Introducción a la Programación Funcional con Haskell

Segunda edición

Richard Bird

Prentice Hall

Introducción a la programación funcional con Haskell

Introducción a la programación funcional con Haskell

Segunda edición

Richard BIRD
University of Oxford

Traducción:
Ricardo Peña Marí
Profesor Titular de Lenguajes y Sistemas Informáticos
Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid

Revisión técnica:
María Alpuente Frasnedo
Catedrática de Lenguajes y Sistemas Informáticos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Salvador Lucas Alba
Profesor Titular de Lenguajes y Sistemas Informáticos
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

PRENTICE HALL
Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • Sao Paulo • White Plains

Datos de catalogación bibliográfica

RICHARD BIRD
Introducción a la programación funcional con Haskell. Segunda edición
PEARSON EDUCACIÓN, S.A., Madrid, 2000

ISBN: 84-8322-176-4
Materia: Informática 681.3

Formato 195 x 250 Páginas: 384

RICHARD BIRD
Introducción a la programación funcional con Haskell. Segunda edición

Todos los derechos reservados.
No está permitida la reproducción total o parcial de esta obra ni su tratamiento o transmisión por cualquier medio o método sin autorización escrita de la Editorial.

DERECHOS RESERVADOS
© 2000 respecto a la primera edición en español por:
PEARSON EDUCACIÓN, S.A.
Núñez de Balboa, 120
28006 Madrid

ISBN: 84-8322-176-4
Depósito Legal: M-10.443-2000

PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

Traducido de:
Introduction to functional Programaming Using Haskell. Second Edition
Copyright © Prentice Hall Europe
MCMXCVIII
ISBN 0-13-484346-0

Este libro pertenece a la serie de Informática:
PRENTICE HALL SERIES IN COMPUTTER SCIENCE
Editores de la serie: C.A.R. Hoare and Richard Bird

Edición en español:
Editor: Andrés Otero
Asistente editorial: Ana Isabel García
Diseño de cubierta: DIGRAF, S. A.
Composición: Jesús Soto
Impreso por: COFÁS, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

Contenido

Prólogo a la segunda edición ix

Prólogo a la edición en español xiii

1 Conceptos fundamentales 1
- 1.1 Sesiones y guiones . . . 1
- 1.2 Evaluación . . . 4
- 1.3 Valores . . . 6
- 1.4 Funciones . . . 8
- 1.5 Definiciones . . . 15
- 1.6 Tipos . . . 19
- 1.7 Especificaciones . . . 22
- 1.8 Notas del capítulo . . . 24

2 Tipos de datos simples 27
- 2.1 Booleanos . . . 27
- 2.2 Caracteres . . . 32
- 2.3 Enumeraciones . . . 35
- 2.4 Tuplas . . . 37
- 2.5 Otros tipos . . . 41
- 2.6 Renombramientos . . . 42
- 2.7 Cadenas . . . 45
- 2.8 Notas del capítulo . . . 49

3 Números 51
- 3.1 Números naturales . . . 51
- 3.2 Inducción . . . 55
- 3.3 La función *fold* . . . 62
- 3.4 Números en Haskell . . . 65
- 3.5 Ejemplo: los racionales . . . 68
- 3.6 Ejemplo: búsqueda lineal y binaria . . . 70
- 3.7 Números de Church . . . 74
- 3.8 Notas del capítulo . . . 77

4 Listas 79
4.1 Notación para listas 79
4.2 Operaciones sobre listas 82
4.3 Las funciones *map* y *filter* 93
4.4 La función *zip* 99
4.5 La familia de funciones *fold* 102
4.6 Leyes relativas a *fold* 109
4.7 Notas del capítulo 116

5 Ejemplos 119
5.1 Convirtiendo números en palabras 119
5.2 Produciendo una lista de notas 122
5.3 Aritmética entera de rango arbitrario 128
5.4 Impresión de un calendario 138
5.5 Procesamiento de textos 144

6 Árboles 153
6.1 Árboles binarios 153
6.2 Árboles binarios de búsqueda 160
6.3 Árboles binarios montículo 164
6.4 Rosadelfas 167
6.5 Ejemplo: árboles de Huffman 173
6.6 Ejemplo: el número de Meertens 178
6.7 Notas del capítulo 183

7 Eficiencia 185
7.1 Evaluación perezosa 185
7.2 Estudio asintótico del coste 189
7.3 Parámetros acumuladores 194
7.4 Tuplamiento 198
7.5 Controlando el espacio 203
7.6 Fusión, diferenciación finita y deforestación 207
7.7 Notas del capítulo 211

8 Tipos abstractos de datos 213
8.1 Conceptos básicos 213
8.2 Módulos 222
8.3 Conjuntos 224
8.4 Bolsas 234
8.5 Vectores flexibles 238
8.6 Colas 244
8.7 Notas del capítulo 248

9 Listas infinitas 249
9.1 Recapitulación 249
9.2 Las listas infinitas como límites 251
9.3 Propiedades de las listas infinitas 256
9.4 Estructuras cíclicas 258
9.5 Ejemplo: el juego de tijeras-piedra-papel 262
9.6 Interacción basada en el flujo de datos 269
9.7 Notas del capítulo 272

10 Mónadas 275
10.1 Interacción monádica 275
10.2 Variaciones sobre un evaluador 283
10.3 Leyes de las mónadas 291
10.4 Combinando mónadas 297
10.5 Notas del capítulo 304

11 Análisis sintáctico 307
11.1 Secuenciación 307
11.2 Alternativa 310
11.3 Repetición 312
11.4 Eficiencia 314
11.5 Notas del capítulo 317

12 Un demostrador automático 319
12.1 Consideraciones básicas 319
12.2 Expresiones, leyes y cómputos 324
12.3 Ajustes y sustituciones 329
12.4 Subexpresiones y reescritura 333
12.5 Comprobando el calculador 336
12.6 Notas del capítulo 347

A Algunas funciones estándar 349

Bibliografía 357

Índice analítico 365

Prólogo a la segunda edición

La segunda edición es una versión completamente reorganizada y expandida de la primera. Sin embargo, he seguido siendo fiel a los objetivos de la original, tal como se expresa en los dos primeros párrafos del prólogo a la primera edición:

> Éste es un texto introductorio sobre programación en general y sobre programación funcional en particular. No se requiere ningún conocimiento sobre computadores ni experiencia en escribir programas. El libro es, por tanto, adecuado para un curso de programación dirigido a estudiantes de primer año, pero también se puede usar como una introducción a la programación funcional para estudiantes que ya tienen experiencia como programadores.
>
> Para obtener el máximo provecho del libro, el estudiante debería saber algo de matemáticas o al menos tener conocimientos generales de los principios del razonamiento matemático. Nuestra primera intención al escribir este libro es enfocar la programación como actividad matemática y así el razonamiento matemático subyace en los fundamentos de nuestra materia. La programación funcional involucra notaciones y conceptos que deberían ser familiares a cualquiera que tenga algo de experiencia matemática. Por ejemplo, cualquier estudiante familiarizado con el uso de las funciones trigonométricas básicas para formular problemas de geometría y que haya aplicado leyes trigonométricas simples e identidades para buscar soluciones a estos problemas, pronto apreciará que se requiere un tipo de actividad similar para resolver problemas computacionales por medio de programas funcionales. Se desprende de lo anterior que la clase de razonamiento matemático requerido no es muy complicada o especializada, tan sólo se necesita la habilidad general para seguir las manipulaciones de fórmulas mediante la aplicación de leyes algebraicas y la capacidad de apreciar por qué tales manipulaciones pueden ser útiles para resolver problemas prácticos.

Un cambio importante de la presente edición es el uso de un lenguaje específico, Haskell, para expresar los programas funcionales. Cuando apareció la primera edición, el diseño de los lenguajes funcionales perezosos se encontraba en un estado de transición, con varios dialectos diferentes, aunque estrechamente relacionados, compitiendo

por captar la atención. En aquel momento intentamos tener en cuenta este estado de cambio usando una notación neutral, similar aunque no idéntica a la de Miranda (Miranda es una marca registrada de Research Software Limited), uno de los lenguajes funcionales perezosos más populares en ese momento. Diez años después, la situación se ha estabilizado de modo que Haskell y un pariente popular, Gofer, diseñado por Mark Jones, se han acreditado ahora como el lenguaje preferido por los programadores de lenguajes funcionales perezosos. En particular, HUGS (Haskell Users' Gofer System) se diseñó como un sistema interactivo simple apto para ser usado en PCs con capacidades limitadas. Al final del Capítulo 1, se dan referencias a Haskell y a HUGS y se indica cómo obtenerlos. La versión de Haskell que usamos en este libro es Haskell 1.3 (y también HUGS 1.3). El propio lenguaje Haskell ha sido sometido a un proceso de estandarización y evolución modestas, habiéndose publicado los detalles de *Standard Haskell* en el verano de 1998. Sin embargo, puesto que no cubrimos todos los detalles de Haskell, es improbable que cualquier estandarización que se proponga afecte a este texto de modo significativo.

El segundo cambio importante está relacionado con la organización del material. La edición actual introduce muy al principio la idea de definir nuevos tipos de datos. El núcleo de la primera parte del libro está dedicado ahora al estudio de tres tipos de datos recursivos: números naturales, listas y árboles. Cada tipo de datos se presenta junto con el modo de definir funciones recursivas sobre ese tipo de datos y junto con la técnica de demostración por inducción usada para establecer las propiedades de esas funciones recursivamente definidas. Así, se introducen tres ideas importantes como aspectos diferentes de una sola. Los ejemplos y los casos de estudio se han cambiado o actualizado. Hay capítulos completamente nuevos sobre tipos abstractos de datos, eficiencia y programación funcional monádica. La bibliografía se ha ampliado considerablemente y se han añadido unas breves notas bibliográficas al final de cada capítulo.

El tercer cambio importante está en que, debido a diversos compromisos y a un nuevo cambio de entorno laboral, mi coautor Philip Wadler ha podido participar sólo de manera limitada en la preparación de este texto. Ha suministrado sugerencias y comentarios detallados sobre un borrador de los seis primeros capítulos, así como el material que constituye la base del capítulo sobre programación monádica. Aunque sus contribuciones a la comprensión de la programación funcional durante los últimos diez años aparecen por doquier en el libro, es únicamente mía la responsabilidad de los errores, interpretaciones equivocadas o énfasis inapropiados. Cuando se necesite una tercera edición, espero que podamos unir nuestras fuerzas otra vez para escribirla.

Advertencia para el profesor

La segunda edición ha crecido en unas cien páginas y el material no puede ser abarcado completamente en un solo curso. Un primer curso sobre programación funcional podría basarse en los primeros seis capítulos, tratando el Capítulo 3 de un modo más superficial que los otros. De acuerdo con nuestra experiencia, los estudiantes comienzan a apreciar lo que puede ofrecer un estilo de programación funcional sólo cuando se aborda el tema de las listas. El resto del material se puede adaptar a un segundo curso de programación, poniendo el énfasis en la modularidad y en las estructuras

de datos. El material del libro se ha usado también para un curso de diseño de algoritmos y de análisis de la eficiencia, así como en cursos sobre las matemáticas de la verificación y síntesis de programas. Se deberían complementar las clases magistrales con trabajo práctico de laboratorio. Gran parte de los ejemplos y casos de estudio se ha usado como base de proyectos prácticos con un éxito considerable. Muchas de las secciones individuales también contienen diversos ejercicios para autoestudio. Está previsto publicar las soluciones a estos ejercicios en la página web

`http://www.comlab.ox.ac.uk/oucl/publications/books/functional/`

Esta página también contiene muchos de los programas que aparecen en el texto, así como enlaces a Haskell, HUGS y material adicional.

Agradecimientos

Esta edición, como la primera, llevó más tiempo del esperado. Mi mayor deuda de gratitud es con Phil Wadler. Además de las razones citadas anteriormente, su generosidad me permitió completar el texto por mí mismo. La segunda edición fue escrita durante un año sabático, libre de tareas docentes y administrativas y estoy verdaderamente agradecido a la Universidad de Oxford y al Lincoln College por soportar duras cargas académicas a causa de este libro.

El libro se ha beneficiado enormemente del apoyo entusiasta y del consejo contructivo de colegas y estudiantes, tanto de Oxford como de otras universidades. Las sugerencias de los colegas del *Programming Research Group* de Oxford han sido particularmente relevantes puesto que ellos han utilizado personalmente el material del libro para la impartición de lecciones a sus estudiantes mientras el curso estaba ya en marcha. Otras personas que nos han escrito o enviado correos electrónicos a Phil o a mí con sugerencias, o simplemente para indicar nuestras erratas y pequeños errores en la primera edición, son las siguientes: Nils Andersen, Franklin Chen, Sharon Curtis, Martin Filby, Simon Finn, Jeroen Fokker, Maarten Fokkinga, Jeremy Gibbons, Robert Giegerich, Kevin Hammond, Gerard Huet, Michael Hinchey, Tony Hoare, Iain Houston, John Hughes, Graham Hutton, Stephen Jarvis, Geraint Jones, Mark Jones, John Launchbury, Paul Licameli, David Lester, Iain MacCullum, Ursula Martin, Lambert Meertens, Erik Meijer, Quentin Miller, Oege de Moor, Chris Okasaki, Oskar Permvall, Simon Peyton Jones, Mark Ramaer, Hamilton Richards, Dan Russell, Don Sannella, Antony Simmons, Deepak D'Souza, John Spanondakis, Mike Spivey, Joe Stoy, Bernard Sufrin, Masato Takeichi, Peter Thiemann, David Turner, Colin Watson y Stephen Wilson. En particular, Jeremy Gibbons y Rob Hoogerwoord leyeron borradores de este manuscrito y sugirieron gran número de correcciones. Quisiera también agradecer a Jim Davies el haberme ayudado a mejorar mis conocimientos sobre LaTeX y a Jackie Harbor de Prentice Hall por su constante apoyo. El texto se preparó en un Apple Macintosh Powerbook 180, usando BBEdit como editor y OzTeX como procesador LaTeX. Fue un placer utilizar estos tres sistemas.

Oxford
Diciembre, 1997

Richard Bird

Prólogo a la edición en español

El criterio principal que ha guiado esta traducción ha sido el de emplear términos castellanos para *todos* los conceptos desarrollados en el libro y para la *mayoría* de los programas que contiene. Tan sólo se han dejado sin traducir los nombres de las funciones que forman parte del preludio estándar de Haskell, el lenguaje base del libro, y los mensajes emitidos por el intérprete de dicho lenguaje.

Se ha decidido así para ofrecer al lector la posibilidad de ejecutar en dicho intérprete los programas que aquí se presentan. En unas pocas ocasiones, esta decisión da lugar a definir funciones con nombres mixtos en español y en inglés (p.e. *mapArbin*) porque derivan de una función del preludio y de un tipo o una función definidos en español. En un caso concreto, el ejemplo de conversión de números a su expresión equivalente en palabras del Capítulo 5, se han modificado los algoritmos originales para que resuelvan el mismo problema en español. No podría ser de otro modo ya que las reglas españolas e inglesas no son las mismas para este problema. Los algoritmos presentados han sido comprobados exhaustivamente, por lo que no es probable que contengan errores.

La traducción de un libro especializado como éste proporciona al traductor la oportunidad de acuñar terminología en español para conceptos que han nacido y se han desarrollado en otra lengua. Es un privilegio que lleva aparejada una gran responsabilidad. Por ello se ha procurado huir de las traducciones *perezosas* que suelen utilizar la palabra española más parecida sintácticamente a la palabra original inglesa aunque sus respectivas semánticas pudieran ser bien distintas. Ello lleva a veces a emplear términos semánticamente más correctos pero poco familiares al oído. Se ha procurado consensuar las decisiones tomadas en cada caso con los revisores técnicos (a los que desde aquí agradezco su minuciosa labor que ha contribuido en gran medida a mejorar la presentación general del libro). Sin embargo, no tienen por qué ser aceptadas por todos los lectores que quizá emplean en algún caso términos distintos para los mismos conceptos. Para facilitar la labor de correspondencia, siempre que aparece un término nuevo, normalmente en letra *inclinada*, se incluye a pie de página una *Nota del Traductor* (abreviada *N. del T.*) con el término original inglés.

Agradezco la labor realizada por las numerosas personas que han colaborado en la confección práctica de esta traducción, en sus numerosas fases de introducción de texto, dar formato, revisión técnica, añadir correcciones, etc. También expreso mi reconocimiento a la difícil tarea de coordinación de todas estas labores llevada a cabo

por la editorial y reclamo para mí la responsabilidad de cualquier error o de cualquier expresión malsonante que hayan podido pasar inadvertidos.

Deseo agregar que ha sido un placer traducir este excelente libro. Espero que los lectores sepan disculpar las deficiencias encontradas y que disfruten tanto de él como lo ha hecho este traductor.

Madrid
Diciembre de 1999

Ricardo Peña Marí

Capítulo 1

Conceptos fundamentales

La programación en un lenguaje funcional consiste en construir definiciones y en usar el computador para evaluar expresiones. El papel principal del programador es la definición de una función que permita resolver un problema dado. Esta función, que puede involucrar a un cierto número de funciones auxiliares, se describe en una notación fundada en principios matemáticos comunes. El computador actúa como un evaluador o calculadora; su tarea es evaluar expresiones y mostrar los resultados. A este respecto, el computador se parece mucho a una calculadora ordinaria de bolsillo. Lo que distingue un calculador funcional de sus parientes más humildes es la capacidad del programador de introducir definiciones que incrementan su potencia de cálculo. Las expresiones que contienen apariciones de nombres de funciones definidas por el programador se evalúan usando las definiciones dadas como reglas de simplificación que convierten dichas expresiones en un valor apto para ser presentado como salida.

1.1 Sesiones y guiones

Para ilustrar el uso de un computador como una calculadora, imaginemos que estamos sentados delante de la pantalla de un terminal que muestra un signo de invitación

```
?
```

en una ventana. Ahora podemos escribir una expresión, seguida por un carácter de fin de línea y el computador responderá mostrando el resultado de la evaluación de la expresión, seguido de una nueva invitación en la línea siguiente, indicando que el proceso puede comenzar de nuevo con otra expresión.

Un tipo de expresión que podríamos escribir es un número:

```
? 42
42
```

Aquí la respuesta del computador consiste simplemente en reproducir el número que hemos escrito. El número decimal 42 es una expresión que está en la forma más simple posible y evaluar su resultado no requiere ninguna simplificación.

Podríamos escribir un tipo de expresión ligeramente más interesante:

? 6 × 7
42

Aquí el computador puede simplificar la expresión llevando a cabo la multiplicación. En este libro usaremos las notaciones matemáticas habituales para escribir expresiones; en particular, el operador de multiplicación será denotado por el signo ×, en lugar del asterisco * empleado en Haskell.

No continuaremos, por el momento, con las formas posibles de expresiones numéricas y de otras clases que pueden ser sometidas a evaluación; el punto importante a retener ahora es que se pueden simplemente escribir expresiones y hacer que el computador las evalúe. Esta secuencia de interacciones entre el usuario y el computador se denomina *sesión*.

Un aspecto de la programación funcional que resulta intelectualmente más interesante es la construcción de definiciones. A una lista de definiciones se le designa *guión*[1]. Aquí se muestra un ejemplo de guión:

$$
\begin{array}{lcl}
\mathit{cuadrado} & :: & \mathit{Integer} \to \mathit{Integer} \\
\mathit{cuadrado}\ x & = & x \times x \\
\mathit{minimo} & :: & (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\
\mathit{minimo}\ (x, y) & = & \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y
\end{array}
$$

En este guión se han definido dos funciones llamadas *cuadrado* y *minimo*. La función *cuadrado* toma un entero como argumento y devuelve su cuadrado; la función *minimo* toma un par de enteros como argumentos y devuelve el valor más pequeño. La sintaxis para construir definiciones se ha tomado de Haskell, el lenguaje de programación adoptado en este libro, y será explicada en su momento. Nótese, no obstante, que las definiciones han sido escritas como ecuaciones entre ciertas clases de expresiones; las expresiones pueden contener *variables*, denotadas aquí por los símbolos x e y. Más aún, cada función está acompañada por una descripción de su *tipo*; por ejemplo, $(\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer}$ describe el tipo de las funciones que toman un par de enteros como argumento y devuelven un entero como resultado. Tales descripciones de tipo reciben el nombre de *asignaciones de tipo* o *signaturas de tipo*.

Una vez creado un guión, podemos cargarlo en el computador y abrir una nueva sesión. Por ejemplo, ahora es posible realizar la siguiente sesión:

? *cuadrado* 3768
14197824

? *cuadrado* 14198724
201578206334976

? *cuadrado* (*minimo* (5, 3+4))
25

Nótese, de paso, que la aritmética del tipo *Integer* es exacta: no existe restricción alguna sobre el tamaño de los enteros que se pueden calcular.

[1] N. del T. En inglés, *script*.

El objetivo de una definición es introducir un *vínculo* que asocia un cierto nombre y cierta definición. Un conjunto de vínculos se denomina *entorno* o *contexto*. Las expresiones se evalúan siempre respecto a algún contexto y pueden contener nombres que tienen vinculada alguna definición en ese contexto. El evaluador de Haskell usará las definiciones asociadas con estos nombres como reglas para simplificar expresiones.

Algunas expresiones pueden ser evaluadas sin tener que suministrar un contexto. En Haskell se dan un cierto número de operaciones como primitivas en el sentido de que sus reglas de simplificación están incluidas en el evaluador. Por ejemplo, las operaciones básicas de la aritmética son primitivas. Otras operaciones usadas frecuentemente se encuentran predefinidas en guiones especiales llamados *preludios* o *librerías*, que se cargan cuando se arranca el computador.

El programador puede en cualquier momento alterar el guión para añadir o modificar definiciones. El nuevo guión se puede cargar de nuevo en el evaluador para suministrar un nuevo contexto y empezar otra sesión. Por ejemplo, supongamos que cambiamos el guión anterior para que aparezca del modo siguiente:

$$
\begin{array}{lcl}
\mathit{cuadrado} & :: & \mathit{Float} \to \mathit{Float} \\
\mathit{cuadrado}\ x & = & x \times x \\
\mathit{delta} & :: & (\mathit{Float}, \mathit{Float}, \mathit{Float}) \to \mathit{Float} \\
\mathit{delta}\ (a, b, c) & = & \mathit{sqrt}\ (\mathit{cuadrado}\ b - 4 \times a \times c)
\end{array}
$$

El tipo asignado a *cuadrado* ha sido cambiado por $\mathit{Float} \to \mathit{Float}$. En Haskell, el tipo *Float* representa el conjunto de los números en coma flotante de precisión simple. La función *delta* depende de una función predefinida *sqrt* que extrae raíces cuadradas.

Una vez cargado de nuevo el guión podemos comenzar una nueva sesión y escribir por ejemplo:

? *delta* (4.2, 7, 2.3)
3.2187

Resumiendo los puntos importantes explicados hasta el momento son éstos:

- Los guiones son colecciones de definiciones suministradas por el programador.
- Las definiciones se expresan como ecuaciones entre ciertas clases de expresiones y describen funciones matemáticas. Las definiciones están acompañadas por signaturas de tipo.
- Durante una sesión, se introducen expresiones para ser evaluadas; estas expresiones pueden contener referencias, tanto a las funciones definidas en el guión, como a otras funciones definidas en los preludios o librerías.
- En Haskell se pueden usar al menos dos clases diferentes de números en los cálculos: enteros de rango arbitrario (elementos del tipo *Integer*) y números en coma flotante de precisión simple (elementos del tipo *Float*).

Ejercicios

1.1.1 Usando la función *cuadrado*, diseñar una función *cuarta* que eleva su argumento a la cuarta potencia.

1.1.2 Definir una función *maximo* que devuelve el mayor de sus dos argumentos.

1.1.3 Definir una función para calcular el área de un círculo, dado su radio r (usar 22/7 como aproximación de π).

1.2 Evaluación

El computador evalúa una expresión reduciéndola a la forma equivalente más simple y mostrando el resultado. Para describir este proceso, se usarán indistintamente los términos *evaluación*, *simplificación* y *reducción*. Para dar una idea intuitiva, considérese la expresión *cuadrado* (3 + 4); una secuencia posible es

```
        cuadrado (3 + 4)
=         {definición de +}
        cuadrado 7
=         {definición de cuadrado}
        7 × 7
=         {definición de ×}
        49
```

Los pasos primero y tercero hacen referencia al uso de reglas predefinidas para sumar y multiplicar, mientras que el segundo exige utilizar la regla sumistrada por el programador para definir *cuadrado*. Esto significa que el computador simplemente interpreta la definición *cuadrado* $x = x \times x$ como una regla de reescritura de izquierda a derecha que reduce las expresiones que contienen *cuadrado*. La expresión '49' no se puede reducir más, de modo que éste es el resultado mostrado por el computador. Una expresión se dice que está en *forma canónica*, o en *forma normal*, si no puede reducirse más. Por tanto, '49' está en forma normal.

Otra secuencia de reducción para *cuadrado* (3 + 4) es

```
        cuadrado (3 + 4)
=         {definición de cuadrado}
        (3 + 4)×(3 + 4)
=         {definición de + (aplicada al primer término)}
        7×(3 + 4)
=         {definición de +}
        7×7
=         {definición de ×}
        49
```

En esta secuencia de reducción se aplica primero la regla para *cuadrado*, pero el resultado final es el mismo. Una característica típica de la programación funcional es que si dos secuencias de reducción diferentes terminan, conducen al mismo resultado. En

otras palabras, el significado de una expresión es su valor y la tarea del computador consiste simplemente en obtenerlo.

Demos otro ejemplo. Considérese el guión

```
tres      :: Integer → Integer
tres x    =  3

infinito  :: Integer
infinito  =  infinito + 1
```

No está claro qué entero define la segunda ecuación, si es que define alguno. Pero, sin embargo, el computador puede utilizar la ecuación como una regla de reescritura. Ahora considérese la simplificación de la expresión *tres infinito*. Si tratamos de simplificar primero *infinito*, entonces obtenemos la secuencia de reducción

```
        tres infinito
=         {definición de infinito}
        tres (infinito + 1)
=         {definición de infinito}
        tres ((infinito + 1) + 1)
=         {y así sucesivamente...}
        ...
```

Esta secuencia de reducción no termina. En cambio, si tratamos de simplificar primero *tres*, entonces obtenemos la secuencia

```
        tres infinito
=         {definición de tres}
        3
```

Esta secuencia termina en un paso. Por tanto, ciertos modos de simplificar una expresión pueden terminar y otros no. En el Capítulo 7 describiremos una estrategia de reducción, llamada *evaluación perezosa*, que garantiza la terminación siempre que esto sea posible y resulte también razonablemente eficiente. Haskell es un lenguaje funcional perezoso y exploraremos las consecuencias de adaptar tal estrategia en el resto del libro. Sin embargo, cualquiera que sea la estrategia utilizada, lo esencial es que las expresiones se evalúan mediante un proceso de sustitución y simplificación conceptualmente simple, usando tanto reglas predefinidas como reglas suministradas por el programador en forma de definiciones.

Ejercicios

1.2.1 Para evaluar $x \times y$, se reducen las expresiones x e y a sus repectivas formas normales y después se lleva a cabo la evaluación. ¿Termina la evaluación de *cuadrado infinito*?

1.2.2 ¿Cuántas secuencias de reducción existen para la expresión *cuadrado* (3 + 4)?

1.2.3 Imaginemos un lenguaje de expresiones para representar enteros definidos por las reglas sintácticas: (i) *cero* es una expresión; (ii) si e es una expresión, entonces

también lo son *suc* (*e*) y *pred* (*e*). Un evaluador reduce expresiones de este lenguaje aplicando las reglas siguientes repetidamente hasta que no pueden aplicarse más:

$$\begin{array}{lcl} suc\ (pred(e)) & = & e \\ pred\ (suc(e)) & = & e \end{array}$$

Simplificar la expresión *suc* (*pred* (*suc*(*pred* (*pred* (*cero*))))).

¿De cuántas maneras se pueden aplicar las reglas de reducción a esta expresión? ¿Conducen todas ellas al mismo resultado final? Demostrar que el proceso de reducción ha de terminar para todas las expresiones dadas. (*Sugerencia:* Definir una noción apropiada de tamaño de una expresión y demostrar que cada paso de reducción reduce efectivamente el tamaño).

1.2.4 Continuando con el ejercicio anterior, supongamos que se añade una regla sintáctica adicional al lenguaje: (iii) si e_1 y e_2 son expresiones, entonces también lo es *suma* (e_1, e_2). Las reglas de reducción correspondientes son:

$$\begin{array}{lcl} suma\ (cero, e_2) & = & e_2 \\ suma\ (suc\ (e_1), e_2) & = & suc\ (suma\ (e_1, e_2)) \\ suma\ (pred\ (e_1), e_2) & = & pred\ (suma\ (e_1, e_2)) \end{array}$$

Simplificar la expresión *suma* (*suc* (*pred* (*cero*)), *cero*).

Contar de cuántos modos diferentes se pueden aplicar las reglas de reducción a la expresión anterior. ¿Conducen todos ellos al mismo resultado final?

1.2.5 Ahora supongamos que definimos el tamaño de una expresión mediante las reglas siguientes:

$$\begin{array}{lcl} tam\ (cero) & = & 1 \\ tam\ (suc\ (e)) & = & 1 + tam\ (e) \\ tam\ (pred\ (e)) & = & 1 + tam\ (e) \\ tam\ (suma\ (e_1, e_2)) & = & 1 + 2 \times (tam\ (e_1) + tam\ (e_2)) \end{array}$$

Mostrar que la aplicación de cualquiera de las cinco reglas de reducción dadas más arriba reduce el tamaño de la expresión. ¿Por qué demuestra esto que el proceso de reducción debe terminar siempre para cualquier expresión inicial dada?

1.3 Valores

En la programación funcional, como en las matemáticas, una expresión se usa exclusivamente para describir (o *denotar*) un *valor*. Entre las clases de valores que una expresión puede denotar, se incluyen: números de varios tipos, valores booleanos, caracteres, tuplas, funciones y listas. Todos ellos serán descritos en su momento. Como también veremos, se pueden introducir nuevas clases de valores y definir operaciones para generarlos y manipularlos.

Es importante distinguir entre los valores y su representación mediante expresiones. La forma equivalente más simple de una función, cualquiera que sea ésta, *no* es un valor sino una representación del mismo. En algún lado, quizá en el espacio exterior, se podría imaginar un universo de valores abstractos, pero en la Tierra éstos pueden

ser reconocidos y manipulados exclusivamente mediante representaciones concretas. Pueden existir muchas representaciones para un mismo valor. Por ejemplo, el número abstracto cuarenta y nueve puede ser representado mediante el número decimal 49, el número romano XLIX, o la expresión 7×7. Los computadores operan habitualmente con la representación binaria de los números, en la cual cuarenta y nueve se representa mediante un cierto patrón de dígitos binarios consistente en un número arbitrario de ceros seguido por 110001.

El evaluador de un lenguaje funcional muestra un valor empleando su representación canónica; esta representación depende tanto de la sintaxis con la que se forman las expresiones, como de la definición precisa de las reglas de reducción.

Algunos valores no tienen representación canónica por ejemplo, los valores funcionales. Es difícil imaginar una representación canónica para la función *sqrt* :: *Float* → *Float*; se puede describir esta función de diversos modos, pero no se puede considerar ninguna de las descripciones como canónica. Otros valores pueden tener representaciones razonables, pero ninguna de ellas finita. Por ejemplo, el número π no tiene una representación decimal finita. Se puede pedir a un computador que presente la expansión decimal de π dígito a dígito, pero el proceso nunca terminará.

Para algunas expresiones, el proceso de reducción nunca termina y nunca produce un resultado. Por ejemplo, la expresión *infinito*, definida en la sección precedente, conduce a una secuencia de reducción infinita. Recordemos que la definición era

$$\begin{array}{lcl} infinito & :: & Integer \\ infinito & = & infinito + 1 \end{array}$$

Tales expresiones no denotan valores bien definidos en el sentido matemático normal. Por poner otro ejemplo, suponiendo que el operador / denota la división numérica y devuelve un número de tipo *Float*, la expresión 1/0 no denota un número en coma flotante bien definido. La petición de evaluar 1/0 puede hacer que el evaluador responda con un mensaje de error tal como "intento de dividir por cero" o que inicie una secuencia infinita de cálculos, sin producir ningún resultado.

Para que podamos decir que, sin excepción, toda expresión sintácticamente bien formada denota un valor, es conveniente introducir un símbolo especial $\perp$, pronunciado "bottom"[2], que representa el valor indefinido del tipo bajo consideración. En particular, el valor de *infinito* es el valor indefinido $\perp$ de tipo *Integer* y 1/0 es el valor indefinido $\perp$ de tipo *Float*. De aquí que podamos afirmar que $1/0 = \perp$.

No se debe esperar que el computador sea capaz de producir el valor $\perp$. Ante una expresión cuyo valor es $\perp$, el computador puede, o bien dar un mensaje de error, o bien permanecer en silencio indefinidamente. La primera situación es detectable, pero la segunda no lo es (después de todo, la evaluación podría haber terminado normalmente al instante siguiente de que el programador decidiera abortarla). Así, $\perp$ es un tipo especial de valor bastante similar al valor especial ∞ del cálculo matemático. Igual que los valores especiales manejados en otras ramas de las matemáticas, $\perp$ puede admitirse en el universo de valores únicamente si establecemos con precisión qué propiedades se requiere que tenga y su relación con otros valores.

[2] N. del T. La traducción literal de *bottom* es "parte inferior". El nombre hace referencia a que, en la semántica habitual de un tipo de datos, $\perp$ es el valor más pequeño posible, según un cierta relación de orden, del conjunto de valores pertenecientes a dicho tipo.

Es posible, conceptualmente al menos, aplicar funciones a ⊥. Por ejemplo, con las definiciones *tree* x = 3 y *cuadrado* x = x × x, tenemos

? *tres infinito*
3
? *cuadrado infinito*
{Interrupted!}

En la primera evaluación, no hacía falta el valor de *infinito* para completar el cálculo, por tanto, no fue calculado. Esto se debe al empleo de la estrategia de reducción perezosa mencionada anteriormente. Por otra parte, en la segunda evaluación se necesitó el valor de *infinito* para completar el cálculo: no es posible calcular $x \times x$ sin saber el valor de x. Por tanto, el evaluador entró en una secuencia de reducción infinita al intentar obtener la forma normal de *infinito*. Aburridos de esperar una respuesta que sabemos nunca llegará, hemos pulsado la tecla de interrupción.

Si $f\perp = \perp$, entonces se dice que f es una función *estricta*; en otro caso, se dice que es *no estricta*. Así *cuadrado* es una función estricta, mientras que *tres* es no estricta. La evaluación perezosa permite definir funciones no estrictas, mientras que otras estrategias no lo permiten.

Ejercicios

1.3.1 Supongamos que definimos *multiplicar* mediante

```
multiplicar        :: (Integer, Integer) → Integer
multiplicar (x, y) =  if x==0 then 0 else x × y
```

El símbolo == se usa para comprobar la igualdad entre dos enteros. Supongamos que la evaluación de $e_1 == e_2$ comienza por reducir e_1 y e_2 a forma normal y continúa comprobando si ambos resultados son idénticos. Bajo evaluación perezosa, ¿cuál podría ser el valor de *multiplicar* (0, *infinito*)? ¿Cuál sería el valor de *multiplicar* (*infinito*, 0)?

1.3.2 Supóngase que definimos la función h mediante la ecuación $h\,x = f\,(g\,x)$. Mostrar que si tanto f como g son estrictas, entonces también h lo es.

1.4 Funciones

Por supuesto, el tipo más importante de valor en la programación funcional es la función. Aunque no podemos mostrar en pantalla un valor funcional, podemos aplicar funciones a argumentos y mostrar los resultados (suponiendo, por supuesto, que dichos resultados puedan ser mostrados). Desde el punto de vista matemático, una función f es una regla de correspondencia que asocia, a cada elemento de un tipo A determinado, con un único elemento de un segundo tipo B. Al tipo A se le llama el tipo *fuente* y a B se le llama el tipo *resultado* de la función. Expresamos dicha información sobre la función escribiendo $f :: A \to B$. Esta fórmula establece que el tipo de f es $A \to B$. En otras palabras, la expresión de tipo $A \to B$ denota un tipo siempre que A y B lo sean

y describe el tipo de las funciones de A a B. Por ejemplo, hemos encontrado ya las funciones

```
tres      :: Integer → Integer
cuadrado  :: Integer → Integer
delta     :: (Float, Float, Float) → Float
```

La definición de *tres* describe una regla de correspondencia que asocia a cada entero, incluyendo el entero especial ⊥, el número 3. La definición de *cuadrado* asocia cada entero bien definido con su cuadrado, y asocia ⊥ con el entero indefinido ⊥.

Diremos que una función $f :: A \to B$ toma *argumentos* en A y devuelve *resultados* en B. Si x denota un elemento de A, entonces escribiremos $f(x)$, o simplemente $f\ x$, para denotar el resultado de *aplicar* la función f al argumento x. Este valor es el único elemento de B asociado con x mediante la regla de correspondencia de f. La primera notación, $f(x)$, es la que se emplea normalmente en matemáticas para denotar la aplicación de funciones, pero los paréntesis no son realmente necesarios y usaremos la segunda forma, $f\ x$, en su lugar. Por otra parte, los paréntesis son necesarios cuando el argumento no es una simple constante o variable. Por ejemplo, hemos de escribir *cuadrado* (3 + 4) (si eso es lo que queremos expresar) porque *cuadrado* 3 + 4 significa (*cuadrado* 3) + 4. Esto es así debido a que la aplicación tiene mayor *precedencia* que + (ver más abajo). Similarmente, hemos de escribir *cuadrado* (*cuadrado* 3) y no *cuadrado cuadrado* 3.

Tendremos cuidado de no confundir nunca una función con su aplicación a un argumento. En algunos textos matemáticos se encuentra frecuentemente la frase "la función $f(x)$", cuando lo que se quiere decir realmente es "la función f". En tales textos, raramente se considera a las funciones como valores que puedan ser usados, por sí mismos, como argumentos de otras funciones y la forma tradicional de expresarse no produce confusión. Sin embargo, en la programación funcional las funciones son valores, exactamente con el mismo status que cualesquiera otros valores; en particular, pueden ser pasadas como argumento a otras funciones y devueltas como resultados. De acuerdo con ello, no podemos admitir imprecisiones acerca de la diferencia entre una función y el resultado de aplicarla a un argumento.

1.4.1 Extensionalidad

Dos funciones son iguales si devuelven resultados iguales para argumentos iguales. Así, $f = g$ si y sólo si $f\ x = g\ x$ para todo x. Este principio recibe el nombre de *principio de extensionalidad*. Dice que lo importante acerca de una función es la correspondencia entre argumentos y resultados y no cómo se describe dicha correspondencia.

Por ejemplo, podemos definir la función que duplica su argumento de las dos maneras siguientes:

```
doble, doble'  :: Integer → Integer
doble x        =  x + x
doble' x       =  2 × x
```

Las dos definiciones describen diferentes *procedimientos* para obtener la correspondencia, una utiliza la suma y la otra utiliza la multiplicación, pero *doble* y *doble'* definen el mismo valor funcional y podemos considerar como verdad matemática evidente

que *doble* = *doble'*. Consideradas como procedimientos de evaluación, una definición puede ser más o menos "eficiente" que la otra, pero la noción de eficiencia no va asociada a los valores funcionales propiamente dichos. Por supuesto, esto no quiere decir que la eficiencia no sea importante; después de todo, queremos que las expresiones sean evaluadas en un tiempo razonable. Lo importante es que la eficiencia es una propiedad *intensional* (y no extensional) de las definiciones.

La extensionalidad significa que podemos demostrar $f = g$ demostrando simplemente que $f\ x = g\ x$ para todo x. Dependiendo de las definiciones de f y g, podemos también ser capaces de demostrar $f = g$ directamente. Al primer estilo de demostración se le llama *aplicativo* o *punto a punto*, mientras que al segundo se le llama estilo *no aplicativo*. Veremos ejemplos de ambos estilos a lo largo de este libro.

1.4.2 Currificación

Un artificio útil para reducir el número de paréntesis de una expresión consiste en reemplazar un argumento estructurado por una secuencia de argumentos más simples. Para ilustrar esta idea, consideremos otra vez la función *minimo* definida anteriormente:

```
minimo        :: (Integer, Integer) → Integer
minimo (x, y) =  if x ≤ y then x else y
```

La función *minimo* toma un argumento simple consistente en un par de enteros y devuelve un entero. Otra definición, esencialmente equivalente, de la misma función es:

```
minimoc       :: Integer → (Integer → Integer)
minimoc x y   =  if x ≤ y then x else y
```

La función *minimoc* toma dos argumentos, uno después del otro. Más precisamente, *minimoc* es una función que toma un entero x como argumento y devuelve una función *minimoc* x; la función *minimoc* x toma un entero y como argumento y devuelve un entero, a saber el menor entre x e y.

He aquí otro ejemplo:

```
suma          :: (Integer, Integer) → Integer
suma (x, y)   =  x + y

sumac         :: Integer → (Integer → Integer)
sumac x y     =  x + y
```

Para cada entero x la función *sumac* x añade x a un entero. En particular, *sumac* 1 es la función sucesora que incrementa su argumento en 1, y *sumac* 0 es la función identidad sobre los enteros.

Este simple artificio, que permite reemplazar argumentos estructurados por una secuencia de argumentos más simples, se conoce como *currificación*, en memoria del lógico americano Haskell B. Curry (en honor de quien el lenguaje de programación Haskell toma su nombre). Para que la currificación funcione de un modo consistente, necesitamos que la operación de aplicación funcional asocie a la izquierda en las expresiones. Así,

minimoc 3 4	significa	(*minimoc* 3) 4
sumac x y	significa	(*sumac x*) *y*
cuadrado cuadrado 3	significa	(*cuadrado cuadrado*) 3

Aunque *cuadrado cuadrado* 3 es sintácticamente una expresión legal, no tiene sentido porque *cuadrado* toma un argumento entero simple y no una función seguida de un entero. De hecho, la expresión será rechazada por el computador porque no puede asignarle un tipo adecuado. Volveremos sobre este tema en una sección posterior.

La currificación de funciones conlleva dos ventajas. En primer lugar, la currificación puede ayudar a reducir el número de paréntesis que han de escribirse en las expresiones. En segundo lugar, las funciones currificadas pueden ser aplicadas a un solo argumento, dando como resultado otra función que puede ser útil por sí misma. Por ejemplo, consideremos la función *twice* que aplica dos veces una función dada:

```
twice       :: (Integer → Integer) → (Integer → Integer)
twice f x   =  f (f x)
```

Ésta es una definición perfectamente legítima en Haskell. El primer argumento de *twice* es una función (de tipo *Integer* → *Integer*) y el segundo argumento es un entero. Al aplicar *twice* al primer argumento f, obtenemos una función *twice* f que aplica f dos veces. Podemos definir, por ejemplo,

```
cuarta   :: Integer → Integer
cuarta   =  twice cuadrado
```

La función *cuarta* eleva su argumento a la cuarta potencia. Supongamos, por otro lado, que hubiéramos definido *twice* mediante

```
twice       :: (Integer → Integer, Integer) → Integer
twice (f, x) =  f (f x)
```

Ahora no hay manera de nombrar la función que aplica una función dos veces sin mencionar también el argumento al cual se aplica la segunda función. En vez de decir "*cuarta*, donde *cuarta* = *twice cuadrado*", hubiéramos tenido que decir "*cuarta*, donde *cuarta x* = *twice* (*cuadrado*, *x*) para todo *x*". El segundo estilo es más enrevesado.

Si lo deseamos, siempre podemos convertir una función no currificada en otra currificada. La función *curry* toma una función no currificada y devuelve una versión currificada de la misma función; su definición es

```
curry       :: ((α, β) → γ) → (α → β → γ)
curry f x y =  f (x, y)
```

La signatura de tipo de *curry* será explicada en la sección 1.6. Nótese que *curry* es en sí misma un ejemplo de función currificada: *curry* toma tres argumentos, uno después del otro. Ahora podremos referirnos a *curry f* como la versión currificada de f. Por ejemplo, *sumac* = *curry suma*.

Se deja como ejercicio el definir una función *uncurry* que hace lo opuesto y convierte una función currificada en otra que no lo es.

1.4.3 Operadores

Algunas funciones se escriben entre sus (dos) argumentos en lugar de precederlos. Por ejemplo, escribimos

$3 + 4$ en lugar de *sumac* 3 4
$3 \leq 4$ en lugar de *menorIgual* 3 4

A una función escrita usando la notación infija se le llama un *operador*. Para eliminar la ambigüedad se emplean símbolos especiales para denotar operadores. Ocasionalmente, utilizaremos nombres en lugar de símbolos para los operadores, pero los escribiremos en negrita. Por ejemplo, escribiremos (15 **div** 4) y (15 **mod** 4), usando los nombres en negrita para los operadores asociados con la división y resto enteros. El convenio Haskell consiste en encerrar el nombre entre comillas inversas; por ejemplo, en Haskell escribiríamos `(15 ‘div‘ 4)` y `(15 ‘mod‘ 4)`.

Al encerrar un operador entre paréntesis, lo convertimos en una función prefija currificada que puede aplicarse a sus argumentos como cualquier otra función. Por ejemplo,

$$(+)\ 3\ 4 = 3 + 4$$
$$(\leq)\ 3\ 4 = 3 \leq 4$$

En particular, *sumac* = (+). Como cualquier otro nombre, un operador encerrado entre paréntesis puede usarse en expresiones y ser pasado como argumento a funciones. Por ejemplo,

$$suma = uncurry\ (+)$$

introduce *suma* como otro nombre para la versión no currificada de la suma.

1.4.4 Secciones

El artificio notacional de encerrar un operador entre paréntesis para convertirlo en una función prefija normal puede ser extendido algo más: se puede encerrar también un argumento junto con el operador. Si $\oplus$ denota un operador binario, entonces $(x \oplus)$ y $(\oplus\ x)$ son funciones con las siguientes definiciones:

$$(x \oplus)\ y = x \oplus y$$
$$(\oplus\ y)\ x = x \oplus y$$

Estas dos formas reciben el nombre el nombre de *secciones*. Por ejemplo,

$(\times\ 2)$	es la función "duplicar"
$(>\ 0)$	es la comprobación de "número positivo"
$(1\ /)$	es la función "inverso"
$(/\ 2)$	es la función "dividir por dos"
$(+\ 1)$	es la función "sucesor"

Hay una excepción a la regla de formación de secciones: $(-\ x)$ se interpreta como la aplicación del operador unario de negación al número x. Las secciones no se usarán demasiado en lo sucesivo, pero, ocasionalmente, suministrarán un método simple, conveniente y claro de describir expresiones.

1.4.5 Precedencia

Cuando varios operadores aparecen juntos en una expresión, se aplican ciertas reglas de *precedencia* para resolver posibles ambigüedades. Las reglas de precedencia para los operadores aritméticos usuales se han aprendido en la infancia, incluso sin haber sido presentadas formalmente. Su único propósito en la vida es reducir el número de paréntesis de una expresión.

En particular, la exponenciación, que será denotada mediante $\uparrow$, tiene precedencia sobre la multiplicación, que a su vez la tiene sobre la suma; por ejemplo,

$?\ 1 + 3 \uparrow 4 \times 2$
163

Así, $1 + 3 \uparrow 4 \times 2 = 1 + ((3 \uparrow 4) \times 2)$. Más aún, la aplicación funcional, es decir, el operador que se simboliza mediante un espacio en blanco, tiene precedencia sobre cualquier otro operador. Por ejemplo, *cuadrado* 3 + 4 significa (*cuadrado* 3) + 4.

1.4.6 Asociación

Otro mecanismo para reducir paréntesis es suministrar un orden de *asociación* para cada operador. Está claro que cuando el mismo operador aparece dos veces en sucesión, la regla de precedencia no es suficiente para resolver la ambigüedad. Los operadores pueden asociar a la *izquierda* o a la *derecha*. Hemos encontrado ya un ejemplo de declaración de tal precedencia: la aplicación funcional asocia a la izquierda en las expresiones. En la aritmética, los operadores del mismo nivel de precedencia se declaran usualmente como asociativos por la izquierda. Así, $5 - 4 - 2$ significa $(5 - 4) - 2$ y no $5 - (4 - 2)$. Un operador que asocia a la derecha es el operador de tipo funcional $(\rightarrow)$; así,

$$A \rightarrow B \rightarrow C \quad \text{significa} \quad A \rightarrow (B \rightarrow C)$$

No es necesario insistir en la necesidad de prescribir un orden de asociación para cada operador. Si no se indica preferencia alguna, entonces se deben usar los paréntesis para evitar la ambigüedad. De hecho, para no complicar más esta idea, básicamente simple, usaremos siempre paréntesis para eliminar la ambigüedad en secuencias de operadores diferentes con la misma precedencia.

Una declaración de un orden específico de asociación no ha de confundirse con otra propiedad de los operadores, diferente aunque relacionada, conocida como *asociatividad*. Se dice que un operador $\oplus$ es asociativo si

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

para todos los valores x, y y z del tipo apropiado. Por ejemplo, + y $\times$ son operadores asociativos. Para tales operadores, la elección de un orden de asociación no tiene efecto alguno sobre el significado.

1.4.7 Composición funcional

La composición de dos funciones f y g se denota[3] mediante $f \cdot g$ y se define mediante la ecuación

[3] N. del T. Al igual que en otras ocasiones, indicadas en el texto o en notas como ésta, el autor se desvía aquí de la sintaxis de Haskell en aras de la estética. Para la composición funcional, Haskell utiliza el punto bajo '.' en lugar del punto centrado '·'.

$$
\begin{array}{lcl}
(\cdot) & :: & (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma) \\
(f \cdot g)\ x & = & f\ (g\ x)
\end{array}
$$

La signatura de tipo será explicada en la Sección 1.6. En pocas palabras, la función $f \cdot g$ aplicada a x se define como el resultado de aplicar primero g a x, y aplicar después f al resultado. No todo par de funciones puede componerse porque los tipos tienen que encajar: requerimos que g tenga tipo $g :: \alpha \to \beta$ para ciertos tipos α y β, y que f tenga tipo $f :: \beta \to \gamma$ para algún tipo γ. Entonces obtenemos $f \cdot g :: \alpha \to \gamma$. Por ejemplo, dado *cuadrado* :: *Integer* → *Integer*, podemos definir

$$
\begin{array}{lcl}
cuarta & :: & Integer \to Integer \\
cuarta & = & cuadrado \cdot cuadrado
\end{array}
$$

Por definición de composición, esto describe exactamente la misma función *cuarta* que

$$
\begin{array}{lcl}
cuarta & :: & Integer \to Integer \\
cuarta\ x & = & cuadrado\ (cuadrado\ x)
\end{array}
$$

Este ejemplo ilustra la ventaja principal de usar la composición funcional en los programas: se pueden escribir las definiciones de un modo más conciso. Utilizar un estilo no aplicativo o un estilo punto a punto es, en parte, cuestión de gusto y veremos funciones definidas en ambos estilos en el resto del libro. Sin embargo, cualquiera que sea el estilo empleado, es una buena práctica de programación construir funciones complicadas mediante la composición de funciones más simples.

La composición funcional es una operación asociativa. Tenemos

$$(f \cdot g) \cdot h \quad = \quad f \cdot (g \cdot h)$$

para todas las funciones f, g y h de tipos apropiados. En consecuencia, no hay necesidad de poner paréntesis al escribir secuencias de composiciones.

Ejercicios

1.4.1 Supongamos que f y g tienen los tipos siguientes:

$$
\begin{array}{lcl}
f & :: & Integer \to Integer \\
g & :: & Integer \to (Integer \to Integer)
\end{array}
$$

Sea h definida como

$$
\begin{array}{lcl}
h & :: & \ldots \\
h\ x\ y & = & f\ (g\ x\ y)
\end{array}
$$

Escribir el tipo correcto para h. Determinar ahora cuál de las siguientes definiciones, caso de serlo alguna, es correcta:

$$
\begin{array}{lcl}
h & = & f \cdot g \\
h\ x & = & f \cdot (g\ x) \\
h\ x\ y & = & (f \cdot g)\ x\ y
\end{array}
$$

1.4.2 Supongamos que currificamos los argumentos de la función *delta*, de modo que escribiremos *delta a b c* en lugar de *delta* (*a*, *b*, *c*). ¿Cuál es el tipo de la versión currificada?

1.4.3 En matemáticas, se usan frecuentemente logaritmos en varias bases. Por ejemplo, $\log_2$, $\log_e$ y $\log_{10}$. Dar un tipo apropiado para una función *log* que toma como argumento una base y devuelve la función logaritmo en esa base.

1.4.4 Dar un tipo apropiado para la integral definida de una función en análisis matemático, tal como se usa en la frase "la integral de f de a a b".

1.4.5 Dar ejemplos de funciones con los tipos siguientes:

$$
\begin{array}{l}
(Integer \to Integer) \to Integer \\
(Integer \to Integer) \to (Integer \to Integer)
\end{array}
$$

1.4.6 ¿Cuál de las siguientes definiciones, caso de serlo alguna, es correcta?:

$$
\begin{array}{lcl}
(\times)\ x & = & (\times x) \\
(+)\ x & = & (x+) \\
(-)\ x & = & (-x)
\end{array}
$$

1.4.7 Definir una función *uncurry* que convierta una función currificada en su versión no currificada. Mostrar que

$$
\begin{array}{lcl}
curry\ (uncurry\ f)\ x\ y & = & f\ x\ y \\
uncurry\ (curry\ f)\ (x, y) & = & f\ (x, y)
\end{array}
$$

para todo x e y.

1.5 Definiciones

Hasta ahora, hemos visto una o dos definiciones simples de funciones, pero también es posible definir otras clases de valores. Por ejemplo,

$$
\begin{array}{lcl}
pi & :: & Float \\
pi & = & 3.14159
\end{array}
$$

define una aproximación de π como un número en coma flotante de precisión simple.

La definición de *minimo* vista anteriormente hacía uso de una *expresión condicional*. Recuérdese que su definición era

$$
\begin{array}{lcl}
minimo & :: & (Integer, Integer) \to Integer \\
minimo\ (x, y) & = & \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y
\end{array}
$$

La condición $x \le y$ se evalúa a un valor *booleano*, o valor de verdad, *True* o *False*. Los valores booleanos se considerarán en detalle en el próximo capítulo.

Otro modo de expresar esencialmente la misma definición de *minimo* es escribir

$$
\begin{array}{l}
minimo :: (Integer, Integer) \to Integer \\
minimo\ (x, y) \\
\quad \left| \begin{array}{lcl} x \le y & = & x \\ x > y & = & y \end{array} \right.
\end{array}
$$

Esta forma de definición utiliza *ecuaciones con guardas*. La sintaxis de las ecuaciones con guardas ha sido tomada de Haskell y consiste en una secuencia de *cláusulas* delimitadas por una barra vertical. Cada cláusula consiste en una condición o *guarda* y en una expresión separada de la guarda por un signo de igualdad (=). En la definición de *minimo* (x, y) las guardas son $(x \leq y)$ y $(x > y)$, mientras que las expresiones asociadas son x e y.

En adelante usaremos ecuaciones con guardas sólo de vez en cuando, prefiriendo en su lugar las expresiones condicionales. La principal ventaja de las ecuaciones con guardas aparece cuando hay tres o más cláusulas en una definición. Para ilustrarlo, consideremos la función *signo* que toma un argumento entero x y devuelve -1 si x es negativo, 0 si x es cero y 1 si x es positivo. Usando ecuaciones con guardas escribiríamos

```
signo :: Integer → Integer
signo x
  | x < 0  =  -1
  | x == 0 =  0
  | x > 0  =  1
```

Mediante expresiones condicionales, habríamos escrito algo como

```
signo   :: Integer → Integer
signo x =  if x < 0 then -1 else
           if x == 0 then 0 else 1
```

La definición que emplea ecuaciones con guardas es más clara porque se hacen explícitas las tres condiciones. Nótese que la comprobación de igualdad se escribe como $x == y$. Se hace así para distinguirla de una ecuación de definición, que se escribe de la forma $x = y$. La igualdad y otras comparaciones se consideran más adelante, en el siguiente capítulo.

1.5.1 Definiciones recursivas

Las definiciones pueden ser también *recursivas*. He aquí un ejemplo bien conocido:

```
fact   :: Integer → Integer
fact n =  if n == 0 then 1 else n × fact (n − 1)
```

La función *fact* es la función factorial. Las definiciones recursivas se evalúan como cualquier otra definición. Por ejemplo, una secuencia de reducción para evaluar *fact* 1 es


```
    fact 1
=     {definición de fact}
    if 1 == 0 then 1 else 1 × fact (1 − 1)
=     {puesto que (1 == 0) se evalúa a False}
    1 × fact (1 − 1)
=     {definición de fact}
    1 × ( if (1 − 1 ) == 0 then 1 else (1 − 1) × fact ((1 − 1) − 1))
=     {definición de (−)}
    1 × ( if 0 == 0 then 1 else (1 − 1) × fact ((1 − 1) − 1))
=     {puesto que (0 == 0) se evalúa a True}
    1 × 1
=     {definición de ×}
    1
```

Hay, por supuesto, otras secuencias de reducción de *fact* 1, pero todas conducen al mismo resultado.

La definición anterior de *fact* no es completamente satisfactoria: si aplicamos *fact* a un entero negativo, entonces el cómputo nunca termina. Por ejemplo,

```
    fact (−1)
=     {definición de fact}
    if −1 == 0 then 1 else (−1) × fact (− 1 − 1)
=     {puesto que (−1 == 0) se evalúa a False}
    (−1) × fact (− 1 − 1)
=     {mismo razonamiento}
    (−1) × ((−2) × fact (− 1 − 1 − 1))
```

y así sucesivamente. Si bien es cierto que *fact* $x = \perp$ para valores negativos de x, hubiéramos preferido que el cómputo terminase con un mensaje de error apropiado, en vez de continuar indefinidamente realizando cómputos inútiles. Una manera de conseguir esto, es reescribir la definición del modo siguiente:

```
fact :: Integer → Integer
fact n
  | n < 0  =  error "argumento negativo para fact"
  | n == 0 =  1
  | n > 0  =  n × fact (n − 1)
```

La función predefinida *error* toma una cadena de caracteres como argumento; cuando se evalúa, provoca la terminación inmediata del programa y en la pantalla se muestra el mensaje de error dado:

```
? fact (−1)
Program error: argumento negativo para fact
```

Hay otras maneras de definir *fact* que discutiremos más adelante en el libro.

1.5.2 Definiciones locales

La última construcción sintáctica que introduciremos aquí se llama definición *local*. En las descripciones matemáticas es frecuente encontrar una expresión cualificada por

una frase de la forma "donde ...". Por ejemplo, podríamos encontrar "$f(x, y) = (a + 1)(a + 2)$, donde $a = (x + y)/2$". Se puede utilizar el mismo artificio en una definición formal:

```
f        :: (Float, Float) → Float
f (x, y) =  (a + 1) × (a + 2) where a = (x + y) / 2
```

La palabra clave **where** se usa para introducir una definición local cuyo contexto (o *ámbito*[4]) es la expresión en la parte derecha de la definición de f.

Cuando hay dos o más definiciones locales, podemos disponerlas siguiendo dos estilos diferentes. Por ejemplo, se podría escribir

```
f        :: (Float, Float) → Float
f (x, y) =  (a + 1) × (b + 2)
            where a  =  (x + y) / 2
                  b  =  (x + y) / 3
```

Pero también podríamos escribir

```
f        :: (Float, Float) → Float
f (x, y) =  (a + 1) × (b + 2)
            where a = (x + y) / 2 ; b = (x + y) / 3
```

En el segundo estilo, se usa un punto y coma para separar las dos definiciones locales. Una definición local se puede usar conjuntamente con una definición que usa ecuaciones con guardas. Consideremos la siguiente definición:

```
f :: Integer → Integer → Integer
f x y
    | x ≤ 10  =  x + a
    | x > 10  =  x − a
    where a = cuadrado (y + 1)
```

En esta definición, la cláusula **where** cualifica a *ambas* ecuaciones con guardas.

Ejercicios

1.5.1 Los números de Fibonacci $f_0, f_1, \ldots$ se definen mediante la regla $f_0 = 0$, $f_1 = 1$ y $f_{n+2} = f_n + f_{n+1}$ para todo $n \geq 0$. Dar una definición de la función *fib* que toma un entero n y devuelve f_n.

1.5.2 Definir una función *abs* :: *Integer* → *Integer* que devuelva el valor absoluto de un entero.

[4]N. del T. En inglés, *scope*. Otros autores emplean *alcance* como traducción.

1.6 Tipos

En la programación funcional, el universo de valores está dividido en colecciones organizadas llamadas *tipos*. Hasta ahora, hemos mencionado *Integer* y *Float*, pero hay también otras clases de números que incluyen *Int* y *Double*, así como booleanos (elementos de *Bool*), caracteres (elementos de *Char*), listas, árboles, y otros. Más aún, hemos visto ya cómo combinar tipos entre sí para construir una variedad infinita de otros tipos; por ejemplo, *Integer* → *Float*, (*Float*, *Float*), etc. En el próximo capítulo veremos cómo se pueden definir algunos de estos tipos y cómo definir tipos nuevos.

Cada tipo tiene asociadas ciertas operaciones que no son significativas para otros tipos. Por ejemplo, no es razonable sumar un número con un carácter o multiplicar dos funciones entre sí. Un principio importante de muchos lenguajes de programación es que a toda expresión bien formada se le puede asignar un tipo. Más aún, este tipo se puede deducir de las partes constituyentes de la expresión. En otras palabras, así como el valor de una expresión depende únicamente de los valores de sus expresiones constituyentes, ocurre lo mismo con su tipo. Este principio se denomina *disciplina de tipos*.

La principal consecuencia de la disciplina de tipos es que se considera que las expresiones a las cuales no se les puede asignar un tipo no están bien formadas y, por tanto, son rechazadas por el computador sin intentar evaluarlas. Tales expresiones se consideran simplemente ilegales. Vimos un ejemplo anteriormente: la expresión *cuadrado cuadrado* 3 es rechazada por el computador por no estar bien formada. Análogamente, el guión

```
cuarta     :: Integer → Integer
cuarta x   =  cuadrado cuadrado x
```

sería rechazado por el computador porque la expresión *cuadrado cuadrado x* no está bien formada.

Una gran ventaja de la disciplina de tipos es que permite que una gran cantidad de errores, desde simples erratas tipográficas a definiciones complicadas, se detecten antes de la evaluación. La otra gran ventaja es que encamina al programador a una cierta disciplina de pensamiento, consistente en plantearse tipos apropiados para los valores que se están definiendo, antes de escribir las propias definiciones. En otras palabras, ceñirse a la disciplina de tipos puede ayudar significativamente al diseño de programas claros y bien estructurados.

Podemos distinguir dos etapas de análisis cuando se somete una expresión a evaluación. Primero, se comprueba si la sintaxis de la expresión es correcta; si no lo es, el computador señala *error de sintaxis*. Si la expresión es sintácticamente correcta, se analiza para ver si tiene un tipo correcto. Si la expresión no supera esta comprobación, el computador señala *error de tipos*. Solamente si la expresión pasa con éxito ambas etapas, puede comenzar el proceso de evaluación. Similares observaciones se aplican a las definiciones que forman parte de un guión.

1.6.1 Tipos polimórficos

Algunas funciones y operaciones trabajan con muchos tipos. Por ejemplo, supongamos

```
cuadrado   :: Integer → Integer
```

$$sqrt \quad :: \quad Integer \rightarrow Float$$

Las expresiones *cuadrado* · *cuadrado* y *sqrt* · *cuadrado* son ambas correctas y tienen los tipos siguientes:

$$\begin{array}{lll} cuadrado \cdot cuadrado & :: & Integer \rightarrow Integer \\ sqrt \cdot cuadrado & :: & Integer \rightarrow Float \end{array}$$

Sin embargo, el operador de composición funcional tiene tipos diferentes en cada una de estas expresiones, a saber

$$\begin{array}{lll} (\cdot) & :: & (Integer \rightarrow Integer) \rightarrow (Integer \rightarrow Integer) \rightarrow (Integer \rightarrow Integer) \\ (\cdot) & :: & (Integer \rightarrow Float) \rightarrow (Integer \rightarrow Integer) \rightarrow (Integer \rightarrow Float) \end{array}$$

Así, a la operación de composición funcional se le asignan tipos diferentes en expresiones diferentes.

El problema de asignar un tipo único a $(\cdot)$ se resuelve introduciendo *variables de tipo*. El tipo asignado a $(\cdot)$ es

$$(\cdot) :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$

Aquí, α, β y γ denotan variables de tipo. Usaremos letras griegas para denotar variables de tipo. Como cualquier otra variable, una variable de tipo puede ser concretada a tipos diferentes en circunstancias diferentes. Un tipo que contenga variables de tipo se denomina *tipo polimórfico*.

He aquí otro ejemplo. Veamos de nuevo la definición de *fact*:

$$\begin{array}{l} fact :: Integer \rightarrow Integer \\ fact\ n \\ \quad \left| \begin{array}{lll} n < 0 & = & error\ \text{"argumento negativo para fact"} \\ n == 0 & = & 1 \\ n > 0 & = & n \times fact\ (n-1) \end{array} \right. \end{array}$$

Considérese la función *error*. Toma una cadena de caracteres como argumento; por tanto su tipo es $String \rightarrow A$ para algún tipo A. En el programa anterior, es claro que $A = Integer$; sólo con este tipo estaría bien formado el programa *fact*. Después de todo, la segunda y tercera cláusulas producen enteros, por tanto la primera debería hacerlo también. No importa cuál es el entero producido, porque el único propósito de evaluar *error* es abortar el cómputo con un mensaje de error. Si, por otra parte, la función general de error tuviera como tipo $error :: String \rightarrow Integer$, su utilidad sería limitada. En su lugar, el tipo asignado a *error* es $String \rightarrow \alpha$. De nuevo, el problema se resuelve usando variables de tipo.

Como último ejemplo por ahora, consideremos la función *curry* definida en la Sección 1.4.2 mediante la ecuación

$$curry\ f\ x\ y = f\ (x, y)$$

Esta función se utiliza para convertir funciones cuyo tipo es $(A, B) \rightarrow C$ en funciones de tipo $A \rightarrow B \rightarrow C$. Los tipos A, B y C en la definición de *curry* no necesitan satisfacer ningún requisito especial; por tanto se le asigna el tipo polimórfico

$$curry \quad :: \quad ((\alpha, \beta) \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta \rightarrow \gamma)$$

El par de paréntesis más a la derecha podría haberse omitido, porque el operador de tipos $(\rightarrow)$ asocia a la derecha, pero en este caso resulta más claro ponerlos.

Ahora tenemos los rudimentos de un lenguaje de expresiones que denotan tipos. Este lenguaje contiene expresiones constantes, como *Integer* o *Float*, variables como α y β, y funciones que convierten tipos en otros tipos, como el operador de tipo $(\rightarrow)$.

1.6.2 Clases de tipos

Una lectura cuidadosa de la primera parte de este capítulo revela que hemos usado multiplicación (currificada) con dos signaturas de tipo diferentes:

$$\begin{array}{lll} (\times) & :: & Integer \rightarrow Integer \rightarrow Integer \\ (\times) & :: & Float \rightarrow Float \rightarrow Float \end{array}$$

Al igual que $(\cdot)$ y *error*, parecería que a $(\times)$ debería asignársele un tipo polimórfico, por ejemplo

$$(\times) \quad :: \quad \alpha \rightarrow \alpha \rightarrow \alpha$$

Pero se puede argumentar que este tipo es demasiado general. Por ejemplo, no podríamos multiplicar dos caracteres o dos booleanos.

En Haskell, la solución consiste en agrupar ciertos tipos en *clases de tipos*. En particular, *Integer* y *Float* pertenecen a la misma clase, la clase de los números. El tipo asignado a $(\times)$ es

$$(\times) \quad :: \quad Num\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

La parte derecha debería leerse como el tipo $\alpha \rightarrow \alpha \rightarrow \alpha$ restringido a aquellos α que son concreciones[5] de la clase de tipos *Num*.

Se utiliza el mismo mecanismo para las constantes numéricas. Por ejemplo, 3 se puede usar para describir un cierto número en coma flotante, así como un entero. En consecuencia, el tipo asignado a 3 es $Num\ \alpha \Rightarrow \alpha$. Dicho con palabras, cualquier tipo, con tal de que sea un tipo numérico.

Existen otras clases de tipos, además de los numéricos. Por ejemplo, están los tipos cuyos valores pueden ser mostrados en pantalla, los tipos cuyos valores pueden ser comparados por igualdad, los tipos cuyos valores se pueden enumerar, y así sucesivamente. Un tipo concreción de una clase de tipos, podría ser también concreción de otra. Por ejemplo, podemos comparar números por igualdad y también podemos mostrarlos en pantalla. En el próximo capítulo explicaremos cómo se pueden definir clases de tipos y cómo se puede declarar que ciertos tipos específicos son concreciones de esas clases.

Ejercicios

1.6.1 Dar tipos polimórficos apropiados para las siguientes funciones:

[5] N. del T. Utilizaremos la palabra *concreción*, como traducción de *instance* y el verbo *concretar*, como traducción de *to instantiate*. Por desgracia, están muy extendidas las traducciones *instancia* e *instanciar* respectivamente para estos términos. En el primer caso, la palabra es española pero sus significados —cada uno de los escalones del aparato judicial; solicitud— son totalmente ajenos al significado de la palabra inglesa. En el segundo, el verbo ni tan siquiera existe en español.

$$
\begin{aligned}
\mathit{const}\ x\ y &= x \\
\mathit{subst}\ f\ g\ x &= f\ x\ (g\ x) \\
\mathit{aplicar}\ f\ x &= f\ x \\
\mathit{flip}\ f\ x\ y &= f\ y\ x
\end{aligned}
$$

1.6.2 Definir una función *swap* de modo que

$$\mathit{flip}\ (\mathit{curry}\ f) = \mathit{curry}\ (f \cdot \mathit{swap})$$

para todo $f :: (\alpha, \beta) \to \gamma$.

1.6.3 ¿Pueden encontrarse tipos polimórficos para las siguientes funciones?

$$
\begin{aligned}
\mathit{rara}\ f\ g &= g\ (f\ g) \\
\mathit{masRara}\ f &= f\ f
\end{aligned}
$$

1.6.4 Encontrar un tipo polimórfico para

$$\mathit{cuadrado}\ x = x \times x$$

1.7 Especificaciones

En programación, una *especificación* es una descripción de la tarea que un programa ha de realizar, mientras que una *implementación* es un programa que satisface la especificación. Las especificaciones y las implementaciones tienen objetivos diferentes: las especificaciones expresan la intención del programador (o las expectativas del cliente) y su propósito es ser lo más claras posible; las implementaciones son expresiones pensadas para ser ejecutadas por el computador y su objetivo es ser lo suficientemente eficientes como para ser ejecutadas en el tiempo y espacio disponibles. La conexión entre ambas es el requisito de que las implementaciones satisfagan, o *cumplan*, la especificación y el programador puede estar obligado a *demostrar* que ello es realmente así.

Una especificación para una función es alguna afirmación sobre la relación pretendida entre argumentos y resultados. Damos un ejemplo simple mediante la siguiente especificación de una función *aumentar*:

$$
\begin{aligned}
\mathit{aumentar} &:: \mathit{Integer} \to \mathit{Integer} \\
\mathit{aumentar}\ x &> \mathit{cuadrado}\ x, \text{ siempre que } x \geq 0
\end{aligned}
$$

Esta especificación dice que el resultado de *aumentar* debería ser estrictamente mayor que el cuadrado de su argumento, siempre que el argumento sea no negativo. La especificación no dice cómo debería calcularse *aumentar*, tan sólo expresa una propiedad que cualquier implementación debe cumplir. La especificación *no* es parte de nuestro lenguaje de programación, aunque se expresa de forma similar.

Una implementación posible es definir $\mathit{aumentar}\ x = \mathit{cuadrado}\ (x + 1)$. La demostración de que esta definición satisface su especificación, es la siguiente:

$$
\begin{aligned}
& \mathit{aumentar}\ x \\
= \quad & \{\text{definición de } \mathit{aumentar}\} \\
& \mathit{cuadrado}\ (x + 1) \\
= \quad & \{\text{definición de } \mathit{cuadrado}\} \\
& (x + 1) \times (x + 1) \\
= \quad & \{\text{álgebra}\} \\
& x \times x + 2 \times x + 1 \\
> \quad & \{\text{puesto que } x \geq 0 \text{ implica } 2 \times x + 1 > 0\} \\
& x \times x \\
= \quad & \{\text{definición de } \mathit{cuadrado}\} \\
& \mathit{cuadrado}\ x
\end{aligned}
$$

En el resto del libro, seguiremos el formato de demostración utilizado aquí. De hecho, ya lo hemos usado al estudiar las secuencias de reducción. Una secuencia de reducción es también una especie de demostración, si bien se lleva a cabo mediante un conjunto muy restringido de reglas de razonamiento. De hecho, es tan restringido que se puede instruir a un computador para realizar todos los pasos de una manera puramente mecánica.

Anteriormente, inventamos primero una definición de *aumentar* y después verificamos que satisfacía su especificación. Hay muchas otras funciones que satisfacen la especificación y, puesto que ése es el único requisito, todas son igualmente buenas.

Un modo de especificar una función consiste en establecer explícitamente la regla de correspondencia entre argumentos y resultados. La notación de la programación funcional puede ser muy expresiva y, a veces, una especificación de una función es de hecho un programa. En este caso se puede ejecutar la especificación directamente. Sin embargo, puede ser tan altamente ineficiente que la posibilidad de ser ejecutada será simplemente de interés teórico. A pesar de haber escrito una especificación ejecutable, el programador no queda necesariamente eximido de la tarea (o del placer) de producir una versión alternativa equivalente, pero aceptablemente eficiente.

Como ejemplo muy simple de esta idea, consideremos la especificación

$$
\begin{aligned}
\mathit{cuarta} &:: \mathit{Integer} \to \mathit{Integer} \\
\mathit{cuarta}\ x &= x \times x \times x \times x
\end{aligned}
$$

A la búsqueda de mejores definiciones, podemos calcular:

$$
\begin{aligned}
& \mathit{cuarta}\ x \\
= \quad & \{\text{especificación}\} \\
& x \times x \times x \times x \\
= \quad & \{\text{puesto que } \times \text{ es asociativo}\} \\
& (x \times x) \times (x \times x) \\
= \quad & \{\text{definición de } \mathit{cuadrado}\} \\
& \mathit{cuadrado}\ x \times \mathit{cuadrado}\ x \\
= \quad & \{\text{definición de } \mathit{cuadrado}\} \\
& \mathit{cuadrado}\ (\mathit{cuadrado}\ x)
\end{aligned}
$$

El resultado es que podemos implementar *cuarta* con dos multiplicaciones en lugar de con tres, un ahorro significativo utilizando la aritmética de rango arbitrario.

En este caso, no inventamos primero la implementación de *cuarta*, sino que la desarrollamos a partir de la especificación. La derivación no fue completamente mecánica:

el paso creativo consistió en emplear la asociatividad de la multiplicación para poner los paréntesis de un modo inteligente. Por supuesto, este ejemplo es absurdamente simple, pero veremos otros ejemplos más convincentes de desarrollos sistemáticos de programas, o *síntesis de programas*, más adelante en el libro.

Este paradigma de desarrollo de software —primero escribir una especificación clara y después desarrollar una implementación aceptablemente eficiente—, ha sido un foco de investigación activa durante los últimos veinte años, y no debería tomarse como un método panacea aplicable en todas circunstancias. Dos fuentes potenciales de problemas son que la especificación formal puede no satisfacer los requisitos informales del cliente y que la demostración de que la implementación satisface la especificación puede ser tan extensa que no se pueda garantizar la ausencia de errores. Sin embargo, se puede mejorar mucho la fiabilidad de los programas tratando de aplicar estos principios, siempre que podamos.

Ejercicios

1.7.1 Dar otra definición de *aumentar* que satisfaga su especificación.

1.8 Notas del capítulo

El uso interactivo de un lenguaje funcional, tal como se describe en este texto, lo proporciona el entorno HUGS (Haskell Users Gofer System) desarrollado por Mark Jones de la Universidad de Nottingham. HUGS está disponible por FTP en la dirección

```
ftp://ftp.cs.nott.ac.uk/haskell/hugs
```

Haskell, en sí mismo, no es un lenguaje interactivo. Se pueden encontrar compiladores para Haskell en las Universidades de Chalmers, Glasgow y Yale, mediante FTP a las direcciones

```
ftp://ftp.cs.chalmers.se/pub/haskell
ftp://ftp.dcs.glasgow.ac.uk/pub/haskell
ftp://nebula.cs.yale.edu/pub/haskell
```

El lenguaje usado en este libro es Haskell 1.3, aunque no se cubren todas las características de Haskell 1.3. Más aún, preferimos utilizar símbolos matemáticos en lugar de los símbolos de Haskell que utilizan un conjunto restringido de caracteres. Por ejemplo, Haskell usa * para la multiplicación. Las palabras **if**, **then**, **else** y **where** son palabras reservadas en Haskell.

Se pueden encontrar páginas Web para Haskell, incluyendo una versión interactiva del manual de Haskell 1.3, extensiones para Haskell e información acerca de implementaciones de Haskell, en la siguiente dirección:

```
http://www.haskell.org/
```

Existe una introducción didáctica a Haskell escrita por Hudak, Fasel y Peterson (1996). Otro texto elemental sobre programación funcional perezosa que utiliza Haskell 1.3 es Thompson (1996).

Mientras este texto estaba en preparación, ha aparecido una nueva versión, Haskell 1.4. Actualmente, el comité de Haskell está intentando moverse hacia una estandarización del lenguaje, Standard Haskell[6]. No es probable que los cambios que están en discusión afecten a los detalles escritos en este texto.

Otros lenguajes funcionales no estrictos son Gofer y Miranda (Miranda es una marca registrada de Research Software Ltd.). Miranda, que es bastante próximo a Haskell, está descrito en Thompson (1995) y Clack, Myers y Poon (1995). Otro lenguaje funcional popular es ML, que difiere de Haskell en que es estricto, en lugar de perezoso. ML se describe en Paulson (1996).

Para más información acerca de los aspectos denotacionales de los lenguajes de programación, consultar Stoy (1977) o Gordon (1979). La implementación de lenguajes funcionales perezosos se trata en Peyton Jones (1987) y Peyton Jones y Lester (1991). La derivación formal de programas a partir de sus especificaciones es el tema de Morgan (1996) y Kaldewaij (1990), aunque los lenguajes utilizados son imperativos y no funcionales. Para un tratamiento funcional y relacional de la derivación de programas en un marco categórico, consúltese Bird y de Moor (1997). Se trata de un texto avanzado, adecuado sólo para aquellos que estén particularmente interesados en las matemáticas de la programación, y se puede estudiar después de este texto.

[6]N. del T. En Febrero de 1999 se publicó el report de *Haskell 98*, una estandarización del lenguaje cuyo objetivo es "congelar" una versión estable de Haskell que ha de ser soportada por todos los compiladores, independientemente de que también soporten versiones posteriores. Las diferencias entre Haskell 1.4 y Haskell 98 son mínimas.

Capítulo 2

Tipos de datos simples

Este capítulo introduce algunos tipos de datos básicos, que incluyen los booleanos, caracteres, tuplas y cadenas. Describiremos cómo se representan los valores de cada tipo y daremos algunas operaciones útiles para manipularlos. También describiremos el mecanismo para declarar nuevos tipos de datos y haremos una revisión general de algunas de las principales clases de tipos en Haskell. Una clase de tipos que no se trata en este capítulo son los números en sus diversas manifestaciones. Nos ocuparemos de los números en el capítulo siguiente. Sin embargo, es difícil dar ejemplos interesantes de programas que no hagan uso de los números de un modo u otro, así es que simplemente supondremos por el momento que existen y que están disponibles.

2.1 Booleanos

Como vimos en el primer capítulo, necesitamos valores de verdad para definir una función que use expresiones condicionales. Como mínimo, nos gustaría comprobar si dos valores son iguales y, en caso de que ello tenga sentido, si un valor es menor o mayor que otro. Hay dos valores de verdad, *True* y *False*. Estos dos valores constituyen el tipo de datos *Bool* de los valores booleanos (llamados así en honor de George Boole, lógico del siglo XIX). Nótese que los nombres *True*, *False* y *Bool* empiezan todos ellos con una letra mayúscula.

El tipo de datos *Bool* se puede definir con una *declaración de tipos de datos*:

$$\textbf{data}\ Bool \quad = \quad False \mid True$$

Esta declaración denomina *Bool* al tipo de datos, y *True* y *False* a sus dos valores constituyentes. Una vez introducido *Bool*, podemos definir funciones que tomen argumentos booleanos mediante *ajuste de patrones*. Por ejemplo, la función de negación se define mediante

$$\begin{array}{lcl} not & :: & Bool \rightarrow Bool \\ not\ False & = & True \\ not\ True & = & False \end{array}$$

El computador usa estas dos ecuaciones como reglas de reescritura para simplificar expresiones de la forma *not e*. Primeramente, *e* se reduce a forma normal. Si este pro-

ceso termina y devuelve *False*, entonces se utiliza la primera ecuación; si la reducción devuelve *True*, entonces se usa la segunda ecuación. Si no se puede reducir e a forma normal, entonces el valor de *not e* es indefinido. Así, $not\ \bot = \bot$, por tanto *not* es una función estricta. Nótese que las dos apariciones de $\bot$ se refieren aquí al valor indefinido booleano. En consecuencia, no hay dos sino *tres* valores booleanos, a saber *False*, *True* y $\bot$. De hecho, cada declaración de tipo de datos introduce un valor anónimo extra, el valor indefinido de ese tipo de datos.

Dos funciones básicas sobre los booleanos son las operaciones de conjunción, denotada por el operador binario $\wedge$, y disyunción, denotada por $\vee$[1]. Estas operaciones se pueden definir mediante

$$
\begin{array}{lcl}
(\wedge), (\vee) & :: & Bool \rightarrow Bool \rightarrow Bool \\
False \wedge x & = & False \\
True \wedge x & = & x \\
 & & \\
False \vee x & = & x \\
True \vee x & = & True
\end{array}
$$

Las definiciones imponen ajuste de patrones sobre el argumento izquierdo. Por ejemplo, para simplificar expresiones de la forma $e_1 \wedge e_2$, el computador reduce primero e_1 a su forma normal. Si el resultado es *False*, se puede usar la primera ecuación de $\wedge$, de forma que el computador devuelve inmediatamente el valor final *False*. Si e_1 se reduce a *True*, se usará la segunda ecuación, de forma que e_2 se evalúa a continuación. A partir de esta descripción del funcionamiento del ajuste de patrones, se deduce que

$$
\begin{array}{lcl}
\bot \wedge False & = & \bot \\
False \wedge \bot & = & False \\
True \wedge \bot & = & \bot
\end{array}
$$

Así, $\wedge$ es estricto en su argumento izquierdo, pero no es estricto en su argumento derecho. La misma observación se aplica a $\vee$.

Podríamos haber definido $\wedge$ y $\vee$ de un modo diferente. Por ejemplo, una definición alternativa de $\wedge$ es mediante cuatro ecuaciones:

$$
\begin{array}{lcl}
False \wedge False & = & False \\
False \wedge True & = & False \\
True \wedge False & = & False \\
True \wedge True & = & True
\end{array}
$$

Esta versión impone ajuste de patrones sobre los dos argumentos, de modo que $\wedge$ sería estricto en ambos argumentos.

Con cualquiera de las definiciones, $\wedge$ es un operador asociativo, de modo que no hacen falta paréntesis cuando se escriban expresiones de la forma $x \wedge y \wedge z$. La misma observación es aplicable a $\vee$. Sin embargo, siempre es una buena práctica poner paréntesis cuando las expresiones impliquen a ambos, $\wedge$ y $\vee$. Estos operadores tienen una precedencia menor que la de los operadores de igualdad y orden (que se estudian más adelante), de modo que

$$x \leq y \wedge y < z \qquad \text{significa} \qquad (x \leq y) \wedge (y < z)$$

[1] N.T. En Haskell se emplean respectivamente los operadores && y ||. Véase la Sección 2.8.

2.1.1 Operadores de igualdad y de orden

Hay dos operadores de igualdad, == y ≠. Para argumentos booleanos se definen del modo siguiente

$$
\begin{array}{lcl}
(==) & :: & Bool \rightarrow Bool \rightarrow Bool \\
x == y & = & (x \wedge y) \vee (not\ x \wedge not\ y) \\
 & & \\
(\neq) & :: & Bool \rightarrow Bool \rightarrow Bool \\
x \neq y & = & not\ (x == y)
\end{array}
$$

Es importante observar la distinción entre == y =. El símbolo == denota una comparación computable de igualdad, mientras que = se usa tanto en definiciones como en su sentido matemático normal. En matemáticas, la afirmación *double* = *square* es falsa, y la afirmación $\bot = \bot$ es cierta, puesto que cualquier cosa es igual a sí misma. Sin embargo, no se puede comprobar la igualdad de dos funciones y, por otra parte, el resultado de evaluar $\bot == \bot$ es $\bot$, no *True*. Ello no quiere decir que el computador sea una máquina contraria a las matemáticas, lo que ocurre es que su comportamiento está descrito por un número limitado de reglas matemáticas, escogidas de tal modo que puedan ser ejecutadas mecánicamente.

El objetivo final de introducir un operador de igualdad es poder usarlo para una variedad de tipos diferentes, no simplemente con los elementos de *Bool*. En otras palabras, == y (≠) son operadores *sobrecargados*. Estos operadores se definirán de modo diferente para diferentes tipos de datos y, por tanto, el modo apropiado de introducirlos es declarar primero una clase de tipos *Eq* que conste de todos aquellos tipos para los cuales == y (≠) están definidos. La declaración de *Eq* tiene el siguiente aspecto

class *Eq* α **where**
$\quad (==), (\neq) \quad :: \quad \alpha \rightarrow \alpha \rightarrow Bool$

Esta declaración establece que la clase de tipos *Eq* contiene dos funciones miembro (o *métodos*), a saber, == y (≠). Estas funciones tienen el siguiente tipo:

$$(==), (\neq) \quad :: \quad Eq\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow Bool$$

Para declarar que un cierto tipo es una concreción de la clase de tipos *Eq* tenemos que dar una *declaración de concreción*. Por ejemplo,

instance *Eq Bool* **where**

$$
\begin{array}{lcl}
(x == y) & = & (x \wedge y) \vee (not\ x \wedge not\ y) \\
(x \neq y) & = & not\ (x == y)
\end{array}
$$

Esta declaración define que el tipo *Bool* es un tipo de la clase *Eq*, introduciendo definiciones específicas de == y (≠) para argumentos booleanos. Veremos cómo definir otras concreciones de *Eq* en secciones posteriores.

Si lo deseamos, los booleanos también se pueden comparar mediante (<), (≤), (≥) y (>). Estas operaciones también están sobrecargadas y tienen sentido cuando se usan con elementos de una variedad de tipos diferentes. Si aceptamos esto, podemos declarar otra clase de tipos *Ord*, que se define del modo siguiente:

```
class (Eq α) ⇒ Ord α where
    (<), (≤), (≥), (>)  ::  α → α → Bool
    (x ≤ y)             =   (x < y) ∨ (x == y)
    (x ≥ y)             =   (x > y) ∨ (x == y)
    (x > y)             =   not (x ≤ y)
```

La primera línea dice que *Ord* es una *subclase* de *Eq*, esto es, sólo puede dotarse con un orden a los tipos con igualdad. Esto tiene sentido puesto que, en cualquier definición razonable de concreción (≤), esperaríamos que $x == y$ si y sólo si $x \leq y$ e $y \leq x$. Las últimas tres líneas dan definiciones *por defecto* de las operaciones (≤), (≥) y (>). Haskell proporciona definiciones por defecto para ahorrarnos el trabajo de tener que repetirlas en cada declaración de concreción. Es, por tanto, suficiente con dar la definición de (<) para cada concreción. Nótese que las definiciones por defecto utilizan el hecho de que *Ord* es una subclase de *Eq*, puesto que hacen uso de la comparación de igualdad (==).

Ahora podemos declarar que *Bool* es una concreción de *Ord* escribiendo

```
instance Ord Bool where
    False < False  =  False
    False < True   =  True
    True < False   =  False
    True < True    =  False
```

La definición alternativa, a saber $x < y = not\ x \vee y$, no funciona del modo que esperaríamos (ver el Ejercicio 2.1.2).

2.1.2 Ejemplo: años bisiestos

Daremos ahora un par de ejemplos que tienen que ver con valores booleanos. Primero, supongamos que deseamos una función que determine si un año es bisiesto o no. En el calendario Gregoriano, un año bisiesto es un año que es divisible por 4, excepto si es divisible por 100, en cuyo caso debe ser también divisible por 400. Podemos expresarlo de varios modos equivalentes. Uno de ellos es definir

```
bisiesto    ::  Int → Bool
bisiesto y  =   (y mod 4 == 0) ∧ (y mod 100 ≠ 0 ∨ y mod 400 == 0)
```

El tipo *Int* describe el tipo de los enteros de rango acotado y se verá más adelante, en el capítulo siguiente. Podríamos haber usado *Integer* en lugar de *Int*, pero es improbable que deseáramos considerar años con un número muy grande de dígitos. Nótese que las tres apariciones de (==) en la definición de *bisiesto* se refieren a la comprobación de igualdad de *Int*. Los tipos numéricos, incluyendo *Int*, son también concreciones de la clase de tipos *Eq*.

Otro modo de definir *bisiesto* es usar una expresión condicional:

```
bisiesto y
  = if (y mod 100 == 0) then (y mod 400 == 0) else (y mod 4 == 0)
```

2.1.3 Ejemplo: triángulos

Supongamos ahora que queremos construir una función, llamémosla *triangulo*, que tome tres enteros positivos x, y y z en orden no decreciente. Estos números representan las longitudes de los lados de un posible triángulo. La función *triangulo* debe determinar si los tres lados constituyen de hecho un triángulo y, de ser así, si el triángulo es escaleno (tres lados diferentes), isósceles (dos lados iguales) o equilátero (los tres lados iguales). Tres lados forman un triángulo si y sólo si la longitud del lado más largo es menor que la suma de las longitudes de los otros dos lados.

Así, *triangulo* debe devolver uno de cuatro valores distintos. Podríamos usar cuatro números diferentes con este fin, pero también podemos usar cuatro valores creados especialmente para este propósito:

```
data Triangulo  =  Fallo | Isosceles | Equilatero | Escaleno
```

Esta declaración define un nuevo tipo de datos *Triangulo* con cuatro valores. Con ello, el programa sería

```
triangulo :: (Int, Int, Int)  →  Triangulo
triangulo (x, y, z)
    | x + y ≤ z          =  Fallo
    | x == z             =  Equilatero
    | (x == y) ∨ (y == z)  =  Isosceles
    | otherwise          =  Escaleno
```

El nombre *otherwise* está predefinido (en Haskell) como el valor *True*, de modo que la última condición se satisface, siempre que las condiciones precedentes devuelvan *False*. Esta definición de *triangulo* es correcta sólo bajo la suposición de que $x \leq y \leq z$. Nótese de nuevo que las apariciones de (==) y de (≤) en esta definición se refieren a los operadores de comparación sobre valores de tipo *Int*.

Ejercicios

2.1.1 Definir ∧ y ∨ usando expresiones condicionales.

2.1.2 La definición de (<) dada en el texto para argumentos de tipo *Bool* hace que *False* < *True*. Ello refleja el hecho de que *False* está antes que *True* en la declaración de valores de tipo *Bool*. Naturalmente, esperaríamos que tanto *False* < *False* como *True* < *True* se evaluaran a *False*.

Ahora, considérese la definición alternativa $(x < y) = not\ x \vee y$, donde ∨ se define mediante ajuste de patrones sobre el argumento izquierdo. ¿Por qué esta definición de (<) no es correcta?

2.1.3 En lógica, la implicación, denotada por ⇒, se define mediante la propiedad de que $x \Rightarrow y$ es falso sólo si x es cierto e y es falso. Dar una definición formal de la implicación como operación sobre *Bool*.

2.1.4 Reescribir la declaración de la clase de tipos *Eq*, dando una definición por defecto para (≠).

2.1.5 Reescribir la definición de *triangulo* de modo que los casos del análisis por casos no dependan del orden en que se han escrito.

2.1.6 Definir una función *Ordena3* que ordene tres enteros en orden creciente. A partir de ella, definir una función, llamémosla *triangulo'*, que no dependa de la suposición de que sus argumentos están en orden no decreciente.

2.1.7 ¿Cuántas ecuaciones tendríamos que escribir para definir *Triangulo* como una concreción de la clase de tipos *Ord* ? (Véase la Sección 2.3 para consultar otros modos de conseguir esto).

2.1.8 ¿Existen números que puedan ser comparados mediante (==) pero que no puedan serlo razonablemente mediante (<)?

2.1.9 La definición de (==) sobre un tipo de datos debería asegurar que esta operación es: (a) reflexiva, esto es, $x == x$ para todo x; (b) transitiva, esto es, $x == y$ e $y == z$ implica $x == z$; y (c) simétrica, esto es, $x == y$ implica $y == x$. Muéstrese que estas propiedades se cumplen en la definición del operador (==) sobre *Bool*.

2.1.10 ¿Qué propiedades esperaríamos que cumpliera (<) en cualquier declaración de concreción?

