; quedan pues cuatro candidatos. Para determinar cuál o cuáles son los buenos, construiremos una nueva SY-álgebra a partir del álgebra de términos, incorporando la información que proporcionan las ecuaciones, y estudiaremos la isomorfía entre ella y las álgebras. Concretamente, consideramos que dos términos son equivalentes si y sólo si se deduce su igualdad sintáctica manipulándolos con las ecuaciones de la especificación; el resultado es la llamada álgebra cociente de términos, que se introduce a continuación.

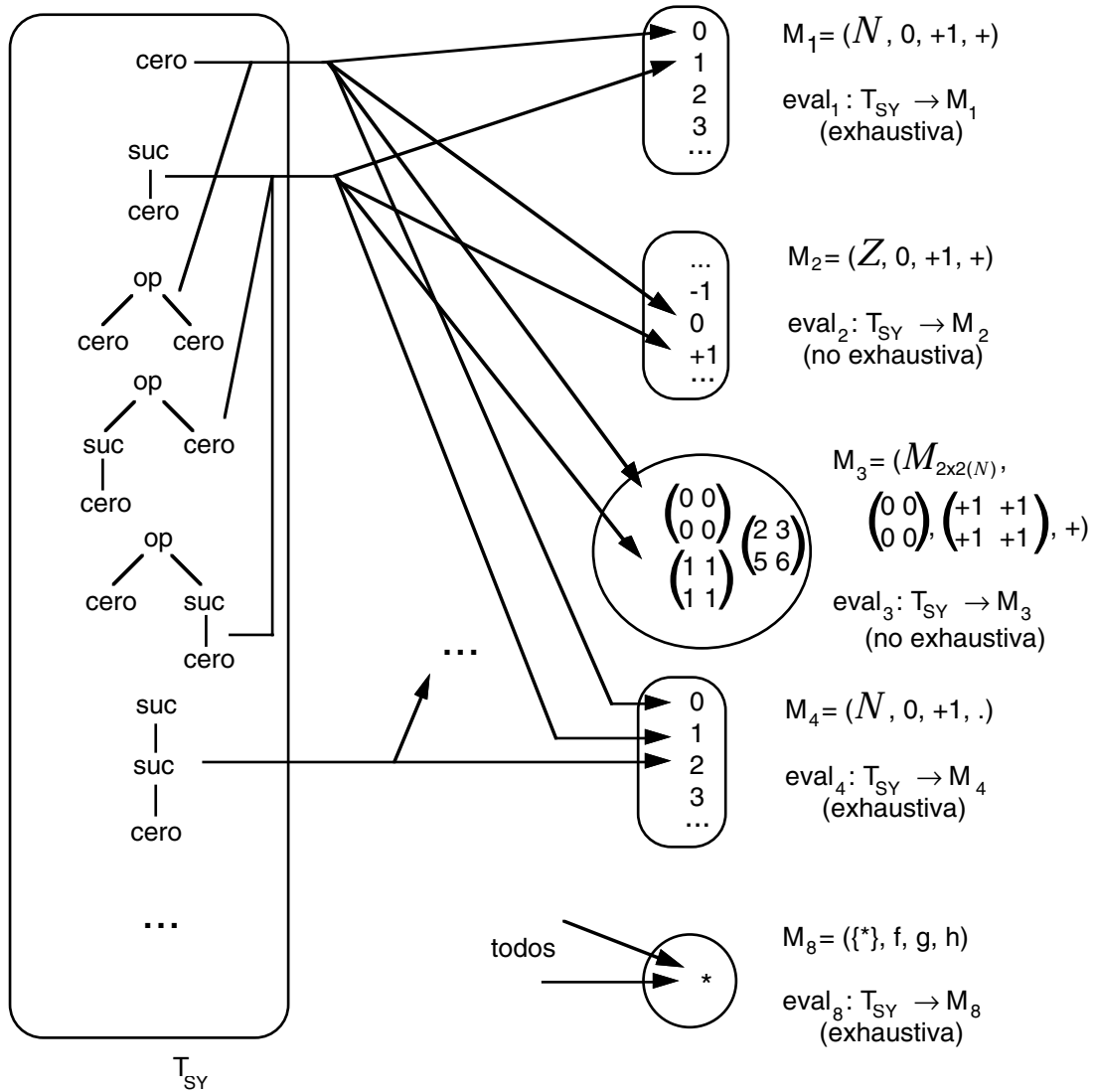


Fig. 1.11: algunas SY-álgebras y las funciones de evaluación de T_{SY} dentro de ellas.

Dada la especificación $SPEC = (SIG, E)$, siendo $SIG = (S, OP)$, se define la congruencia \equiv_E inducida por las ecuaciones de E como la menor relación que cumple:

- \equiv_E es una relación de equivalencia.
- Están dentro de una misma clase de equivalencia todos aquellos términos que puede demostrarse por las ecuaciones que son iguales:

$$\forall e \equiv t_1 = t_2 : e \in E : \forall as_{Vars_e, T_{SIG}} : Vars_e \rightarrow T_{SIG} : eval_{Vars_e, T_{SIG}}(t_1) \equiv_E eval_{Vars_e, T_{SIG}}(t_2).$$

- Cualquier operación aplicada sobre diversas parejas de términos congruentes da como resultado dos términos igualmente congruentes:

$$\forall op: op \in OP_{s_1 \dots s_n \rightarrow s} : \forall t_1, t'_1 \in T_{SIG, s_1} \dots \forall t_n, t'_n \in T_{SIG, s_n} : \\ t_1 \equiv_E t'_1 \wedge \dots \wedge t_n \equiv_E t'_n \Rightarrow op(t_1, \dots, t_n) \equiv_E op(t'_1, \dots, t'_n).$$

Entonces se define el álgebra cociente de términos (ing., quotient-term algebra), denotada por T_{SPEC} , como el resultado de particionar T_{SIG} usando \equiv_E , $T_{SPEC} \equiv T_{SIG} / \equiv_E$; concretamente, $T_{SPEC} = (S_Q, OP_Q)$, siendo S_Q un S -conjunto y OP_Q un $\langle S^* \times S \rangle$ -conjunto, definidos:

- $\forall s: s \in S: s_Q \in S_Q$, siendo $s_Q \equiv \{ [t] / t \in T_{SIG, s} \}$, siendo $[t] \equiv \{ t' \in T_{SIG} / t' \equiv_E t \}$
- $\forall c: c \in OP \rightarrow s: c_Q \in OP_Q$, donde $c_Q \equiv [c]$
- $\forall op: op \in OP_{s_1 \dots s_n \rightarrow s} : op_Q: s_{1_Q} \times \dots \times s_{n_Q} \rightarrow s_Q \in OP_Q$, donde $op_Q([t_1], \dots, [t_n]) \equiv [op(t_1, \dots, t_n)]$

Los elementos de los conjuntos de base de este álgebra son las clases de equivalencia resultado de particionar T_{SPEC} usando \equiv_E , compuestas por términos sin variables cuya igualdad es deducible por las ecuaciones. En la fig. 1.12 se muestra el álgebra cociente de términos T_Y para la especificación Y ; las correspondencias (1), (2) y (3) se calculan:

- (1) $op(cero, cero) = cero$, aplicando la ecuación 1 con $n = cero$.
- (2) $op(suc(cero), cero) = suc(cero)$, aplicando la ecuación 1 con $n = suc(cero)$.
- (3) $op(cero, suc(cero)) = suc(op(cero, cero)) = suc(cero)$, aplicando la ecuación 2 con $m = n = cero$ y después la ecuación 1 con $n = cero$.

La importancia del álgebra cociente de términos radica en que, si consideramos cada una de las clases componentes como un objeto del TAD que se quiere especificar, entonces T_{SPEC} es el modelo. Para ser más concretos, definimos como modelo del tipo cualquier álgebra isomorfa a T_{SPEC} ; en el ejemplo, M_1 es isomorfa a T_Y , mientras que M_2 , M_3 y M_8 no lo son (como queda claro en la fig. 1.12), y por ello se puede decir que el modelo del TAD son los naturales con operaciones cero, incremento en uno y suma. La isomorfía establece la insensibilidad a los cambios de nombre de los símbolos de la signatura: es lo mismo escribir $[cero]$ que 0, $[suc(cero)]$ que 1, etc., siempre que las propiedades que cumplan los símbolos sean las mismas.

El álgebra cociente de términos asociada a la especificación $SPEC$ cumple ciertas propiedades:

- T_{SPEC} es generada: todos sus valores son generados por las operaciones del tipo (no contiene datos inalcanzables desde las operaciones).
- T_{SPEC} es típica: dos términos están dentro de la misma clase si y sólo si por las ecuaciones se demuestra su igualdad (se dice que T_{SPEC} no confunde los términos).
- T_{SPEC} es inicial dentro de la clase de $SPEC$ -álgebras: para toda $SPEC$ -álgebra A , se puede encontrar un único homomorfismo de T_{SPEC} en A (v. fig. 1.13).

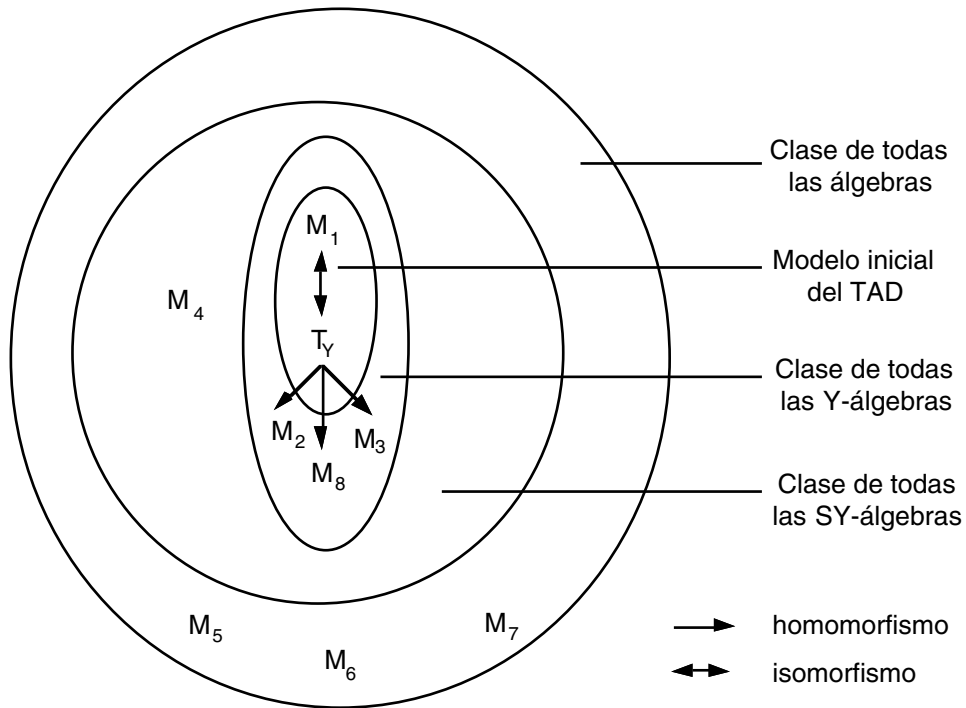


Fig. 1.13: clasificación de todas las álgebras existentes respecto a la especificación Y.

En la fig. 1.14 se muestra el álgebra cociente de términos T_{ENTERO} asociada a la especificación ENTERO. Se podría demostrar que T_{ENTERO} es isomorfa a los enteros $(\mathbb{Z}, 0, +1, -1, +)$ tomando $\text{entero}_Q \approx \mathbb{Z}$, $[\text{suc}^n(\text{cero})] \approx n$, $[\text{pred}^n(\text{cero})] \approx -n$, $[\text{cero}] \approx 0$ y la asociación intuitiva de los símbolos de operación con las operaciones de $(\mathbb{Z}, 0, +1, -1, +)$.

universo ENTERO es

tipo entero

ops cero: \rightarrow entero

suc, pred: entero \rightarrow entero

suma: entero entero \rightarrow entero

ecns $\forall n, m \in \text{entero}$

suc(pred(m)) = m

pred(suc(m)) = m

suma(n, cero) = n

suma(n, suc(m)) = suc(suma(n, m))

suma(n, pred(m)) = pred(suma(n, m))

funiverso

$T_{\text{ENTERO}} = ((\text{entero}_Q), \{\text{cero}_Q, \text{suc}_Q, \text{pred}_Q, \text{suma}_Q\})$

$\text{entero}_Q \equiv \{ [\text{suc}^n(\text{cero})] / n \geq 1 \} \cup \{ [\text{cero}] \} \cup \{ [\text{pred}^n(\text{cero})] / n \geq 1 \}$

$\text{cero}_Q \equiv [\text{cero}]$

$\text{suc}_Q([\text{suc}^n(\text{cero})]) \equiv [\text{suc}^{n+1}(\text{cero})], n \geq 0$

$\text{suc}_Q([\text{pred}^n(\text{cero})]) \equiv [\text{pred}^{n-1}(\text{cero})], n \geq 1$

$\text{pred}_Q([\text{suc}^n(\text{cero})]) \equiv [\text{suc}^{n-1}(\text{cero})], n \geq 1$

$\text{pred}_Q([\text{pred}^n(\text{cero})]) \equiv [\text{pred}^{n+1}(\text{cero})], n \geq 0$

$\text{suma}_Q([\text{suc}^n(\text{cero})], [\text{suc}^m(\text{cero})]) \equiv$

$\equiv [\text{suc}^{n+m}(\text{cero})], n, m \geq 0$

$\text{suma}_Q([\text{suc}^n(\text{cero})], [\text{pred}^m(\text{cero})]) \equiv$

$\equiv [\text{suc}^{n-m}(\text{cero})], n \geq m \geq 1 \dots \text{etc.}$

Fig. 1.14: un álgebra cociente de términos para los enteros.

1.2.6 Otros modelos posibles

En el apartado anterior se ha asociado como modelo (semántica) de un TAD la clase isomorfa al álgebra cociente de términos; es lo que se conoce como semántica inicial (ing., initial semantics) de un TAD (llamada así porque hay un homomorfismo único de T_{SPEC} a todas las demás álgebras de Alg_{SPEC}), que se caracteriza porque todos los valores son alcanzables a partir de las operaciones y porque dos términos son iguales, si y sólo si puede deducirse de las ecuaciones. En el resto de este libro se interpretará la especificación de un TAD con este significado; no obstante, conviene saber que el enfoque inicial no es el único y en este apartado introduciremos muy brevemente otros existentes.

Una crítica que recibe frecuentemente el modelo inicial es su bajo grado de abstracción. Por ejemplo, consideremos la especificación de la fig. 1.15, de la que se pretende que represente los conjuntos de naturales y dos términos sobre su signatura, $t_1 = \text{añade}(\text{añade}(\emptyset, \text{cero}), \text{cero})$ y $t_2 = \text{añade}(\emptyset, \text{cero})$.

```

universo CJT_NATS es
  tipo cjt, nat, bool
  ops  Ø: → cjt
        añade: cjt nat → cjt
        _∈_: nat cjt → bool
        cero: → nat
        suc: nat → nat
        ig: nat nat → bool
        cierto, falso: → bool
        _∨_: bool bool → bool
  ecns ... { especificación de ∨ e ig }
        n ∈ Ø = falso
        n2 ∈ añade(s, n1) = ig(n1, n2) ∨ n2 ∈ s
funiverso

```

Fig. 1.15: ¿una especificación para los conjuntos?

Según el enfoque inicial, t_1 y t_2 denotan valores diferentes porque no puede deducirse por las ecuaciones que sean iguales; ahora bien, ambos términos se comportan exactamente igual, porque todo natural que pertenece al conjunto representado por t_1 también pertenece al conjunto representado por t_2 . Podríamos interpretar, pues, que t_1 y t_2 son el mismo valor si rechazamos el principio de tipicidad del modelo y afirmamos que dos términos son iguales, salvo que se pueda deducir por las ecuaciones que son diferentes. Esta es la semántica final (ing., final semantics) de un TAD (llamada así porque hay un homomorfismo único desde

todas las demás álgebras de Alg_{SPEC} hacia el modelo), que es igualmente una clase de isomorfía y que también es generada, como la semántica inicial.

Notemos que la semántica inicial de la especificación de la fig. 1.15 es N^* (las secuencias de naturales) y la semántica final es $P(N)$ (los conjuntos de naturales). Para conseguir que $P(N)$ sea el modelo inicial, hay que añadir las ecuaciones:

$$\begin{aligned}\text{añade}(\text{añade}(s, n_1), n_2) &= \text{añade}(\text{añade}(s, n_2), n_1) \\ \text{añade}(\text{añade}(s, n), n) &= \text{añade}(s, n)\end{aligned}$$

Es decir, la semántica inicial, a veces, obliga a sobre-especificar los universos introduciendo ecuaciones que no son importantes para el uso del universo dentro de un programa.

Para considerar la igualdad dentro de la semántica final (es decir, la congruencia inducida por las ecuaciones), se hacen experimentos sobre unos géneros especiales llamados géneros observables; a tal efecto, se cambia la definición de una especificación SPEC y se define $\text{SPEC} = (V, S, \text{OP}, E)$, siendo V (subconjunto de S) los géneros observables; por eso, a los términos que son de un género V - S se los denomina términos no observables. Intuitivamente, dos términos de un género no observable son iguales si, al aplicarles cualquier operación que devuelva un valor observable, el resultado es el mismo. En el ejemplo de los conjuntos, si definimos `bool` y `nat` como los géneros observables, la única operación sobre términos no observables que da resultado observable es la de pertenencia, por lo que dos términos no observables t_1 y t_2 son iguales dentro de la semántica final si $\forall n: n \in \text{nat}: n \in t_1 = n \in t_2$.

Una tercera opción es la semántica de comportamiento (ing., behavioural semantics), que tiene un principio de funcionamiento parecido a la semántica final obviando la restricción de isomorfía de la clase de modelos. Por ejemplo, en el caso de la especificación `CJT_NAT`, se toman como modelo N^* y $P(N)$ a la vez y, además, todas las álgebras con el mismo comportamiento observable, como los conjuntos con repeticiones y las secuencias sin repeticiones. Todos estos modelos forman parte del álgebra de comportamiento por `CJT_NAT`, a pesar de que sus conjuntos base no son isomorfos entre sí.