2.2 Caracteres

Los matemáticos estarían contentos si sólo les dieran números y booleanos para trabajar, pero los programadores prefieren llenar las pantallas de sus terminales símbolos de otras clases más interesantes. Con este fin, se proporciona el tipo de datos predefinido *Char* que contiene 256 caracteres, incluyendo tanto caracteres visibles como caracteres de control. Los caracteres se denotan encerrándolos entre símbolos de comillas simples, ambas cerradas[2]. Por ejemplo:

```
? 'a'
'a'
? '7'
'7'
? ' '
' '
```

Los caracteres son expresiones en forma canónica, por tanto el evaluador simplemente los muestra, incluyendo las comillas. Es importante comprender que el carácter '7' es totalmente diferente del número decimal 7: el primero denota un carácter, mientras que el segundo denota un número. El tercer ejemplo de los anteriores muestra un modo de denotar el carácter blanco. En el contexto de este libro, es conveniente introducir símbolos especiales para denotar los dos caracteres no visibles más importantes: el blanco y el fin de línea. El carácter de fin de línea será denotado mediante el signo '↴' y el blanco como '␣'.

Se proporcionan dos funciones primitivas, *ord* y *chr*, para procesar caracteres. Sus tipos son

$$\begin{array}{lcl} ord & :: & Char \to Int \\ chr & :: & Int \to Char \end{array}$$

[2]N.T. En el original, en aras de la estética, se emplea una comilla simple abierta y otra cerrada, pero en la sintaxis de Haskell ambas son cerradas. Para evitar confusiones a los programadores, hemos preferido la segunda opción.

Recordamos que *Int* describe el tipo de los enteros de rango acotado. La función *ord* convierte un carácter c a un entero $ord\ c$ en el rango $0 \le ord\ c < 256$, y la función *chr* hace lo contrario, convertir un entero al carácter que éste representa. Así $chr\ (ord\ c) = c$ para todos los caracteres c. Por ejemplo:

```
? ord 'b'
98

? chr 98
'b'

? chr (ord 'b' + 1)
'c'

? ord '↴'
10
```

La correspondencia entre los caracteres y los primeros 256 enteros proporciona un modo alternativo para describir el tipo *Char*. Se podría pensar en el tipo *Char* como el que resulta de la siguiente declaración de tipo:

```
data Char = Char0 | Char1 | ... | Char255
```

en la cual el convenio para nombrar los valores de *Char* no es estándar. Por ejemplo, en lugar de *Char*98 escribimos 'b', y en lugar de *Char*10 escribimos '↴'. El entero $ord\ c$ describe la posición del carácter c en la enumeración.

Al igual que con *Bool*, se pueden definir funciones sobre *Char* mediante ajuste de patrones, pero el número de patrones es tan grande que es preferible una definición mediante análisis por casos.

Se pueden comprobar la igualdad y el orden entre caracteres. Podemos hacer que *Char* sea un tipo con igualdad escribiendo

```
instance Eq Char where
    (x == y)  =  (ord x == ord y)
```

Esta definición utiliza el hecho de que *Int* es también un tipo con igualdad. De forma similar, podemos declarar

```
instance Ord Char where
    (x < y)  =  (ord x < ord y)
```

El orden total de los caracteres es en parte el que esperaríamos. Por ejemplo:

```
? '0' < '9'
True

? 'a' < 'z'
True

? 'A' < 'Z'
True
```

Las letras mayúsculas preceden a las letras minúsculas en la enumeración, de forma que

```
?'A' < 'a'
True
```

Usando esta información, se pueden definir funciones simples sobre caracteres. Por ejemplo, he aquí tres funciones que determinan si un carácter es un dígito, una letra minúscula o una letra mayúscula:

```
isDigit, isLower, IsUpper :: Char → Bool
isDigit c   =  ('0' ≤ c) ∧ (c ≤ '9')
isLower c   =  ('a' ≤ c) ∧ (c ≤ 'z')
isUpper c   =  ('A' ≤ c) ∧ (c ≤ 'Z')
```

Ahora, podemos definir una función que convierta minúsculas en mayúsculas:

```
aMayuscula    ::  Char → Char
aMayuscula c  =  if isLower c then chr (base + ord c) else c
                 where base = ord 'A' − ord 'a'
```

Esta definición utiliza el hecho de que las letras minúsculas y mayúsculas tienen códigos que están en una secuencia numérica, pero no depende de sus valores concretos. En particular, podemos calcular

```
        aMayuscula 'a'
=          {por la definición y dado que isLower 'a' = True}
        chr (base + ord 'a')
=          {por la definición de base}
        chr ((ord 'A' − ord 'a') + ord 'a')
=          {aritmética}
        chr (ord 'A')
=          {por ser chr (ord c) = c para todo c}
'A'
```

En el cálculo no necesitamos saber los códigos de 'A' y 'a'.

Una lista de caracteres se denomina *cadena*. Las cadenas serán tratadas en una sección posterior, donde se mostrará cómo se pueden usar para tener un control completo de lo que aparece en la pantalla.

Ejercicios

2.2.1 Definir una función *siguienteLetra* que tome como argumento una letra del alfabeto y devuelva la letra que le sigue. Suponer que la letra 'A' sigue a la 'Z'.

2.2.2 Definir una función *valorNum* que convierta un carácter dígito en su correspondiente valor numérico.

2.3 Enumeraciones

Hemos visto, en el caso de *Bool* y *Char*, que un modo de definir un nuevo tipo de datos es enumerar explícitamente sus valores. He aquí otro ejemplo. Supongamos que estamos interesados en un problema que tiene que ver con los días de la semana. Podemos introducir un nuevo tipo *Dia* mediante la declaración

```
data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
```

que define el tipo *Dia* como un conjunto de ocho valores distintos, siete de los cuales se representan mediante las constantes *Dom*, *Lun*, y así sucesivamente, y el octavo por el siempre presente ⊥ que se considera un valor incluido en todos los tipos. Las siete nuevas constantes se denominan *constructoras* del tipo de datos *Dia*. Por convenio, los nombres de las constructoras se distinguen de otras clases de nombres haciéndolos comenzar por una letra mayúscula. El nombre de un tipo declarado, también comienza por una letra mayúscula.

Se pueden comparar los elementos del tipo *Dia*; por tanto *Dia* se puede declarar como una concreción de la clases *Eq* y *Ord*. Sin embargo, una definición de (==) y (<) basada en ajuste de patrones tendría un gran número de ecuaciones. Es mejor codificar los elementos de *Dia* como enteros, y usar en su lugar comparaciones entre enteros. Como esta misma idea se puede emplear con otros tipos enumerados, introducimos una nueva clase de tipos *Enum* que describe tipos cuyos elementos pueden ser enumerados. La declaración es

```
class Enum α where
    toEnum    ::  α → Int
    fromEnum  ::  Int → α
```

Para declarar un tipo como concreción de *Enum* se han de dar definiciones para *toEnum* y *fromEnum*, las funciones de conversión entre elementos del tipo e *Int*. En las declaraciones de concreción, *fromEnum* debería ser inversa por la izquierda de *toEnum*, es decir,

```
fromEnum (toEnum x)  =  x
```

para todo *x*. Este requisito, al igual que el requisito de que (<) sea una relación transitiva de *Ord*, no se puede expresar en Haskell. La especificación de *fromEnum* como inversa por la izquierda de *toEnum* no es constructiva y no da lugar a una definición por defecto. El programador tiene la tarea de dar una definición constructiva de *fromEnum* que satisfaga esta especificación.

Ahora declaramos que *Dia* es un miembro de *Enum* por medio de la siguiente declaración de concreción

```
instance Enum Dia where
    toEnum Dom  =  0
    toEnum Lun  =  1
    toEnum Mar  =  2
    toEnum Mie  =  3
    toEnum Jue  =  4
    toEnum Vie  =  5
    toEnum Sab  =  6
```

Se puede definir la función *fromEnum* mediante siete ecuaciones, como veremos en el capítulo siguiente cuando estudiemos el ajuste de patrones con números naturales.

Al tratar sobre el tipo *Char* utilizamos exactamente esta misma idea de asociar valores de un tipo enumerado con enteros. De hecho, podemos declarar

```
instance Enum Char where
  toEnum   = ord
  fromEnum = chr
```

Dada la función *toEnum* de *Dia*, podemos declarar ahora

```
instance Eq Dia where
  (x == y)  =  (toEnum x == toEnum y)

instance Ord Dia where
  (x < y) = (toEnum x < toEnum y)
```

He aquí algunos ejemplos simples de uso del tipo *Dia*. Primero, podríamos definir

```
laborable    :: Dia → Bool
laborable d  = (Lun ≤ d) ∧ (d ≤ Vie)

festivo      :: Dia → Bool
festivo d    = (d == Sab) ∨ (d == Dom)
```

Después, podemos usar *toEnum* y *fromEnum* para definir una función *diaDespues* que devuelva el día siguiente a un día dado:

```
diaDespues    :: Dia → Dia
diaDespues d  = fromEnum ((toEnum d + 1) mod 7)
```

En particular, *diaDespues Sab* = *Dom*.

2.3.1 Declaraciones automáticas de concreción

Hasta ahora, hemos encontrado tres clases de tipos *Eq*, *Ord* y *Enum*. Las declaraciones de concreción para *Bool*, *Char* y *Dia* son, en cada caso, las obvias: el orden en que se enumeran los elementos en la declaración de tipo da toda la información necesaria. Haskell ofrece un modo abreviado para ahorrar al programador el trabajo de escribir estas declaraciones directamente. Por ejemplo, se podría declarar *Dia* escribiendo

```
data Dia  =  Dom | Lun | Mar | Mie | Jue | Vie | Sab
             deriving (Eq, Ord, Enum)
```

La cláusula **deriving** hace que el computador genere automáticamente las declaraciones de concreción de las clases de tipos nombradas, y estas concreciones serían las que hemos visto más arriba. Una declaración de concreción derivada automáticamente para *Enum* es posible únicamente para un tipo de datos que esté declarado mediante enumeración explícita de sus valores.

Ejercicios

2.3.1 Definir una función *diaAnterior* que devuelva el día anterior a un día dado.

2.3.2 Definir un tipo de datos *Direccion* cuyos valores describan los puntos cardinales de la brújula y definir una función *invertir* para invertir una dirección.

2.3.3 Declarar el tipo *Bool* como miembro de la clase de tipos *Enum*, dando una declaración explícita de concreción.

2.4 Tuplas

Un modo de combinar tipos para crear tipos nuevos es construir pares. Por ejemplo, el tipo (*Integer*, *Char*) consiste en todos los pares de valores (x, c) para los cuales x es un entero de rango arbitrario y c es un carácter. En particular, (3, 'a') y (1765341, '+') son valores de este tipo. El tipo (A, B) corresponde a la operación de producto cartesiano de la teoría de conjuntos, aunque allí es más frecuente la notación $A \times B$

El tipo polimórfico (α, β) es, de hecho, el mismo tipo *Par* α β que el introducido por la declaración

```
data Par α β  =  Emparejar α β
```

El tipo de datos *Par* α β tiene una sola constructora *Emparejar*. A diferencia de las constructoras de *Bool* y *Char*, esta constructora es una función; su tipo es

```
Emparejar  :: α → β → Par α β
```

Lo que distingue a *Emparejar x y* de (x, y) es que la segunda expresión utiliza una sintaxis especial y más tradicional.

Dos funciones básicas sobre pares son *fst* y *snd*, definidas mediante

```
fst       :: (α, β) → α
fst (x, y) = x

snd       :: (α, β) → β
snd (x, y) = y
```

Las definiciones de *fst* y *snd* utilizan ajuste de patrones: la expresión (x, y) en la parte izquierda de las ecuaciones es un patrón en el cual encaja cualquier par de valores. Así, *fst e* se evalúa reduciendo *e* a la forma (x, y) para ciertos valores x e y, y después devolviendo x.

Como los demás tipos, el tipo (α, β) contiene un valor adicional, a saber, el valor indefinido $\perp$. En particular, *indefinido* = $\perp$, donde

```
indefinido  :: (α, β)
indefinido  = indefinido
```

El valor $(\perp, \perp)$ es diferente de $\perp$. Para ver por qué, considérese la definición

```
comprobar       :: (α, β) → Bool
comprobar (x, y) = True
```

Como hemos definido *comprobar* mediante ajuste de patrones, se verifica la igualdad *comprobar* $\perp = \perp$. Por otra parte, *comprobar* $(\perp, \perp) = True$. En la evaluación de *comprobar p*, se requiere ajuste de patrones para reducir p a la forma (x, y), pero no hace falta determinar los valores de x e y. Como $\perp \neq True$, se deduce que $(\perp, \perp) \neq \perp$.

Más en general, si α se concreta a un tipo enumerado con m constructoras, e igualmente β se concreta a un tipo enumerado con n constructoras, entonces (α, β) tendrá $1 + (m+1) \times (n+1)$ valores en total.

También podemos definir funciones que tomen pares de funciones como argumento; por ejemplo,

$$\begin{array}{lcl} par & :: & (\alpha \to \beta, \alpha \to \gamma) \to \alpha \to (\beta, \gamma) \\ par\ (f, g)\ x & = & (f\,x, g\,x) \end{array}$$

$$\begin{array}{lcl} prod & :: & (\alpha \to \beta, \gamma \to \delta) \to (\alpha,\gamma) \to (\beta,\delta) \\ prod\ (f, g) & = & par\ (f \cdot fst, g \cdot snd) \end{array}$$

La palabra *prod* proviene de la teoría de categorías, donde se usa la notación $f \times g$ en lugar de *prod* (f, g). Nótese que *prod* $(f, g)\ \perp = (f\,\perp, g\,\perp)$, lo cual podemos ver argumentando:

$$\begin{array}{ll} & prod\ (f, g)\ \perp \\ = & \quad \{\text{definición de } prod\} \\ & par\ (f \cdot fst, g \cdot snd)\ \perp \\ = & \quad \{\text{definición de } par\} \\ & ((f \cdot fst)\ \perp, (g \cdot snd)\ \perp) \\ = & \quad \{\text{definición de composición}\} \\ & (f\,(fst\ \perp), g\,(snd\ \perp)) \\ = & \quad \{\text{dado que tanto } fst \text{ como } snd \text{ son estrictas}\} \\ & (f\,\perp, g\,\perp) \end{array}$$

Compárese esta definición con la siguiente de *prod'*:

$$prod'\ (f, g)\ (x, y) \quad = \quad (f\,x, g\,y)$$

El valor $\perp$ no encaja en el patrón (x, y), por tanto *prod'* $(f, g)\ \perp = \perp$ y, en consecuencia, *prod'* es una función estricta. La función *prod* es más perezosa que *prod'* y la preferiremos porque puede dar resultados adicionales.

Las funciones *par* y *prod* forman parte de un grupo de funciones de propósito general que pueden ser utilizadas para un estilo de programación alternativo que no es punto a punto. Por ejemplo, la función *menorIgual* = *uncurry* $(\leq)$, versión no currificada de $(\leq)$ sobre un tipo de adtos enumerado, se puede definir en términos de la misma función *menorIgualInt* sobre *Int* mediante

$$\begin{array}{lcl} menorIgual & :: & (Enum\ \alpha) \Rightarrow (\alpha, \alpha) \to Bool \\ menorIgual & = & menorIgualInt \cdot prod\ (toEnum, toEnum) \end{array}$$

Por dar otro ejemplo, podemos definir también la función *laborable* de la sección precedente mediante

$$laborable \quad = \quad uncurry\ (\wedge) \cdot par\ ((Lun \leq), (\leq Vie))$$

Este estilo no aplicativo da mejores resultados cuando los operadores binarios son funciones no currificadas. Como Haskell asigna tipos currificados a los operadores, tenemos que descurrificarlos explícitamente.

Las funciones *par* y *prod* satisfacen un cierto número de propiedades que son útiles en los cálculos no aplicativos:

$$\begin{array}{lcl} fst \cdot par\ (f, g) & = & f \\ snd \cdot par\ (f, g) & = & g \\ par\ (f, g) \cdot h & = & par\ (f \cdot h, g \cdot h) \\ prod\ (f, g) \cdot par\ (h, k) & = & par\ (f \cdot h, g \cdot k) \end{array}$$

Presentamos una demostración de la cuarta ecuación, que usa las otras tres ecuaciones:

$$\begin{array}{ll} & prod\ (f, g) \cdot par\ (h, k) \\ = & \quad \{\text{por la definición de } prod\} \\ & par\ (f \cdot fst, g \cdot snd) \cdot par\ (h, k) \\ = & \quad \{\text{tercera propiedad}\} \\ & par\ (f \cdot fst \cdot par\ (h, k), g \cdot snd \cdot par\ (h, k)) \\ = & \quad \{\text{primera y segunda propiedades}\} \\ & par\ (f \cdot h, g \cdot k) \end{array}$$

El cálculo utiliza exclusivamente funciones y no se usa el principio de extensionalidad descrito en la Sección 1.4. En el Capítulo 12, construiremos un programa que lleva a cabo tales demostraciones automáticamente.

Del mismo modo que formamos pares de valores, podemos también formar ternas, cuádruplas, y así sucesivamente. Nótese, sin embargo, que los tipos $(\alpha, (\beta, \gamma))$, $((\alpha, \beta), \gamma)$ y (α, β, γ) son distintos: el primero es un par cuya segunda componente es también un par, el segundo es un par cuya primera componente es un par y el tercero es una terna. Los pares, las ternas, las cuádruplas etc. pertenecen, pues, a tipos diferentes.

Por completitud, Haskell también proporciona una tupla *vacía*. Por definición, la expresión (), leída *unit*, tiene tipo (). El tipo () tiene dos elementos, $\perp$ y (). Un uso posible de () es convertir las constantes en funciones; por ejemplo,

$$\begin{array}{lcl} pifun & :: & () \to Float \\ pifun\ () & = & 3.14159 \end{array}$$

Al elevar las constantes al nivel de las funciones, se puede ir más allá en el estilo de programación no aplicativo. Por ejemplo, se podría escribir

$$cuadrado \cdot cuadrado \cdot pifun \quad \text{en lugar de} \quad (cuadrado \cdot cuadrado)\ pi$$

La expresión a la izquierda es la composición de tres funciones, mientras que la expresión a la derecha consiste en la composición de dos funciones aplicada a un argumento. Con constantes convertidas en funciones, no necesitamos mezclar composición con aplicación.

Finalmente, como ejemplo práctico del uso de tuplas, he aquí una función que devuelve un par de números, las dos raíces reales de una ecuación de segundo grado con coeficientes (a, b, c):

$$\begin{array}{l} raices :: (Float, Float, Float) \to (Float, Float) \\ raices\ (a, b, c) \end{array}$$

```
| a == 0      = error "no es de segundo grado"
| e < 0       = error "raíces complejas"
| otherwise   = ((-b - r)/d, (-b + r)/d)
where r = sqrt e
      d = 2 × a
      e = b × b - 4 × a × c
```

La definición local define tres valores, cuyo ámbito es toda la ecuación de *raices*.

2.4.1 Operadores de comparación

Podemos comparar elementos de (α, β) por igualdad, si tanto los elementos de α como los de β pueden ser comparados por igualdad. He aquí la declaración de concreción apropiada:

```
instance (Eq α, Eq β) ⇒ (α, β) where
    (x, y) == (u, v)  =  (x == u) ∧ (y == v)
```

De forma similar, podemos declarar los pares como un tipo ordenado:

```
instance (Ord α, Ord β) ⇒ Ord (α, β) where
    (x, y) < (u, v)  =  (x < u) ∨ (x == u ∧ y < v)
```

Este modo particular de ordenar pares de valores se llama *lexicográfico*. En particular,

```
? (1, indefinido) < (2, indefinido)
True

? (indefinido, 1) < (indefinido, 2)
{Interrupted!}
```

El primer cómputo termina porque ∨ se define mediante ajuste de patrones sobre el argumento izquierdo. Por tanto,

$$(1, \bot) < (2, \bot) = (1 < 2) \vee (1 == 2 \wedge \bot < \bot) = True$$

Por otra parte, el segundo cómputo no termina, por eso lo interrumpimos.

Ejercicios

2.4.1 Demostrar que $(f, g) \cdot prod\ (h, k) = prod\ (f \cdot h, g \cdot k)$.

2.4.2 Dar una declaración del tipo de datos *Terna* que corresponde a ternas.

2.4.3 Supongamos que una fecha se representa mediante una terna (d, m, a) de tres enteros donde d es el día, m es el mes y a es el año. Definir una función *edad* que tome dos fechas, siendo la primera la fecha actual y la segunda la fecha del cumpleaños de alguna persona P, y devuelva la edad de P como un número entero de años.

2.4.4 ¿Se puede declarar (α, β) como miembro de la clase de tipos *Enum*, sabiendo que tanto α como β son concreciones de *Enum*?

2.5 Otros tipos

Del mismo modo que declaramos un tipo enumerando sus constantes, también podemos declarar tipos cuyos valores dependen de los valores de otros tipos. Por ejemplo,

```
data Either = Left Bool | Right Char
```

Esta declaración introduce un tipo *Either*, cuyos valores se denotan mediante expresiones de la forma *Left b*, siendo b un booleano, y *Right c*, siendo c un carácter. Existen tres valores booleanos (incluyendo $\bot$) y 257 caracteres (incluyendo $\bot$), por tanto hay 261 valores distintos del tipo *Either*; éstos incluyen a *Left* $\bot$, *Right* $\bot$ y $\bot$. Cuando son funciones, las constructoras de un tipo son no estrictas, por tanto *Left* $\bot \neq \bot$ y *Right* $\bot \neq \bot$.

El tipo *Either* combina los booleanos y los caracteres en un único tipo. Podemos generalizar esta idea y definir

```
data Either α β  =  Left α | Right β
```

Con esta generalización, el tipo anterior se describe mediante *Either Bool Char*.

Los nombres *Left* y *Right* introducen dos constructoras para construir valores del tipo *Either*; estas constructoras son funciones no estrictas cuyos tipos son

```
Left  :: α → Either α β
Right :: β → Either α β
```

Hay dos propiedades claves que distinguen a las constructoras tales como *Left* y *Right* de otras funciones. En primer lugar, no tienen una definición asociada. Las constructoras simplemente construyen. Por ejemplo, *Left* 3 es una expresión que ya se encuentra en su forma más simple:

```
? Left 3
Left 3
```

La segunda propiedad es que las expresiones que incluyen *Left* y *Right* pueden aparecer como patrones en la parte izquierda de alguna definición. Por ejemplo,

```
casos              :: (α → γ, β → γ) → Either α β → γ
casos (f, g) (Left x)  =  f x
casos (f, g) (Right y) =  g y
```

Usando *casos* podemos definir *suma* mediante

```
suma       :: (α → β, γ → δ) → Either α β → Either γ δ
suma (f, g)  =  casos (Left · f, Right · g)
```

El término *suma* proviene de la teoría de categorías, donde se usa la notación $f + g$ en lugar *suma* (f, g).

Las propiedades algebraicas de *casos* y de *suma* son duales de las de *par* y *prod* descritas en la sección precedente:

```
casos (f, g) · Left          =  f
casos (f, g) · Right         =  g
h · casos (f, g)             =  casos (h · f, h · g)
casos (f, g) · suma (h, k)   =  casos (f · h, g · k)
```

La demostración de la última ecuación se deja como ejercicio.

Suponiendo que se puedan comparar por igualdad u orden los tipos α y β, podemos definir comparaciones en el tipo *Either* α β. Las declaraciones de concreción son

```
instance (Eq α, Eq β) ⇒ Eq (Either α β) where
    Left x == Left y    =  (x == y)
    Left x == Right y   =  False
    Right x == Left y   =  False
    Right x == Right y  =  (x == y)

instance (Ord α, Ord β) ⇒ Ord (Either α β) where
    Left x < Left y     =  (x < y)
    Left x < Right y    =  True
    Right x < Left y    =  False
    Right x < Right y   =  (x < y)
```

También podríamos haber generado estas concreciones automáticamente escribiendo

```
data Either α β  =  Left α | Right β
                    deriving (Eq, Ord)
```

Excepto un aspecto importante, hemos descrito los principales detalles de cómo se declaran nuevos tipos de datos. El aspecto que falta es que las declaraciones de tipos de datos pueden ser *recursivas*. Los tipos de datos recursivos se verán en los capítulos siguientes.

Ejercicios

2.5.1 Definir una función sobre un argumento de tipo *Either Bool Char* que se comporte de modo diferente sobre tres argumentos de la forma *Left* $\perp$, *Right* $\perp$ y $\perp$.

2.5.2 Demostrar que $casos\ (f, g) \cdot suma\ (h, k) = casos\ (f \cdot h, g \cdot k)$.

2.6 Renombramientos

Antes de seguir, es conveniente introducir un mecanismo notacional simple para dar nombres alternativos a los tipos, nombres que sugieran mejor los propósitos para los cuales se han creado. El mecanismo se llama declaración de *renombramiento*. Introduciremos la idea con un par de ejemplos simples.

En primer lugar, recordemos la función *raices* de la Sección 2.4, que tenía el tipo

```
raices  ::  (Float, Float, Float) → (Float, Float)
```

Podemos introducir dos renombramientos como una descripción alternativa de este de tipo

```
type Coefs   =  (Float, Float, Float)
type Raices  =  (Float, Float)
```


Estas declaraciones no definen nuevos tipos, sino simplemente nombres alternativos para tipos ya existentes. El tipo de *raices* puede ser descrito de modo equivalente

```
raices  ::  Coefs → Raices
```

La nueva descripción de tipos es más corta y más informativa.

Como segundo ejemplo, supongamos que queremos definir una función, digamos *mover*, que tome como argumentos un número, que representa una distancia, un ángulo y un par que representa las coordenadas de una posición, y genera una nueva posición a partir de dicho ángulo y dicha distancia. Podemos introducir un renombramiento para estos valores tal como sigue:

```
type Posicion  =  (Float, Float)
type Angulo    =  Float
type Distancia =  Float
```

Nótese que tanto *Angulo* como *Distancia* se introducen como renombramientos de *Float*. Podemos entonces definir *mover* mediante

```
mover          ::  Distancia → Angulo → Posicion → Posicion
mover d a (x, y)  =  (x + d × cos a, y + d × sin a)
```

Las funciones trigonométricas *cos* y *sin* están predefinidas. Igual que sucedía con el tipo de *raices*, el tipo de *mover* es más corto y ayuda a comprender lo que hace la función.

Los renombramientos se pueden parametrizar con una o más variables de tipo, igual que las definiciones de tipos de datos. Por ejemplo, todas las declaraciones que siguen son renombramientos válidos:

```
type Pares α          =  (α, α)
type Automorfismo α   =  α → α
type Marca α          =  (α, Bool)
```

No se pueden declarar dos renombramientos que se referencien mutuamente, ya que todo renombramiento ha de poder expresarse en términos de tipos de datos existentes. Por otra parte, es legítimo declarar un renombramiento en términos de otro renombramiento. Por ejemplo,

```
type Bools  =  Pares Bool
```

Aquí no hay circularidad porque *Bools* es un renombramiento de *Pares Bool*, que a su vez es un renombramiento de (*Bool*, *Bool*). Por la misma razón, se pueden mezclar renombramientos y definiciones; por ejemplo,

```
data UnoDos α  =  Uno α | Dos (Pares α)
```

2.6.1 Tipos nuevos

Es importante resaltar que los renombramientos de tipos, al no ser tipos nuevos, heredan todas las declaraciones de concreción existentes para el tipo subyacente: no se pueden dar nuevas declaraciones de concreción. Por ejemplo, (==) está definido sobre *Bools* porque está definido sobre el tipo subyacente (*Bool*, *Bool*). De forma similar, el significado de (==) sobre *Angulo* es exactamente el mismo que el de (==) sobre *Float*. En

la práctica, puede ocurrir que no deseemos esto. Por ejemplo, se podría desear declarar que la igualdad sobre *Angulo* sea la igualdad módulo un múltiplo de 2π. Siempre podríamos usar una función especialmente definida con este fin, pero si deseamos usar (==), entonces tenemos que declarar que *Angulo* sea un tipo de datos nuevos y no un renombramiento:

```
data Angulo = Ang Float

instance Eq Angulo where
  Ang x == Ang y = normalizar x == normalizar y

normalizar :: Float -> Float
normalizar x
    | x < 0        =  normalizar (x + rot)
    | x ≥ rot      =  normalizar (x - rot)
    | otherwise    =  x
    where rot      =  2 × pi
```

La función *normalizar* devuelve un ángulo θ en el rango $0 \leq \theta < 2\pi$.

El precio que se paga por usar esta definición de igualdad sobre ángulos es que los elementos de *Angulo* han de ser "encapsulados" mediante una constructora *Ang* para definir un nuevo tipo de datos. Esto tiene dos desventajas. La primera es que las operaciones que construyen y examinan ángulos, han de ser continuamente desencapsuladas y vueltas a encapsular. Esto añade tiempo de ejecución a las operaciones. En segundo lugar, al declarar un nuevo tipo de datos, podemos declarar posiblemente elementos indeseados. Por ejemplo, *Ang* $\perp$ es también un elemento de *Angulo*. En otras palabras, *Angulo* y *Float* no son *isomorfos*.

Reconociendo este problema, Haskell suministra un mecanismo alternativo para crear nuevos tipos. La declaración

```
newtype Angulo  =  Ang Float
```

define *Angulo* como un tipo nuevo cuya representación es exactamente la misma que la de *Float*. Difiere de un renombramiento en que ahora se define un nuevo tipo, que puede tener nuevas declaraciones de concreción de clase. Difiere de una definición **data** en dos aspectos. En primer lugar, encapsular y desencapsular valores con *Ang* no incrementa el tiempo de ejecución; si bien han de existir apariciones de *Ang* en el texto del programa, pueden ser eliminadas antes de que comenzar la evaluación. Los valores de un tipo nuevo son reemplazados sistemáticamente por los valores correspondientes del tipo subyacente. La segunda diferencia, necesaria para la eliminación segura descrita más arriba, es que *Ang* sea una constructora *estricta*, de tal modo que *Ang* $\perp$ y $\perp$ denoten el mismo valor. Por tanto, los valores de *Angulo* y *Float* están en correspondencia exacta uno a uno.

Ejercicios

2.6.1 Supongamos que se desea considerar iguales dos distancias si sólo se diferencian en 10 kms. ¿Podríamos definir una operación de igualdad sobre *Distancia*, siendo *Distancia* un renombramiento? En caso afirmativo, ¿podríamos denominar (==) a dicha operación?

2.6.2 Considérense las declaraciones

```
data Juan     =  J Int
newtype Pedro =  P Int
```

Definiendo funciones apropiadas, demostrar que *Juan* y *Pedro* son diferentes entre sí.

2.7 Cadenas

Llamamos *cadena* a una lista de caracteres. El tipo *String* es un renombramiento:

```
type String  =  [Char]
```

El tipo de datos $[\alpha]$ de las listas de tipo α se estudiará en el Capítulo 4. Las cadenas se escriben empleando una sintaxis especial: los caracteres que componen la cadena se encierran entre dos comillas dobles. La diferencia entre 'a' y "a" es que lo primero es un carácter, mientras que lo segundo es una lista de caracteres que contiene un único elemento. Por ejemplo,

```
? "a"
"a"

? "Hola a todos"
"Hola a todos"

? "Esta frase contiene un ↴fin de línea"
"Esta frase contiene  un ↴fin de línea"
```

Al ser las cadenas un tipo renombrado y no un tipo de datos diferente, no pueden ser declaradas independientemente como concreciones de *Eq* y *Ord*. Heredan todas las concreciones que se hayan declarado para listas generales:

```
instance Eq α ⇒ Eq [α] where
  ...
instance Ord α ⇒ Ord [α] where
  ...
```

Las definiciones de concreción que faltan se darán en el Capítulo 4. Las operaciones de orden sobre cadenas siguen la ordenación lexicográfica normal, puesto que ése será el orden que definiremos para las listas. Por ejemplo,

```
? "hola" < "halo"
False

? "la" < "lata"
True
```

A menudo, deseamos mostrar las cadenas *literalmente*, lo que quiere decir que (i) las dobles comillas no deben aparecer en la salida; y (ii) los caracteres especiales, tales como '↴', se imprimen como el carácter real que representan. Otros efectos que desearíamos son escribir una cadena en un fichero o leer una cadena de un fichero. El tema de la entrada y salida de datos se tratará en el Capítulo 10; por ahora, es suficente observar que Haskell dispone de *órdenes* primitivas para imprimir cadenas. La orden se llama *putStr* y su tipo es

$$putStr \;::\; String \rightarrow IO\ ()$$

Evaluar la orden *putStr* hace que la cadena sea impresa literalmente. Por ejemplo,

? *putStr* "Hola a todos"
Hola a todos

? *putStr* "Esta frase contiene un ↴fin de línea"
Esta frase contiene un
fin de línea

De hecho, cada petición de mostrar un valor en el terminal supone una aplicación implícita, o explícita, de *putStr* como ahora veremos.

2.7.1 La clase de tipos *Show*

Dependiendo de la aplicación, el programador puede desear producir tablas de números, diversos tipos de dibujos, o texto con formato. Para conseguir estos efectos de un modo coherente, Haskell proporciona una clase de tipos especial *Show*, que se declara mediante

```
class Show α where
    showsPrec  ::  Int → α → String → String
```

La función *showsPrec* se utiliza para mostrar valores de tipo α. Esta función se ha diseñado para ser flexible y eficiente, pero es algo complicada y no describiremos aquí el propósito de cada argumento (pero ver las Secciones 5.3 y 7.3 donde se dan más detalles). Baste decir que, usando *showsPrec*, se puede definir una función más simple

$$show \;::\; Show\ \alpha \Rightarrow \alpha \rightarrow String$$

La función *show* toma un valor y lo convierte en una cadena. Por ejemplo, *Bool* pertenece a la clase *Show* y *show* está definido para los booleanos de forma que

```
show False  =  "False"
show True   =  "True".
```

Una petición para evaluar una expresión booleana y mostrar el resultado en la pantalla implica una aplicación oculta de la función *putStr* · *show*. El valor booleano se convierte a una cadena usando *show* y el resultado se imprime mediante *putStr*. Podemos hacer que esta aplicación sea explícita:

? *putStr* (*show True*)
True

Las declaraciones previas para evaluar y mostrar valores de tipo α dependían de la existencia de una declaración de que α es una concreción de *Show*. Los tipos *Char* y *String* son también concreciones de *Show* y, por ejemplo,

```
show 'a'  =  "'a'"
```

Al aplicar *putStr* a la última expresión de la derecha, se elimina el par de comillas dobles más exteriores y eso hace que se muestren los tres caracteres 'a'.

Se ofrecen algunas concreciones de *Show* como primitivas. Por ejemplo, los enteros son imprimibles y el valor *show n* para un elemento *n* de tipo *Integer* es la lista de los caracteres que constituyen la representación decimal de *n*. Esto es también así para los demás tipos numéricos. Por ejemplo,

```
show 42        =  "42"
show (42.73)   =  "42.73"
```

El uso de *show* nos permite controlar el formato de salida. En los ejemplos siguientes, la operación ++ concatena dos listas y se describe en detalle en el Capítulo 4.

? *putStr* ("El año es␣" ++ *show* (3 × 667))
El año es 2001

? *putStr* (*show* 100 ++ "↴" ++ *show* 101 ++ "↴" ++ *show* 102)
100
101
102

Si no declaramos que, por ejemplo, el tipo *Dia* es miembro de la clase *Show*, entonces los valores de *Dia* no pueden ser presentados. Así,

? *Lun*
Error: cannot find definition of *show* for *Dia*[3]

Afortunadamente, no es necesario declarar explícitamente que *Dia* es miembro de *Show*. Es suficiente con declarar *Dia* mediante

```
data Dia  =  Dom | Lun | Mar | Mie | Jue | Vie | Sab
             deriving (Eq, Ord, Enum, Show)
```

Las concreciones de la clase *Show* se pueden derivar automáticamente para todos los tipos de datos algebraicos.

Por otra parte, las definiciones generadas automáticamente pueden no coincidir exactamente con lo que queremos. Por ejemplo, en la Sección 5.3 desearemos que un cierto tipo de datos pertenezca a *Show*, pero no querremos que la apariencia impresa de los valores sea la que da la definición generada automáticamente. Por tanto, necesitaremos dar una declaración de concreción explícita.

También es útil poder introducir tipos de datos cuyos valores *no* han de ser impresos, aunque pudieran serlo incluyendo la clase *Show* en una cláusula **deriving** para ese tipo de datos. Explicaremos esto con mayor detalle en el Capítulo 8.

[3] N.T. Error: no es posible encontrar la definición de *show* para *Dia*

Finalmente, si deseamos diseñar un modo particular de mostrar los elementos de un tipo de datos, siempre podremos hacerlo definiendo una función específica para ese propósito. No tenemos que usar necesariamente la función *show*. Para ilustrar este punto, mostraremos cómo adaptar la impresión de los elementos del siguiente tipo de datos:

```
data Dato  =  Par Bool Int
```

Si lo hubiéremos especificado así, la concreción de *Show* generada automáticamente mostraría los elementos de *Dato* como sigue:

```
? Par True (3 + 4)
Par True 7
```

Sin embargo, nada nos impide definir

```
showDato   ::  Dato → String
showDato (Par b n)
           =  if b then "+" ++show n else "−" ++show n
putDato    =  putStr · showDato
```

Por ejemplo,

```
? putDato (Par True (3 + 4))
+7
```

La definición de *showDato* utiliza la función *show* para *Int*, pero no suministra una función *show* para *Dato*. En otras palabras, *Dato* no es un miembro estricto de *Show* porque hemos creado un modo distinto de imprimir sus elementos. Esta no pertenencia no implica una categoría de segunda clase para *Dato*: las *clases de tipos* están ahí simplemente para nuestra conveniencia y para ofrecer métodos estándar para manejar los tipos predefinidos en Haskell. Los nuevos tipos de datos no tienen por qué seguir este criterio si no lo deseamos.

Ejercicios

2.7.1 Colocar las siguientes cadenas en orden creciente: "tucasa", "tuCasa" y "tu-casa".

2.7.2 ¿Qué valores devuelven las siguientes expresiones?

```
show (show 42)
show 42 ++show 42
show "}"
```

2.7.3 Supongamos que una fecha se representa mediante una terna (d, m, a) de tres enteros de *Int* donde d es el día, m es el mes y a el año. Defínase una función *showFecha* que toma una fecha y la imprime con palabras. Por ejemplo,

showFecha (10, 12, 1997) se imprime como "10 Diciembre, 1997".

Para complicarlo un poco, revisar la definición de modo que

showFecha (10, 12, 1997) se imprima como "10 de Diciembre de 1997"
showFecha (31, 12, 1997) se imprima como "31 de Diciembre de 1997"

2.8 Notas del capítulo

Boole (1847) introdujo un cálculo para razonar acerca de expresiones que devuelven valores de verdad, si bien él escribía $x + y$ en vez de $x \vee y$ y xy en lugar de $x \wedge y$. Tratados modernos de álgebra booleana y de lógica, aptos para informáticos, son Ben-Ari (1993), Burke y Foxley (1996) Gries y Schneider (1995).

Haskell usa la sintaxis && para $\wedge$, y || para $\vee$. La sintaxis de Haskell para $=$ y $\neq$ es == y /=. Los operadores de orden $\leq$ y $\geq$ se escriben <= y >=.

El conjunto de caracteres de Haskell se basa en el estándar ISO Unicode 8859-1. Al igual que la mayoría de los otros lenguajes, Haskell utiliza secuencias de escape para denotar caracteres de control. Por ejemplo, un carácter de fin de línea se denota mediante \n, un tabulador mediante \t y el carácter de retroceso mediante \b. Para más detalles sobre la representación de los códigos de control y de otros caracteres no estándar ver el manual de Haskell 1.3, o el de ISO (1989).

Los fundamentos teóricos de las clases de tipos se estudian en Wadler y Blott (1989) y Jones (1992, 1995). Las clases de tipos estándar de Haskell han evolucionado en las distintas versiones del lenguaje.

Las notaciones $f \times g$ y $f + g$ se usan en una rama de las matemáticas llamada *teoría de categorías* que, desde un cierto punto de vista, se puede considerar como uno de los fundamentos de la programación funcional. Textos adecuados para informáticos son Barr y Wells (1995), Pierce (1991) y Bird y de Moor (1997).

Capítulo 3

Números

Hemos mencionado ya tres tipos distintos de números: enteros de rango arbitrario (elementos de *Integer*), enteros de rango acotado (elementos de *Int*) y números en coma flotante de precisión simple (elementos de *Float*). Otros tipos de números son los números en coma flotante de precisión doble, los números racionales y los números complejos. Las definiciones de las operaciones aritméticas son diferentes para cada tipo de números; por ejemplo, la división entera es bastante diferente de la división real y no es posible comparar números complejos mediante (<). Haskell dispone de una jerarquía sofisticada de clases de tipos para describir distintas variedades de números; haremos un repaso simplificado de dicho sistema en la Sección 3.4.

Si bien (algunos) números se suministran como tipos de datos primitivos, teóricamente es posible introducirlos por medio de declaraciones de tipo adecuadas. Para empezar, estudiaremos cómo se hace esto. La discusión servirá para introducir tres conceptos importantes: tipos de datos recursivos, definiciones recursivas y demostraciones por inducción. Estas tres ideas, que constituyen la Santísima Trinidad de la programación funcional, serán también el tema principal de los tres capítulos siguientes.

3.1 Números naturales

Los números naturales son los números usados para contar: 0, 1, 2, y así sucesivamente. El tipo *Nat* se puede introducir mediante la declaración

$$\textbf{data}\ Nat \quad = \quad Cero \mid Suc\ Nat$$

Nat es nuestro primer ejemplo de declaración *recursiva* de un tipo de datos. La definición dice que *Cero* es un valor de *Nat*, y que *Suc n* es un valor de *Nat* siempre que *n* lo sea. En particular, el tipo de la constructora *Suc* (abreviatura de "sucesor") es $Nat \to Nat$. Por ejemplo, cada una de las expresiones

$$Cero,\ Suc\ Cero,\ Suc\ (Suc\ Cero)$$

es un elemento de *Nat*. El número 7, como elemento del tipo *Nat*, estaría representado por

$$Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ Cero))))))$$

Cada número natural se representa mediante un único valor de *Nat*. Por otra parte, no todo valor de *Nat* representa un número natural bien definido. De hecho, *Nat* también contiene los valores $\perp$, *Suc* $\perp$, *Suc* (*Suc* $\perp$), etc. Estos valores adicionales se estudiarán más adelante.

Veamos cómo programar las operaciones aritméticas y de comparación básicas sobre *Nat*. La suma se puede definir mediante

```
(+)          :: Nat → Nat → Nat
m + Cero     =  m
m + Suc n    =  Suc (m + n)
```

Ésta es una definición recursiva, que define + mediante ajuste de patrones sobre el segundo argumento. Como cada elemento de *Nat*, excepto $\perp$, o bien es *Cero*, o bien es de la forma *Suc n*, donde *n* es un elemento de *Nat*, los dos patrones de las ecuaciones para + representan conjuntos disjuntos y entre ambos cubren todos los números, excepto $\perp$.

He aquí cómo sería evaluada la expresión *Cero* + *Suc* (*Suc Cero*):

```
    Cero + Suc (Suc Cero)
=       {segunda ecuación de +}
    Suc (Cero + Suc Cero)
=       {segunda ecuación de +}
    Suc (Suc (Cero + Cero))
=       {primera ecuación de +}
    Suc (Suc Cero)
```

Este cálculo muestra por qué no es práctico introducir los números naturales mediante el tipo de datos *Nat*: la aritmética sería simplemente demasiado ineficiente. En particular, calcular $m + n$ requeriría $(n + 1)$ pasos de evaluación. Como aspecto positivo, contar con los dedos es una buena manera de entender la suma.

Dado +, podemos definir × mediante

```
(×)          :: Nat → Nat → Nat
m × Cero     =  Cero
m × Suc n    =  (m × n) + m
```

Dado ×, podemos definir la exponenciación (↑) mediante

```
(↑)          :: Nat → Nat → Nat
m ↑ Cero     =  Suc Cero
m ↑ Suc n    =  (m ↑ n) × m
```

Estas definiciones de +, × y ↑ siguen un patrón similar. En la Sección 3.3 veremos cómo se puede explotar este patrón común.

Podemos hacer que *Nat* sea miembro de la clase de tipos *Eq* mediante la declaración de concreción

```
instance Eq Nat where
    Cero == Cero     =  True
    Cero == Suc n    =  False
    Suc m == Cero    =  False
    Suc m == Suc n   =  (m == n)
```


Se trata también de una definición recursiva que emplea ajuste de patrones sobre ambos argumentos. De modo similar, podemos hacer que *Nat* sea miembro de *Ord* mediante la declaración

```
instance Ord Nat where
    Cero < Cero      =  False
    Cero < Suc n     =  True
    Suc m < Cero     =  False
    Suc m < Suc n    =  (m < n)
```

Se pueden mostrar los elementos de *Nat* definiendo una función *showNat*:

```
showNat              :: Nat → String
showNat Cero         =  "Cero"
showNat (Suc Cero)   =  "Suc Cero"
showNat (Suc (Suc n)) =  "Suc␣("++ showNat (Suc n) ++")"
```

La definición de *showNat* hace uso de tres patrones, a saber: *Cero*, *Suc Cero* y *Suc* (*Suc n*). Estos patrones son diferentes entre sí y juntos cubren todos los elementos de *Nat*, excepto $\perp$. Hay tres patrones en la definición de *showNat* porque los dos primeros números naturales se muestran sin paréntesis.

Alternativamente, podríamos haber introducido *Nat* mediante la declaración

```
data Nat  =  Cero | Suc Nat
             deriving (Eq, Ord, Show)
```

Con esta declaración, *Nat* es incluido automáticamente como miembro de cada una de las clases de tipos *Eq*, *Ord* y *Show*. En particular, la función *show* para *Nat* hace lo mismo que *showNat*. Como veremos más tarde, todos los tipos numéricos son tipos con igualdad y son concreciones de *Show*. Sin embargo, no todos los números pueden compararse mediante (<).

La operación que queda por definir, común a todos los números, es la resta (−). Sin embargo, la resta es una operación *parcial* sobre números naturales. Su definición es

```
(−)          :: Nat → Nat → Nat
m − Cero     =  m
Suc m − Suc n =  m − n
```

Esta definición impone ajuste de patrones sobre ambos argumentos; tomados conjuntamente, los patrones son disjuntos, pero no exhaustivos. Por ejemplo,

```
    Suc Cero − Suc (Suc Cero)
=       {segunda ecuación de (−)}
    Cero − Suc Cero
=       {patrón inexistente}
    ⊥
```

El comentario "patrón inexistente" en el último paso indica que no existe una ecuación para (−) con un patrón en el que ajuste (*Cero* − *Suc Cero*). Generalizando, $m - n = \perp$ si $m < n$. La naturaleza parcial de la resta de números naturales es el principal motivo para introducir los números *enteros*; sobre los enteros, (−) es una operación *total*. Por otra parte, siempre se puede definir una operación total ⊖ sobre los números naturales,

de modo que $m \ominus n = m - n$, si $m \geq n$ y $m \ominus n = Cero$, si $m < n$. La definición de $\ominus$ se deja como ejercicio.

Finalmente, presentamos dos ejemplos más de programación con *Nat*. La función factorial se define mediante

```
fact          :: Nat → Nat
fact Cero     =  Suc Cero
fact (Suc n)  =  Suc n × fact n
```

La función Fibonacci se define mediante

```
fib              :: Nat → Nat
fib Cero         =  Cero
fib (Suc Cero)   =  Suc Cero
fib (Suc (Suc n)) = fib (Suc n) + fib n
```

La definición de *fib* usa los mismos tres patrones que *showNat*.

3.1.1 Números parciales

Volvamos ahora al tema de la existencia de valores adicionales en *Nat*. Los valores

$\bot$, *Suc* $\bot$, *Suc* (*Suc* $\bot$), ...

son todos diferentes y cada uno de ellos es también un elemento de *Nat*. El que existan es consecuencia de tres hechos: (i) $\bot$ es un elemento de *Nat* porque toda declaración de un tipo de datos introduce al menos un valor extra, el valor indefinido de dicho tipo; (ii) por defecto, se supone que las operaciones constructoras de un tipo de datos son no estrictas; y (iii) *Suc n* es un elemento de *Nat* siempre que *n* lo sea.

Para apreciar por qué estos valores extras son diferentes entre sí, supongamos que definimos *indefinido* :: *Nat* mediante la ecuación *indefinido* = *indefinido*. Entonces

```
? Cero < indefinido
{Interrupted!}

? Cero < Suc indefinido
True

? Suc Cero < indefinido
{Interrupted!}

? Suc Cero < Suc (Suc indefinido)
True
```

Los valores extras se pueden interpretar del siguiente modo: $\bot$ corresponde al número natural acerca del cual no tenemos absolutamente ninguna información; *Suc* $\bot$ al número natural acerca del cual sólo sabemos que es mayor que *Cero*; *Suc* (*Suc* $\bot$) al número natural del cual sabemos que es mayor que *Suc Cero*; y así sucesivamente.

Hay todavía un valor más de *Nat*, a saber, el número "infinito":

Suc (*Suc* (*Suc* (*Suc* ...)))

Este número se puede definir mediante

```
infinito  :: Nat
infinito  =  Suc infinito
```

Es diferente de todos los otros números porque es el único número *x* para el cual *Suc m* < *x* devuelve *True* para todos los números finitos *m*. En este sentido, *infinito* es el número mayor de *Nat*. Si pedimos el valor de *infinito*, obtendríamos

```
? infinito
Suc (Suc (Suc (Suc (Suc {Interrupted!}
```

El número *infinito* satisface otras propiedades, en particular n + *infinito* = *infinito* para todo número *n*. La ecuación dual *infinito* + n = *infinito* se cumple sólo para los números finitos *n*. Veremos cómo demostrar afirmaciones como éstas en la siguiente sección.

Para resumir esta discusión, podemos dividir los valores de *Nat* en tres clases:

- Los números *finitos*, es decir, los que corresponden a números naturales bien definidos.
- Los números *parciales*, $\bot$, *Suc* $\bot$, etc.
- Los números *infinitos*, de los cuales hay exactamente uno, a saber, *infinito*.

Veremos que esta clasificación se mantiene en *todos* los tipos recursivos. Habrá elementos *finitos* del tipo, elementos *parciales* y elementos *infinitos*. Si bien el número natural infinito no se usa mucho, esto no es así para los valores infinitos de otros tipos de datos.

Ejercicios

3.1.1 Contruir los números positivos como un tipo de datos recursivo.

3.1.2 Definir una función *convertir* :: *Nat* → *Integer* que convierte un número natural a un entero.

3.1.3 Definir versiones de + y × que impongan ajuste de patrones sobre el primer argumento. ¿Se puede hacer lo mismo para ↑?

3.1.4 ¿Cuántos pasos de evaluación se necesitan para evaluar $m \times n$?

3.1.5 Definir una versión total $\ominus$ de la resta de forma que $m \ominus n = Cero$ si $m < n$.

3.2 Inducción

Para razonar sobre las propiedades de las funciones definidas recursivamente sobre un tipo de datos recursivo, podemos utilizar el principio de *inducción estructural*. En el caso de *Nat*, el principio de inducción estructural se puede definir como sigue: para demostrar que una propiedad $P(n)$ se cumple para todo número finito *n* de *Nat*, es suficiente demostrar:

Caso (*Cero*). Que se cumple $P(Cero)$.

Caso (*Suc n*). Que, si se cumple $P(n)$, entonces también se cumple $P(Suc\ n)$.

La inducción es válida por la misma razón que son válidas las definiciones recursivas: todo número finito, o bien es *Cero*, o bien es de la forma *Suc n*, donde *n* es un número finito. Si demostramos el primer caso, entonces hemos mostrado que la propiedad se cumple para *Cero*; si demostramos también el segundo caso, entonces hemos mostrado que la propiedad se cumple para *Suc Cero*, puesto que es cierta para *Cero*. Pero ahora, utilizando el mismo argumento, es también cierta para *Suc* (*Suc Cero*), y así sucesivamente.

El principio necesita ser extendido si deseamos afirmar que alguna propiedad es cierta para *todos* los elementos de *Nat*, pero posponemos la discusión de este punto a la siguiente sección.

Como ejemplo, demostremos que $Cero + n = n$ para todos los números finitos *n*. Dicho con palabras, *Cero* es un elemento neutro *por la izquierda* de +. Recordemos que + se define mediante las dos ecuaciones

$$\begin{aligned} m + Cero &= m \\ m + Suc\ n &= Suc\ (m + n) \end{aligned}$$

La primera ecuación expresa que *Cero* es un elemento neutro *por la derecha* de +. En general, *e* es elemento neutro por la izquierda de $\oplus$ si $e \oplus x = x$ para todo *x*, y es elemento neutro por la derecha si $x \oplus e = x$ para todo *x*. Si *e* es tanto elemento neutro por la izquierda como elemento neutro por la derecha de un operador $\oplus$, entonces se llama elemento neutro de $\oplus$. La terminología es apropiada porque sólo existe un valor que pueda ser a la vez elemento neutro por la derecha y por la izquierda. Por tanto, demostrando que *Cero* es elemento neutro por la izquierda, habremos demostrado que *Cero* es el elemento neutro de +.

Demostración. La demostración se hará por inducción sobre *n*. Concretamente, consideramos que $P(n)$ es el aserto $Cero + n = n$. Nos referiremos a esta ecuación como la *hipótesis de inducción*.

Caso (*Cero*). Tenemos que demostrar que $Cero + Cero = Cero$, lo cual es inmediato a partir de la primera ecuación de +.

Caso (*Suc n*). Tenemos que demostrar que $Cero + Suc\ n = Suc\ n$, lo cual hacemos simplificando la expresión izquierda:

$$\begin{aligned} & Cero + Suc\ n \\ = & \quad \{\text{segunda ecuación de } +\} \\ & Suc\ (Cero + n) \\ = & \quad \{\text{hipótesis de inducción}\} \\ & Suc\ n \end{aligned}$$

□

Este ejemplo ilustra el formato que usaremos para las demostraciones inductivas, tratando cada caso separadamente e indicando el final con el símbolo □. El último paso hace uso de la hipótesis de inducción, lo cual está permitido por el modo en que funciona la inducción.

Como ejemplo más complicado, demostraremos la conocida ley de que para todos los números naturales finitos *x*, *m* y *n*,

$$x \uparrow (m + n) \quad = \quad (x \uparrow m) \times (x \uparrow n)$$

Para demostrar esta ley, necesitaremos las definiciones de × y ↑ de las secciones precedentes que, para facilitar el seguimiento de la demostración, repetimos aquí:

$$\begin{aligned} m \times Cero &= Cero \\ m \times Suc\ n &= (m \times n) + m \\ \\ m \uparrow Cero &= Suc\ Cero \\ m \uparrow Suc\ n &= (m \uparrow n) \times m \end{aligned}$$

Demostración. La demostración se hará por inducción sobre *n*, tomando como hipótesis de inducción

$$x \uparrow (m + n) \quad = \quad (x \uparrow m) \times (x \uparrow n)$$

Es importante determinar sobre qué variable estamos haciendo la inducción, puesto que hay tres posibilidades, a saber: *x*, *m* y *n*.

Caso (*Cero*). La idea consiste en reemplazar *n* por *Cero* en la hipótesis y simplificar ambos lados independientemente.

Para el lado izquierdo, razonamos del modo siguiente:

$$\begin{aligned} & x \uparrow (m + Cero) \\ = & \quad \{\text{primera ecuación de } +\} \\ & x \uparrow m \end{aligned}$$

Para el lado derecho, razonamos del modo siguiente:

$$\begin{aligned} & (x \uparrow m) \times (x \uparrow Cero) \\ = & \quad \{\text{primera ecuación de } \uparrow\} \\ & (x \uparrow m) \times Suc\ Cero \\ = & \quad \{\text{segunda ecuación de } \times\} \\ & (x \uparrow m) \times Cero + (x \uparrow m) \\ = & \quad \{\text{primera ecuación de } \times\} \\ & Cero + (x \uparrow m) \\ = & \quad \{\text{demostración inductiva anterior}\} \\ & x \uparrow m \end{aligned}$$

Los dos lados se simplifican al mismo resultado, por tanto son iguales. La simplificación del lado derecho es la más complicada de las dos, porque implica demostrar que *Suc Cero* es un elemento neutro por la derecha de la multiplicación.

Caso (*Suc n*). De nuevo, la idea es reemplazar *n* por *Suc n* en la hipótesis y simplificar ambos lados independientemente.

Primero, el lado izquierdo:

$$\begin{aligned} & x \uparrow (m + Suc\ n) \\ = & \quad \{\text{segunda ecuación de } +\} \\ & x \uparrow Suc\ (m + n) \end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{segunda ecuación de } \uparrow\} \\
 & (x \uparrow (m + n)) \times x \\
= & \{\text{hipótesis de inducción}\} \\
 & ((x \uparrow m) \times (x \uparrow n)) \times x
\end{array}
$$

Ahora, el lado derecho:

$$
\begin{array}{ll}
 & (x \uparrow m) \times (x \uparrow Suc\ n) \\
= & \{\text{segunda ecuación de } \uparrow\} \\
 & (x \uparrow m) \times ((x \uparrow n) \times x)
\end{array}
$$

Los dos lados se simplifican a expresiones diferentes, que son iguales si suponemos que $\times$ es asociativo. Mostrar que ello es así implica otra demostración por inducción y se deja como ejercicio.

□

Aparte de una decisión, la demostración sigue una secuencia de pasos prácticamente automática: en cada caso, hay una sustitución en ambos lados de la ecuación y a continuación se simplifica cada lado; cada paso de simplificación está determinado por la forma de la expresión en curso de un modo bastante obvio.

La única parte creativa consiste en decidir sobre qué variable hacemos la inducción. En este ejemplo, había tres posibles elecciones, de las cuales sólo dos eran razonables, a saber, m y n. ¿Qué habría sucedido si hubiéramos hecho inducción sobre m? Vamos a verlo haciendo la demostración de nuevo:

Demostración. La demostración se hará por inducción sobre m.

Caso (*Cero*). Para el lado izquierdo, razonamos del modo siguiente:

$$
\begin{array}{ll}
 & x \uparrow (Cero + n) \\
= & \{\text{conjetura: } Cero + n = n \text{ para todo } n\} \\
 & x \uparrow n
\end{array}
$$

Para el lado derecho, razonamos del modo siguiente:

$$
\begin{array}{ll}
 & (x \uparrow Cero) \times (x \uparrow n) \\
= & \{\text{primera ecuación de } \uparrow\} \\
 & Suc\ Cero \times (x \uparrow n) \\
= & \{\text{conjetura: } Suc\ Cero \times n = n \text{ para todo } n\} \\
 & x \uparrow n
\end{array}
$$

La demostración de las dos conjeturas que afirman que *Cero* es elemento neutro por la izquieda de + y que *Suc Cero* es elemento neutro por la izquierda de $\times$, se dejan como ejercicios.

Caso (*Suc m*). Para el lado izquierdo, razonamos del modo siguiente:

$$
\begin{array}{ll}
 & x \uparrow (Suc\ m + n) \\
= & \{\text{conjetura: } Suc\ m + n = Suc\ (m + n)\} \\
 & x \uparrow Suc\ (m + n) \\
= & \{\text{segunda ecuación de } \uparrow\} \\
 & (x \uparrow (m + n)) \times x \\
= & \{\text{hipótesis de inducción}\} \\
 & ((x \uparrow m) \times (x \uparrow n)) \times x
\end{array}
$$

Para el lado derecho, razonamos del modo siguiente:

$$
\begin{array}{ll}
 & (x \uparrow Suc\ m) \times (x \uparrow n) \\
= & \{\text{segunda ecuación de } \uparrow\} \\
 & ((x \uparrow m) \times x) \times (x \uparrow n)
\end{array}
$$

Ahora necesitamos una suposición extra, a saber, que $\times$ es conmutativo, a la vez que asociativo; es decir, $x \times y = y \times x$. Las dos partes son iguales bajo estas dos suposiciones.

□

La segunda demostración es más complicada que la primera porque necesita nada menos que cuatro suposiciones extras, todas las cuales han de ser demostradas por inducción. La lección que podemos aprender aquí es que la elección de la variable sobre la que hacemos inducción (por supuesto, si hay elección) puede ser decisiva. Parecía razonable comenzar haciendo inducción sobre n porque + está definido recursivamente sobre su segundo argumento; por tanto, al menos el primer paso de simplificación del lado derecho se sabía que iba a ser simple. Afortunadamente, el resto de la demostración también fue simple.

3.2.1 Inducción exhaustiva

En la forma mostrada más arriba, el principio de inducción para *Nat* alcanza solamente a demostrar propiedades para los elementos *finitos* de *Nat*. Si queremos demostrar que una propiedad P también se cumple para todo número *parcial*, hemos de demostrar tres cosas:

Caso ($\perp$). Que se cumple $P(\perp)$.

Caso (*Cero*). Que se cumple $P(Cero)$.

Caso (*Suc n*). Que si se cumple $P(n)$, entonces también se cumple $P(Suc\ n)$.

Llamaremos *inducción exhaustiva*[1] a esta forma de inducción. Podemos omitir el segundo caso, pero entonces sólo podremos concluir que $P\ (n)$ se cumple para todo número parcial. La razón por la que el principio es válido es que todo número parcial, o bien es $\perp$, o es de la forma *Suc n* siendo n un número parcial.

Para ilustrar este punto, demostremos la poco intuitiva propiedad $m + n = n$ para todos los números m y todos los números parciales n.

Demostración. La demostración se hará por inducción sobre el número parcial n.

Caso ($\perp$). La ecuación $m + \perp = \perp$ se cumple por falta de patrones en la definición de +. Es decir, $\perp$ no encaja con los patrones *Cero* o *Suc n*.

Caso (*Suc n*). Para el lado izquierdo razonamos del modo siguiente:

$$
\begin{array}{ll}
 & m + Suc\ n \\
= & \{\text{segunda ecuación de } +\} \\
 & Suc\ (m+n) \\
= & \{\text{hipótesis de inducción}\} \\
 & Suc\ n
\end{array}
$$

[1] N.T. En el original *full induction*. Elegimos el término *exhaustiva* en lugar de el término *completa* porque la *inducción completa* es otra forma de inducción que se utiliza en preórdenes bien fundados. La inducción completa también se conoce como *inducción noetheriana*.

Como el lado derecho es también *Suc n*, hemos terminado.

□

El caso omitido, es decir, $m + Cero = Cero$, es falso, razón por la cual la propiedad no se cumple para los números finitos.

Por tanto, es fácil extender la inducción a los números parciales. Además, obtenemos como regalo que, habiendo demostrado que una ecuación se satisface para todos los números parciales, podemos afirmar que se satisface también para el número infinito *infinito*. En el Ejercicio 3.2.7 se trata esta cuestión con más detalle. En particular, ahora podemos afirmar que $m + infinito = infinito$ para todos los números m.

3.2.2 Síntesis de programas

En las demostraciones anteriores definimos algunas funciones y después utilizamos inducción para demostrar determinada propiedad. También podemos ver la inducción como un modo de *sintetizar* definiciones de funciones de forma que satisfagan las propiedades que deseamos.

Ilustremos esto con un ejemplo simple. Supongamos que *especificamos* la resta de números naturales mediante la condición

$$(m + n) - n \quad = \quad m$$

para todo m y n. La especificación no es una definición de $(-)$, sino simplemente una propiedad que ha de satisfacer. Sin embargo, podemos demostrar su validez por inducción sobre n en la ecuación anterior, y ver el cálculo como un modo de generar una definición adecuada para $(-)$. A diferencia de las demostraciones anteriores, razonamos ahora con la ecuación completa, puesto que no es posible una simplificación independiente de ambos lados si no sabemos cuáles son las reglas de simplificación.

Caso (*Cero*). Razonamos del modo siguiente:

$$\begin{array}{ll} & (m + Cero) - Cero = m \\ \equiv & \quad \{\text{primera ecuación de } +\} \\ & m - Cero = m \end{array}$$

Por tanto, podemos definir $m - Cero = m$ para satisfacer este caso. El símbolo $\equiv$ se usa para separar los pasos del cálculo, puesto que estamos calculando con asertos matemáticos y no con valores de un tipo de datos.

Caso (*Suc n*). Razonamos del modo siguiente

$$\begin{array}{ll} & (m + Suc\ n) - Suc\ n = m \\ \equiv & \quad \{\text{segunda ecuación de } +\} \\ & Suc\ (m + n) - Suc\ n = m \\ \equiv & \quad \{\text{hipótesis } (m + n) - n = m\} \\ & Suc\ (m + n) - Suc\ n = (m + n) - n \end{array}$$

Reemplazando $m + n$ en la última ecuación por m, tomamos $Suc\ m - Suc\ n = m - n$ para satisfacer este caso. Por tanto, hemos derivado

$$\begin{array}{lcl} m - Cero & = & m \\ Suc\ m - Suc\ n & = & m - n \end{array}$$

Este es el programa para $(-)$ que hemos visto anteriormente.

Ejercicios

3.2.1 Demostrar que *Suc Cero* es un elemento neutro por la izquierda de $\times$.

3.2.2 Demostrar que si $\oplus$ tiene un elemento neutro por la izquierda a y un elemento neutro por la derecha b, entonces $a = b$.

3.2.3 ¿Se cumple la ley $x \uparrow (m + n) = (x \uparrow m) \times (x \uparrow n)$ para *todos* los números naturales x, m y n?

3.2.4 Demostrar que $+$ es asociativa, es decir, $(m + n) + p = m + (n + p)$ para todos los números naturales m, n y p.

3.2.5 Demostrar que $\times$ es distributivo con respecto a $+$, es decir,

$$k \times (m + n) \quad = \quad (k \times m) + (k \times n)$$

para todos los números naturales k, m y n.

3.2.6 Definimos $\sqcap$ mediante

$$\begin{array}{lcl} Cero \sqcap n & = & Cero \\ Suc\ m \sqcap Cero & = & Cero \\ Suc\ m \sqcap Suc\ n & = & Suc\ (m \sqcap n) \end{array}$$

¿A qué operación sobre enteros corresponde $\sqcap$? Demostrar que $m \sqcap infinito = m$ para todos los elementos m, finitos, parciales o infinitos, de *Nat*.

3.2.7 En el texto se ha hecho la conjetura de que si $P(n)$ es una ecuación que se satisface para todos los números parciales n, entonces también se satisface $P(infinito)$. Esto requiere que todas las variables libres de P estén cuantificadas universalmente. Por ejemplo, podemos tomar como $P(n)$ cualquiera de las proposiciones

$$\begin{array}{ll} (\text{para todo } x \text{ y } m) & x \uparrow (m + n) = (x \uparrow m) \times (x \uparrow n) \\ (\text{para todo } m) & (m + n) - n = m \end{array}$$

Sin embargo, en general no podremos aceptar que $P(n)$ sea una ecuación con variables cuantificadas existencialmente. Consideremos, por ejemplo, la proposición $P(n)$ dada por

$$(\text{para algún número finito } m) \qquad n \ominus m = \bot$$

donde $\ominus$ es la versión total de la resta que satisface $n \ominus m = Cero$, si $n < m$. Demostrar por inducción que $P(n)$ se cumple para todos los números parciales n. Demostrar, también por inducción, que $infinito \ominus m = infinito$ para todos los números finitos m.

3.3 La función *fold*

Muchas de las definiciones recursivas consideradas hasta ahora tienen un formato común, ejemplificado por la siguiente definición de una función f:

$$\begin{array}{lcl} f & :: & Nat \to A \\ f\ Cero & = & c \\ f\ (Suc\ n) & = & h\ (f\ n) \end{array}$$

Aquí, A es cualquier tipo, c es un elemento de A, y $h :: A \to A$. Obsérvese que f funciona tomando un elemento de *Nat* y reemplazando *Cero* por c y *Suc* por h. Por ejemplo, f convierte

$$Suc\ (Suc\ (Suc\ Cero)) \quad \text{a} \quad h\ (h\ (h\ c))$$

Se pueden expresar las dos ecuaciones para f en términos de una sola función, *foldn*, llamada función *fold* para *Nat*. Su definición es

$$\begin{array}{lcl} foldn & :: & (\alpha \to \alpha) \to \alpha \to Nat \to \alpha \\ foldn\ h\ c\ Cero & = & c \\ foldn\ h\ c\ (Suc\ n) & = & h\ (foldn\ h\ c\ n) \end{array}$$

En particular, se cumple

$$\begin{array}{lcl} m + n & = & foldn\ Suc\ m\ n \\ m \times n & = & foldn\ (+\ m)\ Cero\ n \\ m \uparrow n & = & foldn\ (\times\ m)\ (Suc\ Cero)\ n \end{array}$$

Se deduce también que la función identidad *id* para *Nat* satisface $id = foldn\ Suc\ Cero$. Se puede definir una función *fold* apropiada para cada tipo recursivo. Veremos otras funciones *fold* en los capítulos siguientes.

En los ejemplos anteriores, cada aplicación de *foldn* devolvía un elemento de *Nat*. En los ejemplos siguientes, *foldn* devuelve un elemento de (*Nat, Nat*):

$$\begin{array}{lcl} fact & :: & Nat \to Nat \\ fact & = & snd \cdot foldn\ f\ (Cero, Suc\ Cero) \\ & & \textbf{where}\ f\ (m, n) = (Suc\ m, Suc\ m \times n) \end{array}$$

$$\begin{array}{lcl} fib & :: & Nat \to Nat \\ fib & = & fst \cdot foldn\ g\ (Cero, Suc\ Cero) \\ & & \textbf{where}\ g\ (m, n) = (n, m + n) \end{array}$$

La función *fact* calcula la función factorial y *fib* calcula la función de Fibonacci. Cada uno de estos programas funciona calculando primero un resultado más general, concretamente un elemento de (*Nat, Nat*), y extrayendo después el resultado requerido. De hecho,

$$\begin{array}{lcl} foldn\ f\ (Cero, Suc\ Cero)\ n & = & (n, fact\ n) \\ foldn\ g\ (Cero, Suc\ Cero)\ n & = & (fib\ n, fib\ (Suc\ n)) \end{array}$$

Estas ecuaciones pueden demostrarse por inducción. El programa para *fib* es más eficiente que una definición recursiva directa. El programa recursivo exige un número exponencial de operaciones +, mientras que el programa anterior requiere sólo un

número lineal. Trataremos la eficiencia más a fondo en el Capítulo 7, donde la técnica de programación que llevó al diseño de este nuevo programa para *fib* se estudiadará en un marco más general.

Aparecen dos ventajas al dar definiciones recursivas en términos de *foldn*. En primer lugar, la definición es más corta; en vez de tener que escribir dos ecuaciones, sólo hemos de escribir una. En segundo lugar, se pueden demostrar propiedades generales de *foldn* y utilizarlas para demostrar propiedades de cualquiera de sus particularizaciones. En otras palabras, en lugar de tener que escribir muchas demostraciones por inducción, hemos de escribir solamente una.

3.3.1 Fusión

Para motivar el concepto que vamos a introducir, demostremos la siguiente ley de *fusión* para *Nat*. Afirma que

$$f \cdot foldn\ g\ a \quad = \quad foldn\ h\ b$$

bajo ciertas suposiciones sobre las variables implicadas. La ley se llama de fusión porque el cómputo de f se funde con el cómputo de $foldn\ g\ a$. Las leyes de fusión pueden conducir a un aumento significativo de la eficiencia; en vez de procesar un elemento de un tipo de datos y luego procesar el resultado, a veces podemos combinar los dos pasos del proceso en uno solo.

En vez de establecer de entrada cuáles son los supuestos que hacen que la fusión funcione, utilicemos una demostración por inducción para ponerlas de manifiesto.

Demostración. Demostraremos que $f\ (foldn\ g\ a\ n) = foldn\ h\ b\ n$ mediante inducción exhaustiva sobre n, derivando las suposiciones necesarias a medida que procedemos.

Caso ($\perp$). Para la parte izquierda, razonamos del modo siguiente:

$$\begin{array}{cl} & f\ (foldn\ g\ a\ \perp) \\ = & \quad \{\text{agotamiento de patrones en la definición de } foldn\} \\ & f\ \perp \end{array}$$

Para la parte derecha, razonamos del modo siguiente

$$\begin{array}{cl} & foldn\ h\ b\ \perp \\ = & \quad \{\text{agotamiento de patrones en la definición de } foldn\} \\ & \perp \end{array}$$

Los dos resultados han de coincidir, por tanto obtenemos nuestra primera suposición, es decir, f es una función estricta.

Caso (*Cero*). Para la parte izquierda, razonamos del modo siguiente

$$\begin{array}{cl} & f\ (foldn\ g\ a\ Cero) \\ = & \quad \{\text{primera ecuación de } foldn\} \\ & f\ a \end{array}$$

Para la derecha, razonamos del modo siguiente:

$$\begin{array}{cl} & foldn\ h\ b\ Cero \\ = & \quad \{\text{primera ecuación de } foldn\} \\ & b \end{array}$$

Estos dos resultados han de coincidir, por tanto obtenemos nuestra segunda suposición, es decir, $f\,a = b$.

Caso $(Suc\ n)$. Para la parte izquierda, razonamos del modo siguiente:

$$\begin{array}{ll} & f\,(foldn\ g\ a\ (Suc\ n)) \\ = & \quad \{\text{segunda ecuación de } foldn\} \\ & f\,(g\ (foldn\ g\ a\ n)) \end{array}$$

Para la parte derecha, razonamos así:

$$\begin{array}{ll} & foldn\ h\ b\ (Suc\ n) \\ = & \quad \{\text{segunda ecuación de } foldn\} \\ & h\ (foldn\ h\ b\ n) \\ = & \quad \{\text{hipótesis de inducción}\} \\ & h\ (f\ (foldn\ g\ a\ n)) \end{array}$$

Los dos resultados han de coincidir, lo cual es cierto si $f \cdot g = h \cdot f$.

□

Dado que si una ecuación se cumple para todos los números parciales, también se cumple para los números infinitos, hemos demostrado el siguiente

Teorema de fusión para *foldn*. Si f es estricta, $f\,a = b$, y $f \cdot g = h \cdot f$, entonces

$$f \cdot foldn\ g\ a = foldn\ h\ b$$

3.3.2 Ejemplos

Usaremos ahora el teorema de fusión para demostrar que *Cero* es un elemento neutro por la izquierda de +; en símbolos, $Cero + n = n$ para todo número n. Dado que $n = foldn\ Suc\ Cero\ n$, debemos demostrar que

$$(Cero\ +) \cdot foldn\ Suc\ Cero \quad = \quad foldn\ Suc\ Cero$$

Usando la ley de fusión, esta ecuación se cumple si

$$\begin{array}{lcl} Cero + \bot & = & \bot \\ Cero + Cero & = & Cero \\ Cero + Suc\ n & = & Suc\ (Cero + n) \end{array}$$

Las tres ecuaciones son obvias a partir de la definición de +.

Como segundo ejemplo, demostraremos que *Suc Cero* es un elemento neutro por la izquierda de ×. Usando el teorema de fusión, exactamente igual que en el caso anterior, hemos de comprobar las tres condiciones:

$$\begin{array}{lcl} Suc\ Cero \times \bot & = & \bot \\ Suc\ Cero \times Cero & = & Cero \\ Suc\ Cero \times Suc\ n & = & Suc\ (Suc\ Cero \times n) \end{array}$$

Las dos primeras son inmediatas, según la definición de ×; la tercera se demuestra como sigue:

$$\begin{array}{ll} & Suc\ Cero \times Suc\ n \\ = & \quad \{\text{definición de } \times\} \\ & (Suc\ Cero \times n) + Suc\ Cero \\ = & \quad \{\text{definición de } +\} \\ & Suc\ ((Suc\ Cero \times n) + Cero) \\ = & \quad \{\text{definición de } +\} \\ & Suc\ (Suc\ Cero \times n) \end{array}$$

Ninguno de estos dos resultados requiere usar inducción explícitamente. La inducción ya intervino en la demostración de la ley de fusión.

Ejercicios

3.3.1 Demostrar que $foldn\ Suc\ Cero\ n = n$ para todos los elementos n de *Nat*, finitos, parciales o infinitos.

3.3.2 Usar la ley de fusión para demostrar que + es conmutativo.

3.3.3 La división de números naturales se puede especificar mediante la condición $(n \times m)/n = m$ para todo número positivo n y para todo m. Construir un programa para la división y demostrar que satisface dicha especificación.

3.3.4 La función *log* se puede especificar mediante la condición $log\ (2 \uparrow m) = m$ para todo m. Construir un programa para *log* y demostrar que satisface la especificación.

3.4 Números en Haskell

Hay tres clases de enteros: los enteros positivos, el cero y los enteros negativos. Por tanto, podríamos declarar *Integer* mediante una declaración de tipos

$$\begin{array}{lcl} \textbf{data}\ Integer & = & Neg\ Positivo \mid Cero \mid Pos\ Positivo \\ \textbf{data}\ Positivo & = & Uno \mid Suc\ Positivo \end{array}$$

El tipo *Positivo* corresponde a los números naturales positivos y el tipo *Integer* corresponde a los positivos precedidos de un signo o al cero. Una vez construidos los enteros, podríamos construir los racionales como pares de enteros y los racionales complejos como pares de racionales. También podríamos construir aproximaciones adecuadas para los números reales como secuencias de dígitos decimales. En el límite, podríamos recrear toda la aritmética partiendo de nada, sin usar otra cosa que puro cálculo simbólico.

Sin embargo, todo computador viene provisto de una unidad aritmética capaz de llevar a cabo como mínimo una aritmética entera de rango acotado. En muchos computadores también existe una unidad de coma flotante y, en aquellos computadores que no poseen este circuito, la aritmética en coma flotante se lleva a cabo de modo eficiente por medio de rutinas software de bajo nivel. Es mucho más eficiente utilizar estas facilidades que recurrir a alternativas simbólicas.

Los números se tratan de modos diferentes en diferentes lenguajes. Por ejemplo, en Miranda existe un tipo único *num* que contiene tanto a los enteros de rango arbitrario, como a los números de coma flotante de precisión limitada. Por su parte, Haskell divide

a los números en tipos distintos. En este libro seguiremos la terminología Haskell y nos referiremos a los tipos siguientes:

Int	enteros de rango acotado
Integer	enteros de rango arbitrario
Float	números en coma flotante de precisión simple
Double	números en coma flotante de precisión doble

Hay otros tipos numéricos en Haskell, por ejemplo el tipo *Rational*, pero no haremos uso de ellos (la sección siguiente muestra un modo de definir los números racionales). Tampoco diremos qué significan exactamente "precisión simple" y "precisión doble", porque su interpretación puede diferir según el hardware disponible, aunque existe un estándar al cual se adhieren la mayoría de los computadores. Veremos un modo de implementar la aritmética entera de rango arbitrario en la Sección 5.3. La aritmética de *Int* es la más rápida de todas; la aritmética de *Integer* es apreciablemente más lenta. Sin embargo, la aritmética de *Integer* es exacta en cálculos donde la aritmética de *Int* no lo es; más allá de un cierto rango se produce el *desbordamiento* de los enteros y el computador puede, o bien dar un mensaje de error, o simplemente devolver resultados incorrectos.

Nótese que en Haskell no existe el tipo *Nat* de los números naturales. Sin embargo, el fantasma de los números naturales está presente en los enteros porque podemos utilizar todavía el ajuste de patrones. Por ejemplo, podemos definir

```
fact          :: Integer → Integer
fact 0        =  1
fact (n + 1)  =  (n + 1) × fact n
```

Esta definición refleja la definición recursiva vista anteriormente, cuando escribimos *Cero* en lugar de 0 y *Suc n* en lugar de $n + 1$. El ajuste de patrones con enteros está restringido a la subclase de los números naturales. Así, en el patrón $n + 1$ encajan solamente enteros positivos. Aunque podríamos evitar el ajuste de patrones mediante el análisis por casos (o, mejor, usando una versión nueva de *foldn*), hay muchos ejemplos en los que el ajuste de patrones es el método de definición más claro. Más aún, el empleo del ajuste de patrones refleja los casos de una demostración por inducción.

Hay una diferencia crucial entre el constructor *Suc* de *Nat* y la función $(+\ 1)$ sobre los enteros: *Suc* es una función no estricta, mientras que $(+\ 1)$ es estricta. Por tanto, no hay números parciales en la aritmética predefinida.

3.4.1 Las clases de tipo numéricas

Se usan los mismos símbolos $+$, $\times$, etc., para la aritmética sobre cada tipo numérico, aunque estos operadores denotan operaciones diferentes para diferentes tipos. En otras palabras, $+$, $\times$, etc., son funciones sobrecargadas, al igual que $=$ y $<$. Haskell utiliza un sistema sofisticado de clases de tipos para describir los distintos tipos numéricos, pero no lo describiremos en detalle en este libro. En su lugar, presentaremos un esquema simplificado que intenta captar las ideas esenciales.

Todos los tipos numéricos de Haskell son concreciones de la clase de tipos *Num* definida mediante


```
class (Eq α, Show α) ⇒ Num α where
    (+), (−), (×)   :: α → α → α
    negate          :: α → α
    fromInteger     :: Integer → α

    x − y           =  x + negate y
```

La clase *Num* proporciona una definición por defecto de $(-)$ en términos de *negate*. Se puede aplicar la comparación de igualdad a todos los números y todos tienen representaciones imprimibles. Se pueden restar, sumar y multiplicar todos los números. Finalmente, hay una función de conversión *fromInteger* para tratar con las constantes. Una constante entera tal como 3 ó 65472233, representa la aplicación de *fromInteger* al valor *Integer* apropiado. Por tanto,

```
65472233  :: Num α ⇒ α
```

Las constantes numéricas enteras se definen de este modo indirecto, de forma que se pueden interpretar como valores de cualquier tipo numérico apropiado.

No se pueden comparar todos los números mediante $(<)$; por ejemplo, no se puede hacer con los números complejos. La clase de tipos *Real* incluye aquellos tipos de números computables para los cuales la operación $<$ tiene sentido:

```
class (Num α, Ord α) ⇒ Real α where
    toRational  :: α → Rational
```

El tipo *Rational* consiste en pares de números *Integer* y se estudia con más detalle en la sección siguiente. La función adicional *toRational* se basa en la idea de que todo número real de precisión finita se puede expresar como el cociente de dos enteros de precisión arbitraria. Las restantes clases de tipos que mencionaremos son *Integral* y *Fractional*. La clase de tipos *Integral* se declara mediante

```
class (Real α, Enum α) ⇒ Integral α where
    (div), (mod)   :: α → α → α
    toInteger      :: α → Integer
```

Los dos tipos primitivos *Int* e *Integer* pertenecen a *Integral*. Para cada tipo, los operadores **div** y **mod** se proporcionan como primitivos. Si x e y son enteros, e y es distinto de cero, entonces x **div** $y = \lfloor x/y \rfloor$, donde $\lfloor x \rfloor$, que se pronuncia "truncamiento" de x, es el entero más grande n que satisface $n \leq x$. En particular, $\lfloor 13.8 \rfloor = 13$ y $\lfloor -13.8 \rfloor = -14$. Discutiremos cómo calcular $\lfloor x \rfloor$ en la Sección 3.6. El valor x **mod** y se define mediante la ecuación

$$x = (x \textbf{ div } y) \times y + (x \textbf{ mod } y)$$

El número y en $(x$ **mod** $y)$ se denomina *módulo*. En la mayoría de las aplicaciones, el módulo es positivo. Para todo y positivo se cumple $0 \leq x \textbf{ mod } y < y$. De hecho, para todo y positivo, los valores de x **div** y y x **mod** y son los únicos enteros q y r que satisfacen la condición

$$x = q \times y + r \quad \text{y} \quad 0 \leq r < y$$

Cuando y es negativo, se cumple $y < x \textbf{ mod } y \leq 0$.

La clase de tipos *Fractional* contiene aquellos tipos numéricos para los cuales la división tiene sentido y además los números en coma flotante *Float* y *Double*.

```
class (Num α) ⇒ Fractional α where
    (/)            :: α → α → α
    fromRational   :: Rational → α
```

La función de conversión *fromRational* se usa para constantes en coma flotante. Una constante como 2.1414 proviene de una aplicación de *fromRational* al valor apropiado de tipo *Rational*. Así,

```
2.1414  :: Fractional α ⇒ α
```

Las constantes en coma flotante se definen de este modo indirecto, por lo que pueden ser interpretadas como valores de los tipos numéricos apropiados.

Ejercicios

3.4.1 ¿Cuál es el valor de 3 **mod**(−4)?

3.4.2 Demostrar que, cuando y es negativo, $y < x \textbf{ mod } y \leq 0$.

3.4.3 ¿Se cumple $\lfloor \lfloor x \rfloor \rfloor = \lfloor x \rfloor$?

3.5 Ejemplo: los racionales

Un número racional es un par (x, y) de enteros que representan el número x/y. Por ejemplo, $(1,7)$, $(3,21)$ y $(168,1176)$ denotan cada uno de ellos la fracción $1/7$, mientras que $(-1,3)$ y $(1,-3)$ denotan la fracción $-(1/3)$. Sin embargo, únicamente las fracciones (x, y) con $y \neq 0$ representan valores bien definidos.

Dado que existe un número infinito de modos de representar el mismo número racional, es razonable escoger una representación *canónica*. Siempre se puede representar una fracción de un modo único mediante un par (x, y) en el cual $y > 0$ y $mcd(x, y) = 1$, donde $mcd(x, y)$ es el máximo común divisor de x e y, que se define más adelante. En particular, la representación canónica de 0 que satisface estas condiciones es $(0, 1)$.

Nuestra intención en esta sección es definir los racionales como un tipo numérico de Haskell. Primeramente, introducimos *Racional* como un nuevo tipo de datos:

```
newtype Racional = Rac Integer Integer
```

La función *hazRac* toma un par de enteros y devuelve un elemento de *Racional* en forma canónica:

```
hazRac        :: (Integer, Integer) → Racional
hazRac (x, y) = Rac (udiv d) (vdiv d)
                where u = (signo y) × x
                      v = abs y
                      d = mcd (u, v)
```


Construiremos los números racionales únicamente con la función *hazRac*, garantizando, por tanto, que todos los números racionales están en forma canónica.

El tipo *Racional* se declara como un tipo nuevo, y no simplemente como un renombramiento de (*Integer, Integer*), porque las operaciones de comparación de igualdad para *Racional* difieren de las correspondientes operaciones sobre pares. Por ejemplo, $(2,1) < (3,2)$ sobre pares pero, en tanto que racionales, nosotros deseamos que $2 > 3/2$. Por tanto, definimos

```
instance Eq Racional where
    Rac x y == Rac u v  =  (x × v) == (u × y)

instance Ord Racional where
    Rac x y < Rac u v   =  (x × v) < (y × u)
```

La definición de (<) sólo es correcta si tanto *Rac x y* como *Rac u v* están en forma canónica.

Podemos convertir a caracteres un número racional definiendo una función especial *showRac*:

```
showRac (Rac x y)
= if y == 1 then show x else show x ++ "/" ++ show y
```

Esta definición utiliza la función primitiva *show* para el tipo *Integer* y también el operador ++ para concatenar dos cadenas (que se estudia en el capítulo siguiente). Si el argumento está en forma canónica, el resultado de *showRac* tiene un aspecto razonable.

El problema con *showRac* es que no declara *Racional* como miembro de la clase de tipos *Show*. Tal declaración es necesaria si queremos que *Racional* sea un tipo numérico. Se podría usar la definición por defecto, incluyendo una cláusula **deriving** en la declaración de *Racional*, pero el resultado sería una función *show* diferente de *showRac*. Dejando aparte esta cuestión, podemos declarar *Racional* como miembro de *Num* declarando

```
instance Num Racional where
    Rac x y + Rac u v   =  hazRac (x × v + u × y, y × v)
    Rac x y − Rac u v   =  hazRac (x × v − u × y, y × v)
    Rac x y × Rac u v   =  hazRac (x × u, y × v)
    negate (Rac x y)    =  hazRac (−x, y)
    fromInteger x       =  hazRac (x, 1)
```

Queda por definir *mcd*. Existen dos maneras ligeramente diferentes de especificar *mcd*. Una de ellas es decir que *mcd* (x, y) es el mayor entero positivo d que divide a ambos x e y. La otra es decir que *mcd* $(x, y) = d$ si y sólo si (i) d es divisor tanto de x como de y; y (ii) para todo e, si e es divisor tanto de x como de y, entonces e es divisor de d. Bajo la primera especificación, se tiene que *mcd* $(0, 0) = \perp$ porque todo entero positivo divide a 0 de modo que hay ninguno mayor; bajo la segunda especificación, se tiene que *mcd* $(0, 0) = 0$. Excepto para este caso particular, ambas definiciones son equivalentes. Empleando la segunda especificación, se puede calcular recursivamente *mcd* mediante el algoritmo de Euclides:

```
mcd        :: (Integer, Integer) → Integer
mcd (x, y) = if y = 0 then x else mcd (y, x mod y)
```

Por ejemplo, dado que 0 **mod** 3 = 0, (−6) **mod** 4 = 2 y (−3) **mod** 1 = 0, obtenemos

```
mcd (0, 3)   = mcd (3, 0) = 3
mcd (−6, 4)  = mcd (4, 2) = mcd (2, 0) = 2
mcd (−3, 1)  = mcd (1, 0) = 1
```

Finalmente, podemos declarar *Racional* como una concreción de la clase de tipos *Fractional* mediante

```
instance Fractional Racional where
    fromRacional   = id

Rac x y/Rac u v
    | u < 0   = hazRac (−x × v, −y × u)
    | u == 0  = error "división por 0"
    | u > 0   = hazRac (x × u, y × u)
```

Ejercicios

3.5.1 Sabiendo que para todo par de enteros positivos x e y, si x divide a y e y divide a x, entonces $x = y$, demostrar que todo número racional tiene una única representación (x, y), donde $y > 0$ y $mcd\ (x, y) = 1$.

3.5.2 Se puede representar todo entero x mediante un par de enteros (y, z) con $x = 10 \times y + z$. Por ejemplo, se puede representar 27 mediante (2,7), (3,−3) y (1,17), entre otros. Entre otras posibles representaciones, podemos escoger una en la cual $abs\ z < 5$ y $abs\ y$ sea lo más pequeño posible (cumpliendo además que $abs\ z < 5$). Demostrar que todo entero tiene una representación única de esta forma y definir una función *repInt* de modo que *repInt x* devuelva esta representación canónica.

3.6 Ejemplo: búsqueda lineal y binaria

La función $\lfloor - \rfloor$, que toma un número real arbitrario y devuelve un entero, es más primitiva que **div**, pero no se puede implementar en Haskell, que sólo permite representar números reales de precisión limitada. Sin embargo, es instructivo ver cómo se puede programar *truncar* :: *Float* → *Integer*. El problema no es tan sencillo como podría pensarse, por lo cual desarrollaremos un programa de modo sistemático. De hecho, desarrollaremos dos programas, uno mucho más eficiente que el otro. Las dos estrategias subyacentes a ambos programas se denominan búsqueda *lineal* y búsqueda *binaria*, respectivamente. La sección termina con otra aplicación de estas estrategias de búsqueda al problema del cálculo de raíces cuadradas.

Empecemos con la especificación:


```
truncar     :: Float → Integer
truncar x   = n ≡ n ≤ x < n + 1
```

Pensando en la forma que podría tener un programa para *truncar*, es tentador decidirse inmediatamente por un análisis por casos, considerando qué hacer si x es positivo, qué hacer si es negativo y posiblemente qué hacer si es cero. Pero la especificación no menciona casos y dado que los programas que no distinguen casos son más claros y simples que los que lo hacen, trataremos de no mencionar casos por ahora.

Tenemos que buscar un valor de n y la especificación sugiere al menos un modo en que se podría realizar esta búsqueda: podemos buscar valores de n que satisfagan $n \leq x$ y entonces incrementar n hasta que se cumpla también la condición $x < n + 1$. Por supuesto, pensando simétricamente, podríamos empezar la búsqueda por un n tal que $x < n + 1$ y a partir de aquí disminuir n de uno ene uno hasta que se cumpla la condición $n \leq x$. La idea de buscar hasta que se cumpla cierta condición, puede ser encapsulada en una función *hasta*, definida por

```
hasta       :: (α → Bool) → (α → α) → α → α
hasta p f x = if p x then x else hasta p f (f x)
```

Para buscar un n que satisfaga $n \leq x$, siendo x un número fijo, podemos comenzar con cualquier entero y entonces disminuirlo hasta que se cumpla la condición. Por ejemplo, la búsqueda

```
inferior  :: Integer → Integer
inferior  = hasta (≤ x) decrem, where decrem n = n − 1
```

aplicada a cualquier valor inicial, hará este trabajo. La búsqueda termina porque para todo número real x existe un entero n que satisface $n \leq x$.

El entero n hallado por esta primera búsqueda podría ser demasiado pequeño, de modo que necesitamos una segunda búsqueda que incremente n hasta que se cumpla $x < n + 1$. Debemos incrementar n de uno en uno para asegurar que se sigue cumpliendo la condición $n \leq x$. La segunda búsqueda se puede escribir de dos modos, o bien como

```
hasta (> (x − 1)) increm      where increm n = n + 1
```

o como *decrem* · *superior*, donde

```
superior = hasta (> x) increm      where increm n = n + 1
```

En la segunda solución, *superior* es dual de *inferior*. La función *decrem* · *superior* se aplicará al resultado de la primera búsqueda. El programa completo se puede expresar ahora como

```
truncar     :: Float → Intege
truncar x   = buscaDesde 0
              where buscaDesde  = decrem · superior · inferior
                    inferior    = hasta (≤ x) decrem
                    superior    = hasta (> x) increm
                    decrem n    = n − 1
                    increm n    = n + 1
```

Se ha intentado construir este programa del modo más claro posible. Hemos escogido los nombres *buscaDesde*, *increm* (en lugar del equivalente *Suc*) y *decrem* (en lugar de *pred*) para resaltar la contribución de cada parte de la definición, y hemos definido las funciones *inferior* y *superior* de modo que sean duales una de la otra. Finalmente, las diversas funciones auxiliares se han hecho locales a la definición principal. El programa es sorprendentemente breve, debido sobre todo a la ausencia de análisis por casos sobre el signo de x. Es fácil deducir cómo se tratan ambos casos: si $0 \leq x$, entonces el primer bucle termina enseguida y el segundo bucle hace el trabajo real; mientras que si $x < 0$ sucede lo contrario.

El programa anterior no es muy eficiente, dado que necesita aproximadamente *abs* x pasos para encontrar la respuesta. Un método mejor consiste en encontrar dos enteros m y n, tales que $m \leq x < n$, y entonces tratar de acercar m y n de forma que finalmente se cumpla la condición $m + 1 = n$. La ventaja de desacoplar ambas búsquedas es que podemos dar pasos más grandes. Con más precisión, considérese el par de búsquedas

```
inferior  =  hasta (≤ x) duplo
superior  =  hasta (> x) duplo
```

donde *duplo* $n = 2 \times n$. Si *inferior* se aplica a un número negativo y *superior* se aplica a un número positivo, está garantizado que ambas búsquedas encontrarán un par (m, n), que cumpla $m \leq x < n$ en un número de pasos proporcional a $\log_2$ (*abs* x). Estas dos búsquedas han de ser suplementadas con una tercera que acerque el par (m, n) de forma que finalmente $m + 1 = n$. Podemos decidir o bien incrementar m de uno en uno, o disminuir n de uno en uno, pero se puede hacer mejor. Considérese el entero $p = (m + n)$ **div** 2 que está en el punto medio entre m y n. Con más precisión, si $m + 1 < n$, entonces $m < p < n$. Si $p \leq x$, entonces podemos sustituir m por p; simétricamente, si $x < p$, entonces podemos sustituir n por p. El resultado es el siguiente programa:

```
truncar x  =  buscaDesde (−1, 1)
              where buscaDesde  =  fst · buscar · prod (inferior, superior)
                    inferior    =  hasta (≤ x) duplo
                    superior    =  hasta (> x) duplo
                    buscar      =  hasta fin mejorar
                    fin (m, n)  =  (m + 1 = n)
                    mejorar (m, n) = if p ≤ x then (p, n) else (m, p)
                                     where p = (m + n) div 2
```

La función *prod* se definió en la Sección 2.4. El nuevo programa necesita un número proporcional a $\log_2$ (*abs* x). Por ejemplo, si $x = 17.3$, obtenemos *inferior* $(-1) = -1$ y *superior* $1 = 32$. La tercera búsqueda produce los siguientes resultados intermedios

$$(-1, 32) \rightarrow (15, 32) \rightarrow (15, 23) \rightarrow (15, 19) \rightarrow (17, 19) \rightarrow (17, 18)$$

y devuelve 17. Si $x = -17.3$, obtenemos *inferior* $(-1) = -32$ y *superior* $1 = 1$. La tercera búsqueda produce los siguientes resultados intermedios

$$(-32, 1), (-32, -16), (-24, -16), (-20, -16), (-18, -16), (-18, -17)$$

y devuelve -18. La estrategia subyacente a este programa se denomina *búsqueda binaria*. En lugar de acercar dos enteros en pasos de uno en uno mientras se mantiene una cierta relación entre ellos, a veces se puede reducir la distancia directamente a la mitad.

3.6.1 Ejemplo: cálculo de raíces cuadradas

Daremos un segundo ejemplo de búsqueda. Nuestra intención es construir una definición de la función *sqrt* para calcular la raíz cuadrada (no negativa) de un número no negativo. La especificación matemática de *sqrt* es

$$sqrt\ x \geq 0 \quad \text{y} \quad cuadrado\ (sqrt\ x) = x,$$

siempre que $x \geq 0$. La especificación es bastante estricta en el sentido de que no permite usar las operaciones aritméticas de precisión limitada de los computadores reales. Por ejemplo, requiere que

$$sqrt\ 2 = 1.4142135623\ldots$$

sea calculado exactamente. Como veremos en los Capítulos 4 y 9, es posible diseñar una función que devuelva una lista infinita de dígitos, aunque el proceso de imprimir esta lista nunca terminará. El programador puede comprobar entonces que *sqrt* satisface su especificación demostrando que la lista de dígitos, si se dejara continuar el cálculo un tiempo suficiente, aproximaría la respuesta con cualquier grado de precisión requerido. Sin embargo, para los propósitos de este ejemplo, debilitaremos la especificación de *sqrt* :: *Float* → *Float* de forma que satisfaga

$$sqrt\ x \geq 0 \quad \text{y} \quad abs\ (cuadrado\ (sqrt\ x) - x) < eps$$

para un número suficientemente pequeño $eps > 0$.

Para ilustrar esta especificación revisada, supongamos que tomamos $eps = 0.0001$ y $x = 2$. Necesitamos que

$$abs\ (cuadrado\ (sqrt\ 2) - 2) < 0.0001$$

y dado que

$$\begin{aligned} 1.4141 \times 1.4141 &= 1.99967881 \\ 1.4142 \times 1.4142 &= 1.99996164 \\ 1.4143 \times 1.4143 &= 2.00024449 \end{aligned}$$

el valor $sqrt\ 2 = 1.4142$ es una respuesta aceptable.

Para diseñar *sqrt* usaremos el método de Newton para encontrar los ceros de una función dada. Se trata de un método iterativo que en cada paso mejora la aproximación a la respuesta; se procede de este modo hasta que se alcanza el grado de precisión requerido. En el caso de la raíz cuadrada, el método de Newton dice que si y_n es una aproximación a $\sqrt{x}$, entonces

$$y_{n+1} = (y_n + x/y_n)/2$$

es una aproximación mejor. Por ejemplo, si $x = 2$ y $y_0 = x$, se tiene

$$\begin{aligned} y_0 &= 2 \\ y_1 &= (2 + 2/2)/2 &&= 1.5 \\ y_2 &= (1.5 + 2/1.5)/2 &&= 1.4167 \\ y_3 &= (1.4167 + 2/1.4167) &&= 1.4142157 \end{aligned}$$

y así sucesivamente. Iterando este proceso, podemos determinar $\sqrt{2}$ hasta alcanzar el grado de precisión requerido, sujetos a las limitaciones de la aritmética del computador.

Usando la función de búsqueda *hasta* definida anteriormente, podemos implementar *sqrt* mediante el programa

```
sqrt   :: Float → Float
sqrt x  = hasta fin mejorar x
          where fin y      = abs (y × y − x) < eps
                mejorar y  = (y + x/y)/2
```

Ejercicios

3.6.1 Justificar la afirmación de que el segundo programa para *truncar x* necesita $2\log_2(abs\ x)$ pasos.

3.6.2 Sea $p = (m + n)$ **div** 2. Demostrar que $m < p < n$ siempre que $m + 1 < n$. ¿Qué sucede si $m + 1 = n$?

3.6.3 En el método de Newton, el criterio para determinar si una aproximación y de $\sqrt{x}$ es suficientemente buena se ha definido como $abs\ (y \times y - x) < eps$. Otro criterio posible es

$$abs(y \times y - x) < eps \times x$$

Reescribir la función *sqrt* para usar este criterio.

3.6.4 Otro criterio de convergencia diferente consiste en parar cuando dos aproximaciones sucesivas y e y' están suficientemente próximas:

$$abs\ (y - y') < eps \times abs\ y$$

Reescribir la función *sqrt* para usar este nuevo criterio. Justificar por qué es probable que estos criterios sean mejores en la práctica.

3.7 Números de Church

Finalizamos este capítulo sobre los números con un ejercicio de tipo teórico; este material no se usará más adelante y puede ser omitido sin problemas por quienes estén interesados en aspectos más prácticos de la programación.

Hemos visto que no es imprescindible suponer que los números están predefinidos, porque siempre podríamos introducirlos mediante declaraciones de tipos de datos. Ahora veremos que tampoco son necesarias las declaraciones de tipos de datos: ¡todo puede hacerse sólo con funciones! De hecho, se puede hacer cualquier cosa tomando las funciones como única construcción primitiva de las matemáticas; una vez que comprendemos las funciones, lo comprendemos todo. Esto era parte del objetivo que Alonzo Church se propuso en los años cuarenta, cuando estaba trabajando en el cálculo-λ, el sistema formal matemático que subyace en gran parte de la investigación actual sobre programación funcional.

Comencemos, con *Bool* en vez de con los números. ¿Qué son realmente los valores de verdad? Básicamente, se podría contestar que están ahí para tomar decisiones, para ayudarnos a elegir entre dos alternativas. Podemos captar esta idea definiéndolos como funciones:

```
true, false  :: α → α → α
true x y     = x
false x y    = y
```

La función *true* escoge la primera alternativa; la función *false* escoge la segunda. Así, los booleanos de Church pueden definirse como un renombramiento de un cierto tipo funcional:

```
type Cbool α  = α → α → α
```

La definición de la negación utilizando los booleanos de Church es

```
not    :: Cbool (Cbool α) → Cbool α
not x  = x false true
```

De modo similar, la conjunción y la disyunción con los booleanos de Church se definen por

```
and, or  :: Cbool (Cbool α) → Cbool α → Cbool α
and x y  = x y false
or x y   = x true y
```

En cada caso, el booleano de Church x selecciona las alternativas apropiadas. Esta función toma booleanos de Church como argumento, lo que explica las signaturas de tipo de *not*, *and* y *or*.

En la misma línea, se podría preguntar: ¿qué son exactamente los números naturales? La respuesta, se podría decir, es que se utilizan para contar, para controlar el número de veces que se hace algo. De nuevo, podemos captar esta idea definiendo los números naturales como funciones:

```
cero, uno, dos  :: (α → α) → α → α
cero f          = id
uno f           = f
dos f           = f · f
```

y así sucesivamente. La función *cero f* aplica f cero veces a su argumento, la función *uno f* aplica f una vez, y así sucesivamente. Así, los números de Church son también un renombramiento:

```
type Cnum α  = (α → α) → (α → α)
```

En lugar de tener que escribir cada número explícitamente, podemos usar la función sucesor definida mediante

```
suc      :: Cnum α → Cnum α
suc cn f = f · cn f
```

Dicho con palabras, *Suc cn f* primero aplica *f cn* veces, y después una vez más.

He aquí un programa que convierte números naturales en números de Church y a la inversa:

```
church      :: Int → Cnum Int
church      =  foldn suc cero

natural     :: Cnum Int → Int
natural cn  =  cn (+ 1) 0
```

La función *foldn* fue definida para *Nat* en la Sección 3.3. El valor *natural cn* se calcula usando *cn* para aplicar la función (+ 1) exactamente *cn* veces al número 0. Obsérvense los tipos de *church* y *natural*. El tipo del argumento de *natural* ha de ser *Cnum Int* porque su argumento se aplica a una función *Int* → *Int* en la parte derecha. El tipo del resultado de *church* se escoge de forma que sea el mismo, para asegurar que *church* y *natural* son inversas una de la otra.

Podemos también hacer cálculos aritméticos con los números de Church. De hecho, hay más de un modo de implementar las operaciones aritméticas básicas, aunque no a todas ellas se les puede asignar el tipo *Cnum* α → *Cnum* α → *Cnum* α.

Una definición legítima de *sumar* es

```
sumar          :: Cnum α → Cnum α → Cnum α
sumar cn dn f  =  cn f · dn f
```

Esto define *sumar cn dn* como el resultado de aplicar *f* primero *dn* veces, y después *cn* veces más. Una definición todavía más breve de *sumar* es

```
sumar'     :: Cnum (Cnum α) → Cnum α → Cnum α
sumar' cn  =  cn suc
```

La función *sumar' cn dn* aplica *suc* exactamente *cn* veces a *dn*. El primer argumento de *sumar'* es un número de Church que toma *suc* :: *Cnum* α como argumento. Esto explica el cambio de tipo.

Se puede dar también una definición breve para la multiplicación:

```
mult       :: Cnum α → Cnum α → Cnum α
mult cn dn =  cn · dn
```

La función *mult cn dn f* aplica *dn f* exactamente *cn* veces. Una definición alternativa es

```
mult'     :: Cnum (Cnum α) → Cnum α → Cnum (α → α)
mult' cn  =  cn · sumar
```

La función *mult' cn dn* aplica *sumar dn* exactamente *cn* veces. Podemos reemplazar *sumar* por *sumar'* en la definición de *mult'* y así obtener una tercera versión de la multiplicación. Las signaturas de tipos de estas funciones son cada vez más complicadas y no intentaremos justificarlas aquí. De hecho, fueron deducidas con la ayuda del sistema de inferencia de tipos de Haskell.

Finalmente, podemos definir la exponenciación como

```
elevar cn  =  cn · mult
```


Hay una permutación de argumentos aquí: la expresión *elevar m n* significa n^m. La función *elevar cn dn* aplica *mult dn* exactamente *cn* veces. Una definición aún más breve es

```
elevar' cn dn  =  cn dn
```

Dejamos como ejercicio explicar por qué funciona esta versión.

Ejercicios

3.7.1 Escribir una definición de *esCero* :: *Cnum* α → *CBool* α que determina si su argumento es *cero* o no.

3.7.2 Considérese la definición *elevar' m n* = *m n*. ¿Por qué devuelve el número de Church correspondiente a n^m? ¿Cuál es el tipo de *elevar'*?

3.7.3 Usar el computador para descubrir el tipo de *elevar*.

3.7.4 Definir una versión de Church para *Either* α β.

3.8 Notas del capítulo

En Knuth (1981) se puede encontrar un estudio completo de la aritmética de los computadores. Las búsquedas lineal y binaria son técnicas de programación fundamentales y se tratan en Morgan (1996) y Kaldewaij (1990). En Graham, Knuth y Patashnik (1990) se dan las propiedades de *truncar*, así como muchas otras funciones numéricas. Los números de Church se presentan en Church (1941) y se analizan en Gordon (1994).

Capítulo 4

Listas

El tipo de datos de las listas es la espina dorsal de la programación funcional. Las listas se pueden usar para transportar datos de una función a otra; se pueden conservar para uso posterior, reordenar y combinar con otras listas; las listas de números se pueden sumar y multiplicar; las listas de caracteres se pueden leer e imprimir; y así sucesivamente. El listado de las operaciones útiles sobre listas es muy largo. El propósito de este capítulo es introducir las funciones más importantes sobre listas, dar sus nombres y sus definiciones, dar ejemplos simples de cómo se usan y las propiedades que satisfacen. También describimos el principio de inducción sobre listas para demostrar tales propiedades. En el Capítulo 5 se dan ejemplos más extensos sobre el uso de las listas; el material se ha preparado de modo que se pueda estudiar en paralelo con este capítulo.

4.1 Notación para listas

Una lista finita se denota empleando corchetes y comas. Por ejemplo, [1, 2, 3] es una lista de tres números y ["hola", "adiós"] es una lista de dos cadenas. Todos los elementos de una lista han de ser del mismo tipo. La lista vacía se escribe [] y la lista unitaria, que contiene sólo un elemento x, se escribe $[x]$. En particular, [[]] es una lista unitaria cuyo único elemento es la lista vacía.

Si todos los elementos de una lista tienen tipo α, se le asigna a la lista el tipo $[\alpha]$ (pronunciado "lista de α"). Por ejemplo, los siguientes son tipos válidos para listas:

$$
\begin{array}{lcl}
[1, 2, 3] & :: & [Int] \\
['h', 'o', 'l', 'a'] & :: & [Char] \\
[[1, 2], [3]] & :: & [[Int]] \\
[(+), (\times)] & :: & [Int \to Int \to Int]
\end{array}
$$

Por otra parte, [1, "bonito día"] no es una lista bien formada porque sus elementos tienen tipos diferentes. Las cadenas introducidas en el Capítulo 2 son simplemente listas de caracteres escritas con una sintaxis especial. Así, "hola" es tan sólo una sintaxis adicional conveniente para la lista $['h', 'o', 'l', 'a']$. Toda operación genérica sobre listas es, por tanto, aplicable también a las cadenas.

A diferencia de los conjuntos, una lista puede contener el mismo valor más de una vez. Por ejemplo, [1, 1] es una lista de dos elementos, ambos iguales a 1, y es distinta de la lista [1] que contiene sólo un elemento. Dos listas son iguales si y sólo si contienen los mismos valores en el mismo orden.

4.1.1 Las listas como tipo de datos construido

Se puede construir una lista desde cero, comenzando con la lista vacía y añadiendo sucesivamente elementos uno por uno. Se puede añadir elementos al principio de la lista o al final, o en algún punto intermedio. En la siguiente declaración de tipo de datos, las listas no vacías se construyen añadiendo elementos al principio de la lista:

$$\textbf{data}\ \mathit{List}\ \alpha \quad = \quad \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

En el Ejercicio 4.1.4 se da una visión distinta de las listas. El constructor *Cons* (abreviatura para "construir" —el nombre procede del lenguaje de programación LISP) añade un elemento al principio de la lista. Por ejemplo, la lista [1, 2, 3] se representaría como el siguiente elemento de tipo *List Int*:

$$\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))$$

En la programación funcional, las listas son elementos del tipo *List* α. Se usa la sintaxis $[\alpha]$, en lugar de *List* α, la notación [], en lugar de la constructora *Nil*, y la constructora *Cons* se escribe como un operador infijo (:). Además, el operador (:) es asociativo por la derecha; por tanto,

$$[1, 2, 3] \quad = \quad 1 : (2 : (3 : [])) = 1 : 2 : 3 : []$$

En otras palabras, la sintaxis especial de la izquierda puede verse como una abreviatura de la sintaxis de la derecha, que es también especial pero sólo por el hecho de que se les da nombres no estándar a las constructoras.

Al igual que otras funciones sobre tipos de datos construidos, se pueden definir funciones sobre listas mediante ajuste de patrones. Por ejemplo, la siguiente declaración de concreción define el operador (==) sobre listas:

$$\begin{array}{lcl}
\textbf{instance}\ (\mathit{Eq}\ \alpha) \Rightarrow \mathit{Eq}\ [\alpha]\ \textbf{where} & & \\
\quad [] == [] & = & \mathit{True} \\
\quad [] == (y : ys) & = & \mathit{False} \\
\quad (x : xs) == [] & = & \mathit{False} \\
\quad (x : xs) == (y : ys) & = & (x == y) \wedge (xs == ys)
\end{array}$$

Noten los nombres dados a las variables de tipo lista. Por convenio, usaremos las letras *x*, *y*, *z*, etc., para denotar a los componentes de las listas, y los nombres *xs*, *ys*, *szs* etc., para denotar a las propias listas. Se puede extender el convenio, escribiendo *xss*, *yss*, *zss*, y así sucesivamente, para denotar listas cuyos elementos son listas. Este convenio no viene impuesto por Haskell, simplemente es una ayuda útil para la compresión.

Parece obvio que si se desea comparar valores de tipo $[\alpha]$ por igualdad, entonces se necesita una operación de igualdad sobre α. Sólo hay un caso en que esto no es necesario: cuando se desea comprobar si una lista es igual a la lista vacía. La función *null*, definida por

$$\begin{array}{lcl}
\mathit{null} & :: & [\alpha] \rightarrow \mathit{Bool}
\end{array}$$

$$\begin{array}{lcl}
\mathit{null}\ [] & = & \mathit{True} \\
\mathit{null}\ (x : xs) & = & \mathit{False}
\end{array}$$

no requiere que α sea un tipo con igualdad; ver el Ejercicio 4.1.3.

4.1.2 Inducción sobre listas

Recordemos de la Sección 3.2 que, para el tipo de datos *Nat* de los números naturales, la inducción estructural se basaba en tres casos: todo elemento de *Nat*, o bien es $\perp$, o *Zero*, o es de la forma *Succ n* para algún elemento *n* de *Nat*. De modo similar, la inducción estructural sobre listas se basa también en tres casos: toda lista, o bien es la lista indefinida $\perp$, o la lista vacía [], o es de la forma (*x*:*xs*) para algún valor *x* y lista *xs*. Para demostrar por inducción que una propiedad *P*(*xs*) se cumple para todas las listas *xs*, es suficiente por tanto demostrar tres casos:

Caso ($\perp$). Que se cumple $P(\perp)$.

Caso ([]). Que se cumple $P([])$.

Caso ($x : xs$). Que si se cumple $P(xs)$, entonces también se cumple $P(x : xs)$ para todo *x*.

Si sólo demostramos los dos últimos casos, entonces sólo es posible concluir que *P*(*xs*) se cumple para toda lista finita; si demostramos solamente el primero y el tercero, entonces podemos concluir solamente que *P*(*xs*) se cumple para toda lista parcial. Si *P* tiene la forma de una ecuación, como lo tienen todas nuestras leyes, entonces demostrar el primero y el tercer casos es suficiente para probar que *P*(*xs*) se cumple para toda lista infinita. Las listas infinitas y parciales se describen en la sección siguiente. A lo largo del resto del capítulo, se dan ejemplos de demostraciones por inducción.

Ejercicios

4.1.1 Dar un ejemplo de expresión que contenga dos apariciones de la lista vacía, la primera con tipo [*Bool*] y la segunda con tipo [*Char*].

4.1.2 Completar la siguiente declaración de concreción:

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Ord}\ \alpha) \Rightarrow \mathit{Ord}\ [\alpha]\ \textbf{where} \\
\quad \ldots
\end{array}$$

Recordemos del Capítulo 2 que, para el tipo particular [*Char*], el orden sobre cadenas era lexicográfico. La definición de $\leq$ sobre listas debería ser también lexicográfica.

4.1.3 Consideren las siguientes dos definiciones de la función *last*, que devuelve el último elemento de una lista no vacía:

$$\begin{array}{lcl}
\mathit{last}\ (x : xs) & = & \textbf{if}\ \mathit{null}\ xs\ \textbf{then}\ x\ \textbf{else}\ \mathit{last}\ xs \\
\mathit{last'}(x : xs) & = & \textbf{if}\ xs == []\ \textbf{then}\ x\ \textbf{else}\ \mathit{last'}\ xs
\end{array}$$

La diferencia entre *last* y *last'* es que tienen tipos ligeramente diferentes:

$$\begin{array}{lcl}
\mathit{last} & :: & [\alpha] \rightarrow \alpha \\
\mathit{last'} & :: & (\mathit{Eq}\ \alpha) \Rightarrow [\alpha] \rightarrow \alpha
\end{array}$$

Dado que la definición de *last'* necesita una comprobación explícita de igualdad, su tipo está restringido a listas cuyos elementos proceden de un tipo con igualdad. Dar una expresión *e* tal que el evaluador responda de modo diferente a *last e* y *last' e*.

4.1.4 Una visión dual de las listas consiste en construirlas añadiendo elementos por el final:

$$\textbf{data}\ \mathit{Liste}\ \alpha \quad = \quad \mathit{Nil} \mid \mathit{Snoc}\ (\mathit{Liste}\ \alpha)\ \alpha$$

Obsérvese que *Snoc* es simplemente *Cons* leído al revés. Desde esta perspectiva, la lista [1, 2, 3] se representaría como un elemento de *Liste Int* mediante

$$\mathit{Snoc}\ (\mathit{Snoc}\ (\mathit{Snoc}\ \mathit{Nil}\ 1)\ 2)\ 3$$

Ambos enfoquen ofrecen exactamente la misma información, pero organizada de diferente modo. Por ejemplo, la función *head*, que devuelve el primer elemento de una lista no vacía, se puede definir fácilmente con el tipo de datos $[\alpha]$ pero de un modo más complicado con *Liste* α. Dar las definiciones de *head* para ambos tipos. Definir una función

$$\mathit{convert} \quad :: \quad \mathit{Liste}\ \alpha \rightarrow [\alpha]$$

que convierta de un tipo de datos al otro.

4.2 Operaciones sobre listas

Esta sección introduce algunas de las funciones y operaciones más usadas sobre listas. Para cada función daremos su definición, ilustraremos su uso y estableceremos algunas de sus propiedades. También emplearemos la inducción estructural sobre listas para demostrar estas propiedades.

4.2.1 Concatenación

Se pueden concatenar dos listas para formar una lista más grande. Esta función se denota mediante el operador binario ++ (pronunciado *concatenar*). A modo de ejemplos simples tenemos

```
? [1, 2, 3] ++ [4, 5]
[1, 2, 3, 4, 5]

? [1, 2] ++ [] ++ [1]
[1, 2, 1]
```

La definición formal de ++ es

$$\begin{array}{lcl} (++) & :: & [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\ [\,] ++ ys & = & ys \\ (x : xs) ++ ys & = & x : (xs ++ ys) \end{array}$$

La concatenación toma dos listas, ambas del mismo tipo, y produce una tercera lista, también del mismo tipo. De aquí la signatura de tipos. La definición de ++ se hace imponiendo ajuste de patrones sobre el argumento izquierdo; los dos patrones son disjuntos y cubren todos los casos, excepto la lista indefinida $\bot$. Por agotamiento de los patrones, se deduce que $\bot ++ ys = \bot$.

Sin embargo, no es cierto que $xs ++ \bot = \bot$. Por ejemplo,

```
? [1, 2, 3] ++ indefinido
[1, 2, 3 {Interrupted!}
```

La lista $[1, 2, 3] ++ \bot$ es una lista *parcial*; en forma normal es la lista $1 : 2 : 3 : \bot$. El evaluador puede calcular los tres primeros elementos, pero después entra en un cómputo no terminante; por tanto, lo interrumpimos.

La segunda ecuación para ++ es muy sucinta y requiere alguna consideración. Si se comprende bien la definición de ++, se ha comprendido gran parte de cómo funcionan las listas en la programación funcional. Nótese que el número de pasos requeridos para calcular $xs ++ ys$ es proporcional al número de elementos de xs. Por ejemplo,

$$\begin{array}{cl} & [1, 2] ++ [3, 4, 5] \\ = & \quad \{\text{notación}\} \\ & (1 : (2 : [\,])) ++ (3 : (4 : (5 : [\,]))) \\ = & \quad \{\text{segunda ecuación de } ++\} \\ & 1 : ((2 : [\,]) ++ (3 : (4 : (5 : [\,])))) \\ = & \quad \{\text{segunda ecuación de } ++\} \\ & 1 : (2 : ([\,] ++ (3 : (4 : (5 : [\,]))))) \\ = & \quad \{\text{primera ecuación de } ++\} \\ & 1 : (2 : (3 : (4 : (5 : [\,])))) \\ = & \quad \{\text{notación}\} \\ & [1, 2, 3, 4, 5] \end{array}$$

La concatenación es una operación asociativa, cuyo elemento neutro es [ ]:

$$\begin{array}{lcl} (xs ++ ys) ++ zs & = & xs ++ (ys ++ zs) \\ xs ++ [\,] & = & [\,] ++ xs = ys \end{array}$$

Ahora usaremos la inducción sobre listas para demostrar que ++ es asociativa.

Demostración. La demostración se hará por inducción sobre xs.

Caso ($\bot$). Para el lado izquierdo, razonamos

$$\begin{array}{cl} & (\bot ++ ys) ++ zs \\ = & \quad \{\text{agotamiento de patrones}\} \\ & \bot ++ zs \\ = & \quad \{\text{agotamiento de patrones}\} \\ & \bot \end{array}$$

La parte derecha se simplifica también a $\bot$ y, por tanto, se cumple el caso.

Caso ([ ]). Para la parte izquierda, razonamos

$$\begin{array}{cl} & ([\,]) ++ ys) ++ zs \\ = & \quad \{\text{primera ecuación de } ++\} \\ & ys ++ zs \end{array}$$

La parte izquierda se simplifica también a $ys +\!\!+ zs$, lo que demuestra el caso.

Caso $(x : xs)$. Para la parte izquierda, razonamos

$$
\begin{array}{ll}
 & ((x : xs) +\!\!+ ys) +\!\!+ zs \\
= & \quad \{\text{segunda ecuación de } +\!\!+\} \\
 & (x : (xs +\!\!+ ys)) +\!\!+ zs \\
= & \quad \{\text{segunda ecuación de } +\!\!+\} \\
 & x : ((xs +\!\!+ ys) +\!\!+ zs) \\
= & \quad \{\text{hipótesis de inducción}\} \\
 & x : (xs +\!\!+ (ys +\!\!+ zs))
\end{array}
$$

Para la parte derecha, razonamos

$$
\begin{array}{ll}
 & (x : xs) +\!\!+ (ys +\!\!+ zs) \\
= & \quad \{\text{segunda ecuación de } +\!\!+\} \\
 & x : (xs +\!\!+ (ys +\!\!+ zs))
\end{array}
$$

Los dos lados son iguales y, por tanto, el caso queda demostrado.

□

El lector debería considerar por qué en esta demostración se hace inducción sobre *xs* en lugar de sobre *ys* o *zs*. Nótese que se ha demostrado la asociatividad para *todas* las listas, finitas, parciales o infinitas. Por tanto, podemos decir que $+\!\!+$ es asociativo sin más adjetivos. El lector podría comparar la demostración anterior con la demostración similar de que + es una operación asociativa para el tipo *Nat* dada en el capítulo precedente.

Demostraremos ahora un resultado que se cumple para listas parciales e infinitas, pero no para las listas finitas. La propiedad a demostrar es $xs +\!\!+ ys = xs$ para todas las listas parciales *xs*. Dado que la propiedad es una ecuación, también se cumplirá para todas las listas infinitas *xs*.

Demostración. La demostración de la propiedad $xs +\!\!+ ys = xs$ se hará por inducción parcial sobre *xs*.

Caso $(\perp)$. Inmediato a partir del agotamiento de patrones en $+\!\!+$.

Caso $(x : xs)$. Razonamos

$$
\begin{array}{ll}
 & (x : xs) +\!\!+ ys \\
= & \quad \{\text{segunda ecuación de } +\!\!+\} \\
 & x : (xs +\!\!+ ys) \\
= & \quad \{\text{hipótesis de inducción}\} \\
 & x : xs
\end{array}
$$

□

El lector debería comparar esta demostración con otra similar dada en el capítulo precedente, para la propiedad $m + n = n$ para todos los elementos parciales *n* de *Nat*.

Para ilustrar el resultado, considérese la función *from* que devuelve una lista infinita de enteros:

$$
\begin{array}{lll}
from & :: & (Integral\ \alpha) \Rightarrow \alpha \to [\alpha]
\end{array}
$$

$$
\begin{array}{lll}
from\ n & = & n : from\ (n + 1)
\end{array}
$$

La evaluación de *from n* no terminará, por tanto

? *from* 0
[0, 1, 2, 3, 4, 5, 6, 7, {Interrupted!}

Ahora consideremos

? *from* 0 $+\!\!+$ [0]
[0, 1, 2, 3, 4, 5, 6, 7, {Interrupted!}

Ambas evaluaciones, si se continuaran por tiempo suficientemente largo, producirían solamente elementos de la lista infinita [0, 1, ...], que es exactamente lo que predice el resultado.

4.2.2 La función *concat*

La concatenación realiza sobre las listas la misma función que hace la unión $\cup$ sobre los conjuntos. Una función relacionada es *concat*, que contatena una lista de listas en una sola lista. Esta función, que corresponde aproximadamente a la unión extendida $\bigcup$ para conjuntos de conjuntos, se define mediante

$$
\begin{array}{lll}
concat & :: & [[\alpha]] \to [\alpha] \\
concat\ [\,] & = & [\,] \\
concat\ (xs : xss) & = & xs +\!\!+ concat\ xss
\end{array}
$$

Por ejemplo:

? *concat* [[1, 2], [], [3, 2, 1]]
[1, 2, 3, 2, 1]

? *putStr* (*concat* ["Nos", "␣", "gustan", "␣", "las", "␣", "listas."])
Nos gustan las listas.

Una propiedad básica de *concat* es

$$
concat\ (xss +\!\!+ yss) \quad = \quad concat\ xss +\!\!+ concat\ yss
$$

para todo par de listas *xss* e *yss*. Esta propiedad se puede demostrar por inducción sobre *xss*. Se deja como ejercicio.

4.2.3 La función *reverse*

Otra función básica sobre listas es *reverse*, que invierte el orden de los elementos de una lista finita. Por ejemplo:

? *reverse* "Dabale el abad."
".daba le elabaD"

La definición es

$$
\begin{array}{lll}
reverse & :: & [\alpha] \to [\alpha] \\
reverse\ [\,] & = & [\,] \\
reverse\ (x : xs) & = & reverse\ xs +\!\!+ [x]
\end{array}
$$

Dicho en palabras, para invertir una lista $(x : xs)$ se invierte xs y luego se añade x al final. Como programa, la definición anterior no es muy eficiente: para una lista de longitud n, se necesita un número de pasos de reducción proporcional a n^2 para invertir dicha lista. El primer elemento se añadirá al final de una lista de longitud $(n-1)$, lo cual llevará aproximadamente $(n-1)$ pasos, el segundo elemento se añadirá a una lista de longitud $(n-2)$, llevando $(n-2)$ pasos, y así sucesivamente. El tiempo total es, por tanto, aproximadamente

$$(n-1) + (n-2) + \cdots + 1 = n\,(n-1)/2 \quad \text{pasos}$$

En el Capítulo 7 se da un análisis más preciso, y en la Sección 4.5 se da un programa más eficiente para *reverse*.

Demostraremos ahora, por inducción, que

$$reverse\ (reverse\ xs) \quad = \quad xs$$

para todas las listas finitas xs. La demostración es instructiva porque requiere un resultado auxiliar. En lugar de dar el resultado auxiliar antes, comenzaremos con la demostración por inducción y veremos lo que se necesita para completarla.

Demostración. La demostración se hará por inducción sobre xs.

Caso ([]). Para el lado izquierdo, razonamos

$$\begin{array}{ll} & reverse\ (reverse\ []) \\ = & \quad \{\text{primera ecuación de } reverse\} \\ & reverse\ [] \\ = & \quad \{\text{primera ecuación de } reverse\} \\ & [] \end{array}$$

El resultado coincide con el lado derecho.

Caso $(x : xs)$. Para el lado izquierdo, razonamos

$$\begin{array}{ll} & reverse\ (reverse\ (x : xs)) \\ = & \quad \{\text{segunda ecuación de reverse}\} \\ & reverse\ (reverse\ xs \mathbin{++} [x]) \end{array}$$

No podemos transformar más el lado izquierdo, por tanto pasamos al lado derecho:

$$\begin{array}{ll} & x : xs \\ = & \quad \{\text{hipótesis de inducción}\} \\ & x : reverse\ (reverse\ xs) \end{array}$$

Los dos lados se han simplificado a dos expresiones diferentes, por tanto tenemos que demostrar que

$$reverse\ (reverse\ xs \mathbin{++} [x]) \quad = \quad x : reverse\ (reverse\ xs)$$

para todas las listas finitas xs. Tratar de demostrar este resultado directamente por inducción no funciona, como el lector puede verificar. En su lugar, hemos de *generalizar* la ecuación, es decir

$$reverse\ (ys \mathbin{++} [x]) \quad = \quad x : reverse\ ys$$

para todas las listas finitas ys. Aplicándolo al caso particular $ys = reverse\ xs$, y sabiendo que $reverse\ xs$ es una lista finita si xs lo es, podemos completar la demostración del resultado principal.

Para demostrar el resultado auxiliar, necesitamos un segundo argumento de inducción. He aquí el paso de inducción:

Caso $(y{:}ys)$. Para el lado izquierdo, razonamos

$$\begin{array}{ll} & reverse\ ((y : ys) \mathbin{++} [x]) \\ = & \quad \{\text{definición de } ++\} \\ & reverse\ (y : (ys \mathbin{++} [x])) \\ = & \quad \{\text{segunda ecuación de } reverse\} \\ & reverse\ (ys \mathbin{++} [x]) \mathbin{++} [y] \\ = & \quad \{\text{hipótesis de inducción}\} \\ & (x : reverse\ ys) \mathbin{++} [y] \\ = & \quad \{\text{definición de } ++\} \\ & x : (reverse\ ys \mathbin{++} [y]) \\ = & \quad \{\text{segunda ecuación de } reverse\} \\ & x : reverse\ (y : ys) \end{array}$$

El resultado coincide con el lado derecho, quedando establecido el caso y completada la demostración. □

Dado que surgen situaciones similares en muchas demostraciones por inducción, merece la pena reiterar cómo se ha conseguido el resultado principal de la demostración. Primeramente, simplificamos ambos lados de la ecuación en el caso inductivo, llegando a dos expresiones diferentes. Para demostrar que estas dos expresiones eran iguales, necesitamos recurrir a una segunda demostración por inducción. Sin embargo, antes de hacer eso, fue necesario generalizar el resultado auxiliar a una expresión más manejable. Se podría esperar que esto haría la demostración más difícil, pero sucedió lo contrario. La hipótesis de inducción de la ecuación generalizada, al ser más fuerte, puede transferir más información desde un lado de la ecuación al otro.

4.2.4 La función *length*

La longitud de una lista es el número de elementos que contiene:

$$\begin{array}{lcl} length & :: & [\alpha] \to Int \\ length\ [] & = & 0 \\ length\ (x : xs) & = & 1 + length\ xs \end{array}$$

La naturaleza de los elementos de la lista es irrelevante para calcular su longitud, de ahí la signatura de tipos. Por ejemplo,

? *length* [*indefinido, indefinido*]
2

Sin embargo, no todas las listas tienen una longitud bien definida. En particular, las listas parciales $\bot$, $x : \bot$, $x : y : \bot$, y así sucesivamente, tienen una longitud indefinida. Solamente las listas finitas tienen longitud bien definida. La lista $[\bot, \bot]$ es una lista

finita, *no* una lista parcial, porque es la lista $\bot : \bot : []$, que termina en [] y no en $\bot$. El evaluador no puede obtener sus elementos, pero sí puede calcular la longitud de la lista.

Al igual que *concat*, la función *length* satisface una propiedad distributiva:

$$length\ (xs +\!\!+\ ys) \quad = \quad length\ xs + length\ ys$$

La demostración se deja como ejercicio.

4.2.5 Las funciones *head*, *tail*, *init* y *last*

La función *head* obtiene el primer elemento de una lista no vacía y la función *tail* selecciona el resto:

$$\begin{array}{lcl} head & :: & [\alpha] \to \alpha \\ head\ (x : xs) & = & x \\ tail & :: & [\alpha] \to [\alpha] \\ tail\ (x : xs) & = & xs \end{array}$$

Ambas son operaciones que necesitan tiempo constante, dado que devuelven sus resultados en un paso de reducción.

Las funciones relacionadas *last* e *init*, que seleccionan el último elemento de una lista no vacía y lo que queda después de eliminar el último elemento, no pueden ser implementadas tan eficientemente. Una definición breve de *last* sería

$$\begin{array}{lcl} last & :: & [\alpha] \to \alpha \\ last & = & head \cdot reverse \end{array}$$

Sin embargo, la definición análoga de *init*, a saber

$$\begin{array}{lcl} init & :: & [\alpha] \to [\alpha] \\ init & = & reverse \cdot tail \cdot reverse \end{array}$$

define $init\ xs = \bot$ para todas las listas parciales e infinitas *xs* (ver Ejercicio 4.2.6). Una definición mejor de estas dos funciones es

$$\begin{array}{lcl} last\ (x : xs) & = & \textbf{if}\ null\ xs\ \textbf{then}\ x\ \textbf{else}\ last\ xs \\ init\ (x : xs) & = & \textbf{if}\ null\ xs\ \textbf{then}\ []\ \textbf{else}\ x : init\ xs \end{array}$$

Con esta definición, tenemos que se cumple $init\ xs = xs$ para todas las listas infinitas *xs*.

Dado que $[x]$ es una abreviatura de $x : []$, y que puede usarse como patrón en la parte izquierda de las definiciones, se podría estar tentado de escribir las definiciones de *last* e *init* de la forma

$$\begin{array}{lcl} last\ [x] & = & x \\ last\ (x : xs) & = & last\ xs \\ init\ [x] & = & [] \\ init\ (x : xs) & = & x : init\ xs \end{array}$$

Sin embargo, hay un serio peligro de confusión con estas definiciones. Los patrones $[x]$ y $(x : xs)$ *no* son disjuntos porque el segundo incluye al primero como caso particular. Aunque las definiciones son válidas en muchos lenguajes, incluyendo a Haskell, dependen del convenio de que el ajuste de patrones se realice en el orden "de arriba abajo", es decir, desde la primera ecuación hacia la última. Si hubiéramos invertido el orden de las ecuaciones, escribiendo, por ejemplo,

$$\begin{array}{lcl} last'\ (x : xs) & = & last'\ xs \\ last'\ [x] & = & x \end{array}$$

entonces la definición de *last'* sería simplemente incorrecta. De hecho, $last'\ xs = \bot$ para todo *xs*. No es buena práctica escribir definiciones que dependan críticamente del orden en el que se escriben las ecuaciones.

Se pueden resolver estas dificultades con una tercera definición:

$$\begin{array}{lcl} last\ [x] & = & x \\ last\ (x : y : ys) & = & last\ (y : ys) \end{array}$$

En el primer patrón $[x]$ sólo encajan listas unitarias, mientras que en el patrón $x : y : ys$ sólo encajan listas que tienen al menos dos elementos. Por tanto, los dos patrones cubren casos disjuntos y la definición funciona correctamente independientemente del orden de las ecuaciones.

4.2.6 Las funciones *take* y *drop*

Las funcionas *take* y *drop* toman cada una un entero no negativo *n* y una lista *xs* como argumentos. El valor de *take n xs* es la lista constituida por los primeros *n* elementos de la lista *xs* y el de *drop n xs* es lo que queda de la lista. Por ejemplo,

```
? take 3 "funcional"
"fun"

? take 3 [1, 2]
[1, 2]
```

Las definiciones son

$$\begin{array}{lcl} take & :: & Int \to [\alpha] \to [\alpha] \\ take\ 0\ xs & = & [] \\ take\ (n+1)\ [] & = & [] \\ take\ (n+1)\ (x : xs) & = & x : take\ n\ xs \\ \\ drop & :: & Int \to [\alpha] \to [\alpha] \\ drop\ 0\ xs & = & xs \\ drop\ (n+1)\ [] & = & [] \\ drop\ (n+1)\ (x : xs) & = & drop\ n\ xs \end{array}$$

Estas definiciones usan una combinación de ajuste de patrones con números naturales y con listas. Nótese que los patrones son disjuntos y cubren todos los posibles casos: todo número natural, es o bien 0 (primera ecuación) o el sucesor de un número natural; en este último caso, distinguimos entre una lista vacía (segunda ecuación) y una lista no vacía (tercera ecuación). Dado que hay dos argumentos sobre los que se lleva a cabo el ajuste de patrones, es importante saber en qué orden se realiza dicho ajuste. Como hemos dicho, el ajuste se lleva a cabo sobre las ecuaciones de una definición

empezando por la primera y hasta la última. Dentro de cada ecuación, el ajuste de patrones se realiza de izquierda a derecha. Por tanto, en *take* y *drop* el patrón numérico se ajusta primero. En particular, $take\ 0\ \bot = []$ y $take\ \bot\ [] = \bot$.

Podríamos haber escrito la definición de *take* de la forma

$$\begin{array}{lcl} take'\ n\ [] & = & [] \\ take'\ 0\ (x : xs) & = & [] \\ take'\ (n+1)\ (x : xs) & = & x : take'\ n\ xs \end{array}$$

Sin embargo, la definición de *take'* es sutilmente diferente de la de *take*. Ahora, el ajuste de patrones con la primera ecuación da $take'\ \bot\ [] = []$ y el ajuste de patrones con la segunda ecuación da $take'\ 0\ \bot = \bot$.

La definición de *take'* es menos satisfactoria que la de *take* porque el orden en el que se escriben las ecuaciones es significativo. Supongamos que se define *take"* intercambiando las dos primeras ecuaciones de *take'*:

$$\begin{array}{lcl} take''\ 0\ (x : xs) & = & [] \\ take''\ n\ [] & = & [] \\ take''\ (n+1)\ (x : xs) & = & x : take''\ n\ xs \end{array}$$

Tenemos que $take''\ \bot\ [] = \bot$ a causa del patrón numérico de la primera ecuación. Por tanto, $take'' \neq take'$. Pero también $take'' \neq take$ porque $take''\ 0\ \bot = \bot$.

Las funciones *take* y *drop* satisfacen varias leyes útiles, incluyendo la relación básica

$$take\ n\ xs +\!\!+ drop\ n\ xs \quad = \quad xs$$

para todos los números naturales (finitos) n y para todas las listas xs. La demostración es instructiva porque usa una combinación de inducción sobre números naturales y análisis por casos.

Demostración. La demostración se hará por inducción sobre n con un análisis por casos sobre xs.

Caso $(0, xs)$. Para el lado izquierdo, razonamos

$$\begin{array}{ll} & take\ 0\ xs +\!\!+ drop\ 0\ xs \\ = & \quad \{\text{primeras ecuaciones de } take \text{ y } drop\} \\ & [] +\!\!+ xs \\ = & \quad \{\text{primera ecuación de } +\!\!+\} \\ & xs \end{array}$$

Caso $(n+1, \bot)$. Para el lado izquierdo, razonamos

$$\begin{array}{ll} & take\ (n+1)\ \bot +\!\!+ drop\ (n+1)\ \bot \\ = & \quad \{\text{agotamiento de patrones de } take \text{ y } drop\} \\ & \bot +\!\!+ \bot \\ = & \quad \{\text{agotamiento de patrones de } +\!\!+\} \\ & \bot \end{array}$$

Caso $(n+1, [])$. Para el lado izquierdo, razonamos

$$\begin{array}{ll} & take\ (n+1)\ [] +\!\!+ drop\ (n+1)\ [] \\ = & \quad \{\text{definición de } take \text{ y } drop\} \\ & [] +\!\!+ [] \\ = & \quad \{\text{definición de } +\!\!+\} \\ & [] \end{array}$$

Caso $(n+1, x : xs)$. Para el lado izquierdo, razonamos

$$\begin{array}{ll} & take\ (n+1)\ (x : xs) +\!\!+ drop\ (n+1)\ (x : xs) \\ = & \quad \{\text{definición de } take \text{ y } drop\} \\ & (x : take\ n\ xs) +\!\!+ drop\ n\ xs \\ = & \quad \{\text{definición de } +\!\!+\} \\ & x : (take\ n\ xs +\!\!+ drop\ n\ xs) \\ = & \quad \{\text{hipótesis de inducción}\} \\ & x : xs \end{array}$$

□

La demostración es inmediata, pero hay dos aspectos interesantes. En primer lugar, se puede entender la demostración como un argumento de inducción sobre n con un análisis por casos sobre xs. Los diversos casos encajan en los patrones de la definición de *take* y *drop*; de hecho, ésa es precisamente la razón por la que se escogieron tales patrones. La demostración del caso adicional $(n+1, \bot)$ es necesaria si queremos afirmar que la ecuación se cumple sin restricciones para todo xs.

El segundo aspecto a resaltar es que la ecuación no se cumple en el caso $n = \bot$ porque

$$take\ \bot xs +\!\!+ drop\ \bot xs \quad = \quad \bot +\!\!+ \bot = \bot, \text{ y no } xs$$

De modo similar, se pueden demostrar diversas leyes adicionales acerca de *take* y *drop*, entre las cuales están

$$\begin{array}{lcl} take\ m \cdot take\ n & = & take\ (m\ \mathbf{min}\ n) \\ drop\ m \cdot drop\ n & = & drop\ (m+n) \\ take\ m \cdot drop\ n & = & drop\ n \cdot take\ (m+n) \end{array}$$

La expresión $(m\ \mathbf{min}\ n)$ denota el mínimo de m y n.

En ciertas situaciones se necesita el valor $(take\ n\ xs,\ drop\ n\ xs)$. En Haskell existe una función estándar para calcularlo. La función *splitAt* se especifica mediante

$$\begin{array}{lcl} splitAt & :: & Int \to [\alpha] \to ([\alpha], [\alpha]) \\ splitAt\ n\ xs & = & (take\ n\ xs,\ drop\ n\ xs) \end{array}$$

Sin embargo, *splitAt* se implementa mediante las ecuaciones

$$\begin{array}{lcl} splitAt\ 0\ xs & = & ([], xs) \\ splitAt\ (n+1)\ [] & = & ([], []) \\ splitAt\ (n+1)\ (x : xs) & = & (x : ys, zs) \\ & & \quad \mathbf{where}\ (ys, zs) = splitAt\ n\ xs \end{array}$$

Es fácil derivar la segunda versión y se dejan los detalles como ejercicio. Como veremos en el Capítulo 7, el programa alternativo es más eficiente en cuanto al espacio de memoria consumido porque la lista argumento se recorre sólo una vez. La diferencia de tiempo entre ambas versiones es muy pequeña.

4.2.7 Indexación de listas

Para hallar el elemento que ocupa la posición n, se puede indexar una lista mediante un número natural n. Esta operación se denota $xs \,!!\, n$. Por ejemplo,

```
? [1, 2, 3, 4] !! 2
3

? [1, 2, 3, 4] !! 0
1
```

Los índices comienzan por 0, que es una elección preferible a que comiencen por 1 para la mayoría de los usos.

El operador (!!) se define mediante

$$\begin{array}{lcl} (!!) & :: & [\alpha] \to Int \to \alpha \\ (x : xs) \,!!\, 0 & = & x \\ (x : xs) \,!!\, (n+1) & = & xs \,!!\, n \end{array}$$

La indexación de listas se corresponde con el uso de los subíndices en matemáticas y, por tanto, parece una operación muy natural y útil. Sin embargo, la indexación es una operación bastante costosa en la programación funcional, ya que $xs \,!!\, n$ necesita un número de pasos de reducción proporcional a n y debería evitarse su uso siempre que haya otra alternativa. Por otra parte, hay situaciones en las que indexar una lista es apropiado. En la Sección 5.1, que depende exclusivamente del material presentado hasta el momento, se da un ejemplo de esto.

Ejercicios

4.2.1 ¿Cuáles de las siguientes ecuaciones se cumplen para todo xs y cuáles son falsas?

$$\begin{array}{lcl} [\,] : xs & = & xs \\ [\,] : xs & = & [[\,], xs] \\ xs : [\,] & = & xs \\ xs : [\,] & = & [xs] \\ xs : xs & = & [xs, xs] \end{array}$$

4.2.2 ¿Cuáles de las siguientes ecuaciones se cumplen para todo xs y cuáles son falsas?

$$\begin{array}{lcl} [[\,]] ++ xs & = & xs \\ [[\,]] ++ xs & = & [xs] \\ [[\,]] ++ xs & = & [[\,], xs] \\ [[\,]] ++ [xs] & = & [[\,], xs] \\ [xs] ++ [\,] & = & [xs] \\ [xs] ++ [xs] & = & [xs, xs] \end{array}$$

4.2.3 Demostrar por inducción que $xs ++ [\,] = xs$ para toda lista xs.

4.2.4 Demostrar que $concat\ (xss ++ yss) = concat\ xss ++ yss$.

4.2.5 Demostrar que $length\ (xs ++ ys) = length\ xs + lenght\ ys$.

4.2.6 Demostrar que $reverse\ xs = \bot$ para todas las listas infinitas xs, y demostrar que $init\ xs = \bot$ para todas las listas xs, siendo $init = reverse \cdot tail \cdot reverse$.

4.2.7 Considérese otra vez la propiedad $reverse\ (reverse\ xs) = xs$. Dado que

$$reverse\ (reverse\ \bot) = reverse\ \bot = \bot$$

parece que la propiedad se demuestra para todas las listas y no sólo para las finitas. Pero no se puede invertir una lista infinita (ver el ejercicio precedente) de modo que, ¿en qué nos hemos equivocado?

4.2.8 Usando ajuste de patrones con (:), definir una función *reverse2* que invierta todas las listas de longitud 2 y deje las demás sin modificar. Asegurarse de que los patrones son exhaustivos y disjuntos.

4.2.9 Dar una caracterización informal de las listas finitas xs e ys que satisfacen $xs ++ ys = ys ++ xs$.

4.2.10 ¿Cuál es el valor de $[head\ xs] ++ tail\ xs$ si $xs = [\,]$?

4.2.11 ¿Se cumple que $(drop\ m\ xs) \,!!\, n = xs \,!!\, (m + n)$ para todas las listas finitas xs y todos los números naturales m y n?

4.2.12 Demostrar que

$$\begin{array}{lcl} (xs ++ ys) \,!!\, k & = & \textbf{if } k < n \textbf{ then } xs \,!!\, k \textbf{ else } ys \,!!\, (k - n) \\ & & \textbf{where } n = lenght\ xs \end{array}$$

4.2.13 Dado que la concatenación parece ser una operación básica sobre listas, podemos tratar de construir un tipo de datos que utilice la concatenación como primitiva. Por ejemplo,

$$\textbf{data}\ CatList\ \alpha \quad = \quad Nil \mid Unit\ \alpha \mid Concat\ (CatList\ \alpha)\ (CatList\ \alpha)$$

La intención es que *Nil* represente $[\,]$, $Unit\ x$ represente $[x]$ y $Concat\ xs\ ys$ represente $xs ++ ys$. Sin embargo, dado que ++ es una operación asociativa, las expresiones

$$Concat\ xs\ (Concat\ ys\ zs) \qquad y \qquad Concat\ (Concat\ xs\ ys)\ zs$$

deberían ser iguales. Definir concreciones apropiadas de *Eq* y *Ord* para el tipo *CatList*.

4.3 Las funciones *map* y *filter*

Se trata de dos funciones útiles para operar con listas. La función *map* aplica una función a cada elemento de una lista. Por ejemplo,

```
? map cuadrado [9, 3]
[81, 9]

? map (< 3) [1, 2, 3]
[True, True, False]

? map siguienteLetra "HAL"
"IBM"
```

Su definición es

$$\begin{array}{lcl} map & :: & (\alpha \to \beta) \to [\alpha] \to [\beta] \\ map\ f\ [\,] & = & [\,] \\ map\ f\ (x : xs) & = & f\,x : map\ f\ xs \end{array}$$

Como un ejemplo más, consideremos el cálculo de la suma de los cuadrados de los números enteros entre 1 y 100. Las funciones *sum* y *upto* se pueden definir mediante

$$\begin{array}{lcl} sum & :: & (Num\ \alpha) \Rightarrow [\alpha] \to \alpha \\ sum\ [\,] & = & 0 \\ sum\ (x : xs) & = & x + sum\ xs \\ \\ upto & :: & (Integral\ \alpha) \Rightarrow \alpha \to \alpha \to [\alpha] \\ upto\ m\ n & = & \textbf{if}\ m > n\ \textbf{then}\ [\,]\ \textbf{else}\ m : upto\ (m+1)\ n \end{array}$$

Entonces, tenemos

? *sum* (*map cuadrado* (*upto* 1 100))
338700

El uso de *map* queda perfectamente ilustrado en este ejemplo: la frase en español "la suma de los cuadrados de los números enteros entre 1 y 100" se puede traducir literalmente, palabra por palabra, a una expresión formal simple.

La función *upto* y la función relacionada *from* aparecen frecuentemente en los cálculos. Haskell suministra una sintaxis especial y más conveniente para ella. Las expresiones [*m*..*n*] y [*m*..] se definen mediante

$$\begin{array}{lcl} [m..n] & = & upto\ m\ n \\ [m..] & = & from\ m \end{array}$$

Estas formas especiales se pueden usar con tipos enumerados distintos de *Int* e *Integer*, pero no lo haremos en este libro.

Existen bastante leyes algebraicas útiles relacionadas con *map*. Dos hechos básicos son

$$\begin{array}{lcl} map\ id & = & id \\ map\ (f \cdot g) & = & map\ f \cdot map\ g \end{array}$$

La primera ecuación expresa que al aplicar la función identidad a todos los elementos de una lista, la lista queda sin modificar. Las dos apariciones de *id* en la primera ecuación tienen tipos diferentes; el de la izquierda es $id :: \alpha \to \alpha$ y el de la derecha $id :: [\alpha] \to [\alpha]$. La segunda ecuación expresa que aplicar primero g a todo elemento de una lista y después aplicar f a todo elemento de la lista resultado, da el mismo resultado que aplicar $f \cdot g$ a la lista original. Leída de derecha a izquierda, la ecuación permite reemplazar dos recorridos de una lista por uno solo, con la correspondiente ganancia en eficiencia. Las demostraciones de ambas leyes se realizan por inducción sobre listas y se dejan como ejercicios. En la teoría de categorías, las funciones como *map* que satisfacen las dos propiedades básicas anteriores, se llaman *funtores*; por tanto, usaremos la palabra funtor cuando nos refiramos a estas dos propiedades dentro de los cálculos.

He aquí una lista de leyes adicionales, todas sobre este tema común:

$$\begin{array}{lcl} f\,.\,head & = & head \cdot map\ f \\ map\ f \cdot tail & = & tail \cdot map\ f \\ map\ f \cdot reverse & = & reverse \cdot map\ f \\ map\ f \cdot concat & = & concat \cdot map\ (map\ f) \end{array}$$

La primera ecuación se cumple sólo si f es una función estricta pero las otras tres se cumplen para funciones f arbitrarias. Si aplicamos ambos lados de la primera ecuación a [], entonces

$$f\,(head\,[\,]) = head\,(map\ f\,[\,]) = head\,[\,]$$

Por tanto, y dado que $head\,[\,] = \bot$, requerimos que f sea estricta.

Cada una de estas ecuaciones tiene una interpretación simple. Por ejemplo, la última ecuación dice que podemos o bien concatenar una lista de listas y aplicar f a cada elemento del resultado, o bien aplicar $map\ f$ a cada lista componente y después concatenar las listas resultantes.

Dado que $concat\,[xs, ys] = xs +\!\!+\, ys$, un caso particular del último resultado es que

$$map\ f\,(xs +\!\!+ ys) \quad = \quad map\ f\,xs +\!\!+ f\,ys$$

para todo par de listas *xs* e *ys*. Dicho con palabras, $map\ f$ se distribuye sobre ++ para todo f. De hecho, el caso particular se necesita como resultado auxilar durante la demostración por inducción del caso más general.

El punto en común que subyace a todas estas ecuaciones, y a muchas otras similares, tiene que ver con los tipos de las funciones involucradas:

$$\begin{array}{lcl} head & :: & [\alpha] \to \alpha \\ tail & :: & [\alpha] \to [\alpha] \\ reverse & :: & [\alpha] \to [\alpha] \\ concat & :: & [[\alpha]] \to [\alpha] \end{array}$$

El aspecto común que relaciona estas funciones es que las mismas no dependen en modo alguno de la naturaleza de los elementos de la lista; son simplemente combinadores que barajan, reordenan o extraen elementos de las listas. Por eso que tienen tipos polimórficos. De todo esto se desprende que se puede, o bien "renombrar" los elementos de la lista (utilizando $map\ f$) y después hacer la operación, o bien hacer la operación y después renombrar los elementos. Esto es exactamente lo que dicen estas ecuaciones. En la teoría de categorías, a tales ecuaciones se las llama *condiciones de naturalidad*, terminología que usaremos en los cálculos siguientes.

4.3.1 La función *filter*

La segunda función, *filter*, toma una función booleana p y una lista *xs* y devuelve la sublista de *xs* cuyos elementos satisfacen p. Por ejemplo,

? *filter even* [1, 2, 4, 5, 32]
[2, 4, 32]

? (*sum* · *map cuadrado* · *filter even*) [1..10]
220

El último ejemplo calcula la suma de los cuadrados de los números enteros pares en el rango de 1 a 10.

La definición de *filter* es

```
filter          :: (α → Bool) → [α] → [α]
filter p []     =  []
filter p (x : xs) = if p x then x : filter p xs else filter p xs
```

Al igual que *map*, existen bastantes leyes útiles relacionadas con *filter*, incluyendo el hecho básico siguiente:

$$filter\ p \cdot filter\ q \quad = \quad filter\ (p\ \mathbf{and}\ q)$$

donde $(p\ \mathbf{and}\ q)\ x = p\ x \wedge q\ x$. Esta ecuación es válida solamente si p y q son funciones estrictas. Otra ley es

$$filter\ p \cdot concat \quad = \quad concat \cdot map\ (filter\ p)$$

que es válida para todo p.

4.3.2 Listas intensionales

Análogamente a otros lenguajes funcionales, Haskell proporciona una notación alternativa, las llamadas *listas intensionales*, para describir cálculos que necesitan *map* y *filter*. He aquí un ejemplo:

```
? [x × x | x ← [1..5], odd x]
[1, 9, 25]
```

La expresión se lee: la lista de los cuadrados de los números impares en el rango 1 a 5.

Formalmente, una lista intensional es de la forma $[e \mid Q]$, donde e es una expresión, Q es un *cualificador*. Un cualificador es una secuencia, posiblemente vacía, de *generadores* y de *guardas* separados por comas. Un generador toma la forma $x \leftarrow xs$, donde x es una variable o tupla de variables, y xs es una expresión de tipo lista. Una guarda es una expresión booleana. El cualificador Q puede ser vacío, en cuyo caso escribimos simplemente $[e]$. En otro caso definimos

```
[e | x ← xs, Q]  =  concat (map f xs) where f x = [e | Q]
[e | p, Q]       =  if p then [e | Q] else []
```

A la primera ecuación la llamaremos *regla del generador*, y a la segunda *regla de la guarda*. Usando estas reglas, podemos calcular

```
    [x × x | x ← [1..5], odd x]
=       {regla del generador}
    concat (map f [1..5]) where f x = [x × x | odd x]
=       {regla de la guarda}
    concat (map f [1..5]) where f x = if odd x then [x × x] else []
=       {map}
    concat [[1], [], [9], [], [25]]
=       {concat}
    [1, 9, 25]
```


He aquí algunos ejemplos más:

```
? [(a, b) | a ← [1..3], b ← [1..2]]
[(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]
? [(a, b) | b ← [1..2], a ← [1..3]]
[(1, 1), (2, 1), (3, 1), (1, 2), (2, 2), (3, 2)]
```

Los últimos generadores pueden depender de variables introducidas por los precedentes:

```
? [(i, j) | i ← [1..4], j ← [i + 1..4]]
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

Podemos intercalar libremente generadores y guardas:

```
? [(i, j) | i ← [1..4], even i, j ← [i + 1..4], odd j]
[(2, 3)]
```

He aquí un ejemplo en el que el generador tiene la forma $(x, y) \leftarrow xs$. Primero, definimos

```
pares = [(i, j) | i ←[1..2], j ←[1..3]]
```

Entonces tenemos

```
? [i + j | (i, j) ← pares]
[2, 3, 4, 3, 4, 5]
```

Como ejemplo de mayor interés, he aquí un programa que lista todas las triadas pitagóricas en un rango dado. Éstas consisten en ternas de números (x, y, z) tales que $x^2 + y^2 = z^2$. Definimos *triadas* mediante

```
triadas        :: Int → [(Int, Int, Int)]
triadas n      =  [(x, y, z) | (x, y, z) ← triplas n, pitag (x, y, z)]
triplas        :: Int → [(Int, Int, Int)]
triplas n      =  [(x, y, z) | x ←[1..n], y ←[1..n], z ←[1..n]]

pitag          :: (Int, Int, Int) → Bool
pitag (x, y, z) = (x × x + y × y = z × z)
```

Por ejemplo:

```
? triadas 5
[(3, 4, 5), (4, 3, 5)]
```

Dado que *triadas* genera esencialmente la misma triada de dos modos diferentes, la siguiente definición de *triplas* es mejor:

```
triplas n  =  [(x, y, z) | x ←[1..n], y ←[x..n], z ←[y..n]]
```

En la nueva definición, el valor de y está restringido al rango $x \le y \le n$ y el valor de z al rango $y \le z \le n$. Dado que z debe ser al menos tan grande como el mayor de x y y, todavía podemos calcular todas las triadas, pero ninguna de ellas será computada más de una vez. Por ejemplo,

```
? triadas 5
[(3, 4, 5)]
```

4.3.3 Razonando sobre las listas intensionales

Las listas intensionales proporcionan una alternativa atractiva al estilo combinatorio basado en *map*, *filter* y *concat*. Existen numerosas reglas derivadas para razonar sobre listas intensionales, incluyendo las siguientes:

$$\begin{array}{lcl} [f\,x \mid x \leftarrow xs] & = & map\ f\ xs \\ [x \mid x \leftarrow xs,\ p\ x] & = & filter\ p\ xs \\ [e \mid Q,\ P] & = & concat\ [[e \mid P] \mid Q] \\ [e \mid Q,\ x \leftarrow [d \mid P]] & = & [e\,[x := d] \mid Q,\ P] \end{array}$$

En la última ecuación, $e\,[x := d]$ denota la expresión que resulta cuando las apariciones de x en la expresión e se reemplazan por la expresión d. Las últimas dos reglas se pueden demostrar por inducción sobre la estructura del cualificador Q.

Por otro lado, el estilo combinatorio también tiene algunas ventajas. Una de ellas es que las propiedades algebraicas de *map* y *filter* son más fáciles de establecer y de usar. La conclusión más justa es que la elección entre usar listas intensionales o usar un estilo combinatorio es básicamente una cuestión de gusto personal. Lo mismo se puede decir sobre usar un estilo aplicativo frente a un estilo no aplicativo.

Ejercicios

4.3.1 Evaluar *map* (*map cuadrado*) [[1, 2], [3, 4, 5]].

4.3.2 ¿Cuál es el valor de $map\ f\ \bot$? ¿Cuál es el valor de $map\ \bot\ []$?

4.3.3 ¿Cuál es el tipo de *map map*?

4.3.4 Demostrar que $f\ (xs +\!\!+\ ys) = map\ f\ xs +\!\!+\ map\ f\ ys$. Por tanto, demostrar que $map\ f \cdot concat = concat \cdot map\ (map\ f)$.

4.3.5 La función *inits* calcula la lista de los segmentos iniciales de una lista; su tipo es $inits :: [\alpha] \to [[\alpha]]$. ¿Cuál es la condición de naturalidad apropiada para *inits*?

4.3.6 Se puede definir la función *filter* en términos de *concat* y de *map*:

$$\begin{array}{lcl} filter\ p & = & concat \cdot map\ misterio \\ & & \textbf{where}\ misterio\ x = \ldots \end{array}$$

Dar la definición de *misterio*.

4.3.7 Dos nuevas funciones útiles son *takeWhile* y *dropWhile*, que son similares a *take* y *drop*, salvo que ambas toman como primer argumento una función booleana en lugar de un número natural. En este sentido, también son similares a *filter*. El valor de $takeWhile\ p\ xs$ es el mayor segmento inicial de xs tal que todos sus elementos satisfacen p. Por ejemplo,

$$takeWhile\ even\ [2, 4, 6, 1, 5, 6] \quad = \quad [2, 4, 6]$$

El valor de $dropWhile\ p\ xs$ es la lista remanente; por ejemplo,

$$dropWhile\ even\ [2, 4, 6, 1, 5, 6] \quad = \quad [1, 5, 6]$$

Dar definiciones recursivas para *takeWhile* y *dropWhile*.

4.3.8 ¿Bajo qué condiciones sobre xs e ys se cumple la siguiente ecuación?

$$[x \mid x \leftarrow xs,\ y \leftarrow ys] \quad = \quad [x \mid y \leftarrow ys,\ x \leftarrow xs]$$

4.3.9 Definir una función *pares* de modo que *pares n* sea la lista de todos los pares distintos de números enteros que cumplen $1 \leq x, y \leq n$.

4.3.10 Escribir un programa que encuentre todas las cuádruplas (a, b, c, d) esencialmente distintas en el rango $0 < a, b, c, d \leq n$ tales que $a^2 + b^2 = c^2 + d^2$.

4.3.11 Traducir las siguientes listas intensionales a estilo combinatorio:

$$[(x, y) \mid x \leftarrow [1..n],\ odd\ x,\ y \leftarrow [1..n]]$$
$$[(x, y) \mid x \leftarrow [1..n],\ y \leftarrow [1..n],\ odd\ x]$$

¿Son iguales? Comparar el coste de evaluar cada una de las expresiones.

4.4 La función *zip*

La función *zip* toma como argumentos dos listas y devuelve la lista de los pares de elementos que se corresponden. Por ejemplo,

```
? zip [0..3] "hola"
[(0, 'h'), (1, 'o'), (2, 'l'), (3, 'a')]

? zip [0, 1] "hola"
[(0, 'h'), (1, 'o')]
```

Como muestra el segundo ejemplo, si las dos listas no tienen la misma longitud, la longitud de la lista resultante es la menor de las longitudes de los dos argumentos. La definición completa de *zip* es

$$\begin{array}{lcl} zip & :: & [\alpha] \to [\beta] \to [(\alpha, \beta)] \\ zip\ []\ ys & = & [] \\ zip\ (x : xs)\ [] & = & [] \\ zip\ (x : xs)\ (y : ys) & = & (x, y) : zip\ (xs, ys) \end{array}$$

El ajuste de patrones en esta definición es análogo al de *take* y *drop*. El ajuste sobre el argumento izquierdo se realiza antes; por tanto, $zip\ \bot\ [] = \bot$ y $zip\ []\ \bot = []$.

La función *zip* tiene muchos usos. Por ejemplo, el producto escalar de dos vectores x e y se define mediante $\sum_{i=1}^{n} x_i \times y_i$. La función *pe*, para calcular productos escalares, se puede definir mediante

$$\begin{array}{lcl} pe & :: & (Num\ \alpha) \Rightarrow [\alpha] \to [\alpha] \to \alpha \\ pe\ xs\ ys & = & sum\ (map\ mult\ (zip\ xs\ ys)) \\ & & \textbf{where}\ mult\ (x, y) = x \times y \end{array}$$

La función *mult* es la versión descurrificada de ×; de hecho, *mult* = *uncurry* (×).

Como segundo ejemplo, supongamos que deseamos definir una función *nodec* para determinar si una secuencia $[x_0, \ldots, x_{n-1}]$ es no decreciente. Es decir, queremos determinar si $x_k \leq x_{k+1}$ para todo k en el rango $0 \leq k \leq n-2$. El tipo de *nodec* es

```
nodec  ::  (Ord α) ⇒[α] →Bool
```

Es fácil dar una definición recursiva directa de *nodec*, pero la siguiente definición también es correcta:

```
nodec xs  =  and (map menorIgual (zip xs (tail xs)))
             where menorIgual (x, y) = (x ≤ y)
```

La función *and* toma una lista de booleanos y devuelve *True* si todos los elementos son *True*; en otro caso, devuelve *False*:

```
and        ::  [Bool] →Bool
and []     =   True
and (x : xs)  =  x ∧ and xs
```

La expresión *zip xs* (*tail xs*) devuelve la lista de los pares adyacentes de los elementos de *xs*. En particular,

```
zip [] (tail [])    =  zip [] ⊥= []
zip [x] (tail [x])  =  zip [x] [] = []
```

Nótese, además, que *menorIgual* = *uncurry* (≤).

Como tercer ejemplo, considérese el problema de definir una función *posicion* con la propiedad de que *posicion x xs* denota la posición de la primera aparición de *x* en *xs* (contando desde 0), y −1 si *x* no aparece en *xs*. Por tanto, el tipo de *posicion* es

```
posicion  ::  (Eq α) ⇒α →[α] →Int
```

Éste es un problema instructivo porque el mejor modo de abordarlo es resolver primero un problema más general: el cálculo de *todas* las posiciones en las que aparece *x* en *xs*. La definición, que emplea una lista intensional, es

```
posiciones      ::  (Eq α) ⇒α →[α] →[Int]
posiciones x xs =  [i | (i, y) ←zip [0 ..] xs, x == y]
```

La expresión *zip* [0..] *xs* empareja cada elemento de *xs* con su posición en *xs*. Si bien [0..] es una lista infinita, *zip* [0..] *xs* devolverá una lista finita siempre que *xs* sea finita. Ahora podemos definir *posicion* mediante

```
posicion x xs  =  head (posiciones x xs ++[−1])
```

Es curioso observar que la simplicidad de esta definición se logra sin incrementar el coste de evaluación. Para obtener la cabeza de una lista, no es necesario determinar el valor de todos los elementos de la lista.

Estos ejemplos ilustran que las combinaciones de *map*, *filter* y *zip* tienen múltiples usos. Para hacer las definiciones más breves, podemos introducir la función *zipWith*, definida mediante

```
zipWith        ::  (α →β →γ) →([α] →[β] →[γ])
zipWith f xs ys  =  map (uncurry f) (zip xs ys)
```


Por ejemplo,

```
pe xs ys  =  sum (zipWith (×) xs ys)
nodec xs  =  and (zipWith (≤) xs (tail xs))
```

Para mayor brevedad podemos introducir las versiones no currificadas de *zip* y *zipWith*.

```
zipp      ::  ([α], [β]) →[(α, β)]
zipp      =   uncurry zip

zippWith  ::  (α →β →γ) →([α], [β]) →[γ]
zippWith  =   map (uncurry f) · zipp
```

Por ejemplo, siendo *pep* la versión no currificada de *pe*, tenemos

```
pep   =  sum · zippWith (×)
nodec =  and · zippWith (≤) · par (id, tail)
```

La nueva definición de *nodec* usa la función

```
par (f, g) x  =  (f x, g x)
```

definida en la Sección 2.4. En las siguientes definiciones veremos que es posible utilizar un estilo no aplicativo solamente si algunas funciones se dan en forma no currificada. Haskell, al igual que Miranda y otros lenguajes funcionales, define todos los operadores binarios como funciones currificadas y lo mismo sucede con *zip*; por tanto, la conversión a formas no currificadas ha de ser explícita.

4.4.1 La función *unzip*

La función *unzip* toma como argumentos una lista de pares y los separa en dos listas:

```
unzip  ::  [(α, β)] →([α], [β])
unzip  =   par (map fst, map snd)
```

Por ejemplo,

```
? unzip [(1, True), (2, True), (3, False)]
([1, 2, 3], [True, True, False])
```

Las funciones *zipp* (versión no currificada de *zip*) y *unzip* estan relacionadas mediante la ecuación

```
zipp (unzip xys)  =  xys
```

para todas las listas *xys* de pares bien definidos (ver el Ejercicio 4.4.1 para encontrar la razón de esta restricción a pares bien definidos).

Dado que *zip* y *unzip* tienen tipos polimórficos, podríamos esperar que satisficieran condiciones de naturalidad apropiadas del estilo de las descritas en la sección precedente. Por ejemplo, para renombrar los elementos del tipo resultante de *unzip*, necesitamos dos funciones: una función *map f* para renombrar el primer componente, y una función *map g* para renombrar el segundo. La función que hace ambos renombramientos es *prod* (*map f*, *map g*), donde *prod* se define mediante

```
prod (f, g)  =  par (f · fst, g · snd)
```

Esta función ya fue introducida en la Sección 2.4. Para renombrar los elementos del argumento de *unzip*, necesitamos aplicar una única función, y no es difícil ver que la función *map* (*prod* (*f*, *g*)) realiza el trabajo. Por tanto, deberíamos esperar que

$$prod\ (map\ f,\ map\ g) \cdot unzip \quad = \quad unzip \cdot map\ (prod\ (f,\ g))$$

De hecho, esta ecuación se satisface. Su demostración se pide en el Ejercicio 4.4.3. El resultado dual

$$zipp \cdot prod\ (map\ f,\ map\ g) \quad = \quad map\ (prod\ (f,\ g)) \cdot zipp$$

también se cumple.

En la Sección 5.2 se presenta un ejercicio más extenso que usa *zip*. Este ejercicio también utiliza material de la sección siguiente.

Ejercicios

4.4.1 ¿Qué valor tiene *unzip* [⊥]? Usando el hecho de que (⊥, ⊥) ≠ ⊥, construir una lista *xys* tal que *zipp* (*unzip xys*) ≠ *xys*.

4.4.2 ¿Se cumple la ecuación inversa *unzip* · *zipp* = *id*?

4.4.3 Suponiendo que se cumplen otras leyes, se puede demostrar la ecuación

$$prod\ (map\ f,\ map\ g) \cdot unzip = unzip \cdot map\ (prod\ (f,\ g))$$

mediante un cálculo simple con funciones. Las leyes adicionales necesarias son

$$\begin{array}{lcl} map\ (f \cdot g) & = & map\ f \cdot map\ g \\ prod\ (f,\ g) \cdot par\ (h,\ k) & = & par\ (f \cdot h,\ g \cdot k) \\ par\ (f,\ g) \cdot h & = & par\ (f \cdot h,\ g \cdot h) \\ fst \cdot prod\ (f,\ g) & = & f \cdot fst \\ snd \cdot prod\ (f,\ g) & = & g \cdot snd \end{array}$$

Usando estas leyes, junto con la definición de *unzip*, demostrar el resultado pedido.

4.5 La familia de funciones *fold*

En el caso del tipo de datos de *Nat*, hemos visto que muchas definiciones recursivas se pueden expresar de modo sucinto usando una vesión apropiada del operador *fold*. Exactamente la misma idea es aplicable a las listas. Considérese la siguiente definición de la función *h*:

$$\begin{array}{lcl} h\ [\,] & = & e \\ h\ (x : xs) & = & x \oplus h\ xs \end{array}$$

La función *h* toma una lista, reemplaza [] por *e* y (:) por ⊕, y evalúa el resultado. Por ejemplo, *h* convierte la lista

$$x_1 : (x_2 : (x_3 : (x_4 : [\,])))$$

en el valor

$$x_1 \oplus (x_2 \oplus (x_3 \oplus (x_4 \oplus e)))$$

Dado que (:) asocia a la derecha, no hace falta poner paréntesis en la primera expresión. Sin embargo, hemos puesto paréntesis en la segunda expresión porque no hemos de suponer que ⊕ asocia a la derecha.

El esquema de definición expresado por *h* es capturado por la función *foldr* definida como sigue:

$$\begin{array}{lcl} foldr & :: & (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\ foldr\ f\ e\ [\,] & = & e \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs) \end{array}$$

Ahora podemos escribir $h = foldr\ (\oplus)\ e$. El primer argumento de *foldr* es un operador binario que necesita a la izquierda un valor de tipo α, a la derecha un valor de tipo β y devuelve un valor de tipo β. El segundo argumento de *foldr* es un valor de tipo β. El tercer argumento es de tipo $[\alpha]$ y el resultado es un valor de tipo β. En muchos casos, α y β pueden concretarse al mismo tipo; por ejemplo, cuando ⊕ denota una operación asociativa.

Usando sólo *foldr* se puede definir casi cualquiera de las funciones sobre listas que hemos visto hasta ahora. He aquí sólo algunos ejemplos:

$$\begin{array}{lcl} concat & :: & [[\alpha]] \to [\alpha] \\ concat & = & foldr\ (++)\ [\,] \\ & & \\ reverse & :: & [\alpha] \to [\alpha] \\ reverse & = & foldr\ snoc\ [\,] \\ & & \quad \textbf{where}\ snoc\ x\ xs = xs ++ [x] \\ & & \\ length & :: & [\alpha] \to Int \\ length & = & foldr\ increm\ 0 \\ & & \quad \textbf{where}\ increm\ x\ n = 1 + n \\ & & \\ sum & :: & Num\ \alpha \Rightarrow [\alpha] \to \alpha \\ sum & = & foldr\ (+)\ 0 \\ & & \\ and & :: & [Bool] \to Bool \\ and & = & foldr\ (\wedge)\ True \\ & & \\ map & :: & (\alpha \to \beta) \to [\alpha] \to [\beta] \\ map\ f & = & foldr\ (cons \cdot f)\ [\,] \\ & & \quad \textbf{where}\ cons\ x\ xs = x : xs \\ & & \\ unzip & :: & [(\alpha,\ \beta)] \to ([\alpha],\ [\beta]) \\ unzip & = & foldr\ conss\ ([\,],\ [\,]) \\ & & \quad \textbf{where}\ conss\ (x,\ y)\ (xs,\ ys) = (x : xs,\ y : ys) \end{array}$$

En algunas de estas definiciones, el operador ⊕ que reemplaza a (:) es asociativo y la constante *c* que reemplaza a [] es el elemento neutro de ⊕. Por ejemplo, (++, []),

(+, 0), (∧, *True*), y así sucesivamente. En otras definiciones, ⊕ no es asociativo; de hecho ⊕ puede incluso no tener el tipo apropiado para ser un operador asociativo, a saber, $\alpha \to \alpha \to \alpha$.

No todas las funciones sobre listas se pueden definir como concreciones de *foldr*. Por ejemplo, no es posible definir la función *zip*. Incluso para aquellas que sí pueden ser definidas de esta forma, una definición alternativa puede ser más eficiente. Para ilustrar este punto, supongamos que deseamos definir una función *decimal* que toma una lista de dígitos y devuelve el correspondiente número decimal; así,

$$decimal\,[x_0, x_1, \ldots, x_n] = \sum_{k=0}^{n} x_k 10^{(n-k)}$$

Se supone que el dígito más significativo aparece primero en la lista. Una manera de definir *decimal* eficientemente consiste en multiplicar cada dígito por diez y sumarle el siguiente dígito. Por ejemplo,

$$decimal\,[x_0, x_1, x_2] \quad = \quad 10 \times (10 \times (10 \times 0 + x_0) + x_1) + x_2$$

Esta forma de descomponer una suma de potencias se conoce como la *regla de Horner*. Supongamos que definimos ⊕ como $n \oplus x = 10 \times n + x$. Entonces, podemos replantear la ecuación anterior como

$$decimal\,[x_0, x_1, x_2] \quad = \quad ((0 \oplus x_0) \oplus x_1) \oplus x_2$$

Esta expresión corresponde casi a una concreción de *foldr*, salvo que el agrupamiento es por la izquierda y que el valor inicial aparece también a la izquierda, en vez de por la derecha. De hecho, el cómputo es dual. En lugar de procesar de derecha a izquierda, el cálculo procede de izquierda a derecha.

Este ejemplo motiva la introducción de un segundo operador de la familia *fold* llamado *foldl*. Informalmente,

$$foldl\,(\oplus)\,e\,[x_0, x_1, \ldots x_{n-1}] \quad = \quad (\cdots((e \oplus x_0) \oplus x_1)\cdots) \oplus x_{n-1}$$

Los paréntesis asocian por la izquierda, de ahí el nombre del operador. La definición completa de *foldl* es

```
foldl            :: (β → α → β) → β → [α] → β
foldl f e []     =  e
foldl f e (x : xs) = foldl f (f e x) xs
```

Por ejemplo,

```
    foldl (⊕) e [x0, x1, x2]
=   foldl (⊕) (e ⊕ x0) [x1, x2]
=   foldl (⊕) ((e ⊕ x0 ⊕ x1) [x2]
=   foldl (⊕) (((e ⊕ x0) ⊕ x1) ⊕ x2) []
=   ((e ⊕ x0) ⊕ x1) ⊕ x2
```

Si ⊕ es asociativo y tiene a *e* como elemento neutro, entonces, como veremos en la sección siguiente, *foldr* (⊕) *e* y *foldl* (⊕) *e* definen la misma función sobre listas finitas.

Como segundo ejemplo de uso de *foldl*, considérese la siguiente definición:

```
reverse'  :: [α] → [α]
reverse'  = foldl cons [] where cons xs x = x : xs
```


Nótese el orden de los argumentos de *cons*; tenemos que *cons* = *flip* (:), donde la función estándar *flip* se define como *flip f x y* = *f y x*. La función *reverse'* invierte una lista finita. Por ejemplo,

```
    reverse' [x0, x1, x2]
=   cons (cons (cons [] x0) x1) x2
=   cons (cons [x0] x1) x2
=   cons [x1, x0] x2
=   [x2, x1, x0]
```

Es posible demostrar por inducción que *reverse'* = *reverse*, o bien hacerlo como un caso particular de un resultado más general que se describirá en la sección siguiente. Más importante que el simple hecho de que podamos definir *reverse* de un modo diferente, es el que *reverse'* proporciona un programa mucho más eficiente: *reverse'* necesita un tiempo proporcional a *n*, siendo *n* la longitud de la lista, mientras que *reverse'* necesita un tiempo proporcional a n^2.

Hay un número importante de leyes relacionadas con *foldr* y *foldl*, pero posponemos su estudio a la sección siguiente.

4.5.1 Las funciones *fold* sobre listas no vacías

Supongamos que queremos hallar el máximo elemento de una lista finita cuyos elementos pertenecen a un tipo con orden. Nos gustaría hacer esto definiendo

```
maxlist  :: (Ord α) ⇒ [α] → α
maxlist  = foldr(max) e
```

donde (*x* **max** *y*) devuelve el máximo de *x* e *y*. En ese caso, ¿qué valor deberíamos escoger para *e*? En otras palabras, ¿que valor es *maxlist* []? La elección no es arbitraria porque querremos, naturalmente, que *maxlit* [*x*] = *x*, pero la definición proporciona en su lugar (*x* **max** *e*). Por tanto, *e* ha de ser más pequeño que cualquier otro valor del tipo α. Tal valor puede no existir. En Haskell existe una clase de tipos *Bounded* cuyas concreciones son tipos que poseen elementos máximos y mínimos:

```
class Bounded α where
    minBound, maxBound  :: α
```

Por tanto, una solución es redefinir *maxlist* como

```
maxlist  :: (Ord α, Bounded α) ⇒ [α] → α
maxlist  = foldr (max) minBound
```

Una solución alternativa, que no necesita usar *Bounded*, es introducir una nueva función de la familia *fold*, que actúe sobre listas no vacías. De hecho, hay dos funciones, *foldr*1 y *foldl*1. Sus definiciones son