Finalmente, también se puede considerar toda la clase Alg_{SPEC} como la clase de modelos; es la llamada semántica laxa (ing., loose semantics). En este caso, el modelo se toma como punto de partida y se va restringiendo poco a poco (hay diversos tipos de restricciones) hasta llegar a un modelo ya aceptable como resultado. La ventaja sobre otros tipos de semánticas es que se adapta mejor al concepto de desarrollo de aplicaciones. Por ejemplo, supongamos que la especificación de los conjuntos se añade una operación para elegir un elemento cualquiera del conjunto, `elige: cjt \rightarrow nat`; si no importa cuál es el elemento concreto elegido, se puede añadir la ecuación `elige(s) \in s = cierto`, que no impone ninguna estrategia; en algún momento, no obstante, debe restringirse el modelo añadiendo las ecuaciones necesarias para obtener un tipo de comportamiento conocido, por ejemplo, `elige?(s) = mínimo(s)`.

1.3 Construcción sistemática de especificaciones

En la sección anterior hemos estudiado las características de una especificación dada y hemos encontrado su modelo a posteriori. Ahora bien, en el proceso de desarrollo de software la situación acostumbra a ser la contraria: a partir de un TAD que formará parte de un programa, del que se conoce su comportamiento (quizás del todo, quizás sólo se tiene una idea intuitiva), el problema consiste en encontrar una especificación que lo represente y es por ello que en esta sección se estudia un método general para la construcción de especificaciones, basado en una clasificación previa de las operaciones del TAD. Antes introducimos en el primer apartado un concepto útil para escribir especificaciones con mayor comodidad, que se estudiará en profundidad en la sección 1.6.

1.3.1 Introducción al uso de especificaciones

Hasta ahora, al especificar un TAD no hay manera de aprovechar la especificación de otro tipo que se pudiera necesitar, aunque ya esté definida en otro universo; es un ejemplo conocido la especificación de los naturales con operación de igualdad, que precisa repetir la especificación de los booleanos. Es evidente que se necesita un medio para evitar esta redundancia; por tanto, se incluye una nueva cláusula en el lenguaje que permite usar especificaciones ya existentes desde un universo cualquiera. En la fig. 1.16 se muestra una nueva especificación de los naturales que "usa" el universo BOOL de los booleanos y, consecuentemente, puede utilizar todos los símbolos en él definidos.

```

universo NAT es
  usa BOOL
  tipo nat
  ops cero: → nat
      suc: nat → nat
      suma: nat nat → nat
      ig: nat nat → bool
  ecns ∀n,m∈nat
      1) suma(n, cero) = n
      2) suma(n, suc(m)) = suc(suma(n, m))
      3) ig(cero, cero) = cierto
      4) ig(cero, suc(m)) = falso
      5) ig(suc(n), cero) = falso
      6) ig(suc(n), suc(m)) = ig(n, m)
funiverso

```

Fig. 1.16: un universo para los naturales con igualdad usando los booleanos.

1.3.2 Clasificación de las operaciones de una especificación

Como paso previo a la formulación de un método general de construcción de especificaciones, se precisa clasificar las operaciones de la signatura de una especificación respecto a cada género que en ella aparece: dada una especificación $SPEC = (S, OP, E)$ y un género $s \in S$, se define el conjunto de operaciones constructoras de OP respecto a s , $constr_{OP_s}$, como el conjunto de operaciones de OP que devuelven un valor de género s ; y el conjunto de operaciones consultoras de OP respecto a s , $consul_{OP_s}$, como el conjunto de operaciones de OP que devuelven un valor de género diferente de s :

$$constr_{OP_s} \equiv \{op \in OP_{w \rightarrow s}\}.$$

$$consul_{OP_s} \equiv \{op \in OP_{s_1 \dots s_n \rightarrow s'} / s \neq s' \wedge \exists i : 1 \leq i \leq n : s_i = s\}.$$

Dentro de las operaciones constructoras, destaca especialmente el conjunto de operaciones constructoras generadoras, que es un subconjunto mínimo de las operaciones constructoras que permite generar, por aplicaciones sucesivas, los representantes canónicos de las clases de T_{SPEC} (es decir, todos los valores del TAD que queremos especificar); en el caso general, puede haber más de un subconjunto de constructoras generadoras, de los que hay que escoger uno. Las constructoras que no forman parte del conjunto de constructoras generadoras escogidas se llaman (constructoras) modificadoras. Notaremos los dos conjuntos con gen_{OP_s} y $modif_{OP_s}$. El conjunto gen_{OP_s} es puro si no hay relaciones entre las operaciones que lo forman (dicho de otra manera, todo par de términos diferentes formados sólo por constructoras generadoras denota valores diferentes) o impuro si hay relaciones; en este último caso, a las ecuaciones que expresan las relaciones entre las operaciones constructoras generadoras las llamamos impurificadoras.

Por ejemplo, dado el universo $BOOL$ de la fig. 1.9, el conjunto de consultoras es vacío porque todas son constructoras; en lo que se refiere a éstas, parece lógico elegir como representantes canónicos los términos cierto y falso y entonces quedarán los conjuntos $gen_{OP_{bool}} = \{\text{cierto}, \text{falso}\}$ y $modif_{OP_{bool}} = \{\neg, _ \vee _, _ \wedge _ \}$. Ahora bien, también es lícito coger como representantes canónicos los términos cierto y \neg cierto, siendo $gen_{OP_{bool}} = \{\text{cierto}, \neg _ \}$; la propiedad $\neg \neg x = x$ determina la impureza de este conjunto de constructoras generadoras.

No es normal que haya un único género en el universo; por ejemplo, en la especificación de los naturales con igualdad de la fig. 1.16 aparecen nat y $bool$. No obstante, con vistas a la formulación de un método general de especificación (en el siguiente apartado), solamente es necesario clasificar las operaciones respecto a los nuevos géneros que se introducen y no respecto a los que se usan; si sólo se define un nuevo género, como es habitual, se hablará de operaciones constructoras o consultoras sin explicitar respecto a qué género. Volviendo a la especificación de los naturales $NAT = (S, OP, E)$ de la fig. 1.16 y tomando como representantes canónicos los términos (de género nat) $suc^n(\text{cero})$, $n \geq 0$, la clasificación de las operaciones es: $gen_{OP} = \{\text{cero}, \text{suc}\}$, $modif_{OP} = \{\text{suma}\}$ y $consul_{OP} = \{\text{ig}\}$.

1.3.3 Método general de construcción de especificaciones

Para escribir la especificación de un TAD no se dispone de un método exacto, sino que hay que fiarse de la experiencia y la intuición a partes iguales; a veces, una especificación no acaba de resolverse si no se tiene una idea feliz. A continuación se expone un método que da resultados satisfactorios en un elevado número de casos y que consta de tres pasos.

- Elección de un conjunto de operaciones como constructoras generadoras: se buscan las operaciones que son suficientes para generar todos los valores del álgebra cociente de términos del tipo (es decir, para formar los representantes canónicos de las clases). Como ya se ha dicho, puede haber más de uno; en este caso, se puede escoger el conjunto menos impuro o bien el mínimo (criterios que muy frecuentemente conducen al mismo conjunto).
- Aserción de las relaciones entre las constructoras generadoras: escritura de las ecuaciones impurificadoras del tipo. Una posibilidad para conseguir estas ecuaciones consiste en pensar en la forma que ha de tomar el representante canónico de la clase y de qué manera se puede llegar a él.
- Especificación del resto de operaciones, una a una, respecto a las constructoras generadoras (es decir, definición de los efectos de aplicar las operaciones sobre términos formados exclusivamente por constructoras generadoras, pero sin suponer que estos términos son representantes canónicos). Recordemos que el objetivo final es especificar cada operación respecto a todas las combinaciones posibles de valores a los que se puede aplicar, y una manera sencilla de conseguir este objetivo consiste en estudiar los efectos de la aplicación de la operación sobre las constructoras generadoras, que permiten generar por aplicaciones sucesivas todos los valores del género.

En este último paso se debe ser especialmente cuidadoso al especificar las operaciones consultoras para asegurar dos propiedades denominadas consistencia y completitud suficiente: si se ponen ecuaciones de más, se pueden igualar términos que han de estar en clases de equivalencia diferentes del género correspondiente, mientras que si se ponen de menos, se puede generar un número indeterminado de términos (potencialmente infinitos) incongruentes con los representantes de las clases existentes hasta aquel momento y que dan lugar a nuevas clases de equivalencia; en cualquiera de los dos casos, se está modificando incorrectamente la semántica del tipo afectado.

Como ejemplo, se presenta una especificación para el tipo `cjt` de los conjuntos de naturales con operaciones \emptyset : $\rightarrow \text{cjt}$, `añade`: $\text{cjt nat} \rightarrow \text{cjt}$, `_∪_`: $\text{cjt cjt} \rightarrow \text{cjt}$ y `_∈_`: $\text{nat cjt} \rightarrow \text{bool}$. El conjunto $\{\emptyset, \text{añade}\}$ juega el papel de conjunto de constructoras generadoras, porque las operaciones permiten construir todos los valores posibles del tipo (la constructora `_∪_` en particular no es necesaria). La forma general del representante canónico de las clases es `añade(añade(...(añade(\emptyset , n_1), ..., n_k), tomando por ejemplo $n_1 < \dots < n_k$.`

Para modelizar realmente los conjuntos, es necesario incluir dos ecuaciones impurificadoras que expresen que el orden de añadir los naturales al conjunto no es significativo y que dentro del conjunto no hay elementos repetidos:

$$\begin{aligned}\text{añade}(\text{añade}(s, n_1), n_2) &= \text{añade}(\text{añade}(s, n_2), n_1) \\ \text{añade}(\text{añade}(s, n), n) &= \text{añade}(s, n)\end{aligned}$$

No hay ningún otro tipo de relación entre las constructoras generadoras; mediante estas dos ecuaciones todo término formado exclusivamente por constructoras generadoras puede convertirse, eventualmente, en su representante canónico.

Por lo que respecta al resto de operaciones, la aplicación del método da como resultado:

$$\begin{aligned}n \in \emptyset &= \text{falso}; n_1 \in \text{añade}(s, n_2) = \text{ig}(n_1, n_2) \vee (n_1 \in s) \\ \emptyset \cup s &= s; \text{añade}(s_1, n) \cup s_2 = \text{añade}(s_1 \cup s_2, n)\end{aligned}$$

Notemos que hay parámetros que se dejan como una variable, en lugar de descomponerlos en las diversas formas que pueden tomar como combinación de constructoras generadoras, por lo que el parámetro toma igualmente todos los valores posibles del género.

A partir de este enfoque, se puede decir que especificar es dar unas reglas para pasar de un término cualquiera a su representante canónico y así evaluar cualquier expresión sobre un objeto del TAD correspondiente; en la sección 1.7 se insiste en este planteamiento.

1.4 Ecuaciones condicionales, símbolos auxiliares y errores

Introducimos en esta sección tres conceptos adicionales necesarios para poder construir especificaciones que resuelvan problemas no triviales: las ecuaciones condicionales, los tipos y las operaciones auxiliares y el mecanismo de tratamiento de errores.

1.4.1 Ecuaciones condicionales

Hasta ahora, las ecuaciones expresan propiedades que se cumplen incondicionalmente; a veces, no obstante, un axioma se cumple sólo en determinadas condiciones. Esto sucede, por ejemplo, al añadir la operación $\text{saca}: \text{cjt nat} \rightarrow \text{cjt}$ a la especificación de los conjuntos de la sección anterior; su especificación siguiendo el método da como resultado:

$$\begin{aligned}\text{saca}(\emptyset, n) &= \emptyset \\ \text{saca}(\text{añade}(s, n_1), n_2) &= ??\end{aligned}$$

donde la parte derecha de la segunda ecuación depende de si n_1 es igual a n_2 y por ello resultan infinitas ecuaciones:

$$\begin{aligned} \text{saca}(\text{añade}(s, \text{cero}), \text{cero}) &= \text{saca}(s, \text{cero}) \\ \text{saca}(\text{añade}(s, \text{suc}(\text{cero})), \text{cero}) &= \text{añade}(\text{suc}(\text{cero}), \text{saca}(s, \text{cero})) \\ \text{saca}(\text{añade}(s, \text{suc}(\text{cero})), \text{suc}(\text{cero})) &= \text{saca}(s, \text{suc}(\text{cero})) \dots \text{etc.} \end{aligned}$$

El problema se puede solucionar introduciendo ecuaciones condicionales (ing., conditional equation):

- 1) $\text{saca}(\emptyset, n) = \emptyset$
- 2) $[\text{ig}(n_1, n_2) = \text{cierto}] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{saca}(s, n_2)$
- 3) $[\text{ig}(n_1, n_2) = \text{falso}] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{añade}(\text{saca}(s, n_2), n_1)$

(Notemos que la parte derecha de la ecuación 2 no puede ser simplemente s , porque no se puede asegurar que dentro de s no haya más apariciones de n_2 -es decir, no se puede asegurar que el término sea canónico. Es más, incluir esta ecuación en sustitución de 2 provocaría inconsistencias en el tipo, pues se podrían deducir igualdades incorrectas, como $\text{añade}(\emptyset, 1) = \text{saca}(\text{añade}(\text{añade}(\emptyset, 1), 1), 1) = \text{saca}(\text{añade}(\emptyset, 1), 1) = \emptyset$.)

Podemos decir, pues, que una ecuación condicional es equivalente a un número infinito de ecuaciones no condicionales, así como una ecuación con variables es equivalente a un número infinito de ecuaciones sin variables. La sintaxis de Merlí encierra la condición entre corchetes a la izquierda de la ecuación, también en forma de ecuación¹⁰: $[t_0 = t_0'] \Rightarrow t_1 = t_1'$. La ecuación $t_0 = t_0'$ se denomina premisa y $t_1 = t_1'$ se denomina conclusión. Para abreviar, las condiciones de la forma $[t = \text{cierto}]$ o $[t = \text{falso}]$ las escribiremos simplemente $[t]$ o $[\neg t]$, respectivamente; también para abreviar, varias ecuaciones condicionales con idéntica premisa se pueden agrupar en una sola, separando las conclusiones mediante comas. Dada una SPEC-álgebra A con la correspondiente función de evaluación eval_A y una igualdad $=_A$ y, siendo V la unión de las variables de la premisa, diremos que A satisface una ecuación condicional si:

$$\forall s_{V,A}: V \rightarrow A: \{ \text{eval}_{V,A}(t_0) =_A \text{eval}_{V,A}(t_0') \} \Rightarrow \text{eval}_{V,A}(t_1) =_A \text{eval}_{V,A}(t_1').$$

Es necesario destacar un par de cuestiones importantes referentes a las ecuaciones condicionales. Primero, al especificar una operación no constructora generadora usando ecuaciones condicionales hay que asegurarse de que, para una misma parte izquierda, se cubren todos los casos posibles; en otras palabras, para toda asignación de variables de la ecuación debe haber como mínimo una condición que se cumpla. Segundo, notemos que la operación de igualdad sobre los valores de los tipos es necesaria para comprobar la desigualdad pero no la igualdad; así, las tres ecuaciones siguientes son equivalentes:

- 2a) $[\text{ig}(n_1, n_2)] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{saca}(s, n_2)$
- 2b) $[n_1 = n_2] \Rightarrow \text{saca}(\text{añade}(s, n_1), n_2) = \text{saca}(s, n_2)$
- 2c) $\text{saca}(\text{añade}(s, n), n) = \text{saca}(s, n)$

¹⁰ En el marco de la semántica inicial, la premisa no se define normalmente como una única ecuación, sino como la conjunción de varias ecuaciones; no obstante, la forma simplificada aquí adoptada cubre todos los ejemplos que aparecen en el texto y, en realidad, no presenta carencia alguna.

Esto es debido a la existencia de la deducción ecuacional, introducida en la sección 1.7. Lo que nunca se puede suponer es que dos variables diferentes n_1 y n_2 denotan forzosamente dos valores diferentes, porque siempre puede ocurrir que n_1 y n_2 valgan lo mismo (recordemos que la interpretación de una ecuación cuantifica universalmente sus variables).

La redefinición de la congruencia \equiv_E con ecuaciones condicionales queda como sigue:

- Se define \equiv_0 como aquella congruencia en la que todo término del álgebra de términos forma una clase de equivalencia por sí mismo.

- Dada \equiv_i , se define \equiv_{i+1} de la siguiente manera:

$$t \equiv_{i+1} t' \Leftrightarrow t \equiv_i t' \vee$$

$$\exists e \in E, e = [t_0 = t_0'] \Rightarrow t_1 = t_1' \wedge \exists as_{Vars_e, T_{SIG}} : Vars_e \rightarrow T_{SIG} \text{ tal que:}$$

$$1) eval_{Vars_e, T_{SIG}}(t_0) \equiv_i eval_{Vars_e, T_{SIG}}(t_0'), \text{ y}$$

$$2) (eval_{Vars_e, T_{SIG}}(t_1) = t \wedge eval_{Vars_e, T_{SIG}}(t_1') = t') \vee$$

$$(eval_{Vars_e, T_{SIG}}(t_1) = t' \wedge eval_{Vars_e, T_{SIG}}(t_1') = t).$$

- Finalmente, se define \equiv_E como \equiv_∞ .

Vemos que en cada nueva congruencia se fusionan aquellas clases de equivalencia que pueden identificarse como la parte derecha y la parte izquierda de la conclusión de una ecuación condicional, siempre que, para la asignación de variables a la que induce esta identificación, todas las premisas se cumplan (es decir, las dos partes de cada premisa estén en una misma clase de equivalencia, dada la sustitución de variables).

1.4.2 Tipos y operaciones auxiliares

Los tipos y las operaciones auxiliares se introducen dentro de una especificación para facilitar su escritura y legibilidad; algunas veces son incluso imprescindibles para poder especificar las operaciones que configuran una signature dada. Estos símbolos auxiliares son invisibles para los usuarios del TAD (su ámbito es exclusivamente la especificación donde se definen), por lo que también se conocen como tipos y operaciones ocultos o privados (ing., hidden o private).