```
foldr1          :: (α → α → α) → [α] → α
foldr1 f (x : xs) = if null xs then x else f x (foldr1 f xs)

foldl1          :: (α → α → α) → [α] → α
foldl1 f (x : xs) = foldl f x xs
```

Por ejemplo,

$$foldr1\ (\oplus)\ [x_0, x_1, x_2, x_3] \quad = \quad x_0 \oplus (x_1 \oplus (x_2 \oplus x_3))$$

$$foldl1\ (\oplus)\ [x_0, x_1, x_2, x_3] \quad = \quad ((x_0 \oplus x_1) \oplus x_2) \oplus x_3$$

Ahora, podemos resolver el problema definiendo

$$maxlist \quad = \quad foldr1\ (\mathbf{max})$$

Por supuesto, dado que **max** es asociativo, también podríamos haber definido *maxlist* en términos de *foldl1*.

4.5.2 La función *scanl*

A veces, es conveniente aplicar la función *foldl* a cada segmento inicial de una lista. Esto es lo que hace la función *scanl*. Por ejemplo,

$$scanl\ (\oplus)\ e\ [x_0, x_1, x_2] \quad = \quad [e, e \oplus x_0, (e \oplus x_0) \oplus x_1, ((e \oplus x_0) \oplus x_1) \oplus x_2]$$

En particular, *scanl* (+) 0 calcula la lista de las sumas acumuladas de una lista de números, y *scanl* (×) 1 [1..*n*] calcula una lista de los factoriales de los *n* primeros números naturales. En las Secciones 5.2 y 5.3 se dan ejemplos más relevantes del uso de *scanl*. Daremos dos definiciones de *scanl*; la primera es más clara, pero la segunda es más eficiente.

Para la primera definición necesitaremos la función *inits* que devuelve la lista de los segmentos iniciales de una lista. Por ejemplo,

$$inits\ [x_0, x_1, x_2] \quad = \quad [[], [x_0], [x_0, x_1], [x_0, x_1, x_2]]$$

La lista vacía tiene un único segmento: la propia lista vacía; una lista $(x : xs)$ tiene la lista vacía como su segmento inicial más pequeño, y después los otros segmentos iniciales comienzan con x y van seguidos por un segmento inicial de xs. Por tanto,

$$\begin{array}{lcl} inits & :: & [\alpha] \to [[\alpha]] \\ inits\ [] & = & [[]] \\ inits\ (x : xs) & = & [] : map\ (x :)\ (inits\ xs) \end{array}$$

Se puede definir la función *inits* de un modo más sucinto como una concreción de *foldr*:

$$inits \quad = \quad foldr\ f\ [[]]\ \mathbf{where}\ f\ x\ xss = [] : map\ (x :)\ xss$$

Ahora, definimos

$$\begin{array}{lcl} scanl & :: & (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to [\alpha] \\ scanl\ f\ e & = & map\ (foldl\ f\ e) \cdot inits \end{array}$$

Ésta es la definición más clara de *scanl* pero conduce a un programa ineficiente. La función f se aplica k veces en la evaluación de *foldl* f e sobre una lista de longitud k y, dado que los segmentos iniciales de una lista de longitud n son listas de longitudes $0, 1, \ldots, n$, la función f se aplica un total de aproximadamente $n^2/2$ veces.

Sinteticemos ahora un programa más eficiente. La síntesis se hará usando un argumento de inducción sobre xs; por tanto presentamos el cálculo en el mismo formato.

Caso ([]). En este caso obtenemos $scanl\ f\ e\ [] = [e]$. Los detalles son mera rutina y se dejan como ejercicio.

Caso $(x : xs)$. Calculamos

$$\begin{array}{ll} & scanl\ f\ e\ (x : xs) \\ = & \quad \{\text{definición de } scanl\} \\ & map\ (foldl\ f\ e)\ (inits\ (x : xs)) \\ = & \quad \{\text{definición de } inits\} \\ & map\ (foldl\ f\ e)\ ([] : map\ (x :)\ (inits\ xs)) \\ = & \quad \{\text{segunda ecuación de } map\} \\ & foldl\ f\ e\ [] : map\ (foldl\ f\ e)\ (map\ (x :)\ (inits\ xs)) \\ = & \quad \{\text{primera ecuación de } foldl\} \\ & e : map\ (foldl\ f\ e)\ (map\ (x :)\ (inits\ xs)) \\ = & \quad \{\text{dado que } map \text{ es un funtor}\} \\ & e : map\ (foldl\ f\ e.(x :))\ (inits\ xs) \\ = & \quad \{\text{conjetura: } foldl\ f\ e.(x :) = foldl\ f\ (f\ e\ x)\} \\ & e : map\ (foldl\ f\ (f\ e\ x))\ (inits\ xs) \\ = & \quad \{\text{definición de } scanl\} \\ & e : scanl\ f\ (f\ e\ x)\ xs \end{array}$$

La conjetura se deja como un breve ejercicio. Resumiendo, hemos derivado

$$\begin{array}{lcl} scanl\ f\ e\ [] & = & [e] \\ scanl\ f\ e\ (x : xs) & = & e : scanl\ f\ (f\ e\ x)\ xs \end{array}$$

Este programa es más eficiente porque la función f se aplica exactamente n veces, suponiendo que la longitud de la lista es n.

4.5.3 La función *scanr*

El cómputo dual lo proporciona la función *scanr*, definida mediante

$$scanr\ f\ a \quad = \quad map\ (foldr\ f\ a) \cdot tails$$

La función *tails* devuelve los segmentos finales de una lista. Por ejemplo,

$$tails\ [x_0, x_1, x_2] \quad = \quad [[x_0, x_1, x_2], [x_1, x_2], [x_2], []]$$

Nótese que, mientras *inits* produce una lista de segmentos iniciales en orden creciente de longitud, *tails* produce los segmentos finales en orden decreciente de longitud. Se puede definir *tails* mediante

$$\begin{array}{lcl} tails\ [] & = & [[]] \\ tails\ (x : xs) & = & (x : xs) : tails\ xs \end{array}$$

La correspondiente versión eficiente de *scanr* viene dada por

$$\begin{array}{lcl} scanr\ f\ a\ [] & = & [a] \\ scanr\ f\ a\ (x : xs) & = & f\ x\ (head\ ys) : ys \\ & & \mathbf{where}\ ys = scanr\ f\ a\ xs \end{array}$$

Dejaremos la síntesis de esta versión más eficiente como ejercicio.

4.5.4 Las funciones *scan* sobre listas no vacías

Podemos definir dos funciones más del tipo *scan*, *scanl*1 y *scanr*1. Definimos

$$\begin{aligned} scanl1\ f &= map\ (foldl\ 1\ f) \cdot inits1 \\ scanr1\ f &= map\ (foldr\ 1\ f) \cdot tails1 \end{aligned}$$

donde *inits*1 y *tails*1 devuelven, respectivamente, los segmentos no vacíos iniciales y finales de una lista no vacía. Por ejemplo,

$$inits1\ (x : xs) = map\ (x :)\ (inits\ xs)$$

Usando $foldl1\ f \cdot (x :) = foldl\ f\ x$, obtenemos

$$\begin{aligned} scanl1 &:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha] \\ scanl1\ f\ (x : xs) &= scanl\ f\ x\ xs \end{aligned}$$

Se deja como ejercicio la versión eficiente de *scanr*1.

Ejercicios

4.5.1 Definir *filter p* como una concreción de *foldr*.

4.5.2 Considérense las dos funciones *takeWhile* y *dropWhile* que se introdujeron en el Ejercicio 4.3.7. Definir *takeWhile* como una concreción de *foldr*. ¿Se puede obtener *dropWhile* como concreción de una función de la familia *fold*?

4.5.3 ¿Cuál de las siguientes ecuaciones, en caso de serlo alguna, es cierta?

$$\begin{aligned} foldl\ (-)\ x\ xs &= x - sum\ xs \\ foldr\ (-)\ x\ xs &= x - sum\ xs \end{aligned}$$

4.5.4 Considérese la siguiente definición de la función *insert*:

$$insert\ x\ xs = takeWhile\ (\leq x)\ xs +\!\!+ [x] +\!\!+ dropWhile\ (\leq x)\ xs$$

Demostrar que, si los elementos de *xs* están dispuestos en orden creciente, entonces también lo están los de *insert x xs* para todo *x*. Utilizar *insert* para definir una función *isort* que ordene una lista crecientemente. Usar para ello *foldr*.

4.5.5 La función *elimDups* elimina los duplicados adyacentes de una lista. Por ejemplo, *elimDups* [1, 2, 2, 3, 3, 3, 1, 1] = [1, 2, 3, 1]. Definir *elimDups* usando *foldl* o *foldr*.

4.5.6 Dada una lista $xs = [x_1, x_2, \ldots, x_n]$ de números, la secuencia *ssm xs* de máximos sucesivos es la mayor subsecuencia $[x_{j_1}, x_{j_2}, \ldots, x_{j_m}]$ tal que $j_1 = 1$ y $x_j < x_{j_k}$ para $j < j_k$. Por ejemplo, la secuencia de máximos sucesivos de [3, 1, 3, 4, 9, 2, 10, 7] es [3, 4, 9, 10]. Definir *ssm* en términos de *foldl*.

4.5.7 Demostrar que *scan f e* [] = [*e*].

4.5.8 Justificar el programa eficiente dado para *scanr*.

4.5.9 ¿Qué lista produce *scanl* (/) [1..*n*]?

4.5.10 La constante matemática *e* se define mediante $e = \sum_{n=0}^{\infty} 1/n!$. Escribir una expresión que evalúe *e* hasta un cierto grado de precisión requerido.

4.5.11 Considérese el tipo de datos *Liste* α definido en el Ejercicio 4.1.4. La función *fold* para este tipo de datos se define mediante

$$\begin{aligned} &folde && :: (\beta \to \alpha \to \beta) \to \beta \to Liste\ \alpha \to \beta \\ &folde\ f\ e\ Nil && = e \\ &folde\ f\ e\ (Snoc\ xs\ x) && = f\ (folde\ f\ e\ xs)\ x \end{aligned}$$

El tipo asignado a *folde* es muy parecido al asignado a *foldl*, siendo la única diferencia que $[\alpha]$ se reemplaza por *Liste* α. De hecho,

$$folde\ f\ e = foldl\ f\ e \cdot convert$$

donde $convert :: [\alpha] \to Liste\ \alpha$. Definir la función *convert* y demostrar por inducción que se cumple la ecuación anterior.

4.5.12 Derivar un programa eficiente para *scanr*1.

4.6 Leyes relativas a *fold*

Hay un número importante de leyes relacionadas con *foldr* y *foldl*. Como vimos en la Sección 3.3, en lugar de tener que demostrar una propiedad de una función definida recursivamente sobre un tipo de datos recursivo escribiendo una demostración por inducción explícita, a veces se puede expresar la propiedad como una concreción de alguna de la leyes del operador *fold* correspondiente a ese tipo de datos.

4.6.1 Teoremas de dualidad

Las primeras tres leyes se denominan *teoremas de dualidad* y establecen la relación entre *foldr* y *foldl*. El *primer teorema de dualidad* es el siguiente:

Primer teorema de dualidad. Supongamos que $\oplus$ es asociativo y tiene a *e* como elemento neutro. Entonces

$$foldr\ (\oplus)\ e\ xs = foldl\ (\oplus)\ e\ xs$$

para todas las listas finitas *xs*.

Por ejemplo, podríamos haber definido

$$\begin{aligned} sum &= foldl\ (+)\ 0 \\ and &= foldl\ (\wedge)\ True \\ concat &= foldl\ (+\!\!+)\ [] \end{aligned}$$

Sin embargo, como se explicará en el Capítulo 7, a veces es más eficiente implementar una función usando *foldl*, y otras es más eficiente usar *foldr*.

El *segundo teorema de dualidad* es una generalización del primero.

Segundo teorema de dualidad. Supongamos que $\oplus$, $\otimes$ y *e* son tales que para todo *x*, *y* y *z* se cumple

$$\begin{aligned} x \oplus (y \otimes z) &= (x \oplus y) \otimes z \\ z \oplus e &= e \otimes x \end{aligned}$$

En otras palabras, $\oplus$ y $\otimes$ asocian entre sí, y operar con e a la derecha de $\oplus$ es equivalente a operar con e a la izquierda de $\otimes$. Entonces

$$foldr\ (\oplus)\ e\ xs \quad = \quad foldl\ (\otimes)\ e\ xs$$

para todas las listas finitas xs.

Demostración. La demostración del segundo teorema de dualidad se hará por inducción sobre xs.

Caso ([]). Ambos lados se simplifican a e.

Caso $(x : xs)$. Para el lado izquierdo, razonamos

$$\begin{array}{ll} & foldr\ (\oplus)\ e\ (x : xs) \\ = & \quad \{\text{definición de } foldr\} \\ & x \oplus foldr\ (\oplus)\ e\ xs \\ = & \quad \{\text{hipótesis de inducción}\} \\ & x \oplus foldl\ (\otimes)\ e\ xs \end{array}$$

Para el lado derecho, razonamos

$$\begin{array}{ll} & foldl\ (\otimes)\ e\ (x : xs) \\ = & \quad \{\text{definición de } foldl\} \\ & foldl\ (\otimes)\ (e \otimes x)\ xs \\ = & \quad \{\text{hipótesis}\} \\ & foldl\ (\otimes)\ (x \oplus e)\ xs \end{array}$$

Los dos lados se han reducido a dos expresiones diferentes. Para demostrar que son iguales, necesitamos una segunda demostración por inducción; sin embargo, si se toma como segunda hipótesis de inducción la siguiente afirmación obvia

$$x \oplus foldl\ (\otimes)\ e\ xs \quad = \quad foldl\ (\otimes)\ (x \oplus e)\ xs$$

para todo x y para toda lista finita xs, entonces la inducción no funciona. El único modo de apreciar este hecho es intentarlo, una tarea que dejamos como ejercicio. Lo que necesitamos es una hipótesis *más fuerte*, a saber:

$$x \oplus foldl\ (\otimes)\ y\ xs \quad = \quad foldl\ (\otimes)\ (x \oplus y)\ xs$$

para todo x, y y para toda lista finita xs. Como vimos en la Sección 4.2 al estudiar *reverse*, a menudo una hipótesis de inducción más fuerte puede simplificar una demostración.

Caso ([]). Ambos lados se simplifican a $x \oplus a$.

Caso $(z : xs)$. (Se ha utilizado el uso de z para evitar confusiones con la variable x que aparece en la afirmación). Para el lado izquierdo, razonamos

$$\begin{array}{ll} & x \oplus foldl\ (\otimes)\ y\ (z : xs) \\ = & \quad \{\text{definición}\} \\ & x \oplus foldl\ (\otimes)\ (y \otimes z)\ xs \\ = & \quad \{\text{hipótesis de inducción}\} \\ & foldl\ (\otimes)\ (x \oplus (y \otimes z))\ xs \\ = & \quad \{\text{hipótesis}\} \\ & foldl\ (\otimes)\ ((x \oplus y) \otimes z)\ xs \end{array}$$

Para el lado derecho, razonamos

$$\begin{array}{ll} & foldl\ (\otimes)\ (x \oplus y)\ (z : xs) \\ = & \quad \{\text{definición}\} \\ & foldl\ (\otimes)\ ((x \oplus y) \otimes z)\ xs \end{array}$$

□

El segundo teorema de dualidad contiene al primer teorema de dualidad como caso particular, en concreto cuando $(\oplus) = (\otimes)$.

Para ilustrar el segundo teorema de dualidad, considérense las siguientes definiciones de *length* y *reverse*:

$$\begin{array}{lll} length & = & foldr\ increm\ 0, \quad \text{donde } increm\ x\ n = 1 + n \\ length & = & foldl\ increm'\ 0, \quad \text{donde } increm'\ n\ x = n + 1 \\ \\ reverse & = & foldr\ snoc\ [], \quad \text{donde } snoc\ x\ xs = xs +\!\!+ [x] \\ reverse & = & foldl\ cons\ [], \quad \text{donde } cons\ xs\ x = x : xs \end{array}$$

Las funciones *increm*, *increm'*, y 0 satisfacen las condiciones del segundo teorema de dualidad, al igual que lo hacen *snoc*, *cons* y []. Dejamos la comprobación como ejercicio. Por tanto, las dos definiciones de *length* y de *reverse* son equivalentes para todas las listas finitas. Las diferencias prácticas entre las dos definiciones de *length* no son obvias, pero el segundo programa para *reverse* es el más eficiente de los dos.

Se deja como ejercicio la demostración del *tercer teorema de dualidad*.

El tercer teorema de dualidad. Para todas las listas finitas xs,

$$foldr\ f\ e\ xs \quad = \quad foldl\ (flip\ f)\ e\ (reverse\ xs)$$

donde $flip\ f\ x\ y = f\ y\ x$.

Para ilustrar el tercer teorema de dualidad, considérese

$$foldr\ (:)\ []\ xs \quad = \quad foldl\ (flip\ (:))\ []\ (reverse\ xs)$$

Dado que $foldr\ (:)\ [] = id$ y $foldl\ (flip\ (:))\ [] = reverse$, obtenemos

$$xs \quad = \quad reverse\ (reverse\ xs)$$

para todas las listas finitas xs, un resultado que ya hemos demostrado directamente.

4.6.2 Fusión

Los teoremas de dualidad tratan acerca de la relación entre los dos tipos de función *fold* sobre listas; los teoremas de fusión tratan con un solo operador *fold* a la vez. Los teoremas de fusión para *foldr* y *foldl* son los siguientes.

Teorema de fusión para *foldr*. Si f es estricta, con $f\ a = b$, y para todo x e y se tiene $f\ (g\ x\ y) = h\ x\ (f\ y)$, entonces

$$f \cdot foldr\ g\ a \quad = \quad foldr\ h\ b$$

Teorema de fusión para *foldl*. Si f es estricta, con $f\ a = b$, y para todo x e y se tiene $f\ (g\ x\ y) = h\ (f\ x)\ y$, entonces

$$f \cdot foldl\ g\ a \quad = \quad foldl\ h\ b$$

La demostración del teorema de fusión para *foldr* es muy similar a la dada en el capítulo precedente para *foldn* y se deja como ejercicio. La demostración del teorema de fusión para *foldl* es compleja y se estudia en el Ejercicio 4.6.3.

La ley de fusión es muy general y existen varios casos particulares útiles. En la siguiente sección se consideran cuatro de ellos.

4.6.3 Fusión *fold-map*

Recordemos que $map\ g = foldr\ (cons \cdot g)\ []$, donde $cons\ x\ xs = x : xs$. Veamos qué condiciones aseguran que

$$foldr\ f\ a \cdot map\ g \quad = \quad foldr\ h\ b$$

Observando las condiciones del teorema de fusión para *foldr*, tenemos que $foldr\ f\ a$ es una función estricta y que la segunda condición se satisface tomando $a = b$. La tercera condición se expresa como

$$\begin{array}{ll} & foldr\ f\ a\ ((cons \cdot g)\ x\ xs) = h\ x\ (foldr\ f\ a\ xs) \\ = & \quad \{\text{definición de } cons\} \\ & foldr\ f\ a\ (g\ x : xs) = h\ x\ (foldr\ f\ a\ xs) \\ = & \quad \{\text{definición de } foldr\} \\ & f\ (g\ x)\ (foldr\ f\ a\ xs) = h\ x\ (foldr\ f\ a\ xs) \\ \Leftarrow & \quad \{\text{generalización}\} \\ & f\ (g\ x)\ y = h\ x\ y \end{array}$$

El símbolo $\Leftarrow$ se lee "se deduce de". Dado que $h\ x\ y = f\ (g\ x)\ y$ si y sólo si $h\ x = f\ (g\ x)$ si y sólo si $h = f \cdot g$, obtenemos el siguiente resultado

$$foldr\ f\ a \cdot map\ g \quad = \quad foldr\ (f \cdot g)\ a$$

Dicho con palabras, una operación *foldr* después de una operación *map* siempre se puede expresar como una sola operación *foldr*. Llamaremos a esta ley la *ley de fusión fold–map*.

4.6.4 Fusión *fold-concat*

Como segundo ejemplo, recordemos que $concat = foldr\ (+\!+)\ []$. Consideremos qué condiciones aseguran que

$$foldr\ f\ a \cdot concat \quad = \quad foldr\ h\ b$$

De nuevo, observando las condiciones necesarias para la fusión, tenemos que $foldr\ f\ a$ es estricta y que la segunda condición se satisface tomando $a = b$. La tercera condición se reescribe como

$$\begin{array}{ll} & foldr\ f\ a\ (xs +\!+ ys) = h\ xs\ (foldr\ f\ a\ ys) \\ \equiv & \quad \{\text{conjetura: } foldr\ f\ a\ (xs +\!+ ys) = foldr\ f\ (foldr\ f\ a\ ys)\ xs\} \\ & foldr\ f\ (foldr\ f\ a\ ys)\ xs = h\ xs\ (foldr\ f\ a\ ys) \end{array}$$

Por tanto, podemos tomar $h\ xs\ y = foldr\ f\ y\ xs$ o, más brevemente, $h = flip\ (foldr\ f)$. La demostración de la conjetura se deja como ejercicio. Por tanto, hemos demostrado que

$$foldr\ f\ a \cdot concat \quad = \quad foldr\ (flip\ (foldr\ f))\ a$$

Llamaremos a esta propiedad la *ley de fusión fold–concat*.

4.6.5 Ley del contable

Como tercer ejemplo, consideremos qué condiciones sobre f y a aseguran que

$$foldr\ f\ a \cdot concat \quad = \quad foldr\ f\ a \cdot map\ (foldr\ f\ a)$$

Llamaremos a esta ecuación la *ley del contable*. El caso particular $sum \cdot concat = sum \cdot map\ sum$ recuerda al contable que suma los gastos de cada día individual y obtiene la suma del año sumando dichas sumas.

Podemos usar las leyes *fold-concat* y *fold-map* para reescribir la ley del contable de la forma siguiente

$$foldr\ (flip\ (foldr\ f))\ a \quad = \quad foldr\ (f \cdot foldr\ f\ a)\ a$$

Esta ecuación se satisface si $flip\ (foldr\ f) = f \cdot foldr\ f\ a$. Aplicando ambos lados a xs y después a y, obtenemos

$$foldr\ f\ y\ xs \quad = \quad f\ (foldr\ f\ a\ xs)\ y$$

La ecuación se cumple si f es asociativo y tiene a a por elemento neutro. Dejamos la demostración como ejercicio.

Resumiendo, la ley del contable se cumple siempre que los argumentos de *foldr* sean un operador asociativo y su elemento neutro. Por ejemplo, dado que $sum = foldr\ (+)\ 0$, queda justificada la técnica de la suma del contable. Otro ejemplo es

$$concat \cdot concat \quad = \quad concat \cdot map\ concat$$

Esta ecuación dice que aplanar una lista de listas en una sola lista se puede hacer aplanando de fuera hacia dentro o de dentro hacia fuera.

4.6.6 Fusión *fold-scan*

Considérese la ecuación

$$1 + x_0 + x_0 \times x_1 + x_0 \times x_1 \times x_2 \quad = \quad 1 + x_0 \times (1 + x_1 \times (1 + x_2))$$

Esta ecuación puede generalizarse a una lista arbitraria de números y se puede expresar de la forma

$$foldr1\ (+) \cdot scanl\ (\times)\ 1 \quad = \quad foldr\ (\odot)\ 1$$

donde $x \odot y = 1 + x \times y$. Ello conduce a una pregunta más general: ¿cuándo se cumple la ley

$$foldr1\ (\oplus) \cdot scanl\ (\otimes)\ e \quad = \quad foldr\ (\odot)\ e$$

siendo $x \odot y = e \oplus (x \otimes y)$?

Bajo la hipótesis de que $\otimes$ es asociativo y que tiene a e como elemento neutro, podemos expresar *scanl* como una concreción de *foldr*:

$$scanl\ (\otimes)\ e \quad = \quad foldr\ g\ [e] \qquad \text{donde} \qquad g\ x\ xs = e : map\ (x \otimes)\ xs$$

Se deja la demostración como ejercicio. Una vez definido *scanl* como concreción de *foldr*, se puede ver que la ley que hace falta es otra concreción más de la ley de fusión *foldr*. Las primeras dos condiciones son inmediatas y la tercera se expresa como

$$foldr1\ (\oplus)\ (g\ x\ xs) = x \odot (foldr\ 1\ (\oplus)\ xs)$$

$$
\begin{array}{ll}
= & \{\text{definición de } g\} \\
 & \mathit{foldr1}\ (\oplus)\ (e : \mathit{map}\ (x \otimes)\ xs) = x \odot (\mathit{foldr1}\ (\oplus)\ xs) \\
\equiv & \{\text{definición de } \mathit{foldr1} \text{ y } \odot\} \\
 & e \oplus \mathit{foldr1}\ (\oplus)\ (\mathit{map}\ (x \otimes)\ xs) = e \oplus (x \otimes \mathit{foldr1}\ (\oplus)\ xs) \\
\Leftarrow & \{\text{sustitución}\} \\
 & \mathit{foldr1}\ (\oplus) \cdot \mathit{map}\ (x \otimes) = (x \otimes) \cdot \mathit{foldr1}\ (\oplus)
\end{array}
$$

La última ecuación se satisface si $\otimes$ es distributivo con respecto a $\oplus$, es decir, si

$$x \otimes (y \oplus z) \quad = \quad (x \otimes y) \oplus (x \otimes z)$$

para todo x, y y z. Se deja la demostración como ejercicio.

Resumiendo, si $\otimes$ es asociativo, e es su elemento neutro y $\otimes$ es distributivo con respecto a $\oplus$, entonces $\mathit{foldr1}\ (\oplus) \cdot \mathit{scanl}\ (\otimes)\ e = \mathit{foldr}\ (\odot)\ e$, siendo $x \odot y = e \oplus (x \otimes y)$. Llamaremos a esta propiedad la *ley de fusión fold–scan.*

En el resto de esta sección daremos un ejemplo de cómo se pueden usar estas leyes para la síntesis de programas eficientes.

4.6.7 Ejemplo: la mayor suma de segmentos

El problema de la mayor suma de segmentos es famoso y su historia se describe en *Programming Pearls* (1987) de J. Bentley. Dada una secuencia de números, se pide calcular la mayor de las sumas de todos los segmentos de la secuencia. Por ejemplo, la secuencia

$$[-1, 2, -3, 5, -2, 1, 3, -2, -2, -3, 6]$$

tiene como mayor suma 7, que es la suma del segmento $[5, -2, 1, 3]$. Por otra parte, la secuencia $[-1, -2, -3]$ tiene el valor 0 como mayor suma de segmentos, dado que la secuencia vacía es un segmento de toda lista y su suma es cero. Se deduce que la mayor suma de segmentos es siempre mayor o igual que cero. (Los ejercicios tratan una variante del problema en la cual se excluye la secuencia vacía, de forma que la mayor suma de segmentos puede ser negativa.)

Se puede especificar formalmente el problema mediante

$$
\begin{array}{lll}
\mathit{mss} & :: & [\mathit{Int}] \to \mathit{Int} \\
\mathit{mss} & = & \mathit{maxlist} \cdot \mathit{map\ sum} \cdot \mathit{segs}
\end{array}
$$

donde *segs* devuelve la lista de todos los segmentos de una lista. Esta función se puede definir de diferentes maneras, entre ellas

$$\mathit{segs} \quad = \quad \mathit{concat} \cdot \mathit{map\ inits} \cdot \mathit{tails}$$

Esta definición corresponde al proceso de tomar todos los segmentos iniciales de todos los segmentos finales. Por ejemplo,

$$\mathit{segs}\ \text{"abc"} \quad = \quad [\text{""}, \text{"a"}, \text{"ab"}, \text{"abc"}, \text{""}, \text{"b"}, \text{"bc"}, \text{""}, \text{"c"}, \text{""}]$$

La secuencia vacía aparece cuatro veces en esta lista, una vez por cada segmento final de "abc".

La evaluación directa de *mss* necesita un número de pasos proporcional a n^3 para una lista de longitud n. Hay aproximadamente n^2 segmentos, y sumar todos ellos necesita aproximadamente n^3 pasos. No es obvio que se pueda conseguir un tiempo menor que cúbico para este problema. Sin embargo, veremos que se puede lograr con algunos cálculos. Sustituyendo la definición de *segs* en *mss*, hemos de calcular

$$\mathit{maxlist} \cdot \mathit{map\ sum} \cdot \mathit{concat} \cdot \mathit{map\ inits} \cdot \mathit{tails}$$

Lo único que podemos hacer con esta expresión (exceptuando deshacer la sustitución) es usar la ley $\mathit{map}\ f \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ f)$ para sustituir el segundo y el tercer términos, obteniendo

$$\mathit{maxlist} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map\ sum}) \cdot \mathit{map\ inits} \cdot \mathit{tails}$$

Ahora podemos hacer dos cosas: o bien aplicamos la ley del contable a los dos primeros términos, o bien aplicamos la ley $\mathit{map}\ (f \cdot g) = \mathit{map}\ f \cdot \mathit{map}\ g$ al tercer y cuarto término. Por supuesto siempre podemos hacer ambos pasos a la vez, en cuyo caso el resultado será

$$\mathit{maxlist} \cdot \mathit{map\ maxlist} \cdot \mathit{map}\ (\mathit{map\ sum} \cdot \mathit{inits}) \cdot \mathit{tails}$$

Para el paso siguiente podemos aplicar la ley *map* una vez más y obtenemos

$$\mathit{maxlist} \cdot \mathit{map}\ (\mathit{maxlist} \cdot \mathit{map\ sum} \cdot \mathit{inits}) \cdot \mathit{tails}$$

Se puede reemplazar el término $\mathit{map\ sum} \cdot \mathit{inits}$ por *scanl*, una sustitución que se podía haber hecho antes:

$$\mathit{maxlist} \cdot \mathit{map}\ (\mathit{maxlist} \cdot \mathit{scanl}\ (+)\ 0) \cdot \mathit{tails}$$

Dado que $\mathit{maxlist} = \mathit{foldr1}\ (\mathbf{max})$ y $(+)$ es distributivo con respecto a $(\mathbf{max})$, la ley de fusión *fold-scan* nos da

$$\mathit{maxlist} \cdot \mathit{map}\ (\mathit{foldr}\ (\odot)\ 0) \cdot \mathit{tails}$$

donde $x \odot y = 0 \ \ \mathbf{max}\ (x + y)$. Finalmente, se pueden reemplazar los dos últimos términos por *scanr*, obteniendo

$$\mathit{mss} = \mathit{maxlist} \cdot \mathit{scanr}\ (\odot)\ 0$$

El resultado es un programa que calcula en tiempo lineal la mayor suma de segmentos.

Ejercicios

4.6.1 Demostrar el tercer teorema de dualidad.

4.6.2 Demostrar el teorema de fusión para *foldr*.

4.6.3 El teorema de fusión para *foldl* establece que $f \cdot \mathit{foldl}\ g\ a = \mathit{foldl}\ h\ b$ siempre que f sea estricta, $f\ a = b$, y $f\ (g\ x\ y) = h\ (f\ x)\ y$ para todo x e y.

En la demostración por inducción, el caso $(x : xs)$ requiere demostrar que

$$f\ (\mathit{foldl}\ g\ (g\ a\ x)\ xs) = \mathit{foldl}\ h\ (h\ (f\ a)\ x)\ xs$$

Para hacerlo, necesitamos una segunda hipótesis de inducción, a saber:

$$f\ (\mathit{foldl}\ g\ (g\ x\ y)\ zs) = \mathit{foldl}\ h\ (h\ (f\ x)\ y)\ zs$$

para todo x, y y para toda lista zs. Demostrar esta ecuación y completar la demostración de la ley de fusión.

4.6.4 Demostrar las ecuaciones

$$\begin{aligned} \mathit{foldr}\ f\ a\ (xs +\!\!+ ys) &= \mathit{foldr}\ f\ (\mathit{foldr}\ f\ a\ ys)\ xs \\ \mathit{foldl}\ f\ a\ (xs +\!\!+ ys) &= \mathit{foldl}\ f\ (\mathit{foldl}\ f\ a\ xs)\ ys \end{aligned}$$

¿Qué restricciones, caso de haber alguna, son necesarias para xs e ys?

4.6.5 Demostrar que se puede escribir $\mathit{scanl}\ (\otimes)\ e$ como una concreción de foldr si $\otimes$ es asociativo y tiene a e como elemento neutro. ¿Se puede escribir $\mathit{scanl}\ (\otimes)\ e$ como una concreción de foldl sin ninguna suposición sobre $\otimes$ y e?

4.6.6 Demostrar que si $\otimes$ es distributivo con respecto a $\oplus$, entonces

$$\mathit{foldr1}\ (\oplus) \cdot \mathit{map}\ (x \otimes) \quad = \quad (x \otimes) \cdot \mathit{foldr1}\ (\oplus)$$

4.6.7 Es tentador tratar de proseguir un paso más allá el cálculo de la mayor suma de segmentos y usar la ley de fusión *fold-scan* para reemplazar $\mathit{maxlist} \cdot \mathit{scanr}\ (\odot)\ 0$ por un foldl. ¿Por qué no funciona esta idea? (Ver el Ejercicio 4.6.10 para una versión de la fusión *fold-scan* que sí funciona).

4.6.8 Dar una ley de fusión para $\mathit{foldr1}$ y usarla para imponer condiciones a f, g y h de modo que se cumpla la ley

$$\mathit{foldr1}\ f \cdot \mathit{scanl1}\ g \quad = \quad \mathit{foldr1}\ h$$

4.6.9 Definir $\mathit{segs1}$ mediante $\mathit{segs1} = \mathit{concat} \cdot \mathit{map}\ \mathit{inits1} \cdot \mathit{tails1}$, donde $\mathit{inits1}$ devuelve los segmentos iniciales no vacíos de una lista y $\mathit{tails1}$ los segmentos finales no vacíos. Usar esta versión para especificar y resolver una versión del problema de la mayor suma de segmentos que no admita segmentos vacíos.

4.6.10 Existe una versión de la ley de fusión *fold-scan* que no depende de propiedades especiales de los operadores involucrados. Esta ley es

$$\mathit{foldl1}\ (\oplus) \cdot \mathit{scanl}\ (\otimes)\ e \quad = \quad \mathit{fst} \cdot \mathit{foldl}\ (\odot)\ (a, a)$$

siendo $(x, y) \odot z = (x \oplus t, t)$, donde $t = y \otimes z$. Demostrar esta versión de la ley de fusión *fold-scan*.

4.7 Notas del capítulo

Haskell proporciona muchas más funciones estándar sobre listas de las que hemos descrito en este capítulo. Para más detalles, consultar el preludio estándar de Haskell y la librería de funciones sobre listas.

Se ha hecho mucho trabajo sobre generación automática o asistida de demostraciones por inducción; ver, por ejemplo, Boyer y Moore (1979), Gordon, Milner y Wadsworth (1979), Paulson (1983) y Martin y Nipkow (1990).

La relación entre funciones polimórficas y condiciones de naturalidad se explora en Wadler (1989); ver también Bird y de Moor (1997).

Las leyes de fusión para los operadores de la familia *fold* fueron sistematizadas en Malcolm (1990), Fokkinga (1992), Jeuring (1993) y Meijer (1992).

El problema de la mayor suma de segmentos se describe en Bentley (1987). La derivación presentada en este capítulo apareció por primera vez en Bird (1989).

Capítulo 5

Ejemplos

Los ejemplos de tratamiento de listas presentados en este capítulo provienen de diversas fuentes y cubren tanto aplicaciones numéricas como simbólicas. En particular, diseñaremos un módulo simple para realizar aritmética con enteros de rango arbitrario, diseñaremos algunas funciones útiles para el tratamiento de textos y mostraremos cómo imprimir un calendario. Empezamos con un problema simple en el cual la indexación de listas desempeña un papel central.

5.1 Convirtiendo números en palabras

A veces necesitamos expresar los números con letras. Por ejemplo, para rellenar un cheque o una transferencia bancaria correctamente, no sólo la cantidad debe aparecer en cifras sino que también ha de ser escrita en letra. Supongamos, por simplicidad, que el número dado es un entero mayor que cero pero menor que un millón. Queremos diseñar una función *convertir* de tal modo que, siempre que *n* esté en el rango establecido, el valor del *convertir n* sea la lista de caracteres que corresponde a la formulación usual, en español[1] del número *n*.

Esta especificación informal supone que sabemos exactamente cuál es la "formulación usual en español" de un número. De hecho, en diferentes lugares se adoptan diferentes reglas y convenios. Las reglas que seguiremos se ilustran en los siguientes ejemplos:

? *convertir* 308000
"trescientos ocho mil"
? *convertir* 369027
"trescientos sesenta y nueve mil veintisiete"
? *convertir* 369401
"trescientos sesenta y nueve mil cuatrocientos uno"

[1] N. del T.: En el original inglés, el algoritmo de esta sección convertía un número a un texto en inglés. Se ha reescrito para que el texto producido esté en español, utilizando las reglas y excepciones de nuestra lengua en lugar de las correspondientes inglesas. Se han modificado también algunas explicaciones que acompañaban a los algoritmos.

Nótese la diferencia entre las expresiones "veintisiete" y "sesenta y nueve". En el primer caso se emplea una sola palabra, mientras que en el segundo se emplean dos palabras separadas por la conjunción "y". Nótese también la necesidad de la palabra "mil" para indicar millares y la ausencia de una palabra equivalente "ciento" para indicar centenas. Es decir, las reglas de formación para las centenas son diferentes que para los millares.

Una buena manera de tratar problemas como éste es considerar primero un problema más simple. Por supuesto, no hay garantía de que las soluciones obtenidas para problemas más simples se puedan usar directamente en el problema que las inspiró; simplemente pueden servir para familiarizar al diseñador con algunas de las características y dificultades implicadas. Aun así, el trabajo no se desperdicia; la familiaridad con un problema es una de nuestras herramientas más importantes para resolverlo y, a menudo, podremos usar la solución directamente o adaptarla.

Un aspecto obvio por el que empezar es suponer que el número n pertenece a un intervalo más pequeño, digamos $0 < n < 100$. En este caso, n tiene uno o dos dígitos. Estos dígitos van a ser necesarios para la conversión; por tanto, comenzamos por la siguiente definición:

```
convertir2   :: Int → String
convertir2   =  combinar2 · digitos2

digitos2     :: Int → (Int, Int)
digitos2 n   =  (n div 10, n mod 10)
```

Para definir *combinar2*, necesitaremos nombres españoles para los números más simples. Éstos pueden venir dados como listas de cadenas:

```
unidades, dieciAlgo, veintiAlgo, decenas :: [String]
unidades    = ["uno", "dos", "tres", "cuatro", "cinco",
               "seis", "siete", "ocho", "nueve"]
dieciAlgo   = ["diez", "once", "doce", "trece", "catorce",
               "quince", "dieciseis", "diecisiete", "dieciocho", "diecinueve"]
veintiAlgo  = ["veinte", "veintiuno", "veintidos", "veintitres", "veinticuatro",
               "veinticinco", "veintiseis", "veintisiete", "veintiocho", "veintinueve"]
decenas     = ["veinte", "treinta", "cuarenta", "cincuenta",
               "sesenta", "setenta", "ochenta", "noventa"]
```

La definición de *combinar2* emplea estas listas para extraer los elementos apropiados dependiendo de los dígitos:

```
combinar2                 :: (Int, Int) → String
combinar2 (0, u + 1)      =  unidades !! u
combinar2 (1, u)          =  dieciAlgo !! u
combinar2 (2, u)          =  veintiAlgo !! u
combinar2 (t + 3, 0)      =  decenas !! t
combinar2 (t + 3, u + 1)  =  decenas !! t ++ "␣y␣" ++ unidades !! u
```

Recordemos que la indexación de listas comienza por 0, no por 1. Los patrones de la izquierda son mutuamente excluyentes, por lo que no importa el orden de las ecuaciones. Sin embargo, no se ha especificado ningún valor para el patrón (0, 0).

El caso $0 < n < 100$ ha resultado bastante fácil; por tanto, ahora investigaremos el rango $0 < n < 1000$ donde n puede tener hasta tres dígitos. Nos gustaría hacer uso de *convertir2*, así como explotar la estructura simple de su definición; por ello definimos:

```
convertir3   :: Int → String
convertir3   =  combinar3 · digitos3

digitos3     :: Int → (Int, Int)
digitos3 n   =  (n div 100, n mod 100)

cientos      =  ["ciento", "doscientos", "trescientos", "cuatrocientos",
                 "quinientos", "seiscientos", "setecientos", "ochocientos",
                 "novecientos"]

combinar3                 :: (Int, Int) → String
combinar3 (0, t + 1)      =  convertir2 (t + 1)
combinar3 (h + 1, 0)      =  cientos !! h
combinar3 (h + 1, t + 1)  =  cientos !! h ++"␣" ++convertir2 (t + 1)
```

Este paso es el más importante en el diseño del algoritmo completo. Descomponemos n en dígitos en dos etapas: primero entre la parte de las centenas h y una parte t menor que cien; y después, en la definición de *convertir2*, descomponemos t en la parte de las decenas y en otra parte menor que diez.

Ahora ya podemos pasar a la siguiente y última etapa en la que n está en el rango $0 < n < 1000000$ y, por tanto, puede tener hasta 6 dígitos. De forma similar al procedimiento anterior, descomponemos n en dos números m y h, donde m es la parte de los millares y h es un número menor que mil. Por tanto, tenemos

```
convertir6   :: Int → String
convertir6   =  combinar6 · digitos6

digitos6     :: Int → (Int, Int)
digitos6 n   =  (n div 1000, n mod 1000)

combinar6                 :: (Int, Int) → String
combinar6 (0, h + 1)      =  convertir3 (h + 1)
combinar6 (m + 1, 0)      =  convertir3 (m + 1) ++ "␣mil"
combinar6 (m + 1, h + 1)  =  convertir3 (m + 1) ++ "␣mil" ++ convertir3 (h + 1)
```

La función *convertir* requerida es, precisamente, la función *convertir6*; por tanto, hemos terminado. Este ejemplo, además de ilustrar ampliamente el uso de ++ y !!, también muestra las ventajas del ajuste de patrones con respecto a una definición que use expresiones condicionales. Cada caso se puede expresar de forma clara y concisa, y es más fácil comprobar que se han cubierto todos los casos.

Ejercicios

5.1.1 Modificar la solución de modo que se imprima un punto después del número.

5.1.2 Generalizar la solución para tratar números positivos hasta mil millones.

5.1.3 Mostrar cómo se puede adaptar la solución para manejar números negativos y el cero.

5.1.4 Sabiendo que un duro equivale a cinco pesetas, escribir un programa similar para convertir un número exacto de pesetas a letras. Por ejemplo, el número 3649 debería escribirse "setecientos veintinueve duros y cuatro pesetas".

5.1.5 Como ejercicio mucho más difícil, escribir un programa que haga lo contrario que *convertir*. Es decir, la entrada es una formulación en español de un número y la salida es su correspondiente representación numérica.

5.2 Produciendo una lista de notas

Considérese el problema de procesar las notas de un examen para producir una lista final de notas. Hay dos entradas al problema y ambas son listas no vacías. Una de ellas, conocida sólo por el examinador, es la lista en orden alfabético de los nombres de los candidatos, junto con los números de identificación de los mismos. Por ejemplo,

```
GOMEZ        101372
NUÑEZ        101369
PEREZ        101370
RODRIGUEZ    101371
VIDAL        101373
```

La segunda lista, producida por el corrector del examen, es una lista ordenada de números de identificación de canditatos junto con las notas; por ejemplo,

```
101369   62
101370   75
101371   62
101372   30
101373   50
```

La tarea del examinador consiste en utilizar estas dos listas para producir una lista alfabética de candidatos con sus notas y su número de orden final. También se requiere que el resultado quede impreso de una forma atractiva. Por ejemplo,

```
Nombre       Nota    Orden
GOMEZ         30       5
NUÑEZ         62       2
PEREZ         75       1
RODRIGUEZ     62       2
VIDAL         50       4
```

Antes de comenzar a pensar en el problema, introduciremos algunos tipos renombrados apropiados:


```
type Nombre  = String
type Iden    = Integer
type Nota    = Int
type Orden   = Int
```

Por brevedad, introduciremos también

```
type Codigos   = [(Nombre, Iden)]
type Notas     = [(Iden, Nota)]
type Resultado = [(Nombre, Nota, Orden)]
```

Las dos funciones que debemos implementar son

```
listaDeClase :: (Codigos, Notas) → Resultado
mostrar      :: Resultado → String
```

Definimos *listaDeClase* como una función no currificada, lo que significa que podemos explotar un estilo composicional.

Concentrémonos primero en *listaDeClase*. Parece razonable construir esta función en dos fases: primero emparejaremos las notas con los nombres de los candidatos y después produciremos la lista de los números de orden. Por tanto, escribimos

```
listaDeClase = orden · emparejar
```

Los tipos de las funciones auxiliares son

```
emparejar :: (Codigos, Notas) → [(Nombre, Nota)]
orden     :: [(Nombre, Nota)] → Resultado
```

Trataremos cada una de estas funciones por separado.

5.2.1 Emparejamientos

En la construcción de *emparejar* podemos suponer que la lista de los códigos está ordenada alfabéticamente según el *Nombre* y la lista de las notas está en orden numérico de *Iden*. Más aún, no existen dos candidatos con el mismo nombre o el mismo número de identificación, y las dos listas tienen la misma longitud. El resultado ha de ser una lista en orden alfabético de *Nombre*.

Hay una manera obvia de construir *emparejar*: ordenar primero la lista de códigos por números de identificación, aplicar la función *zip* con la segunda lista, eliminar los números de identificación y ordenar el resultado por nombre de candidato. He aquí la definición, que es una transcripción fiel de lo que acabamos de decir:

```
emparejar = ordenar nombre · eliminar · zipp · cross (ordenar iden, id)
            where nombre (xn, xm) = xn
                  iden (xn, xi)   = xi
```

Recordemos, de la Sección 4.4, que *zipp* es la versión no currificada de *zip*. La función *eliminar* se define mediante

```
eliminar :: [((α, β), (β, γ))] → [(α, γ)]
eliminar = map (cross (fst, snd))
```

La función *ordenar* tiene el tipo siguiente:

$$ordenar \;::\; Ord\,\beta \Rightarrow (\alpha \to \beta) \to [\alpha] \to [\alpha]$$

La especificación de *ordenar f xs* es que devuelve una permutación *ys* de *xs* que satisface

$$and\,[f\,x \le f\,y \mid (x, y) \leftarrow zip\,ys\,(tail\,ys)]$$

En particular, la función *ordenar id* ordena una lista en orden creciente. Daremos una definición de *ordenar* al final de la sección. Por ahora sólo mencionaremos que, suponiendo que *f* necesita tiempo constante, se puede implementar *ordenar f* en un tiempo proporcional a $n \log n$ pasos para una lista de longitud *n*.

Se puede escribir el programa para *emparejar* de una forma alternativa. Recordemos la condición de naturalidad

$$map\,(cross\,(f, g)) \cdot zipp = zipp \cdot cross\,(map\,f, map\,g)$$

que se trató en la Sección 4.4. Utilizando esta ley, obtenemos

$$eliminar \cdot zipp = zipp \cdot cross\,(map\,fst, map\,snd)$$

Ahora recordemos la ley

$$cross\,(f, g) \cdot cross\,(h, k) = cross\,(f \cdot h, g \cdot k)$$

demostrada en la Sección 2.4. Utilizando esta ley, obtenemos

$$\begin{aligned} & cross\,(map\,fst, map\,snd) \cdot cross\,(ordenar\,iden, id) \\ = \; & cross\,(map\,fst \cdot ordenar\,iden, map\,snd) \end{aligned}$$

Juntando estos dos resultados, obtenemos

$$emparejar = ordenar\,nombre \cdot zipp \cdot cross\,(map\,fst \cdot ordenar\,iden, map\,snd)$$

5.2.2 Calculando el número de orden

Volvamos ahora al segundo problema, el de calcular *orden*. ¿Qué quiere decir exactamente que un candidato *A* tiene número de orden *n*? Podríamos responder que eso quiere decir que *A* ocupa la posición *n* (contando a partir de 1) en la lista de candidatos por orden decreciente de notas, pero ello no es exacto. Puede haber dos candidatos con las mismas notas y por tanto con el mismo número de orden, pero no pueden ocupar la misma posición en la lista. La respuesta correcta es que *A* tiene número de orden *n* si y sólo si existen exactamente $(n - 1)$ candidatos con notas más altas. Por tanto, un modo de definir *orden* es contar, para cada candidato, el número de candidatos con mejores notas:

$$\begin{aligned} & orden && :: && [(Nombre, Nota)] \to Resultado \\ & orden\,xs && = && map\,(contar\,xs)\,xs \\ & contar\,xs\,(xn, xm) && = && (xn, xm, 1 + length\,(filter\,(mejorQue\,xm)\,xs)) \\ & mejorQue\,xm\,(yn, ym) && = && xm < ym \end{aligned}$$

Siempre que la lista de alumnos sea razonablemente pequeña, los examinadores tienden, de hecho, a usar exactamente este método para calcular *orden*. Es fácil ver que el cálculo de *contar* necesita alrededor de *n* pasos para una lista de tamaño *n*; por tanto, para calcular los números de orden de todos los candidatos se necesitan, aproximadamente, n^2 pasos. Este coste es aceptable para un *n* pequeño, pero las clases tienden a ser grandes en estos días; en cualquier caso, la mayoría de los examinadores pueden usar computadores para ayudarse, de modo que no tendrían por qué restringirse sólo a un método que es fácil de llevar a cabo a mano.

El problema de calcular *orden* sería mucho más fácil si las notas estuvieran en orden decreciente. Entonces podríamos asignar números de orden con un solo recorrido de la lista. Ello sugiere redefinir *orden* mediante

$$\begin{aligned} orden = \; & ordenar\,nombre \cdot asignar \cdot reverse \cdot ordenar\,nota \\ & \cdot \; \textbf{where}\; nota\,(xn, xm) = xm \\ & \qquad\quad nombre\,(xn, xm, xr) = xn \end{aligned}$$

Para definir *asignar* podemos asociar un número de orden inicial, *n*, al enésimo candidato (contando a partir de 1), y después hacerlo decrecer si el candidato tiene la misma nota que el candidato inmediatamente anterior. Dicho de forma más precisa, se le asigna el rango del candidato inmediatamente anterior. El método requiere recorrer la lista de izquierda a derecha. La siguiente implementación de *asignar* usa la función *scanl1*. Recordemos, de la Sección 4.5, que

$$scanl1\,(\oplus)\,[x1, x2, x3] = [x1, x1 \oplus x2, (x1 \oplus x2) \oplus x3]$$

La definición de *asignar* es

$$\begin{aligned} & asignar && :: && [(Nombre, Nota)] \to Resultado \\ & asignar\,xs && = && scanl1\,reducir \cdot inicializar \\ & \\ & inicializar && :: && [(Nombre, Nota)] \to Resultado \\ & inicializar\,xs && = && zipWith\,hazTerna\,xs\,[1..] \\ & && && \textbf{where}\; hazTerna\,((xn, xm), xr) = (xn, xm, xr) \end{aligned}$$

La función *reducir* se define mediante

$$\begin{aligned} & reducir\,(xn, xm, xr)\,(yn, ym, yr) \\ = \; & \textbf{if}\; ym == xm\; \textbf{then}\; (yn, ym, xr)\; \textbf{else}\; (yn, ym, yr) \end{aligned}$$

Por tanto, cada candidato obtiene el mismo número de orden que cualquier candidato previo con la misma nota, o el rango inicialmente asignado si no existe tal candidato previo. Es obvio que, para una entrada de longitud *n*, la función *asignar* emplea un tiempo proporcional a *n*. El resultado es un programa para *orden* que emplea aproximadamente el tiempo necesario para llevar a cabo los dos procedimientos de ordenación. Como se indicó anteriormente, la implementación de *ordenar* necesita un tiempo proporcional a $n \log n$ para una lista de longitud *n*, de modo que la definición revisada de *orden* puede conducir a un ahorro sustancial.

En este punto, hemos obtenido la siguiente definición de *listaDeClase*:

$$\begin{aligned} listaDeClase = \; & ordenar\,nombre \cdot asignar \cdot reverse \cdot \\ & ordenar\,nota \cdot ordenar\,nombre' \cdot zipp \cdot \\ & cross\,(map\,fst \cdot ordenar\,iden, map\,snd) \end{aligned}$$

Las dos definiciones de *nombre* son ligeramente diferentes (ver más arriba), y ésa es la razón por la que la segunda aparición se ha distinguido con una prima. Pero es fácil ver que

$$ordenar\ f \cdot ordenar\ g \quad = \quad ordenar\ f$$

En otras palabras, no tiene sentido ordenar la lista dos veces. Por tanto, el término *ordenar nombre'* en la expresión anterior de *listaDeClase* es redundante.

5.2.3 Imprimiendo la lista de alumnos

La implementación de *mostrar* :: *Resultado* → *String* es inmediata porque no se nos pide una salida sofisticada. El método más simple es construir una lista de líneas de salida y aplanar el resultado para obtener simplemente una cadena:

```
mostrar  ::  [(Nombre, Nota, Orden)] → String
mostrar  =   unlines · (cabecera :) · map linea
```

La definición de *unlines* es similar a la de *concat*, con la sola diferencia de que, antes de concatenar, se insertan caracteres de fin de línea:

```
unlines  ::  [String] → String
unlines  =   foldr1 unir   where unir xs ys = xs ++ "¶" ++ ys
```

La cadena *cabecera* y la función *linea* dependen de cuántas columnas se necesiten para los nombres de los candidatos, la longitud de los números de identificación y el tamaño de las notas en notación decimal. Para concretar, tomamos

```
cabecera
  = ajustarIzq 12 "Nombre" ++ ajustarDer 4 "Nota" ++ ajustarDer 6 "Orden"
linea (xn, xm, xr)
  = ajustarIzq 12 xn ++ ajustarDer 4 (show xm) ++ ajustarDer 6 (show xr)
```

Las funciones *ajustarIzq* y *ajustarDer* se definen como sigue:

```
ajustarIzq, ajustarDer  ::  Int → String → String
ajustarIzq n xs         =   xs ++ espacios (n − length xs)
ajustarDer n xs         =   espacios (n − length xs) ++ xs
```

En particular, *ajustarIzq n xs* es la cadena *xs* completada con espacios extra a la derecha para conseguir una cadena de longitud total *n*. La definición de *espacios* se deja como ejercicio.

5.2.4 Ordenación

Queda por implementar la función *ordenar*. Daremos dos implementaciones, una simple y otra eficiente.

El método simple consiste en insertar los elementos de la lista uno por uno en el orden correcto:

```
ordenar      ::  (Ord β) ⇒ (α → β) → [α] → [α]
ordenar f    =   foldr (insertar f) [ ]
insertar     ::  (Ord β) ⇒ (α → β) → α → [α] → [α]
insertar f x =   insertar' · span test
```



```
where insertar' (xs, ys)  =  xs ++ [x] ++ ys
      test y              =  (f y < f x)
```

Se puede definir la función *span* mediante

```
span       ::  (α → Bool) → [α] → ([α], [α])
span p xs  =   (takeWhile p xs, dropWhile p xs)
```

En el Ejercicio 4.3.7 se consideraron las funciones *takeWhile* y *dropWhile*. En el Capítulo 7 se discute el modo de construir una definición más eficiente para *span*.

El problema con el método anterior de ordenación es que puede necesitar un tiempo proporcional a n^2 pasos para una lista de longitud *n*. Un método más eficiente de implementar *ordenar* consiste en usar una estrategia *divide y vencerás*: Si una lista contiene, a lo sumo, un elemento, entonces ya está ordenada; en caso contrario, se divide en dos mitades iguales, se ordena recursivamente cada mitad y, después, se funden las dos mitades ordenadas en una sola lista. Esta idea conduce al siguiente algoritmo:

```
ordenar f [ ]          =  [ ]
ordenar f [x]          =  [x]
ordenar f (x : y : xs) =  fundir f (divide (x : y : xs))
```

La función *divide* asigna cada elemento de la lista a una de las dos listas de forma estrictamente alternante:

```
divide  ::  [α] → ([α], [α])
divide  =   foldr repartir ([ ], [ ])   where repartir x (ys, zs) = (zs, x : ys)
```

En el Ejercicio 5.2.3 se propone una definición alternativa de *divide*.

La función *fundir* se define como sigue:

```
fundir              ::  Ord β ⇒ (α → β) → ([α], [α]) → [α]
fundir f ([ ], ys)  =   ys
fundir f (x : xs, [ ])  =  x : xs

fundir f (x : xs, y : ys)
  | f x ≤ f y       =  x : fundir f (xs, y : ys)
  | otherwise       =  y : fundir f (x : xs, ys)
```

Suponiendo que *f* necesita un tiempo constante, este programa para *ordenar f* necesita un tiempo proporcional a $n \log n$ pasos para una lista de longitud *n*. Demostraremos esta afirmación en el Capítulo 7.

Además de ilustrar adecuadamente el estilo de programación composicional, el problema de producir una lista de notas también demuestra la importancia de ordenar una lista eficientemente. El tiempo de ejecución del programa completo está dominado por el tiempo empleado para ordenar las diversas listas. Si hay *n* candidatos, entonces se puede producir la lista de alumnos en un tiempo proporcional a $n \log n$ pasos. El método ingenuo que los examinadores utilizan para hacerlo a mano necesita un tiempo proporcional a n^2 pasos.

Ejercicios

5.2.1 En la lista de pares candidato-número de identificación, ¿dónde se usó la suposición de que la lista estaba ordenada por orden alfabético?

5.2.2 Supongamos que *ordenar f* necesita como máximo $cn \log n$ pasos para ordenar una lista de longitud n, y que *fundir f* requiere como máximo cn pasos cuando el resultado es una lista de longitud n. Demostrar que *ordenar f* necesita como máximo $2cn\log(2n)$ para una lista de longitud $2n$.

5.2.3 Dar una definición alternativa (aunque no equivalente) de *divide* que haga uso de la función estándar *splitAt* definida en la Sección 4.2. La definición de *divide* dada en el texto realiza un único recorrido de la lista. ¿Cuántos realiza la suya?

5.2.4 La ecuación $reverse \cdot zipp = zipp \cdot cross\ (reverse, reverse)$ se cumple sobre pares de listas de igual longitud. Usar este hecho, junto con

$$reverse \cdot map\ f \quad = \quad map\ f \cdot reverse$$

para reescribir la definición de *listaDeClase*.

5.2.5 Se podría argumentar con razón que la función *ordenar* no es tan útil como una función más general *ordenarCon* cuyo tipo es

$$ordenarCon \quad :: \quad (\alpha \to \alpha \to Bool) \to [\alpha] \to [\alpha]$$

La especificación de *ordenarCon compara xs* es que devuelve una permutación *ys* de *xs* que satisface

$$and\ [compara\ x\ y \mid (x, y) \leftarrow zip\ ys\ (tail\ ys)]$$

Reescribir el algoritmo de ordenación de la lista de alumnos utilizando *ordenarCon*. Demostrar que no hay necesidad de invertir listas con este nuevo algoritmo.

5.2.6 El examen podría haber correspondido a una asignatura opcional; por tanto, algunos candidatos no aparecerán en la lista de notas. ¿Cómo resolvería este refinamiento del problema?

5.3 Aritmética entera de rango arbitrario

El objetivo de esta sección es mostrar cómo se puede implementar la aritmética entera de rango arbitrario —es decir, la aritmética de *Integer*— utilizando enteros de rango acotado —es decir, la aritmética de *Int*—. La definición completa en Haskell de una clase nueva de números enteros es una tarea ardua, por tanto sólo recorreremos una parte del camino.

Un entero de rango arbitrario será representado mediante una lista de dígitos en alguna base dada *base*. Cada dígito d se hallará en el rango $0 \le d < base$, excepto posiblemente el primero, que se hallará en el rango $-1 \le d < base$. Un número inicial de valor matemático -1 significará un entero negativo. La representación de los números negativos se considerará más adelante; por el momento supondremos que los enteros de rango arbitrario no son negativos.

La declaración

```
newtype Entero  =  Norm [Digito]
type Digito     =  Int
```

introduce *Entero* como un tipo nuevo, en vez de como un renombramiento, porque vamos a hacer que *Entero* sea miembro de la clase *Num* y, por tanto, necesitaremos concreciones de *Show* y *Ord* que son diferentes de las concreciones genéricas de estas clases para las listas. Por simplicidad, nos referiremos a los elementos de *Entero* como *numerales*.

La función de conversión

```
digitos          :: Entero → [Digito]
digitos (Norm xs)  =  xs
```

devuelve los dígitos de un numeral. Representaremos los numerales con el dígito más significativo en primer lugar. Esta representación es la habitual para los números decimales ordinarios, y a falta de una razón imperiosa para hacer lo contrario, es razonable adoptarla en nuestro módulo aritmético. Por tanto, representaremos un entero x como una secuencia $[x_0, x_1, \ldots, x_{n-1}]$ de dígitos, donde

$$x = x_0 \times base^{n-1} + x_1 \times base^{n-2} + \cdots + + x_{n-1} \times base^0$$

El criterio principal que condiciona la elección de la base es que queremos que las multiplicaciones de dos dígitos se efectúen mediante la multiplicación predefinida en *Int* sin correr el riesgo de salirnos de rango. Por otra parte, nos gustaría que nuestros dígitos fueran razonablemente grandes para mantener la representación de los numerales razonablemente pequeña. Para concretar, tomaremos

```
base, digitosBase  :: Int
base               =  10000
digitosBase        =  4
```

Esta elección supone que todos los números hasta 10^8 están en el rango permitido por la aritmética de rango acotado. Se necesita la cantidad *digitosBase* para imprimir los numerales. Para ilustrar la elección de *base*, observemos que

```
[1, 2345, 89]   representa  123450089
[1, 2, 3]       representa  100020003
```

Estas representaciones no son únicas, dado que se puede añadir un número arbitrario de ceros por la izquierda a un numeral sin que cambie su valor.

Podemos normalizar un numeral aplicándole una función *norm* que elimine los ceros no significativos. En particular, representaremos el número 0 mediante la lista vacía. Construiremos todos los numerales usando *norm*, de modo que, en la definición de las operaciones, podemos suponer que todos los argumentos numerales están en forma normal. Será conveniente, sin embargo, definir *norm* como una función más general, una función que convierta una lista arbitraria de enteros, en lugar de simplemente una lista de dígitos, en un numeral. Así, el tipo asignado a *norm* será *norm* :: [*Int*] → *Entero*.

Para especificar *norm*, definimos

```
convert  ::  [Int] → Integer
convert  =   foldl (⊕) 0 where n ⊕ d = base × n + d
```

Por tanto, *norm* se especifica mediante dos condiciones. En primer lugar,

```
convertir · digitos · norm  =  convertir
```

Esta condición establece que *norm* no cambia el valor del entero representado. En segundo lugar, *norm* debería devolver un valor *Norm xs*, donde *xs* es una lista de dígitos x en el rango $0 \le x < base$, excepto posiblemente para el primer dígito que puede estar en el rango $-1 \le x < base$.

Para implementar *norm*, procesamos la lista de enteros de derecha a izquierda y después reducimos cada dígito calculando el módulo con *base*, tras añadir un arrastre desde el paso anterior. El arrastre inicial es 0. Por ejemplo, suponiendo *base* = 10, la lista [11, 9, 16] se normaliza comenzando con un arrastre 0, reduciendo 16 al dígito 6 y a un arrastre 1, reduciendo 9 + 1 al dígito 0 y a un arrastre 1, reduciendo 11 + 1 al dígito 2 y a un arrastre 1 y, finalmente, añadiendo este arrastre para dar el número normalizado [1, 2, 0, 6].

Si el arrastre final c no está en el rango requerido $-1 \le c < base$, entonces se continúa procesando. Por ejemplo, suponiendo de nuevo *base* = 10, la lista [987] se normaliza a [7] con un arrastre de 98. Este arrastre no está en el rango requerido; por tanto, se reduce al dígito 8 y a un nuevo arrastre 9.

Se puede describir este proceso en términos de *foldr*. Definimos

```
type Arrastre   =  Int

arrastre        ::  Digito → (Arrastre, [Digito]) → (Arrastre, [Digito])
arrastre x (c, xs)  =  ((x + c) div base, (x + c) mod base : xs)
```

Entonces, *norm* se define mediante

```
norm  ::  [Int] → Entero
norm  =   Norm · dropWhile (== 0) · tratarArrastre · foldr arrastre (0, [ ])
```

El término (0, [ ]) asegura que la normalización empieza con un arrastre inicial de 0; el término *tratarArrastre* añade el arrastre final en forma de uno o más nuevos dígitos más significativos, y *dropWhile* (== 0) elimina los ceros por la izquierda. En el Ejercicio 4.3.7 se dio la definición de *dropWhile*. La lista final de dígitos se convierte en un numeral aplicando la constructora *Norm*. La definición de *tratarArrastre* es

```
tratarArrastre          ::  (Arrastre, [Digito]) → [Digito]
tratarArrastre (c, xs)  =   if (−1 ≤ c) ∧ (c < base) then c : xs
                            else tratarArrastre (c div base, c mod base : xs)
```

Dejaremos como ejercicio demostrar que *tratarArrastre* termina y que *norm* satisface la primera condición de su especificación. Por construcción, es obvio que *norm* devuelve una lista de dígitos, todos ellos en el rango requerido.

5.3.1 Operaciones de comparación

Dada nuestra elección de la representación, con el dígito más significativo a la izquierda, las operaciones de comparación de numerales pueden basarse en la ordenación lexicográfica estándar de las listas. Debemos alinear primero las dos listas

añadiendo ceros no significativos a la lista más corta. Por ejemplo, 57 es menor que 126 porque 057 es lexicográficamente menor que 126. Podemos alinear dos listas de dígitos mediante

```
alinear :: ([Digito], [Digito]) → ([Digito], [Digito])
alinear (xs, ys)
  | n > 0   =  (replicar n 0 ++ xs, ys)
  | n ≤ 0   =  (xs, replicar (−n) 0 ++ ys)
  where n   =  length ys − length xs
```

La función *replicar* se puede definir mediante

```
replicar      ::  Int → α → [α]
replicar n x  =   [x | k ← [1..n]]
```

Ahora podemos introducir las operaciones de comparación mediante

```
instance Eq Entero where
  (==)  =  traducir (==)

instance Ord Entero where
  (<)   =  traducir (<)

traducir          ::  ([Digito] → [Digito] → Bool) → (Entero → Entero → Bool)
traducir op x y   =   op xs ys
                      where (xs, ys) = alinear (digitos x, digitos y)
```

También se puede implementar directamente la comprobación de si un numeral es cero mediante

```
esCero  ::  Entero → Bool
esCero  =   null · digitos
```

Esta definición asume que el argumento es un numeral en forma normal.

5.3.2 Suma y resta

Se pueden definir con facilidad las funciones (+) y (−) para sumar y restar numerales de rango arbitrario. Podemos alinear las dos listas de dígitos, realizar la operación dígito a dígito y después normalizar el resultado. Por ejemplo, supongamos que queremos sumar los números [7, 3, 7] y [4, 6, 9], donde, temporalmente, supondremos que *base* = 10. La suma dígito a dígito de estos números es [11, 9, 16] y, como vimos antes, el resultado normalizado es [1, 2, 0, 6]. Se pueden definir las operaciones de suma y resta con sus nombres convencionales, haciendo que *Entero* sea una concreción de la clase *Num*:

```
instance Num Entero where
x + y     =  norm (zippWith (+) (alinear (digitos x, digitos y)))
x − y     =  norm (zippWith (−) (alinear (digitos x, digitos y)))
x × y     =  ...
negate x  =  ...
```

la definición de (×) y *negate* se considerará más adelante. Recordemos que *zippWith* es la versión no currificada de *zipWith*. La clase de tipos *Num* tiene a *Eq* y a *Show* como superclases; por tanto, la creación de *Entero* como tipo de la clase *Num* estará incompleta hasta que declaremos *Entero* como concreción de la clase *Show*. Se pospone esta tarea hasta el final de la sección.

El aspecto más interesante de la resta es ver qué sucede cuando el segundo argumento es mayor que el primero, de modo que el resultado sea un número negativo. Por ejemplo, suponiendo de nuevo que *base* = 10, al restar [3, 7, 5] de [1, 0, 6] obtenemos la lista [−2, −7, 1] después de restar dígito a dígito, y la normalización produce *Norm* [−1, 7, 3, 1]. Estos dígitos representan el número

$$-1 \times 10^3 + 7 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 \quad = \quad -269$$

La respuesta es correcta ya que 106 − 375 = −269. Después de la normalización, se indica que un número es negativo añadiendo a la izquierda un dígito de valor −1. Se puede obtener el valor absoluto de un número negativo, negando todos los dígitos de la representación y normalizando. De aquí se deduce que podemos definir el predicado *negativo*, que comprueba si el resultado de una resta es negativo, y la función *negate*, para negar un número, mediante

```
negativo          :: Entero → Bool
negativo (Norm xs) = not (null xs) ∧ (head xs < 0)

negate            :: Entero → Entero
negate             = norm · map neg · digitos   where neg x = −x
```

Este método para representar los números negativos se denomina *complemento con signo*. La representación mediante complemento con signo es conveniente tanto para las operaciones aritméticas como para las de comparación, porque no hace falta tomar medidas especiales para los argumentos y los resultados negativos. Dejamos como ejercicio la demostración de que las operaciones de comparación funcionan, sin necesidad de modificación para los números negativos.

Otra representación de los números negativos es la que se utiliza normalmente en los cálculos decimales realizados a mano. Se llama notación *magnitud con signo*. Aquí, un número negativo se denota mediante su valor absoluto junto con una indicación de si es positivo o negativo. La aritmética utilizando la representación basada en magnitud con signo necesita de un análisis por casos sobre el signo del número.

5.3.3 Multiplicación

A continuación, definimos (×) sobre el tipo *Entero*. La definición inmediata es una traducción del método aprendido en la escuela, según el cual multiplicamos el multiplicando por cada dígito del multiplicador y después sumamos los resultados parciales, tras desplazarlos adecuadamente. La función *sumas* devuelve la lista de las sumas parciales ordenadas decrecientemente, siendo

```
sumas     :: Entero → Entero → [Entero]
sumas x y  = map (mult1 x) (digitos y)

mult1     :: Entero → Digito → Entero
```



```
mult1 x d  = norm (map (× d) (digitos x))
```

La función *mult1* multiplica un numeral por un solo dígito. Para hacer el desplazamiento y la suma, definimos *despSum* mediante

```
despSum          :: Entero → Entero → Entero
despSum x y       = desplazar x + y

desplazar        :: Entero → Entero
desplazar (Norm xs) = if null xs then Norm [ ] else Norm (xs ++ [0])
```

La definición de *desplazar* no utiliza *norm*, pero si *xs* está normalizada y no es vacía, entonces sucede lo mismo con *xs* ++ [0]. Ahora podemos definir

```
x × y  = foldl despSum (Norm [ ]) (sumas x y)
```

El lector debería preguntarse por qué usamos aquí *foldl* en lugar de *foldr* o *foldl1*.

5.3.4 División

Finalmente, nos ocupamos de la división de un entero por otro. Como todo estudiante de primaria sabe, la división es la operación aritmética más difícil de realizar; no sólo el algoritmo es complicado, sino que también necesita una cierta dosis de adivinación. De hecho, el siguiente algoritmo de división es probablemente el programa más complejo de este libro. Definimos la división solamente para números no negativos, dejando su extensión a los números negativos como ejercicio. Nuestra intención es definir una función

```
cocResto  :: Entero → Entero → (Entero, Entero)
```

que devuelva el cociente y el resto de la división de un número no negativo por un número positivo. Sin embargo, no haremos que *Entero* sea una concreción de la clase *Integral*, de modo que nos está prohibido usar los nombres **div** y **mod**. En la jerarquía de las clases de tipos de Haskell, por encima de *Integral* existen otras, incluyendo *Real* y *Enum*, y no deseamos complicar las cosas discutiendo esta jerarquía.

Antes de abordar el diseño del algoritmo, consideremos en primer lugar un problema más simple, a saber, calcular el cociente y el resto cuando dividimos un numeral por un número de un solo dígito. La función correspondiente es

```
cocResto1  :: Entero → Digito → (Entero, Digito)
```

Por ejemplo, consideremos la división de 369 entre 5. Los pasos de este proceso se aprenden en la escuela: primeramente dividimos 3 entre 5 dando por cociente 0 y por resto 3; dividimos después 36 entre 5 dando por cociente 7 y por resto 1; finalmente dividimos 19 entre 5 dando por cociente 3 y por resto final 4. El cociente final es 073.

En general, supongamos que el numeral es $x = Norm\ [x_0, x_1, \ldots, x_{n-1}]$ y que d es el dígito. Los dígitos (no normalizados) $[c_0, c_1, \ldots, c_{n-1}]$ del cociente se pueden calcular estableciendo un resto inicial $r_0 = 0$ y evaluando, para $0 \le i < n$,

$$\begin{aligned} c_i &= (base \times r_i + x_i)\ \mathbf{div}\ d \\ r_{i+1} &= (base \times r_i + x_i)\ \mathbf{mod}\ d \end{aligned}$$

El resto, de un solo dígito, es r_n. Podemos implementar este esquema utilizando la función *scanl*. Recordemos que

$$scanl\ (\oplus)\ e\ [x_0, x_1, \ldots] \quad = \quad [e, e \oplus x_0, (e \oplus x_0) \oplus x_1, \ldots]$$

Podemos definir

```
cocresto1        :: Entero → Digito → (Entero, Digito)
cocresto1 x d    =  terminar1 (scanl (paso1 d) (0, 0) (digitos x))

paso1            :: Digito → (Digito, Digito) → Digito → (Digito, Digito)
paso1 d (c, r) x =  (y div d, y mod d) where y = base × r + x

terminar1        :: [(Digito, Digito)] → (Entero, Digito)
terminar1 crs    =  (norm (map fst crs), snd (last crs))
```

La función *terminar1* finaliza el cálculo normalizando los dígitos del cociente y extrayendo el resto del final.

Para implementar el algoritmo completo podemos adoptar, inicialmente, la misma estructura. Más adelante necesitaremos revisarla ligeramente. Por tanto, tomamos

```
cocresto         :: Entero → Entero → (Entero, Entero)
cocresto x y     =  terminar (scanl (paso y) (0, Norm [ ]) (digitos x))

terminar         :: [(Digito, Entero)] → (Entero, Entero)
terminar crs     =  (norm (map fst crs), snd (last crs))
```

La diferencia entre *cocresto* y *cocresto1* es que, en la primera, el resto obtenido en cada paso no consta necesariamente de un solo dígito. Las funciones *terminar* y *terminar1* difieren solamente en sus tipos. El tipo de *paso* es

```
paso :: Entero → (Digito, Entero) → Digito → (Digito, Entero)
```

La evaluación de *paso y* (*c*, *r*) *d* supone dividir el numeral x = *despSum1 r d* entre el numeral y donde

```
despSum1         :: Entero → Digito → Entero
despSum1 r d     =  Norm (digitos r ++ [d])
```

La función *despSum1* es similar a la función *despSum* utilizada en la definición de la función de multiplicación, excepto por un cambio en el tipo. En el algoritmo de *paso* necesitaremos distinguir tres casos, dependiendo del número de dígitos en x y en y; por tanto, definimos

```
m  =  length (digitos x) (= length (digitos r) + 1)
n  =  length (digitos y)
```

Nótese que $m \leq n + 1$ dado que r, al ser el resto del paso anterior, es menor que y y por tanto no puede tener más dígitos que y. La definición de *paso* es

```
paso y (c, r) d
  | m < n      =  pasoA x y
  | m == n     =  pasoB x y
  | otherwise  =  pasoC x y
  where m      =  length (digitos r) + 1
        n      =  length (digitos y)
        x      =  despSum1 r d
```


Los tres casos, *pasoA*, *pasoB* y *pasoC*, están en orden creciente de dificultad. La definición de *pasoA* es fácil: si $m < n$, entonces el nuevo cociente es 0 y el nuevo resto es x. Por tanto

```
pasoA      :: Entero → Entero → (Digito, Entero)
pasoA x y  =  (0, x)
```

A continuación, consideremos *pasoB*. Si $m = n$, entonces el nuevo cociente c podría ser cualquier valor en el rango $0 \leq c < base$. Sin embargo, veremos que, en la versión final del algoritmo, impondremos como condición al divisor y que $y \geq base/2$. Por tanto ha de ser un cociente que valga a lo sumo 1 y podemos definir

```
pasoB      :: Entero → Entero → (Digito, Entero)
pasoB x y  =  if negativo z then (0, x) else (1, z)
              where z = x − y
```

El caso más complicado es *pasoC* cuando $m = n + 1$. Supongamos que $x = x_0\ x_1 \ldots x_n$, e $y = y_1\ y_2 \ldots\ y_n$ son ambos numerales no negativos que cumplen $x/y < base$. Nos gustaría disponer de un modo rápido de estimar $\lfloor x/y \rfloor$ y podemos hacerlo considerando los dos primeros dígitos de x y el primer dígito de y. Por ejemplo, suponiendo que $base = 10$, $x = 37???$, e $y = 4???$, deberíamos estimar un cociente de 9 dado que $\lfloor 37/4 \rfloor = 9$. De forma similar, con $x = 27???$ y la misma y, estimaríamos un cociente de 6. Por otra parte, con $x = 41???$ y la misma y, estimaríamos un cociente de 9, y no de 10, porque 9 es el mayor cociente posible. En consecuencia, definimos la estimación $\hat{c}$ mediante

$$\hat{c} \quad = \quad \lfloor (x_0 \times base + x_1) / y_1 \rfloor \ \mathbf{min}\ (base - 1)$$

¿Cuánto puede diferir $\hat{c}$ del verdadero cociente $c = \lfloor x/y \rfloor$? La respuesta, demostrada en el Ejercicio 5.3.5, es que si $y_1 \geq base/2$, entonces tenemos $\hat{c} - 2 \leq c \leq \hat{c}$. Dicho con palabras, siempre que y_1 sea suficientemente grande, la estimación $\hat{c}$ sólo puede sobrestimar a c, y además en 2 unidades como máximo.

Nos ocuparemos de la suposición $y_1 \geq base/2$ enseguida; primero mostraremos cómo determinar si $\hat{c}$ es demasiado grande. En primer lugar, por definición de las operaciones de truncado, $\lfloor x/y \rfloor = \hat{c}$ si y sólo si

$$\hat{c} \times y \leq x < \hat{c} \times y + y$$

A continuación, dado que $\hat{c}$ nunca es una estimación por defecto, se cumple

$$x < \lfloor x/y \rfloor \times y + y \leq \hat{c} \times y + y$$

Por tanto, obtenemos que $\lfloor x/y \rfloor = \hat{c}$ si y sólo si $\hat{c} \times y \leq x$. Esto conduce a la siguiente implementación de *pasoC*:

```
pasoC :: Entero → Entero → (Digito, Entero)
pasoC x y
```

```
| not (negativo r0)   =  (c, r0)
| not (negativo r1)   =  (c − 1, r1)
| otherwise           =  (c − 2, r2)
where c   =  estima x y
      r0  =  x − mult1 y c
      r1  =  r0 + y
      r2  =  r1 + y
```

La función *estima* se define mediante

```
estima      ::  Entero → Entero → Digito
estima x y  =   if x0 ≥ y1 then base − 1 else (x0 × base + x1) div y1
                where x0  =  xs !! 0
                      x1  =  xs !! 1
                      xs  =  digitos x
                      y1  =  (digitos y) !! 0
```

Esto nos deja únicamente el problema de asegurar que el primer dígito del divisor y es como mínimo *base/2*. Podemos hacer esto multiplicando tanto el dividendo x como el divisor y por $d = \lfloor base/(y_1 + 1) \rfloor$. El Ejercicio 5.3.6 pide demostrar esta afirmación. El resultado de este aumento hará que el resto final sea demasiado grande y tendremos que dividirlo por d al final del algoritmo. De acuerdo con ello, revisamos nuestra definición anterior de *cocResto* para obtener

```
cocResto      ::  Entero → Entero → (Entero, Entero)
cocResto x y  =   terminar d (scanl (paso y') (0, Norm [ ]) (digitos x'))
                  where x'  =  mult1 x d
                        y'  =  mult1 y d
                        d   =  base div (head (digitos y) + 1)

terminar          ::  Digito → [(Digito, Entero)] → (Entero, Entero)
terminar d crs    =   (norm (map fst crs), div1 (snd (last crs)) d)

div1              ::  Entero → Digito → Entero
div1 x d          =   fst (cocResto1 x d)
```

Como dijimos al principio, la división es un proceso complicado; ahora el lector podrá apreciar por qué.

5.3.5 Convirtiendo los numerales en cadenas

La declaración de *Entero* como concreción de *Num* no es válida en Haskell, a menos que declaremos a *Entero* como concreción de la clase de tipos *Show*. Al tratar la clase *Show* en la Sección 2.7 dijimos que esto implicaba definir una función

```
showsPrec  ::  Int → α → String → String
```

La función *showsPrec* toma un entero como argumento para permitir al usuario controlar si la cadena impresa está encerrada entre paréntesis. Entonces toma el valor que ha de ser convertido y devuelve, no una cadena, sino una función de cadenas en cadenas. La función *showsPrec* ha de satisfacer la ley


```
showsPrec d x xs ++ ys  =  showsPrec d x (xs ++ ys)
```

El argumento extra *ys* se denomina *parámetro acumulador* y se introduce por razones de eficiencia. Este tema se explicará en el Capítulo 7. La función auxiliar *show* se define mediante

```
show x  =  showsPrec 0 x [ ]
```

El argumento 0 significa que el resultado nunca estará delimitado por paréntesis.

Para ser exhaustivos, damos la siguiente declaración de concreción de *show*. Las funciones no definidas son parte del preludio estándar de Haskell (de hecho, *showChar* es un sinónimo de (:) y *showString* es un sinónimo de (++)).

```
instance Show Entero where
showsPrec k x
  | esCero x     =  showChar '0'
  | negativo x   =  showChar '−' · showDigitos (digitos (negate x))
  | otherwise    =  showDigitos (digitos x)

showDigitos :: [Digito] → String → String
showDigitos (d : ds)
  =  showString (show d) · showString (concat (map showDigito ds))

showDigito     ::  Digito → String
showDigito s   =   ceros ++ ds
                   where ceros  =  replicar (digitosBase − length ds) '0'
                         ds     =  show d
```

Todos los dígitos, excepto el que está más a la izquierda, han de ser completados con el número correcto de caracteres 0.

Ejercicios

5.3.1 El programa para *tratarArrastre* termina si, los sucesivos reemplazos de c por c **div** *base* conducen a un valor c en el rango $-1 \le c < base$. Demostrar que esto es efectivamente así.

5.3.2 Demostrar que *convertir* · *digitos* · *norm* = *convertir*.

5.3.3 Hay otro modo de definir *norm* usando la función *scanr*. Recordemos que

$$scanr\ (\oplus)\ e\ [x_0, x_1, x_2] \quad = \quad [x_0 \oplus (x_1 \oplus (x_2 \oplus e)), x_1 \oplus (x_2 \oplus e), x_2 \oplus e, e]$$

Definir $\oplus$ mediante $x \oplus (c, y) = ((x + c)$ **div** $base, (x + c)$ **mod** $base)$. A partir de ello, definir *norm* utilizando *scanr*.

5.3.4 Demostrar que las operaciones de comparación funcionan adecuadamente si se permite que los argumentos sean números negativos.

5.3.5 Sea $x = x_0\ x_1 \ldots x_n$ e $y = y_1\ y_2 \ldots y_n$. Establecemos $\hat{c} = \lfloor (x_0\ b + x_1)/y_1 \rfloor$, de modo que

$$\hat{c}y_1 \le x_0\ b + x_1 < \hat{c}y_1 + y_1$$

Utilizando exclusivamente estas desigualdades, además de las dos siguientes:

$$y_1\, b^{n-1} \le y < y_1\, b^{n-1} + b^{n-1}$$
$$x_0\, b^n + x_1\, b^{n-1} \le x < x_0\, b^n + x_1\, b^{n-1} + b^{n-1}$$

demostrar que si $y_1 \ge b/2$, entonces $\hat{c} - 2 \le \lfloor x/y \rfloor < \hat{c} + 1$.

5.3.6 Supongamos que y y b son enteros que satisfacen $1 \le y < b$. Demostrar que

$$\lfloor b/2 \rfloor \le y \lfloor b/(y+1) \rfloor < b$$

5.3.7 Construir una función *aEntero* :: *String* → *Entero* que convierta una cadena de dígitos decimales en un elemento de *Entero*.

5.3.8 La definición de *cocResto* dada en el texto se puede refinar de varias maneras. En particular, se puede mejorar la estimación de c —véase Knuth (1981)— y se pueden evitar ciertos cálculos de longitudes utilizando versiones de la suma y de la resta que producen resultados de $(n + 1)$ dígitos para argumentos de n dígitos. Mostrar cómo hacer *cocResto* más eficiente.

5.4 Impresión de un calendario

En esta sección, construiremos un programa para imprimir un calendario. La intención es diseñar dos funciones *calendario* y *mes* tales que, por ejemplo, *calendario* 2000 producirá un calendario completo para el año 2000 y *mes* (1, 2000) producirá

```
                Enero 2000
Do Lu Ma Mi Ju Vi Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

La manera adecuada de abordar problemas de esta clase es separar la fase de construcción de la fase de impresión. Podemos considerar como tareas lógicamente independientes la construcción de un calendario abstracto y su impresión en el formato requerido. Si tenemos éxito en esta separación de aspectos, las modificaciones sobre el programa, tales como imprimir el calendario en un formato diferente o imprimir solamente una parte del calendario, serán más fáciles de realizar.

5.4.1 Construyendo un calendario

El propósito principal de un calendario es proporcionar una presentación adecuada de la relación entre lo nombres de los días de la semana y las fechas. Cada fecha está asociada con un único día de la semana; por tanto, el problema abstracto consiste en determinar una función *dia* que tome una fecha y devuelva el nombre correcto del día de la semana que le corresponde. Dado que necesitaremos realizar operaciones

aritméticas sobre los nombres de los días de la semana, los codificaremos como números, siendo el Domingo el 0, el Lunes el 1 y así sucesivamente. De acuerdo con ello, introducimos los siguientes tipos renombrados

```
type Fecha     = (Dia, Mes, Anyo)
type Dia       = Int
type Mes       = Int
type Anyo      = Int
type NombreDia = Int
```

Dada una función *primerDia* :: (*Mes, Anyo*) → *NombreDia* que devuelve el nombre del primer día de la semana del mes dado, podemos definir *dia* mediante

```
dia          :: Fecha → NombreDia
dia (d, m, a) = (primerDia (m, a) + d − 1) mod 7
```

En particular, *dia* (1, *m*, *a*) = *primerDia* (*m*, *a*).

Para determinar *primerDia* necesitamos saber las longitudes de los meses y el día de la semana en que cae el primero de Enero del año considerado. Entonces, el valor de *primerDia* se obtiene indexando en una lista que contiene todos los primeros días de la semana para ese año:

```
primerDia          :: (Mes, Anyo) → nombreDia
primerDia (m, a)   = (primerosDias a) !! (m − 1)

primerosDias       :: Anyo → [nombreDia]
primerosDias       = take 12 · map (mod 7) · acumMes

acumMes            :: Anyo → [Int]
acumMes a          = scanl (+) (unoDeEnero a) (totalesMes y)

totalesMes         :: Anyo → [Int]
totalesMes a       = [31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
                     where feb = if bisiesto a then 29 else 28

bisiesto           :: Anyo → Bool
bisiesto a         = if a mod 100 = 0 then (a mod 400 = 0) else (a mod 4 = 0)
```

La lista de los primeros días de la semana para cada mes se obtiene calculando los totales acumulados de las longitudes de los meses, comenzando por el día en que cae el primero de Enero, reduciendo los totales módulo 7 para encontrar los días de la semana que corresponden a los primeros días de cada mes, y finalmente tomando los primeros 12 valores. Nótese que al aplicar *scanl* a una lista de 12 valores se obtiene una lista de longitud 13; el último valor de esta lista será el día en que comienza el 1 de Enero del año siguiente.

Queda por definir *unoDeEnero*. Según el calendario gregoriano, el 1 de Enero del año 1 fue Lunes. Por tanto, para calcular *unoDeEnero a*, tenemos que contar todos los días de los primeros $(a - 1)$ años, sumarle 1 y calcular el módulo 7 del resultado:

```
unoDeEnero a   :: Anyo → DayNombre
```

```
unoDeEnero a  =  (365 × x + x div 4 − x div 100 + x div 400 + 1) mod 7
                 where x = a − 1
```

Los términos adicionales tienen en cuenta los años bisiestos.

5.4.2 Dibujos

Consideremos ahora cómo dibujar un calendario. El calendario de un año se construye a partir de los calendarios de los meses individuales; por tanto, la idea es construir dibujos grandes a partir de dibujos más pequeños. Por el momento, es mejor considerar que un dibujo es simplemente un elemento de algún tipo *Dibujo*. Cada dibujo es una entidad rectangular que tiene una altura y anchura positivas, de modo que existen dos funciones primitivas definidas sobre dibujos: *altura* y *anchura*. Introducimos dos renombramientos

```
type Altura   =  Int
type Anchura  =  Int
```

y escribiremos los tipos de *altura* y *anchura* de la forma

```
altura   ::  Dibujo → Altura
anchura  ::  Dibujo → Anchura
```

Para empezar, necesitamos algunos dibujos elementales. Como nuestros dibujos son dibujos de texto, y la pieza más pequeña de texto es un simple carácter, podemos resolver el problema con una sola función

```
casilla  ::  Char → Dibujo
```

que convierta un carácter en un dibujo (1 × 1).

Podemos construir unos dibujos a partir de otros de diversas maneras. Dos métodos obvios consisten en colocar un dibujo encima de otro o bien uno al lado del otro. Por tanto, introducimos dos operaciones que combinan dibujos

```
(sobre), (juntoA)  ::  Dibujo → Dibujo → Dibujo
```

Está claro lo que sucedería si las anchuras de los dibujos *p* y *q* son iguales: la anchura de (*p* **sobre** *q*) será la anchura común, y su altura será la suma de las alturas de *p* y *q*. Si las anchuras de *p* y *q* son distintas, entonces hemos de tomar una decisión: podemos alinear el dibujo más estrecho con el más ancho de algún modo o simplemente no definir el valor (*p* **sobre** *q*). Comentarios similares se aplican a la operación (*p* **juntoA** *q*). En vez de decidir ahora, esperaremos a ver cuáles son las decisiones más lógicas que surgen a lo largo del desarrollo. Sea cual sea la decisión que adoptemos, ambos operadores serán asociativos; no importa si ponemos *p* encima de *q* y después el resultado encima de *r*, o ponemos *p* encima del resultado de poner *q* encima de *r*.

Dadas **sobre** y **juntoA**, podemos apilar en vertical, uno sobre otro, los dibujos de una lista no vacía, o apilar en horizontal, uno al lado del otro, los dibujos de una lista no vacía:

```
apilarV, apilarH  ::  [Dibujo] → Dibujo
apilarV           =   foldr1 (sobre)
apilarH           =   foldr1 (juntoA)
```


Para evitar introducir dibujos vacios, escogemos apilar en vertical y en horizontal solamente listas no vacías.

Utilizando *apilarH*, podemos definir una función que convierta una cadena no vacía en un dibujo:

```
fila  ::  String → Dibujo
fila  =   apilarH · map casilla
```

La función *fila* convierte una cadena en un dibujo (1 × n), donde $n > 0$ es la longitud de la cadena. Usando *fila* podemos definir un dibujo en blanco de una altura y anchura dadas:

```
blanco  ::  (Altura, Anchura) → Dibujo
blanco  =   apilarV · map fila · blancos
            where blancos (h, a) = replicar h (replicar a '␣')
```

Introducimos dos variantes útiles de *apilarV* y *apilarH* que insertan dibujos en blanco entre cada dos dibujos además de combinarlos:

```
apilarVCon      ::  Altura → [[Dibujo]] → Dibujo
apilarVCon h    =   foldr1 (⊕)
                    where p ⊕ q = p sobre (blanco (h, anchura q) sobre q)

apilarHCon      ::  Anchura → [[Dibujo]] → Dibujo
apilarHCon a    =   foldr1 (⊕)
                    where p ⊕ q = p juntoA (blanco (altura q, a) juntoA q)
```

La función *apilarVCon h* apila una lista de dibujos en vertical con una separación entre dibujos de una altura *h*; de modo similar podemos definir *apilarHCon a*.

Dada una lista de listas de dibujos podemos construir un dibujo más grande haciendo una cuadrícula, esto es, disponiendo los dibujos en forma de matriz rectangular. Existen dos variantes:

```
cuadricula          ::  [[Dibujo]] → Dibujo
cuadricula          =   apilarV · map apilarH

cuadriculaCon       ::  (Altura, Anchura) → [[Dibujo]] → Dibujo
cuadriculaCon (h, a)  =  apilarVCon h · map (apilarHCon a)
```

Finalmente, necesitaremos una función para mostrar un dibujo en la pantalla:

```
mostrar  ::  Dibujo → String
```

Volveremos a los dibujos y a sus representaciones después de analizar cómo usar los operadores de combinación para dibujar calendarios.

5.4.3 Componiendo el calendario

Recordemos que nuestro objetivo es definir una función *calendario* que muestre el calendario de un año y una función *mes* que dibuje el calendario de un solo mes. Una manera de definir *calendario* consiste en escribir

```
calendario  ::  Anyo → String
calendario  =   mostrar · cuadriculaCon (1, 4) · agrupar 3 · map dibujaUnMes · crear
```

La función *calendario* se define como la composición de cinco funciones más simples: *crear*, que toma un año y lo convierte en una lista de longitud 12 con información sobre cada mes; *dibujaUnMes*, que convierte esta información en el dibujo de un mes; *agrupar* 3, que agrupa estos dibujos en una matriz (4 × 3) de dibujos; *cuadriculaCon* (1, 4) combina estos dibujos poniendo una fila en blanco entre las componentes verticales y cuatro columnas en blanco entre las componentes horizontales; y finalmente, *mostrar*, que convierte el dibujo en una forma imprimible. Se puede definir la función *mes* usando un esquema similar, y se deja como ejercicio. La función *agrupar n* se define mediante

```
agrupar      :: Int → [α] → [[α]]
agrupar n xs  = if null ys then [ ] else ys : agrupar n zs
                where (ys, zs) = splitAt n xs
```

La función *splitAt* se definió en la Sección 4.2.

Quedan por definir *dibujaUnMes* y *crear*. Para cada mes necesitaremos su nombre, el año al cual pertenece, el día de la semana del primer día del mes y el número de días del mes. Por tanto definimos

```
crear     :: Anyo → [(String, Anyo, DayNombre, Int)]
crear a    = zipp4 (nombresMes, replicar 12 a, primerosDias a, totalesMes a)
```

La función *zipp4* es similar a *zipp* excepto por el hecho de que produce una lista de cuádruplas a partir de una cuádrupla de listas.

Los nombres de los meses vienen dados por la siguiente lista:

```
nombresMes  :: [String]
nombresMes  = ["Enero", "Febrero", "Marzo",
               "Abril", "Mayo", "Junio",
               "Julio", "Agosto", "Septiembre",
               "Octubre", "Noviembre", "Diciembre"]
```

Las funciones *primerosDias* y *totalesMes* fueron definidas con anterioridad.

Ahora nos ocuparemos de *dibujaUnMes*. El dibujo de un mes consistirá en una cabecera y un cierto resto, por tanto definimos

```
dibujaUnMes              :: (String, Anyo, nombreDia, Int) → Dibujo
dibujaUnMes (m, a, d, t)  = cabecera (m, a) sobre matriz (d, t)
```

Tratemos primero la matriz. Supongamos que creamos una matriz de números consecutivos dispuestos de tal modo que el primer día del mes ocupa el lugar correcto. Por ejemplo, siendo $d = 6$ (Sábado) obtenemos la matriz

-5	-4	-3	-2	-1	0	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	32	33	34	35	36

Cada uno de estos números se convertirá en un pequeño dibujo de altura 1 y anchura 3 y después se combinarán con la función *cuadricula*:

```
matriz          = cuadricula · agrupar 7 · casillas
casillas (d, t) = map (fila · ajustarDer 3 · dibujaUnDia) [1 − d .. 42 − d]
                  where dibujaUnDia n = if 1 ≤ n ∧ n ≤ t then show n else ""
```

El resultado es un dibujo de anchura $7 \times 3 = 21$. Usando esta información, podemos definir *cabecera* mediante

```
cabecera (m, a) = rotulo (m, a) sobre nombresDia
rotulo (m, a)   = fila (ajustarDer 21 (m ++ "␣" ++ show a))
nombresDia      = fila "␣Do␣Lu␣Ma␣Mi␣Ju␣Vi␣Sa"
```

Con esto completamos la definición de *calendario*.

5.4.4 Representando los dibujos

La tarea final consiste en crear una representación para los dibujos. Una posibilidad es definir

```
type Dibujo = (Altura, Anchura, [[Char]])
```

Sin embargo, no toda terna (*h*, *a*, *xss*) representa un dibujo válido; exigiremos que $h = length\ xss$ y $h > 0$; además, *a* debería ser la longitud común (positiva) de los componentes de *xss*. Con esta representación tendríamos

```
altura (h, a, xss)  = h
anchura (h, a, xss) = a
casilla c           = (1, 1, [[c]])
```

Podemos implementar los operadores de combinación **sobre** y **juntoA** del modo siguiente:

```
(h, a, xss) sobre (h', a', yss)
  | (a == a')    = (h + h', a, xss ++ xss yss)
  | otherwise    = error "sobre: anchuras diferentes"

(h, a, xss) juntoA (h', a', yss)
  | (h == h')    = (h, a + a', zippWith (++) xss yss)
  | otherwise    = error "juntoA: alturas diferentes"
```

Dado que no hemos explotado la idea de colocar dos dibujos con anchuras diferentes uno encima del otro, o dos dibujos con alturas diferentes uno al lado del otro, hemos decidido hacer ambas operaciones parciales.

Queda por definir *mostrar*:

```
mostrar          :: Dibujo → String
mostrar (h a, xss) = unlines xss
```

La función *unlines* se definió en la Sección 5.2 y será analizada de nuevo en la sección siguiente. Las restantes funciones sobre dibujos están definidas en términos de las funciones anteriores; por tanto, hemos terminado.

Ejercicios

5.4.1 Simplificar la expresión para la función *unoDeEnero a* utilizando la propiedad (365 **mod** 7) = 1.

5.4.2 Definir *mes* :: (*Mes, Anyo*) → *String*.

5.4.3 Definir una versión del problema del calendario que imprima un mes con los días de la semana escritos de Lunes a Domingo.

5.4.4 Definir una versión del calendario que tenga los días hacia abajo en el lado izquierdo en lugar de en la parte de arriba.

5.4.5 ¿Bajo qué condiciones se cumple la siguiente ley de intercambio?

$$(p \textbf{ sobre } q) \textbf{ juntoA } (r \textbf{ sobre } s) \quad = \quad (p \textbf{ juntoA } r) \textbf{ sobre } (q \textbf{ juntoA } s)$$

5.4.6 Desarrollar un álgebra de dibujos modificada que contenga un dibujo nulo *vacio* de altura y anchura cero y que sea el elemento neutro de (**sobre**) y (**juntoA**).

5.5 Procesamiento de textos

En esta sección nuestro propósito es construir un grupo de funciones para procesar textos de diversos modos. La mayoría de estas funciones se definirán en parejas; cada función de la pareja hará el trabajo opuesto al de la otra. Se puede considerar un texto de diversas maneras. La perspectiva básica es que un texto es simplemente una lista de caracteres, que incluye tanto los visibles como los no visibles. Por el momento supondremos que existen solamente dos caracteres invisibles, un carácter en blanco '␣' y un carácter de fin de línea '⏎'. Más adelante permitiremos también un carácter de tabulación, que denotaremos mediante '⊣'. Dependiendo del problema, puede ser más conveniente tratar un texto como una lista de palabras o como una lista de líneas, e incluso como una lista de párrafos. Nuestro objetivo principal es desarrollar funciones que conviertan textos pasando de una perspectiva a la otra.

Comenzaremos por introducir algunos tipos renombrados:

```
type Texto   = [Char]
type Linea   = [Char]
type Palabra = [Char]
```

Por definición, una *Linea* es una lista de caracteres que no contenga caracteres de fin de línea y una *Palabra* es una lista no vacía de caracteres que no contengan espacios o fin de línea. Estas restricciones no son capturadas por el tipo renombrado, sino que tendremos que imponerlas por medio de los programas que crean líneas y palabras.

5.5.1 Textos como líneas

Consideremos primero el problema de convertir un texto en una lista de líneas. Estamos interesados en construir una función *lines* cuyo tipo es

```
lines :: Texto → [Linea]
```


Por ejemplo, deseamos tener

```
? lines "Esto es un⏎texto⏎"
[ "Esto es un", "texto", ""]

? lines "Esto es un⏎⏎␣␣␣⏎texto⏎"
[ "Esto es un", "", "␣␣␣", "texto", ""].

? lines "Esto es un texto"
[ "Esto es un texto"]
```

Tal como ilustran estos ejemplos, constituye una línea cualquier secuencia de caracteres entre dos caracteres fin de línea sucesivos, la secuencia de caracteres desde el principio del texto hasta el primer fin de línea (caso de haber alguno), y la secuencia de caracteres desde el último fin de línea hasta el final del texto. Nótese que la secuencia de caracteres después del último fin de línea podría ser vacía. La decisión de partir un texto de esta manera refleja la perspectiva de que un carácter fin de línea es un carácter *separador* entre líneas, en lugar de un carácter *terminador* que señala el final de una línea. En particular, ello implica que el número de líneas de un texto es siempre uno más que el número de caracteres de fin de línea. Un texto vacío consiste, por tanto, en una sola línea vacía.

Se puede especificar la función *lines* como la inversa de una segunda función *unlines*, definida mediante

```
unlines :: [Linea] → Texto
unlines = foldr1 (⊕)
          where xs ⊕ ys = xs ++ "⏎" ++ ys
```

El operador ⊕ es asociativo pero no tiene elemento neutro; por tanto, el valor de *unlines* [ ] no está definido. Entonces, es adecuada la utilización de *foldr*1. Hemos empleado ya la función *unlines* en las Secciones 5.2 y 5.4.

La relación entre *lines* y *unlines* es

$$unlines\ (lines\ xs) \quad = \quad xs$$

para todas las secuencias de caracteres *xs*. Se tiene también que

$$lines\ (unlines\ xss) \quad = \quad xss$$

para todas las listas de listas de caracteres *xss* que no contengan caracteres de fin de línea. Usando la primera ecuación, se puede sintetizar la siguiente implementación de *lines*:

```
lines    :: Texto → [Linea]
lines xs = if null zs then [ys] else ys : lines (tail zs)
           where (ys, zs) = span (≠ '⏎') xs
```

Esta función parte un texto *xs* en *ys* (el segmento inicial más largo posible de *xs* que no contenga un carácter de fin de línea), seguido de *zs* (el resto de *xs*, en caso de haberlo). Si *zs* no está vacía, entonces su primer carácter es un fin de línea, de modo que el proceso de construir líneas adicionales se efectúa sobre *tail zs*. Los detalles de esta

síntesis se dejan como ejercicio. Veremos otras aplicaciones de definiciones similares, de forma que merece la pena generalizar el esquema anterior de cálculo.

Considérense las funciones *partirSi* y *unirCon*, definidas como sigue:

```
partirSi      :: (Eq α) ⇒ α → [α] → [[α]]
partirSi x xs  = if null zs then [xs] else ys : partirSi x (tail zs)
                 where (ys, zs) = span (≠ x) xs

unirCon       :: α → [[α]] → [α]
unirCon x      = foldr1 (⊕)
                 where xs ⊕ ys = xs ++ [x] ++ ys
```

La relación entre estas dos funciones es tal que

```
unirCon x (partirSi x xs)  =  xs
```

para todas las listas *xs*. En particular,

```
lines    = partirSi '↴'
unlines  = unirCon '↴'
```

5.5.2 Líneas como palabras

Por definición, una *palabra* es una secuencia no vacía de caracteres visibles. En una línea similar a la anterior, podemos buscar una definición de una función *words* :: *Linea* → [*Palabra*] que convierta una línea en la lista de sus palabras. La función compañera *unwords* se define mediante

```
unwords  :: [Palabra] → Linea
unwords   = unirCon '␣'
```

Esta función toma una lista no vacía de palabras, pone un espacio en blanco entre cada dos palabras, y concatena el resultado. Aunque es tentador definir *words* = *partirSi* '␣', esta idea no es correcta: a diferencia de una línea, una palabra es una secuencia *no vacía* de caracteres. Dicho de otro modo, si dos caracteres de fin de línea son adyacentes, entonces hay una línea vacía entre ellos; pero si hay dos caracteres en blanco adyacentes, entonces no hay una palabra entre ellos. En vez de lo anterior, definimos

```
words  :: Linea → [Palabra]
words   = filter (not · null) · partirSi '␣'
```

Esta definición filtra las palabras vacías.

Como aplicación de las funciones *lines* y *words*, podemos contar el número de palabras de un texto mediante

```
contarPalabras  :: Texto → Int
contarPalabras   = length · concat · map words · lines
```

Esta función parte un texto en líneas, a continuación parte cada línea en palabras, luego concatena el resultado para obtener todas las palabras del texto, y finalmente calcula cuántas hay.

Como otra aplicación, podemos dar un nuevo formato a un texto mediante


```
formato1  :: Texto → Texto
formato1   = unlines · map formatoLinea · lines

formatoLinea  :: Linea → Linea
formatoLinea   = reconstruir · words
                 where reconstruir ws = if null ws then [ ] else unwords ws
```

La función *formatoLinea* descompone una línea en la lista de sus palabras. Si esta lista está vacía, entonces devuelve una línea vacía; en otro caso, da formato a la línea eliminando los espacios extra redundantes entre palabras. La función *formato1* aplica esta operación a cada línea de un texto. Por tanto, reemplaza líneas que contengan solamente caracteres en blanco por líneas vacías. Más adelante veremos una versión más sofisticada, *formato2*, de esta función.

5.5.3 Caracteres de tabulación

Algunos textos utilizan caracteres de tabulación. Para mostrar un texto que contenga caracteres de tabulación, es necesario reemplazar estos por el número correcto de espacios en blanco para llegar al siguiente tabulador. Para simplificar, supondremos que existen tabuladores a una distancia fija de *t* columnas para una cierta constante *t*.

La función *elimTab*, que elimina los tabuladores, trabaja línea a línea, de modo que podemos definir

```
elimTab  :: Texto → Texto
elimTab   = unlines · map elimTabLinea · lines
```

La función *elimTabLinea* elimina los tabuladores de una sola línea. Un modo de implementarla consiste en dividir la línea en *campos*, es decir, en listas de caracteres que no contengan tabuladores. Si introducimos el tipo renombrado

```
type Campo  =  [Char]
```

se tiene

```
fields  :: Linea → [Campo]
fields   = partirSi '⊣'
```

Ahora podemos definir

```
elimTabLinea  :: Linea → Linea
elimTabLinea   = unfields · fields
```

A diferencia de las funciones *unlines* y *unwords*, la función *unfields* no se define mediante *unirCon* '⊣' porque queremos reemplazar los tabuladores por espacios. Dado que el número de espacios a los que se expande cada tabulador depende del número de caracteres a su izquierda, hemos de reconstruir la línea de izquierda a derecha. Por tanto, definimos

```
unfields  :: [Campo] → Linea
unfields   = foldl1 (⊕)
             where xs ⊕ ys = xs ++ espacios (t − (length xs) mod t ) ++ ys
```

El término $xs \oplus ys$ añade suficientes espacios detrás de xs como para alcanzar el siguiente tabulador, que es el siguiente múltiplo de t.

El problema con el programa para *unfields* es que es bastante ineficiente: para cada campo recalcula la longitud de la línea hasta el siguiente tabulador. Una implementación mejor consiste en usar una estrategia de *tuplamiento*, especificando una función *expandir* mediante

```
expandir     :: [Campo] → (Linea, Int)
expandir fs  =  (unfields fs, length (unfields fs))
```

En el Capítulo 7 se considerará la estrategia de tuplamiento para mejorar la eficiencia de los programas. En todo caso, el siguiente programa para *expandir* es bastante fácil de sintetizar. Se dejan los detalles como ejercicio:

```
expandir
  = foldl1 (⊕) · map ponerLongitud
    where ponerLongitud xs   =  (xs, length xs)
          (xs, m) ⊕ (ys, n)  =  (xs ++ espacios k ++ ys, m + k + n)
                                where k = t − m mod t
```

Ahora podemos definir $unfields = fst \cdot expandir$.

5.5.4 Párrafos

Se pueden considerar ciertos textos como listas de párrafos. Tales textos constan de palabras consecutivas, sin fórmulas matemáticas, ni tablas, poemas, programas, etc. Desde el punto de vista lógico, un párrafo es simplemente una lista no vacía de palabras; por tanto, declaramos

```
type Parrafo  =  [Palabra]
```

La función *paras* convierte un texto en la lista de sus párrafos:

```
paras  ::  Texto → [Parrafo]
paras  =   map concat · filter (not · null) ·
           partirSi [ ] · map words · lines
```

Esta función toma un texto, lo convierte en una lista de líneas y después convierte cada línea en una lista de palabras. El final de un párrafo viene señalado por la presencia de una línea que no contiene palabras. Se eliminan entonces los párrafos vacíos, y los restantes se aplanan a listas de palabras, descartando la información acerca de qué palabras van en que líneas.

El proceso inverso, esto es, la reconstrucción de la estructura de líneas de un párrafo, se conoce como *llenado*. Llenar un párrafo consiste en disponer las palabras del párrafo en una secuencia de líneas no vacías de modo que, en primer lugar, la longitud de la línea no exceda de un máximo fijo y, en segundo lugar, la secuencia completa tenga una apariencia estética agradable. Nótese especialmente la diferencia esencial entre llenar y dar formato: el llenado indica qué palabras van en qué líneas, mientras que el formato indica cómo se deben mostrar en la pantalla.

Un modo de formalizar la noción de párrafo agradable consiste en minimizar una cierta noción de *desperdicio*. Por ejemplo, podríamos querer que el número de líneas

que han de ser llenadas en el párrafo sea lo más pequeño posible. Un algoritmo simple para llenar un párrafo es voraz por naturaleza: en cada etapa seleccionamos el segmento inicial más largo posible de las palabras que quedan por introducir, de forma que quepan en el límite máximo fijado para la línea. Para que este algoritmo funcione, de hecho para que el problema del llenado tenga solución, es necesario suponer que el límite máximo de la línea es lo suficientemente grande como para acomodar cualquier palabra. Se puede demostrar que el algoritmo voraz devuelve la lista de líneas más pequeña posible.

Supongamos que nos dan una función *llenarParrafo* cuyo tipo es

```
llenarParrafo  ::  Int → Parrafo → [[Palabra]]
```

La función *llenarParrafo n* toma una lista no vacía de palabras y la estructura en una lista no vacía de listas no vacías de palabras. Cada una de las listas de palabras constituirá una línea en el texto con formato. Daremos una implementación de *llenarParrafo* más adelante. Podemos ahora dar formato a un texto mediante una función *formato2*, definida mediante

```
formato2     ::  Int → Texto → Texto
formato2 n   =   unparas · map (llenarParrafo n) · paras
```

Esta función parte un texto en párrafos y después lo convierte en una cadena de caracteres. Se puede definir la función *unparas* mediante

```
unparas  ::  [[[Palabra]]] → Texto
unparas  =   unlines · unirCon [ ] · map (map unwords)
```

Esta función convierte en líneas las palabras de cada línea de cada párrafo, une las líneas con líneas vacías entre párrafos y transforma el resultado de nuevo en un texto, insertando caracteres de fin de línea.

Queda por tratar la función *llenarParrafo*; por tanto, concluimos la sección implementando el algoritmo voraz para llenar un párrafo. El programa es

```
llenarParrafo        ::  Int → Parrafo → [[Palabra]]
llenarParrafo n ws   =   if null xs then [ ] else xs : llenarParrafo n ys
                         where (xs, ys) = splitAt (voraz n) ws
```

Esta función parte una lista ws de palabras en, por un lado, el segmento inicial más largo posible que entra en una línea de anchura máxima n y, por otro, las restantes palabras, caso de haber alguna. Este proceso se aplica recursivamente sobre las restantes palabras.

La función *voraz* se define mediante

```
voraz     ::  Int → [Palabra] → Int
voraz n   =   length · last · filter ((≤ n) · length · unwords) inits1
```

La definición de *voraz n ws* se puede leer: calcular la longitud del mayor segmento inicial no vacío xs de la cadena ws que satisface $length\ (unwords\ xs) \leq n$. La función *inits1* devuelve la lista de segmentos iniciales no vacíos de una lista por orden de longitud creciente; de aquí que el último elemento en cualquier subsecuencia de *inits1 ws* es el elemento más largo de la subsecuencia. Recordemos que *unwords* no está definido sobre listas vacías, razón por la cual sólo generamos segmentos iniciales no

vacíos. Recordemos también nuestra suposición de que el límite n de la línea es lo suficientemente largo como para acomodar cualquier palabra; por tanto, filtrando los segmentos iniciales de una secuencia no vacía de palabras se produce una subsecuencia no vacía.

Aunque esto resuelve el problema, el programa de *voraz* es ineficiente porque el valor de *unwords* se recalcula para cada segmento inicial no vacío de la lista de palabras dada. De hecho, *voraz n* necesita tiempo cuadrático en el tamaño del resultado. Concluiremos derivando una versión más eficiente. Como primer paso, podemos reemplazar *filter* por *takeWhile* en la definición de *voraz*. La razón es que si el segmento inicial *xs* no pasa la comprobación, entonces se verifica que $length\ (unwords\ xs) > n$ y, por tanto, tampoco pasa la comprobación cualquier segmento inicial más largo. Podemos combinar esta sustitución con la propiedad

$$length \cdot last \cdot takeWhile\ p \cdot inits1 \quad = \quad length \cdot takeWhile\ p \cdot inits1$$

Es fácil ver por qué es válida esta ley: si $takeWhile\ p\ (inits1\ xs)$ devuelve la lista

$$[[x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]]$$

entonces la longitud del último elemento, a saber n, es la longitud de toda la lista. Al aplicar esta ley se obtiene la siguiente expresión para *voraz n*:

$$length \cdot takeWhile\ ((\leq n) \cdot length \cdot unwords) \cdot inits1$$

La siguiente ley que queremos utilizar es

$$map\ f \cdot takeWhile\ (p \cdot f) \quad = \quad takeWhile\ p \cdot map\ f$$

El objetivo es aplicar esta ley a la expresión *voraz n*, siendo $p = (\leq n)$ y $f = length \cdot unwords$. Para hacer esto, necesitaremos el siguiente hecho

$$length \quad = \quad length \cdot map\ f$$

para cualquier f. Aplicando primero esta ley y después la anterior, obtenemos una nueva expresión para *voraz n*:

$$length \cdot takeWhile\ (\leq n) \cdot map\ (length \cdot unwords) \cdot inits1$$

Para el paso siguiente, recordemos que $unwords = foldr1\ (\oplus)$, donde $xs \oplus ys = xs \mathbin{++} \text{"␣"} \mathbin{++} ys$. Una aplicación de la ley de fusión para *foldr*1 produce

$$\begin{aligned} length \cdot unwords \quad &= \quad foldr1\ (\otimes) \cdot map\ length \\ &\quad\ \ \text{where } m \otimes n = m + 1 + n \end{aligned}$$

Aplicando este resultado, obtenemos la siguiente expresión para *voraz n*:

$$length \cdot takeWhile\ (\leq n) \cdot map\ (foldr1\ (\otimes) \cdot map\ length) \cdot inits1$$

El siguiente paso consiste en usar la propiedad de functor de *map* para obtener una nueva expresión

$$length \cdot takeWhile\ (\leq n) \cdot map\ (foldr1\ (\otimes)) \cdot map\ (map\ length) \cdot inits1$$

Seguidamente hacemos uso de la condición de naturalidad

$$map\ (map\ f) \cdot inits1 \quad = \quad inits1 \cdot map\ f$$

para obtener una simplificación más:

$$length \cdot takeWhile\ (\leq n) \cdot map\ (foldr1\ (\otimes)) \cdot inits1 \cdot map\ length$$

Para el paso final, nos gustaría aplicar la siguiente propiedad de *scanl*1:

$$scanl1\ (\otimes) \quad = \quad map\ (foldl1\ (\otimes)) \cdot inits1$$

Pero parece que nos hemos atascado en el último paso: la expresión anterior para *voraz n* contiene *foldr*1 (⊗), no *foldl*1 (⊗). Afortunadamente, el primer teorema de dualidad viene en nuestra ayuda: ⊗ es un operador asociativo, como es fácil de comprobar. Más aún, los segmentos iniciales de una lista son todos finitos. Por tanto, podemos reemplazar *foldr*1 (⊗) por *foldl*1 (⊗) en la expresión.

El resultado final es que podemos calcular *voraz n* mediante el programa

$$voraz\ n \quad = \quad length \cdot takeWhile\ (\leq n) \cdot scanl1\ (\otimes) \cdot map\ length$$

Esta versión necesita tiempo lineal en la longitud del resultado.

Ejercicios

5.5.1 Demostrar que $unlines \cdot lines = id$. ¿Por qué no se cumple que $lines \cdot unlines = id$?

5.5.2 Describir las líneas para las cuales $unwords \cdot words$ devuelve ⊥.

5.5.3 Supongamos que adoptamos el convenio de que un carácter fin de línea es un carácter terminador, en lugar de un separador. En este caso, el número de líneas del texto es exactamente el número de caracteres de fin de línea. Podemos entonces definir *unlines* mediante

$$unlines \quad = \quad concat \cdot map\ (\mathbin{++} \text{"⏎"})$$

Usar la ley de fusión *fold-map* para reescribir la definición de *unlines* como una concreción de *foldr*. ¿Cuál es la correspondiente definición de *lines*?

5.5.4 Sintetizar el programa eficiente de *expandir*.

5.5.5 Definir una versión de *elimTabLinea* que no comience partiendo una línea en campos.

5.5.6 Un método para comprimir textos es reducir el número de espacios y de tabuladores al mínimo requerido para conseguir el mismo espaciado. Los espacios finales y los tabuladores se pueden eliminar del final de una línea sin afectar a su apariencia visible. Definir una función *insertarTab* para insertar tabuladores.

5.5.7 ¿Por qué un texto que contenga una mezcla de prosa y de poesía no puede ser considerado una lista de párrafos?

5.5.8 En el texto, los párrafos adquieren formato ajustando las líneas a la izquierda. Desarrollar un algoritmo para dar formato a las líneas de forma que, exceptuando la última línea de un párrafo, éstas se extiendan ajustándose tanto al margen izquierdo como al margen derecho.

Capítulo 6

Árboles

Se llama *árbol* a cualquier tipo de datos recursivo cuya estructura no sea lineal. Los árboles se utilizan como representaciones naturales de cualquier tipo de datos organizado jerárquicamente. También se puede modelizar mediante árboles la estructura sintáctica de las expresiones aritméticas o funcionales. Quizá la propiedad más útil es que los árboles constituyen una generalización de las listas que permite organizar y buscar información eficientemente.

Hay numerosas especies y subespecies de árboles. Los árboles se pueden clasificar según la forma precisa de la estructura de ramificación, la localización de la información dentro del árbol, y la relación entre la información almacenada en diferentes partes del mismo. Este capítulo introducirá algunas de las especies más comunes, describirá un poco de la terminología básica asociada a los árboles, y dará algunos ejemplos de aplicación. En el Capítulo 8 se ofrecerán más aplicaciones.

6.1 Árboles binarios

Como su nombre indica, un árbol binario es un árbol con una estructura de ramificación en dos ramas. Existen dos o tres subespecies diferentes de árboles binarios, pero comenzaremos analizando el siguiente tipo de datos:

$$\textbf{data}\ \mathit{ArBin}\ \alpha = \mathit{Hoja}\ \alpha \mid \mathit{Unir}\ (\mathit{ArBin}\ \alpha)\ (\mathit{ArBin}\ \alpha)$$

Un valor de *ArBin* α es, o bien un *nodo hoja*, que contiene un valor de tipo α, o un *nodo interno* que consiste en dos subárboles llamados subárbol *izquierdo* y *derecho* de ese nodo. A veces, a un nodo hoja se le llama nodo *externo* y a un nodo interno se le llama nodo de *bifurcación*.

Por ejemplo, el árbol

$$\mathit{Unir}\ (\mathit{Hoja}\ 1)\ (\mathit{Unir}\ (\mathit{Hoja}\ 2)\ (\mathit{Hoja}\ 3))$$

consta de un nodo con un subárbol izquierdo *Hoja* 1 y un subárbol derecho que a su vez consiste en un subárbol izquierdo *Hoja* 2 y un subárbol *Hoja* 3. Compárese este árbol con este otro:

$$\mathit{Unir}\ (\mathit{Unir}\ (\mathit{Hoja}\ 1)\ (\mathit{Hoja}\ 2))\ (\mathit{Hoja}\ 3)$$

Aunque el segundo árbol contiene en sus hojas la misma secuencia de números que el primero, el modo en que está organizada la información es bastante diferente y las dos expresiones denotan valores distintos. En esencia, los diferentes elementos de *ArBin* α expresan diversos modos de parentizar secuencias de valores. Por ejemplo, 1 (2 3) y (1 2) 3 son los dos modos de parentizar 1 2 3. Hay cinco modos de parentizar cuatro valores, a saber:

1 (2 (3 4)) 1 ((2 3) 4) (1 2) (3 4) (1 (2 3)) 4 ((1 2) 3) 4

Por tanto, hay cinco árboles binarios diferentes con esta secuencia de valores en sus hojas. Dado que hemos empleado los nombres *xs*, *ys*, ... para denotar las variables de tipo lista, usaremos *xt*, *yt*, ... para denotar las variables de tipo árbol.

6.1.1 Inducción sobre árboles

El principio de inducción estructural que hemos visto para números naturales y para listas se puede extender fácilmente a los árboles binarios: para demostrar que una propiedad $P(xt)$ se cumple para todos los árboles binarios *xt* es suficiente probar:

Caso ($\perp$). Que se cumple $P(\perp)$;

Caso (*Hoja x*). Que se cumple *P* (*Hoja x*) para todo *x*;

Caso (*Unir xt yt*). Que si se cumplen tanto $P(xt)$ como $P(yt)$, entonces también se cumple *P*(*Unir xt yt*).

Si sólo queremos demostrar que $P(xt)$ se cumple para todos los árboles finitos, entonces es suficiente probar los dos últimos casos. Por definición, un árbol finito es aquel que tiene un número finito de hojas. Más adelante se da un ejemplo de demostración por inducción.

6.1.2 Tamaño y altura

Existen dos medidas numéricas importantes sobre árboles binarios: el tamaño y la altura. El *tamaño* de un árbol es el número de sus hojas:

```
tam              :: ArBin α → Int
tam (Hoja x)     =  1
tam (Unir xt yt) =  tam xt + tam yt
```

La función *tam* juega el mismo papel para los árboles que el que desempeña *length* para las listas. De hecho, *tam* = *length* · *aplanar*, siendo

```
aplanar              :: ArBin α → [α]
aplanar (Hoja x)     =  [x]
aplanar (Unir xt yt) =  aplanar xt ++ aplanar yt
```

La función *aplanar* produce la secuencia de valores en las hojas al inspeccionar éstas de izquierda a derecha. Un árbol es finito si y sólo si tiene un tamaño bien definido.

Existe una relación simple pero importante entre el número de nodos exteriores e interiores de un árbol binario finito: los primeros son siempre uno más que los segundos. Supongamos que contamos el número de nodos internos mediante la función *numNodos*, siendo

```
numNodos             :: ArBin α → Int
numNodos (Hoja x)    =  0
numNodos (Unir xt yt) =  1 + numNodos xt + numNodos yt
```


Entonces *tam xt* = 1 + *numNodos xt* para todos los árboles finitos *xt*. Este resultado se demuestra por inducción estructural sobre árboles; los detalles de la demostración se dejan como ejercicio.

La segunda medida útil sobre árboles es la *altura* de un árbol. La altura de un árbol mide la distancia a la que está la hoja más lejana:

```
altura              :: ArBin α → Int
altura (Hoja x)     =  0
altura (Unir xt yt) =  1 + (altura xt max altura yt)
```

Por ejemplo, la altura del árbol

Unir (*Hoja* 1) (*Unir* (*Hoja* 2) (*Hoja* 3))

es 2. La altura de un árbol *xt* es la máxima de las *profundidades* de las hojas de *xt*. En el árbol anterior, la profundidad de (*Hoja* 1) es 1, mientras que, en ambos casos, la profundidad de (*Hoja* 2) y (*Hoja* 3) es 2. La función *ponerProfs* toma un árbol y reemplaza el valor almacenado en cada hoja por la profundidad de esa hoja en el árbol:

```
ponerProfs          :: ArBin α → ArBin Int
ponerProfs          =  ponerProfs' 0

ponerProfs'         :: Int → ArBin α → ArBin Int
ponerProfs' n (Hoja x)     =  Hoja n
ponerProfs' n (Unir xt yt) =  Unir (ponerProfs' (n + 1) xt) (ponerProfs' (n + 1) yt)
```

Si definimos *maxArBin* mediante

```
maxArBin            :: (Ord α) ⇒ ArBin α → α
maxArBin (Hoja x)   =  x
maxArBin (Unir xt yt) =  (maxArBin xt) max (maxArBin yt)
```

entonces *altura* = *maxArBin* · *ponerProfs*. Para su demostración, véase el Ejercicio 6.1.6.

Se dice que un árbol binario es *perfecto* si todas sus hojas se encuentran a la misma profundidad. Por ejemplo, el árbol

Unir (*Unir* (*Hoja* 1) (*Hoja* 2)) (*Unir* (*Hoja* 3) (*Hoja* 4))

es un árbol binario perfecto. El tamaño de un árbol binario perfecto es siempre una potencia de dos y, prescindiendo de los valores almacenados en las hojas, existe exactamente un árbol perfecto para cada potencia de dos.

Aunque dos árboles del mismo tamaño no tienen por qué tener la misma altura, las dos magnitudes no son independientes. El siguiente resultado es uno de los más importantes sobre árboles binarios. Dice que

altura xt < *tam xt* ≤ 2 ↑ *altura xt*

para todos los árboles binarios *xt*. De forma equivalente, tomando logaritmos en base dos, tenemos

$$\lceil (tam\ xt) \rceil \leq altura\ xt < tam\ xt$$

donde $\lceil x \rceil$ (el *techo* de *x*) denota el menor entero que es mayor o igual que *x*. La demostración de esta relación fundamental se realiza por inducción estructural. Demostramos una de las desigualdades y dejamos la otra como ejercicio.

Demostración. Demostraremos que $tam\ xt \leq 2 \uparrow altura\ xt$ por inducción sobre *xt*.

Caso (*Hoja x*). En una línea, tenemos

$$tam\ (Hoja\ x) = 1 = 2 \uparrow 0 = 2 \uparrow altura\ (Hoja\ x)$$

Caso (*Unir xt yt*). Razonamos del modo siguiente:

$$\begin{array}{ll}
 & tam\ (Unir\ xt\ yt) \\
= & \quad \{\text{definición de } tam\} \\
 & tam\ xt + tam\ yt \\
\leq & \quad \{\text{hipótesis de inducción}\} \\
 & 2 \uparrow (altura\ xt) + 2 \uparrow (altura\ yt) \\
\leq & \quad \{\text{aritmética, donde } h = (altura\ xt\ \mathbf{max}\ altura\ yt)\} \\
 & 2 \uparrow h + 2 \uparrow h \\
= & \quad \{\text{aritmética}\} \\
 & 2 \uparrow (1 + h) \\
= & \quad \{\text{definición de } h \text{ y de } altura\} \\
 & 2 \uparrow (altura\ (Unir\ xt\ yt))
\end{array}$$

□

Dada cualquier lista *xs* de longitud positiva *n*, se puede construir un árbol *xt* que cumpla

$$aplanar\ xt = xt \quad \text{y} \quad altura\ xt = \lceil\ log\ n\ \rceil$$

Tal árbol tiene la altura más pequeña posible. En general, para una secuencia dada de valores de sus hojas, existirá más de un árbol con una altura mínima. Los árboles de altura mínima son útiles porque aseguran que el coste de encontrar una hoja es el menor posible, incluso en el peor caso.

Se puede obtener un árbol de altura mínima dividiendo la entrada (no vacía) en dos partes iguales y construyendo recursivamente un árbol de altura mínima para cada mitad:

```
construirArBin :: [α] → ArBin α
construirArBin xs
  | (m == 0)      =  Hoja (desenlistar xs)
  | otherwise     =  Unir (construirArBin ys) (construirArBin zs)
  where m         =  (lenght xs) div 2
        (ys, zs)  =  splitAt m xs
```


Recordemos de la Sección 4.2 que *splitAt n xs* = (*take n xs, drop n xs*). La función *desenlistar* se define mediante *desenlistar* [*x*] = *x*. En el capítulo siguiente analizaremos el tiempo de ejecución de este programa, mostrando que *construirArBin* necesita un tiempo proporcional a *n* log *n* pasos para una lista de longitud *n*. Como también veremos en el capítulo siguiente, se puede dar una implementación de *construirArBin* que se ejecute en tiempo lineal.

6.1.3 Las funciones *map* y *fold*

Las funciones *map* y *fold* para listas tienen sus análogas *mapArBin* y *foldArBin* para árboles. Definimos la función *mapArBin* mediante

```
mapArBin                 ::  (α → β) → ArBin α → ArBin β
mapArBin f (Hoja x)      =   Hoja (f x)
mapArBin f (Unir xt yt)  =   Unir (mapArBin f xt) (mapArBin f yt)
```

la función *mapArBin* satisface leyes similares a las de *map*. En particular,

```
mapArBin id      =  id
mapArBin (f · g) =  mapArBin f · mapArBin g
```

Por lo tanto, *mapArBin*, al igual que *map*, es un funtor. Se tiene también

```
map f · aplanar  =  aplanar · mapArBin f
```

La función *foldArBin* ha de proporcionar una función de reemplazamiento para cada una de las constructoras de *ArBin α*. Las dos constructoras tienen tipos

```
Hoja  ::  α → ArBin α
Unir  ::  ArBin α → ArBin α → ArBin α
```

Por tanto, *foldArBin* ha de proporcionar dos funciones de reemplazamiento, *f* y *g*, cuyos tipos sean

```
f  ::  α → β
g  ::  β → β → β
```

La definición de *foldArBin* es

```
foldArBin                ::  (α → β) → (β → β → β) → ArBin α → β
foldArBin f g (Hoja x)   =   f x
foldArBin f g (Unir xt yt) =  g (foldArBin f g xt) (foldArBin f g yt)
```

Muchas operaciones sobre árboles se pueden definir en términos de *foldArBin*. Como muestra, tenemos

```
tam     =  foldArBin (const 1) (+)

altura  =  foldArBin (const 0) (⊕)
           where m ⊕ n = 1 + (m max n)

aplanar =  foldArBin enlistar (++)

maxArBin = foldArBin id (max)
```

$$mapArBin\ f \quad = \quad foldArBin\ (Hoja \cdot f)\ Unir$$

En los primeros dos ejemplos, la función estándar *const* se define mediante

$$\begin{array}{lcl} const & :: & \alpha \to \beta \to \alpha \\ const\ x\ y & = & x \end{array}$$

Se define la función *enlistar* mediante $enlistar\ x = [x]$.

6.1.4 Árboles binarios aumentados

Los árboles binarios introducidos hasta ahora almacenan información exclusivamente en las hojas. Una ligera, pero importante, variación de esta idea básica consiste en permitir que la información se almacene también en los nodos internos. El etiquetado de los nodos internos de un árbol binario tiene importantes aplicaciones en el diseño de algoritmos eficientes sobre árboles. Daremos a continuación un ejemplo de ello. Se darán más en el resto de este capítulo y en el Capítulo 8.

Supongamos que aumentamos los árboles binarios con información relativa al tamaño. Más precisamente, consideremos el tipo de datos

$$data\ ArbAum\ \alpha \quad = \quad Hoja\ \alpha \mid Unir\ Int\ (ArbAum\ \alpha)\ (ArbAum\ \alpha)$$

Para simplificar, utilizamos los mismos nombres de constructoras que empleamos para *ArBin*; no obstante, se tendrían que usar nombres diferentes en cualquier guión que usara ambos tipos de datos. En el tipo de datos *ArbAum* α, los nodos internos se etiquetan con un entero. La idea es que en la expresión *Unir n xt yt*, el entero *n* satisfaga la condición $n = tam\ xt$, donde *tam* se define del mismo modo que para *ArBin* α. Por supuesto, hemos de asegurar que se satisface esta condición mientras construimos el árbol. Haremos esto construyendo los nodos internos utilizando exclusivamente la función *unir* definida mediante

$$\begin{array}{lcl} unir & :: & ArbAum\ \alpha \to ArbAum\ \alpha \to ArbAum\ \alpha \\ unir\ xt\ yt & = & Unir\ n\ xt\ yt \\ & & \quad \textbf{where}\ n = tamA\ xt + tamA\ yt \end{array}$$

$$\begin{array}{lcl} tamA & :: & ArbAum\ \alpha \to Int \\ tamA\ (Hoja\ x) & = & 1 \\ tamA\ (Unir\ n\ xt\ yt) & = & n \end{array}$$

Por ejemplo, se puede modificar la función *construirArBin* para definir una función *construirArbAum* que construya, para una secuencia dada de valores, un árbol aumentado de altura mínima:

$$\begin{array}{l} construirArbAum :: [\alpha] \to ArbAum\ \alpha \\ construirArbAum\ xs \\ \quad \mid (m == 0) \quad = \quad Hoja\ (desenlistar\ xs) \\ \quad \mid otherwise \quad = \quad unir\ (construirArBin\ ys)\ (construirArBin\ zs) \\ \quad \textbf{where}\ m \quad = \quad (length\ xs)\ \textbf{div}\ 2 \\ \qquad\quad (ys, zs) \quad = \quad splitAt\ m\ xs \end{array}$$

Supongamos ahora que queremos implementar una función *buscar*, especificada mediante

$$\begin{array}{lcl} buscar & :: & ArbAum\ \alpha \to Int \to \alpha \\ buscar\ xt\ k & = & (aplanar\ xt)\ !!\ k \end{array}$$

Dicho con palabras, *buscar* indexa un árbol del mismo modo que (!!) indexa una lista. Podemos utilizar el hecho de que

$$\begin{array}{lcl} (xs +\!\!+ ys)\ !!\ k & = & \textbf{if}\ k < m\ \textbf{then}\ xs\ !!\ k\ \textbf{else}\ ys\ !!\ (k - m) \\ & & \quad \textbf{where}\ m = length\ xs \end{array}$$

para sintetizar el siguiente programa más eficiente para *buscar*:

$$\begin{array}{lcl} buscar\ (Hoja\ x)\ 0 & = & x \\ buscar\ (Unir\ m\ xt\ yt)\ k & = & \textbf{if}\ k < m\ \textbf{then}\ buscar\ xt\ k\ \textbf{else}\ buscar\ yt\ (k - m) \end{array}$$

Esta implementación de *buscar* utiliza la información sobre el tamaño almacenada en los nodos binarios para controlar la búsqueda de un elemento en una posición dada. Lo que hemos ganado con esta acción es una alternativa para indexar una lista potencialmente más eficiente que la habitual. En lugar de almacenar información en una lista y utilizar (!!) para obtener los valores, podemos convertir la lista en un árbol binario de altura mínima mediante *construirArbAum* y después utilizar *buscar*. Es claro que *buscar* necesita un número de pasos proporcional a *log n* siendo *n* la longitud de la lista, en tanto que (!!) puede necesitar un número de pasos proporcional a *n*. El primero es mucho más eficiente, siempre que el número de búsquedas sea lo suficientemente grande como para compensar el coste de calcular *construirArbAum*.

Ejercicios

6.1.1 ¿Cuántos árboles binarios de tamaño cinco existen? Escribir un programa que calcule, para un *n* dado, el número de árboles binarios de tamaño *n*.

6.1.2 Demostrar que el número de hojas de un árbol binario es siempre uno más que el número de nodos internos.

6.1.3 Se pueden definir los subárboles de un árbol binario mediante

$$\begin{array}{lcl} subArBin & :: & ArBin\ \alpha \to [ArBin\ \alpha] \\ subArBin\ (Hoja\ x) & = & [Hoja\ x] \\ subArBin\ (Unir\ xt\ yt) & = & [Unir\ xt\ yt] +\!\!+ subArBin\ xt +\!\!+ subArBin\ yt \end{array}$$

Establecer y demostrar una relación entre $length\ (subArBin\ xt)$ y $tam\ xt$.

6.1.4 Demostrar que $altura\ xt < tam\ xt$.

6.1.5 Demostrar que se puede convertir una lista *xs* de longitud 2^n en un árbol de altura mínima *xt* que satisfaga $aplanar\ xt = xs$ exactamente de una manera.

6.1.6 Demostrar que $altura = maxArBin \cdot ponerProfs$ demostrando primero el resultado más general

$$(n +) \cdot altura \quad = \quad maxArBin \cdot ponerProfs'\ n$$

6.2 Árboles binarios de búsqueda

Además de satisfacer una condición extra descrita más adelante, un *árbol binario de búsqueda* es un elemento del tipo de datos

```
data (Ord α) ⇒ ArBus α  =  Vacio | Unir (ArBus α) α (ArBus α)
```

Éste es el primer ejemplo de una declaración de tipos que utiliza un *contexto*: se introduce el tipo *ArBus α* exclusivamente para tipos que sean concreciones de la clase *Ord*. Los elementos de *ArBus α* no tienen ya nodos *Hoja*, y la información almacenada en las etiquetas se ha desplazado, situándose entre los dos subárboles por una razón que explicaremos más adelante. La nueva constructora *Vacio* denota el árbol vacío. Este tipo de datos introduce una variedad de árboles en la que todo árbol, o bien es vacío, o bien consiste en un nodo con un subárbol izquierdo, una etiqueta y un subárbol derecho. A estos árboles se los denomina también *árboles binarios etiquetados*. La función *aplanar*, definida mediante

```
aplanar            ::  (Ord α) ⇒ ArBus α → [α]
aplanar Vacio      =   []
aplanar (Unir xt x yt)  =  aplanar xt ++ [x] ++ aplanar yt
```

devuelve la lista de las etiquetas de un árbol enumeradas de izquierda a derecha. La segunda ecuación de *aplanar* justifica la colocación de la etiqueta de cada árbol entre los dos subárboles.

Existe una condición adicional sobre los elementos de *ArBus α* que los convierte en árboles binarios de búsqueda. La condición es que *aplanar* debe devolver una lista en orden creciente. Ésta es la razón por la que aparece la clase *Ord*. En otras palabras, *xt* es un árbol binario de búsqueda si se cumple *ordenado xt* donde

```
ordenado  ::  (Ord α) ⇒ ArBus α → Bool
ordenado  =   ordenada · aplanar
```

Aquí, *ordenada xs* es la condición de que la lista *xs* esté en orden creciente.

Como su nombre sugiere, los árboles binarios de búsqueda se han diseñado para una búsqueda eficiente. La función *esta*, que determina si un valor aparece como etiqueta de algún nodo en el árbol binario de búsqueda, se puede definir mediante

```
esta :: α → ArBus α → Bool
esta x Vacio = False
esta x (Unir xt y yt)
  | (x < y)   =  esta x xt
  | (x == y)  =  True
  | (x > y)   =  esta x yt
```

En el caso peor, el coste de evaluar *esta xt* es proporcional a la altura de *xt*, donde la definición de altura se obtiene modificando ligeramente la definición anterior:

```
altura            ::  ArBus α → Integer
altura Vacio      =   0
altura (Unir xt x yt)  =  1 + (altura xt max altura yt)
```


El tamaño de un árbol binario de búsqueda es el número de etiquetas del árbol; y la relación fundamental entre tamaño y altura toma ahora el aspecto modificado

$$altura\ xt \leq tam\ xt < 2 \uparrow (altura\ xt)$$

para todos los árboles *xt*. De modo equivalente, un árbol binario de búsqueda de tamaño *n* tiene una altura *h* que satisface

$$\lceil log\ (n+1) \rceil \leq h < n+1$$

A partir de una secuencia de valores, se puede construir un árbol binario de búsqueda modificando la definición de *construirArBin* dada en la sección precedente:

```
construirArBus        ::  (Ord α) ⇒ [α] → ArBus α
construirArBus []     =   Vacio
construirArBus (x : xs)  =  Unir (construirArBus ys) x (construirArBus zs)
                            where (ys, zs) = partir (≤ x) xs
```

La función *partir* se especifica mediante

```
partir    ::  (α → Bool) → [α] → ([α], [α])
partir p xs  =  (filter p xs, filter (not · p) xs)
```

Se deja como ejercicio la implementación de *partir* como una concreción de *foldr*.

La diferencia entre *construirArBus* y *construirArBin* es que *construirArBus* no garantiza que el resultado tenga la menor altura posible. Por ejemplo, el valor *x* en la segunda ecuación puede ser más pequeño que cualquier elemento de *xs*; por tanto, *construirArBus* puede construir un árbol en el cual el subárbol izquierdo de cada nodo sea *Vacio*. Si bien es posible implementar la función *construirArBin* para que se ejecute en tiempo lineal, la implementación más eficiente de *construirArBus* necesita como mínimo *n* log *n* pasos para una lista de longitud *n*.

Se puede utilizar la nueva versión de *construirArBus* para implementar la función *ordenar* que ordena una lista:

```
ordenar  ::  (Ord α) ⇒ [α] → [α]
ordenar  =   aplanar · construirArBus
```

En el capítulo siguiente se estudiará este algoritmo, y veremos que, eliminando el árbol intermedio, se puede llegar a una versión del famoso algoritmo de ordenación llamado *quicksort*[1].

6.2.1 Inserción y supresión

Los árboles binarios de búsqueda son útiles para representar eficientemente conjuntos, un tema que se estudiará con más detalle en el Capítulo 8. Aquí daremos la versión más general de los algoritmos más avanzados, implementando funciones que añaden y eliminan etiquetas de un árbol.

La función *insertar* es fácil de definir:

```
insertar :: (Ord α) ⇒ α → ArBus α → ArBus α
insertar x Vacio = Unir Vacio x Vacio
```

[1] N. del T.: En español se conoce como algoritmo de *ordenación rápida*.

```
insertar x (Unir xt y yt)
  | (x < y)   =  Unir (insertar x xt) y yt
  | (x == y)  =  Unir xt y yt
  | (x > y)   =  Unir xt y (insertar x yt)
```

Nótese que si una etiqueta ya está en el árbol, entonces no se añaden copias adicionales.

La función *borrar* es ligeramente más complicada:

```
borrar :: (Ord α) ⇒ α → ArBus α → ArBus α
borrar x Vacio = Vacio
borrar x (Unir xt y yt)
  | (x < y)   =  Unir (borrar x xt) y yt
  | (x == y)  =  une xt yt
  | (x > y)   =  Unir xt y (borrar x yt)
```

La función auxiliar *une* ha de unir dos árboles de forma que satisfaga la ecuación

```
aplanar (une xt yt)  =  aplanar xt ++ aplanar yt
```

Una posible implementación consiste en reemplazar el subárbol vacío más a la derecha del primer árbol por el segundo árbol:

```
une                  ::  ArBus α → ArBus α → ArBus α
une Vacio yt         =   yt
une (Unir ut x vt) yt =  Unir ut x (une vt yt)
```

Esta definición no es satisfactoria porque da lugar a un árbol de mayor altura que la necesaria. La eficiencia de la consulta de pertenencia a un árbol y de la inserción depende de la altura del árbol que se está recorriendo, y nos gustaría asegurar que cuando unimos árboles mantenemos la altura tan pequeña como sea razonablemente posible.

Una segunda implementación consiste en tomar el valor en el extremo izquierdo del segundo árbol (si existe alguno) como etiqueta del árbol unido. En otras palabras, podemos explotar la identidad

```
xs ++ ys  =  xs ++ [head ys] ++ tail ys
```

Esta ecuación es válida siempre que *ys* no sea la lista vacía. Supongamos que se especifican las funciones *primeroArbol* y *restoArbol* mediante

```
primeroArbol  ::  ArBus α → α
primeroArbol  =   head · aplanar

restoArbol            ::  ArBus α → ArBus α
aplanar · restoArbol  =   tail · aplanar
```

Entonces podemos definir

```
une xt yt
  = if vacio yt then xt else Unir xt (primeroArbol yt) (restoArbol yt)
```


donde

```
vacio              ::  ArBus α → Bool
vacio Vacio        =   True
vacio (Unir xt x yt) =  False
```

Para comprobar esta definición de *une*, razonamos como sigue:

```
    aplanar (une xt yt)
=       {definición de une, suponiendo not (vacio yt)}
    aplanar (Unir xt (primeroArbol yt) (restoArbol yt))
=       {definición de aplanar}
    aplanar xt ++ [primeroArbol yt] ++ aplanar (restoArbol yt)
=       {especificación de primeroArbol y restoArbol}
    aplanar xt ++ [head (aplanar yt)] ++ tail (aplanar yt)
=       {identidad de listas}
    aplanar xt ++ aplanar yt
```

Queda por implementar *primeroArbol* y *restoArbol*. Podemos combinar las dos funciones en una función *partirArbol* que satisfaga

```
partirArbol :: ArBus α → (α, ArBus α)
partirArbol = par (primeroArbol, restoArbol)
```

Se puede sintetizar fácilmente una definición eficiente de *partirArbol*, y resulta ser

```
partirArbol (Unir xt y yt)
  = if vacio xt then (y, yt) else (x, Unir wt y yt)
    where (x, wt) = partirArbol xt
```

Ejercicios

6.2.1 Enunciar el principio de inducción estructural para el tipo *ArBus α* declarado en esta sección y utilizarlo para demostrar la relación fundamental entre el tamaño y la altura de un árbol binario de búsqueda.

6.2.2 Definir *mapArBus* para el tipo de datos *ArBus α*.

6.2.3 Definir *foldArBus* para el tipo de datos *ArBus α*.

6.2.4 Demostrar que *ordenado* (*insertar x xt*) = *True* para todos los árboles finitos *xt*.

6.2.5 Sea *hx* = *altura xt* y *hy* = *altura yt*. Dar cotas para *altura* (*une xt yt*) en términos de *hx* y *hy* para las dos implementaciones de *une* consideradas en el texto.

6.2.6 Sintetizar el programa eficiente de *partirArbol*.

6.3 Árboles binarios montículo

Además de ciertas condiciones que se describen más adelante, un *árbol binario montículo* es un elemento del tipo de datos

```
data (Ord α) ⇒ ArbMon α  =  Vacio | Unir α (ArbMon α) (ArbMon α)
```

El tipo *ArbMon α* es prácticamente idéntico a *ArBus α* de la sección precedente, excepto por el hecho de que se ha colocado la etiqueta antes de los dos subárboles en lugar de en medio de ellos. Los árboles binarios de búsqueda y los árboles binarios montículo difieren solamente en la relación entre los valores de las diferentes partes del árbol. La condición que ha de satisfacer un árbol montículo es que el valor de cada nodo sea menor o igual que los valores contenidos en cualquiera de los dos subárboles. En otras palabras, los valores a lo largo de cualquier camino descendente en el árbol están siempre en orden creciente.

Podemos formalizar esta condición del modo siguiente. Consideremos primero la función *aplanar* definida por

```
aplanar                 :: (Ord α) ⇒ ArbMon α → [α]
aplanar Vacio           =  []
aplanar (Unir x xt yt)  =  x : mezclar (aplanar xt) (aplanar yt)

mezclar                 :: (Ord α) ⇒ [α] → [α] → [α]
mezclar [] ys           =  ys
mezclar (x : xs) []     =  x : xs

mezclar (x : xs) (y : ys)
  = if x ≤ y then x : mezclar xs (y : ys) else y : mezclar (x : xs) ys
```

La idea de mezclar ordenadamente dos listas ordenadas ya se vió en la Sección 5.2. Ahora podemos definir

```
ordenadoMon  :: (Ord α) ⇒ ArbMon α → Bool
ordenadoMon  =  ordenada · aplanar
```

Esta definición es estructuralmente idéntica a la de *ordenado* de la sección precedente.

Los árboles binarios montículo y los árboles binarios de búsqueda se usan para propósitos diferentes: mientras que los árboles de búsqueda son útiles para operaciones de pertenencia, inserción y borrado, los árboles montículo están mejor adaptados a un conjunto diferente de operaciones. En particular, se puede buscar el valor más pequeño de un montículo en un tiempo constante, mientras que con los árboles binarios de búsqueda se necesita un tiempo proporcional a la distancia al nodo más a la izquierda. Se pueden fundir eficientemente dos montículos en uno, mientras que fundir dos árboles binarios es una operación costosa. En el Capítulo 8 se tratan estos temas más ampliamente.

6.3.1 Construyendo un montículo

Una manera de construir un árbol binario montículo consiste en definir una nueva versión de la operación *construirArBin*. Sin embargo, aunque el programa es bastante

breve, no conduce al modo más eficiente de construir un montículo. Como mostraremos a continuación, se puede construir un montículo en tiempo lineal.

La idea consiste en calcular *construirMon* como la composición de dos funciones auxiliares:

```
construirMon  :: (Ord α) ⇒ [α] → ArbMon α
construirMon  =  propMon · construirArbMin
```

La función *construirArbMin* construye un árbol de altura mínima, que no satisface necesariamente la propiedad *ordenadoMon*, y la función *propMon* reorganiza los valores para asegurar que el resultado satisface esta condición. En el capítulo siguiente mostraremos cómo construir eficientemente un árbol, por tanto nos concentraremos en *propMon*. Definimos

```
propMon                 :: ArbMon α → ArbMon α
propMon Vacio           =  Vacio
propMon (Unir x xt yt)  =  cribar x (propMon xt) (propMon yt)
```

La función *cribar* toma un valor y dos árboles montículo y construye un árbol en el cual los valores se desplazan hacia abajo hasta que se verifica la propiedad característica del montículo:

```
cribar                  :: α → ArbMon α → ArbMon α → ArbMon α
cribar x Vacio Vacio    =  Unir x Vacio Vacio

cribar x (Unir y a b) Vacio
  = if x ≤ y then Unir x (Unir y a b) Vacio else Unir y (cribar x a b) Vacio
cribar x Vacio (Unir z c d)
  = if x ≤ z then Unir x Vacio (Unir z c d) else Unir z Vacio (cribar x c d)

cribar x (Unir y a b) (Unir z c d)
  | x ≤ (y min z)  =  Unir x (Unir y a b) (Unir z c d)
  | y ≤ (x min z)  =  Unir y (cribar x a b) (Unir z c d)
  | z ≤ (x min y)  =  Unir z (Unir y a b) (cribar x c d)
```

Por brevedad, en este programa hemos usado las letras *a, b, c, d* para denotar árboles. El código es sencillo y omitiremos una verificación formal de que el resultado satisface la propiedad *ordenadoMon*.

Mostraremos ahora que *propMon* necesita un tiempo lineal cuando se aplica a un árbol *a* de tamaño *n* y altura mínima $h = \lceil \log (n + 1) \rceil$. Obsérvese que la operación *cribar* se aplica en cada nodo del árbol y puede necesitar un tiempo proporcional a la altura del nodo. Ahora bien, *a* contiene un árbol de altura *h*, a saber, el propio *a*, un máximo de dos subárboles de altura $h - 1$, un máximo de cuatro subárboles de altura $h - 2$, y así sucesivamente. Por tanto, el tiempo total requerido para ejecutar *propMon* es, como mucho, proporcional a

$$h + 2\,(h-1) + 4\,(h-2) + \cdots + 2^{h-1} = \sum_{k=1}^{h} 2^{h-k}\,k$$

pasos. Pero

$$\sum_{k=1}^{h} k\,2^{h-k} = 2^h \sum_{k=1}^{h} \left(\tfrac{k}{2^k}\right) < 2^{h+1} < 4\,(n+1)$$

dado que $h = \lceil \log(n+1) \rceil < \log(n+1) + 1$.

Finalmente, podemos usar árboles binarios montículo para obtener otro algoritmo de ordenación:

```
ordenar  ::  (Ord α) ⇒ [α] → [α]
ordenar  =   aplanar · construirMon
```

Este algoritmo se conoce como *heapsort*[2].

Ejercicios

6.3.1 Un modo de construir un árbol a partir de una lista $[x_0, x_1, \ldots]$ consiste en colocar x_0 como raíz del mismo, x_1, x_2 como raíces de los dos subárboles inmediatos, x_3, x_4, x_5, x_6 como raíces de los cuatro árboles nietos, y así sucesivamente. Este modo de dividir la lista en sublistas de longitudes que son potencias sucesivas de dos, se puede implementar mediante una función *niveles* definida por

```
niveles  ::  [α] → [[α]]
niveles  =   niveles' 1
```

Dar la definición de *niveles'*.

6.3.2 A continuación, la función *construirArbs* toma la lista devuelta por *niveles* y construye una lista de árboles:

```
construirArbs  ::  (Ord α) ⇒ [[α]] → [Htree α]
construirArbs  =   foldr crearNivel [Vacio]
```

Aplicada a una lista de valores $[x_0, x_1, \ldots]$ y a una lista de árboles $[xt_0, xt_1, \ldots]$, la función *crearNivel* produce la nueva lista de árboles

$$[Unir\ x_0\ xt_0\ xt_1,\ Unir\ x_1\ xt_2\ xt_3, \ldots]$$

Si la lista de árboles no es suficientemente larga, se rellena con un número adecuado de árboles vacíos. Dar la definición de *crearNivel*.

6.3.3 Utilizando los ejercicios precedentes, dar una definición de *construirArbMin*. Dar un argumento informal para probar que *construirArbMin* necesita un tiempo lineal.

[2]N. del T.: En español se conoce como algoritmo de *ordenación por el método del montículo*.

6.4 Rosadelfas

El término rosadelfa[3], una traducción libre de rododendro, fue acuñado por Lambert Meertens (véase la Sección 6.6) para describir árboles que tienen una estructura de ramificación múltiple. El tipo de datos *Rosa* α se define mediante

```
data Rosa α  =  Nodo α [Rosa α]
```

Un elemento de *Rosa* α consiste en un nodo etiquetado con un valor, junto con una lista de subárboles. Por definición, las *hojas* no tienen subárboles, en tanto que los *nodos internos* tienen al menos uno. Por ejemplo, *Nodo* 0 [ ] es una rosadelfa, y también lo es

```
Nodo 0 [Nodo 1 [ ], Nodo 2 [ ], Nodo 3 [ ]]
```

Incluso es posible tener árboles con un número infinito de subárboles descendientes inmediatos; por ejemplo,

```
Nodo 0 [Nodo n [] | n ← [1..]]
```

Un árbol *a* es *finitario* si todo nodo de *a* tiene un número finito de subárboles descendientes inmediatos. Por lo tanto, al árbol precedente no es finitario. Nótese que un árbol finitario no ha de tener forzosamente un tamaño finito. Una rosadelfa *a* tiene *anchura* *w* si todos los nodos de *a* tienen como máximo *w* subárboles descendientes inmediatos. Todas las rosadelfas finitas tienen anchura finita. Se dice que una rosadelfa es *k*-aria si todo nodo tiene exactamente *k* subárboles descendientes inmediatos. Los árboles binarios son estructuralmente equivalentes a las rosadelfas 2-arias.

El principio de inducción sobre rosadelfas finitarias dice que, para demostrar que se cumple una propiedad $P(xt)$ para todas las rosadelfas finitarias xt, es suficiente demostrar:

Caso (⊥). Que se cumple $P(\bot)$.

Caso (*Nodo x xts*). Que si se cumple $P(xt)$ para todos los árboles xt de la lista (finita) *xts*, entonces se cumple $P(Nodo\ x\ xts)$.

Si omitimos el primer caso, entonces solamente podemos concluir que $P(xt)$ se cumple para todas las rosadelfas finitas. Nótese la restricción a árboles finitarios en el establecimiento del principio de inducción. Se puede conseguir una formulación más cuidadosa de lo que hace falta demostrar en el segundo caso, si admitimos que *xts* pueda ser una lista parcial, pero no lo haremos. La noción de tamaño y altura se extiende fácilmente a las rosadelfas. Tenemos

```
tam                  ::  Rosa α → Int
tam (Nodo x xts)     =   1 + sum (map tam xts)

altura               ::  Rosa α → Int
altura (Nodo x xts)  =   1 + maxlist (map altura xts)
                         where maxlist = foldl (max) 0
```

[3]N. del T.: En el original, *rose tree*. Los rododendros y las rosadelfas pertenecen a la misma familia.

Según esta definición, la altura de una rosadelfa es siempre mayor que cero. Se deja como ejercicio demostrar que

$$altura = maxRosa \cdot ponerProfs$$

donde

$$\begin{array}{lcl} ponerProfs & :: & Rosa\ \alpha \to Rosa\ Int \\ ponerProfs & = & ponerProfs'\ 1 \end{array}$$

$$\begin{array}{lcl} ponerProfs' & :: & Int \to Rosa\ \alpha \to Rosa\ Int \\ ponerProfs'\ n\ (Nodo\ x\ xts) & = & Nodo\ n\ (map\ (ponerProfs'\ (n{+}1))\ xts) \end{array}$$

y

$$\begin{array}{lcl} maxRosa & :: & (Ord\ \alpha) \Rightarrow Rosa\ \alpha \to \alpha \\ maxRosa\ (Nodo\ x\ xts) & = & x\ \mathbf{max}\ maxlist\ (map\ maxRosa\ xts) \end{array}$$

Se dice que una rosadelfa es *perfecta* si todos sus nodos externos tienen la misma profundidad. La relación básica entre *tam* y *altura* dice que

$$altura\ xt \leq tam\ xt \leq (k \uparrow (altura\ xt) - 1)\ /\ (k-1)$$

para todas las rosadelfas finitas k-arias. La demostración se deja como ejercicio de inducción. Una rosadelfa k-aria perfecta con altura h tiene como tamaño $(k^h-1)/(k-1)$.

Se puede definir también la función *foldRosa* para el tipo de datos *Rosa* α. Dado que

$$Nodo :: \alpha \to [Rosa\ \alpha] \to Rosa\ \alpha$$

la función *foldRosa* toma un único argumento f cuyo tipo es

$$f :: \alpha \to [\beta] \to \beta$$

para cualquier tipo β. La definición de *foldRosa* es

$$\begin{array}{lcl} sfoldRosa & :: & (\alpha \to [\beta] \to \beta) \to Rosa\ \alpha \to \beta \\ foldRosa\ f\ (Nodo\ x\ xts) & = & f\ x\ (map\ (foldRosa\ f)\ xts) \end{array}$$

Como ejemplos, tenemos

$$\begin{array}{lcll} tam & = & foldRosa\ f & \text{where } f\ x\ ns = 1 + sum\ ns \\ maxRosa\ f & = & foldRosa\ f & \text{where } f\ x\ ns = x\ \mathbf{max}\ maxlist\ ns \end{array}$$

La función *map* para rosadelfas, *mapRosa*, se define mediante

$$\begin{array}{lcl} mapRosa & :: & (\alpha \to \beta) \to Rosa\ \alpha \to Rosa\ \beta \\ mapRosa\ f & = & foldRosa\ (Nodo \cdot f) \end{array}$$

6.4.1 Representación de árboles binarios

Dada su estructura de ramificación múltiple, las rosadelfas finitas parecen capaces de modelizar estructuras más generales que los árboles binarios; por tanto puede ser una sorpresa que estén en correspondencia biunívoca: toda rosadelfa se puede representar mediante un único árbol binario finito, y a la inversa. De hecho, la correspondencia no es única, pero el método empleado para asociar rosadelfas con árboles binarios es

exactamente el mismo que el utilizado para expresiones escritas en forma currificada y no currificada. Por ejemplo, consideremos las expresiones

$$f\ (g\ (x, y), z, h\ (t)) \quad \text{y} \quad f\ (g\ x\ y)\ z\ (h\ t)$$

La expresión de la izquierda se puede representar mediante la rosadelfa

$$Nodo\ f\ [Nodo\ g\ [Nodo\ x\ [\ ], Nodo\ y\ [\ ]], Nodo\ z\ [\ ]], Nodo\ h\ [Nodo\ t\ [\ ]]]]$$

La expresión de la derecha, en su forma completamente parentizada, es

$$((f\ ((g\ x)\ y)\ z)\ (h\ t))$$

y se puede representar como el árbol binario

$$\begin{array}{l} Unir\ (Unir\ (Unir\ (Hoja\ f) \\ \qquad\qquad (Unir\ (Unir\ (Hoja\ g)\ (Hoja\ x))\ (Hoja\ y))) \\ \qquad\quad (Hoja\ z)) \\ \quad (Unir\ (Hoja\ h)\ (Hoja\ t)) \end{array}$$

Podemos transformar un elemento finito de *Rosa* α en un elemento de *ArBin* α mediante la función *aArBin*, definida por

$$\begin{array}{lcl} aArBin & :: & Rosa\ \alpha \to ArBin\ \alpha \\ aArBin\ (Nodo\ x\ xts) & = & foldl\ Unir\ (Hoja\ x)\ (map\ aArBin\ xts) \end{array}$$

Dicho con palabras, para convertir una rosadelfa en un árbol binario, primero convertimos todos los subárboles (utilizando *map aArBin*) y después combinamos los resultados en un solo árbol binario, aplicando *fold* desde la izquierda, siguiendo el orden de asociación (por la izquierda) de la aplicación. Es más difícil implementar la correspondencia inversa *aRosa*. Esta función se especifica mediante

$$\begin{array}{lcl} aRosa & :: & ArBin\ \alpha \to Rosa\ \alpha \\ aRosa\ (aArBin\ xt) & = & xt \end{array}$$

donde *xt* toma valores en las rosadelfas finitarias. Para sintetizar un programa a partir de esta especificación, necesitaremos las dos propiedades siguientes de *aArBin*:

$$\begin{array}{lcl} aArBin\ (Nodo\ x\ [\,]) & = & Hoja\ x \\ aArBin\ (Nodo\ x\ (xts +\!\!+ [xt])) & = & Unir\ (aArBin\ (Nodo\ x\ xts))\ (aArBin\ xt) \end{array}$$

La primera ecuación es inmediata a partir de la definición de *aArBin*, y la segunda se puede demostrar aplicando primero el tercer teorema de dualidad para reescribir la definición de *aArBin* de la forma

$$aArBin\ (Nodo\ x\ xts) = foldr\ (flip\ Unir)\ (Hoja\ x)\ (reverse\ (map\ aArBin\ xts))$$

Los detalles se dejan como ejercicio.

La síntesis de un programa para *aRosa* se realiza mediante análisis por casos.

Caso ($Nodo\ x\ [\ ]$). Razonamos así:

$$\begin{array}{ll} & aRosa\ (aArBin\ (Nodo\ x\ [\,])) = Nodo\ x\ [\,] \\ \Leftarrow & \quad \{\text{primera propiedad de } aArBin\} \\ & aRosa\ (Hoja\ x) = Nodo\ x\ [\,] \end{array}$$

Caso ($Nodo\ x\ (xts +\!\!+ [xt])$). Razonamos de esta forma:

$$
\begin{array}{ll}
 & aRosa\ (aArBin\ (Nodo\ x\ (xts +\!\!+ [xt]))) = Nodo\ x\ (xts +\!\!+ [xt]) \\
\Leftarrow & \quad \{\text{segunda propiedad de } aArBin\} \\
 & aRosa\ (Unir\ (aArBin\ (Nodo\ x\ xts)))\ (aArBin\ xt) = Nodo\ x\ (xts +\!\!+ [xt]) \\
\equiv & \quad \{\text{introducimos } xb = aArBin\ (Nodo\ x\ xts) \text{ e } yb = aArBin\ xt\} \\
 & aRosa\ (Unir\ xb\ yb) = Nodo\ x\ (xts +\!\!+ [xt]) \\
 & \textbf{where}\ Nodo\ x\ xts = aRosa\ xb \\
 & \qquad\quad xt = aRosa\ yb
\end{array}
$$

En el último paso, hemos usado la especificación de *aRosa* dos veces. Resumiendo, hemos derivado

$$
\begin{array}{lll}
aRosa\ (Hoja\ x) & = & Nodo\ x\ [\,] \\
aRosa\ (Unir\ xb\ yb) & = & Nodo\ x\ (xts +\!\!+ [xt]) \\
 & & \textbf{where}\ Nodo\ x\ xts = aRosa\ xb \\
 & & \qquad\quad xt = aRosa\ yb
\end{array}
$$

Este programa no es muy eficiente porque concatenar *xt* a *xts* necesita un tiempo proporcional a la longitud de *xts*. Para obtener una versión más eficiente, definimos

$$
\begin{array}{lll}
recolectar & :: & ArBin\ \alpha \to [Rosa\ \alpha] \to Rosa\ \alpha \\
recolectar\ xb\ yts & = & Nodo\ x\ (xts +\!\!+ yts)\ \textbf{where}\ Nodo\ x\ xts = aRosa\ xb
\end{array}
$$

En particular, $aRosa\ xb = recolectar\ xb\ [\,]$. Ahora es fácil derivar que

$$
\begin{array}{lll}
recolectar\ (Hoja\ x)\ yts & = & Nodo\ x\ yts \\
recolectar\ (Unir\ xb\ yb)\ yts & = & recolectar\ xb\ (aRosa\ yb : yts)
\end{array}
$$

La idea ha consistido en utilizar un *parámetro acumulador* para eliminar las costosas operaciones de concatenación (++). La técnica del parámetro acumulador se estudiará en el capítulo siguiente. Exactamente la misma idea está detrás de la transformación de la definición ingenua de *reverse* a una más eficiente.

6.4.2 Recorridos en profundidad y en anchura

La función *aplanar* obtiene la lista de los valores de una rosadelfa:

$$
\begin{array}{lll}
aplanar & :: & \alpha \to [\alpha] \\
aplanar\ (Nodo\ x\ xts) & = & x : concat\ (map\ aplanar\ xts)
\end{array}
$$

Esta función enumera, en este orden, el valor de la raíz, los valores del subárbol más a la izquierda, los valores del siguiente subárbol por la izquierda, y así sucesivamente. Los valores se presentan en lo que se conoce como un *recorrido en profundidad*. Como mostraremos en el capítulo siguiente, *aplanar* necesita, como máximo, un tiempo proporcional a $n \log n$, siendo n el tamaño del árbol. No obstante, es posible implementar el recorrido en profundidad en un tiempo proporcional a n. Igualmente importante, la cantidad de espacio requerida para llevar a cabo el recorrido en profundidad es, como máximo, proporcional a la altura del árbol multiplicada por el máximo número de subárboles de cada nodo. La conclusión es que el recorrido en profundidad es eficiente tanto en lo que respecta al tiempo como al espacio.

Hay otra manera de aplanar una rosadelfa: por niveles. Se obtiene primero el valor de la raíz, después los valores de las raíces de los subárboles descendientes inmediatos, después los valores de las raíces del siguiente nivel, y así sucesivamente. Esta manera de enumerar los valores se conoce como *recorrido en anchura*. En la Sección 6.6 se da un ejemplo de un problema en el que hace falta el recorrido en anchura. El recorrido en anchura tiene la ventaja de que se puede aplicar a un árbol infinito, produciendo una lista infinita con todos los valores almacenados en el árbol. Por otra parte, un recorrido en profundidad de un árbol infinito no producirá, en general, una lista infinita que contenga todos los valores: en particular, si el camino más a la izquierda es infinito, entonces *aplanar* sólo producirá valores a lo largo de este camino.

A diferencia del recorrido en profundidad, el recorrido en anchura de un árbol no se puede conseguir a partir del recorrido en anchura de sus subárboles. Podemos, sin embargo, definir una función *niveles* mediante

$$
\begin{array}{lll}
niveles & :: & Rosa\ \alpha \to [[\alpha]] \\
niveles\ (Nodo\ x\ xts) & = & [x] : combinar\ (map\ niveles\ xts)
\end{array}
$$

para una función *combinar* apropiada, y después definir

$$
\begin{array}{lll}
recAnchura & :: & Rosa\ \alpha \to [\alpha] \\
recAnchura & = & concat \cdot niveles
\end{array}
$$

La función *recAnchura* produce los valores del recorrido en anchura.

El problema consiste en especificar $combinar :: [[[\alpha]]] \to [[\alpha]]$. Esta función ha de concatenar los niveles correspondientes de la lista de argumentos. Por ejemplo, dada la lista

$$[[[x1, x2], [x3, x4, x5], [x6]], [[y1]], [[z1, z2, z3], [z4]]]$$

el resultado devuelto por *combinar* será la lista

$$[[x1, x2, y1, z1, z2, z3], [x3, x4, x5, z4], [x6]]$$

Supongamos que definimos *indexar* mediante

$$
\begin{array}{lll}
indexar & :: & Int \to [[\alpha]] \to [\alpha] \\
indexar\ i\ xss & = & \textbf{if}\ i < length\ xss\ \textbf{then}\ xss\,!!\,i\ \textbf{else}\ [\,]
\end{array}
$$

Entonces, podemos especificar *combinar* mediante la condición

$$indexar\ i \cdot combinar = concat \cdot map\ (indexar\ i)$$

para todo i. Dado que $concat = foldr\ (+\!\!+)\ [\,]$, parece razonable buscar una definición de *combinar* de la forma

$$
\begin{array}{lll}
combinar & :: & [[[\alpha]]] \to [[\alpha]] \\
combinar & = & foldr\ (\oplus)\ [\,]
\end{array}
$$

para algún operador $\oplus$ apropiado. Recurrimos a la ley de fusión para *foldr* que nos da la especificación de $\oplus$:

$$indexar\ i\ (xss \oplus yss) = indexar\ i\ xss +\!\!+ indexar\ i\ yss$$

Ahora, es un ejercicio inmediato de síntesis la obtención del siguiente programa para $\oplus$:

$$(\oplus) :: [[\alpha]] \to [[\alpha]] \to [[\alpha]]$$

```
[ ] ⊕ yss              = yss
(xs : xss) ⊕ [ ]       = xs : xss
(xs : xss) ⊕ (ys : yss) = (xs ++ ys) : (xss ⊕ yss)
```

La operación (⊕) es similar a *zipWith* (++), excepto por el hecho de que las listas no se truncan ajustándolas a la longitud de la más corta.

Aunque los dos procedimientos, *aplanar* y *recAnchura*, tienen la misma complejidad en tiempo, tienen costes bastante diferentes en espacio. El espacio requerido para evaluar *recAnchura a* es proporcional al tamaño de *a*. Para apreciar el significado de este dato, imaginemos un árbol de altura h con k subárboles en cada nodo. Entonces, *recAnchura* necesita un espacio proporcional a h^k, mientras que *aplanar* requiere un espacio proporcional a $h \times k$. Simplemente, no se puede aplanar un árbol grande utilizando el recorrido en anchura sin agotar el espacio disponible. La Sección 6.6, que trata un problema que requiere exactamente tal recorrido, retoma esta cuestión.

Ejercicios

6.4.1 ¿Qué relación, caso de haber alguna, existe entre el tamaño y la altura de una rosadelfa?

6.4.2 Supongamos que *a* es una rosadelfa finitaria pero infinita. ¿Produce *aArBin a* un árbol binario infinito?

6.4.3 La definición eficiente de *aRosa* se obtuvo en dos etapas, pero si utilizamos también la propiedad *aArBin* (*aRosa xb*) = *xb* para todos los árboles binarios finitos *xb*, entonces podemos hacerlo en una etapa. Definimos *recolectar* mediante

```
recolectar          :: ArBin α → [Rosa α] → ArBin α
recolectar xb yts   = aRosa (foldl Unir xb (map aArBin yts))
```

Utilizar esta definición de *recolectar* para derivar la que se dio en el texto.

6.4.4 Definimos la función *recProf* (recorrido en profundidad) mediante

```
recProf  :: [Rosa α] → [α]
recProf  = concat · map aplanar
```

Sintetizar el siguiente programa para *recProf*:

```
recProf [ ]                  = [ ]
recProf (Nodo x xts : yts)   = [x] ++ recProf (xts ++ yts)
```

¿Cómo se define *aplanar* en términos de *recProf*?

6.4.5 Definimos la función *recAnchura* (recorrido en anchura) mediante

```
recAnchura :: [Rosa α] → [α]
recAnchura = concat · combinar · map niveles
```

Como ejercicio difícil de síntesis, derivar el siguiente programa para *recAnchura*:

```
recAnchura :: [Rosa α] → [α]
recAnchura [ ] = [ ]
recAnchura (Nodo x xts : yts) = [x] ++ recAnchura (yts ++ xts)
```


6.5 Ejemplo: árboles de Huffman

Una buena ilustración del uso de árboles binarios aparece en el problema de la compresión de datos. Como muchos usuarios de computadores saben, a menudo es necesario almacenar ficheros de información de forma tan compacta como sea posible para liberar un precioso espacio necesario para otros propósitos más urgentes. Supongamos que la información que va a ser almacenada es un texto consistente en una secuencia de caracteres. El código estándar ISO utilizado en Haskell emplea ocho bits para representar cada uno de los $2^8 = 256$ caracteres diferentes; por tanto, un texto de n caracteres contiene $8n$ bits de información. Puesto que el estándar ISO es un código de longitud fija, los caracteres del texto se pueden recuperar decodificando cada grupo sucesivo de ocho bits.

Una idea para reducir el número total de bits requerido para codificar un texto consiste en abandonar la noción de códigos de longitud fija y, en su lugar, buscar un sistema de codificación basado en la frecuencia relativa de aparición de los caracteres en el texto. La idea básica es tomar una muestra del texto, estimar el número de veces que aparecerá cada carácter, y escoger códigos cortos para los caracteres más frecuentes y códigos más largos para los caracteres menos frecuentes. Por ejemplo, si tomamos los códigos

```
a → 0
l → 10
g → 11
```

entonces la palabra "alga" se puede codificar como la secuencia de bits 010110 de longitud 6.

Es importante resaltar que se deben escoger los códigos de modo que se asegure que el texto codificado se pueda descifrar de manera única. Para ilustrar este punto, supongamos que los códigos hubieran sido

```
a → 0
l → 10
g → 1
```

Según este esquema, "alga" habría sido codificada mediante la secuencia 01010 de longitud 5. Sin embargo, la cadena "all" habría sido codificada exactamente con la misma secuencia, y obviamente no se desea esto. El modo más simple de evitar este problema es escoger los códigos de forma que no exista ningún código que sea un *prefijo* inicial de otro.

Además de requerir la propiedad de desciframiento único, también queremos que la codificación sea *óptima*. Un esquema de codificación óptimo es aquel que minimiza la longitud esperada del texto codificado. Más precisamente, si cada carácter diferente c_j, $1 \le j \le n$, tiene una probabilidad de aparición p_j, hemos de escoger los códigos con longitudes l_j tales que

$$\sum_{j=1}^{n} p_j l_j$$

sea lo más pequeño posible.

Un método para conseguir una codificación óptima que satisfaga la propiedad de prefijo enunciada es la codificación de Huffman (en honor de su inventor, David Huffman). Cada carácter se almacena como una hoja de un árbol binario, cuya estructura se determina a partir de las frecuencias calculadas. El código asignado a un carácter *c* es la secuencia de dígitos binarios que describe el camino desde la raíz hasta la hoja que contiene *c*. Para ilustrar esta idea, consideremos el árbol

```
Unir (Unir (Hoja 'x') (Hoja 'e')) (Hoja 't')
```

En este árbol, el carácter 'x' se codifica como 00, el carácter 'e' como 01, y el carácter 't' como 1. Hay cuatro aspectos en el problema de implementar la codificación de Huffman: (i) coleccionar la información a partir de la muestra; (ii) construir un árbol binario; (iii) codificar el texto; y (iv) decodificar una secuencia de bits. Trataremos primero la codificación y la decodificación, después la recolección de información y, finalmente, la construcción del árbol.

6.5.1 Codificación y decodificación

La definición de la función *decodif* que decodifica una secuencia de bits con respecto a un árbol de Huffman dado, es bastante directa. Utilizamos un prefijo de la secuencia de bits para recorrer el árbol hasta que se alcanza una hoja. Se genera el carácter asociado a dicha hoja y, si el resto de la secuencia de bits no es vacía, se recorre otra vez el árbol para encontrar el siguiente carácter. Se supone que la secuencia de bits corresponde a una codificación correcta; si no es así, entonces la decodificación fallará. El programa es

```
decodif :: ArBin Char → [Bit] → [Char]
decodif a cs
  = if null cs then [ ] else decodif1 a cs
    where decodif1 (Hoja x) cs        = x : decodif a cs
          decodif1 (Unir xt yt) (0 : cs) = decodif1 xt cs
          decodif1 (Unir xt yt) (1 : cs) = decodif1 yt cs
```

El tiempo de decodificación es claramente lineal con respecto a la longitud de la secuencia de bits, que es lo mejor que se puede conseguir.

La función *codif* para codificar una secuencia de caracteres con respecto a un árbol de Huffman dado no es fácil de definir si se desea un programa eficiente. Los árboles de Huffman son buenos para encontrar el carácter asociado con una secuencia de bits dada, pero malos para encontrar la secuencia de bits asociada con un carácter dado. Lo que nos gustaría hacer primero es transformar el árbol en alguna estructura para la cual la segunda tarea se pueda hacer más eficientemente. Una posibilidad es definir una función

```
transformar :: ArBin Char → TablaCodigos
```

donde

```
type TablaCodigos = [(Char, [Bit])]
```

La función *transformar* produce una tabla consistente en una lista de pares de caracteres y secuencias de bits. Ahora podemos definir


```
codif :: ArBin Char → [Char] → [Bit]
codif a = concat · map (buscar tablaCodigos)
          where tablaCodigos = transformar a

buscar                  :: TablaCodigos → Char → [Bit]
buscar ((x, bs) : xbs) y = if x == y then bs else buscar xbs y
```

El tiempo requerido para llevar a cabo una operación *buscar* depende precisamente de dónde se encuentre el carácter buscado en la tabla. Parece razonable situar los caracteres menos frecuentes en el texto de entrada antes que los caracteres que aparezcan con menos frecuencia. Como la frecuencia de aparición se corresponde inversamente con la longitud de la secuencia de bits asociada (después de todo, ésta es la clave de la codificación de Huffman), la conclusión es que la tabla de códigos debería estar ordenada por orden creciente de longitudes del segundo componente. Por tanto,

```
transformar (Hoja x)      = [(x, [ ])]
transformar (Unir xt yt)  = fusionaHuf (transformar xt) (transformar yt)
```

La función *fusionaHuf* fusiona dos tablas, añadiendo un bit cero a todos los códigos que provengan de la primera tabla y un bit uno a los códigos que provengan de la segunda:

```
fusionaHuf          :: TablaCodigos → TablaCodigos → TablaCodigos
fusionaHuf [ ] ycs  = [(y, 1 : cs) | (y, cs) ←ycs]
fusionaHuf xbs [ ]  = [(x, 0 : bs) | (x, bs) ←xbs]

fusionaHuf ((x, bs) : xbs) ((y, cs) : ycs)
  | length bs ≤ length cs = (x, 0 : bs) : fusionaHuf xbs ((y, cs) : ycs)
  | otherwise             = (y, 1 : cs) : fusionaHuf ((x, bs) : xbs) ycs
```

Se puede mejorar sustancialmente la eficiencia del programa evitando recalcular las longitudes de las secuencias de bits. Las tablas de códigos deberían haber sido, por tanto, listas de ternas:

```
type TablaCodigos = (Char, [Bit], Int)
```

El tercer componente describe la longitud del segundo. Dejaremos esta optimización como ejercicio.

6.5.2 Análisis de la muestra

Supondremos que la muestra contiene todos los caracteres del texto que se va a codificar. Las frecuencias relativas de los caracteres de la muestra se pueden calcular si primero la ordenamos para agrupar las apariciones de un mismo carácter y después reemplazamos cada secuencia de un mismo carácter por un par que consista en el propio carácter y en el número de veces que éste se repite. Finalmente, podemos ordenar la secuenia de pares por orden creciente de frecuencias. El programa es

```
muestra :: [Char] → [(Char, Int)]
muestra = ordenar frec · colapsar · ordenar id
```

Recordemos de la Sección 5.2 que la función *ordenar* tenía como tipo

```
ordenar :: (Ord β) ⇒ (α → β) → [α] → [α]
```

En particular, *ordenar id* :: [*Char*] → [*Char*] ordena una lista de caracteres por orden alfabético.

La función *colapsar* reemplaza secuencias del mismo carácter por un par constituido por el carácter y la longitud de la secuencia:

```
colapsar          :: [Char] → [(Char, Int)]
colapsar [ ]      = [ ]
colapsar (x : xs) = (x, 1 + length ys) : colapsar zs
                    where (ys, zs) = span (== x) xs
```

Recordemos que *span p xs* = (*takeWhile p xs, dropWhile p xs*).

Finalmente, la función *ordenar frec*, donde

```
frec      :: (Char, Int) → Int
frec (x, m) = m
```

ordena una lista de pares por orden creciente de frecuencias.

6.5.3 Construcción de un árbol de Huffman

Ahora nos ocupamos de la parte más interesante: la construcción de un árbol de Huffman. Después de analizar la muestra, obtenemos una lista de pares:

$$[(c_0, w_0), (c_1, w_1), \ldots, (c_n, w_n)]$$

donde los c_j $(0 \le j \le n)$ son caracters y los w_j $(0 \le j \le n)$ son números, llamados *pesos*, que indican las frecuencias. La probabilidad de que aparezca el carácter c_j es, por tanto, w_j/W, donde $W = \sum w_j$. También se cumple que $w_0 \le w_1 \le \ldots \le w_n$.

Esquemáticamente, el procedimiento para construir un árbol de Huffman consiste en convertir primero la lista de pares anterior en una lista de árboles y fundir después, sistemáticamente, los dos árboles con menores pesos en uno solo, hasta que sólo quede un árbol. El peso de un árbol que conste de una única hoja es el peso del carácter contenido en esa hoja; y el peso de un árbol con un nodo interno es la suma de los pesos de sus dos subárboles. Para evitar recalcular pesos, es conveniente etiquetar los árboles con sus pesos. Por tanto, introducimos el tipo

```
data Huff = HojaHuff Int Char | NodoHuff Int Huff Huff
```

Cuando se ha construido el árbol, podemos eliminar la información de pesos mediante una función *desetiquetar* definida por

```
desetiquetar                 :: Huff → ArBin Char
desetiquetar (HojaHuff w x)  = Hoja x
desetiquetar (NodoHuff w xt yt) = Unir (desetiquetar xt) (desetiquetar yt)
```

La función *construirHuff* para construir un árbol de Huffman tiene el siguiente aspecto:

```
construirHuff :: [(Char, Int)] → Huff
construirHuff = desenlistar · until unitaria combinar · map construirHoja
```


donde

```
construirHoja       :: (Char, Int) → Huff
construirHoja (c, w) = HojaHuff w c
```

La función estándar *until p f* aplica repetidamente *f* hasta que se cumple *p* y *unitaria* comprueba que una lista contiene un único elemento. El efecto de *combinar* sobre una lista *ts* de árboles es combinar los dos árboles de *ts* que tengan menores pesos.

Para determinar en cada etapa cuáles son los árboles con menores pesos, simplemente mantenemos los árboles en orden creciente de peso. Por tanto,

```
combinar                :: [Huff] → [Huff]
combinar (xt : yt : xts) = insertar (NodoHuff w xt yt) xts
                           where w = peso xt + peso yt

peso                    :: Huff → Int
peso (HojaHuff w c)     = w
peso (NodoHuff w x y)   = w
```

La función *insertar* inserta un árbol en el lugar correcto para mantener la propiedad de que la lista de árboles está en orden creciente de pesos:

```
insertar          :: Huff → [Huff] → [Huff]
insertar xt yts   = uts ++ [xt] ++ vts
                    where (uts, vts) = span p yts
                          p yt       = (peso yt ≤ peso xt)
```

Veamos el funcionamiento de este algoritmo sobre un ejemplo. Consideremos la secuencia

```
[('G', 8), ('R', 9), ('A', 11), ('T', 13), ('E', 17)]
```

de caracteres y sus pesos. El primer paso es convertirla en la lista

```
[HojaHuff 8 'G', HojaHuff 9 'R', HojaHuff 11 'A', HojaHuff 13 'T', HojaHuff 17 'E']
```

El siguiente paso es combinar los dos primeros árboles e insertar el resultado en el lugar adecuado

```
[HojaHuff 11 'A',
 HojaHuff 13 'T',
 NodoHuff 17 (HojaHuff 8 'G') (HojaHuff 9 'R'),
 HojaHuff 17 'E']
```

El resultado del segundo paso es la lista

```
[NodoHuff 17 (HojaHuff 8 G) (HojaHuff 9 'R'),
 HojaHuff 17 'E',
 NodoHuff 24 (HojaHuff 11 'A') (HojaHuff 13 'T')]
```

El tercer paso da

```
[NodoHuff 24 (HojaHuff 11 'A') (HojaHuff 13 'T'),
 NodoHuff 34 (NodoHuff 17 (HojaHuff 8 'G') (HojaHuff 9 'R'))
             (HojaHuff 17 'E')]
```

El árbol final es

NodoHuff 58(*NodoHuff* 24 (*HojaHuff* 11 'A') (*HojaHuff* 13 'T'))
(*NodoHuff* 34 (*NodoHuff* 17 (*HojaHuff* 8 'G') (*HojaHuff* 9 'R'))
(*HojaHuff* 17 'E'))

En este árbol, los caracteres 'A', 'T' y 'E' están codificados por una secuencia de dos bits y los caracteres 'G' y 'R' mediante secuencias de tres bits.

La longitud promedio, o *esperada*, de un código de caracteres es

$$\sum w_j\, l_j \,/\, \sum w_j$$

donde l_j es el número de bits asignado al carácter c_j. En el ejemplo anterior este valor es

$$((11 + 13 + 17) \times 2 + (8 + 9) \times 3) \,/\, (8 + 9 + 11 + 13 + 17) = 2.29\ldots$$

La propiedad clave de un código de Huffman es que minimiza la longitud esperada. Dicho de otra manera, un árbol de Huffman tiene la propiedad de ser un árbol binario, con valores en las hojas $w_1, w_2, \ldots, w_n$, que minimiza la suma de las longitudes "ponderadas" $w_j\, l_j$ donde $1 \le j \le n$. Para obtener una demostración de este hecho, el lector deberá consultar las referencias dadas al final del capítulo.

Ejercicios

6.5.1 Explicar por qué el código de Huffman tiene la propiedad de prefijo.

6.5.2 Construir un código que no tenga la propiedad de prefijo pero que, sin embargo, sea tal que todo texto se pueda decodificar unívocamente.

6.5.3 Optimizar la definición de *transformar* incluyendo información acerca de la longitud en las tablas de códigos.

6.6 Ejemplo: el número de Meertens

El propósito de esta sección es ilustrar el uso de rosadelfas para organizar la búsqueda de un número que satisface una propiedad peculiar. El problema está relacionado con la descomposición de un número en factores primos. Por ejemplo, $400 = 2^4 3^0 5^2$ y $432 = 2^4 3^3 5^0$. Supongamos que definimos $g(n)$ tomando la representación decimal $d_1, d_2, \ldots, d_k$ de n y definiendo $g(n) = 2^{d_1} 3^{d_2} \ldots p_k^{d_k}$, donde p_k es el k-ésimo número primo. Así, $g(402) = 400$ y $g(430) = 432$. Los números n y $g(n)$ no están siempre tan cercanos como sugieren estos dos ejemplos; de hecho, la mayor parte de las veces son tremendamente diferentes. Surge la siguiente pregunta: ¿coinciden alguna vez?

La función g recibe su nombre en honor de Kurt Gödel que explotó la idea de codificar una secuencia de caracteres mediante un solo número, en su famoso artículo sobre la incompletitud de la aritmética. Aquí usamos el mismo mecanismo, codificar un número codificando su representación decimal. Si $n = g(n)$, entonces diremos que n es un *número de Meertens*. El primer número de Meertens se llamó así en honor de

un colega nuestro, Lambert Meertens, con ocasión de su veinticinco aniversario en el CWI, de Amsterdam.

La existencia de un número de Meertens no es obvia en absoluto. El número ha de ser par y un pequeño cálculo mental muestra que no puede tener uno, dos o tres dígitos. Por ejemplo, cualquier número de tres dígitos habría de terminar en un cero y por tanto sería a la vez divisible y no divisible por cinco. Por otra parte, existen algunos números que casi cumplen la propiedad; los ejemplos anteriores muestran que existen números n tales que n y $g(n)$ difieren como máximo en dos unidades. Más aún, a medida que n crece el valor de $g(n)$ crece mucho más deprisa; por tanto no resulta evidente que debieran coincidir alguna vez.

Por supuesto, el problema no es específico de la base decimal. En base dos, cada uno de los números 2, 6 y 10 es un número de Meertens. Por ejemplo, 6 en binario es 110 y $6 = 2^1 3^1 5^0$. El número 10 es también un número de Meertens en base tres: 10 en base tres es 101 y $10 = 2^1 3^0 5^1$. Tales resultados son estimulantes, pero encontrar números de Meertens para simples cadenas bits no es suficiente. Por otra parte, serían suficientemente apropiados números de algunas decenas.

Nuestra intención ahora es construir un programa que encuentre números de Meertens. El método más obvio es definir

```
meertens  ::  [Integer]
meertens  =   [n | n <- [1..], n == godel n]

godel     ::  Integer -> Integer
godel n   =   product (zipWith (^) primos (digitos n))
```

La función *digitos* devuelve los dígitos de un número natural. Se puede definir la lista de los números primos de muchas maneras, por ejemplo

```
primos  ::  [Int]
primos  =   [p | p <- [2..], primo p]
```

Dejaremos la definición de *primo*, *product* y *digitos* como ejercicio.

El valor de *meertens*, o bien es una lista infinita (si existe un número infinito de números de Meertens), o bien es una lista parcial. En cualquier caso, la evaluación no terminará. Aun cuando preguntemos por *head meertens*, el cálculo resulta ser penosamente lento. El número de Gödel de cada candidato se calcula cada vez desde cero y esto conlleva un gran número de pasos.

Como preparación para un cambio de representación, podemos reescribir la definición de *meertens* en la forma

```
meertens  =  [n | (n, g) <- map gn [1..], n == g]
             where gn n = (n, godel n)
```

Ahora la idea es definir una rosadelfa *gArbol* de modo que

```
recAnchura gArbol  =  map gn [1..]
```

Recordemos de la Sección 6.4 que *recAnchura* produce los valores de una rosadelfa según el recorrido en anchura. Cada nodo del árbol *gArbol*, excepto el primero, tiene 10 subárboles, uno para cada dígito. El primer árbol tiene 9 subárboles, ya que los números comienzan por un dígito que no es cero. Cada nodo del árbol está etiquetado

por dos enteros, n y g, donde g = *godel* n y la representación decimal de n define el camino en el árbol desde la raíz hasta el nodo dado. Por tanto, el árbol tiene tipo *Arbol*, siendo

```
data Arbol  =  Nodo Valor [Arbol]
type Valor  =  (Integer, Integer)
```

El siguiente programa construye un árbol infinito con un valor inicial de (0, 1), que es correcto dado que $g(0) = 2^0 = 1$:

```
gArbol  ::  Arbol
gArbol  =   podar (construirArbol potPrimos (0, 1))
```

La función *podar* poda la rama situada más a la izquierda para eliminar los números que empiezan por 0:

```
podar              ::  Arbol → Arbol
podar (Nodo x ts)  =   Nodo x (tail ts)
```

La tarea de construir el árbol se encomienda a la función *construirArbol*. Esta función toma la lista infinita *potPrimos* de las potencias de cada número primo y un valor. La lista *potPrimos* se define mediante

```
potPrimos  ::  [[Integer]]
potPrimos  =   map pots primos
```

El valor de *pots p* es una lista infinita $[1, p, p^2, \ldots]$ de potencias de p:

```
pots    ::  Integer → [Integer]
pots p  =   iterate (p ×) 1
```

La función estándar *iterate* se define mediante

```
iterate    ::  (α → α) → α → [α]
iterate f x  =  x : iterate f (f x)
```

Ahora definimos

```
construirArbol              ::  [[Integer]] → Valor → Arbol
construirArbol (ps : pps) x  =  Nodo x (map (construirArbol pps) (valores ps x))
```

donde

```
valores              ::  [Integer] → Valor → [Valor]
valores ps (n, g)    =   zip (map (m +) ds) (map (g ×) ps)
                         where m = 10 × n; ds = [0..9]
```

La lista *meertens* de números de Meertens se puede calcular, al menos en teoría, mediante

```
meertens  =  [n | (n, g) ← recAnchura gArbol, n == g]
```

No existe dificultad teórica alguna en presentar los valores de un árbol infinito recorriéndolo en anchura porque se realiza nivel por nivel. Sin embargo, existe una importante dificultad práctica: el cómputo agota enseguida la memoria y se detiene. Recorrer en anchura un árbol requiere un espacio proporcional al tamaño del árbol, y

el tamaño de los árboles de Meertens crece exponencialmente. Ningún evaluador, real o imaginario, tiene suficientes recursos para llegar muy lejos en el cómputo.

La solución obvia es restringir la altura del árbol y recorrerlo en profundidad. Tenemos que hacer el árbol finito si queremos usar el recorrido en profundidad, porque no se puede recorrer un árbol infinito de Meertens en profundidad. Si construimos un árbol de altura k, entonces podemos buscar números de Meertens n en el rango $1 \le n < 10^k$. La definición revisada es

```
meertens    ::  Int → [Integer]
meertens k  =   [n | (n, g) ← aplanar (gArbol k), n == g]
```

La función *gArbol* devuelve árboles de una altura dada:

```
gArbol    ::  Int → [Integer]
gArbol k  =   podar (construirArbol potPrimos (0, 1))
              where potPrimos = map pots (take k primos)

construirArbol              ::  [[Integer]] → Valor → Arbol
construirArbol [ ] x        =   Nodo x [ ]
construirArbol (ps : pps) x =   Nodo x (map (construirArbol pps) (valores ps x))
```

Obsérvese que hemos reemplazado *recAnchura* por *aplanar* en la definición de *meertens*. Se deduce que ya no está garantizado que *meertens* produzca los números en orden ascendente.

La definición revisada es más rápida, pero todavía muy lenta. Existe otra optimización que podemos aplicar. No tiene sentido generar cualquier subárbol cuya raíz sea (n, g) que cumpla $g \ge 10^k$ porque, por construcción, todos los valores (n, g) en el árbol cumplen $n < 10^k$. Para incorporar esta comprobación sin añadir un parámetro adicional, haremos locales a la definición de *gArbol* las definiciones de *construirArbol* y *valores*:

```
gArbol k = podar (construirArbol potPrimos (0, 1))
  where
  potPrimos                 =  map pots (take k primos)
  construirArbol [ ] x      =  Nodo x [ ]
  construirArbol (ps : pps) x =  Nodo x (map (construirArbol pps)
                                        (valores ps x))
  valores ps (n, g)         =  zip (map (m +) ds) (cortar (map (g ×) ps))
                               where m = 10 × n
  cortar                    =  takeWhile (< 10^k)
  ds                        =  [0..9]
```

Incluso con esas optimizaciones la búsqueda es todavía muy lenta. Aparte de estudiar el problema cuidadosamente para ver qué conocimientos de la teoría de números podemos aplicar, hay una técnica general para mejorar las cosas que consideramos en la siguiente sección.

6.6.1 Deforestación

La deforestación es el nombre general de una técnica de programación que elimina tipos de datos intermedios en un programa. La idea será considerada en un marco

general en el capítulo siguiente, pero ahora la emplearemos para el árbol que tan cuidosamente hemos construido. El ahorro no será enorme, pero sí será significativo. Si se puede acelerar el programa, digamos en "un factor de dos", entonces la optimización merece la pena. (Seguramente en este punto el lector ya habrá sospechado correctamente que el primer número de Meertens no es pequeño.)