Por ejemplo, supongamos una signature para el TAD de los naturales con operaciones cero, sucesor y producto, pero sin operación de suma. Por lo que respecta a la especificación del producto, hay que determinar los valores de $prod(cero, n)$ y de $prod(suc(m), n)$; el segundo término es fácil de igualar aplicando la propiedad distributiva:

$$prod(suc(m), n) = suma(prod(m, n), n)$$

Es necesario introducir, pues, una función auxiliar suma que no aparece en la signature

inicial; suma se declara en la cláusula "ops" de la manera habitual pero precedida de la palabra clave "privada" para establecer su ocultación a los universos que usen los naturales:

privada suma: nat nat \rightarrow nat

El último paso consiste en especificar suma como cualquier otra operación de la signatura; por ello, al definir una operación privada es conveniente plantearse si puede ser de interés general, para dejarla pública y que otros universos puedan usarla libremente.

El efecto de la declaración de símbolos auxiliares en el modelo asociado a una especificación ESP es el siguiente. Se considera la signatura $SIG' \subseteq SIG$ que contiene todos los símbolos no auxiliares de la especificación, y se define la restricción de ESP con SIG' , denotada por $\langle\langle T_{ESP} \rangle\rangle_{SIG'}$, como el resultado de olvidar todas las clases del álgebra cociente T_{ESP} cuyo tipo sea auxiliar en SIG y todas las operaciones de T_{ESP} asociadas a operaciones auxiliares de SIG . Entonces, el modelo es la clase de todas las álgebras isomorfas a $\langle\langle T_{ESP} \rangle\rangle_{SIG'}$. La construcción detallada de este nuevo modelo se encuentra en [EhM85, pp. 145-151].

1.4.3 Tratamiento de los errores

Hasta el momento, se ha considerado que las operaciones de una signatura están bien definidas para cualquier combinación de valores sobre la que se apliquen. Sin embargo, normalmente una o más operaciones de una especificación serán funciones parciales, que no se podrán aplicar sobre ciertos valores del dominio de los datos. Veremos en este apartado cómo tratar estas operaciones, y lo haremos con el ejemplo de la fig. 1.17 de los naturales con igualdad y predecesor.

La especificación no define completamente el comportamiento de la operación *pred*, porque no determina el resultado de *pred*(cero), que es un error. Dada la construcción del modelo de un TAD presentada en la sección 1.2, se puede introducir una nueva clase de equivalencia dentro del álgebra cociente de términos que represente este valor erróneo; para facilitar su descripción y la evolución posterior de la especificación, se añade una constante a la signatura, $error_{nat}: \rightarrow nat$, que modeliza un valor de error dentro del conjunto base de *nat*, que representaremos con $[error_{nat}]$. Entonces se puede completar la especificación de *pred* con la ecuación $pred(cero) = error_{nat}$.

Notemos que el representante canónico de la clase $[error_{nat}]$ no es un término formado íntegramente por aplicaciones de *cero* y *suc*; por ello, se debe modificar el conjunto de constructoras generadoras incluyendo en él $error_{nat}$. Este cambio es importante, porque la definición del método general obliga a determinar el comportamiento del resto de operaciones de la signatura respecto a $error_{nat}$. Entre diversas opciones posibles adoptamos la estrategia de propagación de los errores: siempre que uno de los operandos de una operación sea un error, el resultado también es error. Así, deben añadirse algunas ecuaciones a la especificación, entre ellas:

universo NAT es
usa BOOL
tipo nat
ops cero: \rightarrow nat
 suc, pred: nat \rightarrow nat
 suma, mult: nat nat \rightarrow nat
 ig: nat nat \rightarrow bool
ecns $\forall n, m \in \text{nat}$
 1) suma(cero, n) = n
 2) suma(suc(m), n) = suc(suma(m, n))
 3) mult(cero, n) = cero
 4) mult(suc(m), n) = suma(mult(m, n), n)
 5) ig(cero, cero) = cierto
 6) ig(suc(m), cero) = falso
 7) ig(cero, suc(n)) = falso
 8) ig(suc(m), suc(n)) = ig(m, n)
 9) pred(suc(m)) = m
funiverso

Fig. 1.17: un universo para los naturales con predecesor.

$$\begin{array}{ll}
 \text{E1) } \text{suc}(\text{error}_{\text{nat}}) = \text{error}_{\text{nat}} & \text{E2) } \text{pred}(\text{error}_{\text{nat}}) = \text{error}_{\text{nat}} \\
 \text{E3) } \text{suma}(\text{error}_{\text{nat}}, n) = \text{error}_{\text{nat}} & \text{E4) } \text{suma}(n, \text{error}_{\text{nat}}) = \text{error}_{\text{nat}} \\
 \text{E5) } \text{mult}(\text{error}_{\text{nat}}, n) = \text{error}_{\text{nat}} & \text{E6) } \text{mult}(n, \text{error}_{\text{nat}}) = \text{error}_{\text{nat}}
 \end{array}$$

Hay varios problemas en esta solución. Para empezar, las ecuaciones han introducido inconsistencias: por ejemplo, dado el término $\text{mult}(\text{cero}, \text{error}_{\text{nat}})$, se le pueden aplicar dos ecuaciones, 3 y E6; aplicando E6, con $n = \text{cero}$, se obtiene como resultado $\text{error}_{\text{nat}}$, mientras que aplicando 3, con $n = \text{error}_{\text{nat}}$, se obtiene como resultado cero. Es decir, que el mismo término lleva a dos valores diferentes dentro del álgebra en función de la ecuación que se le aplique. Para evitar este problema intolerable, deben protegerse las ecuaciones normales con una comprobación de que los términos utilizados no son erróneos; en la especificación del producto, por ejemplo, se cambian las ecuaciones 3 y 4 por:

$$\begin{aligned}
 & [\text{correcto}_{\text{nat}}(n)] \Rightarrow \text{mult}(\text{cero}, n) = \text{cero} \\
 & [\text{correcto}_{\text{nat}}(\text{mult}(m, n)) \wedge \text{correcto}_{\text{nat}}(\text{suc}(m))] \Rightarrow \\
 & \quad \Rightarrow \text{mult}(\text{suc}(m), n) = \text{suma}(\text{mult}(m, n), n)
 \end{aligned}$$

y lo mismo para el resto de operaciones.

La operación $\text{correcto}_{\text{nat}}: \text{nat} \rightarrow \text{bool}$, que garantiza que un término no representa $\text{error}_{\text{nat}}$, se puede especificar en función de las operaciones constructoras generadoras:

- E7) $\text{correcto}_{\text{nat}}(\text{cero}) = \text{cierto}$
 E8) $\text{correcto}_{\text{nat}}(\text{error}_{\text{nat}}) = \text{falso}$
 E9) $\text{correcto}_{\text{nat}}(\text{suc}(m)) = \text{correcto}_{\text{nat}}(m)$

Un segundo problema surge al modificar la especificación de la igualdad:

- E10) $\text{ig}(\text{error}_{\text{nat}}, n) = \text{error}_{\text{bool}}$ E11) $\text{ig}(m, \text{error}_{\text{nat}}) = \text{error}_{\text{bool}}$

$\text{error}_{\text{bool}}$ es un error de tipo diferente, pues ig devuelve un booleano; es decir, especificando NAT debe modificarse un universo ya existente, introduciendo en él una constructora generadora que obliga a repetir el proceso (y, eventualmente, a modificar otros universos).

Queda claro, pues, que el tratamiento de los errores expande el número de ecuaciones de un universo para solucionar todos los problemas citados; por ejemplo, la especificación de los naturales con predecesor (cuya finalización queda como ejercicio para el lector) pasa de 9 ecuaciones a 21. Este resultado es inevitable con el esquema de trabajo adoptado; ahora bien, podemos tomar algunas convenciones para simplificar la escritura de la especificación, pero sin que varíe el resultado final (v. [ADJ78] para una explicación detallada):

- Todo género introducido en la especificación ofrece implícitamente dos operaciones visibles, la constante de error y el predicado de corrección convenientemente especificado; en el ejemplo de los naturales, $\text{error}_{\text{nat}}: \rightarrow \text{nat}$ y $\text{correcto}_{\text{nat}}: \text{nat} \rightarrow \text{bool}$.
- Todas las ecuaciones de error que no sean de propagación se escribirán en una cláusula aparte y sin explicitar la parte derecha, que es implícitamente el valor de error del tipo adecuado. En Merlí se escriben los errores antes que las otras ecuaciones, a las que por contraposición denominamos ecuaciones correctas o normales.
- Durante la evaluación de términos, los errores se propagan automáticamente; así, no es necesario introducir las ecuaciones de propagación como $\text{suc}(\text{error}_{\text{nat}}) = \text{error}_{\text{nat}}$.
- En el resto de ecuaciones, tanto la parte izquierda como la parte derecha están libres de error (es decir, existen condiciones implícitas); por ello, la ecuación normal $\text{mult}(\text{cero}, n) = \text{cero}$ en realidad significa $[\text{correcto}_{\text{nat}}(n)] \Rightarrow \text{mult}(\text{cero}, n) = \text{cero}$. Con esta suposición, no es necesario comprobar explícitamente que los valores sobre los que se aplica una operación en una ecuación normal están libres de error.

Como resultado, para completar la especificación ejemplo basta con escribir la cláusula error $\text{pred}(\text{cero})$, con la certeza que las convenciones dadas la expanden correctamente.

1.5 Estudio de casos

Se presentan a continuación algunos ejemplos que permiten ejercitar el método general de especificación introducido en el apartado 1.3.3. y que, sobre todo, muestran algunas

excepciones bastante habituales. La metodología de desarrollo que se sigue en esta sección es la base de la especificación de los diversos TAD que se introducirán en el resto del texto. Los ejemplos han sido elegidos para mostrar la especificación de: a) un par de modelos matemáticos clásicos; b) un módulo auxiliar para la construcción de una aplicación; c) una aplicación entera, de dimensión necesariamente reducida. Pueden encontrarse más ejemplos resueltos en el trabajo "Especificació Algebraica de Tipus Abstractes de Dades: Estudi de Casos", escrito por el autor de este libro y publicado en el año 1991 como report LSI-91-5 del Dept. de Llenguatges i Sistemes Informàtics de la Universitat Politècnica de Catalunya.

1.5.1 Especificación de algunos tipos de datos clásicos

a) Polinomios

Queremos especificar los polinomios $\mathbb{Z}[X]$ de una variable con coeficientes enteros, es decir, $p(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + a_nx^n$, $n \in \mathbb{N}$, $\forall i: 0 \leq i \leq n: a_i \in \mathbb{Z}$, con operaciones: cero: \rightarrow poli, que representa el polinomio $p(x) = 0$; añadir: poli entero nat \rightarrow poli, que añade una pareja coeficiente-exponente (abreviadamente, monomio) a un término que representa un polinomio, y la operación evaluar: poli entero \rightarrow entero, que calcula el valor del polinomio en un punto. Está claro que las operaciones constructoras generadoras son cero y añadir, porque cualquier polinomio se puede expresar como un término compuesto íntegramente por ellas; así, el polinomio $p(x)$ tiene como representante canónico el término añadir(añadir(...(añadir(cero, a_{k_0} , k_0), a_{k_1} , k_1), ...), a_{k_r} , k_r), donde sólo aparecen los términos de coeficiente diferente de cero y donde se cumple, por ejemplo, que $k_i < k_{i+1}$.

Es obvio que las constructoras generadoras exhiben ciertas interrelaciones; así, el polinomio $p(x) = 8x$ puede representarse, entre otros, mediante los términos añadir(cero, 8, 1), añadir(añadir(cero, 8, 1), 0, 5), añadir(añadir(cero, 3, 1), 5, 1) y añadir(añadir(cero, 5, 1), 3, 1), aunque estén contruidos con monomios diferentes, los tengan en diferente orden, o presenten monomios de coeficiente cero. Estas propiedades dan lugar a las ecuaciones:

- 1) añadir(añadir(p, a_1 , n_1), a_2 , n_2) = añadir(añadir(p, a_2 , n_2), a_1 , n_1)
- 2) añadir(añadir(p, a_1 , n), a_2 , n) = añadir(p, ENTERO.suma(a_1 , a_2), n)^{11,12}
- 3) añadir(p, ENTERO.cero, n) = p

Para demostrar la corrección de estas ecuaciones, se podría establecer una biyección entre el subconjunto del álgebra cociente de términos resultado de agrupar los términos de género poli contruidos sobre la signatura de los polinomios y $\mathbb{Z}[X]$.

Por último, se especifica la operación evaluar respecto a las constructoras generadoras, es

¹¹ Cuando a una operación como suma la precede un nombre de universo como ENTERO, se está explicitando, por motivos de legibilidad, en qué universo está definida la operación.

¹² A lo largo del ejemplo, el universo ENTERO define los enteros con las operaciones necesarias.

decir, se evalúa cada monomio sobre el punto dado (se eleva a la potencia y el resultado se multiplica por el coeficiente), y se suman los resultados parciales:

4) evaluar(cero, b) = ENTERO.cero

5) evaluar(añadir(p, a, n), b) =
ENTERO.suma(avaluar(p, b), ENTERO.mult(ENTERO.eleva(b, n)))

Para seguir con el ejercicio, se enriquece la especificación con algunas operaciones más:

coeficiente: poli nat \rightarrow entero

suma, mult: poli poli \rightarrow poli

donde la operación coeficiente devuelve el coeficiente asociado a un término de exponente dado dentro un polinomio, y suma y mult se comportan como su nombre indica. Una primera versión de la especificación de la operación coeficiente podría ser:

6) coeficiente(cero, n) = ENTERO.cero

7) coeficiente(añadir(p, a, n), n) = a

8) $[\neg \text{NAT.ig}(n_1, n_2)] \Rightarrow \text{coeficiente}(\text{añadir}(p, a, n_1), n_2) = \text{coeficiente}(p, n_2)$

Es decir, se examinan los monomios que forman el término hasta encontrar el que tiene el exponente dado, y se devuelve el coeficiente asociado; en caso de que no aparezca, la operación acabará aplicándose sobre el término cero, y dará 0. Esta versión, no obstante, presenta una equivocación muy frecuente en la construcción de especificaciones: la ecuación está diseñada para aplicarse sobre representantes canónicos, pero no se comporta correctamente al aplicarse sobre cualquier otro término. Así, el coeficiente del monomio de exponente 1 de los términos añadir(añadir(cero, 3, 1), 5, 1) y añadir(cero, 8, 1) (que son demostrablemente equivalentes) puede ser diferente en función del orden de aplicación de las ecuaciones. La conclusión es que, al especificar cualquier operación de una signatura, no se puede suponer nunca que los términos utilizados en las ecuaciones son canónicos; dicho de otra manera, no es lícito suponer que las ecuaciones de una signatura se aplicarán en un orden determinado (excepto las ecuaciones de error). En este caso, hay que buscar por todo el término los diversos añadir del exponente dado y sumarlos:

7) coeficiente(añadir(p, a, n), n) = ENTERO.suma(a, coeficiente(p, n))

La especificación de la suma de polinomios es:

9) suma(cero, p) = p

10) suma(añadir(q, a, n), p) = añadir(suma(q, p), a, n)

Es decir, se añaden los monomios del primer polinomio sobre el segundo; las cuestiones relativas al orden de escritura, a la existencia de diversos monomios para un mismo exponente y de monomios de coeficiente cero no deben tenerse en cuenta, porque las ecuaciones impurificadoras ya las tratan. Notemos que no es necesario descomponer el segundo parámetro en función de las constructoras generadoras, porque el uso de una variable permite especificar lo mismo con menos ecuaciones.

Finalmente, la especificación de la multiplicación puede realizarse de diferentes maneras; la más sencilla consiste en introducir una operación auxiliar $\text{mult_un}: \text{poli entero nat} \rightarrow \text{poli}$ que calcula el producto de un monomio con un polinomio, de manera que la multiplicación de polinomios consiste en aplicar mult_un reiteradamente sobre todos los monomios de uno de los dos polinomios, y sumar la secuencia resultante de polinomios:

- 11) $\text{mult}(\text{cero}, p) = \text{cero}$
- 12) $\text{mult}(\text{añadir}(q, a, n), p) = \text{POLI.suma}(\text{mult}(q, p), \text{mult_un}(p, a, n))$
- 13) $\text{mult_un}(\text{cero}, a, n) = \text{cero}$
- 14) $\text{mult_un}(\text{añadir}(p, a_1, n_1), a_2, n_2) =$
 $\text{añadir}(\text{mult_un}(p, a_2, n_2), \text{ENTERO.mult}(a_1, a_2), \text{NAT.suma}(n_1, n_2))$

Una segunda posibilidad no precisa de la operación privada, pero a cambio exige descomponer más los parámetros:

- 11) $\text{mult}(\text{cero}, p) = \text{cero}$
- 12) $\text{mult}(\text{añadir}(\text{cero}, a, n), \text{cero}) = \text{cero}$ (también $\text{mult}(p, \text{cero}) = \text{cero}$)
- 13) $\text{mult}(\text{añadir}(\text{cero}, a_1, n_1), \text{añadir}(q, a_2, n_2)) =$
 $\text{añadir}(\text{mult}(\text{añadir}(\text{cero}, a_1, n_1), q), \text{mult}(a_1, a_2), \text{suma}(n_1, n_2))$
- 14) $\text{mult}(\text{añadir}(\text{añadir}(q, a_1, n_1), a_2, n_2), p) =$
 $\text{suma}(\text{mult}(\text{añadir}(q, a_1, n_1), p), \text{mult}(\text{añadir}(\text{cero}, a_2, n_2), p))$

En realidad, se sigue la misma estrategia, pero en vez de la operación auxiliar se explicita el caso $\text{añadir}(\text{cero}, a, n)$, de manera que se especifica mult respecto a términos sin ningún monomio, términos con un único monomio y términos con dos o más monomios. Notemos que el resultado es más complicado que la anterior versión, lo que confirma que el uso de operaciones auxiliares adecuadas simplifica frecuentemente la especificación resultante.

b) Secuencias

Se propone la especificación del TAD de las secuencias o cadenas de elementos provenientes de un alfabeto: dado un alfabeto $V = \{a_1, \dots, a_n\}$, las secuencias V^* se definen:

- $\lambda \in V^*$.
- $\forall s: s \in V^*: \forall v: v \in V: v.s, s.v \in V^*$.

λ representa la secuencia vacía, mientras que el resto de cadenas se pueden considerar como el añadido de un elemento (por la derecha o por la izquierda) a una cadena no vacía.