Como todas las técnicas correctas de optimización, la deforestación se puede efectuar automáticamente. La clave del cálculo radica en combinar las funciones que producen y consumen elementos del tipo de datos intermedio. Empezamos por calcular, para $k > 0$, lo siguiente

$$
\begin{array}{ll}
 & (aplanar \cdot gArbol)\ k \\
= & \quad \{\text{definición de } gArbol, \text{ siendo } potPrimos = map\ pots\ (take\ k\ primos)\} \\
 & (aplanar \cdot podar \cdot construirArbol\ potPrimos)\ (0, 1) \\
= & \quad \{\text{definición de } construirArbol, \text{ siendo } ps : pps = potPrimos \text{ (dado que } k > 0)\} \\
 & (aplanar \cdot podar \cdot Nodo\ (0, 1) \cdot map\ (construirArbol\ pps) \cdot valores\ ps)\ (0, 1)
\end{array}
$$

Continuamos simplificando

$$
\begin{array}{ll}
 & aplanar \cdot podar \cdot Nodo\ x \cdot map\ (construirArbol\ pps) \\
= & \quad \{\text{dado que } podar \cdot Nodo\ x = Nodo\ x \cdot tail\} \\
 & aplanar \cdot Nodo\ x \cdot tail \cdot map\ (construirArbol\ pps) \\
= & \quad \{\text{dado que } tail \cdot map\ f = map\ f \cdot tail\} \\
 & aplanar \cdot Nodo\ x \cdot map\ (construirArbol\ pps) \cdot tail \\
= & \quad \{\text{dado que } aplanar \cdot Nodo\ x = (x :) \cdot concat \cdot map\ aplanar\} \\
 & (x :) \cdot concat \cdot map\ aplanar \cdot map\ (construirArbol\ pps) \cdot tail \\
= & \quad \{\text{dado que } map \text{ es un funtor}\} \\
 & (x :) \cdot concat \cdot map\ (aplanar \cdot construirArbol\ pps) \cdot tail
\end{array}
$$

La expresión final sugiere introducir la función *buscar*, definida mediante

$$buscar\ pps = aplanar \cdot construirArbol\ pps$$

Ésta es la función que capta la idea de la deforestación. La construcción y el aplanamiento de un árbol se combinan en una sola función.

El objetivo ahora es obtener una definición recursiva de *buscar*. La simplificación de *buscar* [ ] es inmediata y la de *buscar* (*ps* : *pps*) *x* es similar a la de *aplanar* · *gArbol*. El resultado es

$$
\begin{array}{lcl}
buscar\ [\ ]\ x & = & [x] \\
buscar\ (ps : pps)\ x & = & x : concat\ (map\ (buscar\ ps)\ (valores\ ps\ x))
\end{array}
$$

He aquí el programa final, en el cual suponemos $k > 0$:

$$
\begin{array}{lcl}
meertens\ k = [n \mid (n, g) \leftarrow candidatos\ (0, 1), n == g] & & \\
\quad \text{where} & & \\
\quad candidatos & = & concat \cdot map\ (buscar\ pps) \cdot tail \cdot valores\ ps \\
\quad ps : pps & = & map\ pots\ (take\ k\ primos) \\
\quad buscar\ [\ ]\ x & = & [x] \\
\quad buscar\ (ps : pps)\ x & = & x : concat\ (map\ (buscar\ pps)\ (valores\ ps\ x)) \\
\quad valores\ ps\ (n, g) & = & zip\ (map\ (m +)\ ds)\ (cortar\ (map\ (g \times)\ ps)) \\
 & & \quad \text{where } m = 10 \times n
\end{array}
$$

$$
\begin{array}{lcl}
\quad cortar & = & takeWhile\ (< 10^k) \\
\quad ds & = & [0..9]
\end{array}
$$

Este programa se ejecutó en una máquina pequeña y bastante antigua con una cantidad de memoria limitada:

```
? meertens 8
[81312000]
(2227857 reductions, 5423538 cells, 57 garbage collections)
```

Hasta donde llega nuestro conocimiento, 81312000 es el único número de Meertens conocido. Repetimos que tiene la siguiente propiedad peculiar:

$$2^8 3^1 5^3 7^1 11^2 13^0 17^0 19^0 = 81312000$$

Ejercicios

6.6.1 ¿Cuál es el primer número de Meertens en base cinco?

6.6.2 En la definición de $g(n)$, el dígito más significativo de n se asocia con el primo más pequeño. Existe una variante en la cual el dígito más significativo se asocia con el primo más grande. Definir h tomando $h(n) = 2^{d_k}\, 3^{d_{k-1}} \ldots p_k^{d_1}$, donde $d_1\, d_2 \ldots d_k$ es la representación decimal de n. ¿Cuál es el número n más pequeño para el cual $n = h(n)$?

6.6.3 ¿Existe algún número *amigable* de Meertens, esto es, pares de enteros diferentes m y n tales que $m = g(n)$ y $n = g(m)$? De forma más general, ¿existe algún tipo de grupos cíclicos de números?

6.6.4 Encontrar otro número de Meertens, o ¡demostrar que no existen más!

6.7 Notas del capítulo

Cormen, Leiserson y Rivest (1990) y Knuth (1973b) son buenos libros de consulta para cuestiones relacionadas con los árboles. Las rosadelfas, que también se llaman *árboles generales*, recibieron aquel nombre en Meertens (1987). La tesis de Gibbons (1991) contiene un tratamiento sistemático de los algoritmos sobre árboles.

Los códigos de Huffman están descritos en Huffman (1952); véanse también Hu (1982) y Knuth (1973a). Thompson (1996) presenta asimismo un programa funcional para la codificación de Huffman.

El problema de calcular el número de Meertens se consideró por primera vez en Bird (1991). Los números de Gödel fueron usados por Gödel, en 1931, en la demostración de su famoso teorema sobre la incompletitud de la aritmética; véase Gödel (1990), la colección de todos sus trabajos. Hofstadter (1979) y Penrose (1994) son obras populares de divulgación que discuten la importancia de este resultado en el pensamiento del siglo XX.

Capítulo 7

Eficiencia

A lo largo del capítulo precedente se han ido multiplicando las referencias a la eficiencia y ha llegado el momento de abordar este importante tema. En este capítulo, trataremos más de cerca el modelo subyacente de reducción y los recursos, tanto en tiempo como en espacio, que son requeridos para evaluar expresiones. También describiremos un cálculo simple para estimar el tiempo de ejecución de los programas y presentaremos diversas técnicas de programación, algunas de las cuales ya se han explotado, para mejorar la eficiencia.

7.1 Evaluación perezosa

Comencemos retomando la evaluación de la expresión *cuadrado* (3 + 4) considerada en el Capítulo 1. Recordemos que una posible secuencia de reducción es

$$
\begin{array}{ll}
 & \textit{cuadrado}\ (3+4) \\
= & \quad \{\text{definición de } +\} \\
 & \textit{cuadrado}\ 7 \\
= & \quad \{\text{definición de } \textit{cuadrado}\} \\
 & 7 \times 7 \\
= & \quad \{\text{definición de } \times\} \\
 & 49
\end{array}
$$

Otra posible secuencia de reducción es

$$
\begin{array}{ll}
 & \textit{cuadrado}\ (3+4) \\
= & \quad \{\text{definición de } \textit{cuadrado}\} \\
 & (3+4) \times (3+4) \\
= & \quad \{\text{definición de } +\} \\
 & 7 \times (3+4) \\
= & \quad \{\text{definición de } +\} \\
 & 7 \times 7 \\
= & \quad \{\text{definición de } \times\} \\
 & 49
\end{array}
$$

Estas dos secuencias ilustran las dos estrategias de reducción, llamadas respectivamente *más interna* y *más externa*. En la primera secuencia, cada paso reduce un *redex más interno*. La palabra "redex" es una abreviatura de "expresión reducible"[1] y un redex más interno es aquel que no contiene otro redex. En la segunda secuencia, cada paso reduce un *redex más externo*. Un redex más externo es aquel que no está contenido en otro redex.

He aquí otro ejemplo. Primero, la reducción más interna:

```
      fst (cuadrado 4, cuadrado 2)
=        {definición de cuadrado}
      fst (4 × 4, cuadrado 2)
=        {definición de ×}
      fst (16, cuadrado 2)
=        {definición de cuadrado}
      fst (16, 2 × 2)
=        {definición de ×}
      fst (16, 4)
=        {definición de fst}
      16
```

Esta reducción necesita cinco pasos. En los dos primeros pasos, cabía la posibilidad de elegir entre dos redexes más internos y se eligió el redex situado *más a la izquierda*. La estrategia de reducción más externa para la misma expresión produce

```
      fst (cuadrado 4, cuadrado 2)
=        {definición de fst}
      cuadrado 4
=        {definición de cuadrado}
      4 × 4
=        {definición de ×}
      16
```

Esta secuencia de reducción necesita tres pasos. Usando la estrategia de reducción más externa se ha evitado la evaluación de *cuadrado* 2.

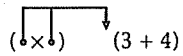
Las dos estrategias de reducción tienen características diferentes. A veces, la reducción más externa dará una respuesta donde la reducción más interna no termina (considérese reemplazar *cuadrado* 2 por *undefined* en la expresión anterior). Sin embargo, si ambos métodos terminan, entonces darán el mismo resultado.

La reducción más externa tiene la propiedad importante de que si una reducción tiene forma normal, entonces la reducción más externa la encontrará. Debido a esta propiedad, la reducción más externa también se llama *orden normal*. Por tanto, podría pensarse que la reducción más externa es mejor que la reducción más interna, pero hay que realizar otras consideraciones. Como muestra el primer ejemplo, la reducción más externa a veces puede necesitar más pasos que la reducción más interna. El problema surge con cualquier función cuya definición contenga apariciones repetidas de un argumento. Vinculando dicho argumento a una expresión convenientemente

[1] N. del T.: En el original, *reducible expression*. Habitualmente se usa sin traducir.

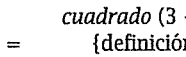
grande, la diferencia entre la reducción más externa y la más interna puede hacerse arbitrariamente grande.

Se puede resolver este problema representando las expresiones como *grafos* en lugar de como árboles. A diferencia de los árboles, los grafos pueden compartir sus expresiones. Por ejemplo, el grafo

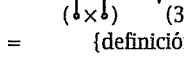


representa la expresión (3 + 4) × (3 + 4). Cada aparición de 3 + 4 se representa mediante una flecha, llamada *puntero*, a una única aparición de (3 + 4). Ahora, utilizando la *reducción de grafos* más externa, tenemos

```
      cuadrado (3 + 4)
=        {definición de cuadrado}
```




```
=        {definición de +}
```



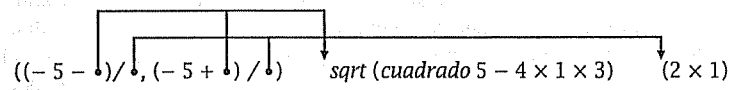

```
=        {definición de ×}
      49
```

La secuencia de reducción consta sólo de tres pasos. La representación de expresiones mediante grafos significa que se pueden compartir y reducir como máximo una vez las subexpresiones duplicadas. Cuando se utiliza la reducción de grafos, la reducción más externa nunca emplea más pasos que la reducción más interna. Por tanto, nos referiremos a la reducción más externa de grafos por su nombre más habitual, *evaluación perezosa*, y a la reducción de grafos más interna como *evaluación impaciente*.

También se pueden introducir subexpresiones compartidas en un grafo mediante definiciones locales. Por ejemplo, dada la definición

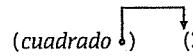
```
raices a b c  =  ((− b − d)/e, (− b + d)/e)
                 where d  =  sqrt (cuadrado b − 4 × a × c)
                       e  =  2 × a
```

el primer paso de reducción de *raices* 1 5 3 produce el grafo

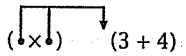


Realmente, existen tres punteros adicionales que no se muestran en este grafo, un puntero para cada una de las variables *a*, *b* y *c*. Estos punteros apuntan a apariciones únicas de las expresiones 1, 5 y 3.

Las flechas no significan nada, tan sólo indican compartición o vínculos. Tanto



como *cuadrado* (3 + 4) son modos equivalentes de escribir la misma expresión. Se puede medir el tamaño de una expresión o de un grafo contando el número total de argumentos involucrados. Por ejemplo, la expresión anterior tiene tamaño tres porque *cuadrado* tiene un argumento y + tiene dos. De forma similar, la expresión *cuadrado* 3 + *cuadrado* 4 tiene tamaño cuatro porque cada aparición de *cuadrado* tiene un argumento y + tiene dos. El grafo



también tiene tamaño cuatro (× y + tienen cada uno dos argumentos).

Adoptaremos la evaluación perezosa como nuestro modelo de cálculo porque tiene dos propiedades deseables: (i) termina siempre que cualquier otro orden de reducción termine, y (ii) no requiere más (sino posiblemente menos) pasos que la evaluación impaciente. Para ser precisos, el tiempo y el espacio requeridos para evaluar una expresión se definen como sigue. Supongamos que

$$e_0 \Rightarrow e_1 \Rightarrow e_2 \Rightarrow \cdots \Rightarrow e_n$$

es una secuencia de reducciones más externas de grafos que calcula la forma normal e_n. Entonces, el tiempo requerido para reducir e_0 a la forma normal se define como n y el espacio requerido se define como el tamaño del mayor grafo presente en la secuencia de reducción, esto es, el máximo de los tamaños de $e_0, \ldots, e_n$. Es razonable escoger el tamaño de la expresión máxima en vez de la suma de los tamaños de todas las expresiónes, porque el espacio utilizado en un paso de reducción siempre puede ser reutilizado en el paso siguiente. Dicho de otra manera, la limitación crítica es la anchura del trozo de papel en el cual se hace el cálculo, no su longitud. Con la ayuda de una goma de borrar (en términos de programación, un *recolector de basura*) se puede recuperar el espacio para los pasos subsiguientes.

El definir de esta manera el tiempo y el espacio necesarios para los cómputos tiene sus limitaciones. En particular, el número de pasos de reducción no se corresponde exactamente con el tiempo que transcurre desde que se somete a evaluación una expresión hasta que se obtiene el resultado. No se ha tenido en cuenta el tiempo requerido para encontrar un redex más externo en una expresión posiblemente complicada. Dado que las expresiones pueden ser arbitrariamente grandes, no se puede suponer que este tiempo de búsqueda sea constante. La medida del espacio tiene también sus limitaciones. Un problema es que no se tiene en cuenta el proceso de *impresión*. Para ilustrar este punto, pensemos que para imprimir la lista [1..1000] no se necesitan 1000 unidades de espacio, incluso considerando que la reducción de la expresión [1..1000] a su forma normal 1:2:...:1000:[ ] requiere esta cantidad. Imprimir una lista exige imprimir la cabeza de la lista, desechar dicha cabeza, e imprimir el resto. Sin embargo, la situación se complica por otro aspecto. Supongamos que definimos

$$numeros \;=\; [1..1000]$$

en un guión. Esta vez, la evaluación de *numeros* usa 1000 unidades de espacio. El espacio no se puede desechar después de imprimir porque existe un puntero *numeros* en el programa. Una consecuencia de la evaluación perezosa es que ninguna expresión al final de una flecha se evalúe más de una vez. Por tanto, la primera evaluación de *numeros* deja *numeros* vinculado a una lista evaluada de longitud 1000. Como

resultado, el espacio ocupado por un guión puede crecer con las evaluaciones repetidas de las funciones que contiene.

Por varias razones, nos centraremos casi exclusivamente en el análisis de la eficiencia en tiempo de los programas. Sin embargo, en la Sección 7.5 se da una técnica con limitaciones para un análisis del espacio.

Ejercicios

7.1.1 Describir las secuencias de reducción más interna, más externa, y más externa para grafos para cada una de las expresiones siguientes:

$$\begin{array}{ll} head\,[1..1000] & \\ map\,f\,[1..3] & \textbf{where } f\,n = 10 \times 10 + n \end{array}$$

7.1.2 Una definición posible de *ordenar* es *ordenar* = *foldr insert* [ ], donde

$$\begin{array}{lcl} insert\,x\,[\,] & = & [x] \\ insert\,x\,(y:ys) & = & \textbf{if } x \le y \textbf{ then } x:y:ys \textbf{ else } y:insert\,x\,ys \end{array}$$

Dar en detalle las secuencias de reducción impaciente y perezosa para la expresión *ordenar* [3, 4, 2, 1], explicando dónde difieren.

7.1.3 Supongamos que definimos *minlist* = *head* · *ordenar*, donde *ordenar* se ha definido en el ejercicio anterior. ¿Cuántos pasos de reducción se requieren (aproximadamente) para evaluar *minlist*, bajo evaluación perezosa e impaciente, sobre una lista de n números?

7.1.4 Misma cuestión, para *maxlist* = *last* · *ordenar*.

7.2 Estudio asintótico del coste

En general, estamos menos interesados en estimar el coste de evaluar una expresión particular que en comparar las diferencias de eficiencia de una definición de función con respecto a otra. Por ejemplo, consideremos los dos programas siguientes para invertir una lista:

$$\begin{array}{lcl} reverse\,[\,] & = & [\,] \\ reverse\,(x:xs) & = & reverse\,xs +\!\!+ [x] \end{array}$$

$$reverse' \;=\; foldl\,prefijar\,[\,] \quad \textbf{where } prefijar\,xs\,x = x:xs$$

En la Sección 4.5 se afirmó que el segundo programa es más eficiente que el primero, necesitando como mucho un número de pasos proporcional a n para una lista de longitud n, mientras que el primer programa necesita n^2 pasos. El objetivo de esta sección es mostrar cómo se pueden hacer más precisas tales afirmaciones y cómo justificarlas.

7.2.1 La notación-O

Para evitar explicar notaciones matemáticas especiales, se ha utilizado la críptica frase "necesita como máximo un número de pasos proporcional a" siempre que hemos hablado de eficiencia. Ha llegado el momento de reemplazarla por algo más breve. La notación-O se describe en muchos libros dedicados a las matemáticas de la programación; por tanto, nos contentaremos con recordar las definiciones básicas y dar algunos ejemplos.

Dadas dos funciones f y g sobre números naturales, decimos que *f está en el orden de*[2] g, o que *f es* $O(g)$, y lo escribimos $f = O(g)$, si existe una constante positiva C y un número natural n_0 tales que $f(n) \leq Cg(n)$ para todo $n \geq n_0$. En otras palabras, f está acotada superiormente por g multiplicada por una constante para valores de n suficientemente grandes. Por abuso de notación, se escribe convencionalmente, por ejemplo, $f(n) = O(n^2)$ en lugar de la expresión más correcta $f = O(cuadrado)$. Del mismo modo, escribimos $f(n) = O(n)$ en lugar de $f = O(id)$.

El interés principal de la notación-O es ignorar las constantes muliplicativas; por ejemplo, podemos escribir

$$\sum_{j=1}^{n} j = O(n^2) \quad \text{y} \quad \sum_{j=1}^{n} j^2 = O(n^3)$$

sin preocuparnos de qué constantes exactas están implicadas. Sin embargo, la notación-O debe utilizarse con cuidado. Si bien escribimos $f = O(g)$, realmente no queremos decir que f es igual a $O(g)$. Es mejor pensar que $O(g)$, considerado aisladamente en la parte derecha de una ecuación, denota el *conjunto* de funciones f que satisfacen la propiedad descrita anteriormente, e interpretar el signo igual de $f = O(g)$ como pertenencia a un conjunto. Así nunca estaremos tentados de escribir $O(g) = f$, o de concluir que $f = g$ a partir de las afirmaciones $f = O(h)$ y $g = O(h)$.

Sin embargo, escribiremos expresiones como las siguientes:

$$O(n) + O(n), \qquad \sum_{n=1}^{N} O(n^2), \; T(n) = 2T(n/2) + O(n)$$

Cuando $O(g)$ aparece en una fórmula se refiere a alguna función no nombrada f que satisface $f = O(g)$. En particular, $O(n) + O(n)$ denota la suma de dos funciones anónimas f_1 y f_2, siendo $f_1(n) = O(n)$ y $f_2(n) = O(n)$. Dado que se cumple que $f_1(n) + f_2(n) = O(n)$, es correcto concluir que $O(n) + O(n) = O(n)$.

Por otra parte, la expresión $\sum_{n=1}^{N} O(n^2)$ *no* significa

$$O(1^2) + O(2^2) + \cdots + O(N^2)$$

En lugar de ello, denota la suma $\sum_{n=1}^{N} f(n)$, donde f es una función *cualquiera* que satisface $f(n) = O(n^2)$. En este caso, tenemos que $\sum_{n=1}^{N} O(n^2) = O(N^3)$.

Lo que proporciona la notación-O es una cota superior al crecimiento *asintótico* de las funciones. Por esta razón, el empleo de la notación-O para estimar la eficiencia de un programa recibe el nombre de *análisis asintótico de la cota superior*. Por ejemplo, la complejidad en tiempo de *reverse* aplicada a una lista de longitud n es $O(n^2)$, y la

[2] N. del T.: En el original, *f is order at most g*. La traducción elegida es coherente con el hecho, explicado más adelante, de que $O(g)$ representa un conjunto de funciones.

complejidad de *reverse'* es $O(n)$. Sin embargo, decir que *reverse* necesita $O(n^2)$ pasos para una lista de longitud n no significa que no requiera $O(n)$ pasos. Para conseguir más precisión, necesitaremos alguna notación adicional.

Diremos que f está en el orden de *al menos g*, o que f es $\Omega(g)$, y lo escribiremos $f = \Omega(g)$, si existe una constante positiva C y un número natural n_0 tales que $f(n) \geq Cg(n)$ para todo $n \geq n_0$. Dado que no exigimos que C sea un número entero, tenemos que $f = O(g)$ si y sólo si $g = \Omega(f)$. Ahora, decir que *reverse* requiere $\Omega(n^2)$ pasos para una lista de longitud n es poner una cota inferior al número de pasos requerido para evaluar *reverse* sobre cualquier lista de longitud n.

Poniendo juntas las dos clases de cotas, diremos que *f está en el orden exacto* de g, o que f es $\Theta(g)$, y lo escribiremos $f = \Theta(g)$, si $f = O(g)$ y $f = \Omega(g)$. En otras palabras, $f = \Theta(g)$ si existen dos constantes positivas C_1 y C_2 tales que

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

para todo n suficientemente grande. Entonces podemos afirmar que el tiempo de *reverse* es $\Theta(n^2)$ y el tiempo de *reverse'* es $\Theta(n)$. Estas dos funciones son atípicas en el sentido de que normalmente las cotas superiores e inferiores de los tiempos de ejecución no coinciden. En general, el tiempo requerido para evaluar una función f sobre una lista de longitud n dependerá no sólo de n sino también de los elementos concretos de la lista. Para algunas listas, la evaluación puede ser rápida, mientras que para otras la evaluación puede ser lenta. Por este motivo, sólo podremos hacer afirmaciones del estilo de "el tiempo de ejecución de f es $\Omega(g)$ y $O(h)$". Si queremos más información, debemos considerar por separado el caso mejor y el caso peor (y, quizá, el caso promedio). Entonces podremos hacer afirmaciones del estilo de "en el caso peor, el tiempo de ejecución de f es $\Theta(n^2)$".

7.2.2 Análisis del tiempo

Dada una función f escribiremos $T(f)(n)$ para denotar una estimación asintótica del número de pasos de reducción requeridos en el peor caso para evaluar f sobre un argumento de "tamaño" n. Más aún, por razones que explicaremos enseguida, asumiremos la evaluación impaciente, no la perezosa, como estrategia de reducción. En particular, podemos escribir

$$\begin{aligned} T(reverse)(n) &= \Theta(n^2) \\ T(reverse')(n) &= \Theta(n) \end{aligned}$$

La definición de T requiere alguna extensión. En primer lugar, $T(f)$ no se refiere a la complejidad en tiempo de una función f sino a la complejidad de una *definición* dada de f. La complejidad en tiempo es una propiedad de una expresión, no del valor de esa expresión.

En segundo lugar, no formalizamos la noción de tamaño, porque para diferentes situaciones convienen diferentes medidas. Por ejemplo, el coste de evaluar xs ++ ys se mide mejor en términos de m y n, siendo $m = length\ xs$ y $n = length\ ys$. De hecho, se cumple

$$T(++)(m, n) = \Theta(m)$$

Se deja la demostración como ejercicio. Consideremos ahora *concat xss*. Aquí el tamaño de *xss* es más difícil de definir. En el caso simple en el que *xss* es una lista de longitud m compuesta de listas todas ellas de longitud n, tenemos

$$T(concat)(m, n) = \Theta(mn)$$

Demostraremos este resultado más adelante. La estimación para $T(concat)$ se refiere, por tanto, exclusivamente a listas de listas con una longitud común a todas ellas; aunque lo limitan, tales restricciones hacen que el análisis del tiempo sea más tratable.

La tercera observación consiste en detacar que $T(f)(n)$ es exclusivamente una estimación para el caso del peor tiempo de ejecución. Esto será suficiente para nuestros propósitos, aunque en la práctica también son importantes los análisis del mejor caso y del caso promedio.

La cuarta y crucial observación es que $T(f)(n)$ determina bajo un modelo impaciente de evaluación. La razón es simplemente que estimar el número de pasos de reducción bajo evaluación perezosa es difícil y todavía es un tema de investigación abierto. Para ilustrarlo, consideremos la definición

$$minlist = head \cdot ordenar$$

Bajo evaluación impaciente, el tiempo para evaluar *minlist* sobre una lista de longitud n viene dado por

$$T(minslist)(n) = T(ordenar)(n) + T(head)(n)$$

En otras palabras, para calcular *minlist* sobre una lista de longitud n, primero ordenamos la lista, lo que necesita $T(ordenar)(n)$ pasos de reducción, y después calculamos la cabeza del resultado (que es también una lista de longitud n), lo que necesita $T(head)(n)$ pasos. Esta ecuación no se cumple bajo evaluación perezosa, dado que el número de pasos de reducción requeridos para encontrar la cabeza de *ordenar xs* no implica necesariamente que *ordenar xs* haya de ser reducida a forma normal. Ello depende del algoritmo preciso usado para *ordenar*. El análisis de tiempo bajo evaluación impaciente es más simple porque es composicional. Dado que la evaluación perezosa nunca necesita más pasos de reducción que la evaluación impaciente, cualquier cota superior para $T(f)(n)$ será también una cota superior bajo evaluación perezosa. Más aún, en muchos casos de interés, una cota inferior para $T(f)(n)$ será también una cota inferior bajo evaluación perezosa.

7.2.3 Las funciones *reverse* y *concat*

Analicemos ahora el programa para *reverse*. Tenemos

$$\begin{aligned} T(reverse)(0) &= O(1) \\ T(reverse)(n+1) &= T(reverse)(n) + T(++)(n, 1) \end{aligned}$$

La expresión $O(1)$ denota una constante, cuyo valor preciso no tiene interés. Dado que $\Theta(1) = O(1)$, podríamos haber usado $\Theta(1)$ en la primera ecuación. La segunda ecuación dice que para invertir una lista de longitud $n + 1$ hemos de invertir una lista de longitud n y después concatenar con ella una lista de longitud uno. La ecuación explota la propiedad de que *reverse* no cambia la longitud de la lista. El análisis de programas depende de hechos como éste, razón por la cual no se trata simplemente de un ejercicio mecánico cuya realización puede ser delegada a un computador.

La ecuación recursiva para $T(reverse)$ es simple de resolver. Utilizando la estimación $T(++)(n, m) = \Theta(n)$, obtenemos

$$T(reverse)(n) = \sum_{k=0}^{n} \Theta(k) = \Theta(n^2)$$

Para el análisis de la segunda función *reverse'* podemos proceder de dos maneras diferentes. La primera es eliminar *foldl* en favor de una definición recursiva directa:

$$\begin{aligned} reverse'\ xs &= acum\ [\,]\ xs \\ acum\ ws\ [\,] &= ws \\ acum\ ws\ (x : xs) &= acum\ (x : ws)\ xs \end{aligned}$$

La función *acum* tiene dos argumentos. Si denotamos las longitudes de los dos argumentos por m y n, entonces

$$\begin{aligned} T(acum)(m, 0) &= O(1) \\ T(acum)(m, n+1) &= O(1) + T(acum)(m, n) \end{aligned}$$

El término $O(1)$ en la segunda ecuación es debido a la operación *cons*. La solución es $T(acum)(m, n) = \Theta(n)$ y por tanto $T(reverse')(n) = \Theta(n)$.

El segundo método es menos formal. Evaluar *foldl f e* sobre una lista de longitud n requiere calcular n veces f. Si f es una operación que necesita tiempo constante, tal como *prefijar*, obtenemos de una vez que el tiempo total es de $\Theta(n)$ pasos.

Como ejemplo adicional, la función *concat* se puede definir directamente mediante

$$\begin{aligned} concat\ [\,] &= [\,] \\ concat\ (xs : xss) &= xs +\!\!+ concat\ xss \end{aligned}$$

Para plantear el análisis del tiempo, sea el argumento de *concat* una lista de longitud m que consta de listas tales que cada una de ellas es de longitud n. La longitud de la lista resultante es mn. Entonces tenemos

$$\begin{aligned} T(concat)(0, n) &= O(1) \\ T(concat)(m+1, n) &= T(concat)(m, n) + T(++)(n, mn) \end{aligned}$$

Dado que $T(++)(n, mn) = \Theta(n)$, obtenemos

$$T(concat)(m, n) = \sum_{k=0}^{m} \Theta(n) = \Theta(mn)$$

La definición anterior de *concat* es exactamente la que obtendríamos a partir de la definición *concat* = *foldr* (++) [ ] eliminando la llamada a *foldr*. Como ++ es asociativo y su elemento neutro es [ ], podríamos haber definido también *concat* mediante *concat* = *foldl* (++) [ ]. Eliminando *foldl* tal como hicimos en la definición de *reverse'* obtenemos el programa

$$\begin{aligned} concat\ xss &= acum\ [\,]\ xss \\ acum\ ws\ [\,] &= wss \\ acum\ ws\ (xs : xss) &= acum\ (ws +\!\!+ xs)\ xss \end{aligned}$$

Para el análisis de *acum* supongamos que la primera lista tiene longitud k y que la segunda es una lista de m listas de longitud n. Entonces tenemos

$$\begin{aligned} T(acum)(k, 0, n) &= O(1) \\ T(acum)(k, m+1, n) &= T(acum)(k+n, m, n) + T(++)(k, n) \end{aligned}$$

Dado que $T(++)(k, n) = \Theta(k)$, obtenemos

$$T(acum)(k, m, n) = \sum_{j=0}^{m-1} \Theta(k + jn) = \Theta(k + m^2\, n)$$

Por tanto, para esta definición de *concat* tenemos que $T(concat)(m, n) = \Theta(m^2\, n)$. La conclusión es que si utilizamos *foldr* en lugar de *foldl* en la definición de *concat* conseguimos un programa asintóticamente más rápido.

Ejercicios

7.2.1 Demostrar que $f = O(g)$ si y sólo si $g = \Omega(f)$.

7.2.2 ¿Qué significa exactamente $O(1)$? Demostrar que $f = O(1)$ si y sólo si $f = \Theta(1)$.

7.2.3 Demostrar que $n^{100} = O(2^n)$.

7.2.4 Utilizando la notación-O, estimar $\sum_{k=0}^{n} k2^k$.

7.2.5 Demostrar que $T(++)(m, n) = \Theta(m)$.

7.2.6 Demostrar por inducción que si $k \geq 2$ y

$$\begin{aligned} T(0) &= O(1) \\ T(n+1) &= kT(n) + \Theta(k^{n+1}) \end{aligned}$$

entonces $T(n) = \Theta(nk^n)$. ¿Cuál es la solución si $k = 1$?

7.3 Parámetros acumuladores

A veces podemos mejorar el tiempo de ejecución añadiendo un parámetro adicional a una función. El uso más importante de esta idea consiste en eliminar posibles operaciones costosas ++ de un programa. Para mostrar un ejemplo básico, consideremos la función *revConcat* definida mediante

```
revConcat        :: [α] → [α] → [α]
revConcat xs ys  =  reverse xs ++ ys
```

Es obvio que *reverse xs* = *revConcat xs* [ ], por tanto podemos definir *reverse* en términos de *revConcat*. Es sencillo derivar un programa recursivo directo para *revConcat*, siendo el resultado

```
revConcat [ ] ys       =  ys
revConcat (x : xs) ys  =  revConcat xs (x : ys)
```

Dado que *revConcat xs ys* = *acum ys xs*, donde *acum* fue definido en la sección precedente, es obvio, a partir del análisis de tiempo de *acum*, que el nuevo programa para *reverse* emplea tiempo lineal.

Para apreciar realmente lo que sucede, es instructivo escribir el programa para *revConcat* en términos de composición de funciones:

```
revConcat [ ]      =  id
revConcat (x : xs) =  revConcat xs · (x :)
```

De modo equivalente, *revConcat* = *foldr f*, siendo $f\ x\ g = g \cdot (x :)$. Aplicando exactamente el mismo método, obtenemos

```
(++)  =  foldr consPunto id where consPunto x g = (x :) · g
```

Es esta expresión de ++ la que subyace en la mayoría de los usos de la técnica de parámetros acumuladores: la concatenación queda reemplazada por la composición de operaciones *cons*. Por ejemplo, podemos argumentar que

```
    reverse xs
=     {dado que [ ] es el elemento neutro de ++}
    (++)(reverse xs) [ ]
=     {expresión anterior de ++}
    (foldr consPunto id (reverse xs) [ ]
=     {tercer teorema de dualidad}
    foldl (flip consPunto) id xs [ ]
```

Esto nos da otro programa de tiempo lineal para *reverse* que es esencialmente el mismo de antes, excepto por el hecho de que *revConcat* se expresa en términos de *foldl* en lugar de en términos de *foldr*.

7.3.1 Aplanando un árbol binario

He aquí otro ejemplo: la función *aplanar* para aplanar un árbol binario se definió en la Sección 6.1 mediante

```
aplanar            :: ArBin α → [α]
aplanar (Hoja x)   =  [x]
aplanar (Unir xt yt) =  aplanar xt ++ aplanar yt
```

Ahora definimos

```
apConcat        :: ArBin α → [α] → [α]
apConcat xt xs  =  aplanar xt ++ xs
```

Tenemos que *aplanar xt* = *apConcat xt* []. La síntesis de un programa recursivo para *apConcat* nos da

```
apConcat (Hoja x) xs      =  x : xs
apConcat (Unir xt yt) xs  =  apConcat xt (apConcat yt xs)
```

Comparemos ahora los tiempos de ejecución de *aplanar* y *apConcat*. Sea $T(aplanar)(h)$ la estimación del tiempo requerido para aplanar un árbol binario perfecto de altura h. Recordemos del capítulo precedente que un árbol tal tiene 2^h hojas. Tenemos

$$\begin{aligned} T(aplanar)(0) &= O(1) \\ T(aplanar)(h+1) &= 2T(aplanar)(h) + T(++)(2^h, 2^h) \end{aligned}$$

La segunda ecuación se satisface porque los dos subárboles de un árbol binario perfecto de altura $h + 1$ tienen ambos tamaño 2^h.

Dado que $T(++)(m, n) = \Theta(m)$, obtenemos

$$T(aplanar)(h + 1) = 2T(aplanar)(h) + \Theta(2^h)$$

Una demostración por inducción nos da $T(aplanar)(h) = \Theta(h2^h)$; por tanto, *aplanar* emplea $O(s \log s)$ pasos sobre un árbol binario perfecto de tamaño s.

Sea ahora $T(apConcat)(h, n)$ la estimación del tiempo requerido para calcular la expresión *apConcat xt xs*, donde *xt* es un árbol binario perfecto de altura h, y *xs* es una lista de longitud n. Entonces tenemos

$$\begin{aligned} T(apConcat)(0, n) &= O(1) \\ T(apConcat)(h + 1, n) &= O(1) + T(apConcat)(h, 2^h + n) + T(apConcat)(h, n) \end{aligned}$$

Es fácil demostrar por inducción que $T(apConcat)(h, n) = \Theta(2^h)$; por tanto, el nuevo programa para *aplanar* emplea tiempo lineal en el tamaño del árbol.

7.3.2 Recorrido en profundidad

Se puede aplicar exactamente la misma técnica a la función *aplanar* de las rosadelfas. Aquí la definición de *aplanar* es

```
aplanar              :: Rosa α → [α]
aplanar (Nodo x xts) =  x : concat (map aplanar xts)
```

Esta vez, vamos a analizar el programa para *aplanar* antes de tratar de mejorarlo. Sea $T(aplanar)(h, k)$ la estimación del tiempo necesario para aplanar un árbol k-ario perfecto de altura $(h + 1)$. Recordemos que un árbol de estas características tiene por tamaño

$$s(h, k) = (k^{h+1} - 1) / (k - 1) = \Theta(k^h)$$

Ahora tenemos

$$\begin{aligned} T(aplanar)(0, k) &= O(1) \\ T(aplanar)(h + 1, k) &= O(1) + T(concat)(k, s) + T(map\ aplanar)(h, k) \end{aligned}$$

donde $s = s(h, k)$. El término $T(map\ aplanar)(h, k)$ estima el tiempo de ejecución de *map aplanar* sobre una lista de k árboles de altura h. En general, dada una lista de longitud k, consistente en elementos todos ellos de tamaño n, tenemos que

$$T(map\ f)(k, n) = kT(f)(n) + \Theta(k)$$

Por tanto, dado que $O(1) + \Theta(k) = \Theta(k)$, obtenemos

$$T(aplanar)(h + 1, k) = T(concat)(k, s) + kT(aplanar)(h, k) + \Theta(k)$$

Puesto que $T(concat)(k, s) = \Theta(ks) = \Theta(k^{h+1})$, obtenemos

$$T(aplanar)(h + 1, k) = \Theta(k^{h+1}) + kT(aplanar)(h, k)$$

Podemos demostrar, por inducción, que $T(aplanar)(h, k) = \Theta(hk^h) = \Theta(s \log s)$. Por tanto, para aplanar un árbol *aplanar* se necesita un tiempo que es asintóticamente mayor que el tamaño del árbol.

Una vez establecido que el método precedente para aplanar un árbol no es el mejor posible, veamos qué podemos hacer mediante un parámetro acumulador. Definimos *apConcat* mediante

```
apConcat          :: Rosa α → [α] → [α]
apConcat xts xs   =  concat (map aplanar xts) ++xs
```

Tenemos que *aplanar xt* = *apConcat* [*xt*] [ ]. El objetivo ahora es sintetizar un programa recursivo para *apConcat*. He aquí el caso inducctivo:

```
    apConcat (Nodo x xts : yts) xs
=       {definición}
    concat (map aplanar (Nodo x xts : yts)) ++xs
=       {definición de map y concat}
    aplanar (Nodo x xts) ++(map aplanar yts) ++xs
=       {definición de aplanar}
    x : concat (map aplanar xts) ++concat (map aplanar yts) ++xs
=       {definición de apConcat}
    x : concat (map aplanar xts) ++apConcat yts xs
=       {definición de apConcat}
    x : apConcat xts (apConcat yts xs)
```

El resultado es el siguiente programa para *apConcat*:

```
apConcat [ ] xs              =  xs
apConcat (Nodo x xts : yts) xs  =  x : apConcat xts (apConcat yts xs)
```

Para realizar el análisis del tiempo, sea $T(apConcat)(h,n)$ la estimación del tiempo requerido para calcular *apConcat xts xs*, siendo *xts* una lista de n árboles k-arios perfectos de altura $(h+1)$. Pasaremos por alto el tamaño de *xs* ya que no afecta a la estimación.

Tenemos

$$\begin{aligned} T(apConcat)(h, 0) &= O(1) \\ T(apConcat)(0, n+1) &= O(1) + T(apConcat)(0, n) \\ T(apConcat)(h+1, n+1) &= O(1) + T(apConcat)(h, k) + T(apConcat)(h+1, n) \end{aligned}$$

La primera estimación contabiliza el coste de evaluar la primera ecuación de *apConcat*, la segunda contabiliza el coste de evaluar la segunda ecuación de *apConcat* cuando el primer elemento de la primera lista es un árbol de altura 1 (y por tanto no tiene subárboles), y la tercera estimación contabiliza los restantes costes.

Las dos primeras ecuaciones dan $T(apConcat)(0, n) = \Theta(n)$. Una demostración por inducción da $T(apConcat)(h, n) = \Theta(k^h\ n)$, por tanto $T(apConcat)(h, 1) = \Theta(k^h) = \Theta(s)$. Utilizando un parámetro acumulador se puede aplanar un árbol en un tiempo proporcional al tamaño del árbol.

7.3.3 La clase *Show*

Como ejemplo final de utilización de parámetros acumuladores, consideremos la clase de tipos *Show*. Recordemos de las Secciones 2.7 y 5.3 que la declaración de clase para *Show* contiene el método

```
showsPrec :: Int → α → String → String
```

Ahora podemos decir qué es el tercer argumento de *showsPrec*: es un parámetro acumulador. El uso de un parámetro acumulador es necesario si queremos imprimir tipos de datos no lineales en un tiempo lineal.

Ejercicios

7.3.1 Se puede definir la función *digitos* :: *Int* → [*Digito*] mediante

$$digitos\ n \quad = \quad \textbf{if}\ n < 10\ \textbf{then}\ [n]\ \textbf{else}\ digitos\ (n\ \textbf{div}\ 10) +\!\!+ [n\ \textbf{mod}\ 10]$$

¿Cuánto tarda el cálculo *digitos n*? Mejorar el tiempo de ejecución utilizando un parámetro acumulador.

7.3.2 Consideremos la función *unfoldl* definida mediante

$$\begin{array}{lcl} unfoldl & :: & (\alpha \to Bool) \to (\alpha \to (\alpha, \beta)) \to \alpha \to [\beta] \\ unfoldl\ p\ f\ x & = & \textbf{if}\ p\ x\ \textbf{then}\ [\ ]\ \textbf{else}\ unfoldl\ p\ f\ y +\!\!+ [z] \\ & & \textbf{where}\ (y, z) = f\ x \end{array}$$

Utilizar un parámetro acumulador para dar una definición más eficiente.

7.4 Tuplamiento

La técnica de optimización de programas conocida como *tuplamiento*[3] es dual de la de los parametros acumuladores: se generaliza una función, no incluyendo un argumento extra sino incluyendo un resultado adicional. Nuestro objetivo en esta sección es ilustrar esta importante técnica por medio de diversos ejemplos instructivos.

7.4.1 Construyendo un árbol

Consideremos la función *construirArBin* definida en la Sección 6.1 para construir un árbol de altura mínima a partir de una secuencia no vacía de hojas:

$$\begin{array}{l} construirArBin :: [\alpha] \to ArBin\ \alpha \\ construirArBin\ xs \\ \quad = \ \textbf{if}\ m == 0\ \textbf{then}\ Hoja\ (head\ xs)\ \textbf{else}\ Unir\ (construirArBin\ ys)\ (construirArBin\ zs) \\ \quad\ \ \textbf{where}\ ys = take\ m\ xs \\ \qquad\qquad zs = drop\ m\ xs \\ \qquad\qquad m = (length\ xs)\ \textbf{div}\ 2 \end{array}$$

Para analizar este programa, supongamos que *xs* es una lista de longitud 2^k. El tiempo para calcular *length xs*, *take m xs* y *drop m xs*, siendo $m = 2^k - 1$, es por tanto $\Theta(2^k)$. Como consecuencia de esto se tiene que

$$\begin{array}{lcl} T(construirArBin)(0) & = & O(1) \\ T(construirArBin)(k+1) & = & 2T(construirArBin)(k) + \Theta(2^k) \end{array}$$

[3]N. del T.: En el original *tupling*.

La solución es $T(construirArBin)(k) = \Theta(k2^k)$. Por tanto, para construir un árbol de altura mínima a partir de una lista de longitud n se necesitan $\Theta(n \log n)$ pasos. Consideremos ahora la función *construirDos* definida mediante

$$\begin{array}{lcl} construirDos & :: & Int \to [\alpha] \to (ArBin\ \alpha, [\alpha]) \\ construirDos\ n\ xs & = & (construirArBin\ (take\ n\ xs), drop\ n\ xs) \end{array}$$

Es obvio que *construirArBin xs* = *fst* (*construirDos* (*length xs*) *xs*); por tanto, se puede calcular *construirArBin* calculando *construirDos*. Ahora, el objetivo es dar un programa recursivo para *construirDos*. Es fácil calcular

$$construirDos\ 1\ xs \quad = \quad (Hoja\ (head\ xs), tail\ xs)$$

Para el caso recursivo, necesitaremos las leyes

$$\begin{array}{lcl} take\ m \cdot take\ n & = & take\ m \\ drop\ m \cdot take\ n & = & take\ (n-m) \cdot drop\ m \\ drop\ n & = & drop\ m \cdot drop\ (n-m) \end{array}$$

que se satisfacen si $m \leq n$. Ahora, para $n > 1$ razonamos como sigue:

$$\begin{array}{ll} & construirDos\ n\ xs \\ = & \quad \{\text{definición de } construirDos\} \\ & (construirArBin\ (take\ n\ xs), drop\ n\ xs) \\ = & \quad \{\text{definición de } construirArBin, \text{ usando que } m = n\ \textbf{div}\ 2\} \\ & (Unir\ (construirArBin\ (take\ m\ xs)) \\ & \qquad (construirArBin\ (take\ (n-m)(drop\ m\ xs))), drop\ n\ xs) \\ = & \quad \{\text{defininiendo } (xt, ys) = construirDos\ m\ xs \text{ y utilizando la tercera ley}\} \\ & (Unir\ xt\ (construirArBin\ (take\ (n-m)\ ys)), drop\ (n-m)\ ys) \\ = & \quad \{\text{definiendo } (yt, zs) = construirDos\ (n-m)\ ys\} \\ & (Unir\ xt\ yt, zs) \end{array}$$

Resumiendo:

$$\begin{array}{l} construirDos\ n\ xs \\ \quad = \ \textbf{if}\ n == 1\ \textbf{then}\ (Hoja\ (head\ xs), tail\ xs)\ \textbf{else}\ (Unir\ xt\ yt, zs) \\ \quad\ \ \textbf{where}\ (xt, ys) = construirDos\ m\ xs \\ \qquad\qquad (yt, zs) = construirDos\ (n-m)\ ys \\ \qquad\qquad m = n\ \textbf{div}\ 2 \end{array}$$

En el caso recursivo, el análisis de la complejidad en tiempo de *construirDos* da

$$\begin{array}{lcl} T(construirDos)(n) & = & T(construirDos)(m) + T(construirDos)(n-m) + O(1) \\ & & \textbf{where}\ m = n\ \textbf{div}\ 2 \end{array}$$

La solución es $T(construirDos)(n) = \Theta(n)$. Utilizando *construirDos* como función auxiliar, hemos mejorado la complejidad asintótica de *construirArBin* por un factor logarítmico.

7.4.2 Función de Fibonacci

Otro ejemplo donde la técnica de tuplamiento puede mejorar el orden de complejidad en tiempo de un programa lo tenemos en la función de Fibonacci:

$$fib\ 0 \quad = \quad 0$$

$$\begin{aligned} fib\ 1 &= 1 \\ fib\ (n+2) &= fib\ n + fib\ (n+1) \end{aligned}$$

El tiempo para evaluar *fib n* mediante estas ecuaciones viene dado por $T(fib)(n)$, donde

$$\begin{aligned} T(fib)(0) &= O(1) \\ T(fib)(1) &= O(1) \\ T(fib)(n+2) &= T(fib)(n) + T(fib)(n+1) + O(1) \end{aligned}$$

La función $T(fib)$ satisface por tanto ecuaciones muy similares a las de *fib*. Es fácil comprobar por inducción que $T(fib)(n) = \Theta(fib\ n)$, por tanto el tiempo para calcular *fib* es proporcional al tamaño del resultado. Dado que $fib\ n = \Theta(\Phi^n)$, donde Φ es la razón áurea $\Phi = (1+\sqrt{5})/2$, el tiempo es exponencial en n.

Consideremos ahora la función *fibDos* definida mediante

$$fibDos\ n = (fib\ n, fib\ (n+1))$$

Claramente, $fib\ n = fst\ (fibDos\ n)$. La síntesis de un programa recursivo para *fibDos* proporciona

$$\begin{aligned} fibDos\ 0 &= (0, 1) \\ fibDos\ (n+1) &= (b, a+b) \quad \textbf{where } (a, b) = fibDos\ n \end{aligned}$$

Es obvio que este programa emplea tiempo lineal. En este ejemplo, la estrategia de tuplamiento conduce a una mejora drástica en la eficiencia, de exponencial a lineal.

7.4.3 Llenando párrafos

Los ejemplos anteriores tuplaban dos funciones, pero se puede utilizar la misma idea con buenos resultados tuplando una *lista* de funciones. En esta versión, la técnica de tuplamiento recibe usualmente el nombre de *tabulación*. El problema de llenar párrafos, un problema que consideramos en la Sección 5.5, proporciona un ejemplo sencillo. Se nos da una lista *ws* de palabras y queremos calcular una lista de líneas *wss* tal que *ws* = *concat wss*, con la restricción adicional de que cada línea de *wss* debería tener como máximo una longitud dada y además *wss* debe minimizar cierta noción de desperdicio. Para simplificar, supondremos que se determina el desperdicio mediante una función numérica dada *desp* que toma el valor ∞ si el párrafo contiene una línea que no cabe en la anchura dada. El algoritmo de la Sección 5.5 es voraz por naturaleza: si en un momento del cómputo la lista remanente de palabras es *ws*, entonces la línea siguiente consta del mayor segmento inicial de *ws* que cabe en la anchura dada. Por tanto, resolvemos el problema general seleccionando la mejor posibilidad para la línea siguiente, no necesariamente el segmento válido inicial más largo de la entrada remanente.

La definición de *llenar* toma la forma

$$\begin{aligned} llenar &:: [Palabra] \to [Linea] \\ llenar &= mejor \cdot llenados \end{aligned}$$

$$\begin{aligned} mejor &:: [[Linea]] \to [Linea] \\ mejor &= foldl1\ mejorQue \\ & \quad \textbf{where } mejorQue\ uss\ vss = \textbf{if } desp\ uss \le desp\ vss \textbf{ then } uss \textbf{ else } vss \end{aligned}$$

La función *llenados* da todos los modos posibles de llenar el párrafo:

$$\begin{aligned} llenados &:: [Palabra] \to [[Linea]] \\ llenados\ [\,] &= [[\,]] \\ llenados\ (w : ws) &= [us : vss \mid (us, vs) \leftarrow descomp\ (w : ws),\ vs \leftarrow llenados\ vs] \end{aligned}$$

Se especifica la función *descomp* mediante la condición de que si *xs* es una lista no vacía, entonces *descomp xs* devuelve la lista de todos los pares (us, vs) tales que $us +\!\!+ vs = xs$ y *us* es no vacía. Se puede definir la función *descomp* de varias maneras; una de ellas es la siguiente:

$$\begin{aligned} descomp &:: [\alpha] \to [([\alpha], [\alpha])] \\ descomp\ (w : ws) &= zip\ (map\ (w{:})\ (inits\ ws))\ (tails\ ws) \end{aligned}$$

Aquí, *inits ws* devuelve la lista de los segmentos iniciales de *ws* en orden creciente de longitud, y *tails ws* devuelve los segmentos finales en orden decreciente de longitud. Por tanto, la función *descomp* devuelve la lista de todas las formas posibles de partir la entrada de manera que el primer componente de cada par sea una lista no vacía.

¿Cuánto tarda este programa para *llenar*? Hay $\Theta(2^n)$ modos de partir una secuencia de longitud n en una lista de listas no vacías; por tanto, *llenados* devuelve $\Theta(2^n)$ resultados y la complejidad en tiempo de *llenar* es al menos $\Omega(2^n)$.

Para mejorar el programa, necesitamos la suposición adicional

$$mejor \cdot map\ (us{:}) \equiv (us{:}) \cdot mejor$$

Dicho con palabras: fijada una primera línea, se obtiene el mejor párrafo tomando el mejor párrafo para las siguientes líneas. Esta condición es una aplicación de lo que en los textos de diseño de algoritmos se conoce como *principio de optimalidad*.

Dada la suposición anterior, es fácil comprobar que *llenar* se puede definir mediante

$$\begin{aligned} llenar\ [\,] &= [\,] \\ llenar\ (w : ws) &= mejor\ [us : llenar\ vs \mid (us, vs) \leftarrow descomp\ (w : ws)] \end{aligned}$$

Suponiendo que tanto *mejor* como *descomp* emplean tiempo lineal, tenemos

$$\begin{aligned} T(llenar)(0) &= O(1) \\ T(llenar)(n+1) &= \Theta(n) + \textstyle\sum_{k=0}^{n} T(llenar)(k) \end{aligned}$$

La solución de esta relación de recurrencia es $T(llenar)(n) = \Theta(2^n)$; por tanto, *llenar* necesita todavía tiempo exponencial. La razón es que, para toda secuencia final de *ws*, se recalcula *llenar vs*. La situación es muy similar a la de la función de Fibonacci, excepto que en vez de tener dos términos recursivos, tenemos una lista de ellos. La solución es similar: definimos *llenarColas* mediante

$$llenarColas = map\ llenar \cdot tails$$

Tenemos que $llenar = head \cdot llenarColas$; por tanto, se puede calcular *llenar* calculando *llenarColas*. Sólo resta dar un programa recursivo para *llenarColas*. El caso base

$$llenarColas\ [\,] = [[\,]]$$

es fácil, y para el caso recursivo argumentamos primero que

$$
\begin{array}{ll}
 & llenar\ (w : ws) \\
= & \quad \{\text{definición de } llenar\} \\
 & mejor\ [us : llenar\ vs \mid (us, vs) \leftarrow descomp\ (w : ws)] \\
= & \quad \{\text{definición de } descomp\} \\
 & mejor\ [us : llenar\ vs \mid (us, vs) \leftarrow zip\ (map\ (w :)\ (inits\ ws))\ (tails\ ws)] \\
= & \quad \{\text{conjetura}\} \\
 & mejor\ [(w : us) : vss \mid (us, vss) \leftarrow zip\ (inits\ ws)\ (map\ llenar\ (tails\ ws))] \\
= & \quad \{\text{definición de } llenarColas\} \\
 & mejor\ [(w : us) : vss \mid (us, vss) \leftarrow zip\ (inits\ ws)\ (llenarColas\ ws)]
\end{array}
$$

La demostración de la conjetura empleada se deja como ejercicio. Ahora podemos deducir

$$
\begin{array}{ll}
 & llenarColas\ (w : ws) \\
= & \quad \{\text{definición de } llenarColas, \text{ más algunas simplificaciones de rutina}\} \\
 & llenar\ (w : ws) : llenarColas\ ws \\
= & \quad \{\text{por el cálculo anterior de } llenar, \text{ definiendo } fts = llenarColas\ ws\} \\
 & mejor\ [(w : us) : vss \mid (us, vss) \leftarrow zip\ (inits\ ws)\ fts)] : fts
\end{array}
$$

En resumen,

$$
\begin{array}{lcl}
llenarColas\ [\,] & = & [[\,]] \\
llenarColas\ (w : ws) & = & mejor\ [(w : us) : vss \mid (us, vss) \leftarrow zip\ (inits\ ws, fts)] : fts \\
 & & \textbf{where}\ fts = llenarColas\ ws
\end{array}
$$

Para el análisis de tiempos, tenemos

$$
\begin{array}{lcl}
T(llenarColas)(0) & = & O(1) \\
T(llenarColas)(n+1) & = & \Theta(n) + T(llenarColas)(n)
\end{array}
$$

La solución es $T(llenarColas)(n) = \Theta(n^2)$. Por tanto, el programa reformado para *llenar* emplea tiempo cuadrático.

Se puede mejorar todavía más el programa para *llenar*. Una vez que se ha generado una línea demasiado larga para entrar en la anchura dada, no tiene sentido generar líneas más largas puesto que tampoco entrarán. Este hecho obvio, pero importante, no está explícito en el programa anterior, sino que está escondido en la definición de *desp*. Sacándolo a la superficie y explotándolo, se puede conseguir truncar *llenarColas* a una lista de longitud como máximo m, para algún m dependiente de la anchura dada. El programa para *llenarColas* empleará entonces $O(mn)$ pasos, que es lineal para un m fijo.

7.4.4 La media aritmética

El último ejemplo para ilustrar la técnica de tuplamiento consigue una mejora de eficiencia más modesta. Comparada con las mejoras asintóticas conseguidas anteriormente, el ejemplo conduce a una mejora en el tiempo de ejecución de como mucho un factor constante.

Se define la función *media* para listas no vacías de números mediante

$$
\begin{array}{lcl}
media & :: & [Float] \to Float \\
media\ xs & = & (sum\ xs)/(length\ xs)
\end{array}
$$

La evaluación de *media* necesita dos recorridos del argumento, pero mediante tuplamiento podemos conseguir que se haga sólo uno. Definimos *sumaLong* mediante

$$sumaLong\ xs \quad = \quad (sum\ xs, length\ xs)$$

Se tiene que $media' = uncurry\ (/) \cdot sumaLong$. La definición recursiva directa de *sumaLong* es

$$
\begin{array}{lcl}
sumaLong\ [x] & = & (x, 1) \\
sumaLong\ (x : y : xs) & = & (x + z, n + 1), \textbf{where}\ (z, n) = sumaLong\ (y : xs)
\end{array}
$$

La función *media'* supone un único recorrido de su argumento. Ambas versiones necesitan $\Theta(n)$ pasos para una lista de longitud n, y desde el punto de vista del tiempo no es importante escoger entre ellas. El tiempo ahorrado al no recorrer la lista por segunda vez se puede perder teniendo que formar un par en cada paso. Exactamente la misma situación surge con *splitAt* (véase la Sección 4.2) y *span* (véase la Sección 5.2). Entonces, ¿por qué preocuparnos de que un programa recorra su argumento sólo una vez?

La respuesta tiene que ver con el espacio. Se logra una gran ventaja utilizando *media'* en lugar de *media* cuando se considera el espacio. Consideremos la evaluación de

$$media\ [1..1000]$$

Dado que *media xs* contiene dos apariciones de *xs* en la parte derecha de su definición, la lista creada en la evaluación de [1..1000] como resultado del primer recorrido, es retenida para el segundo recorrido. En otras palabras, el cálculo del promedio de una lista muy larga utilizando la primera definición puede ocasionar que el evaluador consuma todo el espacio. Con la segunda definición de *media* podemos conseguir que el cálculo se realice utilizando una cantidad de espacio constante. Éste es el tema de la siguiente sección.

Ejercicios

7.4.1 Supongamos que $T(k+1) = 2T(k) + \Theta(2^k)$. Demostrar que $T(k) = \Theta(2^k)$.

7.4.2 Supongamos que $T(n) = T(m) + T(n-m) + O(1)$, donde $m = n$ **div** 2. Demostrar que $T(n) = \Theta(n)$.

7.4.3 Demostrar por inducción que $fib\ n = (\phi^n - \hat{\phi}^n) / \sqrt{5}$, donde $\phi = (1 + \sqrt{5})/2$ y $\hat{\phi} = (1 - \sqrt{5})/2$ son las dos raíces de $x^2 = x + 1$.

7.4.4 Suponiendo que $mejorQue\ (us : uss)\ (us : vss) = us : mejorQue\ uss\ vss$, demostrar que $mejor \cdot map\ (us :) = (us :) \cdot mejor$.

7.5 Controlando el espacio

Consideremos la reducción del término *sum* [1..1000], donde $sum = foldl\ (+)\ 0$:

$$sum\ [1..1000]$$

$$
\begin{aligned}
&= \mathit{foldl}\ (+)\ 0\ [1..1000] \\
&= \mathit{foldl}\ (+)\ (0+1)\ [2..1000] \\
&= \mathit{foldl}\ (+)\ ((0+1)+2)\ [3..1000] \\
&\vdots \\
&= \mathit{foldl}\ (+)\ (\ldots((0+1)+2)+\ldots+1000)\ [\,] \\
&= (\ldots((0+1)+2)+\ldots+1000) \\
&= 500500
\end{aligned}
$$

El punto a resaltar es que, al calcular *sum* [1..*n*] mediante reducción más externa, la expresión crece hasta un tamaño proporcional a *n*. Por otra parte, si utilizamos una mezcla juiciosa de pasos de reducción más externa y más interna, entonces podemos obtener la siguiente secuencia de reducción:

$$
\begin{aligned}
& \mathit{sum}\ [1..1000] \\
&= \mathit{foldl}\ (+)\ 0\ [1..1000] \\
&= \mathit{foldl}\ (+)\ (0+1)\ [2..1000] \\
&= \mathit{foldl}\ (+)\ 1\ [2..1000] \\
&= \mathit{foldl}\ (+)\ (1+2)\ [3..1000] \\
&= \mathit{foldl}\ (+)\ 3\ [3..1000] \\
&\vdots \\
&= \mathit{foldl}\ (+)\ 500500\ [\,] \\
&= 500500
\end{aligned}
$$

El tamaño máximo de cualquier expresión de esta secuencia está acotado por una constante. Resumiendo, al reducir *sum* [1..*n*] a forma normal mediante reducción más externa pura necesitamos espacio en $\Omega(n)$, mientras que una combinación de reducción más externa y más interna requiere solamente espacio en $O(1)$.

Esto sugiere que sería útil tener un modo de controlar el orden de reducción y para permitirlo introducimos una función especial *strict*.

7.5.1 Forma normal débil y la función *strict*

Se puede controlar el orden de reducción utilizando la función especial *strict*. Para reducir un término de la forma *strict f e*, primero se reduce *e* a *forma normal débil*[4], y después se aplica *f*. Una expresión *e* está en forma normal débil si *e* es una función o *e* tiene la forma de una constructora de tipos de datos aplicada a cero o más argumentos. Toda expresión en forma normal está en forma normal débil, pero no a la inversa. Por ejemplo, $e_1 : e_2$ está en forma normal débil, pero está en forma normal sólo si e_1 y e_2 están ambos en forma normal. De forma similar, *Unir* e_1 e_2 y (e_1, e_2) están en forma normal débil, pero no están en formal normal a menos que e_1 y e_2 estén ambos en forma normal. En la expresión *strict f e*, el término *e* será reducido mediante reducción más externa, excepto, por supuesto, si aparecen llamadas a *strict* mientras reducimos *e*.

A modo de ejemplo sencillo, sea $\mathit{succ}\ x = x + 1$. Entonces

$$
\begin{aligned}
& \mathit{succ}\ (\mathit{succ}\ (8 \times 5)) \\
&= \mathit{succ}\ (8 \times 5) + 1
\end{aligned}
$$

[4]N. del T.: En el original *head-normal form*. A veces se emplea también *weak head normal form*. Aunque técnicamente difieren ligeramente, a menudo se emplean indistintamente.

$$
\begin{aligned}
&= ((8 \times 5) + 1) + 1 \\
&= (40 + 1) + 1 \\
&= 41 + 1 \\
&= 42
\end{aligned}
$$

Por otra parte,

$$
\begin{aligned}
& \mathit{strict}\ \mathit{succ}\ (\mathit{strict}\ \mathit{succ}\ (8 \times 5)) \\
&= \mathit{strict}\ \mathit{succ}\ (\mathit{strict}\ \mathit{succ}\ 40) \\
&= \mathit{strict}\ \mathit{succ}\ (\mathit{succ}\ 40) \\
&= \mathit{strict}\ \mathit{succ}\ (40 + 1) \\
&= \mathit{strict}\ \mathit{succ}\ 41 \\
&= \mathit{succ}\ 41 \\
&= 41 + 1 \\
&= 42
\end{aligned}
$$

Ambos casos realizan los mismos pasos de reducción, pero en orden diferente.

La currificación se aplica a *strict* del modo habitual. De ello se deduce que si *f* es una función de tres argumentos, la expresión *strict* $(f\ e_1)\ e_2\ e_3$ hace que el segundo argumento e_2 se reduzca antes, pero no el primero ni el tercero.

A partir de aquí, podemos definir una función *sfoldl*, una versión estricta de *foldl*, como sigue:

$$
\begin{aligned}
\mathit{sfoldl}\ (\oplus)\ a\ [\,] &= a \\
\mathit{sfoldl}\ (\oplus)\ a\ (x : xs) &= \mathit{strict}\ (\mathit{sfoldl}\ (\oplus))\ (a \oplus x)\ xs
\end{aligned}
$$

Siendo $\mathit{sum} = \mathit{sfoldl}\ (+)\ 0$ tenemos ahora:

$$
\begin{aligned}
& \mathit{sum}\ [1..1000] \\
&= \mathit{sfoldl}\ (+)\ 0\ [1..1000] \\
&= \mathit{strict}\ (\mathit{sfoldl}\ (+))\ (0+1)\ [2..1000] \\
&= \mathit{sfoldl}\ (+)\ 1\ [2..1000] \\
&= \mathit{strict}\ (\mathit{sfoldl}\ (+))\ (1+2)\ [3..1000] \\
&= \mathit{sfoldl}\ (+)\ 3\ [3..1000] \\
&\vdots \\
&= \mathit{sfoldl}\ (+)\ 500500\ [\,] \\
&= 500500
\end{aligned}
$$

Esta secuencia de reducción evalúa *sum* en espacio constante.

Consideremos ahora la función *media'* de la sección precedente; se puede definir esta función mediante

$$
\begin{aligned}
\mathit{media'} &= \mathit{uncurry}\ (/) \cdot \mathit{sumaLong} \\
\mathit{sumaLong} &= \mathit{foldl}\ f\ (0, 0) \\
&\quad \text{where } f\ (s, n)\ x = (s + x, n + 1)
\end{aligned}
$$

Supongamos que reemplazamos *foldl* por *sfoldl* en la definición de *sumaLong*; ¿obtenemos entonces un programa con espacio constante para *media'*? Para apreciar por qué la respuesta es no, consideremos la siguiente secuencia de reducción:

$$\mathit{sumaLong}\ [1..1000]$$

$$
\begin{aligned}
&= \mathit{sfoldl}\ f\ (0, 0)\ [1..1000] \\
&= \mathit{strict}\ (\mathit{sfoldl}\ f)\ (0 + 1, 0 + 1)\ [2..1000] \\
&= \mathit{sfoldl}\ f\ (0 + 1, 0 + 1)\ [2..1000] \\
&= \mathit{strict}\ (\mathit{sfoldl}\ f)\ ((0 + 1) + 2, (0 + 1) + 1)\ [3..1000] \\
&= \mathit{sfoldl}\ f\ ((0 + 1) + 2, (0 + 1) + 1)\ [3..1000]
\end{aligned}
$$

y así sucesivamente. El problema es que una expresión como $((0 + 1) + 2, (0 + 1) + 1)$ está en forma normal débil y, por tanto, *strict* no requiere más evaluación. La evalución resultante de *sumaLong* $[1..n]$ utilizará $\Theta(n)$ unidades de espacio. La solución es que no solamente *foldl f* sea estricta, sino hacer que *f* también lo sea, definiendo

$$
\begin{aligned}
f'\ (s, n)\ x \quad = \quad & \mathit{strict}\ (\mathit{strict}\ \mathit{tupla}\ (s + x))\ (n + 1) \\
& \text{where } \mathit{tupla}\ a\ b = (a, b)
\end{aligned}
$$

La secuencia de reducción revisada ahora es

$$
\begin{aligned}
& \mathit{sumaLong}\ [1..1000] \\
= \ & \mathit{sfoldl}\ f'\ (0, 0)\ [1..1000] \\
= \ & \mathit{strict}\ (\mathit{sfoldl}\ f')\ (\mathit{strict}\ (\mathit{strict}\ \mathit{tupla}\ (0 + 1))\ (0 + 1))\ [2..1000] \\
= \ & \mathit{strict}\ (\mathit{sfoldl}\ f')\ (\mathit{strict}\ (\mathit{strict}\ \mathit{tupla}\ (0 + 1))\ 1)\ [2..1000] \\
= \ & \mathit{strict}\ (\mathit{sfoldl}\ f')\ (\mathit{strict}\ (\mathit{strict}\ \mathit{tupla}\ 1)\ 1)\ [2..1000] \\
= \ & \mathit{strict}\ (\mathit{sfoldl}\ f')\ (1, 1)\ [2..1000] \\
= \ & \mathit{sfoldl}\ f'\ (1, 1)\ [2..1000]
\end{aligned}
$$

y así sucesivamente. Es obvio que esta secuencia tiene el comportamiento deseado.

Se puede reexpresar la definición operacional de *strict* del modo siguiente:

$$\mathit{strict}\ f\ x \quad = \quad \textbf{if}\ x = \bot\ \textbf{then}\ \bot\ \textbf{else}\ f\ x$$

Recordemos que se dice que una función *f* es estricta si $f\ \bot = \bot$. De la ecuación se deduce que $f = \mathit{strict}\ f$ si y sólo si *f* es una función estricta. Para ver esto, simplemente consideremos los valores de $f\ x$ y $\mathit{strict}\ f\ x$ en los dos casos $x = \bot$ y $x \neq \bot$. Esto explica el nombre *strict*.

Más aún, si *f* es estricta, pero no totalmente indefinida, y $e \neq \bot$, entonces la reducción de $f\ e$ implica finalmente la reducción de *e*. Así, si *f* es una función estricta, la evaluación de $f\ e$ realiza exactamente los mismos pasos de reducción que la de $\mathit{strict}\ f\ e$, aunque posiblemente en orden diferente. En otras palabras, cuando *f* es estricta, el reemplazarla por $\mathit{strict}\ f$ no cambia el significado o el tiempo asintótico requerido para aplicarla, aunque puede cambiar el espacio requerido para el cálculo.

Es fácil demostrar que si $\bot \oplus x = \bot$ para todo *x*, entonces $\mathit{foldl}\ (\oplus)\ \bot\ xs = \bot$ para toda lista finita *xs*. En otras palabras, si $(\oplus)$ es estricta en su argumento izquierdo, entonces $\mathit{foldl}\ (\oplus)$ es estricta, y por tanto es equivalente a $\mathit{strict}\ (\mathit{foldl}\ (\oplus))$, y por tanto también equivalente a $\mathit{sfoldl}\ (\oplus)$. Se desprende que reemplazar *foldl* por *sfoldl* en la definición de *sum* es válido, y dicho reemplazamiento es válido siempre que *foldl* se aplique a un operador binario que sea estricto en su primer argumento.

7.5.2 Revisión de las funciones *fold*

El primer teorema de dualidad dado en la Sección 4.5 establece que si $(\oplus)$ es asociativo y tiene como elemento neutro *e*, entonces

$$\mathit{foldr}\ (\oplus)\ e\ xs \quad = \quad \mathit{foldl}\ (\oplus)\ e\ xs$$

para todas las listas finitas *xs*. Por otra parte, las dos funciones pueden tener diferentes complejidades en tiempo y en espacio. La decisión de cuál usar depende de las propiedades de $\oplus$.

Primero, supongamos que $\oplus$ es estricta en ambos argumentos, y que se puede calcular en tiempo O(1) y en espacio O(1). Ejemplos que entran en esta categoría son $(+)$ y $(\times)$. En este caso, no es difícil verificar que tanto $\mathit{foldr}\ (\oplus)\ e$ como $\mathit{foldl}\ (\oplus)\ e$ requieren ambos tiempo $O(n)$ y espacio $O(n)$ sobre una lista de longitud *n*. Sin embargo, el mismo argumento usado anteriormente para *sum* se generaliza para demostrar que, en este caso, se puede reemplazar de forma segura *foldl* por *sfoldl*. Mientras que $\mathit{sfoldl}\ (\oplus)\ a$ requiere todavía tiempo $O(n)$ para evaluarse sobre una lista de longitud *n*, solamente necesita espacio $O(1)$. Por tanto, en este caso, *sfoldl* es el ganador indiscutible.

Si $\oplus$ no satisface las propiedades anteriores, entonces la elección de un ganador no es tan fácil. Sin embargo, una buena regla práctica es que si $\oplus$ es no estricta en cualquiera de los argumentos, entonces normalmente *foldr* es más eficiente que *foldl*. Vimos un ejemplo en la Sección 7.2: la función *concat* se calcula más eficientemente utilizando *foldr* que *foldl*. Observemos que, mientras que $+\!\!+$ es estricta en su primer argumento, es no estricta en el segundo.

Otro ejemplo viene dado por la función $\mathit{and} = \mathit{foldr}\ (\wedge)\ \mathit{True}$. Al igual que $+\!\!+$, el operador $\wedge$ es estricto en su primer argumento, pero no estricto en el segundo. En particular, $\mathit{False} \wedge x$ devuelve *False* sin evaluar *x*. Supongamos que nos dan una lista *xs* de *n* booleanos y *k* es el primer valor para el cual $xs\ !!\ k = \mathit{False}$. Entonces la evaluación de $\mathit{foldr}\ (\wedge)\ \mathit{True}\ xs$ necesita $O(k)$ pasos, mientras que $\mathit{foldl}\ (\wedge)\ \mathit{True}\ xs$ requiere $\Omega(n)$ pasos. De nuevo, *foldr* es una elección mejor.

Resumiendo: para funciones como $+$ o $\times$, que son estrictas en ambos argumentos y se pueden calcular en tiempo y espacio constantes, *sfoldl* es más eficiente, pero para funciones como $\wedge$ y $+\!\!+$, que son no estrictas en algún argumento, a menudo *foldr* es más eficiente.

Ejercicios

7.5.1 Escribir una definición de la función *sum* que no use *strict* pero que, sin embargo, se evalúe en espacio constante.

7.5.2 ¿Cómo deberíamos definir *minlist*?

7.6 Fusión, diferenciación finita y deforestación

El grupo final de técnicas tienen un aspecto común y se tratan mejor juntas. En todas ellas, una vez expresada la solución de un problema en términos de la composición $f_1 \cdot f_2 \cdot \cdots \cdot f_n$ de un cierto número de funciones, se puede incrementar la eficiencia combinando, o *fusionando*, dos o más de las funciones auxiliares f_k en una sola función.

Hemos encontrado ya la fusión en el contexto de la operaciones *fold* sobre listas. Recordemos de la Sección 4.5 que

$$f \cdot \mathit{foldr}\ g\ a \quad = \quad \mathit{foldr}\ h\ (f\ a)$$

siempre que f sea una función estricta y $f \cdot (g\ x) = (h\ x) \cdot f$. Operacionalmente, el lado izquierdo dice que se calcula un valor a partir de los elementos de una lista y dicho valor es consumido por una función para producir un segundo valor. El lado derecho dice que, siempre que se cumpla una cierta condición, se puede producir directamente el segundo valor a partir de los elementos de la lista. La condición $f \cdot (g\ x) = (h\ x) \cdot f$ establece que se puede calcular *incrementalmente* el valor devuelto por f. También se han utilizado los términos *diferenciación finita* y *diferenciación formal* para describir esta situación. Hemos visto una aplicación paradigmática de la diferenciación finita en el problema de la suma máxima de segmentos considerado en la Sección 4.5. Recordemos que redujo el tiempo de ejecución de $\Theta(n^3)$ a $\Theta(n)$.

Un caso particular del uso de fusión es la eliminación de tipos de datos intermedios de un programa. Esta técnica particular se denomina *deforestación*. Vimos un ejemplo en la Sección 6.6, y veremos otro más a lo largo de esta sección.

7.6.1 Quicksort

Recordemos la función *ordenar* de la Sección 6.2:

```
ordenar  =  aplanar · construirArBus
```

La función *construirArBus* construye un árbol binario de búsqueda y *aplanar* enumera sus valores en orden creciente. Las funciones eran

```
data (Ord α) ⇒ArBus α = Vacio     | Unir (ArBus α) α (ArBus α)

construirArBus              ::  (Ord α) ⇒[α] →ArBus α
construirArBus [ ]          =   Vacio
construirArBus (x : xs)     =   Unir (construirArBus ys) x (construirArBus zs)
where (ys, zs) = partir (≤ x) xs

partir       ::  (α →Bool) →[α] ([α], [α])
partir p xs  =   (filter p xs, filter (not · p) xs)

aplanar              ::  ArBus α →[α]
aplanar Vacio        =   [ ]
aplanar (Unir xt x yt)  =  aplanar xt ++[x] ++aplanar yt
```

Ahora el objetivo es fusionar el cálculo de *aplanar* y *construirArBus*, eliminando por tanto los valores del tipo *ArBus* α del programa. El resultado es una función *qsort* definida mediante

```
qsort        ::  (Ord α) ⇒ [α] →[α]
qsort [ ]    =   [ ]
qsort (x : xs)  =  qsort ys ++[x] ++qsort zs
                   where (ys, zs) = partir (≤ x) xs
```

Ésta es una versión del famoso algoritmo llamado *quicksort*.

Para evaluar el tiempo de *qsort xs*, supongamos que *xs* es una lista de longitud $n > 0$. La lista *filter* $(\leq x)$ *xs* puede tener cualquier longitud k en el rango $0 \leq k < n$, en cuyo caso *filter* $(> x)$ *xs* tiene longitud $n - k - 1$. Dado que T estima el tiempo de ejecución en el caso peor, tenemos que escoger la máxima estimación sobre todas las posibles particiones. Por tanto

$$T(qsort)(n) \quad = \quad \max_{0 \leq k < n} (T(qsort)(k) + T(qsort)(n-k-1) + \Theta(n))$$

El término $\Theta(n)$ contabiliza tanto el tiempo necesario par evaluar *ys* y *zs* sobre una lista de longitud n, como el requerido para concatenar los resultados ordenados. Dado que el peor caso sucede cuando $k = 0$ o $k = n - 1$, obtenemos

$$T(qsort)(n+1) \quad = \quad T(qsort)(n) + \Theta(n)$$

La solución es $T(qsort)(n) = \Theta(n^2)$. Por tanto, *quicksort* es un agoritmo de tiempo cuadrático en el peor caso. Por otra parte, *quicksort* es eficiente en promedio, necesitando $O(n \log n)$ pasos.

¿Podemos mejorar la eficiencia de *qsort*? La presencia de la operación ++ señala que es probable que resulte útil la técnica del parámetro acumulador. Supongamos que definimos *qConcat* mediante

```
qConcat        ::  (Ord α) ⇒[α] →[α] →[α]
qConcat xs ys  =   qsort xs ++ys
```

Una síntesis rutinaria nos proporciona

```
qsort xs              =  qConcat xs [ ]
qConcat [ ] ws        =  ws
qConcat (x : xs) ws   =  qConcat ys (x : qConcat zs ws)
                         where (ys, zs) = partir (≤ x) xs
```

El resultado es un programa más rápido que el anterior, pero todavía con complejidad cuadrática en el tiempo.

Existe otra optimización de *qsort* que implica mejora en espacio en lugar de en tiempo. En el cálculo de *partir* $(\leq x)$ *xs*, se recorre la lista *xs* dos veces. Una situación similar aparece con la función *media* de la Sección 7.4; por tanto, deberíamos combinar los dos recorridos en uno empleando la técnica de tuplamiento.

Es fácil reescribir la definición de *partir* de la forma siguiente

```
partir p  =  foldr f ([], [])
             where f y (ys, zs) = if p y then (y : ys, zs) else (ys, y : zs)
```

En esta versión, la lista del argumento se recorre sólo una vez.

Una vez reescrita la definición de *partir*, se puede pensar que el programa de *qsort* es ahora tan eficiente en espacio como sea posible, pero desafortunadamente no lo es. Para algunas entradas de longitud n el espacio requerido por el programa es $\Omega(n^2)$. Nos referiremos a esta situación como una *fuga de espacio*[5]. Una fuga de espacio es una pérdida oculta de eficiencia en espacio. La fuga de espacio en el programa anterior *quicksort* se produce cuando la entrada está ya ordenada pero en orden decreciente. Para explicar por qué surge, reescribamos el caso recursivo de *qsort* de la siguiente forma equivalente

```
qsort (x : xs)  =  qsort (fst p) ++[x] ++qsort (snd p)
                   where p = partir (≤ x) xs
```

[5]N. del T.: En el original *space leak*.

Si $(x : xs)$ tiene longitud $(n + 1)$ y está en orden decreciente, entonces $p = partir\ (\leq x)\ xs$ es un par de listas cuyo primer componente tiene longitud n y cuyo segundo componente tiene longitud cero. La evaluación de p es activada al evaluar el término $qsort\ (fst\ p)$. Sin embargo, las n unidades de espacio ocupadas por p no pueden ser recuperadas hasta que el otro término $qsort\ (snd\ p)$ se evalúa, instante en el cual se puede desechar el puntero p. Entre estas dos evaluaciones se generan más ternas. La primera ocupa $n - 1$ unidades de espacio, la segunda $n - 2$ unidades, y así sucesivamente. El espacio total requerido para llevar a cabo el cálculo es por tanto $\Omega(n^2)$.

En la práctica, esto significa que la evaluación de *qsort* sobre algunas entradas grandes abortará debido a la insuficiencia de espacio disponible. La solución consiste en forzar de algún modo la evaluación de *partir* y, lo que es igualmente importante, vincular *ys* y *zs* a los respectivos componentes del par resultante, no al par propiamente dicho.

Una manera de conseguir esta situación es fusionar el cálculo de *partir* en el programa principal. Definimos la función *qaux* mediante

$$\begin{array}{lcl} qaux\ x\ xs\ us\ vs & = & qsort\ (us +\!\!+ ys) +\!\!+ [x] +\!\!+ qsort\ (vs +\!\!+ zs) \\ & & \textbf{where}\ (ys, zs) = partir\ (\leq x)\ xs \end{array}$$

En particular, tenemos

$$\begin{array}{ll} & qsort\ (x : xs) \\ = & \quad \{\text{definición de } qsort\text{, haciendo } (ys, zs) = partir\ (\leq x)\ xs\} \\ & qsort\ ys +\!\!+ [x]\ qsort\ zs \\ = & \quad \{\text{definición de } qaux\} \\ & qaux\ x\ xs\ [\,]\ [\,] \end{array}$$

Se deduce que se puede definir *qsort* en términos de *qaux* mediante

$$\begin{array}{lcl} qsort\ [\,] & = & [\,] \\ qsort\ (x : xs) & = & qaux\ x\ xs\ [\,]\ [\,] \end{array}$$

El paso final es sintetizar una definición recursiva de *qaux*. El caso base es

$$qaux\ x\ [\,]\ us\ vs \quad = \quad qsort\ us +\!\!+ [x] +\!\!+ qsort\ vs$$

Para el caso recursivo $qaux\ x\ (y : xs)\ us\ vs$, observemos que

$$\begin{array}{lcl} partir\ (\leq x)\ (y : xs) & = & \textbf{if}\ y \leq x\ \textbf{then}\ (y : ys, zs)\ \textbf{else}\ (ys, y : zs) \\ & & \textbf{where}\ (ys, zs) = partir\ (\leq x)\ xs \end{array}$$

Por brevedad, calcularemos el caso recursivo suponiendo $y \leq x$:

$$\begin{array}{ll} & qaux\ x\ (y : xs)\ us\ vs \\ = & \quad \{\text{definición de } qaux\text{, haciendo } (ys, zs) = partir\ (\leq x)\ xs\} \\ & qsort\ (us +\!\!+ y : ys) +\!\!+ [x] +\!\!+ qsort\ (vs +\!\!+ zs) \\ = & \quad \{\text{conjetura: (ver más adelante)}\} \\ & qsort\ (y : us +\!\!+ ys) +\!\!+ [x] +\!\!+ qsort\ (vs +\!\!+ zs) \\ = & \quad \{\text{definición de } qaux\} \\ & qaux\ x\ xs\ (y : us)\ vs \end{array}$$

La conjetura es que si *ys* es cualquier permutación de *xs*, entonces $qsort\ xs = qsort\ ys$. Intuitivamente, la conjetura es obvia: ordenar una lista depende solamente de los elementos de la lista y no de su orden. Se omite una demostración formal. Esta propiedad de la ordenación es crucial para desarrollar una implementación eficiente en espacio de *quicksort*.

El resultado es el siguiente programa para *qsort*:

$$\begin{array}{lcl} qsort\ [\,] & = & [\,] \\ qsort\ (x : xs) & = & qaux\ x\ xs\ [\,]\ [\,] \\ qaux\ x\ [\,]\ us\ vs & = & qsort\ us +\!\!+ [x]\ qsort\ vs \\ qaux\ x\ (y : xs)\ us\ vs & = & \textbf{if}\ y \leq x \\ & & \textbf{then}\ qaux\ x\ xs\ (y : us)\ vs \\ & & \textbf{else}\ qaux\ x\ xs\ us\ (y : vs) \end{array}$$

El resultado es un programa bastante poco atractivo que, sin embargo, tiene complejidad $O(n)$ en espacio. Más aún, se puede utilizar de nuevo la técnica del parámetro acumulador para reducir el tiempo de ejecución por un factor constante.

En la programación funcional existen métodos de ordenación mejores que *quicksort*. Sin embargo, resulta fascinante estudiar un problema en el que pueden desempeñar un papel las técnicas de deforestación, de parámetros acumuladores, de tuplamiento y de fusión.

Ejercicios

7.6.1 El valor $selec\ k\ xs$, donde $selec\ k = (!!k) \cdot ordenar$, selecciona el k-ésimo menor elemento de una lista no vacía. En particular, $minlist = selec\ 0$. Tomando $ordenar = qsort$, utilizar la técnica de fusión para producir un algoritmo más eficiente para *selec*.

7.6.2 Si suponemos que toda posible partición en el caso recursivo es igualmente probable, y se cumple lo mismo para todas las subsiguientes particiones, entonces el caso promedio de *qsort*, $T_m(qsort)$, satisface

$$T_m(qsort)(n + 1) = 1/n \sum_{k=0}^{n} (T_m(qsort)(k) + T_m(qsort)(n - k) + \Theta(n))$$

Demostrar que $T_m(qsort)(n) = \Theta(n \log n)$

7.7 Notas del capítulo

Para más información sobre la evaluación perezosa y sobre la reducción de grafos, consúltense los libros de Peyton Jones (1987) y de Peyton Jones y Lester (1991). La notación asintótica y la solución de relaciones de recurrencia utilizando la notación-O se tratan en Cormen et cia. (1990). El tema del análisis asintótico del tiempo para programas funcionales perezosos es todavía un área de investigación; para ver varios enfoques de este problema, consúltense Bjerner y Holmström (1989), Sands (1995) y Wadler (1987).

La idea de fusionar dos cálculos es tan vieja como el tema de la transformación de programas propiamente dicho; si se desea una referencia pionera sobre la técnica

en un marco funcional, véase Burstall y Darlington (1977). La técnica del parámetro acumulador se estudia en Bird (1984).

Para una ampliación de la estrategia de tuplamiento para la optimización de programas, veánse Bird (1980) y Pettorossi (1984). El principio de optimalidad y su importancia para los algoritmos de programación dinámica (de los cuales el problema del llenado de párrafos es un ejemplo) se discute en la mayoría de los libros sobre diseño de algoritmos; véase Bellman (1957), donde se estudió la técnica por primera vez, y también de Moor (1994) que presenta las ideas en un marco categórico. Bird y de Moor (1997) contiene una exposición sistemática del uso de la programación dinámica para resolver problemas de optimización. El problema del llenado de párrafos se trata en Bird (1986) y en Knuth y Plass (1981).

La idea de eliminar tipos de datos intermedios en un cálculo se estudia en Wadler (1984, 1990b); para ejemplos adicionales, véase también Bird y de Moor (1997).

El algoritmo *quicksort* se debe a Hoare (1962). Hughes (1984) fue el primero en darse cuenta de la fuga de espacio de *quicksort* cuando se expresa como programa funcional. Véase Runciman y Röjemo (1996) para una exposición de técnicas que identifican fugas de espacio.

Capítulo 8

Tipos abstractos de datos

El objetivo de este capítulo es introducir la idea de *tipo abstracto de datos* y los mecanismos que Haskell suministra para definirlos. Grosso modo, los tipos abstractos de datos difieren de los introducidos hasta ahora mediante una declaración **data** en que se puede escoger cómo representar sus valores. Cada elección de representación conduce a una implementación diferente del tipo de datos. Las ideas principales se exponen mediante el estudio de cuatro tipos abstractos: conjuntos, multiconjuntos, vectores y colas. En cada ejemplo, se especifica el tipo abstracto formalmente, se sugieren posibles representaciones y se compara la eficiencia de implementaciones diferentes.

8.1 Conceptos básicos

El efecto de una declaración **data** es introducir un nuevo tipo de datos describiendo cómo se construyen sus valores. Cada valor se nombra mediante una expresión construida exclusivamente a partir de las constructoras del tipo de datos. Más aún, elementos con nombres diferentes denotan valores diferentes. Utilizando el mecanismo de ajuste de patrones sobre las constructoras, se pueden definir operaciones que generan y procesan elementos del tipo de datos de la manera que deseemos. De acuerdo con esto, no hay necesidad de postular la existencia de ninguna operación primitiva. Los tipos en los cuales se describen los valores pero no las operaciones, se llaman tipos *concretos*.

La situación es justo la opuesta en los tipos *abstractos*. Un tipo abstracto de datos se define nombrando, no sus valores, sino sus operaciones. Por ejemplo, *Float* es un tipo abstracto de datos en Haskell. Se nos proporcionan operaciones primitivas para realizar comparaciones y cálculos aritméticos y también un modo de mostrar valores en coma flotante, pero no se especifica cómo se representan tales números en el evaluador de Haskell y tampoco podemos hacer ajuste de patrones con ellos. En general, el programador que usa un tipo abstracto puede no saber, e incluso se le impide saber, cómo se representan los elementos del tipo de datos. Tales *barreras de abstracción* son útiles cuando en un proyecto trabaja más de un programador o incluso cuando un solo programador trabaja en una tarea no trivial. En particular, se puede cambiar la representación sin afectar a la validez de los guiones que usan dicho tipo de datos.

8.1.1 Un ejemplo

Daremos ahora un ejemplo de tipo abstracto de datos. Supongamos que un programador quiere utilizar el tipo de datos *Cola* α de las colas sobre un tipo arbitrario α. La idea básica de una cola es que se le pueden añadir elementos al final y eliminarlos del principio (si no está vacía). Por tanto, una cola es una especie de lista finita con un conjunto restringido de operaciones. El programador desea una implementación eficiente de *Cola* α pero no está interesado en escribirla. En su lugar, él o ella puede desear que otro programador suministre una implementación apropiada, o puede utilizar una solución predefinida ofrecida por el entorno del sistema. ¿Cómo se puede llevar a cabo este esquema?

En primer lugar, el programador desea saber cuáles son las operaciones primitivas disponibles. Supongamos que las operaciones sobre colas son

$$\begin{array}{lcl} vacia & :: & Cola\ \alpha \\ poner & :: & \alpha \to Cola\ \alpha \to Cola\ \alpha \\ prim & :: & Cola\ \alpha \to \alpha \\ elim & :: & Cola\ \alpha \to Cola\ \alpha \\ esVacia & :: & Cola\ \alpha \to Bool \end{array}$$

Dicha lista de operaciones, junto con sus tipos, recibe el nombre de *signatura* del tipo de datos *Cola* α.

En segundo lugar, el programador necesita saber lo que significan estas operaciones. Sus nombres y sus tipos dan alguna pista; también ayuda una descripción informal, pero se necesita una información más precisa. Los usuarios de un tipo de datos necesitan saber en qué propiedades pueden confiar y el implementador de un tipo de datos necesita saber qué es lo que debe proporcionar.

8.1.2 Especificaciones algebraicas

Un modo de proporcionar dicha información consiste en dar lo que se llama una *especificación algebraica* o *axiomática* de *Cola* α. Ésta consiste en una lista de *axiomas* que las operaciones deben satisfacer. Una posible lista es la siguiente:

$$\begin{array}{lcl} esVacia\ vacia & = & True \\ esVacia\ (poner\ x\ xq) & = & False \\ \\ prim\ (poner\ x\ vacia) & = & x \\ prim\ (poner\ x\ (poner\ y\ xq)) & = & prim\ (poner\ y\ xq) \\ \\ elim\ (poner\ x\ vacia) & = & vacia \\ elim\ (poner\ x\ (poner\ y\ xq)) & = & poner\ x\ (elim\ (poner\ y\ xq)) \end{array}$$

El primer punto a resaltar es que estas ecuaciones se parecen mucho a las definiciones formales de *esVacia*, *prim* y *elim*, basadas en un tipo de datos cuyas constructoras son *vacia* y *poner*. De hecho, se diseñaron precisamente con esta idea. Sin embargo, deberían ser entendidas, más adecuadamente, sólo como un modo de expresar ciertas relaciones entre las cinco funciones. El implementador no está obligado a implementar las colas utilizando un tipo de datos con las constructoras *vacia* y *poner*.

La especificación axiomática de *Cola* α, junto con las propiedades que se puedan derivar de ella, son las únicas en las que los usuarios del tipo de datos pueden confiar. En particular, la especificación no obliga a ningún valor para el término *elim vacia*. Desde el punto de vista del usuario, *elim vacia* $= \bot$, es decir, la cola indefinida. Dependiendo de cómo interpretemos los axiomas, es posible distinguir entre $\bot$ y *poner* $x\ \bot$. Por ejemplo, los axiomas de *esVacia* no obligan a asignar un valor para *esVacia* $\bot$ pero, dependiendo de la interpretación del segundo axioma de *esVacia*, podríamos ser capaces de deducir que *esVacia* (*poner* $x\ \bot$) = *False*. Para resolver esta ambigüedad, añadimos otro axioma a la especificación, a saber, que *poner* x es una función *estricta* para todo x. Entonces tenemos

$$esVacia\ (poner\ x\ \bot) = esVacia\ \bot = \bot$$

La principal consecuencia de que *poner* x sea estricta es que no existen colas parciales (excepto $\bot$) ni colas infinitas. Se deja como ejercicio demostrar por inducción que si *poner* x es estricta, entonces *foldr poner vacia xs* $= \bot$ para todas las listas infinitas *xs*. Por tanto, el implementador de *Cola* α no ha de tener en cuenta las colas infinitas.

Se deduce que toda cola, excepto $\bot$, se puede expresar como un número finito de aplicaciones de *poner* y *elim* a la cola *vacia*. Más aún, utilizando los axiomas de *elim*, podemos eliminar las apariciones de *elim* en tal expresión. Es también obvio que las diferentes expresiones en términos de *poner* y *vacia* describen colas diferentes. Por tanto, toda cola se puede nombrar por medio de una expresión única en términos de *poner* y *vacia*. Veremos, sin embargo, que en general los valores de un tipo abstracto de datos no se pueden nombrar por medio de expresiones únicas.

8.1.3 Representaciones

Para implementar un tipo abstracto, el programador ha de describir cómo representar sus valores, definir las operaciones del tipo en términos de esta representación y mostrar que las operaciones implementadas satisfacen la especificación dada. Satisfechas estas obligaciones, el implementador es libre de elegir entre diferentes representaciones en función de la eficiencia, de la simplicidad o del gusto personal. En la sección siguiente describiremos los mecanismos suministrados por Haskell para ocultar la representación de un tipo abstracto de modo que no se permita a otras partes del guión hacer referencia a la representación escogida.

Continuemos ahora el ejemplo proporcionando dos implementaciones del tipo abstracto *Cola* α.

Primera implementación. La primera implementación utiliza listas finitas para representar colas. La implementación de las operaciones es como sigue:

$$\begin{array}{lcl} ponerc & :: & \alpha \to [\alpha] \to [\alpha] \\ ponerc\ x\ xs & = & xs +\!\!+ [x] \\ \\ vaciac & :: & [\alpha] \\ vaciac & = & [\ ] \\ \\ esVaciac & :: & [\alpha] \to Bool \\ esVaciac\ xs & = & null\ xs \end{array}$$

$$
\begin{array}{lcl}
primc & :: & [\alpha] \to \alpha \\
primc\ (x : xs) & = & x \\
\\
elimc & :: & [\alpha] \to [\alpha] \\
elimc\ (x : xs) & = & xs
\end{array}
$$

Se han dado nombres diferentes a las operaciones implementadas para distinguirlas de las abstractas. Todas las operaciones emplean tiempo constante, excepto *ponerc* que emplea $\Theta(n)$ pasos, donde n es el número de elementos de la cola. (Realmente, esta afirmación aparentemente obvia requiere ser matizada, pero pospondremos la discusión hasta el final de la sección.)

La siguiente tarea es mostrar que las operaciones implementadas satisfacen los axiomas de la especificación. En este ejemplo resulta sencillo porque existe una correspondencia uno a uno entre colas y listas finitas (añadiendo $\perp$) y, por tanto, cada uno de los axiomas se cumple cuando las operaciones implementadas se sustituyen por las correspondientes abstractas. Podemos formalizar esta correspondencia con la ayuda de dos funciones *abstr* y *reprn*, definidas mediante

$$
\begin{array}{lcl}
abstr & :: & [\alpha] \to Cola\ \alpha \\
abstr & = & foldr\ poner\ vacia \cdot reverse \\
\\
reprn & :: & Cola\ \alpha \to [\alpha] \\
reprn\ vacia & = & [\ ] \\
reprn\ (poner\ x\ xq) & = & reprn\ xq \mathbin{+\!\!+} [x]
\end{array}
$$

Por ejemplo, $abstr\ [1, 2] = poner\ 2\ (poner\ 1\ vacia)$. Se deja como ejercicio demostrar que $reprn \cdot abstr$ es la función identidad sobre listas finitas y que $abstr \cdot reprn$ es la función identidad sobre colas.

Segunda implementación. En la segunda implementación, se representa una cola xq mediante un par de listas (xs, ys) de modo que los elementos de xq son, en ese orden, los elementos de la lista $xs \mathbin{+\!\!+} reverse\ ys$. Al dividir la lista en dos partes e invertir la segunda de ellas, esperamos hacer más eficientes las operaciones sobre las colas. Impondremos también una restricción adicional sobre la representación: las colas serán representadas mediante pares (xs, ys) tales que si xs es vacía, entonces ys será también vacía. Así, no todo par de listas representa una cola; en particular, $([\ ], [x])$ no representa una cola válida. Esta restricción ha de ser conservada por los programas que crean colas. Nótese también que dos pares distintos de listas pueden representar la misma cola. Por ejemplo, tanto $([1, 2], [\ ])$ como $([1], [2])$ representa la cola $poner\ 2\ (poner\ 1\ vacia)$.

Para esta implementación podemos definir la función *abstr* mediante

$$
\begin{array}{lcl}
abstr & :: & ([\alpha], [\alpha]) \to Cola\ \alpha \\
abstr\ (xs, ys) & = & (foldr\ poner\ vacia \cdot reverse)\ (xs \mathbin{+\!\!+} reverse\ ys)
\end{array}
$$

La función *abstr* no es inyectiva; por tanto, no existe una función *reprn* que sea la inversa por la izquierda de *abstr*, y por ello no existe una correspondencia biyectiva entre colas y pares de listas finitas.

También necesitamos formalizar el hecho de que no todas las representaciones son válidas. Hacemos esto utilizando una función *valida* definida mediante

$$
\begin{array}{lcl}
valida & :: & ([\alpha], [\alpha]) \to Bool \\
valida\ (xs, ys) & = & not\ (null\ xs) \vee null\ ys
\end{array}
$$

La función *valida* formaliza lo que se conoce como un *invariante de la representación*, una idea que ya hemos encontrado en otros lugares. Por ejemplo, al tratar los números racionales de la Sección 3.5 representábamos éstos exclusivamente mediante pares de enteros que estaban en lo que se llamó forma canónica. De forma similar, los árboles binarios de búsqueda eran árboles restringidos mediante un cierto invariante de la representación. El par $(abstr, valida)$ formaliza la representación de las colas por medio de pares de listas.

He aquí la implementación de las operaciones:

$$
\begin{array}{lcl}
vaciac & = & ([\ ], [\ ]) \\
esVaciac\ (xs, ys) & = & null\ xs \\
ponerc\ x\ (ys, zs) & = & validar\ (ys, x : zs) \\
primc\ (x : xs, ys) & = & x \\
elimc\ (x : xs, ys) & = & validar\ (xs, ys) \\
\\
validar & :: & ([\alpha], [\alpha]) \to ([\alpha], [\alpha]) \\
validar\ (xs, ys) & = & \textbf{if}\ null\ xs\ \textbf{then}\ (reverse\ ys, [\ ])\ \textbf{else}\ (xs, ys)
\end{array}
$$

La función *validar* mantiene el invariante de la representación, utilizando el hecho siguiente:

$$
[\ ] \mathbin{+\!\!+} reverse\ ys \quad = \quad reverse\ ys \mathbin{+\!\!+} [\ ]
$$

Todas las operaciones emplean ahora tiempo constante excepto *elimc*. En particular, el cálculo de $poner\ x\ (ys, zs)$ emplea tiempo constante porque si ys es vacía, entonces zs es vacía e invertir $[x]$ emplea tiempo constante. La función *elim* necesita tiempo constante excepto cuando el primer elemento es una lista unitaria (de nuevo, al final de esta sección matizaremos estas afirmaciones sobre estimaciones de tiempo). La segunda implementación, por tanto, tiene una eficiencia diferente de la primera, y al final de la sección mostraremos que, de hecho, es más eficiente.

La demostración de que la segunda implementación satisface la especificación difiere también de la presentada en la sección precedente en que los axiomas, tal como están escritos, *no* se cumplen para las operaciones implementadas. Consideremos los dos axiomas de *elim*:

$$
\begin{array}{lcl}
elim\ (poner\ x\ vacia) & = & vacia \\
elim\ (poner\ x\ (poner\ y\ xq)) & = & poner\ x\ (elim\ (ponerc\ y\ xq))
\end{array}
$$

Si estos axiomas fueran satisfechos por las operaciones implementadas, entonces debería ser cierta la siguiente ecuación:

$$
\begin{array}{l}
elimc\ (ponerc\ x\ (ponerc\ y\ (ponerc\ z\ vaciac))) \quad = \\
elimc\ (ponerc\ x\ (elimc\ (ponerc\ y\ (ponerc\ z\ (ponerc\ w\ vaciac)))))
\end{array}
$$

Sin embargo, como se puede comprobar fácilmente, la parte izquierda se simplifica a $([y, x], [\,])$ y la parte derecha a $([y], [x])$. Lo que no hemos tenido en cuenta es que, aunque los dos pares de listas sean diferentes, representan la misma cola.

Este problema se resuelve admitiendo que un axioma ecuacional significa que ambos lados deben proporcionar la misma cola, no necesariamente la misma representación. Así, tan sólo deberíamos pedir que la ecuación

$$abstr \cdot elimc \cdot ponerc\, x \cdot ponerc\, y \quad = \quad abstr \cdot ponerc\, x \cdot elimc \cdot ponerc\, y$$

se verificara. En general, para todo axioma de la forma $f = g$, siendo f y g funciones que devuelven colas, requerimos solamente que

$$abstr \cdot fc \quad = \quad abstr \cdot gc$$

donde fc y gc son los resultados cuando se reemplazan las operaciones abstractas por las correspondientes operaciones implementadas. Si f y g devuelven otro tipo de valores, entonces se omite la función *abstr* en la ecuación anterior.

Existe otro problema potencial, aunque no se observa en el modo en que hemos escrito las definiciones de las operaciones. Podríamos haber escrito la definición de *poner* de la forma equivalente

$$\begin{array}{lcl} poner\, x\, ([\,], [\,]) & = & ([x], [\,]) \\ poner\, x\, (y : ys, zs) & = & (y : ys, x : zs) \end{array}$$

Esta definición es correcta debido al invariante del tipo de datos sobre representaciones válidas. Ahora, suponiendo que hubiéramos definido *poner* mediante las dos ecuaciones precedentes, consideremos el axioma

$$esVacia\, (poner\, x\, xq) \quad = \quad False$$

Reemplazando *esVacia* y *poner* por *esVaciac*, y *ponerc* y *xq* por el par (ys, zs), el axioma se convierte en

$$esVaciac\, (ponerc\, x\, (ys, zs)) \quad = \quad False$$

Esta ecuación no se satisface para todo ys y zs. Si $ys = [\,]$ y zs es una lista no vacía, la parte izquierda se reduce a $\perp$ por agotamiento de patrones en la definición de *ponerc*.La solución del problema es clara: en los axiomas, las colas deben ser reemplazadas solamente por representaciones *válidas*, de modo que la ecuación anterior sólo debe cumplirse cuando (ys, zs) es una representación válida.

Podemos resumir la situación usando la siguiente notación: Si $f = g$ es un axioma, donde f y g devuelven colas, entonces es suficiente demostrar

$$abstr \cdot fc \quad = \quad abstr \cdot gc \text{ (módulo } valida)$$

donde fc y gc resultan de reemplazar en f y g respectivamente, las operaciones abstractas por las operaciones implementadas.

Existe otro modo de proceder más modular. Es suficiente demostrar que se cumplen las siguientes ecuaciones (todas ellas módulo *valida*):

$$\begin{array}{lcl} abstr\, vaciac & = & vacia \\ abstr \cdot ponerc\, x & = & poner\, x \cdot abstr \\ primc & = & prim \cdot abstr \end{array}$$

$$\begin{array}{lcl} abstr \cdot elimc & = & elim \cdot abstr \\ esVaciac & = & esVacia \cdot abstr \end{array}$$

Utilizando el hecho de que, por hipótesis, las operaciones abstractas satisfacen los axiomas, ahora podemos demostrar que las operaciones implementadas también lo hacen (módulo *abstr* y *valida*). Por ejemplo,

$$\begin{array}{lcl} abstr \cdot elimc \cdot ponerc\, x \cdot ponerc\, y & = & elim \cdot poner\, x \cdot poner\, y \cdot abstr \\ abstr \cdot ponerc\, x \cdot elimc \cdot ponerc\, y & = & poner\, x \cdot elim \cdot poner\, y \cdot abstr \end{array}$$

sabiendo que las dos partes derechas son iguales por la segunda ecuación de *elim*.

8.1.4 Interacción y costes amortizados

Si todo lo demás es igual, la mejor implementación de un tipo abstracto de datos es aquella que hace sus operaciones tan eficientes como sea posible. Sin embargo, en la práctica se ha de alcanzar un compromiso: se puede hacer más eficiente una operación sólo al precio de encarecer otra. El diseño de una implementación requiere buen juicio para decidir dónde es más rentable esforzarse para mejorar la eficiencia. También pueden existir requisitos asociados a la especificación del tipo de datos en el sentido de que ciertas operaciones pueden requerir tener determinada eficiencia. Existe un factor de complicación cuando utilizamos un tipo abstracto de datos con un lenguaje perezoso: la evaluación de una operación particular puede tener lugar en etapas distribuidas a lo largo de todo un cómputo. Por ejemplo, consideremos la primera implementación de las colas. Dijimos antes que todas las operaciones emplean tiempo constante excepto *poner*, que emplea tiempo lineal en la longitud de la cola. De hecho, esta afirmación es correcta solamente bajo una estrategia de evaluación impaciente; bajo evaluación perezosa la situación es más complicada. Para ver por qué, consideremos la evaluación de

$$(prim \cdot poner\, n \cdot poner\, (n-1) \cdots poner\, 1)\; vacia$$

Para calcular *prim* la cola ha de ser reducida a la forma $x : xs$. Por tanto, primero la cola se reduce en $\Theta(n)$ pasos a

$$((([1] +\!\!+ [2]) +\!\!+ [3]) +\!\!+ \cdots) +\!\!+ [n]$$

y se devuelve el primer elemento en $\Theta(n)$ pasos adicionales. Posteriormente, una demanda para obtener el primer elemento de la cola resultante de eliminar un elemento necesitará otros $\Theta(n)$ pasos. En otras palabras, cuando requerimos información sobre los elementos, debe pagarse un coste total de $\Theta(n^2)$ pasos para reducir la cola.

Éste puede parecer un tema secundario —después de todo, se pagan todos los costes a lo largo del cómputo—, pero podemos imaginar un escenario en el que la diferencia sea importante. Supongamos que diseñamos un programa *interactivo* para simular colas. La escritura de tales programas se tratará en el Capítulo 10, pero la siguiente sesión dará una idea general:

```
? interact cola

Bienvenidos a las Colas. Teclee "salir" para salir.
> poner 1
ok
> poner 2
ok
> prim
1
> salir
adios
```

El programa interactivo inicializa una cola vacía. Entonces se da una secuencia de *órdenes* para añadir elementos a la cola, mostrar el primer elemento, etc. Cada orden es reconocida, bien mostrando el resultado pedido, bien escribiendo "ok".

El punto importante en este escenario es que el programador que implementa la interacción (para una cierta representación de las colas) puede ejercer un control total sobre el momento en que se realiza cada evaluación. Se puede hacer que la interacción se ejecute impacientemente, en cuyo caso la evaluación del resultado de una orden *poner* se realiza antes de su reconocimiento, o se puede hacer que se ejecute perezosamente, en cuyo caso la evaluación tiene lugar cuando se da la orden *prim* o *esVacia*, o se puede seguir alguna estrategia entre estas dos. Dependiendo de la estrategia, el usuario percibirá diferentes retrasos entre el instante en que da la orden y el instante en que recibe el reconocimiento. Dado este escenario, podemos hablar de *costes amortizados* así como de costes *en el caso peor*. Existen numerosas maneras de definir costes amortizados, pero un enfoque simple es el siguiente. Supongamos que podemos demostrar que para todo n, una secuencia de n órdenes sobre la cola emplea en el peor caso un total de $T(n)$ pasos. Entonces el promedio o coste *amortizado* por operación es $T(n)/n$ pasos. Según este esquema, cada operación recibe el mismo coste amortizado; por tanto, los costes amortizados son una medida de la eficiencia de la representación elegida como un todo. No depende de la velocidad con la que tiene lugar cada interacción, que sólo controla cuándo se pagan los costes reales.

En contraste, el coste en el caso peor de una operación individual, es decir, el máximo retraso que se puede producir entre dar una orden y recibir un reconocimiento, depende tanto de la representación como de la impaciencia de la interacción. Por definición, el coste en el caso peor $T(n)$ de una operación es el máximo número de pasos de reducción requeridos para procesar la operación cuando aparece en cualquier secuencia de n órdenes. Las estimaciones de tiempo dadas anteriormente para la primera implementación de las colas, son estimaciones en el peor caso bajo una interacción impaciente.

Veamos ahora los costes amortizados en las dos implementaciones de colas. En la primera implementación, el tiempo en el caso peor para las n operaciones sobre las colas es de $\Theta(n^2)$ pasos en total. Por ejemplo, éste es el tiempo requerido para ejecutar $n/2$ operaciones de adición seguidas de $(n/2) - 1$ de operaciones de eliminación, y después preguntar si el resultado es o no vacío. El coste amortizado por operación es, por tanto, de $\Theta(n)$ pasos. Éste es también el tiempo, en el caso peor, de cualquier operación individual.

Compárese este resultado con la segunda implementación de las colas. Aquí se invierte una lista de longitud k solamente cuando hemos realizado k operaciones de añadir desde que invertimos la lista por última vez, o desde el comienzo de la interacción. Se deduce que una secuencia de n operaciones sobre la cola necesita un total de $O(n)$ pasos, por tanto el coste amortizado por operación es $O(1)$ pasos. Por otra parte, la eficiencia en el caso peor para cualquier operación individual de la secuencia es de $\Theta(n)$ pasos. Por tanto, la segunda implementación de las colas es mejor. En la Sección 8.6 describiremos dos implementaciones de colas aún mejores.

Ejercicios

8.1.1 Las listas finitas también pueden considerarse un tipo abstracto de datos. Por ejemplo, consideremos las operaciones

$$\begin{array}{lll} nil & :: & Lista\ \alpha \\ null & :: & Lista\ \alpha \to Bool \\ cons & :: & \alpha \to Lista\ \alpha \to Lista\ \alpha \\ head & :: & Lista\ \alpha \to \alpha \\ tail & :: & Lista\ \alpha \to Lista\ \alpha \end{array}$$

Dar una especificación algebraica para las listas.

8.1.2 Siguiendo con la cuestión precedente, se puede representar un elemento de *Lista* α mediante un par (f, n) donde $f :: Nat \to \alpha$ y $n :: Nat$. El valor n es la longitud de la lista y $f\ k$ es el elemento en la posición k. Formalizar esta representación dando una función de abstracción apropiada e implementar las operaciones sobre listas.

8.1.3 Comprobar que la primera implementación de las colas satisface la especificación algebraica.

8.1.4 Utilizando las leyes de fusión de *foldr*, mostrar que $reprn \cdot foldr\ poner\ vacia = reverse$. Por tanto, concluir que $reprn \cdot abstr$ es la función identidad sobre las listas finitas. Utilizando la caracterización de que toda cola o bien es *vacia* o bien es de la forma *poner x xq* para cierto x y cierta cola xq, demostrar que $abstr \cdot reprn$ es la función identidad sobre las colas.

8.1.5 Simplificar la representación de las colas con pares de listas no imponiendo el invariante de la representación mencionado en el texto. Es decir, no se requiere que el segundo componente sea vacío si el primero lo es. ¿Cómo cambia la implementación de las operaciones?

8.1.6 Consideremos el problema de describir un editor de líneas simple. Supongamos que una *Linea* es una secuencia de caracteres $c_1 c_2 \ldots c_n$ junto con una posición p, siendo $0 \le p \le n$, llamada el *cursor*. Se requieren las siguientes operaciones sobre líneas:

$$\begin{array}{lll} vacia & :: & Linea \\ moverIni & :: & Linea \to Linea \\ moverFin & :: & Linea \to Linea \\ moverIzq & :: & Linea \to Linea \end{array}$$

```
moverDer  :: Linea → Linea
insertar  :: Char → Linea → Linea
borrar    :: Linea → Linea
```

La descripción informal de estas operaciones es la siguiente: (i) la constante *vacia* denota una línea vacía; (ii) la operación *moverIzq* mueve el cursor una posición a la izquierda (siempre que ello sea posible); análogamente para *moverDer*; (iii) la operación *moverIni* sitúa el cursor al comienzo de la línea; análogamente, la operación *moverFin* sitúa el cursor al final de la línea; (iv) la operación *borrar* borra el carácter en la posición del cursor; finalmente, (v) la operación *insertar x* inserta un carácter *x* en la posición del cursor. Dar una especificación algebraica para *Linea*. Sugerir una representación apropiada para *Linea*, implementar las operaciones dadas y demostrar que satisfacen sus especificaciones.

8.2 Módulos

Haskell suministra un mecanismo, el sistema de *módulos*, para definir tipos abstractos de datos. De hecho, se puede utilizar el mecanismo de módulos de Haskell para usos más generales, pero nos limitaremos a explicar cómo funciona para tipos abstractos de datos. He aquí un ejemplo de módulo:

```
module Cola (Cola, vacia, esVacia, poner, prim, elim) where
  newtype Cola α = Mcc ([α], [α])
  vacia   :: Cola α
  vacia   = Mcc ([ ], [ ])

  esVacia                :: Cola α → Bool
  esVacia (Mcc (xs, ys))     =     null xs

  poner                  :: α → Cola α → Cola α
  poner x (Mcc (ys, zs)  = validar (ys, x : zs)

  prim                   :: Cola α → α
  prim (Mcc (x : xs, ys)) = x

  elim                   :: Cola α → Cola α
  elim (Mcc (x : xs, ys)) = validar (xs, ys)

  validar :: ([α], [α]) → Cola α
  validar (xs, ys)
    = if null xs then Mcc (reverse ys, [ ]) else Mcc (xs, ys)
```

Este módulo contiene la segunda implementación para colas descrita en la sección precedente. Nótese que *Cola α* se implementa mediante un declaración **newtype** en lugar de como un renombramiento para pares de listas. La razón de ello se explica más adelante.

La primera línea de la definición introduce un módulo llamado *Cola*. Los nombres de módulo comienzan con una letra mayúscula. La definición también contiene una lista de *exportación*, consistente en el nombre del tipo abstracto de datos (también llamado *Cola*) junto con sus operaciones. Ningún otro tipo o valor declarado en el módulo puede ser usado fuera de él. En particular, no se exporta la constructora *Mcc*[1], de forma que no se pueden crear valores del tipo *Cola α* en guiones que usen el tipo de datos, excepto con la ayuda de las cinco operaciones nombradas. En otras palabras, la implementación de *Cola α* descrita en el módulo permanece oculta para cualquier guión que haga uso del módulo.

Para utilizar el módulo en un guión hemos de *importarlo* con una declaración

```
import Cola
```

Esta declaración se hace al comienzo de un guión. Por ejemplo, he aquí un guión que importa el tipo de datos *Cola* y define dos funciones para traducir de listas a colas y viceversa:

```
import Cola

aCola    :: [α] → Cola α
aCola    = foldr poner vacia · reverse

aLista   :: Cola α → [α]
aLista q = if esVacia q then [ ] else prim q : aLista (elim q)
```

Se deja como ejercicio demostrar que *aLista · aCola* es la función identidad sobre listas finitas.

El último punto que necesita explicación es por qué hemos decidido implementar *Cola α* como un tipo nuevo en lugar de como un renombramiento. La razón tiene que ver con el deseo de mantener la implementación oculta. Los renombramientos son mecanismos que simplemente dan nombres alternativos a tipos ya existentes. Un sinónimo y su definición son completamente intercambiables. En particular, un tipo sinónimo hereda los métodos de las clases definidas para el tipo subyacente. Por tanto, si *Cola α* fuera declarado como un renombramiento, podríamos romper la barrera de la abstracción. Por ejemplo,

```
? poner 1 (poner 2 vacia)
([2], [1])

? poner 1 (poner 2 vacia) == poner 2 (poner 1 vacia)
False
```

No teníamos intención de suministrar una comprobación de igualdad para las colas, pero dado que (==) está definido sobre ([α], [α]) siempre que lo está para α, inevitablemente tenemos una. Similarmente, dado que ([α], [α]) es una concreción de la

[1] N. del T.: Las letras corresponden a la frase *Mirar como cola*. Con ello se quiere expresar que el tipo se va a utilizar para representar a otro más abstracto. En adelante, en casos semejantes emplearemos el mismo esquema para dar nombre a estas constructoras que simplemente convierten valores de un tipo en valores de otro tipo isomorfo a él.

clase *Show* siempre que α lo sea, podemos revelar cómo están representadas las colas. Al definir *Cola* α mediante una declaración **newtype** se pueden declarar nuevas concreciones para *Cola* α, pero las viejas concreciones definidas para el tipo subyacente ya no son aplicables. De este modo, se pueden mantener las barreras de abstracción sin sacrificar la eficiencia.

Ejercicios

8.2.1 Demostrar que *aLista* · *aCola* es la función identidad sobre listas finitas.

8.3 Conjuntos

El ejemplo más característico de un tipo abstracto de datos es el tipo de los conjuntos. Los conjuntos aparecen por doquier en matemáticas y son también útiles en muchas tareas de programación. Los conjuntos se pueden representar mediante listas, listas sin repetición, listas ordenadas, árboles, funciones booleanas y otras. En diferentes circunstancias son apropiadas diferentes representaciones, dependiendo del repertorio preciso de operaciones que haya que suministrar y de la eficiencia requerida para cada una de ellas.

Hay numerosas operaciones sobre conjuntos, incluyendo las siguientes:

$$\begin{array}{lcl}
vacio & :: & Conj\ \alpha \\
esVacio & :: & Conj\ \alpha \to Bool \\
pertenece & :: & Conj\ \alpha \to \alpha \to Bool \\
insertar & :: & \alpha \to Conj\ \alpha \to Conj\ \alpha \\
borrar & :: & \alpha \to Conj\ \alpha \to Conj\ \alpha \\
\\
union & :: & Conj\ \alpha \to Conj\ \alpha \to Conj\ \alpha \\
intersec & :: & Conj\ \alpha \to Conj\ \alpha \to Conj\ \alpha \\
restar & :: & Conj\ \alpha \to Conj\ \alpha \to Conj\ \alpha
\end{array}$$

Es posible que en una aplicación particular no se desee tener estas ocho operaciones. A un tipo que se base en las cinco primeras operaciones se le suele llamar *diccionario*. Los diccionarios son útiles como mecanismo general para mantener una lista de tareas que han de ser realizadas. Las tareas se pueden realizar en cualquier orden y deben efectuarse sólo una vez. Se crean tareas nuevas una por una y se añaden a la lista; las tareas son eliminadas cuando han sido completadas.

La especificación algebraica de *Conj* α puede formularse mediante la siguiente lista de axiomas:

$$\begin{array}{lcl}
insertar\ x\ (insertar\ x\ xs) & = & insertar\ x\ xs \\
insertar\ x\ (insertar\ y\ xs) & = & insertar\ y\ (insertar\ x\ xs)
\end{array}$$

$$\begin{array}{lcl}
esVacio\ vacio & = & True \\
esVacio\ (insertar\ x\ xs) & = & False
\end{array}$$

$$\begin{array}{lcl}
pertenece\ vacio\ y & = & False \\
pertenece\ (insertar\ x\ xs)\ y & = & (x = y) \vee pertenece\ xs\ y
\end{array}$$

$$\begin{array}{l}
borrar\ x\ vacio = vacio \\
borrar\ x\ (insertar\ y\ xs) \\
\quad = \textbf{if}\ x = y\ \textbf{then}\ borrar\ x\ xs\ \textbf{else}\ insertar\ y\ (borrar\ x\ xs)
\end{array}$$

$$\begin{array}{lcl}
union\ xs\ vacio & = & xs \\
union\ xs\ (insertar\ y\ ys) & = & insertar\ y\ (union\ xs\ ys)
\end{array}$$

$$\begin{array}{l}
intersec\ xs\ vacio = vacio \\
intersec\ xs\ (insertar\ y\ ys) \\
\quad = \textbf{if}\ pertenece\ xs\ y\ \textbf{then}\ insertar\ y\ (intersec\ xs\ ys)\ \textbf{else}\ intersec\ xs\ ys
\end{array}$$

$$\begin{array}{lcl}
restar\ xs\ vacio & = & xs \\
restar\ xs\ (insertar\ y\ ys) & = & restar\ (borrar\ y\ xs)\ ys
\end{array}$$

Al igual que hicimos con las colas, supondremos también que todas las operaciones son estrictas; por tanto, sólo se pueden construir conjuntos finitos.

Los dos primeros axiomas son de naturaleza diferente a los restantes. El primero afirma que insertar un elemento dos veces en un conjunto equivale a insertarlo sólo una vez. La segunda ecuación establece que no importa en qué orden se insertan dos elementos en un conjunto. Aunque es posible expresar cualquier conjunto en términos exclusivamente de *vacio* e *insertar*, tales expresiones no son únicas. En otras palabras: no existen formas canónicas para los conjuntos.

Dos operaciones cruciales son *esVacio* y *pertenece*; sin ellas sería imposible distinguir dos conjuntos. A diferencia de las colas, no existe una operación para extraer un elemento de un conjunto. Simplemente no es posible, sin suposiciones adicionales sobre el tipo de los elementos de un conjunto, especificar una función que devuelva un elemento de un conjunto. Tal elemento no puede escogerse arbitrariamente, porque las funciones devuelven siempre el mismo resultado para un mismo argumento. Por ejemplo, la definición

$$\begin{array}{lcl}
escoger & :: & Conj\ \alpha \to \alpha \\
escoger\ (insertar\ x\ xs) & = & x
\end{array}$$

no funciona porque conduce a una inconsistencia[2]:

$$\begin{array}{lcl}
escoger\ (insertar\ x\ (insertar\ y\ xs)) & = & x \\
escoger\ (insertar\ x\ (insertar\ y\ xs)) & = & escoger\ (insertar\ y\ (insertar\ x\ xs)) = y
\end{array}$$

Por otra parte, si se restringe α a concreciones de la clase de tipos *Ord*, siempre podríamos extraer el mínimo (o el máximo) elemento de un conjunto finito no vacío.

Por una razón distinta, no se puede implementar *pertenece* a menos que *Conj* α se restrinja a aquellos tipos α que son concreciones de la clase de tipos *Eq*. Para

[2] N. del T.: La inconsistencia consiste en que es posible deducir que todos los valores del tipo de datos de los elementos colapsan a un solo valor.

determinar si un valor está en un conjunto, hemos de ser capaces de compararlo con los elementos de éste. En particular, no es posible implementar conjuntos de funciones y ser capaces de decir si dos conjuntos no vacíos de funciones son diferentes.

8.3.1 Representación mediante listas

Suponiendo que α sea una concreción de *Eq*, se pueden representar los conjuntos mediante listas. En este caso la función de abstracción es

```
abstr  ::  [α] → Conj α
abstr  =   foldr insertar vacio
```

Hay dos definiciones razonables de representaciones válidas:

```
valida xs  =  True
valida xs  =  noRepetidos xs
```

Con la primera definición, se permite cualquier lista como representación válida de dos conjuntos. En tal caso, podemos implementar las operaciones de conjuntos de un modo bastante simple. Por ejemplo,

```
pertenece xs x  =  any (== x) xs
insertar x xs   =  x : xs
borrar x xs     =  filter (≠ x) xs
union xs ys     =  xs ++ ys
restar xs ys    =  filter (not · pertenece ys) xs
```

La función *any* se define mediante

```
any    ::  (α → Bool) → [α] → Bool
any p  =   or · map p
```

La función *any p* es muy útil en programación, como lo es su función asociada *all p* = *and* · *map p*. Nótese, de paso, cómo se explota el tipo asignado a *pertenece* en la definición de *restar*. En la definición de la función currificada *pertenece* se ha realizado una elección sobre qué argumento debería ser el primero, el conjunto o el elemento. Es más conveniente colocar primero el conjunto. De hecho, la signatura de tipos

```
pertenece  ::  Conj α → (α → Bool)
```

expresa la mitad de la correspondencia entre conjuntos y funciones booleanas.

Esta representación simple presenta algunas ventajas, pero las desventajas son mayores. La principal ventaja es que *insertar* es una operación que emplea tiempo constante. Sin embargo, el problema es que la longitud de la lista de la representación puede ser mucho mayor que el tamaño del conjunto representado; por tanto, la implementación de las otras operaciones es ineficiente. Para ser más precisos, supongamos que *xs* e *ys* tienen ambas longitud n y representan sendos conjuntos de tamaño N. Entonces, en el peor caso, la evaluación de *borrar x xs* necesita $\Omega(n)$ pasos y la evaluación de *restar xs ys* necesita $\Omega(n^2)$ pasos. Dado que n puede ser arbitrariamente mayor que N, no se puede garantizar que las operaciones se ejecuten dentro de una cota temporal prefijada.

Para ser aún más precisos, deberíamos considerar costes amortizados. No es demasiado difícil demostrar que el coste del caso peor al requerir alguna información de un

conjunto (por ejemplo, mediante la operación *pertenece*) para un conjunto construido mediante una secuencia de n operaciones de diccionario es $\Theta(n^2)$ pasos; por tanto, el coste amortizado de las operaciones del diccionario es $\Theta(n)$.

Una representación mejor surge al adoptar la segunda definición de *valida* y restringir las listas de modo que todos sus elementos sean distintos. Bajo esta representación, las operaciones *insertar* y *union* se implementan como sigue:

```
insertar x xs  =  x : filter (≠ x) xs
union xs ys    =  xs ++ filter (not · pertenece xs) ys
```

Bajo esta representación, el tamaño de la lista y el tamaño del conjunto representado coinciden. Sin embargo, el tiempo de *insertar* crece hasta $\Theta(N)$, siendo N el tamaño del conjunto, y el tiempo de *union* es $\Omega(N^2)$ en el peor caso. El coste amortizado de las operaciones de diccionario es de $\Theta(N)$ pasos.

Para mejorar la representación anterior necesitamos suponer que α es una concreción de la clase de tipos *Ord*. Recordemos que *Ord* es una subclase de *Eq*. Entonces podemos representar los conjuntos mediante listas ordenadas crecientemente. El invariante de la representación *valida* es, en este caso, que *valida* se satisface para una lista si y sólo si sus elementos están en orden estrictamente creciente. Bajo esta representación, podemos mejorar la implementación de *pertenece* como sigue:

```
pertenece xs x  =  if null ys then False else (x == head ys)
                   where ys = dropWhile (< x) xs
```

Aunque el coste en tiempo es todavía $\Omega(N)$ en el peor caso, la búsqueda de x en *xs* se hace más eficiente porque no se recorre la lista más allá del primer elemento y para el cual $y \geq x$.

He aquí la implementación de *union* bajo esta nueva representación:

```
union [ ] ys        =  ys
union (x : xs) [ ]  =  x : xs

union (x : xs) (y : ys)
  | (x < y)  =  x : union xs (y : ys)
  | (x == y) =  x : union xs ys
  | (x > y)  =  y : union (x : xs) ys
```

La función *union* es similar a *fundir* excepto por el hecho de que se eliminan los elementos duplicados. Las implementaciones de *restar* e *intersec* son similares y se dejan como ejercicio.

La ventaja de la nueva representación es que los tiempos de ejecución para *union*, *intersec* y *restar* bajan hasta $\Theta(N)$ pasos. Por otra parte, los tiempos de ejecución de las operaciones de diccionario *pertenece*, *insertar* y *borrar* son $\Omega(N)$ en el peor caso. En el resto de la sección mostraremos cómo se pueden implementar de modo más eficiente las operaciones de diccionario mediante árboles.

8.3.2 Representación mediante árboles

En la Sección 6.2 hemos sugerido ya cómo se pueden usar los árboles de búsqueda para representar conjuntos. He aquí de nuevo las definiciones relevantes, ahora organizadas como un módulo:

```
module Conj (Conj, vacio, esVacio, pertenece, insertar, borrar) where
data ArBus α = Vacio      | Unir (ArBus α) α (ArBus α)

vacio  :: Conj α
vacio  = Vacio

esVacio                 :: Conj α → Bool
esVacio Vacio           = True
esVacio (Unir xt y zt)  = False

pertenece :: (Ord α) ⇒ ArBus α → α → Bool
pertenece Vacio x = False
pertenece (Unir xt y zt) x
    | (x < y)  =  pertenece xt x
    | (x == y) =  True
    | (x > y)  =  pertenece zt x

insertar :: (Ord α) ⇒ α → ArBus α → ArBus α
insertar x Vacio = Unir Vacio x Vacio
insertar x (Unir xt y zt)
    | (x < y)  =  Unir (insertar x xt) y zt
    | (x == y) =  Unir xt y zt
    | (x > y)  =  Unir xt y (insertar x zt)

borrar :: (Ord α) ⇒ α → ArBus α → ArBus α
borrar x Vacio = Vacio
borrar x (Unir xt y zt)
    | (x < y)  =  Unir (borrar x xt) y zt
    | (x == y) =  une xt zt
    | (x > y)  =  Unir xt y (borrar x zt)

une      :: ArBus α → ArBus α → ArBus α
une xt yt = if esVacio yt then xt else Unir xt y zt
            where (y, zt) = partirArbol yt

partirArbol :: ArBus α → (α, ArBus α)
partirArbol (Unir xt y zt)
  = if esVacio xt then (y, zt) else (u, Unir vt y zt)
    where (u, vt) = partirArbol xt
```

Se deja como ejercicio definir las funciones *abstr* y *valida* y por tanto formalizar la representación de conjuntos mediante árboles binarios de búsqueda.

En este módulo se definen dos funciones adicionales, *une* y *partirArbol*, pero no se encuentran entre las exportadas en la cabecera del módulo, de modo que no se pueden utilizar en los guiones que importen el módulo *Conj*. Nótese especialmente que *une* no puede ser utilizada como implementación de la función *union* sobre conjuntos. El valor de *une xt yt* está bien definido bajo la asunción de que todo elemento de *xt* es menor que cualquier elemento de *yt*. De hecho, esta representación de conjuntos mediante árboles no es particularmente útil si se desea implementar eficientemente las operaciones que no son de diccionario.

El tiempo de ejecución de las operaciones *pertenece*, *insertar* y *borrar* depende de la altura del árbol que representa el conjunto. En el peor caso, cada operación emplea $\Theta(altura\ a)$ pasos, siendo

```
altura                 :: ArBus α → Int
altura Vacio           = 0
altura (Unir xt y zt)  = 1 + (altura xt max altura zt)
```

Recordemos que un árbol de tamaño n tiene una altura h que satisface

$$\lceil \log(n+1) \rceil \leq h < n+1$$

Dado que un conjunto de tamaño n puede ser representado mediante un árbol de altura n, las operaciones de diccionario emplean $\Omega(n)$ pasos en el peor caso. Por otra parte, un conjunto de tamaño n puede ser representado mediante un árbol de altura $\log(n+1)$; por tanto, las operaciones de diccionario necesitan $O(\log n)$ en el mejor caso. Para acercar estas dos cotas sólo es necesario construir árboles cuya altura h satisfaga $h = O(\log n)$ y esto es lo que nos proponemos a continuación.

8.3.3 Árboles equilibrados

Existen varias estrategias para construir árboles de búsqueda con una altura razonablemente pequeña. La que adoptaremos aquí se denomina *árboles equilibrados en altura*, y también *árboles AVL* en honor de sus inventores, los rusos G. Adelson-Velski e Y. Landis. Los algoritmos de inserción y borrado para árboles AVL son muy similares a sus equivalentes para árboles binarios de búsqueda ordinarios. Comenzamos definiendo formalmente los árboles equilibrados.

Diremos que un árbol binario de búsqueda está *equilibrado en altura* si las alturas de los subárboles izquierdo y derecho de cada nodo difieren como máximo en uno. Esta propiedad se formaliza mediante una función *equilibrado* definida por

```
equilibrado                :: ArBus α → Bool
equilibrado Vacio          = True
equilibrado (Unir xt x zt) = abs (altura xt − altura zt) ≤ 1 ∧
                             equilibrado xt ∧ equilibrado zt
```

El árbol *xt* está equilibrado si *equilibrado xt* = *True*. En particular, un árbol binario de búsqueda perfecto está equilibrado puesto que todos los nodos tienen dos subárboles de alturas iguales. Un árbol equilibrado que represente a un conjunto puede no tener la menor altura posible de todos los árboles que representan a ese conjunto, pero su altura será siempre razonablemente pequeña. Con más precisión, si *xt* es un árbol binario de búsqueda equilibrado de tamaño n y altura h, entonces

$$h \leq 1.4404 \log(n+1) + O(1)$$

La demostración de este resultado utiliza inducción de un modo algo indirecto.

Demostración. Supongamos que $H(n)$ es la máxima altura posible de un árbol equilibrado de tamaño n. Nuestro objetivo es calcular una cota superior para $H(n)$. Lo haremos invirtiendo el problema. Supongamos que $S(h)$ es el mínimo tamaño posible de un árbol equilibrado de altura h. Tomando un árbol de tamaño n y altura $H(n)$, tenemos por tanto $S(H(n)) \leq n$, luego podemos estimar H estimando previamente S.

Dado que *Vacio* es el único árbol de altura 0, es obvio que $S(0) = 0$. De forma similar, sólo existe un árbol de altura 1, a saber, un árbol de la forma *Unir Vacio x Vacio*; por tanto, $S(1) = 1$. El árbol equilibrado de altura $h + 2$ más pequeño posible tiene dos subárboles equilibrados, uno de altura $h + 1$ y el otro de altura h. Por tanto

$$S(h+2) \quad = \quad S(h+1) + S(h) + 1$$

En este punto utilizaremos inducción. Un razonamiento inductivo simple demuestra que $S(h) = fib\ (h + 2) - 1$, siendo *fib* la función de Fibonacci. Para completar la demostración necesitaremos el siguiente hecho acerca de la función de Fibonacci, que también se puede demostrar por inducción. Sean ϕ y ψ las dos raíces de la ecuación de segundo grado $x^2 - x - 1 = 0$, esto es, $\phi = (1 + \sqrt{5}) / 2$ y $\psi = (1 - \sqrt{5}) / 2$. Entonces $fib\ n > (\phi^n - 1)/\sqrt{5}$. Por lo tanto

$$(\phi^{H(n)+2} - 1) / \sqrt{5} - 1 < fib(H(n) + 2) - 1 = S(H(n)) \leq n$$

Tomando logaritmos, obtenemos

$$(H(n) + 2) \log \phi < \log (n + 1) + O(1)$$

Dado que $\log \phi = 1/1.4404 \ldots$, se deduce el resultado buscado.

□

Representando los conjuntos por medio de árboles de búsqueda binarios equilibrados, podemos garantizar que cada una de las operaciones de diccionario emplean $O(\log n)$ pasos, siendo n el tamaño del conjunto. La implementación de las distintas operaciones siguen esencialmente el mismo esquema anterior, excepto que en diferentes momentos es necesario reequilibrar el árbol para mantener el invariante *equilibrado*. En el resto de la sección presentaremos los detalles de las operaciones de inserción y borrado.

8.3.4 Representación mediante árboles equilibrados

Querremos mantener información sobre las alturas de los subárboles; por tanto, primero enriquecemos el tipo *ArBus* α con información de alturas:

```
data ArbAVL α = Vacio | Unir Int (ArbAVL α) α (ArbAVL α)

unir          :: ArbAVL α → α → ArbAVL α → ArbAVL α
unir xt y zt  =  Unir h xt y zt
                 where h = 1 + (ht xt max ht zt)

ht            :: ArbAVL α → Int
ht Vacio      =  0
ht (Unir h xt y zt)  =  h
```


Construyendo los árboles mediante *unir* mantenemos el invariante de que la etiqueta entera de un árbol es su altura. Más aún, si *xt* y *zt* son árboles binarios de búsqueda, y el valor de *y* está entre el mayor valor de *xt* y el menor valor de *zt*, entonces *unir xt y zt* construye un árbol binario de búsqueda. Sin embargo, el resultado será un árbol equilibrado sólo si tanto *xt* como *zt* son equilibrados y sus alturas difieren como máximo en 1.

Para asegurar que sólo se construyen árboles equilibrados, necesitamos una variante de *unir* que llamaremos *unirAVL*. Su tipo es el mismo que el de *unir*:

```
unirAVL  :: ArbAVL α → α → ArbAVL α → ArbAVL α
```

Las nuevas definiciones de *insertar* y *borrar* son exactamente las mismas de antes, excepto por el hecho de que usaremos *unirAVL* en lugar de *Unir*:

```
insertar :: (Ord α) ⇒ α → ArbAVL α → ArbAVL α
insertar x Vacio = unir Vacio x Vacio
insertar x (Unir h xt y zt)
  | (x < y)  =  unirAVL (insertar x xt) y zt
  | (x == y) =  Unir h xt y zt
  | (x > y)  =  unirAVL xt y (insertar x zt)

borrar :: (Ord α) ⇒ α → ArbAVL α → ArbAVL α
borrar x Vacio = Vacio
borrar x (Unir h xt y zt)
  | (x < y)  =  unirAVL (borrar x xt) y zt
  | (x == y) =  unir xt zt
  | (x > y)  =  unirAVL xt y (borrar x zt)

unir      :: ArbAVL α → ArbAVL α → ArbAVL α
unir xt yt =  if esVacio yt then xt else unirAVL xt y zt
              where (y, zt) = partirArbol yt

partirArbol :: ArbAVL α → (α, ArbAVL α)
partirArbol (Unir h xt y zt)
  =  if esVacio xt then (y, zt) else (u, unirAVL vt y zt)
     where (u, vt) = partirArbol xt
```

Queda por implementar *unirAVL*. Dado que una sola inserción o un borrado pueden alterar como máximo en 1 la altura de cualquier subárbol de un árbol, es suficiente implementar *unirAVL xt y zt* asumiendo que tanto *xt* como *zt* son árboles de búsqueda binarios equilibrados cuyas alturas pueden diferir a lo sumo en 2. Analizaremos solamente el caso $ht\ xt = ht\ zt + 2$; el otro caso interesante, $ht\ zt = ht\ xt + 2$, es dual del anterior. Y, por supuesto, si las alturas de *xt* y *zt* difieren como máximo en uno, podemos implementar *unirAVL* como un *unir*.

Como *xt* tiene altura positiva, no es el árbol vacío; por tanto, supongamos que

$$xt \quad = \quad Unir\ n\ ut\ v\ wt$$

El valor de *unirAVL* depende de las alturas relativas de *ut* y *wt*. Hay dos casos, en primer lugar, supongamos que $ht\ wt \leq ht\ ut$; por tanto, se satisfacen las siguientes relaciones:

$$ht\ zt = ht\ xt - 2 = ht\ ut - 1 \leq ht\ wt \leq ht\ ut$$

En este caso podemos definir

```
unirAVL xt y zt  =  rotD (unir xt y zt)
```

donde *rotD* rota un árbol a la derecha:

```
rotD                          :: ArbAVL α → ArbAVL α
rotD (Unir m (Unir n ut v wt) y zt)  =  unir ut v (unir wt y zt)
```

Para comprobar que esto es correcto, razonamos como sigue:

```
      abs (ht ut − ht (unir wt y zt))
=        {definición de altura}
      abs (ht ut − 1 − (ht wt max ht zt))
=        {dado que ht zt ≤ ht wt}
      abs (ht ut − 1 − ht wt)
≤        {dado que ht ut − 1 ≤ ht wt ≤ ht ut}
      1
```

En el segundo caso, supongamos que $ht\ wt = 1 + ht\ ut$. Entonces, *wt* no puede ser el árbol vacío; por tanto, supongamos que *wt* = *Unir p rt s tt*. En este caso, *xt* tiene el siguiente aspecto:

```
xt  =  Unir n ut v wt where wt = Unir p rt s tt
```

Más aún, se satisfacen las siguientes relaciones:

$$ht\ zt = ht\ xt - 2 = ht\ ut = ht\ wt - 1 = (ht\ rt \text{ max } ht\ tt)$$

Luego podemos definir

```
unirAVL xt y zt  =  rotD (unir (rotI xt) y zt)
```

donde *rotI* rota un árbol a la izquierda:

```
rotI                          :: ArbAVL α → ArbAVL α
rotI (Unir n ut v (Unir p rt s tt))  =  unir (unir ut v rt) s tt
```

Para comprobar que esto es correcto, nótese que

```
unirAVL xt y zt  =  unir (unir ut v rt) s (unir tt y zt)
```

Por tanto, podemos razonar

```
      abs (ht (unir ut v rt) − ht (unir tt y zt))
=        {definición de altura}
      abs ((ht ut max ht rt) − (ht tt max ht zt))
=        {dado que ht tt ≤ ht zt}
      abs ((ht ut max ht rt) − ht zt)
=        {dado que ht rt ≤ ht ut}
```



```
      abs (ht ut − ht zt)
=        {dado que ht ut = ht zt}
      0
```

La definición completa de *unirAVL* es la siguiente:

```
unirAVL xt y zt
  | (hz + 1 < hx)                  ∧ (deseq xt < 0) = rotD (unir (rotI xt) y zt)
  | (hz + 1 < hx)                  = rotD (unir xt y zt)
  | (hx + 1 < hz) ∧ (0 < deseq zt) = rotI (unir xt y (rotD zt))
  | (hx + 1 < hz)                  = rotI (unir xt y zt)
  | otherwise                      = unir xt y zt
  where ht = ht xt; hz = ht zt

deseq :: ArbAVL α → Int
deseq (Unir h xt y zt)  =  ht xt − ht zt

rotD :: ArbAVL α → ArbAVL α
rotD (Unir m (Unir n ut v wt) y zt)  = unir ut v (unir wt y zt)

rotI :: ArbAVL α → ArbAVL α
rotI (Unir m ut v (Unir n rt s tt))  =  unir (unir ut v rt) s tt
```

La evaluación de *unirAVL* emplea tiempo constante, como se puede comprobar fácilmente por simple inspección.

Resumiendo: se pueden implementar las operaciones de diccionario utilizando un árbol binario de búsqueda equilibrado de modo que *pertenece*, *insertar* y *borrar* empleen todas ellas $O(\log n)$ pasos, siendo n el tamaño del conjunto.

Ejercicios

8.3.1 Suponiendo que los conjuntos se representan mediante listas en orden estrictamente creciente, implementar *intersec* y *restar*.

8.3.2 Los conjuntos se pueden representar mediante funciones booleanas. ¿Cuál es la función de abstracción en este caso? Todas las operaciones descritas en este texto, excepto una, se pueden implementar muy fácilmente con esta representación. Dar los detalles y descubrir la operación difícil.

También se puede representar un conjunto mediante un par de valores, una función booleana y un entero que represente el tamaño del conjunto. Mostrar que, bajo esta nueva representación, la operación conflictiva es fácil de implementar. ¿Esta decisión ocasiona dificultades a alguna de las otras operaciones?

8.3.3 Formalizar la representación de conjuntos mediante árboles binarios de búsqueda, dando las definiciones de *abstr* y *valida*.

8.3.4 Insertar las letras de ECUACION en un árbol inicialmente vacío y dibujar el árbol resultante. Hacerlo para árboles binarios de búsqueda y para árboles binarios de búsqueda equilibrados.

8.3.5 Supongamos que queremos representar conjuntos mediante árboles binarios de búsqueda, pero también queremos implementar *union xs ys* con una complejidad en tiempo $O(m + n)$, donde m y n son los tamaños de *xs* y *ys*. Mostrar cómo puede hacerse.

8.3.6 Recordemos que $S(h)$ denota el tamaño del menor árbol equilibrado de altura h, y $H(n)$ denota la altura del árbol más alto de tamaño n. Tanto S como H son funciones monótonas con $h \leq H(S(h))$ y $S(H(n)) \leq n$. Utilizar estos hechos para demostrar que $S(h) \leq n \equiv h \leq H(n)$. (Las funciones S y H son un ejemplo de lo que se conoce como *conexión de Galois*.)

8.3.7 ¿Por qué después de una operación de inserción en un árbol binario de búsqueda equilibrado tan sólo se requiere ser reequilibrar como máximo un árbol?

8.3.8 ¿Cuántos árboles binarios de tamaño 3 existen y cuál es su altura media? Ahora supongamos que se construye un árbol binario de búsqueda insertando alguna permutación de ABC en un árbol inicialmente vacío. ¿Cuántos árboles diferentes son posibles y cuál es su altura media? Responder a la misma cuestión asumiendo que las inserciones son equilibradas.

8.3.9 Escribir una tabla cuyas filas representen las diferentes operaciones de los conjuntos y cuyas columnas codifiquen las diferentes representaciones de los conjuntos. Utilizando la notación-O, rellenar la tabla con los tiempos empleados por cada operación bajo cada una de las representaciones.

8.4 Bolsas

Las bolsas, a diferencia de los conjuntos, son más frecuentes en la programación que en las matemáticas, donde suelen denominarse *multiconjuntos*. Una bolsa se parece a un conjunto en que el orden de sus elementos es irrelevante y se parece a una lista en que los elementos duplicados cuentan. Por ejemplo,

$$\{[1, 2, 2, 3]\} = \{[3, 2, 1, 2]\} \quad \text{y} \quad \{[1, 2, 2, 3]\} \neq \{[1, 2, 3]\}$$

Supongamos que queremos un tipo abstracto de datos *Bolsa* α, restringido a aquellos tipos α que son concreciones de *Ord*, sobre el cual se definen las siguientes operaciones:

```
crearBolsa  :: [α] → Bolsa α
esVacia     :: Bolsa α → Bool
union       :: Bolsa α → Bolsa α → Bolsa α
minBolsa    :: Bolsa α → α
elimMin     :: Bolsa α → Bolsa α
```

La función *crearBolsa* convierte una lista finita en la bolsa de sus elementos, *esVacia* determina si una bolsa está vacía, y *union* realiza la unión de dos bolsas. Por ejemplo,

$$union\ \{[1, 2, 2, 3]\}\ \{[2, 4]\} \quad = \quad \{[1, 2, 2, 2, 3, 4]\}$$

La función *minBolsa* devuelve el elemento mínimo de una bolsa no vacía y *elimMin* elimina una aparición del elemento mínimo. Estas dos últimas operaciones son significativas solamente si se restringen las bolsas a tipos que sean concreciones de *Ord*.

Daremos ahora una especificación algebraica de *Bolsa* α. Una posibilidad es la siguiente lista de axiomas:

```
esVacia (crearBolsa xs)                = null xs
union (crearBolsa xs) (crearBolsa ys)  = crearBolsa (xs ++ ys)
minBolsa (crearBolsa xs)               = minlist xs
elimMin (crearBolsa xs)                = crearBolsa (borrarMin xs)
```

La función *borrarMin* elimina una aparición del elemento mínimo de una lista no vacía; su definición se deja como ejercicio.

Otro modo de especificar *Bolsa* α es introducir dos *constructoras ocultas*, *vacia* e *insertar*, y definir el resto de las operaciones en función de aquellas. En este caso, los axiomas son

```
insertar x (insertar y xb)          = insertar y (insertar x xb)
crearBolsa                          = foldr insertar vacia
union xb vacia                      = xb
union xb (insertar y yb)            = insertar y (union xb yb)
minBolsa (insertar x vacia)         = x
minBolsa (insertar x (insertar y xb)) = x min minBolsa (insertar y xb)
```

Se deja como ejercicio la especificación de los axiomas de *elimMin*. Las funciones ocultas no forman parte de las operaciones proporcionadas por las bolsas, sino que se utilizan para ayudar a especificarlas. La primera ecuación de *insertar* dice que el orden en que se añaden dos elementos a una bolsa es irrelevante; a diferencia de la operación correspondiente para los conjuntos, añadir el mismo elemento dos veces a una bolsa es diferente de añadirlo una sola vez.

Podríamos haber proporcionado las operaciones *vacia* e *insertar* como primitivas del tipo bolsa. De hecho, junto con *minBolsa* y *elimMin*, constituyen un tipo de datos útil conocido como *cola de prioridad*. Los elementos que se añaden a una cola de prioridad poseen un cierto nivel de prioridad: los elementos de la cola son procesados en orden de prioridades. Así, *minBolsa* reemplaza a *prim* y *elimMin* reemplaza a *elim*.

Es fácil implementar bolsas utilizando listas crecientes, del mismo modo que lo hicimos para los conjuntos en la sección precedente. Ahora, la operación de unión es exactamente la operación *fundir*, dado que no se eliminan las repeticiones; la operación *crearBolsa* se implementa mediante *sort*. Bajo esta representación, los tiempos de ejecución para las diversas operaciones son, en el peor caso, los siguientes:

crearBolsa xs	emplea	$\Theta(n \log n)$ pasos
esVacia xb	emplea	$\Theta(1)$ pasos
union xb yb	emplea	$\Theta(m + n)$ pasos
minBolsa xb	emplea	$\Theta(1)$ pasos
elimMin xb	emplea	$\Theta(1)$ pasos

En la primera línea, n = *length xs* y, en la tercera línea, m y n son los tamaños de las bolsas *xb* e *yb* respectivamente.

En el resto de la sección implementaremos las operaciones sobre las bolsas de modo que tendrán las siguientes complejidades en tiempo en el peor caso:

crearBolsa xs	emplea	$\Theta(n)$ pasos
esVacia xb	emplea	$\Theta(1)$ pasos
union xb yb	emplea	$\Theta(\log m + \log n)$ pasos
minBolsa xb	emplea	$\Theta(1)$ pasos
elimMin xb	emplea	$\Theta(\log n)$ pasos

El tiempo para *crearBolsa* se reduce por un factor logarítmico, mientras que el tiempo para *elimMin* se incrementa por un factor logarítmico. La principal ventaja se obtiene, sin embargo, en la operación *union* cuya complejidad en tiempo decrece pasando de lineal a logarítmica.

8.4.1 Implementación mediante montículos

Implementaremos las bolsas mediante árboles binarios montículo, zurdos y aumentados con un valor de tamaño. Los árboles binarios montículo se definieron en la Sección 6.3, y la versión que lleva incorporado el tamaño se define mediante

```
data ArbMon α = Vacio | Unir Int α (ArbMon α) (ArbMon α)
```

Un árbol *zurdo* tiene la propiedad de que el tamaño de cada subárbol izquierdo es mayor o igual que el tamaño del correspondiente subárbol derecho. Construiremos tales árboles con la ayuda de una función *unir*, definida mediante

```
unir          :: α → ArbMon α → ArbMon α
unir x yt zt  =  if m < n then Unir p x zt yt else Unir p x yt zt
                 where m  =  tam yt
                       n  =  tam zt
                       p  =  m + n + 1

tam           :: ArbMon α → Int
tam Vacio     =  0
tam (Unir n x yt zt)  =  n
```

La función *unir* toma un valor y dos montículos y construye un nodo *Unir* incorporando la información necesaria de tamaño y permutando, si hace falta, el orden de los dos montículos para asegurar que el izquierdo es el más grande de los dos. Se deja como ejercicio definir las funciones *abstr* y *valida* que formalizan la representación de las bolsas mediante montículos zurdos.

Dejando, por el momento, *crearBolsa* a un lado, las restantes operaciones del montículo se implementan como sigue:

```
esVacia           :: ArbMon α → Bool
esVacia Vacio     =  True
esVacia (Unir n x yt zt)  =  False

minBolsa          :: ArbMon α → α
minBolsa (Unir n x yt zt)  =  x
```



```
elimMin           :: ArbMon α → ArbMon α
elimMin (Unir n x yt zt)  =  union yt zt

union             :: ArbMon α → ArbMon α → ArbMon α
union Vacio yt    =  yt
union (Unir m u vt wt) Vacio  =  Unir m u vt wt

union (Unir m u vt wt) (Unir n x yt zt)
  | u ≤ x  =  unir u vt (union wt (Unir n x yt zt))
  | x < u  =  unir x yt (union (Unir m u vt wt) zt)
```

La operación *union* se implementa fusionando dos montículos a lo largo de sus *espinas derechas*; la espina derecha de un montículo es el camino que lleva desde el nodo raíz al árbol *Vacio* situado más a la derecha. El coste de evaluar *union* es claramente proporcional a la suma de las longitudes de las espinas derechas de los dos montículos; por tanto, la siguiente tarea es estimar esta cantidad. Definimos *led* (longitud de la espina derecha) mediante

```
led           :: ArbMon α → Int
led Vacio     =  0
led (Unir n x yt zt)  =  1 + led zt
```

Probaremos que si *xt* es un árbol zurdo de tamaño *n*, entonces $led\ xt \le \lfloor \log(n+1) \rfloor$.

Demostración. Realizaremos la demostración por inducción sobre *xt*. El caso base es inmediato. Para abordar el paso de inducción, supongamos que $xt = Unir\ n\ x\ yt\ zt$, donde $tam\ yt = p$, $tam\ zt = q$ y $p \ge q$. Ahora razonamos como sigue:

$$
\begin{array}{ll}
 & led\ (Unir\ n\ x\ yt\ zt) \\
= & \quad \{\text{definición de } led\} \\
 & 1 + led\ zt \\
\le & \quad \{\text{hipótesis de inducción}\} \\
 & 1 + \lfloor \log(q+1) \rfloor \\
= & \quad \{\text{aritmética}\} \\
 & \lfloor \log(2q+2) \rfloor \\
\le & \quad \{\text{dado que } n = p + q + 1 \ge 2q + 1\} \\
 & \lfloor \log(n+1) \rfloor
\end{array}
$$

□

De este resultado se deduce que la función *union* tiene la complejidad en tiempo requerida. Dado que *elimMin* implica una sola llamada a *union*, esta función tiene también la complejidad requerida.

Queda por implementar *crearBolsa*. Utilizamos la técnica de tuplamiento de la Sección 7.4:

```
crearBolsa    :: [α] → ArbMon α
crearBolsa xs =  fst (crearDos (length xs) xs)
```

```
crearDos :: Int → [α] → (ArbMon α, [α])
crearDos n xs
  | (n == 0)     =  (Vacio, xs)
  | (n == 1)     =  (unir (head xs) Vacio Vacio, tail xs)
  | otherwise    =  (union xt yt, zs)
  where (xt, ys)  =  crearDos m xs
        (yt, zs)  =  crearDos (n − m) ys
        m         =  n div 2
```

Como vimos en la Sección 7.4, esta implementación de *crearBolsa* requiere un tiempo lineal en la longitud de la lista.

Ejercicios

8.4.1 Construir dos montículos para las letras de HOLA y TODOS y dibujar el montículo que resulta de su unión.

8.4.2 Definir *borrarMin* :: (*Ord* α) ⇒ [α] → [α].

8.4.3 Dar los axiomas de *elimMin* en términos de *insertar* y *vacia*.

8.4.4 ¿Cuál es la ventaja de tener una operación *crearBolsa* cuando se puede definir fácilmente como *foldr insertar vacia*?

8.4.5 Convertir *Bolsa* α en un tipo abstracto de datos dando una definición apropiada de módulo.

8.4.6 ¿Qué hace que n == 1 sea un caso especial en la definición de *crearBolsa*?

8.4.7 Escribir una función *ordenarBolsa* :: (*Ord* α) ⇒ *Bolsa* α → [α]. Esta función sólo deberá utilizar las operaciones especificadas para las bolsas.

8.5 Vectores flexibles

Hemos visto cómo se pueden utilizar árboles de diferentes tipos para implementar los conjuntos y las bolsas eficientemente. Ahora veremos cómo se pueden utilizar para implementar un tipo de datos llamado *vectores flexibles*. El tipo *Flex* α tiene las siguientes operaciones:

```
vacio    :: Flex α
esVacio  :: Flex α → Bool
elem     :: Flex α → Int → α
modif    :: Flex α → Int → α → Flex α
extD     :: α → Flex α → Flex α
elimD    :: Flex α → Flex α
extI     :: α → Flex α → Flex α
elimI    :: Flex α → Flex α
```


Los vectores flexibles se comportan como listas finitas cuyos elementos, identificados por su posición en la lista, se pueden acceder y reemplazar, pero también podemos añadir y eliminar elementos en los dos extremos de dicha lista. Los nombres *extD* y *elimD* son abreviaturas de *extender/eliminar por la derecha*. De modo similar, *extI* y *elimI* son abreviaturas de *extender/eliminar por la izquierda*. Los índices del vector comienzan en 0; por tanto, al añadir un elemento en el extremo izquierdo las posiciones de los elementos que ya estaban en el vector se incrementan en 1. El Ejercicio 8.5.6 trata una variante de este tipo de datos en la cual se incluyen dos cantidades adicionales, los límites inferior y superior, para utilizar un esquema de indexación más general.

Podemos basar una especificación formal de *Flex* α en las tres operaciones constructoras *vacio*, *extD* y *extI*. El primer axioma establece que

$$extD\ x \cdot extI\ y \quad = \quad extI\ y \cdot extD\ x$$

Aunque todo vector puede expresarse en términos de las constructoras, las expresiones no son únicas. Sin embargo, aplicando el axioma anterior, se puede nombrar cualquier vector mediante una expresión única en la que todas las apariciones de *extI* preceden a cualquier aparición de *extD*. Las restantes operaciones se caracterizan en términos de las constructoras. Por ejemplo,

```
elimD vacio                =  error
elimD (extD x xf)          =  xf
elimD (extI x vacio)       =  vacio
elimD (extI x (extD y xf)) =  extI x xf
elimD (extI x (extI y xf)) =  extI x (elimD (extI y xf))

elem vacio k               =  error "índice fuera de rango"
elem (extI x xf) 0         =  x
elem (extI x xf) (k + 1)   =  elem xf k

elem (extD x xf) k
  | k < n   =  elem xf k
  | k == n  =  x
  | k > n   =  error
  where n = longitud xf
```

El último axioma de *elem* depende de una función oculta *longitud*, definida mediante

```
longitud            ::  Flex α → Int
longitud vacio      =   0
longitud (extD x xf) =  longitud xf + 1
longitud (extI x xf) =  longitud xf + 1
```

La definición de *elimI* es dual de la de *elimD* mientras que *modif* es similar a *elem*. Se dejan como ejercicio las especificaciones de estas funciones y la de *esVacio*.

Si los vectores flexibles se implementan mediante listas, las operaciones *elem*, *modif*, *elimD* y *extD* emplearán en el peor caso $\Theta(n)$ pasos, siendo n el tamaño del vector. Las otras operaciones emplean tiempo constante. El coste amortizado de las operaciones del vector es también $\Theta(n)$ pasos. En el resto de esta sección describiremos una

implementación simple, diseñada por V. Dielissen y A. Kaldewaij, que garantiza que todas las operaciones pueden realizarse en $O(\log n)$ pasos.

8.5.1 Implementación

La implementación utiliza árboles binarios *casi perfectos* aumentados con un valor relacionado con su tamaño. Pospondremos la explicación de lo que es un árbol casi perfecto hasta que hayamos descrito alguna de las operaciones básicas. Se declara el tipo *Flex* α mediante

```
data Flex α  =  Vacio | Hoja α | Unir Int (Flex α) (Flex α)
```

El tipo *Flex* α es una variante de los árboles binarios, con una constructora añadida *Vacio* e información sobre su tamaño en los nodos internos.

El invariante del tipo de datos sobre los elementos de *Flex* α consta de tres condiciones. La primera es que sólo construiremos un árbol *Unir n xt yt* en el cual *n* es la suma de los tamaños de *xt* e *yt*. Esto significa que podemos definir *tam* mediante

```
tam              :: Flex α → Int
tam Vacio        =  0
tam (Hoja x)     =  1
tam (Unir n xt yt) = n
```

La segunda restricción sobre *Flex* α es que únicamente construiremos árboles formados por *hojas*. Un árbol formado por hojas es, o bien el árbol vacío *Vacio*, o bien un árbol no vacío en el cual el árbol vacío no aparece como subárbol. Formalizamos esta condición mediante

```
formadoHojas     :: Flex α → Bool
formadoHojas xt  =  esVacio xt ∨ fHojas xt
```

donde

```
fHojas Vacio        =  False
fHojas (Hoja x)     =  True
fHojas (Unir n xt yt) = fHojas xt ∧ fHojas yt

esVacio             :: Flex α → Bool
esVacio Vacio       =  True
esVacio (Hoja x)    =  False
esVacio (Unir n xt yt) = False
```

De acuerdo con la Sección 6.1, definimos la altura de un árbol formado por hojas mediante

```
altura              :: Flex α → Int
altura Vacio        =  0
altura (Hoja x)     =  0
altura (Unir n xt yt) = 1 + (altura xt max altura yt)
```

Un árbol formado por hojas de tamaño *n* tiene una altura mayor o igual que $\lceil \log n \rceil$. La tercera restricción, considerada más adelante, es que construiremos únicamente árboles casi perfectos.

Las operaciones estáticas —aquellas que no extienden o contraen el vector— se implementan como en la Sección 6.1:

```
vacio  :: Flex α
vacio  =  Vacio

elem :: Flex α → Int → α
elem (Hoja x) 0 = x
elem (Unir n xt yt) k
  = if k < m then elem xt k else elem yt (k − m)
    where m = tam xt

modif :: Flex α → Int → α → Flex α
modif (Hoja y) 0 x = Hoja x
modif (Unir n xt yt) k x
  | k < m       = Unir n (modif xt k x) yt
  | otherwise   = Unir n xt (modif yt k x)
  where m = tam xt
```

Las operaciones *elem* y *modif* emplean $O(h)$ pasos, siendo *h* la altura del árbol binario.

8.5.2 Flexibilizando las operaciones de uno de los extremos

Olvidemos por un momento las operaciones *extI* y *elimI*, y supongamos que sólo podemos extender y contraer los vectores por el extremo derecho. La estrategia básica consiste en implementar *extD* y *elimD* de tal manera que el árbol resultante tenga una altura logarítmica en su tamaño. Por esta razón, rechazamos la siguiente implementación directa de *extD*:

```
extD                :: α → Flex α → Flex α
extD x Vacio        =  Hoja x
extD x (Hoja y)     =  Unir 2 (Hoja y) (Hoja x)
extD x (Unir n xt yt) = Unir (n + 1) (Unir n xt yt) (Hoja x)
```

Esta implementación de *extD* incrementa siempre en 1 la altura del árbol; en consecuencia, construir un árbol de tamaño *n* comenzando con el árbol vacío y aplicando *extD* a *n* valores siempre tendrá como resultado un árbol de altura *n*.

En su lugar, el objetivo es incrementar la altura del árbol sólo cuando sea absolutamente necesario. En particular, añadir un elemento a un árbol perfecto —un árbol con 2^k hojas todas ellas a la misma profundidad— incrementará inevitablemente la altura del árbol. Para un árbol perfecto *xt* de tamaño 2^k, podemos definir

```
extD x xt  =  Unir (2^k + 1) xt (Hoja x)
```

Esta idea se puede extender. Diremos que un árbol es *perfecto por la izquierda* si el subárbol izquierdo de todo nodo es perfecto. Recordemos también, de la sección precedente, que un árbol *zurdo* es aquel en el que el tamaño de cada subárbol izquierdo es mayor o igual que el tamaño del correspondiente subárbol derecho. Para cualquier tamaño dado existe exactamente un árbol perfecto por la izquierda que a la vez es zurdo (lo llamaremos para abreviar un *árbol-piz*); más aún, tiene la altura más pequeña

posible. La demostración se realiza por inducción. En el caso inductivo, supongamos que xt es un árbol-piz de tamaño n, siendo $n \geq 2$. El subárbol izquierdo de xt es perfecto y tiene tamaño 2^k para algún k, y el subárbol derecho tiene tamaño $m > 0$, donde

$$m \leq 2^k \quad \text{y} \quad n = 2^k + m$$

Por tanto $2^k < n \leq 2^{k+1}$. Es decir, el subárbol izquierdo de xt tiene un tamaño que es la mayor potencia de 2 estrictamente más pequeña que n. Dado que los subárboles izquierdo y derecho de xt son también árboles-piz, el tamaño de xt está unívocamente determinado. Más aún, su altura es $k + 1 = \lceil \log n \rceil$, que es la menor posible.

Otro hecho inmediato es que si xt es un árbol-piz, entonces xt es perfecto si y sólo si sus subárboles izquierdo y derecho tienen el mismo tamaño. Usando este resultado, podemos implementar *extD* sobre árboles-piz mediante

```
extD x Vacio       = Hoja x
extD x (Hoja y)    = Unir 2 (Hoja y) (Hoja x)

extD x (Unir n xt yt)
  | tam xt == tam yt  = Unir (n + 1) (Unir n xt yt) (Hoja x)
  | otherwise         = Unir (n + 1) xt (extD x yt)
```

La función *extD* devuelve un árbol-piz siempre que su segundo argumento lo sea.

La operación *elimD* se implementa como sigue:

```
elimD                  :: Flex α → Flex α
elimD (Hoja x)         = Vacio
elimD (Unir (n + 1) xt yt) = unir n xt (elimD yt)

unir :: Int → Flex α → Flex α → Flex α
unir n xt yt
  | esVacio yt  = xt
  | otherwise   = Unir n xt yt
```

8.5.3 Flexibilizando ambos extremos

Las operaciones *extI* y *elimI* son duales de *extD* y *elimD* y pueden ser implementadas mediante árboles *perfectos por la derecha* y *diestros* (árboles-pdd). El problema es cómo combinar ambas clases de árboles. Una solución es no combinarlos y representar un vector flexible mediante un par de árboles siendo el componente izquierdo la mitad inferior del vector y el componente derecho la mitad superior. Ésta es una solución perfectamente adecuada excepto por el hecho de que conduce a un análisis por casos en *elimD* y *elimI* cuando uno de los dos árboles componentes es vacío.

La solución de Dielissen y Kaldewaij es más sutil y más elegante: se permite que las cuatro propiedades coexistan en un solo árbol. Por definición, un árbol *casi perfecto* es aquel en el que todos sus subárboles izquierdos son diestros y perfectos por la derecha y todos sus subárboles derechos son zurdos y perfectos por la izquierda. Los árboles perfectos son casi perfectos y la forma de un árbol casi perfecto *Unir n xt yt* está completamente determinada por los tamaños de xt e yt. La altura máxima posible h de un árbol casi perfecto de tamaño $n > 0$ viene dada por

$$h = 1 + \max_{0 \leq m < n} (\lceil \log m \rceil \max \lceil \log(n - m) \rceil)$$

O mejor, $h = 1 + \lceil \log n \rceil$, uno más que la altura mínima posible.

Un árbol casi perfecto no es ni un árbol-piz ni un árbol-pdd: el subárbol izquierdo de cualquier subárbol derecho del árbol es perfecto pero el subárbol izquierdo del árbol propiamente dicho no es necesariamente perfecto. De acuerdo con esto, hemos de modificar ligeramente la definición de *extD*. Comentarios duales se aplican a *extI* y el programa resultante tiene la forma

```
extD x Vacio       = Hoja x
extD x (Hoja y)    = Unir 2 (Hoja y) (Hoja x)
extD x (Unir n xt yt) = Unir (n + 1) xt (der x yt)

der :: α → Flex α → Flex α
der x (Hoja y) = Unir 2 (Hoja y) (Hoja x)
der x (Unir n xt yt)
  | tam xt == tam yt  = Unir (n + 1) (Unir n xt yt) (Hoja x)
  | otherwise         = Unir (n + 1) xt (extD x yt)
```

La definición de *extI* es dual:

```
extI x Vacio       = Hoja x
extI x (Hoja y)    = Unir 2 (Hoja x) (Hoja y)
extI x (Unir n xt yt) = Unir (n + 1) (izq x xt) yt

izq :: α → Flex α → Flex α
izq x (Hoja y) = Unir 2 (Hoja x) (Hoja y)
izq x (Unir n xt yt)
  | tam xt == tam yt  = Unir (n + 1) (Hoja x) (Unir n xt yt)
  | otherwise         = Unir (n + 1) (izq x xt) yt
```

La definición de *elimD* es la misma de antes, y se deja como ejercicio la definición de *elimI*.

Ejercicios

8.5.1 Escribir todos los árboles casi perfectos de tamaño 6.

8.5.2 Dar las funciones *abstr* y *valida* para la representación de vectores flexibles mediante árboles binarios casi perfectos.

8.5.3 Dar la implementación de *elimI*.

8.5.4 Los vectores flexibles se pueden utilizar para representar colas. Explicar cómo, y dar las complejidades en tiempo de las diversas operaciones para las colas.

8.5.5 Supongamos que se extienden los vectores flexibles con dos operaciones adicionales

```
minFlex, maxFlex :: (Ord α) ⇒ Flex α → α
```

La operación *minFlex* devuelve el elemento mínimo de un vector no vacío. Mostrar cómo se pueden implementar estas operaciones en tiempo constante extendiendo los árboles binarios con información adicional.

8.5.6 Un vector flexible acotado es un vector flexible con dos valores enteros, *inf* y *sup*, que denotan los límites inferior y superior de los índices. El extremo izquierdo de un vector *xs* es *xs* !! *inf* y el derecho es *xs* !! (*sup* − 1). Mostrar cómo se pueden implementar vectores flexibles acotados mediante árboles casi perfectos.

8.6 Colas

Para completar un cuarteto de tipos abstractos de datos, reconsideremos el tipo *Cola* α de la Sección 8.1. Recordemos que su implementación mediante pares de listas era

```
vacia            = ([ ], [ ])
esVacia (xs, ys) = null xs
poner x (ys, zs) = validar (ys, x : zs)
prim (x : xs, ys) = x
elim (x : xs, ys) = validar (xs, ys)

validar (xs, ys) = if null xs then (reverse ys, [ ]) else (xs, ys)
```

Las funciones *abstr* y *valida*, que formalizan la representación de las colas como pares de listas, se definieron así:

```
abstr (xs, ys) = (foldr poner vacia · reverse) (xs ++ reverse ys)
valida (xs, ys) = not (null xs) ∨ null ys
```

Recordemos también que el coste amortizado de las operaciones sobre las colas era de $O(1)$ pasos, mientras que el coste en el caso peor era de $\Theta(n)$ pasos, para una cola de tamaño n.

Para esta representación de las colas, el coste en el peor caso es aplicable en cualquier escenario interactivo. Si la interacción se realiza impacientemente, entonces una orden *elim* puede emplear $\Theta(n)$ pasos porque hay que invertir una lista de longitud n. Si la interacción es perezosa, entonces la ejecución de una orden *prim* después de una orden *elim* puede emplear $\Theta(n)$ pasos porque éste es el tiempo requerido para producir el primer elemento de una lista invertida.

El objetivo de esta sección es mostrar cómo lo podemos mejorar. En primer lugar, existe una implementación de las colas que asegura que se puede reducir el coste en el peor caso a $\Theta(\log n)$ pasos siempre que la interacción se realice a una cierta velocidad. En segundo lugar, existe una implementación todavía mejor que reduce el coste del peor caso hasta $O(1)$ pasos. Las dos implementaciones se deben a Chris Okasaki.

8.6.1 Primera implementación

La primera implementación representa una cola mediante cuatro componentes: un par de listas (igual que antes) y sus longitudes. Se define la función *abstr* mediante

```
abstr :: (Int, Int, [α], [α]) → Cola α
```



```
abstr (m, n, xs, ys) = foldr poner vacia (ys ++ reverse xs)
```

Definimos la función *valida* mediante

```
valida :: (Int, Int, [α], [α]) → Bool
valida (m, n, xs, ys) = (m ≥ n) ∧ m = length xs ∧ n = length ys
```

El invariante de la representación es más fuerte que antes: anteriormente sólo requeríamos que, estando *xs* vacía, también lo estuviera *ys*; ahora exigimos también que la longitud de *xs* sea mayor o igual que la de *ys*.

Excepto por la definición de *validar*, la implementación es prácticamente la misma de antes:

```
vacia            = (0, 0, [ ], [ ])
esVacia (m, n, xs, ys) = (m == 0)
poner x (m, n, xs, ys) = validar (m, n + 1, xs, x : ys)
prim (m, n, x : xs, ys) = x
elim (m + 1, n, x : xs, ys) = validar (m, n, xs, ys)
```

He aquí parte de la definición de *validar*:

```
validar (m, n, xs, ys)
  | m ≥ n     = (m, n, xs, ys)
  | m + 1 == n = ...
```

Para completarla, hemos de efectuar una *rotación*. Esta operación convierte

```
(m, n, xs, ys)  en  (m + n, 0, xs ++ reverse ys, [ ])
```

En pos de una implementación más rápida, repartimos el cálculo de *xs* ++ *reverse ys* entre las restantes operaciones, asegurando que cada pieza produce exactamente un elemento del resultado. Para llevar a cabo esta idea, recordemos que una implementación eficiente de *reverse* utilizaba un parámetro acumulador:

```
reverse xs          = revConcat xs [ ]
revConcat [ ] ys    = ys
revConcat (x : xs) ys = revConcat xs (x : ys)
```

Se utiliza la misma idea para el cálculo de *xs* ++ *reverse ys*. Definimos así la función *rot* (abreviatura de rotar) mediante

```
rot :: [α] → [α] → [α] → [α]
rot xs ys zs = xs ++ revConcat ys zs
```

Claramente, *xs* ++ *reverse ys* = *rot xs ys* [ ]. Esta expresión será calculada solamente en el caso en que *length xs* + 1 = *length ys*; por tanto, podemos implementar *rot* mediante

```
rot [ ] [y] zs        = y : zs
rot (x : xs) (y : ys) zs = x : rot xs ys (y : zs)
```

Nótese que cada paso de la evaluación de *rot* produce un nuevo elemento del resultado. Nótese también que se mantiene el invariante *length xs* + 1 = *length ys* en la llamada recursiva a *rot*.

La definición completa de *validar* es

```
validar (m, n, xs, ys)
  | m ≥ n        =  (m, n, xs, ys)
  | m + 1 = n    =  (m + n, 0, rot xs ys [ ], [ ])
```

La interacción se realiza a una velocidad intermedia entre completamente impaciente y completamente perezosa. Al procesar cada orden *poner* o *elim*, se reduce la función *validar*, pero sólo hasta forma normal débil, esto es, hasta una cuádrupla. Esta reducción emplea un paso de evaluación. Después de procesar cualquier orden, la cola está siempre en la forma de una cuádrupla; por tanto, procesar una orden *esVacia* o *poner* emplea $O(1)$ pasos.

Queda por estimar el coste de *prim* y *elim*. Tanto *prim* como *elim* se definen mediante ajuste de patrones y requieren que el tercer componente de la cuádrupla sea reducido a la forma $x : xs$. En general, el tercer componente xs_0 de la cuádrupla (m, n, xs_0, ys_0) es de la forma

$$\begin{aligned} xs_0 &= rot\ xs_1\ ys_1\ zs_1 \\ xs_1 &= rot\ xs_2\ ys_2\ zs_2 \\ &\vdots \\ xs_{n-1} &= rot\ xs_n\ ys_n\ zs_n \end{aligned}$$

para algún $n \geq 0$, donde las listas xs_n e ys_k $(0 \leq k \leq n)$, así como zs_k $(1 \leq k \leq n)$ están completamente evaluadas. Más aún, para $1 \leq k \leq n$, tenemos

$$length\ xs_k + 1 = length\ ys_k$$

gracias al invariante de *rot*. También, a partir de la definición de *rot*, se tiene para $0 \leq k < n$

$$length\ xs_{k+1} + length\ ys_{k+1} \leq length\ xs_k$$

Por tanto, $length\ xs_{k+1} < (length\ xs_k)\ /\ 2$ para $0 \leq k < n$.

Se deduce que reducir xs_0 a la forma $x : xs$ supone como mucho, un número logarítmico de reducciones de *rot* a dicha forma. Pero cada *rot* se puede reducir a esta forma en un paso; por tanto, la evaluación de *prim* y *elim* emplea $O(\log n)$ pasos para una cola de longitud n.

Para entender mejor lo que sucede, puede ser útil considerar la evaluación de las órdenes

poner 1 ; *poner* 2; ...; *poner* 14

He aquí los estados de la cola después de procesar estas órdenes sobre una cola inicialmente vacía:

```
    (0, 0, [ ], [ ])
→       {después de procesar poner 1}
    (1, 0, rot [ ] [1] [ ], [ ])
→       {después de procesar poner 2}
    (1, 1, rot [ ] [1] [ ], [2])
→       {después de procesar poner 3}
    (3, 0, rot (rot [ ] [1] [ ]) [2, 3] [ ], [ ])
→       {después de procesar poner 4, poner 5 y poner 6}
    (3, 3, rot (rot [ ] [1] [ ]) [2, 3] [ ], [4, 5, 6])
→       {después de procesar poner7}
    (7, 0, rot (rot (rot [ ] [1] [ ]) [2, 3] [ ]) [4, 5, 6, 7] [ ], [ ])
```


Ahora debería estar claro el esquema: los siguientes 7 *poner* dan lugar a

(7, 7, *rot* (*rot* (*rot* [ ] [1] [ ]) [2, 3] [ ]) [4, 5, 6, 7] [ ], [8, ..., 14])

Sólo cuando se aplica *poner* 15 se provoca otra rotación. Si aplicamos una orden *elim* a la cola anterior se producen tres reducciones de *rot*.

8.6.2 Segunda implementación

El problema consiste ahora en asegurar que ninguna operación sobre las colas emplea más de $O(1)$ pasos en el peor caso. Podemos hacerlo asegurando que *rot xs ys zs* se aplica únicamente a listas *xs* ya evaluadas; es decir, *xs* no es de la forma *rot* aplicada a algunos argumentos. Conseguimos esto forzando un paso de la evaluación de *xs* durante cada evaluación de las otras operaciones. La evaluación se fuerza manteniendo una lista extra en la representación; por otra parte, ya no es necesario conservar las longitudes de las listas.

Representamos las colas mediante ternas de listas. Definimos la función *abstr* mediante

```
abstr            ::  ([α], [α], [α]) → Cola α
abstr (xs, ys, zs)  =  (foldr poner vacia · reverse) (xs ++ reverse ys)
```

Definimos la función *valida* mediante

```
valida           ::  ([α], [α], [α]) → Bool
valida (xs, ys, zs)  =  length xs = length ys + length zs ∧ esResto xs zs
```

La comprobación *esResto xs zs* devuelve *True* si *zs* es un segmento final de *xs*. De hecho, la definición de *valida* no expresa completamente la verdad porque las representaciones de *xs* y *zs* están *compartidas* en el modelo de evaluación mediante reducción de grafos; *zs* no es una lista diferente sino que apunta a la porción no evaluada de *xs*. Si esta porción no está vacía, entonces al reducir *zs* a forma normal débil se provoca automáticamente la generación de un elemento adicional de *xs*.

He aquí las operaciones implementadas:

```
vacia                =  ([ ], [ ], [ ])
esVacia (xs, ys, zs) =  null xs
poner x (xs, ys, zs) =  validar (xs, x : ys, zs)
prim (x : xs, ys, zs) =  x
elim (x : xs, ys, zs) =  validar (xs, ys, zs)

validar (xs, ys, [ ])      =  (zs, [ ], zs)
                              where zs = rot xs ys [ ]
validar (xs, ys, z : zs)   =  (xs, ys, zs)
```

La compartición del primero y tercer elemento se logra mediante la cláusula **where** en la primera ecuación de *validar*. La evaluación de la segunda ecuación de *validar* da lugar a una evaluación más del primer componente.

La interacción se produce a la misma velocidad de antes. La evaluación de la comprobación *esVacia* (*xs, ys, zs*) supone reducir *xs* a forma normal débil. Esto puede implicar una llamada a *rot*, pero *rot* se reduce a forma normal débil en un paso de reducción; por tanto, *esVacia* emplea $O(1)$ pasos. La evaluación de *poner* implica reducir *validar* (*xs*, *x* : *ys, zs*) a forma normal débil. Al reducir *zs* a forma normal débil se emplean $O(1)$ pasos; por tanto, *poner* emplea $O(1)$ pasos. Como efecto lateral, siempre que *zs* se reduce a una lista no vacía, se calcula un elemento más de *xs*. Evaluar *prim* (*x* : *xs, ys, zs*) necesita $O(1)$ pasos porque el primer componente ya está evaluado. Finalmente, *elim* emplea $O(1)$ pasos por la misma razón que *poner*.

Ejercicios

8.6.1 Mostrar el estado final de la cola después de ejecutar

poner 1; ...; *poner* 14; *elim*; *prim*

8.6.2 ¿Qué representaciones alternativas de las colas aseguran un tiempo de ejecución en el peor caso de $\Theta(\log n)$ para cualquier operación sobre una cola de tamaño n?

8.6.3 Por definición, una *doble cola* es una cola con dos extremos de forma que se pueden añadir o eliminar elementos en cualquiera de ellos. Se puede extender la representación de colas mediante pares de listas para representar dobles colas. ¿Cómo se puede hacer, asegurando un coste amortizado de $O(1)$ pasos por cada operación sobre una doble cola?

8.7 Notas del capítulo

La especificación algebraica de tipos abstractos de datos se describe en Guttag y Horning (1978). Dos textos útiles sobre tipos de datos y sus especificaciones son Martin (1986) y Harrison (1989). La complejidad amortizada se describe en Cormen *et al.* (1990) y, en un marco funcional, en Shoenmakers (1992) y Okasaki (1996).

La presentación original de los árboles AVL se puede encontrar en Adelson-Velsky y Landis (1962); véase también Knuth (1973b). Existen muchos otros esquemas de árboles equilibrados; por ejemplo, Cormen *et al.* (1990) presenta los *árboles roji-negros* y Andersson (1993) presenta un esquema simple basado en dos operaciones *skew* y *split*.

La implementación simple de vectores flexibles se desarrolla en Dielissen y Kaldewaij (1995); véase también Hoogerwoord (1991). Se pueden encontrar las implementaciones rápidas para las colas en Okasaki (1995). La tesis de Okasaki (1996) contiene gran cantidad de material adicional sobre estructuras de datos puramente funcionales.

Capítulo 9

Listas infinitas

Aunque las listas y los árboles infinitos se han introducido a la vez que sus contrapartidas finitas, hasta ahora hemos puesto principalmente el énfasis en el cálculo con estructuras finitas. El objetivo de este capítulo es explicar con algo más de detalle qué es exactamente una lista infinita y cómo puede representarse mediante una estructura *cíclica* en un lenguaje funcional perezoso. Describiremos una nueva técnica para demostrar propiedades sobre listas infinitas, y discutiremos varias aplicaciones en las que las listas infinitas y cíclicas se pueden usar con provecho. En particular, las listas infinitas pueden ser útiles para modelar secuencias de eventos en el tiempo, de modo que proporcionan un marco adecuado en el que estudiar procesos *interactivos*.

9.1 Recapitulación

Comenzamos por resumir algunos aspectos sobre listas infinitas. Al igual que denotamos las listas finitas de enteros de m a n mediante $[m..n]$, denotamos la lista infinita de todos los enteros desde m en adelante mediante $[m..]$. Así, en una sesión podríamos tener

```
?[1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, {Interrupted!}
```

Nos llevaría un tiempo infinito imprimir esta lista completamente; por tanto, interrumpimos la ejecución.

Las listas infinitas se pueden utilizar en un programa exactamente igual que cualquier otra lista. Una función puede tomar como argumento una lista infinita o devolver como resultado una lista infinita. Por ejemplo, las siguientes afirmaciones sobre listas infinitas son todas ciertas:

$$\begin{array}{lcl} \mathit{take}\ n\ [1..] & = & [1..n] \\ [m..]\ !!\ n & = & m + n \\ \mathit{map}\ \mathit{fact}\ [0..] & = & \mathit{scanl}\ (\times)\ 1\ [1..] \end{array}$$

También es posible definir listas infinitas mediante listas intensionales. Por ejemplo, la expresión

$$[\mathit{cuadrado}\ x \mid x \leftarrow [1..],\ \mathit{odd}\ x]$$

denota las listas de todos los cuadrados de números impares. Incluso es posible tener una lista infinita de listas infinitas. Por ejemplo, en la Sección 6.6 utilizamos la lista infinita de las listas infinitas de las potencias de los números primos:

```
potPrimos  ::  [[Integer]]
potPrimos  =   map pots primos
```

Es importante *no* suponer que las listas infinitas manejadas en la programación tienen las mismas clases de propiedades que los conjuntos infinitos de las matemáticas convencionales. Por ejemplo, en la teoría de conjuntos, la expresión

$$\{x^2 \mid x \in \{0, 1, 2, 3, \ldots\}; x^2 < 10\}$$

denota el conjunto finito de todos los cuadrados menores que 10. Sin embargo, si escribimos la correspondiente lista intensional en una sesión, entonces tenemos

```
? [cuadrado x | x ← [0..], cuadrado x < 10]
[0, 1, 4, 9
```

Lo que ha sucedido aquí es que el evaluador determina los cuatro primeros elementos y después se mete en un bucle infinito buscando más elementos en la lista infinita [4, 5, 6, ...] cuyo cuadrado sea menor que 10. Aunque es razonable esperar que un matemático reconozca que no existe tal valor, no es razonable esperar el mismo grado de sofisticación de un programa de computador. En otras palabras, no supondremos que un evaluador mecánico sea capaz de realizar demostraciones, por muy triviales que puedan ser. Esto no significa que el comportamiento de un computador sea contrario a las matemáticas; únicamente, que la teoría de conjuntos no es la teoría adecuada para describir cómputos. Como hemos visto, el valor de la expresión anterior es la lista *parcial* 0:1:4:9:⊥.

Incidentalmente, no es difícil reescribir la expresión de modo que devuelva la lista finita de todos los cuadrados menores que 10. La lista intensional es equivalente a

```
filter (< 10) (map cuadrado [0..])
```

y cambiando *filter* por *takeWhile* obtenemos el resultado deseado:

```
? takeWhile (< 10) (map cuadrado [0..])
[0, 1, 4, 9]
```

El conocimiento matemático de que se puede detener la búsqueda tan pronto como se encuentra el primer cuadrado mayor que 10 está aquí codificado en la decisión del programador de utilizar *takeWhile* para escribir la expresión.

9.1.1 Inducción

Vimos en el Capítulo 4 cómo se puede usar la inducción para demostrar propiedades de las listas infinitas. Si $P(xs)$ es una ecuación que nombra la lista xs, podemos demostrar que $P(xs)$ se satisface para todas las listas infinitas xs demostrando:

Caso (⊥). Que se satisface $P(\bot)$.