En la fig. 1.18 se da una especificación para el alfabeto; como no hay ninguna ecuación impurificadora, todas las constantes denotan valores distintos del conjunto base del modelo inicial; se requiere una operación de igualdad para comparar los elementos. Por lo que respecta las cadenas, se propone una signature de partida con las siguientes operaciones: λ , la cadena vacía; $[v]$, que, dado un elemento v , lo transforma en una cadena que sólo consta de v ; $v.c$, que añade por la izquierda el elemento v a una cadena c ; $c_1.c_2$, que concatena dos cadenas c_1 y c_2 (es decir, coloca los elementos de c_2 a continuación de los

elementos de c_1); $\|c\|$, que devuelve el número de elementos de la cadena c ; y $v \in c$, que comprueba si el elemento v aparece o no en la cadena c . A la vista de esta signatura, queda claro que hay dos conjuntos posibles de constructoras generadoras, $\{\lambda, _._ \}$ y $\{\lambda, _, _._ \}$. La fig. 1.19 muestra las especificaciones resultantes en cada caso, que indican claramente la conveniencia de seleccionar el primer conjunto, que es puro; con éste, el representante canónico es de la forma $v_1.v_2. \dots .v_n.\lambda$, que abreviaremos por $v_1v_2 \dots v_n$ cuando convenga.

universo ALFABETO es
usa BOOL
tipo alf
ops $a_1, \dots, a_n: \rightarrow \text{alf}$
 $_ = _$: $\text{alf alf} \rightarrow \text{bool}$
ecns $(a_1 = a_1) = \text{cierto}$; $(a_1 = a_2) = \text{falso}$; ...
funiverso

Fig. 1.18: especificación del alfabeto.

universo CADENA es
usa ALFABETO, NAT, BOOL
tipo cadena
ops
 λ : $\rightarrow \text{cadena}$
 $_ _$: $\text{alf} \rightarrow \text{cadena}$
 $_._$: $\text{alf cadena} \rightarrow \text{cadena}$
 $_ _$: $\text{cadena cadena} \rightarrow \text{cadena}$
 $\| _ \|$: $\text{cadena} \rightarrow \text{nat}$
 $_ \in _$: $\text{alf cadena} \rightarrow \text{bool}$
funiverso

- | | |
|---|--|
| 1) $[v] = v.\lambda$ | 1) $\lambda.c = c$ |
| 2) $\lambda.c = c$ | 2) $c.\lambda = c$ |
| 3) $(v.c_1) \cdot c_2 = v.(c_1 \cdot c_2)$ | 3) $(c_1 \cdot c_2) \cdot c_3 = c_1 \cdot (c_2 \cdot c_3)$ |
| 4) $\ \lambda\ = \text{cero}$ | 4) $v.c = [v].c$ |
| 5) $\ v.c\ = \text{suc}(\ c\)$ | 5) $\ \lambda\ = 0$ |
| 6) $v \in \lambda = \text{falso}$ | 6) $\ [v] \ = \text{suc}(\text{cero})$ |
| 7) $v_1 \in (v_2.c) = (v_1 = v_2) \vee v_1 \in c$ | 7) $\ c_1 \cdot c_2\ = \text{suma}(\ c_1\ , \ c_2\)$ |
| | 8) $v \in \lambda = \text{falso}$ |
| | 9) $v_1 \in [v_2] = (v_1 = v_2)$ |
| | 10) $v \in (c_1 \cdot c_2) = v \in c_1 \vee v \in c_2$ |

Fig. 1.19: signatura (arriba) y especificación (abajo) de las cadenas con los dos conjuntos posibles de constructoras generadoras: $\{\lambda, _._ \}$ (izquierda) y $\{\lambda, _, _._ \}$ (derecha).

A continuación se incorporan diversas operaciones a la signatura inicial. Para empezar, se añade una operación de comparación de cadenas, $_ = _$: $\text{cadena cadena} \rightarrow \text{bool}$, cuya especificación respecto del conjunto de constructoras generadoras es sencilla (no lo sería tanto respecto el conjunto que se ha descartado previamente):

$$\begin{aligned} 8) (\lambda = \lambda) &= \text{cierto} & 9) (v.c = \lambda) &= \text{fals} \\ 10) (\lambda = v.c) &= \text{falso} & 11) (v_1.c_1 = v_2.c_2) &= (v_1 = v_2) \wedge (c_1 = c_2) \end{aligned}$$

Una operación interesante sobre las cadenas es $i_ésimo$: $\text{cadena nat} \rightarrow \text{alf}$, que obtiene el elemento i -ésimo de la cadena definido como $i_ésimo(v_1 \dots v_{i-1} v_i v_{i+1} \dots v_n, i) = v_i$. Primero, se controlan los errores de pedir el elemento que ocupa la posición cero o una posición mayor que la longitud de la cadena:

$$12) \text{error}[(||c|| < i) \vee (\text{NAT.ig}(i, \text{cero}))] \Rightarrow i_ésimo(c, i)^{13}$$

Para el resto de ecuaciones, notemos que el elemento i -ésimo no es el que se ha insertado en i -ésima posición en la cadena, sino que el orden de inserción es inverso a la numeración asignada a los caracteres dentro de la cadena; por ello, las ecuaciones resultantes son:

$$\begin{aligned} 13) i_ésimo(v.c, \text{suc}(\text{cero})) &= v \\ 14) i_ésimo(v.c, \text{suc}(\text{suc}(i))) &= i_ésimo(c, \text{suc}(i)) \end{aligned}$$

Destacamos que la operación ha sido especificada, no sólo respecto a las constructoras generadoras de cadena, sino también respecto a las constructoras generadoras de nat, distinguiendo el comportamiento para el cero, el uno y el resto de naturales.

Introducimos ahora la operación rotar_dr : $\text{cadena} \rightarrow \text{cadena}$ para rotar los elementos una posición a la derecha, $\text{rotar_dr}(v_1 \dots v_{n-1} v_n) = v_n v_1 \dots v_{n-1}$; en vez de especificarla respecto a λ y $_.$, se incorpora una nueva operación privada para añadir elementos por la derecha, de signatura $_.$: $\text{cadena alf} \rightarrow \text{cadena}$ (la sobrecarga del identificador $_.$ no provoca ningún problema), que define un nuevo conjunto de constructoras generadoras; la especificación de rotar_dr respecto este nuevo conjunto es:

$$\begin{aligned} 15) \text{rotar_dr}(\lambda) &= \lambda \\ 16) \text{rotar_dr}(c.v) &= v.c \end{aligned}$$

Es decir, y este es un hecho realmente importante, no es obligatorio especificar todas las operaciones del tipo respecto al mismo conjunto de constructoras generadoras. Es más, en realidad no es obligatorio especificar las operaciones respecto a un conjunto de constructoras generadoras, sino que sólo hay que asegurarse que cada operación se especifica respecto a absolutamente todos los términos posibles del álgebra de términos; ahora bien, siendo la especificación que usa las constructoras generadoras una manera clara y sencilla de conseguir este objetivo, se sigue esta estrategia siempre que sea posible.

¹³ Suponemos que los naturales definen las operaciones de comparación que se necesiten.

La especificación de la operación privada queda:

$$17) \lambda.v = v.\lambda$$

$$18) (v_1.c).v_2 = v_1.(c.v_2)$$

Consideramos a continuación la función medio: cadena \rightarrow alf, que devuelve el elemento central de una cadena. Aplicando el razonamiento anterior, en vez de especificarla respecto a las constructoras generadoras, se hace un estudio por casos según la longitud de la cadena (0, 1, 2 o mayor que 2), y el último caso se expresa de la manera más conveniente. La especificación del comportamiento de la operación respecto a todos estos casos garantiza que el comportamiento de medio está definido para cualquier cadena:

$$19) \text{error medio}(\lambda)$$

$$20) \text{medio}([v]) = v$$

$$21) \text{medio}(v_1.\lambda.v_2) = v_2 \text{ (o bien } v_1)$$

$$22) [|l| > \text{cero}] \Rightarrow \text{medio}(v_1.c.v_2) = \text{medio}(c)$$

En la ecuación 22 es necesario proteger con la condición, de lo contrario, dada una cadena de dos elementos se podría aplicar tanto 21 como 22, y se provocaría error en este caso.

Para aquellas operaciones que, como rotar_dr y medio, se han especificado sin seguir el método general, puede ser conveniente demostrar que su definición es correcta. Para ello, aplicaremos la técnica empleada en el apartado 1.2.4 para verificar la corrección de la suma según se explica a continuación. Sea $\{\lambda, _._ \}$ el conjunto de operaciones constructoras generadoras, siendo $_.$ la operación de añadir un carácter por la izquierda. Ya se ha dicho que este conjunto es puro, y es posible establecer una biyección entre el conjunto de base de las cadenas en el álgebra cociente y el modelo matemático de las cadenas. Para demostrar la corrección de rotar_dr y medio, debe comprobarse que:

$$\text{rotar_dr}_Q([v_n . v_{n-1} . \dots . v_2 . v_1 . \lambda]) = [v_1 . v_n . v_{n-1} . \dots . v_2 . \lambda],$$

$$\text{medio}_Q([\lambda]) = [\text{error}_{\text{alf}}], \text{ y}$$

$$\text{medio}_Q([v_n . v_{n-1} . \dots . v_1 . \lambda]) = [v_{\lfloor (n+1)/2 \rfloor}], n > 0$$

significando el subíndice "Q" la interpretación de la operación en el álgebra cociente. Ambas demostraciones se desarrollan con la ayuda de un lema que demuestra la equivalencia de los añadidos por la derecha y por la izquierda si se aplican en el orden correcto.

Lema. Todo término representando una cadena no vacía de la forma $v_n . v_{n-1} . \dots . v_2 . v_1 . \lambda$, $n > 0$, es equivalente por las ecuaciones al término $v_n . v_{n-1} . \dots . v_2 . \lambda . v_1$.

Demostración. Por inducción sobre n.

$n = 1$. Queda el término $v_1 . \lambda$, que es igual a $\lambda . v_1$ aplicando la ecuación 17.

$n = k$. Hipótesis de inducción: $v_k . v_{k-1} . \dots . v_1 . \lambda$ se transforma en $v_k . v_{k-1} . \dots . v_2 . \lambda . v_1$.

$n = k+1$. Debe demostrarse que $t = v_{k+1} . v_k . \dots . v_1 . \lambda$ cumple el lema. Ello es inmediato ya que puede aplicarse la hipótesis de inducción sobre el subtérmino $t_0 = v_k . \dots . v_1 . \lambda$, obteniendo $t_1 = v_k . v_{k-1} . \dots . \lambda . v_1$. Sustituyendo t_0 por t_1 dentro de t se obtiene el término $v_{k+1} . v_k . v_{k-1} . \dots . \lambda . v_1$, que cumple el enunciado del lema.

Teorema. Todo término de la forma $\text{rotar_dr}(v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot v_1 \cdot \lambda)$ es equivalente por las ecuaciones al término $v_1 \cdot v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda$.

Demostración. Por análisis de casos sobre n .

$n = 0$. El término es $\text{rotar_dr}(\lambda)$, que se transforma en λ aplicando 15, y se cumple el enunciado.

$n > 0$. Sea $t = \text{rotar_dr}(v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot v_1 \cdot \lambda)$. Como $n > 0$, se aplica el lema y t queda igual a $\text{rotar_dr}(v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda \cdot v_1)$. A continuación, se aplica la ecuación 16 con $c = v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda$ y queda $t = v_1 \cdot v_n \cdot v_{n-1} \cdot \dots \cdot v_2 \cdot \lambda$, como se quería demostrar.

Teorema. Todo término de la forma $\text{medio}(v_n \cdot v_{n-1} \cdot \dots \cdot v_1 \cdot \lambda)$ es equivalente por las ecuaciones al término $v_{\lfloor (n+1)/2 \rfloor}$ si $n > 0$, o bien es un error si $n = 0$.

Demostración. Por análisis de casos sobre n .

$n = 0$. El término es de la forma $\text{medio}(\lambda)$, que es igual al error de tipo alf aplicando 19.

$n > 0$. Por inducción sobre n .

$n = 1$. Queda el término $\text{medio}(v_1 \cdot \lambda)$, que es igual a v_1 aplicando 20. Como la expresión $\lfloor (n+1)/2 \rfloor$ vale 1 con $n = 1$, se cumple el enunciado.

$n = k$. Hipótesis de inducción: $\text{medio}(v_k \cdot v_{k-1} \cdot \dots \cdot v_1 \cdot \lambda)$ se transforma en $v_{\lfloor (k+1)/2 \rfloor}$.

$n = k+1$. Debe demostrarse que $t = \text{medio}(v_{k+1} \cdot v_k \cdot \dots \cdot v_1 \cdot \lambda)$ cumple el teorema.

Es necesario primero aplicar el lema sobre el subtérmino $v_{k+1} \cdot v_k \cdot \dots \cdot v_2 \cdot v_1 \cdot \lambda$ (lo que es posible ya que $k+1 > 0$), resultando en $v_{k+1} \cdot v_k \cdot \dots \cdot v_2 \cdot \lambda \cdot v_1$. Si reescribimos t con esta sustitución y con el cambio $w_i = v_{i+1}$, obtenemos el término $t = \text{medio}(w_k \cdot w_{k-1} \cdot \dots \cdot w_1 \cdot \lambda \cdot v_1)$. A continuación, deben distinguirse dos casos:

$k = 1$: queda $t = \text{medio}(w_1 \cdot \lambda \cdot v_1)$, con lo que se aplica la ecuación 21 y se obtiene v_1 , que cumple el enunciado ya que $\lfloor (n+1)/2 \rfloor = \lfloor (k+1+1)/2 \rfloor = 1$.

$k > 1$: se aplica la ecuación 22 sobre el término t mediante la asignación $c = w_{k-1} \cdot \dots \cdot w_1 \cdot \lambda$, cuya longitud $\|c\|$ es mayor que 0 (por lo que se cumple la premisa de la ecuación), resultando en $t = \text{medio}(w_{k-1} \cdot \dots \cdot w_1 \cdot \lambda)$. A continuación, se aplica la hipótesis de inducción y se obtiene $w_{\lfloor k/2 \rfloor}$, que al deshacer el cambio se convierte en $v_{\lfloor k/2 \rfloor + 1} = v_{\lfloor (k+2)/2 \rfloor} = v_{\lfloor (k+1)+1/2 \rfloor}$, y se cumple el enunciado del teorema.

1.5.2 Especificación de una tabla de símbolos

Durante la compilación de un programa es necesario construir una estructura que asocie a cada objeto que en él aparece, identificado con un nombre, un cierto número de características, como su tipo, la dirección física en memoria, etc. Esta estructura se denomina tabla de símbolos (ing., symbol table) y se construye a medida que se examina el texto fuente. Su estudio es interesante tanto desde el punto de vista clásico de la implementación como desde la vertiente ecuacional. Diversos tipos de lenguajes pueden exigir tablas que

presenten características ligeramente diferentes; en este apartado nos referimos a un caso concreto: la especificación de una tabla de símbolos para un lenguaje de bloques.

Los lenguajes de bloques, como Pascal o C, definen ámbitos de existencia para los objetos que corresponden a los diferentes bloques que forman el programa principal y que normalmente se asocian a la idea de subprograma; como los bloques pueden anidarse, el lenguaje ha de definir exactamente las reglas de visibilidad de los identificadores de los objetos, de manera que el compilador sepa resolver posibles ambigüedades durante la traducción del bloque en tratamiento, que denominaremos bloque en curso. Normalmente, las dos reglas principales son: a) no pueden declararse dos objetos con el mismo nombre dentro del mismo bloque y, b) una referencia a un identificador denota el objeto más cercano con este nombre, consultando los bloques de dentro a fuera a partir del bloque en curso. Bajo estas reglas, se propone la siguiente signatura para el tipo `ts` de las tablas de símbolos, donde `cadena` representa el género de las cadenas de caracteres (con la especificación que hemos definido en el punto anterior) y `caracts` representa las características de los objetos:

`crea`: $\rightarrow ts$, crea la tabla vacía, siendo el programa principal el bloque en curso
`entra`: $ts \rightarrow ts$, registra que el compilador entra en un nuevo bloque
`sal`: $ts \rightarrow ts$, registra que el compilador abandona el bloque en curso; el nuevo bloque en curso pasa a ser aquel que lo englobaba
`declara`: $ts\ cadena\ caracts \rightarrow ts$, declara dentro del bloque en curso un objeto identificado por la cadena y con las características dadas
`consulta`: $ts\ cadena \rightarrow caracts$, devuelve las características del objeto identificado por la cadena dada, correspondientes a su declaración dentro del bloque más próximo que englobe el bloque en curso
`declarado?`: $ts\ cadena \rightarrow bool$, indica si la cadena identifica un objeto que ha sido declarado dentro del bloque en curso

En la fig. 1.20 se presenta un programa Pascal y la tabla de símbolos `T` asociada, justo en el momento en que el compilador está en el punto `writeln(c)`; como características se dan la categoría del identificador, su tipo (si es el caso) y su dimensión; la raya gruesa delimita los dos bloques existentes, el programa principal y el bloque correspondiente al procedimiento `Q` en curso. En esta situación, `consulta(T, a)` devuelve la descripción de la variable local `a`, que oculta la variable global `a`, mientras que `consulta(T, b)` devuelve la descripción de la variable global `b`. La variable global `a` volverá a ser visible cuando el compilador salga del bloque `Q`.

Aplicando el método, primero se eligen las operaciones constructoras generadoras. Además de la operación `crea`, está claro que `declara` también es constructora generadora, ya que es la única operación que permite declarar identificadores dentro de la tabla. Además, la tabla de símbolos necesita incorporar la noción de bloque y, por ello, es necesario tomar como constructora generadora la operación `entra`. Con este conjunto mínimo de tres operaciones se puede generar cualquier tabla de símbolos, eso sí, hay dos interrelaciones claras que se


```

program P;
  var a, b: integer;
  procedure Q (c: integer);
    var a, d: real;
  begin
    writeln(c)
  end;
begin
  Q(a)
end.

```



P	program		10230
a	var	int	1024
b	var	int	1026
Q	proc		10500
c	param	int	1028
a	var	real	1030
d	var	real	1032

Fig. 1.20: un programa Pascal y su tabla de símbolos asociada.

deben explicitar: dentro de un mismo bloque, es un error repetir la declaración de un identificador y, además, no importa el orden de la declaración de identificadores:

- 1) error [declarado?(t, id)] \Rightarrow declara(t, id, c)
- 2) declara(declara(t, id₁, c₁), id₂, c₂) = declara(declara(t, id₂, c₂), id₁, c₁)

La operación sal ha de eliminar todas las declaraciones realizadas en el bloque en curso, y por eso se obtienen tres ecuaciones aplicando directamente el método:

- 3) error sal(crea)
- 4) sal(entra(t)) = t
- 5) sal(declara(t, id, c)) = sal(t)

Se ha considerado un error intentar salir del programa principal. Notemos que 4 realmente se corresponde con la idea de salida del bloque, mientras que 5 elimina todos los posibles identificadores declarados en el bloque en curso.

Por lo que a consulta se refiere, el esquema es idéntico; la búsqueda aprovecha que la estructura en bloques del programa se refleja dentro del término, de manera que se para al encontrar la primera declaración:

- 6) error consulta(crea, id)
- 7) consulta(entra(t), id) = consulta(t, id)
- 8) consulta(declara(t, id, c), id) = c
- 9) $[\neg (id_1 = id_2)] \Rightarrow$ consulta(declara(t, id₁, c₁), id₂) = consulta(t, id₂)

Por último, declarado? no presenta más dificultades:

- 10) declarado?(crea, id) = falso
- 11) declarado?(entra(t), id) = falso
- 12) declarado?(declara(t, id₁, c₁), id₂) = (id₁ = id₂) \vee declarado?(t, id₂)

1.5.3 Especificación de un sistema de reservas de vuelos

Se quiere especificar el sistema de reservas de vuelos de la compañía "Averia". Se propone una signatura que permita añadir y cancelar vuelos dentro de la oferta de la compañía, realizar reservas de asientos y consultar información diversa; en concreto, las operaciones son:

crea: \rightarrow averia, devuelve un sistema de reservas sin información
 añade: averia vuelo \rightarrow averia, añade un nuevo vuelo al sistema de reservas
 reserva: averia vuelo pasajero \rightarrow averia, registra una reserva hecha por un pasajero dentro de un vuelo determinado; no puede haber más de una reserva a nombre de un pasajero en un mismo vuelo
 cancela: averia vuelo \rightarrow averia, anula un vuelo del sistema de reservas; todas las reservas de pasajeros para este vuelo, si las hay, han de ser eliminadas
 asiento?: averia vuelo pasajero \rightarrow nat, determina el número de asiento asignado a un pasajero dentro de un vuelo; se supone que esta asignación se realiza por orden de reserva, y que los asientos se numeran en orden ascendente a partir del uno; devuelve 0 si el pasajero no tiene ninguna reserva
 lista: averia \rightarrow cadena_vuelos, obtiene todos los vuelos en orden de hora de salida
 a_suspender: averia aeropuerto \rightarrow cadena_vuelos, dado un aeropuerto, proporciona la lista de todos los vuelos que se dirigen a este aeropuerto

Supondremos que los tipos vuelo, aeropuerto, hora, pasajero y nat, especificados en los universos VUELO, AEROPUERTO, HORA, PASAJERO y NAT, respectivamente, disponen de operaciones de comparación ig; además, el universo VUELO presenta las operaciones:

origen?, destino?: vuelo \rightarrow aeropuerto, da los aeropuertos del vuelo
 hora_salida?: vuelo \rightarrow hora, da la hora de salida del vuelo
 capacidad?: vuelo \rightarrow nat, da el número máximo de reservas que admite el vuelo

Las cadenas de vuelos tendrán las operaciones sobre cadenas vistas en el punto 1.5.1.

El sistema de vuelos necesita información tanto sobre vuelos como sobre reservas; por ello, son constructoras generadoras las operaciones añade y reserva, y también la típica crea. Se debe ser especialmente cuidadoso al establecer los errores que presentan estas operaciones: añadir un vuelo repetido, reservar dentro un vuelo que no existe, reservar para un pasajero más de un asiento en el mismo vuelo y reservar en un vuelo lleno. Hay también varias relaciones conmutativas:

- 1) error [existe?(s, v)] \Rightarrow añade(s, v)
- 2) error [\neg existe?(s, v) \vee lleno?(s, v) \vee \neg ig(asiento(s, v, p), 0)] \Rightarrow reserva(s, v, p)
- 3) añade(añade(s, v₁), v₂) = añade(añade(s, v₂), v₁)
- 4) [\neg VUELO.ig(v₁, v₂)] \Rightarrow reserva(reserva(s, v₁, p₁), v₂, p₂) =
reserva(reserva(s, v₂, p₂), v₁, p₁)
- 5) [\neg VUELO.ig(v₁, v₂)] \Rightarrow añade(reserva(s, v₁, p), v₂) = reserva(añade(s, v₂), v₁, p)

Para expresar estas relaciones hemos introducido dos operaciones auxiliares:

privada lleno?: averia vuelo \rightarrow bool, comprueba si el vuelo no admite más reservas

6) $\text{lleno?}(s, v) = \text{NAT.ig}(\text{cuenta}(s, v), \text{VUELO.capacidad}(v))$

privada existe?: averia vuelo \rightarrow bool, comprueba si el vuelo ya existía

7) $\text{existe?}(\text{crea}, v) = \text{falso}$

8) $\text{existe?}(\text{añade}(s, v_1), v_2) = \text{VUELO.ig}(v_1, v_2) \vee \text{existe?}(s, v_2)$

9) $\text{existe?}(\text{reserva}(s, v_1, p), v_2) = \text{existe?}(s, v_2)$

La operación cuenta: averia vuelo \rightarrow nat da el número de reservas realizadas en el vuelo y se especifica más adelante, así como el resto de operaciones.

a) cancela

Además del error de cancelar en un vuelo no existente, el comportamiento de cancela depende de si el vuelo que se anula es el mismo que se está examinando o no; en el primer caso, desaparece la información sobre el vuelo, y en el segundo caso debe conservarse:

10) error cancela(crea, v)

11) $\text{cancela}(\text{añade}(s, v), v) = s$

12) $[\neg \text{VUELO.ig}(v_1, v_2)] \Rightarrow \text{cancela}(\text{añade}(s, v_1), v_2) = \text{añade}(\text{cancela}(s, v_2), v_1)$

13) $\text{cancela}(\text{reserva}(s, v, p), v) = \text{cancela}(s, v)$

14) $[\neg \text{VUELO.ig}(v_1, v_2)] \Rightarrow \text{cancela}(\text{reserva}(s, v_1, p), v_2) = \text{reserva}(\text{cancela}(s, v_2), v_1, p)$

Notemos que la segunda ecuación no precisa controlar posibles repeticiones del vuelo dentro del término, porque en este caso habrían actuado las ecuaciones de error.

b) asiento?

Para especificar esta operación, notemos que el orden de numeración de los asientos es inverso al orden de exploración del término; lo que debe hacerse, pues, es saltar todas las reservas realizadas posteriormente y contar las reservas del mismo vuelo que todavía quedan.

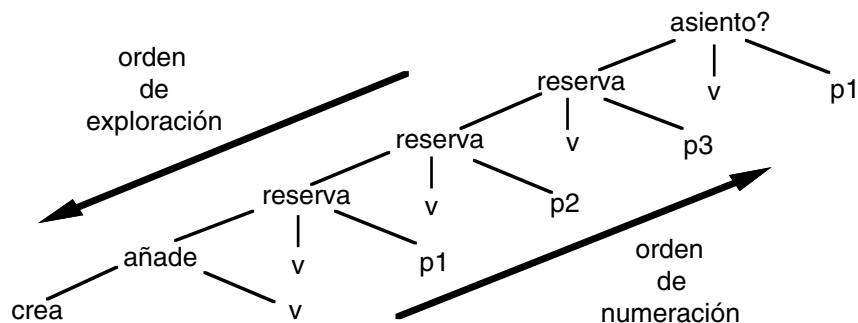


Fig. 1.21: relación entre la numeración y la exploración de las reservas.

Es decir, la operación ha de cambiar de comportamiento al encontrar la reserva; por ello, usamos la operación privada `cuenta` ya introducida, y comparamos si los vuelos y los pasajeros que se van encontrando en el término son los utilizados en la operación (controlando su existencia):

- 15) `error asiento?(crea, v, p)`
- 16) $[VUELO.ig(v_1, v_2)] \Rightarrow asiento?(añade(s, v_1), v_2, p) = \text{cero}$
- 17) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow asiento?(añade(s, v_1), v_2, p) = asiento?(s, v_2, p)$
- 18) $asiento?(reserva(s, v, p), v, p) = suc(cuenta(s, v))$
- 19) $[\neg(VUELO.ig(v_1, v_2) \wedge PASAJERO.ig(p_1, p_2))] \Rightarrow$
 $asiento?(reserva(s, v_1, p_1), v_2, p_2) = asiento?(s, v_2, p_2)$

La especificación de `cuenta` es realmente sencilla: se incrementa en uno el resultado de la función siempre que se encuentre una reserva en el vuelo correspondiente:

- 20) `error cuenta?(crea, v)`
- 21) $cuenta(añade(s, v), v) = \text{cero}$
- 22) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow cuenta(añade(s, v_1), v_2) = cuenta(s, v_2)$
- 23) $cuenta(reserva(s, v, p), v) = suc(cuenta(s, v))$
- 24) $[\neg VUELO.ig(v_1, v_2)] \Rightarrow cuenta(reserva(s, v_1, p), v_2) = cuenta(s, v_2)$

c) lista

La resolución presenta diversas alternativas; por ejemplo, se puede considerar la existencia de una operación de ordenación sobre las cadenas, `ordena` (que queda como ejercicio para el lector), y entonces escribir la especificación de `lista` como:

`lista(s) = ordena(enumera(s))`

donde `enumera` es una operación auxiliar del sistema de vuelos que los enumera todos:

- `privada enumera: averia → cadena_vuelos`
- 25) $enumera(crea) = \lambda$
 - 26) $enumera(reserva(s, v, p)) = enumera(s)$
 - 27) $enumera(añade(s, v)) = v . enumera(s)$

d) a_suspender

No hay ninguna situación especial; si el sistema de reservas está vacío, devuelve la cadena vacía, y el resultado sólo se modifica al encontrar un vuelo con el mismo aeropuerto destino:

- 28) $a_suspender(crea, a) = \lambda$
- 29) $[AEROPUERTO.ig(a, VUELO.destino?(v))] \Rightarrow$
 $a_suspender(añade(s, v), a) = v . a_suspender(s, a)$
- 30) $[\neg AEROPUERTO.ig(a, VUELO.destino?(v))] \Rightarrow$
 $a_suspender(añade(s, v), a) = a_suspender(s, a)$
- 31) $a_suspender(reserva(s, v, p), a) = a_suspender(s, a)$

1.6 Estructuración de especificaciones

Hasta ahora, nos hemos centrado en la construcción de especificaciones simples, universos dentro los cuales se definen todos los géneros que forman parte de la especificación; es obvio que se necesitan mecanismos que permitan estructurar estos universos para poder especificar aplicaciones enteras como la composición de especificaciones más simples.

Hay varios lenguajes de especificación que definen mecanismos de estructuración. Las primeras ideas fueron formuladas por R.M. Burstall y J.A. Goguen en "Putting Theories together to make Specifications" (Proceedings of 5th International Joint Conference on Artificial Intelligence, Cambridge M.A., 1977), y dieron como resultado el lenguaje CLEAR por ellos desarrollado. Hay otros lenguajes igualmente conocidos: OBJ (inicialmente definido por el mismo J.A. Goguen), ACT ONE (desarrollado en la Universidad de Berlín y descrito en [EhM85]), etc. En Merlí se incorporan las características más interesantes de estos lenguajes, si bien aquí sólo se definen las construcciones que son estrictamente necesarias para especificar los TAD que en él aparecen. Debe destacarse la ausencia de construcciones relativas a la orientación a objetos (principalmente, un mecanismo para declarar subtipos) porque su inclusión aumentaría la complejidad de este texto y no es imprescindible para el desarrollo de los temas.

1.6.1 Uso de especificaciones

Mecanismo necesario para usar desde una especificación los géneros, las operaciones y las ecuaciones escritos en otra; se puede considerar una simple copia literal. Si la especificación $SPEC_1 = (S_1, OP_1, E_1)$ usa la especificación $SPEC_2 = (S_2, OP_2, E_2)$, se puede decir de forma equivalente que $SPEC_1 = (S_1 \cup S_2, OP_1 \cup OP_2, E_1 \cup E_2)$ (considerando la unión de un A-conjunto y un B-conjunto como un $A \cup B$ -conjunto). La relación de uso es transitiva por defecto: si A usa B y B usa C, A está usando C implícitamente; ahora bien, esta dependencia se puede establecer explícitamente para favorecer la legibilidad y la modificabilidad. Alternativamente, la palabra "privado" puede preceder al nombre del universo usado, con lo que se evita la transitividad (dicho de otra manera, el uso introduce los símbolos de $SPEC_2$ como privados de $SPEC_1$, el cual no los exporta).

En el apartado 1.3.1 ya se introdujo este mecanismo como una herramienta para la definición de nuevos TAD que necesiten operaciones y géneros ya existentes; por ejemplo, para definir el TAD de los naturales con igualdad se usa el TAD de los booleanos, que define el género `bool` y diversas operaciones. Normalmente, un universo declara un único tipo nuevo, denominado tipo de interés.

Otra situación común es el enriquecimiento (ing., *enrichment*) de uno o varios tipos añadiendo nuevas operaciones. Un escenario habitual es la existencia de una biblioteca de

universos donde se definen algunos TAD de interés general que, como serán manipulados por diferentes usuarios con diferentes propósitos, incluyen el máximo número de operaciones posible que no estén orientadas hacia ninguna utilización particular. En una biblioteca de este estilo se puede incluir, por ejemplo, un TAD para los conjuntos con operaciones habituales (v. fig. 1.22, izq.). En una aplicación particular, no obstante, puede ser necesario definir nuevas operaciones sobre el tipo; por ejemplo, una que elimine del conjunto todos los elementos mayores que cierta cota. Es suficiente con escribir un nuevo universo que, además de las operaciones sobre conjuntos de la biblioteca, también ofrezca esta operación (v. fig. 1.22, derecha); cualquier universo que lo use, implícitamente está utilizando CJT_NAT por la transitividad de los usos.

<u>universo</u> CJT_NAT <u>es</u>	<u>universo</u> MI_CJT_NAT <u>es</u>
<u>usa</u> NAT, BOOL	<u>usa</u> CJT_NAT, NAT, BOOL
<u>tipo</u> cjt_nat	<u>ops</u>
<u>ops</u> \emptyset : \rightarrow cjt_nat	<u>filtra</u> : cjt_nat nat \rightarrow cjt_nat
<u>{_}</u> : nat \rightarrow cjt_nat	<u>ecns</u> ...
<u>elige</u> : cjt_nat \rightarrow nat	<u>funiverso</u>
<u>_</u> \cup <u>_</u> , <u>_</u> \cap <u>_</u> , ...: cjt_nat cjt_nat \rightarrow cjt_nat	
<u>ecns</u> ...	
<u>funiverso</u>	

Fig. 1.22: especificación de los conjuntos de naturales (izq.) y un enriquecimiento (der.).

1.6.2 Ocultación de símbolos

Así como el mecanismo de uso se caracteriza por hacer visibles los símbolos residentes en una especificación, es útil disponer de una construcción complementaria que se ocupe de ocultar los símbolos de una especificación a todas aquellas que la usen. De esta manera, los especificadores pueden controlar el ámbito de existencia de los diferentes símbolos de un universo en las diversas partes de una aplicación, y evitar así usos indiscriminados.

Hay varias situaciones en las que la ocultación de símbolos es útil; una de ellas es la redefinición de símbolos, que consiste en cambiar el comportamiento de una operación. En la fig. 1.23, se redefine la operación elige introducida en el universo CJT_NAT de la fig. 1.22 por otra del mismo nombre que escoge el elemento más grande del conjunto; para ello, se oculta la operación usada y, a continuación, se especifica la operación como si fuera nueva. Cualquier uso del universo MI_CJT_NAT contempla la nueva definición de elige.

Otra situación consiste en restringir el conjunto de operaciones válidas sobre un TAD. En el capítulo 3 se especifican de forma independiente los tipos de las pilas y las colas, ambos variantes de las secuencias vistas en la sección anterior. Una alternativa, que queda como

ejercicio para el lector, consiste en especificar ambos tipos a partir de un TAD general para las secuencias, restringiendo las operaciones que sobre ellos se pueden aplicar: inserciones, supresiones y consultas siempre por el mismo extremo en las pilas, e inserciones por un extremo y supresiones y consultas por el otro, en las colas.

```

universo MI_CJT_NAT es
  usa CJT_NAT, NAT, BOOL
  esconde elige
  ops elige: cjt_nat → nat
  error elige(∅)
  ecns elige({n}) = n; ...
funiverso

```

Fig. 1.23: redefinición de la operación elige de los conjuntos.

Por lo que respecta al modelo, los símbolos ocultados se pueden considerar parte de la signatura privada usada para restringir el álgebra cociente de términos según se explica en el apartado 1.4.2.

1.6.3 Renombramiento de símbolos

El renombramiento (ing., renaming) de símbolos consiste en cambiar el nombre de algunos géneros y operaciones de un universo sin modificar su semántica inicial (porque el álgebra inicial de una especificación es una clase de álgebras isomorfas, insensible a los cambios de nombre). Se utiliza para obtener universos más legibles y para evitar conflictos de tipo al instanciar universos genéricos (v. apartado siguiente).

Por ejemplo, supongamos la existencia de un universo que especifique las tuplas de dos enteros (v. fig. 1.24). Si en una nueva aplicación se necesita introducir el TAD de los racionales, hay dos opciones: por un lado, definir todo el tipo partiendo de la nada; por el otro, aprovechar la existencia del universo DOS_ENTEROS considerando los racionales como parejas de enteros numerador-denominador. En la fig. 1.25 (izq.) se muestra la segunda opción: primero, se renombra el género y las operaciones de las parejas de enteros dentro del universo RAC_DEF y, seguidamente, se introducen las operaciones del álgebra de los racionales que se crean necesarias, con las ecuaciones correspondientes, dentro del universo RACIONALES. El proceso se puede repetir en otros contextos; así, en la fig. 1.25 (derecha) se muestra la definición de las coordenadas de un espacio bidimensional donde se renombra el género, pero no las operaciones, de las parejas de enteros, por lo que las operaciones de construcción y consulta de coordenadas tienen el mismo nombre que las operaciones correspondientes de las parejas de enteros.

universo DOS_ENTEROS es
usa ENTERO
tipo 2enteros
ops $<_, _>$: entero entero \rightarrow 2enteros
 $_.c_1, _.c_2$: 2enteros \rightarrow entero
ecns $\forall z_1, z_2 \in \text{entero}$
 $<z_1, z_2>.c_1 = z_1; <z_1, z_2>.c_2 = z_2$
funiverso

Fig. 1.24: especificación para las tuplas de dos enteros.

<u>universo</u> RAC_DEF es <u>usa</u> DOS_ENTEROS <u>renombra</u> 2enteros <u>por</u> rac $<_, _>$ <u>por</u> $_/_, _.c_1$ <u>por</u> num, $_.c_2$ <u>por</u> den <u>funiverso</u>	<u>universo</u> COORD_DEF es <u>usa</u> DOS_ENTEROS <u>renombra</u> 2enteros <u>por</u> coord <u>funiverso</u>
<u>universo</u> RACIONALES es <u>usa</u> RAC_DEF <u>ops</u> $(_)^{-1}, -_$: rac \rightarrow rac $+_-, _-_, _*_$: rac rac \rightarrow rac <u>ecns</u> ... <u>funiverso</u>	<u>universo</u> COORDENADAS es <u>usa</u> COORD_DEF <u>ops</u> dist: coord coord \rightarrow coord <u>ecns</u> ... <u>funiverso</u>

Fig. 1.25: especificación de los racionales (izq.) y de las coordenadas (derecha).

1.6.4 Parametrización e instanciación

La parametrización permite formular una descripción genérica, que puede dar lugar a diversas especificaciones concretas mediante las asociaciones de símbolos adecuadas. Por ejemplo, sean dos especificaciones para los conjuntos de naturales y de enteros (v. fig. 1.26); se puede observar que los universos resultantes son prácticamente idénticos, excepto algunos nombres (nat por entero, cjt_nat por cjt_ent), por lo que el comportamiento de las operaciones de los conjuntos es independiente del tipo de sus elementos. Por esto, en vez de especificar unos conjuntos concretos es preferible especificar los conjuntos de cualquier tipo de elementos (v. fig. 1.27). Esta especificación se llama especificación parametrizada o genérica (ing., *parameterised* o *generic specification*) de los conjuntos; los símbolos que condicionan el comportamiento del universo genérico se llaman parámetros formales (ing., *formal parameter*) y se definen en universos de caracterización, cuyo nombre aparece en la cabecera del universo genérico. En el caso de los conjuntos, el parámetro formal es el género elem, que está definido dentro del universo de caracterización ELEM.

<u>universo CJT_NAT es</u>	<u>universo CJT_ENT es</u>
<u>usa</u> NAT	<u>usa</u> ENTER
<u>tipo</u> cjt_nat	<u>tipo</u> cjt_ent
<u>ops</u> $\emptyset: \rightarrow \text{cjt_nat}$	<u>ops</u> $\emptyset: \rightarrow \text{cjt_ent}$
<u>_u{ }_</u> : cjt_nat nat \rightarrow cjt_nat	<u>_u{ }_</u> : cjt_ent enter \rightarrow cjt_ent
<u>ecns</u> $\forall s \in \text{cjt_nat}; \forall n, m \in \text{nat}$	<u>ecns</u> $\forall s \in \text{cjt_ent}; \forall n, m \in \text{enter}$
$s \cup \{n\} \cup \{n\} = s \cup \{n\}$	$s \cup \{n\} \cup \{n\} = s \cup \{n\}$
$s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$	$s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$
<u>funiverso</u>	<u>funiverso</u>

Fig. 1.26: especificación de los conjuntos de naturales y de enteros.

<u>universo CJT (ELEM) es</u>	<u>universo ELEM caracteriza</u>
<u>tipo</u> cjt	<u>tipo</u> elem
<u>ops</u> $\emptyset: \rightarrow \text{cjt}$	<u>funiverso</u>
<u>_u{ }_</u> : cjt elem \rightarrow cjt	
<u>ecns</u> $\forall s \in \text{cjt}; \forall n, m \in \text{elem}$	
$s \cup \{n\} \cup \{n\} = s \cup \{n\}; s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$	
<u>funiverso</u>	

Fig. 1.27: especificación de los conjuntos genéricos (izq.) y caracterización de los elementos (der.).

Para crear un universo para los conjuntos de naturales debe efectuarse lo que se denomina una instancia (ing., instantiation o actualisation) del universo genérico (v. fig. 1.28), que consiste en asociar unos parámetros reales (ing., actual parameter) a los formales; por este motivo, la instancia también se denomina paso de parámetros (ing., parameter passing). La cláusula "instancia" indica que se crea un nuevo tipo a partir de la descripción genérica dada en CJT(ELEM) a través de la asociación de nat a elem. Además, se renombra el símbolo correspondiente al género que se había definido, porque, de lo contrario, toda instancia de CJT(ELEM) definiría un género con el mismo nombre; opcionalmente, y para mejorar la legibilidad, se podría renombrar algún otro símbolo. El resultado de la instancia es idéntico a la especificación de los conjuntos de naturales de la fig. 1.26; nótese, sin embargo, la potencia del mecanismo de parametrización que permite establecer el comportamiento de más de un TAD en un único universo.

universo CJT_NAT es
usa NAT
instancia CJT(ELEM) donde elem es nat
renombra cjt por cjt_nat
funiverso

Fig. 1.28: instancia de los conjuntos genéricos para obtener conjuntos de naturales.

Este caso de genericidad es el más simple posible, porque el parámetro es un símbolo que no ha de cumplir ninguna condición; de hecho, se podría haber simulado mediante renombramientos adecuados. No obstante, los parámetros formales pueden ser símbolos de operación a los que se exija cumplir ciertas propiedades. Por ejemplo, consideremos los conjuntos con operación de pertenencia de la fig. 1.29; en este caso se introduce un nuevo parámetro formal, la operación de igualdad $_=_$ sobre los elementos de género elem (no confundir con el símbolo '=' de las ecuaciones). Es necesario, pues, construir un nuevo universo de caracterización, $ELEM_=_$, que defina el género de los elementos con operación de igualdad. Para mayor comodidad en posteriores usos del universo, se requiere también la operación $_ \neq _$ definida como la negación de la igualdad; para abreviar, y dada la ecuación que define totalmente su comportamiento, consideraremos que $_ \neq _$ se establecerá automáticamente a partir de $_ = _$ en toda instancia de $ELEM_=_$. Notemos que las ecuaciones de $_ = _$ no definen su comportamiento preciso, sino que establecen las propiedades que cualquier parámetro real asociado ha de cumplir (reflexividad, simetría y transitividad). Al efectuar una instancia de $CJT_ \in$, sólo se puede tomar como tipo base de los conjuntos un género que tenga una operación con la signatura correcta y que cumpla las propiedades citadas; por ejemplo, la definición de los enteros de la fig. 1.14 no permite crear conjuntos de enteros, porque no hay ninguna operación que pueda desempeñar el papel de la igualdad, mientras que sí que lo permiten los naturales de la fig. 1.10 (v. fig. 1.30). Destaquemos por último que la instancia puede asociar expresiones (que representan operaciones anónimas) a los parámetros formales que sean operaciones, siempre y cuando cada expresión cumpla las ecuaciones correspondientes.

<u>universo</u> $CJT_ \in$ ($ELEM_ =$) es	<u>universo</u> $ELEM_ =$ caracteriza
<u>usa</u> BOOL	<u>usa</u> BOOL
<u>tipo</u> cjt	<u>tipo</u> elem
<u>ops</u> $\emptyset: \rightarrow cjt$	<u>ops</u> $_ = _, _ \neq _: elem\ elem \rightarrow bool$
$_ \cup \{ _ \}: cjt\ elem \rightarrow cjt$	<u>ecns</u> $\forall v, v_1, v_2, v_3 \in elem$
$_ \in _ : elem\ cjt \rightarrow bool$	$(v = v) = \text{cierto}$
<u>ecns</u> $\forall s \in cjt; \forall n, m \in elem$	$[v_1 = v_2] \Rightarrow (v_2 = v_1) = \text{cierto}$
$s \cup \{n\} \cup \{n\} = s \cup \{n\}$	$[(v_1 = v_2) \wedge (v_2 = v_3)] \Rightarrow (v_1 = v_3) = \text{cierto}$
$s \cup \{n\} \cup \{m\} = s \cup \{m\} \cup \{n\}$	$(v_1 \neq v_2) = \neg (v_1 = v_2)$
$n \in \emptyset = \text{falso}$	<u>funiverso</u>
$n \in s \cup \{m\} = (n = m) \vee (n \in s)$	
<u>funiverso</u>	

Fig. 1.29: especificación de los conjuntos genéricos con operación de pertenencia (izquierda) y caracterización de sus elementos con la operación de igualdad (derecha).

Destaquemos que tanto $CJT_ \in$ como $ELEM_ =$ se podrían haber creado como enriquecimientos de CJT y $ELEM$, respectivamente, con la cláusula "usa". Precisamente, el mecanismo de uso se ve afectado por la existencia de los universos de caracterización:

```

universo CJT_∈_NAT es
  usa NAT
  instancia CJT_∈ (ELEM_=) donde elem es nat, = es NAT.ig
  renombra cjt por cjt_nat
funiverso

```

Fig. 1.30: instancia correcta de los conjuntos genéricos con operación de pertenencia.

- Si un universo A de caracterización usa otro B de definición, todos los símbolos de B están incluidos en A, pero no son parámetros. Es el caso de ELEM_= usando BOOL.
- Si un universo A de caracterización usa otro B, también de caracterización, todos los símbolos de B están incluidos en A y además se consideran parámetros. Es el caso de ELEM_= definido como enriquecimiento de ELEM.
- Un universo de definición no puede usar uno de caracterización.

La sintaxis introducida puede producir ambigüedades en algunos contextos. Por ejemplo, supongamos que queremos especificar los pares de elementos cualesquiera dentro de un universo parametrizado y preguntémonos cómo se pueden caracterizar los dos parámetros formales resultantes (los tipos de los componentes de los pares). Una primera opción es definir un universo de caracterización DOS_ELEMS que simplemente incluya los dos géneros; otra posibilidad consiste en aprovechar la existencia del universo ELEM y repetirlo dos veces en la cabecera, indicando que se necesitan dos apariciones de los parámetros que en él residen. Ahora bien, en este caso los dos parámetros formales son indistinguibles: ¿cómo referenciar el tipo de los primeros elementos si tiene el mismo nombre que el de los segundos? Por ello, al nombre del universo de caracterización le debe preceder un identificador, que después se puede usar para distinguir los símbolos, tal como se muestra en la fig. 1.31; notemos que la instancia para definir los pares de enteros da como resultado el mismo TAD especificado directamente en la fig. 1.24. En general, siempre se puede preceder el nombre de un universo de caracterización de otro identificador, si se considera que la especificación resultante queda más legible.

<pre> <u>universo</u> PAR (A, B <u>son</u> ELEM) <u>es</u> <u>tipo</u> par <u>ops</u> <_ , _>: A.elem B.elem → par _ .c₁: par → A.elem _ .c₂: par → B.elem <u>ecns</u> <v₁, v₂> .c₁ = v₁; <v₁, v₂> .c₂ = v₂ <u>funiverso</u> </pre>	<pre> <u>universo</u> DOS_ENTEROS <u>es</u> <u>usa</u> ENTERO <u>instancia</u> PAR (A, B <u>son</u> ELEM) <u>donde</u> A.elem <u>es</u> entero, B.elem <u>es</u> entero <u>renombra</u> par <u>por</u> 2enteros <u>funiverso</u> </pre>
---	---

Fig. 1.31: especificación de los pares de elementos (izq.) y una posible instancia (derecha).

A veces, una instancia sólo se necesita localmente en el universo que la efectúa; en este caso, la palabra clave "instancia" va seguida del calificativo "privada", indicando que los símbolos definidos en el universo parametrizado no son exportables tras la instancia.

Estudiamos a continuación el concepto de instancia parcial que generaliza el mecanismo de genericidad presentado hasta ahora. Lo hacemos con un ejemplo de futura utilidad: los conjuntos estudiados en esta sección son conjuntos infinitos, sin ninguna restricción sobre su capacidad; si posteriormente se implementan usando un vector dimensionado con un máximo, y no tratamos ecuacionalmente esta limitación sobre el modelo, obtenemos una implementación que contradice la especificación. Para explicitar los requerimientos de espacio, se incorpora al universo un nuevo parámetro formal que dé el número máximo de elementos que puede tener un conjunto, definiéndose dentro de un nuevo universo de caracterización, VAL_NAT. El resultado aparece en la fig. 1.32; destaca la operación cuántos para contar el número de elementos del conjunto, que ha de controlar siempre la aparición de repetidos. Se define una nueva operación visible, lleno?, para que exista un medio para saber si un conjunto está lleno sin tener que provocar el error correspondiente.

```

universo CJT_∈_ACOTADO (ELEM_∈, VAL_NAT) es
  usa NAT, BOOL
  tipo cjt
  ops Ø: → cjt
    _∪_: cjt elem → cjt
    _∈_: elem cjt → bool
    lleno?: cjt → bool
    privada cuántos: cjt → nat
  error ∀s∈cjt; ∀n∈elem: [lleno?(s) ∧ ¬ n∈s] ⇒ s∪{n}
  ecns ∀s∈cjt; ∀n,m∈elem_cjt
    s∪{n}∪{n} = s∪{n}; s∪{n}∪{m} = s∪{m}∪{n}
    n∈Ø = falso; n∈s∪{m} = (m = n) ∨ (n ∈ s)
    lleno?(s) = NAT.ig(cuántos(s), val)
    cuántos(Ø) = cero
    [n∈s] ⇒ cuántos(s∪{n}) = cuántos(s)
    [¬ n∈s] ⇒ cuántos(s∪{n}) = suc(cuántos(s))
funiverso
universo VAL_NAT caracteriza
  usa NAT, BOOL
  ops val: → nat
  ecns val > cero = cierto
funiverso

```

Fig. 1.32: especificación de los conjuntos acotados (arriba) y del parámetro formal val (abajo).

Ahora pueden realizarse instancias parciales. Por ejemplo, se puede definir un universo para los conjuntos de naturales sin determinar todavía su capacidad máxima, tal como se muestra en la fig. 1.33; el resultado es otro universo genérico que, a su vez, puede instanciarse con un valor concreto de máximo para obtener finalmente un conjunto concreto.

```

universo CJT_∈_ACOTADO (ELEM_∈, VAL_NAT) es ...
universo CJT_NAT_∈_ACOTADO (VAL_NAT) es
  usa NAT
  instancia CJT_∈_ACOTADO (ELEM_∈, VAL_NAT) donde
    elem es nat, = es NAT.ig
  renombra cjt por cjt_nat
funiverso
universo CJT_50_NAT_∈ es
  usa NAT
  instancia CJT_NAT_∈_ACOTADO (VAL_NAT) donde
    val es suc(...suc(cero)...) {50 veces suc}
  renombra cjt_nat por cjt_50_nat
funiverso

```

Fig. 1.33: cabecera para los conjuntos acotados (arriba), instancia parcial para los conjuntos acotados de naturales (en el medio) e instancia determinada a partir de la misma (abajo).

Por último, comentamos el impacto de la parametrización en el significado de una especificación. El modelo de los universos genéricos deja de ser una clase de álgebras para ser una función entre clases de álgebras; concretamente, un morfismo que asocia a las clases de álgebras correspondientes a la especificación de los parámetros formales (es decir, todas aquellas álgebras que cumplen las propiedades establecidas sobre los parámetros formales) las clases de álgebras correspondientes a todos los resultados posibles de una instancia. Así, el paso de parámetros es fundamentalmente un morfismo que tiene como dominio la especificación correspondiente a los parámetros formales y como codominio la especificación correspondiente a los parámetros reales; este morfismo induce otro, que va de la especificación parametrizada a la especificación resultante de la instancia. La principal condición de corrección de la instancia es la protección de los parámetros reales, es decir, que su semántica no se vea afectada en el universo resultado de la instancia.

Técnicamente hablando, el modelo de una especificación parametrizada no es una correspondencia entre clases de álgebras sino entre categorías (ing., category) de álgebras; la diferencia consiste en que una categoría incluye no sólo álgebras sino también morfismos entre ellas (y alguna cosa más). El modelo se convierte en lo que se denomina un

functor (ing., functor), que asocia clases a clases y morfismos a morfismos. Los conceptos teóricos dentro del campo de las categorías y los funtores son realmente complejos y quedan fuera del ámbito de este libro; en [EhM85, caps. 7 y 8] se puede encontrar una recopilación de la teoría de categorías aplicada al campo de la especificación algebraica, así como la construcción del álgebra cociente de términos para especificaciones parametrizadas, que se usa también para caracterizar el modelo.

1.6.5 Combinación de los mecanismos

En los apartados anteriores ha surgido la necesidad de combinar dentro de un mismo universo el mecanismo de instanciación y el de renombramiento. Se puede generalizar esta situación y permitir combinar dentro de un universo todos los mecanismos vistos, en el orden y el número que sea conveniente para la construcción de un universo dado. Por ejemplo, en el apartado 1.6.3 se han definido los racionales en dos pasos: primero se ha escrito RAC_DEF como renombramiento de DOS_ENTEROS y después se ha enriquecido aquél para formar el universo definitivo, RACIONALES. Esta solución presenta el inconveniente de la existencia del universo RAC_DEF, que no sirve para nada. En la fig. 1.34 se muestra una alternativa que combina el renombramiento y la definición de las nuevas operaciones directamente sobre RACIONALES, sin ningún universo intermedio.

```

universo RACIONALES es
  usa DOS_ENTEROS
  renombra 2enteros por racional, <_ , _> por _/_
    _.c1 por num, _.c2 por den
  ops (_)-1, -_: racional → racional
    _+_, _-_, _*__: racional racional → racional
  ecns ...
funiverso

```

Fig. 1.34: definición de los racionales a partir de las tuplas de enteros en un único universo.

Más compleja es la siguiente situación: el mecanismo de parametrización ha de poder usarse imbricadamente, de manera que un parámetro formal dentro un universo pueda ser, a su vez, parámetro real de otro universo. Así, supongamos una especificación parametrizada para las cadenas de elementos reducida a las operaciones λ y $_.$ (v. el apartado 1.5.1) y una nueva, menores, que dada una cadena y un elemento devuelve un conjunto de todos los elementos de la cadena más pequeños que el elemento dado. El universo genérico resultante (v. fig. 1.35) tiene dos parámetros formales, el género *elem* y una operación de orden, $_<_$, que compara dos elementos y dice cuál es menor; los dos parámetros se

encapsulan en el universo de caracterización ELEM_< (v. fig. 1.36). Dado que la operación menores devuelve un conjunto de elementos del mismo tipo de los de la cadena, será necesario efectuar una instancia de una especificación para los conjuntos (por ejemplo, elegimos la de la fig. 1.29); los parámetros reales de esta instancia son los parámetros formales de las cadenas, para garantizar que los elementos de ambos TAD sean los mismos.

```

universo CADENA (A es ELEM_<) es
  tipo cad
  instancia CJT(B es ELEM) donde B.elem es A.elem
  renombra cjt por cjt_elem
  ops  $\lambda$ :  $\rightarrow$  cad
   $\_.$  : A.elem cad  $\rightarrow$  cad
  menores: cad A.elem  $\rightarrow$  cjt_elem
  ecns  $\forall c \in \text{cad}; \forall n, m \in \text{A.elem}$ 
    menores( $\lambda$ , n) =  $\emptyset$ 
     $[m < n] \Rightarrow \text{menores}(m.c, n) = \text{menores}(c, n) \cup \{m\}$ 
     $[\neg m < n] \Rightarrow \text{menores}(m.c, n) = \text{menores}(c, n)$ 
  funiverso

```

Fig. 1.35: especificación de las cadenas con operación menores.

```

universo ELEM_< caracteriza
  usa BOOL
  tipo elem
  ops  $\_ < \_$ : elem elem  $\rightarrow$  bool
  ecns  $\forall v, v_1, v_2, v_3 \in \text{elem}$ 
     $v < v = \text{falso}$ 
     $((v_1 < v_2) \Rightarrow \neg(v_2 < v_1)) = \text{cierto}$ 
     $((v_1 < v_2) \wedge (v_2 < v_3) \Rightarrow v_1 < v_3) = \text{cierto}$ 
  funiverso

```

Fig. 1.36: definición de los elementos con operación de orden.

1.7 Ejecución de especificaciones

En las secciones anteriores y en los capítulos sucesivos, se considera una especificación como una herramienta de descripción de tipo de datos, que vendrá seguida por una implementación posterior. Para acabar este capítulo, se estudia la posibilidad de aprovechar el trabajo de especificar para obtener un primer prototipo ejecutable del programa (altamente ineficiente, eso sí) mediante un mecanismo llamado reescritura, que es una evolución de unas reglas de cálculo conocidas con el nombre de deducción ecuacional.

1.7.1 La deducción ecuacional

Dado que la especificación de un TAD se puede considerar como la aserción de propiedades elementales, parece lógico plantearse la posibilidad de demostrar propiedades de estos TAD. Por ejemplo, dada la especificación de la fig. 1.16, se puede demostrar la propiedad de los naturales $\text{suma}(\text{suc}(\text{suc}(\text{cero})), \text{suc}(\text{cero})) = \text{suc}(\text{suc}(\text{suc}(\text{cero})))$ aplicando las ecuaciones adecuadas previa asignación de las variables:

$$\begin{aligned} \text{suma}(\text{suc}(\text{suc}(\text{cero})), \text{suc}(\text{cero})) &= \{\text{aplicando 2 con } n = \text{suc}(\text{suc}(\text{cero})) \text{ y } m = \text{cero}\} \\ &= \text{suc}(\text{suma}(\text{suc}(\text{suc}(\text{cero})), \text{cero})) = \{\text{aplicando 1 con } n = \text{suc}(\text{suc}(\text{cero}))\} \\ &= \text{suc}(\text{suc}(\text{suc}(\text{cero}))) \end{aligned}$$

Este proceso se denomina deducción ecuacional (ing., equational calculus). La deducción ecuacional permite construir, mediante una manipulación sistemática de los términos, la llamada teoría ecuacional de una especificación SPEC, abreviadamente Teo_{SPEC} , que es el conjunto de propiedades válidas en todas las álgebras de esta especificación; llamaremos teorema ecuacional a cada una de las propiedades $t_1 = t_2$ que forman Teo_{SPEC} y lo denotaremos mediante $\text{SPEC} \mapsto t_1 = t_2$.

Dada la especificación $\text{SPEC} = (\text{SIG}, E)$, $\text{SIG} = (S, OP)$, y los conjuntos V y W de variables sobre SIG, $V, W \in \text{Vars}_{\text{SIG}}$, la teoría ecuacional Teo_{SPEC} se construye según las reglas siguientes:

- Todas las ecuaciones básicas están dentro de Teo_{SPEC} : $\forall t=t' \in E: t=t' \in \text{Teo}_{\text{SPEC}}$.
- Los teoremas ecuacionales son reflexivos, simétricos y transitivos:

$$\begin{aligned} \forall t, t_1, t_2, t_3: t, t_1, t_2, t_3 \in T_{\text{SIG}}(V): \\ \diamond t=t \in \text{Teo}_{\text{SPEC}}. \\ \diamond t_1=t_2 \in \text{Teo}_{\text{SPEC}} \Rightarrow t_2=t_1 \in \text{Teo}_{\text{SPEC}}. \\ \diamond t_1=t_2 \in \text{Teo}_{\text{SPEC}} \wedge t_2=t_3 \in \text{Teo}_{\text{SPEC}} \Rightarrow t_1=t_3 \in \text{Teo}_{\text{SPEC}}. \end{aligned}$$
- Toda asignación de términos a las variables de un teorema ecuacional da como resultado otro teorema ecuacional:

$$\begin{aligned} \forall t_1, t_2: t_1, t_2 \in T_{\text{SIG}}(V). \forall as_V: as_V: V \rightarrow T_{\text{SIG}}(W): \\ t_1=t_2 \in \text{Teo}_{\text{SPEC}} \Rightarrow \text{eval}_{V, T_{\text{SIG}}(W)}(t_1) = \text{eval}_{V, T_{\text{SIG}}(W)}(t_2) \in \text{Teo}_{\text{SPEC}}. \end{aligned}$$
- Una operación aplicada sobre dos conjuntos diferentes de términos que, tomados por parejas, formen parte de Teo_{SPEC} , da como resultado un nuevo teorema ecuacional:

$$\begin{aligned} - \forall op: op \in OP_{s_1 \dots s_n \rightarrow s}: \forall t_1, t'_1: t_1, t'_1 \in T_{\text{SIG}, s_1}(V) \dots \forall t_n, t'_n: t_n, t'_n \in T_{\text{SIG}, s_n}(V) \\ \{ \forall i: 1 \leq i \leq n: t_i = t'_i \in \text{Teo}_{\text{SPEC}} \} \Rightarrow op(t_1, \dots, t_n) = op(t'_1, \dots, t'_n) \in \text{Teo}_{\text{SPEC}} \end{aligned}$$
- Nada más pertenece a Teo_{SPEC} .

La definición de Teo_{SPEC} recuerda a la de la congruencia \equiv_E inducida por las ecuaciones; la diferencia es que los teoremas son propiedades universales que todavía pueden contener variables, mientras que en el álgebra cociente de términos sólo hay términos sin variables. El nexo queda claro con la formulación del teorema de Birkhoff, que afirma: $\text{SPEC} \mapsto t_1 = t_2$ si y sólo si $t_1 = t_2$ es válida dentro de todas las SPEC-álgebras.

En general, dada una especificación SPEC, no existe un algoritmo para demostrar que $\text{SPEC} \mapsto t_1 = t_2$; el motivo principal es que las ecuaciones se pueden aplicar en cualquier sentido y por eso pueden formarse bucles durante la demostración del teorema. El tratamiento de este problema conduce al concepto de reescritura de términos.

1.7.2 La reescritura

La reescritura de términos (ing., term-rewriting) o, simplemente, reescritura, es una derivación del mecanismo de deducción ecuacional que, en determinadas condiciones, permite usar una especificación como primer prototipo ejecutable de un TAD. Consiste en la manipulación de un término t usando las ecuaciones de la especificación hasta llegar a otro término lo más simplificado posible; en el caso de que t no tenga variables, el resultado estará dentro de la misma clase de equivalencia que t en el álgebra cociente (es decir, representa el mismo valor), y se denominará forma normal de t , denotada por $t\downarrow$; usualmente, la forma normal coincide con el representante canónico de la clase del término.

Dada una especificación $\text{SPEC} = (S, OP, E)$, se puede construir un sistema de reescritura (ing., term-rewriting system) asociado a SPEC orientando las ecuaciones de E para obtener reglas de reescritura (ing., term-rewriting rules), que indican el sentido de la sustitución de los términos; el punto clave estriba en que esta orientación ha de preservar un orden en el "tamaño" de los términos, de manera que la parte izquierda de la regla sea "mayor" que la parte derecha. Por ejemplo, dada la especificación habitual de la suma de naturales:

$$\begin{aligned} E1) & \text{suma}(\text{cero}, n) = n \\ E2) & \text{suma}(\text{suc}(m), n) = \text{suc}(\text{suma}(m, n)) \end{aligned}$$

es necesario orientar las ecuaciones de izquierda a derecha para obtener las reglas:

$$\begin{aligned} R1) & \text{suma}(\text{cero}, n) \rightarrow n \\ R2) & \text{suma}(\text{suc}(m), n) \rightarrow \text{suc}(\text{suma}(m, n)) \end{aligned}$$

Este sistema de reescritura permite ejecutar el término $\text{suma}(\text{suc}(\text{cero}), \text{suc}(\text{cero}))$ aplicando primero la regla 2 con $m = \text{cero}$ y $n = \text{suc}(\text{cero})$ y, a continuación, la regla 1 con $n = \text{suc}(\text{cero})$ para obtener la forma normal $\text{suc}(\text{suc}(\text{cero}))$ que denota el valor 2 de los naturales.

Dado un sistema de reescritura R , se plantean dos cuestiones fundamentales:

- ¿Todo término tiene alguna forma normal usando las reglas de R ? En tal caso, se dice que R es de terminación finita o noetheriano. Algunas situaciones son especialmente inconvenientes para la terminación finita, como la conmutatividad y la asociatividad; así, es imposible orientar la ecuación $\text{suma}(m, n) = \text{suma}(n, m)$. Por ello, existen algunas técnicas para la generación de sistemas de reescritura que tratan estas propiedades.
- ¿Todo término tiene una única forma normal según R ? En tal caso, se dice que R es confluente, y cumple el lema del diamante: si un término se puede reescribir en otros dos diferentes usando las reglas de R , la forma normal de estos dos es la misma.

Todo sistema de reescritura R asociado a una especificación SPEC que sea confluente y noetheriano se llama canónico (ing., canonical) y cumple dos propiedades fundamentales: todo término sin variables tiene una única forma normal usando las reglas de R , y la demostración de teoremas ecuacionales es decidible, siendo $\text{SPEC} \mapsto t_1 = t_2 \Leftrightarrow t_1 \downarrow =_t t_2 \downarrow$, donde la igualdad de las formas normales $=_t$ representa la igualdad sintáctica de los términos. Si una especificación puede convertirse en un sistema de reescritura canónico, la ejecución de especificaciones se reduce al cálculo de formas normales: dado un término t sobre una signatura, su manipulación por un sistema de reescritura canónico resulta en su forma normal (única y siempre existente), que es su valor dentro del álgebra cociente de términos.

Debe tenerse en cuenta que, generalmente, la terminación finita es indecidible, a pesar de que existen condiciones suficientes sobre la forma de las ecuaciones para garantizarla. Además, la confluencia tampoco suele ser decidible; ahora bien, si un sistema de reescritura es de terminación finita entonces sí que lo es. En este contexto, es importante conocer la existencia de un algoritmo llamado algoritmo de complección o de Knuth-Bendix, que intenta convertir una especificación en un sistema canónico de reescritura. La aplicación de este algoritmo puede no acabar, en cuyo caso no se puede afirmar nada sobre la confluencia o noetherianidad (simplemente el método ha fracasado en esta especificación concreta), o bien acabar de dos maneras posibles: correctamente, obteniéndose el sistema de reescritura canónico resultado de orientar las reglas, o bien sin éxito, porque ha sido imposible orientar alguna ecuación y entonces no se puede asegurar la terminación finita. En el caso de que finalice con éxito, el algoritmo puede haber añadido más ecuaciones para garantizar la confluencia del sistema, que serán deducibles de las ecuaciones iniciales.

Se han publicado varios artículos de introducción al tema de la reescritura; se puede consultar, por ejemplo, la panorámica ofrecida por J.P. Jouannaud y P. Lescanne en "Rewriting Systems" (Technique et Science Informatiques, 6(3), 1987), o bien el capítulo 6 del libro Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics, Editorial Elsevier, 1990, capítulo escrito por N. Dershowitz y J.-P. Jouannaud.

Ejercicios

1.1 Dada la signatura:

universo Y es
tipo y, nat
ops crea: $\rightarrow y$
 modif: $y \text{ nat nat} \rightarrow y$
 f: $y \text{ y} \rightarrow \text{nat}$
 h: $y \text{ y} \rightarrow y$
 cero: $\rightarrow \text{nat}$
 suc: $\text{nat} \rightarrow \text{nat}$
funiverso

- a) Clasificar las operaciones por su signatura (es decir, construir los conjuntos $OP_{w \rightarrow s}$).
- b) Escribir unos cuantos términos de género y.
- c) Repetir b) considerando un conjunto de variables $X = (X_y, X_{\text{nat}}, X_y = \{z\}, X_{\text{nat}} = \{x, y\})$.
- d) Escribir al menos tres Y-álgebras que respondan a modelos matemáticos o informáticos conocidos, y dar la interpretación de los géneros y los símbolos de operación de Y.
- e) Describir el álgebra de términos T_Y .
- f) Evaluar el término $t = f(h(\text{crea}, \text{modif}(\text{crea}, \text{suc}(\text{cero}), \text{suc}(\text{suc}(\text{cero}))))$, crea) dentro de todas las álgebras de d).
- g) Dar una función de asignación de las variables de c) dentro de las álgebras de d).
- h) Evaluar el término $t' = f(h(z, \text{modif}(\text{crea}, x, \text{suc}(\text{cero}))))$, crea) dentro de las álgebras de d), usando las asignaciones de g).

1.2 a) Dada la signatura:

universo Y es
tipo y, nat
ops crea: $\rightarrow y$
 añ: $y \text{ nat} \rightarrow y$
 f: $y \rightarrow \text{nat}$
 cero: $\rightarrow \text{nat}$
 suc: $\text{nat} \rightarrow \text{nat}$
funiverso

- i) Clasificar las operaciones por su signatura.
- ii) Escribir al menos cinco Y-álgebras que respondan a modelos matemáticos o informáticos conocidos, y dar la interpretación de los géneros y los símbolos de operación de Y.
- iii) Escribir el álgebra de términos T_Y , encontrando una expresión general recursiva para

los términos de los conjuntos base.

iv) Evaluar el término $t = f(añ(añ(añ(crea, suc(cero)), cero), suc(suc(cero))))$ dentro de todas las álgebras de ii).

b) Dada la especificación $SY = (Y, EY)$, $EY = \{ f(añ(s, n)) = suc(f(s)), f(crea) = cero \}$, se pide:

i) Decir si las ecuaciones se satisfacen o no en las álgebras de a.ii).

ii) Encontrar el álgebra cociente de términos T_{SY} , y escoger representantes canónicos para las clases.

iii) Determinar si una o más de una de las álgebras de a.ii) es isomorfa a T_{SY} (en caso contrario, buscar otra Y-álgebra que lo sea).

iv) Clasificar las álgebras de a.ii) en: modelos iniciales de SY, SY-álgebras e Y-álgebras.

1.3 Dada la especificación:

universo MISTERIO es
tipo misterio
ops $z: \rightarrow \text{misterio}$
 $f: \text{misterio} \rightarrow \text{misterio}$
funiverso

a) ¿Cuál es su modelo inicial?

b) ¿Cuál es el modelo inicial de MISTERIO si añadimos la ecuación $f(z) = z$?

c) ¿Cuál es el modelo inicial de MISTERIO si añadimos la ecuación $f(f(z)) = z$?

1.4 Dada la especificación QUI_LO_SA:

universo QUI_LO_SA es
tipo qui_lo_sa
ops $z: \rightarrow \text{qui_lo_sa}$
 $f: \text{qui_lo_sa} \rightarrow \text{qui_lo_sa}$
 $h: \text{qui_lo_sa} \text{ qui_lo_sa} \rightarrow \text{qui_lo_sa}$
ecns $\forall r, s \in \text{qui_lo_sa}$
 $h(r, z) = r$
 $h(r, f(s)) = f(h(r, s))$
funiverso

justificar brevemente si los modelos siguientes son o no son iniciales:

a) Secuencias de ceros (0^* , λ , $añ$, $conc$); $añ$ es añadir un "0" y $conc$ es la concatenación.

b) Naturales con operación de suma (\mathbb{N} , 0, +1, +).

c) Matrices 2x2 de naturales ($M_{2 \times 2}(\mathbb{N})$, (0 0; 0 0), (+1 +1; +1 +1), +).

d) Igual que c), con la restricción de que todos los elementos de la matriz deben ser iguales.

1.5 Sea la especificación de la figura siguiente, donde se supone que las especificaciones BOOL y NAT especifican el álgebra de Bool B y los naturales N , respectivamente:

universo QUÉ_SOY_YO es
usa BOOL, NAT
tipo qué_soy_yo
ops λ : \rightarrow qué_soy_yo
 añ: qué_soy_yo nat \rightarrow qué_soy_yo
 está?: qué_soy_yo nat \rightarrow bool
ecns $\forall n, m \in \text{nat}; \forall r \in \text{qué_soy_yo}$
 está?(λ , n) = falso
 está?(añ(r , m), m) = NAT.ig(m , n) \vee está?(r , m)
funiverso

a) Demostrar que las álgebras siguientes son QUÉ_SOY_YO-álgebras y comprobar que no son el modelo inicial.

- i) $A_{\text{bool}} \equiv B, A_{\text{nat}} \equiv N, A_{\text{qué_soy_yo}} \equiv P(N)$ {conjuntos de naturales}
 $\lambda_A \equiv \emptyset$ {conjunto vacío}
 añ $_A$: $P(N) \times N \rightarrow P(N)$
 $(C, n) \rightarrow \text{añ}_A(C, n) \equiv C \cup \{n\}$
 está? $_A$: $P(N) \times N \rightarrow B$
 $(C, n) \rightarrow \text{está?}_A(C, n) \equiv n \in C$
- ii) $B_{\text{bool}} \equiv B, B_{\text{nat}} \equiv N, B_{\text{qué_soy_yo}} \equiv N^*_{<}$ {secuencias ordenadas de naturales}
 $\lambda_B \equiv \lambda$ {secuencia vacía}
 añ $_B$: $N^*_{<} \times N \rightarrow N^*_{<}$
 $(l, n) \rightarrow \text{añ}_B(l, n) \equiv \text{añadir } n \text{ a } l \text{ ordenadamente}$
 está? $_B$: $N^*_{<} \times N \rightarrow B$
 $(l, n) \rightarrow \text{está?}_B(l, n) \equiv \text{si } n \text{ está dentro de } l \text{ entonces cierto, sino falso}$
- iii) $C_{\text{bool}} \equiv B, C_{\text{nat}} \equiv N, C_{\text{qué_soy_yo}} \equiv M(N)$ {multiconjuntos de naturales¹⁴}
 $\lambda_C \equiv \emptyset$ {multiconjunto vacío}
 añ $_C$: $M(N) \times N \rightarrow M(N)$
 $(M, n) \rightarrow \text{añ}_C(M, n) \equiv M \cup_M \{n\}$ { \cup_M es la unión de multiconjuntos¹²}
 está? $_C$: $M(N) \times N \rightarrow B$
 $(M, n) \rightarrow \text{está?}_C(M, n) \equiv n \in C$

b) Comprobar que $T_{\text{QUÉ_SOY_YO}, \text{qué_soy_yo}} \approx N^*$ y que el modelo inicial de esta especificación es $T_{\text{QUÉ_SOY_YO}} = (B, N, N^*)$ con las operaciones adecuadas.

c) Determinar el modelo inicial de la especificación QUÉ_SOY_YO al añadirle la ecuación $\text{añ}(\text{añ}(r, x), y) = \text{añ}(\text{añ}(r, y), x)$. Justificar la elección dando el isomorfismo con $T_{\text{QUÉ_SÓY_YO}}$.

¹⁴ Un multiconjunto es un conjunto que admite repeticiones; es decir, el multiconjunto $\{1\}$ es diferente del multiconjunto $\{1, 1\}$. La unión de multiconjuntos conserva, pues, las repeticiones.

d) Determinar y justificar el modelo inicial de la especificación QUÉ_SOY_YO cuando se le añade la ecuación $\text{añ}(\text{añ}(r, x), x) = \text{añ}(r, x)$.

e) Determinar y justificar el modelo inicial de la especificación QUÉ_SOY_YO cuando se le añaden las dos ecuaciones de c) y d) a la vez.

1.6 Sea la especificación de la teoría de grupos:

```

universo TEORIA_GRUPOS es
  tipo g
  ops e: → g {elemento neutro}
  _+_ : g g → g
  i: g → g {inverso}
  ecns ∀x,y,z∈g: x + e = x; x + i(x) = e; (x + y) + z = x + (y + z)
funiverso

```

Usando deducción ecuacional, determinar si las ecuaciones siguientes son teoremas ecuacionales de este modelo, especificando claramente los pasos de las demostraciones.

a) $(x + y) + i(y) = x$. **b)** $x + i(e) = x$. **c)** $e + i(i(x)) = x$. **d)** $x + i(i(y)) = x + y$. **e)** $e + x = x$. **f)** $i(e) = e$. **g)** $i(x) + x = e$.

1.7 Especificar los vectores de índices los naturales de 1 a n que ofrecen los lenguajes de programación imperativos (Pascal, C, etc.) con operaciones de: crear un vector con todas las posiciones indefinidas, modificar el valor del vector en una posición dada y consultar el valor del vector en una posición dada; si se consulta el valor de una posición todavía indefinida, dar error. Expresar el resultado dentro de un universo parametrizado. Hacer una instancia.

1.8 Especificar las matrices $n \times m$ de naturales con las operaciones habituales. Expresar el resultado dentro de un universo parametrizado. Hacer una instancia.

1.9 Dada la especificación de los conjuntos genéricos con operación de pertenencia y un número máximo de elementos, $\text{CONJUNTO}_{\in} \text{ACOTADO}(\text{ELEM}_{=}, \text{VAL}_{\text{NAT}})$, escribir otro universo de cabecera $\text{CONJUNTO2}(\text{ELEM}_{=}, \text{VAL}_{\text{NAT}})$ que use el anterior y defina las operaciones $_ \cap _$ (intersección de conjuntos), $_ \cup _$ (unión de conjuntos) y $_ \supset _$ (inclusión de conjuntos). Hacer una instancia para los conjuntos de, como máximo, 50 naturales, y otra para los conjuntos de, como máximo, 30 conjuntos de, como máximo, 20 naturales.

1.10 Especificar una calculadora de polinomios, que disponga de n memorias para guardar resultados intermedios, y de un polinomio actual (que se ve por el visualizador de la calculadora). Las operaciones de la calculadora son: crear la calculadora con todas las memorias indefinidas y de polinomio actual cero, añadir un término al polinomio actual, recuperar el contenido de una memoria sobre el polinomio actual, guardar el polinomio actual sobre una memoria, sumar, restar y multiplicar los polinomios residentes en dos memorias,

derivar el polinomio residente en una memoria (estas cuatro últimas operaciones dejan el resultado sobre el polinomio actual) y evaluar el polinomio residente en una memoria en un punto dado. Las operaciones han de controlar convenientemente los errores típicos, básicamente referencias a memorias indefinidas o inexistentes. Usar la especificación de los polinomios del apartado 1.5.1, y enriquecerla si es necesario.

1.11 a) Enriquecer la especificación de las cadenas dada en el apartado 1.5.1 (considerándola parametrizada por el tipo de los elementos) con las operaciones:

elimina: cadena \rightarrow cadena, elimina de la cadena todas las apariciones consecutivas de un mismo elemento y deja sólo una

selecciona: cadena \rightarrow cadena, forma una nueva cadena con los elementos que aparecen en una posición par de la cadena; por ejemplo, selecciona(especificación) = seiiain

es_prefijo?: cadena cadena \rightarrow bool, dice si la primera cadena es prefijo de la segunda; considerar que toda cadena es prefijo de sí misma

es_subcadena?: cadena cadena \rightarrow bool, dice si la primera cadena es subcadena de la segunda; considerar que toda cadena es subcadena de sí misma

especula: cadena \rightarrow cadena, obtiene la imagen especular de una cadena; así, especular(especificación) = nóicacifcepse

b) A continuación, hacer una instancia del universo del apartado anterior con cadenas de cualquier cosa, de manera que el resultado sea un género de nombre cad_cad que represente las cadenas de cadenas de cualquier cosa.

c) Enriquecer el universo del apartado b) con las operaciones siguientes:

aplana: cad_cad \rightarrow cadena, concatena los elementos de la cadena de cadenas dada; así, aplana([hola, que, λ , tal]) = holaquetal, donde los corchetes denotan una cadena de cadenas y λ denota la cadena vacía

distr_izq: elem cad_cad \rightarrow cad_cad, dado un elemento v y una cadena de cadenas L , añade v por la izquierda a todas las cadenas de L ; por ejemplo, distr_izq(x , [hola, que, λ , tal]) = [xhola, xque, x , xtal]

subcadenas: cadena \rightarrow cad_cad, dada una cadena l , devuelve la cadena de todas las subcadenas de l conservando el orden de aparición de los elementos; por ejemplo, subcadenas(abc) = [abc, ab, ac, bc, a, b, c, λ]

inserta: elem cad_cad \rightarrow cad_cad, dado un elemento v y una cadena de cadenas L , devuelve la cadena con todas las cadenas resultantes de insertar v en todas las posiciones posibles de las cadenas de L ; por ejemplo, inserta(x , [abc, d]) = [xabc, axbc, abxc, abcx, xd, dx]

1.12 a) Especificar un universo que describa el comportamiento de una urna que pueda ser utilizada por diferentes colectivos (sociedades anónimas, partidos políticos, asambleas de alumnos, etc.) para aceptar o rechazar propuestas a través de votación, cuando les falle la vía del consenso o el engaño. Este universo ha de definir un TAD urna con operaciones:

inic: colectivo \rightarrow urna, prepara la urna para ser usada por un colectivo
 vota: urna miembro bool \rightarrow urna, un miembro emite un voto favorable o desfavorable
 may_absoluta: urna \rightarrow bool, dice si el número de votos favorables supera la mitad de la totalidad de votos del colectivo
 may_simple: urna \rightarrow bool, dice si el número de votos favorables supera la mitad de la totalidad de votos emitidos por el colectivo

El universo ha de poder ser instanciado con cualquier clase de miembro y colectivo, y ha de prever que hay colectivos en los que miembros diferentes pueden disponer de un número diferente de votos. De todas formas, para garantizar un cariz mínimamente democrático, la urna sólo ha de poder ser usada si no hay ningún miembro que tenga más del 50% de los votos totales del colectivo. Especificar la urna en un universo genérico con los parámetros que se consideren oportunos.

b) Construir una especificación que sirva para modelizar una sociedad especuladora, fundada por tres personas y regida por las siguientes directrices:

- Los miembros de la sociedad (socios, entre los cuales están los fundadores) se identifican mediante una cadena de caracteres.
- Los tres fundadores disponen de dos votos cada uno a la hora de tomar decisiones; el resto de socios sólo dispone de un voto por cabeza.

La sociedad estará dotada de un mecanismo de toma de decisiones por mayoría simple.

El universo donde se encapsule esta especificación ha de tener las operaciones:

fundación: socio socio socio \rightarrow sociedad, crea la sociedad con los tres socios fundadores
 añadir: sociedad socio \rightarrow sociedad, añade un nuevo socio; error si ya estaba
 es_socio?: sociedad socio \rightarrow bool, dice si el socio está en la sociedad
 inicio_votación: sociedad \rightarrow sociedad, prepara la futura votación
 añade_voto: sociedad socio bool \rightarrow sociedad, cuenta el voto del socio
 prospera: sociedad \rightarrow bool, contesta si la propuesta se acepta

Hay un universo SOCIO que define el género socio y la operación ident: socio \rightarrow cadena (entre otras). También se puede utilizar el resultado del apartado a) y el universo PAR de la fig. 1.31.

1.13 Una hoja de cálculo puede considerarse como una matriz rectangular de celdas, las cuáles se identifican por su coordenada (par de naturales que representan fila y columna). Dentro de una celda puede no haber nada, puede haber un valor entero o puede haber una fórmula. De momento, para simplificar la especificación, supondremos que las fórmulas son de la forma $a + b$, donde a y b son coordenadas de celdas. Definimos la evaluación de una celda recursivamente del siguiente modo: si la celda está vacía, la evaluación es 0; si la celda contiene el entero n , la evaluación es n ; en cualquier otro caso, la celda contiene una fórmula $c_1 + c_2$ y el resultado es la evaluación de la celda de coordenada c_1 , sumado a la evaluación de la celda de coordenada c_2 . La signatura de este tipo es:

crea: $\text{nat nat} \rightarrow \text{hoja}$, crea una hoja de cálculo con las filas y las columnas numeradas del uno a los naturales dados; inicialmente, todas las celdas están vacías
 ins_valor: $\text{hoja coordenada enter} \rightarrow \text{hoja}$, inserta el entero en la celda de coordenada dada, sobrescribiendo el contenido anterior de la celda
 ins_fórmula: $\text{hoja coordenada coordenada coordenada} \rightarrow \text{hoja}$, siendo c_1 , c_2 y c_3 las coordenadas dadas, inserta la fórmula $c_2 + c_3$ en la celda de coordenada c_1 de la hoja, sobrescribiendo el contenido anterior
 valor: $\text{hoja coordenada} \rightarrow \text{entero}$, devuelve la evaluación de la celda dada

Las coordenadas se definen como instancia del universo PAR (v. fig. 1.31), siendo los dos parámetros formales el tipo de los naturales.

- a) Especificar el tipo hoja, controlando todos los errores posibles (referencias a coordenadas fuera de rango, ciclos en las fórmulas, etc.).
- b) Pensar en la extensión en caso de permitir fórmulas arbitrarias (quizás se podría definir el tipo fórmula, con algunas operaciones adecuadas...).

1.14 Se quiere especificar una tabla de símbolos para lenguajes modulares (como Merlí) que organice los programas como jerarquías de módulos. Las operaciones escogidas son:

crea: $\rightarrow \text{ts}$, crea la tabla vacía
 declara: $\text{ts módulo cadena caracts} \rightarrow \text{ts}$, registra que el identificador ha sido definido dentro del módulo con una descripción dada
 usa: $\text{ts módulo módulo} \rightarrow \text{ts}$, registra que el primer módulo usa el segundo, es decir, dentro del módulo usador son visibles los objetos declarados dentro del módulo usado
 usa?: $\text{ts módulo} \rightarrow \text{cjt_módulos}$, da todos los módulos que usa el módulo dado. Hay que considerar que el uso de módulos es transitivo, es decir, si m_1 usa m_2 y m_2 usa m_3 , entonces m_1 usa m_3 , aunque no esté explícitamente declarado. Para simplificar, puede considerarse que todo módulo se usa a sí mismo
 consulta: $\text{ts módulo cadena} \rightarrow \text{cjt_caracts}$, devuelve el conjunto de todas las descripciones para el identificador que sean válidas dentro del módulo dado (es decir, las descripciones declaradas en todos los módulos usados por el módulo dado)

(Notar que no hay una operación explícita de declaración de módulos; un módulo existe si hay alguna declaración -de uso o de identificador- que así lo indique.)

Se puede suponer la existencia de las operaciones necesarias sobre los conjuntos y las cadenas.

1.15 Se quiere especificar un diccionario de palabras compuestas de n partes como máximo, $n < 10$. Cada parte es una cadena de caracteres. El diccionario ha de presentar operaciones de acceso individual a las palabras y de recorrido alfabético; concretamente:

crea: $\rightarrow \text{dicc}$, crea el diccionario vacío
 inserta: $\text{dicc cadena ... cadena (10 cadenas)} \rightarrow \text{dicc}$, registra la inserción de una palabra

compuesta de k partes, $k \leq n$; todas las cadenas de la $k+1$ a la n son la cadena vacía; da error si la palabra ya estaba en el diccionario

borra: `dicc cadena ... cadena (10 cadenas)` → `dicc`, borra una palabra de k partes, $k \leq n$; todas las cadenas de la $k+1$ a la n son la cadena vacía; error si la palabra no está en el diccionario

está?: `dicc cadena ... cadena (10 cadenas)` → `bool`, comprueba si una palabra de k partes está o no en el diccionario, $k \leq n$; todas las cadenas de la $k+1$ a la n son la cadena vacía

lista: `dicc` → `lista_cadenas`, devuelve la lista ordenada alfabéticamente de todas las palabras del diccionario sin considerar las partes que las componen

Es importante definir correctamente la igualdad de palabras: dos palabras son iguales si y sólo si están compuestas por las mismas partes; así, la palabra *méd/ic/amente* es diferente de la palabra *médic/a/mente* porque, a pesar de tener los mismos caracteres, su distribución en partes no concuerda (considerando '/' como separador de partes).

Se puede suponer la existencia de las operaciones necesarias sobre las cadenas y las listas.

1.16 a) En el juego del mastermín participan dos jugadores A y B con fichas de colores. A imagina una combinación "objetivo" de cinco fichas (sin repetir colores) y B debe adivinarla. El jugador B va proponiendo jugadas (combinaciones de cinco fichas de colores no repetidos; se dice que cada ficha va en una posición determinada), y el jugador A responde con el número de aciertos y de aproximaciones que contiene respecto al objetivo. Se produce acierto en cada posición de la jugada propuesta que contenga una ficha del mismo color que la que hay en la misma posición del objetivo. Se produce aproximación en las posiciones de la jugada propuesta que no sean aciertos, pero sí del mismo color que alguna otra posición del objetivo. En el siguiente ejemplo, la propuesta obtiene un acierto y dos aproximaciones:

Posiciones	1	2	3	4	5
OBJETIVO:	Amarillo	Verde	Rojo	Azul	Fucsia
PROPUESTA:	Azul	Verde	Amarillo	Marrón	Naranja

La secuencia de jugadas hechas hasta un momento dado proporciona una pista para la siguiente propuesta del jugador B, de manera que, cuando B propone una jugada, puede comprobar previamente que no se contradiga con los resultados de las jugadas anteriores. Una propuesta coherente a partir del ejemplo anterior (donde B supone que el acierto era el marrón y las aproximaciones el amarillo y el verde) sería:

Posiciones	1	2	3	4	5
PROPUESTA:	Amarillo	Púrpura	Gris	Marrón	Verde

Especificar este juego con la signatura siguiente:

crea: jugada \rightarrow mm, deposita en el tablero la jugada objetivo

juega: mm jugada \rightarrow mm, deposita en el tablero la jugada siguiente propuesta

aciertos, aproxs: mm \rightarrow nat, da el número de aciertos y aproximaciones de la última jugada propuesta, respectivamente

coherente: mm jugada \rightarrow bool, comprueba si una jugada es coherente con las anteriores

Suponer que se dispone de una constante para cada color: blanco, amarillo, ...: \rightarrow color, así como de una operación de comparación, ig: color color \rightarrow bool. Se dispone, además, de la definición del género jugada, con unas operaciones básicas que se pueden suponer correctamente especificadas:

crea: color color color color color \rightarrow jugada, constructora de jugadas

qué_color: jugada nat \rightarrow color, devuelve el color de la ficha que hay en la posición dada de la jugada

b) Modificar la especificación permitiendo la repetición de colores en las jugadas.