Caso $(x : xs)$. Que si se satisface $P(xs)$, entonces también se satisface $P(x : xs)$ para todo x.

Obsérvese con atención que hemos impuesto una cautela en el establecimiento del principio de inducción, a saber, que $P(xs)$ ha de tomar la forma de una ecuación. Sin esta precaución, los dos casos anteriores son suficientes para demostrar que $P(xs)$ se satisface para todas las listas *parciales* xs. Con la cautela, obtenemos gratis que $P(xs)$ se satisface también para todas las listas infinitas xs. En la sección siguiente explicaremos la naturaleza de la relación entre listas parciales y listas infinitas, y mostraremos por qué es necesaria dicha cautela, o al menos una generalización de ella.

Ejercicios

9.1.1 ¿Se pueden imprimir todos los elementos de *potPrimos* aplanando primero la lista de listas mediante la aplicación de *concat*? Si no, mostrar cómo se puede definir una lista infinita que devuelva la lista de todas las potencias de todos los números primos.

9.1.2 ¿Se satisface *map* (× 3) [0..] = *iterate* (+3) 0? ¿Cuál es el valor de esta expresión si reemplazamos (=) por (==)?

9.1.3 Dar condiciones bajo las cuales se cumple *filter p xs* = *takeWhile p xs*.

9.1.4 La Criba de Eratóstenes es un método para generar la lista de todos los números primos. La idea es comenzar con la lista infinita 2, 3, ... y después aplicar repetidamente el siguiente proceso: (i) marcar el primer elemento p como primo; y (ii) borrar de la lista todos los múltiplos de p. El paso (ii) se puede implementar mediante la función *cribar*, definida por

```
cribar (p : xs)  =  [x | x ←xs, x mod p ≠ 0]
```

Escribir una expresión para la lista *primos* de todos los primos utilizando *iterate*. A partir de ella escribir expresiones que devuelvan los primeros 1000 primos y todos los primos menores que 1000.

9.2 Las listas infinitas como límites

En matemáticas, los objetos infinitos se definen como *límites* de secuencias infinitas de aproximaciones. Por ejemplo, el número irracional

$$\pi = 3.14159265358979323846\ldots$$

se puede definir como el límite de la secuencia infinita de aproximaciones racionales

$$3, 3.1, 3.14, 3.141, 3.1415, \ldots$$

El primer elemento de la secuencia, 3, es una aproximación bastante burda de π. El siguiente elemento, 3.1, es un poco mejor; 3.14 es aún mejor, y así sucesivamente.

De manera análoga, también podemos considerar una lista infinita como el límite de una secuencia de aproximaciones. Por ejemplo, la lista infinita [1..] es el límite de la secuencia infinita de listas parciales

$$\bot, 1 : \bot, 1 : 2 : \bot, 1 : 2 : 3 : \bot, \ldots$$

De nuevo, la secuencia consiste en aproximaciones cada vez mejores al límite pretendido; el primer término $\perp$ es el elemento indefinido, y por tanto una aproximación muy burda: no nos dice nada acerca del límite. El siguiente término, $1 : \perp$, es una aproximación ligeramente mejor: nos dice que el límite es una lista cuyo primer elemento es 1, pero no nos dice nada más sobre el resto de la lista. El siguiente término, $1 : 2 : \perp$, es un poco mejor, y así sucesivamente. Cada aproximación sucesivamente mejor se deriva reemplazando $\perp$ por un valor más definido que nos da más información sobre el límite.

He aquí otra secuencia de aproximaciones cuyo límite es [1..]:

$$\perp, 1 : 2 : \perp, 1 : 2 : 3 : 4 : \perp, 1 : 2 : 3 : 4 : 5 : 6 : \perp, \ldots$$

Esta secuencia es una subsecuencia de la anterior pero *converge* al mismo límite.

He aquí una secuencia de aproximaciones que *no* converge a límite alguno:

$$\perp, 1 : \perp, 2 : 1 : \perp, 3 : 2 : 1 : \perp, \ldots$$

El problema con esta secuencia es que da información contradictoria. El segundo término dice que el límite comienza por 1; sin embargo, el tercer término dice que el límite empieza por 2, y el cuarto término dice que comienza por 3, y así sucesivamente. Ninguna de las aproximaciones nos dice nada sobre el límite pretendido, y por tanto la secuencia no converge.

No deberíamos pensar que el límite de toda secuencia de listas es necesariamente una lista infinita. Por ejemplo, la secuencia

$$\perp, 1 : \perp, [1], [1], \ldots$$

en la que todos los elementos después de los dos primeros son [1], es una secuencia perfectamente válida cuyo límite es [1]. De forma similar,

$$\perp, 1 : \perp, 1 : 2 : \perp, 1 : 2 : \perp, \ldots$$

es una secuencia cuyo límite es $1 : 2 : \perp$. Las listas parciales y finitas son límites de secuencias que tienen solamente un número finito de elementos diferentes.

9.2.1 Orden de aproximación

El modo de formalizar la propiedad de que una secuencia infinita de listas parciales converge a un límite es introducir la noción de *orden de aproximación* $\sqsubseteq$ sobre los elementos del tipo. La afirmación $x \sqsubseteq y$ significa que x es una aproximación de y. El orden $\sqsubseteq$ será reflexivo ($x \sqsubseteq x$), transitivo ($x \sqsubseteq y$ e $y \sqsubseteq z$ implican $x \sqsubseteq z$), y antisimétrico ($x \sqsubseteq y$ e $y \sqsubseteq x$ implican $x = y$). Sin embargo, no es necesario que todo par de elementos hayan de ser comparables mediante $\sqsubseteq$. Por tanto, $\sqsubseteq$ es lo que se conoce como un orden *parcial*. Nótese que $\sqsubseteq$ es un operador matemático (lo mismo que =) y no un operador de Haskell que devuelve un resultado booleano.

Daremos ahora un bosquejo de cómo se define $\sqsubseteq$ para diferentes tipos. El tema, llamado *teoría de dominios*, es bastante amplio y se pueden encontrar todos los detalles en las referencias del final del capítulo. El orden de aproximación para números, booleanos, caracteres y para cualquier otro tipo enumerado, se define mediante

$$x \sqsubseteq y \quad \equiv \quad (x = \perp) \vee (x = y)$$

El operador $\vee$ situado a la derecha se debe leer como un operador matemático. La primera cláusula dice que $\perp$ es una aproximación a cualquier valor. En otras palabras, $\perp$ es el elemento *mínimo* del orden. Esto explica por qué $\perp$ se pronuncia "bottom"[1]. El valor $\perp$ es el elemento mínimo de $\sqsubseteq$ en cualquier tipo de datos. El orden anterior es *plano*. En un orden plano, o bien sabemos todo acerca de un valor, o bien no sabemos nada.

El orden de aproximación sobre el tipo (α, β) se define mediante $\perp \sqsubseteq (x, y)$ y

$$(x, y) \sqsubseteq (x', y') \quad \equiv \quad (x \sqsubseteq x') \wedge (y \sqsubseteq y')$$

Las apariciones de $\sqsubseteq$ en la parte derecha hacen referencia al orden en α y en β. El orden $\sqsubseteq$ sobre (α, β) no es plano, incluso aunque los órdenes sobre los componentes α y β lo sean. Por ejemplo, en (*Bool, Bool*) tenemos la siguiente cadena de elementos diferentes:

$$\perp \sqsubseteq (\perp, \perp) \sqsubseteq (\perp, False) \sqsubseteq (True, False)$$

El orden $\sqsubseteq$ sobre $[\alpha]$ se define mediante $\perp \sqsubseteq xs$ y

$$\begin{array}{lll} [\,] \sqsubseteq xs & \equiv & xs = [\,] \\ (x : xs) \sqsubseteq (y : ys) & \equiv & (x \sqsubseteq y) \wedge (xs \sqsubseteq ys) \end{array}$$

Deberíamos entender estas ecuaciones como una definición inductiva de una propiedad matemática, no como una definición del lenguaje Haskell. La segunda condición dice que [ ] sólo se aproxima a sí misma, y la tercera condición dice que $(x : xs)$ es una aproximación de $(y : ys)$ si y sólo si x es una aproximación de y y xs es una aproximación de ys. La primera aparición de $\sqsubseteq$ en la parte derecha hace referencia al orden de aproximación sobre α.

A modo de ejemplos, tenemos

$$[1, \perp, 3] \sqsubseteq [1, 2, 3] \quad \text{y} \quad 1 : 2 : \perp \sqsubseteq [1, 2, 3]$$

Sin embargo, $1 : 2 : \perp$ y $[1, \perp, 3]$ no están relacionadas mediante $\sqsubseteq$.

Supondremos que el orden de aproximación en cada tipo α tiene, además de las descritas anteriormente, una propiedad adicional: toda *cadena* de aproximaciones $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ ha de poseer un límite que sea también miembro de α. El límite, que denotaremos por $\lim_{n\to\infty} x_n$, se define mediante dos condiciones:

1. $x_n \sqsubseteq \lim_{n\to\infty} x_n$ para todo n. Esta condición establece que el límite es una *cota superior* de la secuencia de aproximaciones.

2. Si $x_n \sqsubseteq y$ para todo n, entonces $\lim_{n\to\infty} x_n \sqsubseteq y$. Esta condición establece que el límite es la *menor* cota superior.

La definición del límite de una cadena de aproximaciones se aplica a cualquier tipo. Los órdenes parciales que tienen esta propiedad se llaman *completos*. La teoría de dominios es el estudio de los órdenes parciales completos (OPC para abreviar). Hay una función útil para listas, *aprox*, que produce aproximaciones de una lista dada. Su definición es

[1] N. del T. *bottom* significa "la parte inferior" de algo.

$$
\begin{array}{lcl}
aprox & :: & Integer \to [\alpha] \to [\alpha] \\
aprox\ (n+1)\ [\,] & = & [\,] \\
aprox\ (n+1)\ (x : xs) & = & x : aprox\ n\ xs
\end{array}
$$

La definición de *aprox* es muy similar a la de *take* excepto por el hecho de que, debido al agotamiento de patrones, se tiene que $aprox\ 0\ xs = \bot$ para toda xs. Por ejemplo,

$$
\begin{array}{lcl}
aprox\ 0\ [1] & = & \bot \\
aprox\ 1\ [1] & = & 1 : \bot \\
aprox\ 2\ [1] & = & 1 : [\,]
\end{array}
$$

Para $n \geq 2$ tenemos que $aprox\ n\ [1] = [1]$. La propiedad importante de *aprox*, que explotaremos en la sección siguiente, es que

$$\lim_{n\to\infty} aprox\ n\ xs \quad = \quad xs$$

para todas las listas xs finitas, parciales o infinitas. La demostración, por inducción sobre xs, constituye el Ejercicio 9.2.6.

9.2.2 Funciones computables

Se pueden definir muchas funciones, pero sólo algunas de ellas son computables. Hay dos propiedades de las funciones computables que no comparten con las funciones arbitrarias. En primer lugar, una función computable f es *monótona* con respecto al orden de aproximación. Expresado con símbolos, para todo e e y,

$$x \sqsubseteq y \quad \Rightarrow \quad f\,x \sqsubseteq f\,y$$

Por así decirlo, la monotonía expresa que cuanta más información se proporciona sobre el argumento, más información se obtiene sobre el resultado.

En segundo lugar, una función computable f es *continua*, lo que significa que

$$f\,(\lim_{n\to\infty} x_n) \quad = \quad \lim_{n\to\infty} f\,x_n$$

para todas las cadenas de aproximación $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$. Podemos interpretar la continuidad como la expresión formal de que no hay sorpresas al pasar al límite.

Por ejemplo, dado que $[1\ ..]$ es el límite de secuencia

$$\bot,\ 1 : \bot,\ 1 : 2 : \bot,\ 1 : 2 : 3 : \bot, \ldots$$

Podemos calcular *map cuadrado* $[1\ ..]$ como sigue:

$$
\begin{array}{lcl}
map\ cuadrado\ \bot & = & \bot \\
map\ cuadrado\ (1 : \bot) & = & 1 : \bot \\
map\ cuadrado\ (1 : 2 : \bot) & = & 1 : 4 : \bot \\
map\ cuadrado\ (1 : 2 : 3 : \bot) & = & 1 : 4 : 9 : \bot \\
\ldots & & \\
\end{array}
$$

El límite de esta secuencia es la lista infinita $[1, 4, 9, \ldots]$ de los cuadrados de los enteros positivos. De manera similar,

$$
\begin{array}{lcl}
filter\ (<10)\ \bot & = & \bot \\
filter\ (<10)\ (1 : \bot) & = & 1 : \bot \\
filter\ (<10)\ (1 : 4 : \bot) & = & 1 : 4 : \bot \\
filter\ (<10)\ (1 : 4 : 9 : \bot) & = & 1 : 4 : 9 : \bot \\
filter\ (<10)\ (1 : 4 : 9 : 16 : \bot) & = & 1 : 4 : 9 : \bot \\
\ldots & &
\end{array}
$$

Todos los elementos de la secuencia después del tercero son iguales a $1 : 4 : 9 : \bot$ y por tanto también lo es el límite de la secuencia. Por otra parte,

$$
\begin{array}{lcl}
takeWhile\ (<10)\ \bot & = & \bot \\
takeWhile\ (<10)\ (1 : \bot) & = & 1 : \bot \\
takeWhile\ (<10)\ (1 : 4 : \bot) & = & 1 : 4 : \bot \\
takeWhile\ (<10)\ (1 : 4 : 9 : \bot) & = & 1 : 4 : 9 : \bot \\
takeWhile\ (<10)\ (1 : 4 : 9 : 16 : \bot) & = & 1 : 4 : 9 : [\,] \\
takeWhile\ (<10)\ (1 : 4 : 9 : 16 : 25 : \bot) & = & 1 : 4 : 9 : [\,] \\
\ldots & &
\end{array}
$$

El límite de esta secuencia es $1 : 4 : 9 : [\,] = [1, 4, 9]$.

9.2.3 Completitud de cadena

Supongamos que hemos demostrado por inducción que se satisface una propiedad matemática $P(xs)$ para todas las listas parciales xs. ¿Bajo qué condiciones podemos afirmar que se satisface $P(xs)$ para todas las listas infinitas xs? Dado que una lista infinita xs es el límite de una cadena de aproximaciones parciales $xs_0, xs_1, \ldots$, sabemos que $P(xs_i)$ se satisface para todo i. Es obvio, por tanto, que la propiedad que estamos buscando es que P se satisfaga en el límite si se satisface en cada aproximación. Llamamos a esta propiedad de P *completitud de cadena*. La completitud de cadena es aparentemente similar a la continuidad pero se diferencia de ella en dos aspectos. Uno es que la completitud de cadena de P no implica la propiedad contraria; esto es, que si P es falsa para todas las aproximaciones, entonces P es falsa para el límite, es decir, que $\neg P$ posee completitud de cadena. En segundo lugar, P es una propiedad matemática, no una función que devuelva un valor booleano; por tanto, no está en el mismo universo que las funciones continuas.

No todas las propiedades poseen completitud de cadena. Por ejemplo, consideremos la afirmación de que $drop\ n\ xs = \bot$ para algún n. Esta afirmación se satisface claramente para toda lista parcial, pero no es cierta para las listas infinitas.

Afortunadamente, una gran variedad de propiedades poseen completitud de cadena. En particular, una propiedad $P(xs)$ de la forma $e_1 \sqsubseteq e_2$, donde e_1 y e_2 son expresiones que nombran xs, posee completitud de cadena. Se supone quen todas las variables libres de estas expresiones están *cuantificadas universalmente*; así, si n e ys son variables que aparecen en e_1 y e_2, entonces la propiedad significa que $e_1 \sqsubseteq e_2$ para *todo* n e ys. Nótese que la propiedad anterior no cumplía este requisito porque la variable n estaba *cuantificada existencialmente*: la ecuación $drop\ n\ xs = \bot$ se establece para algún n, no para todo n.

Si $P(xs)$ y $Q(xs)$ tienen completitud de cadena, también la tiene $P(xs) \wedge Q(xs)$ (de nuevo, aquí se utiliza $\wedge$ como conectiva del cálculo proposicional, no como una operación sobre valores booleanos). En particular, una propiedad de la forma $e_1 = e_2$ tiene completitud de cadena porque es equivalente a $e_1 \sqsubseteq e_2 \wedge e_2 \sqsubseteq e_1$. Asimismo, una propiedad que consista en una lista de ecuaciones tiene completitud de cadena.

Ejercicios

9.2.1 Dibujar un grafo dirigido que muestre el orden de aproximación sobre el tipo (*Bool*, *Bool*), dibujando una arista de x a y si $x \sqsubseteq y$.

9.2.2 ¿Es computable la función $f : Int \to Int$ definida mediante $f\,\bot = 0$ y $f\,x = 1$ para $x \neq \bot$?

9.2.3 Utilizando el hecho de que todas las funciones computables son monótonas y continuas, justificar por qué no es posible definir una función computable *ordenar* que ordene correctamente una lista infinita.

9.2.4 ¿Es $[2, 4, 6, 8, \ldots]$ el límite de la siguiente secuencia de aproximaciones?

$$\bot,\ 2 : \bot,\ 2 : \bot : 6 : \bot,\ 2 : \bot : 6 : \bot : 10 : \bot, \ldots$$

Si no, ¿cuál es dicho límite?

9.2.5 Demostrar por inducción que $aprox\ n\ xs \sqsubseteq aprox\ (n+1)\ xs$ para todo n y xs.

9.2.6 Demostrar por inducción que $aprox\ n\ xs \sqsubseteq xs$ para todo n, y que, si $aprox\ n\ xs \sqsubseteq ys$ para todo n, entonces $xs \sqsubseteq ys$.

9.3 Propiedades de las listas infinitas

Lamentablemente, no siempre es suficiente el principio de inducción para establecer todas las propiedades que desearíamos sobre las listas infinitas. Consideremos, por ejemplo, la función *iterate* definida mediante

$$\begin{array}{lcl} iterate & :: & (\alpha \to \alpha) \to \alpha \to [\alpha] \\ iterate\ f\ x & = & x : iterate\ f\ (f\ x) \end{array}$$

La función *iterate* devuelve una lista infinita y satisface la siguiente ecuación:

$$iterate\ f\ x \quad = \quad x : map\ f\ (iterate\ f\ x)$$

Esta ecuación establece que dos listas infinitas son iguales. Sin embargo, no podemos emplear la inducción para demostrar el resultado porque no hay un argumento apropiado sobre el que hacer inducción.

Por tanto, ¿cómo demostramos tales resultados? Una posibilidad tentadora es demostrar que los elementos que se hallan en posiciones correspondientes son iguales; en otras palabras, hacer uso de la afirmación razonable de que dos listas xs y ys son iguales siempre que $xs\ !!\ n = ys\ !!\ n$ para todos los números naturales n. Lamentablemente, esta afirmación es falsa. Como contraejemplo, consideremos $xs = \bot$ e $ys = [\bot]$. Estas listas son diferentes pero indexándolas con cualquier número natural obtenemos $\bot$ en todos los casos.

Otra posibilidad es utilizar la función *aprox* definida en la sección precedente. Recordemos que

$$\lim_{n\to\infty} aprox\ n\ xs \quad = \quad xs$$

para todas las listas xs. Se deduce que si $aprox\ n\ xs = aprox\ n\ ys$ para todo n, entonces $xs = ys$. En general, se puede probar que $xs \sqsubseteq ys$ demostrando que se verifica $aprox\ n\ xs \sqsubseteq aprox\ n\ ys$ para todo n.

Utilicemos ahora esta idea para demostrar la propiedad de *iterate* enunciada anteriormente. Usando la definición de *iterate*, podemos formular la siguiente afirmación equivalente al resultado deseado:

$$aprox\ n\ (iterate\ f\ (f\ x)) \quad = \quad aprox\ n\ (map\ f\ (iterate\ f\ x))$$

Demostración. Demostraremos que

$$aprox\ n\ (iterate\ f\ (f\ x)) \quad = \quad aprox\ n\ (map\ f\ (iterate\ f\ x))$$

por inducción sobre n.

Caso (0). Obvio, dado que $aprox\ 0\ xs = \bot$ para cualquier lista xs.

Caso $(n+1)$. Para el lado izquierdo, razonamos como sigue:

$$\begin{array}{ll} & aprox\ (n+1)\ (iterate\ f\ (f\ x)) \\ = & \quad \{\text{definición de } iterate\} \\ & aprox\ (n+1)\ (f\ x : iterate\ f\ (f\ (f\ x))) \\ = & \quad \{\text{definición de } aprox\} \\ & f\ x : aprox\ n\ (iterate\ f\ (f\ (f\ x))) \\ = & \quad \{\text{hipótesis de inducción}\} \\ & f\ x : aprox\ n\ (map\ f\ (iterate\ f\ (f\ x))) \end{array}$$

Para el lado derecho, razonamos así:

$$\begin{array}{ll} & aprox\ (n+1)\ (map\ f\ (iterate\ f\ x)) \\ = & \quad \{\text{definición de } iterate\} \\ & aprox\ (n+1)\ (map\ f\ (x : iterate\ f\ (f\ x))) \\ = & \quad \{\text{definición de } map\} \\ & aprox\ (n+1)\ (f\ x : map\ f\ (iterate\ f\ (f\ x))) \\ = & \quad \{\text{definición de } aprox\} \\ & f\ x : aprox\ n\ (map\ f\ (iterate\ f\ (f\ x))) \end{array}$$

Los dos lados son iguales, estableciendo así el caso inductivo y completando la demostración.

□

Daremos ahora un ejemplo más. Consideremos la lista infinita *nats* definida mediante

$$\begin{array}{lcl} nats & :: & [Integer] \\ nats & = & 0 : map\ (+1)\ nats \end{array}$$

Probaremos que $nats = [0..]$ demostrando que $aprox\ n\ nats = aprox\ n\ [0..]$ para todos los números naturales n. Para hacerlo, necesitamos el siguiente resultado auxiliar:

$$aprox\ n \cdot map\ f \quad = \quad map\ f \cdot aprox\ n$$

para todo f y n. En otras palabras, $aprox\ n$ conmuta con $map\ f$. Se deja la demostración como ejercicio. El paso de inducción para nuestra propiedad es

$$
\begin{array}{ll}
 & aprox\ (n+1)\ nats \\
= & \quad \{\text{definición de } nats\} \\
 & aprox\ (n+1)\ (0 : map\ (+1)\ nats) \\
= & \quad \{\text{definición de } aprox\} \\
 & 0 : aprox\ n\ (map\ (+1)\ nats) \\
= & \quad \{\text{dado que } aprox \text{ y } map \text{ conmutan}\} \\
 & 0 : map\ (+1)\ (aprox\ n\ nats) \\
= & \quad \{\text{hipótesis de inducción}\} \\
 & 0 : map\ (+1)\ aprox\ n\ [0..] \\
= & \quad \{\text{dado que } aprox \text{ y } map \text{ conmutan}\} \\
 & 0 : aprox\ n\ (map\ (+1)\ [0..]) \\
= & \quad \{\text{definición de } map\} \\
 & 0 : aprox\ n\ [1..] \\
= & \quad \{\text{definición de } aprox\} \\
 & aprox\ (n+1)\ [0..]
\end{array}
$$

Ejercicios

9.3.1 Demostrar que $iterate\ (+a)\ b = [(i \times a) + b \quad | \ i \leftarrow [0..]]$.

9.3.2 Supongamos que definimos $fibs = 0 : 1 : [x + y \quad | \ (x, y) \leftarrow zip\ \ fibs\ (tail\ fibs)]$. Demostrar que $fibs = map\ fib\ [0..]$, donde fib es la función de Fibonacci. Establecer cuidadosamente cualquier resultado auxiliar que se utilice en la demostración.

9.3.3 Demostrar que $xs = ys$ si y sólo si $take\ n\ xs = take\ n\ ys$ para todo n.

9.4 Estructuras cíclicas

Las estructuras de datos, al igual que las funciones, se pueden definir recursivamente. Como ejemplo sencillo, consideremos la definición

```
unos  ::  [Int]
unos  =   1 : unos
```

Se tiene

```
unos  =  1 : unos
      =  1 : 1 : unos
      =  1 : 1 : 1 : unos
      ⋮
```

El nombre *unos* queda vinculado a la lista infinita [1, 1, 1, ...].

Recordemos, del Capítulo 7, que las expresiones se representan en el evaluador mediante grafos. La representación de *unos* como un grafo es particularmente interesante, porque se trata de una estructura cíclica:

La lista infinita completa se representa, por tanto, con una cantidad constante de espacio.

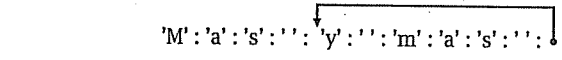
Como segundo ejemplo, consideremos la definición

```
mas  ::  String
mas  =   "Mas " ++ymas
         where ymas = "y mas " ++ymas
```

El valor de *mas* es también una lista infinita:

```
? putStr mas
Mas y mas y mas y mas y mas y m{Interrupted!}
```

Una vez evaluada, *mas* quedará representada mediante el grafo



que implica también un bucle.

Consideremos ahora tres ejemplos más del uso de estructuras cíclicas.

9.4.1 La función *repeat*

La función *repeat* se define de manera que $repeat\ x$ es la lista infinita $[x, x, x, \ldots]$, de modo que la definición de *unos* anterior es equivalente a

```
unos  =  repeat 1
```

Un modo de definir *repeat* es

```
repeat     ::  α → [α]
repeat x   =   x : repeat x
```

Esta definición es correcta, pero *no* crea una estructura cíclica. Si definimos así *unos* y *repeat* entonces, tras mostrar los cincos primeros elementos, *unos* estará representada por el grafo

$$1 : 1 : 1 : 1 : 1 : repeat\ 1$$

que no es cíclico. Si mostramos el siguiente elemento de la lista, el subtérmino *repeat* 1 será reemplazado por 1 : *repeat* 1. En consecuencia, la lista *unos* crece un poco más con cada evaluación. Por otra parte, si cambiamos la definición de *repeat* a

```
repeat x  =  xs   where xs = x : xs
```

entonces la definición de *unos* en términos de *repeat* producirá la estructura cíclica anterior.

9.4.2 La función *iterate*

He aquí una nueva definición de la función *iterate*, utilizando esta vez una estructura cíclica:

```
iterate f x  =  xs where xs = x : map f xs
```

Consideremos el término *iterate* (2 ×) 1. Los primeros pasos de la evaluación de este término son los siguientes:

```
   iterate (2 ×) 1
⇒  1 : map (2 ×)
⇒  1 : 2 : map (2 ×)
⇒  1 : 2 : 4 : map (2 ×)
```

En particular, si podemos calcular $f\,x$ en $O(1)$ pasos, entonces podemos calcular los primeros n elementos de *iterate* $f\,x$ en $O(n)$ pasos.

Si eliminamos la cláusula where de la definición anterior, obtenemos otra definición de *iterate*:

```
iterate f x  =  x : map f (iterate f x)
```

Demostramos ya antes, en este capítulo, que *iterate* satisface esta ecuación. La nueva definición no utiliza listas cíclicas y resulta ser mucho menos eficiente que la definición precedente. Considerando de nuevo el término *iterate* (2 ×) 1, los primeros pasos de la evaluación de este término son

```
   iterate (2 ×) 1
⇒  1 : map (2 ×) (iterate (2 ×) 1)
⇒  1 : 2 : map (2 ×) (map (2 ×) (iterate (2 ×) 1))
⇒  1 : 2 : 4 : map (2 ×) (map (2 ×) (map (2 ×) (iterate (2 ×) 1)))
```

Se puede ver que la evaluación de los primeros n términos requiere $\Omega(n^2)$ pasos. En este ejemplo, el uso de estructuras cíclicas es esencial para conseguir eficiencia.

9.4.3 El problema de Hamming

Un problema bien conocido, debido al matemático W. R. Hamming, consiste en escribir un programa que produzca una lista infinita de números con las siguientes propiedades:

i. La lista está en orden estrictamente creciente.

ii. La lista comienza con el número 1.

iii. Si la lista contiene el número x, entonces también contiene los números $2 \times x$, $3 \times x$ y $5 \times x$.

iv. La lista no contiene otros números.

Así, la lista requerida empieza con los números

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 16, ...

El problema de Hamming es importante, pues es típico de una clase de problemas conocidos como *problemas de cierre*. En general, un problema de cierre especifica un conjunto de *elementos iniciales* y un conjunto de *funciones generadoras*. En este caso,

se nos pide encontrar el cierre del elemento inicial bajo las funciones generadoras (2 ×), (3 ×) y (5 ×). El problema de Hamming tiene una solución particularmente eficiente porque las funciones generadoras son *monótonas* con respecto a (<); por ejemplo, (2 ×) es monótona porque $x < y$ implica $2x < 2y$; de igual manera para las otras generadoras.

La clave de la solución es utilizar la función *fundir*, que recibe dos listas infinitas de números en orden creciente y las funde en una sola lista también en orden creciente, eliminando las repeticiones:

```
fundir  ::  [Integer] → [Integer] → [Integer]
fundir (x : xs) (y : ys)
    | x < y  =  x : fundir xs (y : ys)
    | x == y =  x : fundir xs ys
    | x > y  =  y : fundir (x : xs) ys
```

A partir de *fundir* es fácil definir *hamming*:

```
hamming  ::  [Integer]
hamming  =   1 : fundir (map (2 ×) hamming)
                        (fundir (map (3 ×) hamming))
                                (map (5 ×) hamming))
```

Inicialmente, *hamming* estará representado por la siguiente estructura cíclica:

```
1: fundir (map (2 ×) • )
          (fundir (map (3 ×) • )
                  (map (5 ×) • ))
```

Una vez que se han obtenido los siete primeros elementos de *hamming*, la estructura anterior se habrá reducido a la siguiente:

```
1 : 2 :  3 :  4 : 5 :  6 : 8 : fundir (10 : map (2 ×) • )
                                      (9 : fundir (map (3 ×) • )
                                                  (10 : map (5 ×) • ))
```

Nótese que se pueden calcular los primeros n elementos de la lista *hamming* en $O(n)$ pasos.

Una generalización obvia del problema de Hamming es reemplazar los números 2, 3 y 5 por números arbitrarios positivos a, b y c. Entonces tenemos

```
hamming' a b c = 1 : fundir (map (a ×) (hamming' a b c))
                            (fundir (map (b ×) (hamming' a b c))
                                    (map (c ×) (hamming' a b c)))
```

Esta solución produce la respuesta correcta pero no forma una estructura cíclica y, por tanto, necesita $\Omega(n^2)$ pasos para calcular los primeros n elementos de la lista

resultante. Se deja como ejercicio modificar *hamming'* para que forme una estructura cíclica.

Ejercicios

9.4.1 Definimos *duda* mediante

```
duda = si
       where si        = "Si." ++ no
             no        = "No." ++ puedeSer
             puedeSer  = "Puede ser." ++ si
```

¿Qué se imprime si tecleamos *duda* en una sesión? Dibujar el grafo de la estructura cíclica que representa este valor.

9.4.2 Dibujar los cuatro grafos cíclicos que representan *hamming* tras haber obtenido los primeros 1, 2, 3 y 4 elementos.

9.4.3 Modificar *hamming'* para que forme una estructura cíclica.

9.4.4 Podemos generalizar el problema de Hamming reemplazando los multiplicandos 2, 3, 5 por una lista *as* de números positivos. Es decir, deseamos encontrar una lista creciente que comience con 1 y tal que si x está en la lista y a está en *as*, entonces $a \times x$ está en la lista. Escribir un programa que resuelva el problema de Hamming generalizado.

9.4.5 También podemos generalizar el problema de Hamming reemplazando 1 por una lista *bs* de números positivos ordenada en orden creciente. Es decir, ahora deseamos calcular una lista creciente tal que todo elemento b de *bs* esté en la lista y, si x está, entonces también han de estarlo $2x$, $3x$ y $5x$. Escribir un programa que resuelva esta versión del problema de Hamming.

9.5 Ejemplo: el juego de tijeras-piedra-papel

Nuestro siguiente ejemplo es instructivo a la vez que entretenido. No solamente introduce la idea de utilizar listas infinitas para modelizar una secuencia de interacciones entre procesos, sino que también proporciona una ilustración concreta de la necesidad de un análisis formal.

El juego de tijeras-piedra-papel resulta familiar a los niños, aunque se conoce con nombres diferentes en diferentes lugares. En el juego participan dos personas, una frente a la otra. Cada jugador se lleva la mano a la espalda en forma de, o bien una piedra (el puño cerrado), una pieza de papel (la palma abierta) o un par de tijeras (dos dedos extendidos). En un instante dado, ambos jugadores traen su mano escondida hacia delante. El ganador queda determinado por las reglas "el papel envuelve a la piedra", "la piedra machaca a las tijeras" y "las tijeras cortan el papel". Así, si el jugador uno saca una piedra y el jugador dos saca unas tijeras, entonces el jugador uno gana porque la piedra puede a las tijeras. Si ambos jugadores sacan el mismo

objeto, entonces es un empate y ninguno gana. El juego continúa de esta manera durante un número fijo de jugadas acordado de antemano.

Nuestro objetivo en esta sección es escribir un programa que juegue a este juego y que contabilice los puntos acumulados. Comenzamos introduciendo los tipos

```
data Accion  = Papel | Piedra | Tijeras
type Jugada  = (Accion, Accion)
```

Para marcar un punto, necesitamos la función

```
anotar :: Jugada → (Int, Int)
anotar (x, y)
  | x ganaA y   = (1, 0)
  | y ganaA x   = (0, 1)
  | otherwise   = (0, 0)
```

La función de comparación **ganaA** se define mediante nueve ecuaciones. Alternativamente, podemos definir

```
(ganaA)     :: Accion → Accion → Bool
x ganaA y   = (m + 1 == n) ∨ (m == n + 2)
              where m = codigo x; n = codigo y

codigo        :: Accion → Int
codigo Papel  = 0
codigo Piedra = 1
codigo Tijeras = 2
```

Cada jugador estará representado por una estrategia. Por ejemplo, una estrategia simple es, después de la primera jugada, sacar siempre lo que el jugador contrario mostró en la jugada anterior. Llamaremos a esta estrategia *reciproca*, abreviado *recip*. Otra estrategia, que llamaremos *inteligente*, consiste en determinar un movimiento analizando el número de veces que el oponente ha mostrado cualquiera de los tres posibles objetos, y calcular una respuesta apropiada basada en probabilidades.

Más adelante, consideraremos los detalles de cada estrategia particular y cómo se pueden representar. Por el momento, supongamos que el tipo *Estrategia* está definido de algún modo. El tipo de la función *jugadas* será

```
jugadas  :: (Estrategia, Estrategia) → [Jugada]
```

Esta función toma un par de estrategias y devuelve la lista infinita de jugadas que se produce cuando cada jugador sigue su estrategia asignada. Dada la función *jugadas*, podemos definir una función *juego* que determine el resultado de realizar un determinado número de jugadas del juego. Su definición es

```
juego     :: Int → (Estrategia, Estrategia) → (Int, Int)
juego n   = total · map anotar · take n · jugadas

total  :: [(Int, Int)] → (Int, Int)
total  = par (sum · map fst, sum · map snd)
```

9.5.1 Representando las estrategias

Para completar el modelo del juego, debemos decidir cómo se representan las estrategias y proporcionar la definición apropiada de *jugadas*. Existen al menos dos métodos para representar las estrategias y es instructivo compararlos con cierto detalle. En el primero, tomamos

```
type Estrategia  =  [Accion] → Accion
```

Aquí, una estrategia es una función que toma la lista (finita) de movimientos realizados por el adversario hasta el momento y devuelve un movimiento adecuado para la siguiente jugada.

Por ejemplo, se puede implementar la estrategia *recip* mediante

```
recip      ::  Estrategia
recip ms   =   if null ms then Piedra else last ms
```

El primer movimiento es arbitrario y aquí hemos elegido *Piedra*. Se puede implementar la segunda estrategia *inteligente* mediante

```
inteligente      ::  Estrategia
inteligente ms   =   if null ms then Piedra else escoger (contar ms)

contar  ::  [Accion] → (Int, Int, Int)
contar  =   foldl (⊕) (0, 0, 0)

(⊕)                      ::  (Int, Int, Int) → Accion → (Int, Int, Int)
(pa, pi, ti) ⊕ Papel     =   (pa + 1, pi, ti)
(pa, pi, ti) ⊕ Piedra    =   (pa, pi + 1, ti)
(pa, pi, ti) ⊕ Tijeras   =   (pa, pi, ti + 1)
```

De nuevo, *Piedra* es un movimiento arbitrariamente elegido como el primero. La función *contar* cuenta el número de veces que se ha realizado cada movimiento.

La función *escoger* determina un movimiento analizando los tres números *pa*, *pi* y *ti*. Para definir esta función, supongamos que *aleatorio* :: *Int* → *Int* es una función que toma un entero positivo n y devuelve algun entero m en el rango $0 \le m < n$. Una manera de implementar esta función es generar una lista infinita de números seudoaleatorios y escoger el n-ésimo elemento de esta lista, escalado apropiadamente. No daremos más detalles, pero véase el Ejercicio 9.5.1.

Ahora podemos definir *escoger* mediante

```
escoger :: (Int, Int, Int) → Accion
escoger (pa, pi, ti)
  | m < pa        =  Tijeras
  | m < pa + pi   =  Papel
  | otherwise     =  Piedra
  where m = aleatorio (pa + pi + ti)
```

La función *escoger* determina un movimiento adecuado dependiendo de si m cae en uno de los tres rangos:

$$0 \le m < pa \quad \text{o} \quad pa \le m < pa + pi \quad \text{o} \quad pa + pi \le m < pa + pi + ti$$

Por ejemplo, si *pa* es grande, entonces *inteligente* escogerá *Tijeras* con una gran probabilidad (porque las tijeras cortan al papel); si *pi* es grande, es muy probable que *inteligente* escoja *Papel* (porque el papel envuelve a la piedra); y así sucesivamente.

Ahora podemos definir la función *jugadas*:

```
jugadas        ::  (Estrategia, Estrategia) → [Jugada]
jugadas (f, g) =   (map last · tail · iterate (extender (f, g))) [ ]

extender          ::  (Estrategia, Estrategia) → [Jugada] → [Jugada]
extender (f, g) rs  =  rs ++ [(f (map snd rs), g (map fst rs))]
```

La función *extender* añade un nuevo par de movimientos a la lista de las jugadas existentes, y *jugadas* genera la lista infinita de jugadas aplicando repetidamente *extender* a una lista inicialmente vacía.

La definición de *jugadas* es laboriosa. Más importante aún, no es muy eficiente. Supongamos que una estrategia emplea un tiempo proporcional a la longitud de la entrada para calcular su resultado. Se deduce que *extender* empleará $\Omega(n)$ pasos para actualizar un juego de n jugadas con una más. Por tanto, calcular un juego de N jugadas necesita $\Omega(N^2)$ pasos.

Una segunda representación. Para comparar, consideremos ahora otra forma razonable de representar las estrategias. Ahora definimos

```
type Estrategia  =  [Accion] → [Accion]
```

En la nueva representación, una estrategia es una función que toma la lista infinita de movimientos hechos por el adversario y devuelve la lista infinita de las respuestas. Por ejemplo, ahora implementamos la estrategia *recip* mediante la ecuación

```
recip ms  =  Piedra : ms
```

La estrategia devuelve *Piedra* la primera vez y después devuelve exactamente el movimiento hecho por el adversario en la jugada anterior. Obsérvese que esta versión de *recip* produce cada acción sucesiva en un tiempo constante.

Podemos reprogramar la estrategia *inteligente* como sigue:

```
inteligente xs  =  Piedra : map escoger (contadores xs)
contadores      =  tail · scanl (⊕) (0, 0, 0)
```

La función *contadores* calcula las sumas acumuladas de los tres posibles movimientos. La estrategia *inteligente* es también eficiente porque produce cada acción sucesiva en un tiempo constante.

Con nuestro nuevo modelo de estrategias podemos redefinir la función *jugadas* del modo siguiente:

```
jugadas (f, g)  =  zip xs ys
                   where xs = f ys; ys = g xs
```

Aquí, *xs* es la lista de respuestas calculadas por f en respuesta a la lista *ys* que, a su vez, es la lista de las respuestas realizadas por g en respuesta a *xs*. Para asegurar que *jugadas* (f, g) genera una lista infinita de movimientos bien definidos (es decir, que no

es ⊥) necesitamos que el par de ecuaciones mutuamente recursivas, $xs = f\,ys$ y $ys = g\,xs$, generen listas infinitas de elementos bien definidos. Discutiremos este aspecto más adelante. Si f y g satisfacen este requisito, entonces la nueva definición de *jugadas* calcula los primeros n movimientos del juego en $O(n)$ pasos, suponiendo que f y g calculan cada nuevo movimiento en un tiempo constante. Así, el segundo método para modelizar estrategias conduce a un programa más eficiente que el anterior.

9.5.2 Haciendo trampas

Lamentablemente, sin embargo, existe un fallo crucial en la segunda representación: ¡no ofrece protección contra una estrategia que haga trampas! Consideremos la estrategia

$$trampas\ xs = map\ ganar\ xs$$

donde

$$\begin{array}{lcl} ganar & :: & Accion \to Accion \\ ganar\ Papel & = & Tijeras \\ ganar\ Piedra & = & Papel \\ ganar\ Tijeras & = & Piedra \end{array}$$

La primera respuesta de *trampas* es el movimiento que garantiza ganar al movimiento del primer oponente; ocurre del mismo modo para los siguientes movimientos. Para ver que no se puede impedir que *trampas* subvierta el juego, consideremos una secuencia en la que juega contra *recip*. Supongamos que definimos xs e ys mediante $xs = trampas\ ys$ e $ys = recip\ xs$. Los valores de xs e ys son, respectivamente, los límites de la cadena de aproximaciones $xs_0, xs_1, \ldots$ e $ys_0, ys_1, \ldots$, donde $xs_0 = ys_0 = \bot$ y

$$\begin{array}{lcl} xs_{n+1} & = & trampas\ ys_n \\ ys_{n+1} & = & recip\ xs_n \end{array}$$

Ahora, tenemos

$$\begin{array}{lclcl} xs_1 & = & trampas\ \bot & = & \bot \\ ys_1 & = & recip\ \bot & = & Piedra : \bot \\ & & & & \\ ys_2 & = & trampas\ (Piedra : \bot) & = & Papel : \bot \\ ys_2 & = & recip\ \bot & = & Piedra : \bot \\ & & & & \\ xs_3 & = & trampas\ (Piedra : \bot) & = & Papel : \bot \\ ys_3 & = & recip\ (Papel : \bot) & = & Piedra : Papel : \bot \\ & & & & \\ xs_4 & = & trampas\ (Piedra : Papel : \bot) & = & Papel : Tijeras : \bot \\ ys_4 & = & recip\ (Papel : \bot) & = & Piedra : Papel : \bot \end{array}$$

Continuando de este modo, vemos que los límites de esta secuencia son realmente listas infinitas de movimientos bien definidos y, que además, *trampas* siempre gana.

La estrategia *trampas* no es la única que puede hacer trampas. He aquí tres más:

$$\begin{array}{lcl} masTrampas\ xs & = & ganar\ (head\ xs) : masTrampas\ (tail\ xs) \\ unaTrampa\ xs & = & ganar\ (head\ xs) : recip\ (tail\ xs) \\ hipocrita\ xs & = & take\ 2\ (recip\ xs) +\!\!+ trampas\ (drop\ 2\ xs) \end{array}$$

La estrategia *masTrampas* es igual a *trampas* sobre listas infinitas, en tanto que se tiene $trampas\ \bot = \bot$ y $masTrampas\ \bot = \bot : \bot : \bot : \ldots$ La estrategia *unaTrampa* hace trampas en el primer movimiento, pero después actúa como *recip*. La estrategia *hipocrita* juega limpio durante dos movimientos y después hace trampas.

¿Podemos hallar un modo de protegernos contra estas estrategias? Para contestar esta pregunta, necesitamos estudiar más atentamente qué es una estrategia honrada. Informalmente hablando, una estrategia f es honrada si el primer movimiento se calcula careciendo de cualquier información sobre el primer movimiento del adversario, el segundo movimiento se calcula sin ninguna información sobre el segundo movimiento del adversario y así sucesivamente. Más aún, cada uno de estos movimientos debe estar bien definido, siempre que los movimientos del adversario estén bien definidos.

Con más precisión, llamamos $\mathrm{def}\ (n, xs)$ a la afirmación de que los primeros n movimientos de la lista xs están bien definidos; entonces f es *honrada* si

$$\mathrm{def}\ (n, xs) \Rightarrow \mathrm{def}\ (n+1, f\,xs)$$

para todo n y toda lista xs. Es fácil demostrar que *recip* es honrada. Por otra parte, *trampas* no es honrada porque $\mathrm{def}\ (0, \bot)$ se satisface pero $\mathrm{def}\ (1, \bot)$ no lo hace, y

$$\mathrm{def}\ (1, trampas\ \bot) \equiv \mathrm{def}\ (1, \bot)$$

Es fácil comprobar que las otras estrategias que hacen trampa no son honradas. Tampoco lo es, por cierto, la estrategia *somnolienta*, siendo

$$somnolienta\ xs = repeat\ indefinido$$

Aunque *somnolienta* no hace realmente trampas, nunca devuelve un movimiento bien definido.

Una vez identificada la fuente de los comportamientos deshonestos, ¿podemos asegurar que sólo serán admitidas al juego las estrategias honradas? La respuesta es un rotundo sí: si bien no es posible que un evaluador mecánico reconozca que se hacen trampas (del mismo modo que no puede reconocer ⊥o estrategias que no devuelven movimientos bien definidos), sí que se puede definir una función *policia* tal que, si f es una estrategia honrada y xs es una secuencia infinita de movimientos bien definidos, entonces $policia\ f\ xs = f\ xs$. Por otra parte, si f no es honrada en algún momento, entonces el juego finaliza en ese instante con el valor ⊥. En el Ejercicio 9.5.8 se da una caracterización más precisa. Desde el punto de vista operacional, *policia* funciona haciendo que f devuelva el primer elemento (bien definido) de su salida antes de dar a f el primer elemento de su entrada. Para los restante elementos se comporta del mismo modo. La definición es

$$policia\ f\ xs = ys\ \textbf{where}\ ys = f\ (sincroniza\ xs\ ys)$$

$$\begin{array}{l} sincroniza :: [Accion] \to [Accion] \to [Accion] \\ sincroniza\ (x : xs)\ (y : ys) \\ \quad = \textbf{if}\ definida\ y\ \textbf{then}\ x : sincroniza\ xs\ ys\ \textbf{else}\ indefinido \end{array}$$

La función *definida y* devuelve *True* si y es un movimiento bien definido y ⊥ en otro caso; se deja como ejercicio encontrar una definición adecuada. La demostración, debida a Berend Sijtsma, de que esta implementación satisface su especificación es bastante laboriosa; por tanto, la relegamos a los ejercicios.

Se deduce del análisis anterior que, para impedir que se hagan trampas, debemos reescribir la definición de *jugadas* de modo siguiente:

$$\begin{aligned} jugadas\ (f, g) &= zip\ xs\ ys \\ &\quad \textbf{where}\ xs = policia\ f\ ys;\ ys = policia\ g\ xs \end{aligned}$$

Ejercicios

9.5.1 Un modo de generar una lista infinita de números seudoaleatorios es mediante

$$\begin{aligned} aleatorios &:: [Int] \\ aleatorios &= iterate\ f\ semilla \\ &\quad \textbf{where}\ f\,x = (a \times x + c)\ \textbf{mod}\ m \end{aligned}$$

Los cuatro números mágicos que aparecen son: (i) el valor de partida *semilla*; (ii) el multiplicador a; (iii) el incremento c; (iv) el módulo m. Este método se conoce como el *método de la congruencia lineal*. No deben elegirse al azar estos cuatro números; para escogerlos de un modo adecuado véase Knuth (1981). Utilizar este método para construir una implementación adecuada de la función *aleatorio* del texto.

9.5.2 Dar una definición apropiada de $definida :: Accion \to Bool$.

9.5.3 Supongamos que, en la definición de *policia*, se reemplaza la definición de *sincroniza* por otra más sencilla:

$$sincroniza\ (x : xs)\ (y : ys) = x : sincroniza\ xs\ ys$$

Demostrar que la nueva versión de *policia* permite que la estrategia *masTrampas* escape a la detección.

9.5.4 Se puede definir inductivamente la propiedad def (n, xs) haciendo que def $(0, xs)$ sea cierto y

$$\text{def}\ (n + 1, xs) = \text{definido}\ (head\ xs) \wedge \text{def}\ (n, tail\ xs)$$

La propiedad definido (x) es cierta si x es un movimiento definido y es falsa en otro caso. Demostrar los siguientes hechos:

$$\begin{aligned} &\text{def}\ (n, xs) \equiv \text{def}\ (n, aprox\ n\ xs) \\ &\text{def}\ (n, xs) \wedge xs \sqsubseteq ys \Rightarrow aprox\ n\ xs = aprox\ n\ ys \\ &\text{def}\ (n, xs) \wedge xs \sqsubseteq ys \Rightarrow \text{def}\ (n, ys) \end{aligned}$$

9.5.5 Utilizando los resultados del ejercicio precedente, demostrar que si f es honrada y def (n, xs), entonces $aprox\ (n + 1)\ (f\,xs) = aprox\ (n + 1)\ (f\ (aprox\ n\ xs))$.

9.5.6 Demostrar que

$$\text{def}\ (n, zs) \Rightarrow aprox\ n\ (sincroniza\ xs\ ys) = aprox\ n\ xs$$

para todas las listas zs y todas las listas parciales (y también infinitas) xs.

9.5.7 Utilizar los dos ejercicios precedentes para demostrar que, si f es honrada y def (n, xs), entonces

$$aprox\ (n + 1)\ (policia\ f\ xs) = aprox\ (n + 1)\ (f\,xs)$$

A partir de ello, demostrar que $(policia\ f\ xs)\ !!\ k = (f\,xs)\ !!\ k$ para todo k $(0 \le k \le n)$.

9.5.8 Demostrar que

$$\text{def}\ (n, f\,xs) \wedge \neg\text{def}\ (n + 1, f\,xs) \Rightarrow (policia\ f\ xs)\ !!\ n = \bot$$

Explicar por qué este hecho es demasiado débil para detectar que se hacen trampas. (Ayuda: hacer que xs sea una lista infinita de movimientos bien definidos y $f = trampas$.) El resultado que estamos buscando en los ejercicios siguientes es

$$\begin{aligned} &\text{def}\ (n, f(aprox\ n\ xs)) \wedge \neg\text{def}\ (n + 1, f\ (aprox\ n\ xs)) \\ &\quad \Rightarrow (policia\ f\ xs)\ !!\ n = \bot \end{aligned}$$

Dicho con palabras: si f hace trampas en algún movimiento n impidiendo devolver $(n + 1)$ respuestas bien definidas para n entradas bien definidas, entonces $policia\ f\ xs$ devuelve $\bot$ en el movimiento número n.

9.5.9 Demostrar que

$$\neg\text{def}\ (n + 1, zs) \Rightarrow sincroniza\ xs\ zs \sqsubseteq aprox\ n\ xs$$

para todas las listas zs y todas las listas parciales (y también infinitas) xs.

9.5.10 Suponiendo que $\neg\text{def}\ (n + 1, f\ (aprox\ n\ xs))$, demostrar que

$$\begin{aligned} &\neg\text{def}\ (n + 1, zs) \\ &\quad \Rightarrow f\ (sincroniza\ xs\ zs) \sqsubseteq f\ (aprox\ n\ xs) \wedge \neg\text{def}\ (n + 1, f\ (sincroniza\ xs\ zs)) \end{aligned}$$

9.5.11 Si $ys = f\ (sincroniza\ xs\ ys)$, entonces ys es el límite de la secuencia de aproximaciones ys_k, donde $ys_0 = \bot$ e $ys_{k+1} = f\ (sincroniza\ xs\ ys_k)$. Utilizando el ejercicio precedente, demostrar que

$$\neg\text{def}\ (n + 1, f\ (aprox\ n\ xs)) \Rightarrow policia\ xs \sqsubseteq f\ (aprox\ n\ xs)$$

lo cual completa la demostración de la conjetura del Ejercicio 9.5.8.

9.6 Interacción basada en el flujo de datos

Hasta ahora, todas nuestras sesiones con el computador han seguido un patrón de interacción uniforme y simple: el usuario escribe una expresión que ha de ser evaluada en el teclado y entonces el valor de la expresión se muestra en la pantalla. Este estilo de interacción es adecuado para una amplia gama de propósitos, pero a veces se requieren otras formas de interacción. Como ejemplo trivial, podríamos desear que todo lo que se pulsa en el teclado sea escrito en la pantalla, pero convirtiendo las letras minúsculas en mayúsculas. Podríamos desear también jugar interactivamente con el computador. Ejemplos más complejos incluyen un programa que simule interactivamente un computador funcional o incluso un sistema operativo complejo.

En esta sección, nuestro objetivo es dar una descripción muy simplificada de una técnica que permite escribir programas interactivos. El método, llamado interación

basada en el flujo de datos[2], se ha adoptado en algunos lenguajes funcionales; también se utilizó en algunas versiones anteriores de Haskell. Sin embargo, Haskell proporciona actualmente un enfoque alternativo mucho más general para manejar interacciones, que será tratado en el capítulo siguiente. En el método basado en el flujo de datos, escribimos un programa interactivo como una función *f* cuyo tipo es

$$f \;::\; String \rightarrow String$$

Cuando *f* se ejecuta interactivamente, la entrada de *f* será la secuencia de caracteres pulsada en el teclado. La salida de *f* se presentará del modo habitual. Como ejemplo simple, la función

$$map\ aMayuscula \;::\; String \rightarrow String$$

es un programa interactivo que convierte caracteres en minúscula a mayúscula. En la Sección 2.2 dimos la definición de *aMayuscula*.

Para ejecutar un programa interactivo, Haskell proporciona la función

$$interact \;::\; (String \rightarrow String) \rightarrow IO\ ()$$

Explicaremos el tipo *IO* () en el capítulo siguiente. Ahora, si nos hallamos dentro de una sesión, tecleamos *interact* (*map aMayuscula*) y entonces el evaluador entra en modo interactivo, esperando que introduzcamos caracteres por el teclado. Si tecleamos la cadena "Hola a todos", entonces veremos la salida siguiente

```
? interact (map aMayuscula)
HOLA A TODOS
```

El programa es completamente interactivo: es decir, tan pronto como pulsamos 'H' en el teclado, aparece 'H' en la pantalla, y después cuando pulsamos 'o', aparece 'O' en la pantalla, y así sucesivamente. El programa se ejecutará hasta que pulsemos la tecla de interrupción o apaguemos el computador.

También podemos diseñar un programa interactivo que termina. Por ejemplo,

$$\begin{aligned} aMayusculas &:: String \rightarrow String \\ aMayusculas &= takeWhile\ (\neq '.') \cdot map\ aMayuscula \end{aligned}$$

se comporta del mismo modo que *map aMayuscula*, pero termina la ejecución cuando se teclea un punto. Nótese que no se hará eco del carácter punto en la pantalla.

Supongamos que deseamos un estilo diferente de interacción, ejemplificado mediante

```
? interact ecoMayus
Hola a todos
HOLA A TODOS

Adios, mundo cruel
ADIOS, MUNDO CRUEL
```

[2]N. del T. En el original, *stream-based interaction*. En lo sucesivo, traducimos *stream* por "flujo de datos".

Ahora, cada vez que el usuario escribe una línea, primero se hace eco de ella en la pantalla y después se convierte a mayúsculas. Aquí hemos de diseñar *ecoMayus* de modo que mezcle el eco y el resultado de la conversión a mayúsculas:

$$\begin{aligned} ecoMayus\ xs &= ys \mathbin{++} \text{"\textbackslash n"} \mathbin{++} \\ &\quad map\ aMayuscula\ ys \mathbin{++} \text{"\textbackslash n"} \mathbin{++} ecoMayus\ (tail\ zs) \\ &\quad \textbf{where}\ (ys, zs) = span\ (\neq \text{'\textbackslash n'})\ xs \end{aligned}$$

La entrada *xs* se descompone en la cadena *ys* de caracteres anteriores al primer fin de línea y el resto de la cadena *zs*. Hacemos eco de la cadena *ys* en la pantalla, seguida de un fin de línea, y después se muestra también la misma cadena convertida a mayúsculas. Se imprime otro fin de línea y el resto de la entrada se trata de modo similar. Nótese especialmente que la responsabilidad de entremezclar correctamente la entrada y la salida está por completo en manos del programador.

Esta forma de interacción se denomina *basada en el flujo de datos* porque hace un uso esencial del hecho de que las listas se evalúan perezosamente. La frase "flujo de datos" se utiliza a menudo como sinónimo de lista perezosa. Por ejemplo, el comportamiento del programa *aMayusculas* descrito anteriormente se modeliza mediante la siguiente secuencia de listas parciales:

$$\begin{aligned} aMayuscula\ \bot &= \bot \\ aMayuscula\ ('H' : \bot) &= 'H' : \bot \\ aMayuscula\ ('H' : 'o' : \bot) &= 'H' : 'O' : \bot \\ aMayuscula\ ('H' : 'o' : 'l' : \bot) &= 'H' : 'O' : 'L' : \bot \\ aMayuscula\ ('H' : 'o' : 'l' : 'a' : \bot) &= 'H' : 'O' : 'L' : 'A' : \bot \\ \ldots \end{aligned}$$

Podemos ver la secuencia como una historia de la sesión interactiva. Cuando no se ha tecleado nada todavía, no aparece nada en la pantalla. Cuando hemos pulsado 'H', aparece 'H' en la pantalla. Cuando escribimos la cadena "Ho", entonces aparece "HO" en la pantalla. Y así sucesivamente. Si tecleamos un punto, entonces la historia termina:

$$aMayuscula\ ('H' : 'o' : 'l' : 'a' : '.' : \bot) = ['H', 'O', 'L', 'A']$$

De igual modo que hemos utilizado $\bot$ anteriormente para denotar un cómputo que aún no ha terminado, aquí utilizamos $\bot$ para denotar una entrada que todavía no ha sido introducida y una salida que todavía no ha aparecido en la pantalla. Un caso especial de cómputo que todavía no ha sido completado es uno que nunca se completará, es decir, un bucle infinito. De forma similar, dos casos particulares de entrada y salida son la entrada que se produce cuando el teclado no se vuelve a tocar y la salida que hace que no aparezca nada más en la pantalla. También estos casos se denotarán mediante $\bot$.

9.6.1 Discusión

El modelo que acabamos de presentar es demasiado simple para la mayoría de los propósitos prácticos. En una aplicación seria, deseamos hacer cosas distintas de leer caracteres con el teclado y presentarlos en la pantalla. Por ejemplo, podemos desear abrir, leer, escribir o borrar ficheros y, en general, interaccionar con todos los mecanismos disponibles en el mundo exterior que rodea a un lenguaje funcional.

Las interacciones tienen lugar en el tiempo físico y el orden en que ocurren los eventos ha de ser controlado correctamente por el programador. En el enfoque basado en el flujo de datos, esta ordenación de los eventos se representa mediante el orden de los elementos en una lista; en otras palabras, está representado en los datos y no reflejado esencialmente en el modo en que está construido el programa. En el capítulo siguiente consideraremos otro enfoque para la interacción, concretamente, un método general para escribir programas que hayan de controlar una secuencia ordenada de eventos. En este enfoque, el orden se hace explícito explícito en el modo en que está construido el programa.

Ejercicios

9.6.1 Escribir un programa interactivo que entremezcle caracteres individuales y sus equivalentes en mayúsculas. Por ejemplo, al teclear "Hola a todos", se debería producir

HHoOlLaA aA tToOdDoOsS

9.6.2 Escribir un programa de utilidad *read* cuyo tipo sea

$$read \;::\; String \to (String \to String \to String) \to String \to String$$

La función *read msj f* denota un programa interactivo que primero presenta un mensaje *msj* en la pantalla, después lee la línea siguiente *xs* del teclado haciendo eco de ella a medida que es leída y finalmente evalúa la función *f xs* sobre el resto de la entrada.

9.6.3 Escribir un programa *write* de utilidad similar cuyo tipo sea

$$write \;::\; String \to (String \to String) \to String \to String$$

La función *write msj f* primero escribe el mensaje *msj* en la pantalla y después evalúa *f* sobre la entrada.

9.6.4 Escribir un programa interactivo que modelice una cola, tal como se ilustró en la Sección 8.1.

9.6.5 Cuando se introduce por teclado, el carácter de retroceso '←' indica que se debe eliminar el carácter previamente tecleado y, cuando se presenta en la pantalla, hace que el cursor se mueva hacia la izquierda una posición. Escribir una función de utilidad *readEditar*, que sea igual que *read* excepto por el hecho de que se pueda utilizar '←' para editar la entrada. Por ejemplo, si tecleamos "biel←n" seguido por un fin de línea, debería introducirse en el programa la cadena "bien". (Para el eco, nótese que un carácter ya impreso en la pantalla puede ser eliminado imprimiendo "←␣←"; esta secuencia retrocede sobre el carácter, escribe un espacio en su lugar y después mueve el cursor de nuevo sobre el espacio).

9.7 Notas del capítulo

La teoría de dominios se desarrolló a partir del trabajo de Dana Scott a finales de los años sesenta, véanse Scott (1976) y Scott (1982). Una recopilación accesible de las

matemáticas básicas necesarias se da en Davey y Priestley (1990); para un tratamiento moderno de los dominios y de su importancia en la semántica de los lenguajes de programación, veáse también Gunter (1992).

La tesis de Sijtsma (1988) estudia varios aspectos de los programas que tratan con listas infinitas, y da numerosas técnicas para razonar sobre estas listas. Se dedica un capítulo a la demostración de equidad en el juego de piedra-tijera-papel.

La tesis de Gordon (1993) contiene una historia útil de los distintos enfoques para el tratamiento de la interacción en un entorno funcional. Uno de los desarrollos recientes más interesantes en la semántica de la programación ha sido el uso de juegos y de estrategias para proporcionar un modelo de los procesos interactivos. Para una recopilación legible de las ideas básicas véanse, en Pitts y Dyber (1997), los capítulos de S. Abramsky y M. Hyland.

Capítulo 10

Mónadas

Este capítulo introduce un nuevo estilo de programación funcional, conocido como programación *monádica*. Los programas monádicos proporcionan una manera simple y atractiva de abordar la interacción en un entorno funcional. Pero, como veremos, el estilo de programación monádico es capaz de mucho más: proporciona un mecanismo de estructuración para tratar con una amplia variedad de problemas, que incluyen el tratamiento de excepciones, análisis sintáctico y computaciones basadas en estados. En cierto sentido, el estilo monádico nos permite escribir programas funcionales que semejan programas imperativos, es decir programas construidos con lenguajes imperativos como Pascal y C.

Comenzamos considerando una mónada particular, la mónada *IO* α de entrada-salida, para expresar programas que implican interacción con el entorno, tema que se introdujo al final del capítulo precedente. Usaremos este ejemplo para motivar la definición de una mónada, así como para dar varios ejemplos de programas interactivos. Después, exploraremos otros modos beneficiosos de utilizar las mónadas e introduciremos las leyes fundamentales para razonar sobre los programas monádicos.

10.1 Interacción monádica

El tipo de las órdenes de entrada-salida en Haskell se denota como *IO* (). Una expresión del tipo *IO* () denota una *acción*; cuando se evalúa, la acción se *realiza*. El tipo *IO* () es un tipo abstracto en el sentido del Capítulo 8; por tanto, no se nos dice cómo se representan los valores del tipo; lo importante son las operaciones que proporciona el tipo. De hecho, estaremos interesados en las operaciones de un tipo más general *IO* α, pero empezaremos en primer lugar por la versión simple.

Una operación fundamental es una función que imprime un carácter:

$$putChar \quad :: \quad Char \rightarrow IO\ ()$$

Por ejemplo, al evaluar *putChar* '!' se imprime un símbolo de admiración:

```
? putChar '!'
!
```

Otro ejemplo es *done* :: *IO* (); cuando se realiza, *done* no hace nada:

? *done*
?

A continuación, necesitamos un modo de combinar acciones. Un operador para ello es

$$(\gg) \quad :: \quad IO\,() \to IO\,() \to IO\,()$$

Si p y q son acciones, entonces $p \gg q$ es una acción que, cuando se realice, primero efectuará p y después q. Utilizando $(\gg)$, podemos implementar la función *write*, que tiene el mismo efecto que la función *putStr* predefinida de Haskell:

```
write          :: String → IO ()
write [ ]      =  done
write (c : cs) =  putChar c ≫ write cs
```

También puede expresarse *write* en términos de *foldr*:

```
write  =  foldr (≫) done · map putChar
```

La función *writeln* es similar a *write*, excepto que imprime un carácter de fin de línea después de imprimir la cadena:

```
writeln     :: String → IO ()
writeln cs  =  write cs ≫ putChar '\'
```

El tipo *IO* () es adecuado para operaciones de escritura, pero necesita ser generalizado también para lectura. El tipo *IO* α es el tipo de las acciones que devuelven un valor del tipo α. Así, el caso especial *IO* () se consigue cuando α se concreta al tipo nulo (), que contiene exactamente un valor, también escrito (). En el Capítulo 2 discutimos brevemente este tipo.

Una operación primitiva sobre el tipo más general es aquella que lee un carácter:

```
getChar  :: IO Char
```

Cuando el usuario introduce algunos caracteres mediante el teclado, *getChar* devuelve el primer carácter tecleado. En Haskell se asume, además, que *getChar* hace eco del carácter en la pantalla. Veremos enseguida cómo utilizar *getChar*.

La generalización de *done* es la acción que no hace nada y devuelve un valor dado:

```
return  :: α → IO α
```

Por ejemplo, si realizamos la acción *return* 42, devolvemos el valor 42 sin consumir ninguna entrada. En particular,

```
done  :: IO ()
done  =  return ()
```

La generalización de $(\gg)$ es un operador cuyo tipo es

$$(\gg) \quad :: \quad IO\,\alpha \to IO\,\beta \to IO\,\beta$$

Si p y q son acciones, entonces $p \gg q$ es una acción que, cuando se realice, primero efectúa p, ignora el valor devuelto y después realiza q. Por ejemplo,

? *getChar* ≫ *done*
x

Una vez introducida la orden (terminada por un carácter de fin de línea), el usuario tecleó la letra 'x'. La acción *getChar* realizó eco de ésta en la pantalla y después la acción *done* terminó la interacción.

El tipo de *getChar* ≫ *done* es *IO* () y, como hemos dicho, se pueden evaluar en una sesión expresiones de este tipo particular, haciendo que se realicen tales acciones. Por otra parte, durante una sesión no se pueden evaluar expresiones del tipo *IO* α para $\alpha \neq ()$; por ejemplo,

? *getChar*
ERROR: Cannot find show function for *IO Char*[1]

Haskell no proporciona mecanismo alguno para imprimir valores del tipo *IO Char*. Veremos enseguida cómo el programador puede procesar el valor devuelto por una acción del tipo *IO* α.

Es obvio que la acción $p \gg q$ únicamente es útil cuando el valor devuelto por p no es interesante, ya que no hay modo alguno de hacer que q dependa de él. Lo que se necesita es un operador más general

$$(\rhd) \quad :: \quad IO\,\alpha \to (\alpha \to IO\,\beta) \to IO\,\beta$$

La combinación $p \rhd q$ es una acción que, cuando se realice, primero efectuará p, devolviendo un valor x del tipo α, y después realizará $q\ x$, devolviendo un valor final y del tipo β. Dejaremos como ejercicio el demostrar que podemos definir $(\gg)$ en términos de $(\rhd)$. La sintaxis Haskell para $\rhd$ es >>=.

He aquí un ejemplo que utiliza $(\rhd)$:

? *getChar* ▷ *putChar*
xx

Después de introducir esta orden, cuyo tipo es *IO* (), el usuario pulsó la letra 'x'; la acción *getChar* hizo eco de este carácter en la pantalla; después se aplicó la función *putChar* al carácter leído, haciendo que fuera impreso en la pantalla una vez más.

Utilizando $(\rhd)$, podemos definir una función *readn* que lea una cantidad fija de caracteres del teclado:

```
readn          :: Int → IO String
readn 0        =  return [ ]
readn (n + 1)  =  getChar ▷ q
                  where q c  =  readn n ▷ r
                                where r cs = return (c : cs)
```

Este programa tiene una lectura directa: para obtener una cadena de longitud $(n + 1)$, primero obtenemos un carácter c, después obtenemos una cadena cs de longitud n y finalmente devolvemos la cadena $(c : cs)$. El uso de cláusulas **where** anidadas es farragoso, por lo que enseguida introduciremos una notación mejor.

Quizá más útil que la función *readn* es la función *readln* que lee una línea de texto del teclado, es decir, la secuencia de caracteres hasta (pero sin incluir) el primer carácter de fin de línea:

[1] N. del T.: Error, no es posible encontrar la definición de la función *show* para el tipo *IO Char*

```
readln  ::  IO String
readln  =   getChar ▷ q
            where q c  =  if c == '\n'
                          then return [ ]
                          else readln ▷ r
                               where r cs = return (c : cs)
```

Resumiendo, la mónada de entrada-salida *IO* α es un tipo abstracto para el cual tenemos como mínimo, las siguientes operaciones:

```
return   ::  α → IO α
(▷)      ::  IO α → (α → IO β) → IO β

putChar  ::  Char → IO ()
getChar  ::  IO Char
```

Las dos primeras operaciones son combinadores que caracterizan la clase de tipos denominada *mónadas*. Así, por definición, *M* es una mónada cuando posee las dos operaciones siguientes:

```
return   ::  α → M α
(▷)      ::  M α → (α → M β) → M β
```

Se requiere que estas dos operaciones satisfagan ciertas leyes, a las cuales llegaremos a su debido tiempo. Las restantes operaciones primitivas del tipo *IO* α, *putChar* y *getChar*, son específicas para entrada-salida. Haskell proporciona una amplia variedad de operaciones primitivas para entrada-salida, algunas de las cuales mencionaremos más adelante.

10.1.1 Mónadas y la notación do

Podemos definir las mónadas como una clase de tipos en Haskell:

```
class Monad m where
  return  ::  α → m α
  (▷)     ::  m α → (α → m β) → m β
```

La clase *Monad* es nuestro primer ejemplo de una clase de tipos en el que la variable definida toma valor sobre constructores de tipos[2] en lugar de sobre tipos. Para remarcar este hecho, utilizaremos letras normales en lugar de letras griegas (de hecho, Haskell utiliza letras normales para ambas situaciones). El constructor de tipos *IO* es un ejemplo de mónada, para el cual se suministra como primitiva la correspondiente declaración de concreción.

Para las concreciones de la clase *Monad*, Haskell proporciona una notación alternativa —denominada notación **do**— para escribir combinaciones de ▷ con cláusulas **where** anidadas. Por ejemplo, el programa anterior para *readn* también puede ser escrito como sigue:

[2]N. del T.: En el original *type constructors*. Nótese que empleamos *constructoras* como traducción de *data constructors*. La diferencia entre unos y otras esencial. Los primeros, tales como *Arbin* o *Lista*, construyen tipos y las segundas, tales como *Hoja*, *Unir* o *[]*, consruyen valores de un tipo.


```
readn          ::  Int → IO String
readn 0        =   return [ ]
readn (n + 1)  =   do c ← getChar
                      cs ← readn n
                      return (c : cs)
```

El programa se puede leer de esta manera: realizar primero la acción *getChar* y vincular el resultado a *c*; realizar después la acción *readn n* y vincular el resultado a *cs*; finalmente, devolver (*c* : *cs*).

Análogamente, podemos escribir *readln* utilizando la notación **do** del modo siguiente:

```
readln  ::  IO String
readln  =   do c ← getChar
               if c == '\n'
                 then return [ ]
                 else do cs ← readln
                         return (c : cs)
```

Nótese la disposición espacial de la expresión condicional, con un sangrado de las cláusulas **then** y **else**. Esto es necesario según las leyes de formato de Haskell. Alternativamente, al igual que con las cláusulas **where**, podemos escribir una cláusula **do** con llaves y puntos y coma para separar explícitamente las cláusulas. Por ejemplo,

```
readn (n + 1)  =  do {c ← getChar; cs ← readn n; return (c : cs)}
```

Para una mónada general *M*, una expresión **do** tiene la forma **do** {*C*; *r*}, donde *C* es una lista de una o más acciones separadas por punto y coma, y *r* es una expresión de tipo *M* β, que es también el tipo de toda la expresión **do**. Cada orden tiene la forma '$x \leftarrow p$', donde *x* es una variable o una tupla de variables; si *p* es una expresión de tipo *M* α, entonces el tipo de *x* es α. En el caso particular en que $\alpha = ()$, la orden $() \leftarrow p$ se puede abreviar a *p*.

La traducción de una expresión **do** a operaciones con ▷ y cláusulas **where** se rige por dos reglas:

```
do {r}          =  r

do {x ← p; C; r}  =  p ▷ q   where q x = do {C; r}
```

Es fácil comprobar que las anteriores definiciones de *readn* y *readln* son equivalentes a las nuevas aplicando estas reglas, una tarea que dejamos como ejercicio.

10.1.2 Ejemplos

Veamos ahora algunos ejemplos simples de uso del tipo *IO* α. En primer lugar, he aquí un programa interactivo para leer un carácter e imprimir su valor numérico:

```
revelar  ::  IO ()
revelar  =   do c ← getChar
                putChar '\n'
                write (show (ord c))
```

Cuando se ejecuta *revelar*, el usuario teclea un carácter y el computador imprime su valor numérico.

A continuación, presentamos un programa interactivo que invita a introducir una cadena y después indica si la cadena es o no un palíndromo:

```
palindromo  :: IO ()
palindromo  = do write "Introduzca una cadena:␣"
                 cs ← getLine
                 if palin cs
                   then writeln "Si"
                   else writeln "No"

palin       :: String → Bool
palin xs    = (ys == reverse ys)
              where ys = map aMayuscula (filter esLetra xs)
```

Se deja como ejercicio la definición de *esLetra*. Un ejemplo de sesión podría ser el siguiente:

```
? palindromo
Introduzca una cadena: Dabale el abad
Si

? palindromo
Introduzca una cadena: Un hombre, un plan, un canal – ¡Suez!
No
```

A modo de ejemplo más complicado, he aquí un programa que juega al juego del "ahorcado". En este juego, un jugador piensa en una palabra y el otro jugador intenta averiguar la palabra realizando una secuencia de conjeturas. Tras cada intento, el computador (que actúa como mediador entre los jugadores) muestra la palabra secreta con todas aquellas letras no adivinadas hasta el momento reemplazadas por un '-'. El juego termina cuando todas las letras se han adivinado correctamente. Un ejemplo de sesión podría ser

```
? ahorcado
Piense una palabra
-----
Ahora trate de adivinarla
conjetura: hola
-o-al
conjetura: bozal
-o-al
conjetura: rosal
-oral
conjetura: moral
¡Bravo!
```

El programa tiene la siguiente forma


```
ahorcado    :: IO ()
ahorcado    = do writeln "Piense una palabra"
                 palabra ← leerLinSecreta
                 writeln "Ahora trate de adivinarla"
                 adivinar palabra
```

La acción *leerLinSecreta* lee una línea del teclado, pero no hace eco de los caracteres. En su lugar, presenta un '-' por cada carácter leído. Para implementarla, necesitamos otra función primitiva de Haskell, *getCh*, similar a *getChar* pero que no hace eco del carácter. De hecho

```
getChar     = do {c ← getCh; putChar c; return c}
```

Ahora implementamos la acción *leerLinSecreta* como sigue:

```
leerLinSecreta   :: IO String
leerLinSecreta   = do c ← getCh
                      if c == '}'
                        then do putChar c
                                return [ ]
                        else do putChar '–'
                                cs ← leerLinSecreta
                                return (c : cs)
```

La acción *adivinar palabra* acepta entradas desde el teclado hasta que la palabra introducida como adivinanza sea idéntica a *palabra*:

```
adivinar            :: String → IO ()
adivinar palabra    = do write "conjetura:␣"
                         cs ← readln
                         if cs == palabra
                           then writeln "¡Bravo!"
                           else do writeln (comparar palabra cs)
                                   adivinar palabra

comparar            :: String → String → String
comparar palabra cs = map comprobar palabra
                      where comprobar c = if member cs c then c else '–'
```

Finalmente, he aquí un ejemplo sugestivo que utiliza dos primitivas de Haskell para el tratamiento de ficheros:

```
readFile    :: FilePath → IO String
writeFile   :: FilePath → String → IO ()
```

El tipo *FilePath* es un renombramiento de *String*; los ficheros se nombran mediante cadenas utilizando un convenio dependiente de la implementación. El ejemplo consiste en leer un fichero, eliminar todos los caracteres que no sean ASCII y escribir el resultado en un segundo fichero:

```
filtrarFichero   :: IO ()
```

```
filtrarFichero  =  do write "Nombre del fichero de entrada:␣"
                      entrada ← getLine
                      write "Nombre del fichero de salida:␣"
                      salida ← getLine
                      xs ← readFile entrada
                      writeFile salida (filter isAscii xs)
                      writeln "Filtrado completado"
```

La función *isAscii* está definida en una librería de Haskell.

Como muestran los ejemplos precedentes, la escritura de programas monádicos utilizando la notación **do** es muy parecida a la escritura de programas imperativos tradicionales en lenguajes como Pascal o C. Pero existen diferencias; por ejemplo, la función *readFile* lee perezosamente un fichero en función de la demanda de su contenido. En consecuencia, las dos acciones

```
xs ← readFile infile
writeFile outfile (filter isAscii xs)
```

no tienen por efecto leer primero todo el fichero en memoria, y después filtrarlo y escribirlo de nuevo en disco.

Otra diferencia con la programación imperativa es que el estilo monádico existe en Haskell como una opción más: en aquellas partes de un programa que sean independientes de la interacción, se pueden aplicar todas las técnicas funcionales que hemos visto en los capítulos precedentes.

Ejercicios

10.1.1 Definir $(\gg)$ en términos de $(\rhd)$.

10.1.2 ¿Cuál es el efecto de la acción *foldl* $(\gg)$ *done* · *map putChar*?

10.1.3 Cuando se realiza, *getChar* lee un carácter y devuelve dicho carácter. Describir el efecto de ejecutar *getChar* $\rhd$ *return*. ¿Qué relación entre $\rhd$ y *return* sugiere este hecho?

10.1.4 Describir el efecto de ejecutar *return* '!' $\rhd$ *putChar*. ¿Qué relación entre $\rhd$ y *return* sugiere este hecho?

10.1.5 Utilizando $\rhd$, definir un operador $\diamond$ cuyo tipo sea

$$(\diamond) \quad :: \quad (\alpha \to IO\ \beta) \to (\beta \to IO\ \gamma) \to (\alpha \to IO\ \gamma)$$

Definir $\rhd$ utilizando $\diamond$.

10.1.6 Expandiendo la notación **do**, demostrar que

$$\mathbf{do}\ \{C; \mathbf{do}\ \{D; r\}\} \quad = \quad \mathbf{do}\ \{C; D; r\}$$

10.1.7 Reescribir el programa *revelar* de modo que invite repetidamente a teclear un carácter, imprima su valor numérico, y proceda de este modo hasta que el usuario pulse un carácter concreto, por ejemplo 'q', para terminar la interacción.

10.1.8 Quizá el juego del ahorcado es demasiado fácil. Modificar la respuesta a una adivinanza de modo que las letras adivinadas se muestren sólo si están presentes en la adivinanza en la posición correcta.

10.2 Variaciones sobre un evaluador

En esta sección vamos a examinar diferentes versiones de un evaluador de expresiones aritméticas simples. El objetivo es mostrar cómo un estilo monádico puede capturar las similitudes esenciales entre las distintas versiones, y —lo que es más importante— permitir crear cada versión sin cambiar la estructura básica del programa. Comenzaremos por un evaluador simple y después le añadiremos ciertas características. Específicamente, le añadiremos:

- Tratamiento de errores, lo que implica modificar cada llamada recursiva para comprobar y tratar adecuadamente los errores.
- Un contador de operaciones, lo que implica modificar todas las llamadas recursivas para manejar adecuadamente tales contadores.
- Una traza de ejecución, lo cual requiere modificar cada llamada recursiva para pasar adecuadamente tales trazas.

En cada caso, las tareas involucradas son fáciles pero tediosas. Peor aún, amenazan con invadir con una gran cantidad de detalles la simplicidad esencial del evaluador. Volveremos a considerar estas variaciones una vez introducidas, mostrando cómo utilizar un enfoque monádico para crear cada versión con menos esfuerzo.

10.2.1 El evaluador básico

El evaluador actúa sobre términos, que para los fines que nos proponemos serán muy simples:

```
data Term  =  Con Int | Div Term Term
```

Un término es o bien una constante *Con x*, siendo *x* un entero, o bien una división, *Div t u*, siendo *t* y *u* términos.

La función básica de evaluación se define mediante

```
eval          ::  Term → Int
eval (Con x)  =   x
eval (Div t u)  =  (eval t) div (eval u)
```

La función *eval* convierte un término en un entero. Si el término es una constante, se devuelve dicha constante. Si el término es una división, se evalúan sus subtérminos y después se calcula el cociente.

Se utilizarán los siguientes términos como ejemplo:

```
prueba, erroneo  ::  Term
prueba           =   Div (Div (Con 1972) (Con 2)) (Con 23)
erroneo          =   Div (Con 2) (Div (Con 1) (Con 0))
```

He aquí una ejecución simple del evaluador:

```
? eval prueba
42
? eval erroneo
[¡BOM!]
```

El segundo ejemplo presenta el problema de que la división por cero devuelve un resultado indefinido. Las operaciones indefinidas se pueden implementar de cualquier modo, y este computador particular implementa la división por cero con una explosión.

10.2.2 Excepciones

Una vez comprado un nuevo computador, deseamos modificar nuestro evaluador para que detecte la división por cero y devuelva un mensaje de error apropiado. No es adecuado utilizar la función primitiva *error* con este fin, ya que el evaluador puede ser usado interactivamente. La intención no es abortar el evaluador, sino simplemente tratar el error y mostrar un mensaje apropiado al usuario. Podemos implementar dicho *tratamiento de excepciones* introduciendo un tipo que represente los cálculos que puedan dar lugar a una excepción:

```
data Exc α        = Activar Excepcion | Devolver α
type Excepcion    = String
```

Un cálculo *Exc α* provocará una excepción o devolverá un valor de tipo *α*. Las excepciones pueden llevar asociado un mensaje de error, representado mediante una cadena de caracteres.

Es inmediato, pero tedioso, adaptar el evaluador a esta representación. Para enfatizar el orden en que ha de llevarse a cabo el cálculo, escribimos el nuevo evaluador de un modo particular:

```
eval          :: Term → Exc Int
eval (Con x)  = Devolver x

eval (Div t u)
  = h (eval t)
    where
    h (Activar e)    = Activar e
    h (Devolver x)   = h' (eval u)
                       where
                       h' (Activar e') = Activar e'
                       h' (Devolver y)
                         = if y == 0
                             then Activar "division por cero"
                             else Devolver (x div y)
```

En cada llamada al evaluador, se comprueba el aspecto del resultado mediante las funciones auxiliares de tratamiento *h* y *h'*. Si se produjo una excepción, se vuelve a activar, y si se devolvió un valor, se procesa dicho valor.

Podemos especificar cómo mostrar una excepción, haciendo que *Exc α* sea una concreción de *Show* (por simplicidad, únicamente definimos la función *show*):


```
instance Show α ⇒ Show (Exc α) where
  show (Activar e)    = "excepcion: " ++ e
  show (Devolver x)   = "valor: " ++ show x
```

He aquí una sesión con el nuevo evaluador:

```
? eval prueba
valor: 42

? eval erroneo
excepcion: division por cero
```

10.2.3 Estado

En esta versión, supongamos que deseamos contar el número de divisiones realizadas durante la evaluación. Un modo de hacerlo es introducir un componente adicional, que llamaremos *estado*. En este ejemplo el estado es un entero, inicializado a cero al comienzo del cálculo e incrementado en uno cada vez que se realiza una división. Podemos simular dicho estado introduciendo un tipo *transformador de estados* que represente los cálculos que se realizan sobre un estado:

```
newtype Est α    = McEst (Estado → (α, Estado))
type Estado      = Int
```

Un transformador de estados de tipo *Est α* es una función que toma un estado y devuelve un par formado por un valor de tipo *α* y un nuevo estado. Usaremos el estado para contar el número de divisiones y lo representaremos, por tanto, mediante un entero.

La constructora *McEst*[3] hace que *Est* sea un nuevo tipo, diferente del tipo funcional subyacente. La función auxiliar *aplicar* aplica un transformador de estados a un estado, dando como resultado un par (valor, estado):

```
aplicar            :: Est α → Estado → (α, Estado)
aplicar (McEst f) s = f s
```

He aquí el nuevo evaluador:

```
eval          :: Term → Est Int
eval (Con x)  = McEst f
                where f s = (x, s)

eval (Div t u)  = McEst f
                  where
                  f s  = (x div y, s'' + 1)
                         where
                         (x, s')   = aplicar (eval t) s
                         (y, s'')  = aplicar (eval u) s'
```

[3]N. del T.: Siguiendo el criterio habitual, el nombre es abreviatura de la frase *Mirar como estado*.

En cada llamada al evaluador, se aplica el transformador de estados resultante al viejo estado, y se devuelve un valor y un nuevo estado. Se ha de tener cuidado para pasar el estado apropiadamente —es demasiado fácil escribir accidentalmente *s* donde deberíamos haber escrito *s'*.

También podemos especificar cómo mostrar en la pantalla un transformador de estados:

```
instance Show α ⇒ Show (Est α) where
  show f  =  "valor: " ++ show x ++ "contador: " ++ show s
             where (x, s) = aplicar f 0
```

Mostramos en pantalla un transformador de estados *f* aplicándolo al estado inicial 0, y mostrando después el valor final y el contador. He aquí una sesión con el nuevo evaluador:

```
? eval prueba
valor: 42, contador: 2
```

10.2.4 Escritura

Finalmente, queremos mostrar una traza de los pasos de ejecución. Podemos simular dicha escritura introduciendo un tipo que represente los cálculos que generan escritura:

```
newtype Escri α = McEsc (Traza, α)
  type Traza  =  String
```

Un valor de tipo *Escri α* consiste en un par formado por la salida generada, junto con el valor calculado.

He aquí el evaluador modificado:

```
eval          ::  Term → Escri Int
eval (Con x)  =   McEsc (linea (Con x) x, x)
eval (Div t u)  =  McEsc (ox ++ oy ++ linea (Div t u) z, z)
                   where McEsc (ox, x)  =  eval t
                         McEsc (oy, y)  =  eval u
                         z              =  x div y

linea     ::  Term → Int → Traza
linea t x  =  "termino: " ++ show t ++ ", devuelve " ++ show x ++ "¬"
```

En cada llamada al evaluador se recolectan las salidas y se unen para formar la salida de la llamada en curso. La función *linea* genera una línea de la salida. Suponemos que existe la definición usual de *show* para términos y especificamos cómo mostrar en pantalla un cálculo con escritura:

```
instance Show α ⇒ Show (Escri α) where
  show (McEsc (ox, x))  =  ox ++ "valor: " ++ show x
```

He aquí una sesión con el nuevo evaluador:


```
? eval prueba
termino: Con 1972, devuelve 1972
termino: Con 2, devuelve 2
termino: Div (Con 1972) (Con 2), devuelve 986
termino: Con 23, devuelve 23
termino: Div (Div (Con 1972) (Con 2)) (Con 23), devuelve 42
valor: 42
```

De paso, merece la pena indicar que es fácil crear una traza de ejecución que se imprima en orden contrario: basta con reemplazar

$$ox \mathbin{++} oy \mathbin{++} linea\ (Div\ t\ u)\ z \quad \text{por} \quad linea\ (Div\ t\ u)\ z \mathbin{++} oy \mathbin{++} ox$$

10.2.5 Un evaluador monádico

El evaluador básico se puede reescribir fácilmente en términos de una mónada *m*:

```
eval          ::  Monad m ⇒ Term → m Int
eval (Con x)  =   return x
eval (Div t u)  =  do x ← eval t
                      y ← eval u
                      return (x div y)
```

El tipo de *eval* indica que ésta toma un término como argumento y realiza un "cómputo" *m* que devuelve un entero. Para evaluar *Con x*, simplemente devolvemos *x*. Para evaluar *Div t u*, primeramente evaluamos *t*, y vinculamos *x* al resultado, después evaluamos *u*, vinculando *y* al resultado, y finalmente devolvemos *x* **div** *y*.

El nuevo evaluador es ligeramente más complejo que el original pero es mucho más flexible. Podemos implementar cada una de las variaciones discutidas anteriormente simplemente haciendo que *m* sea una concreción apropiada de la clase *Monad*, y realizando una o dos modificaciones locales.

Primeramente, en el evaluador básico un cómputo coincide con el valor devuelto. La mónada *identidad* se declara introduciendo primero un nuevo tipo *Id α*, isomorfo a *α*:

```
newtype Id α = McId α

instance Monad Id where
  return x        =  McId x
  (McId x) ▷ q    =  q x
```

En la mónada identidad, *Id* es isomorfa a la función identidad sobre tipos, *return* es isomorfo a la función identidad, y (▷) es isomorfa a la aplicación funcional, salvo que recibe el argumento antes que la función.

También podemos especificar cómo mostrar en pantalla los elementos del tipo *Id α*:

```
instance Show α ⇒ Show (Id α) where
  show (McId x)  =  "valor: " ++ show x
```

El evaluador *evalId* es exactamente *eval*, pero especializado mediante $m = Id$:

```
evalId  ::  Term → Id Int
evalId  =   eval
```

Por ejemplo:

```
? evalId prueba
valor: 42
```

En la mónada de excepciones, un cómputo puede, o bien provocar una excepción, o bien devolver un valor. Aquí definimos

```
instance Monad Exc where
  return x          =  Devolver x
  (Activar e) ▷ q   =  Activar e
  (Devolver x) ▷ q  =  q x
```

La llamada *return x* simplemente devuelve *x*. La llamada $p \triangleright q$ examina el resultado del cómputo *p*; si se trata de una excepción, la vuelve a activar; en otro caso, se aplica la función *q* al valor devuelto. (Es un buen ejercicio comprobar los tipos.)

También definimos operaciones específicas de la mónada. En este caso, hay una operación que provoca una excepción:

```
activar    ::  Excepcion → Exc α
activar e  =   Activar e
```

Para añadir un tratamiento de errores, tomamos el evaluador monádico anterior y simplemente reemplazamos el término *return* (*x* **div** *y*) por una expresión condicional:

```
evalEx          ::  Term → Exc Int
evalEx (Con x)  =   return x
evalEx (Div t u)  =  do x ← eval t
                        y ← eval u
                        if y == 0
                          then activar "división por cero"
                          else return (xdiv y)
```

En lugar de escribir de nuevo todo el programa, sólo hemos tenido que hacer un pequeño cambio en el código.

A continuación, en la mónada de estados, un cómputo acepta un estado inicial y devuelve un par constituido por un valor y el estado final. La declaración de concreción para *Est* α es

```
instance Monad Est where
  return x  =  McEst f where f s = (x, s)
  p ▷ q     =  McEst f
               where
               f s  =  aplicar (q x) s'
                       where (x, s') = aplicar p s
```


La llamada *return x* devuelve un transformador de estados que devuelve el valor *x* y deja el estado sin cambiar. La llamada $p \triangleright q$ aplica el transformador de estados *p* al estado inicial *s*, devolviendo un valor *x* y un estado intermedio *s'*; a continuación aplica el transformador de estados *q x* al estado *s'*.

Definimos también algunas operaciones específicas de la mónada. En este caso, existe una operación que incrementa el estado.

```
tic  ::  Est ()
tic  =   McEst f where f s = ((), s + 1)
```

Para incorporar un contador, tomamos el evaluador monádico anterior y simplemente le añadimos una llamada a *tic* en el punto adecuado. En lugar de reescribir completamente el programa, sólo hemos de realizar un pequeño cambio local. He aquí el resultado:

```
evalEst          ::  Term → Est Int
evalEst (Con x)  =   return x
evalEst (Div t u)  =  do x ← evalEst u
                         y ← evalEst t
                         tic
                         return (xdiv y)
```

Finalmente, en la mónada con escritura *Escri* α, un cómputo consiste en el par formado por la escritura junto con el valor final. La declaración de concreción para *Escri* α es

```
instance Monad Escri where
  return x  =  McEsc (" ", x)
  p ▷ q     =  McEsc (ox ++ oy, y)
               where McEsc (ox, x)  =  p
                     McEsc (oy, y)  =  q x
```

La llamada *return x* devuelve la salida vacía junto con *x*. La llamada $p \triangleright q$ extrae la salida *ox* y el valor *x* a partir del cálculo *p*, después extrae la salida *oy* y el valor *y* a partir del cálculo *q x* y devuelve finalmente la salida formada concatenando *ox* con *oy*, junto con el valor *y*.

También definimos algunas operaciones específicas de la mónada. En este caso, existe una operación que genera la salida:

```
escri     ::  Traza → Escri ()
escri ox  =   McEsc (ox, ())
```

Para añadir trazas de ejecución al evaluador monádico, tomamos el evaluador monádico anterior y simplemente lo decoramos con llamadas que generen escritura:

```
evalEscri          ::  Term → Escri Int
evalEscri (Con x)  =   do escri (linea (Con x) x)
                          return x
evalEscri (Div t u)  =  do x ← evalEscri t
                           y ← evalEscri u
                           escri (linea (Div t u) (x div y))
                           return (xdiv y)
```

Para obtener la escritura en orden inverso, lo único que hace falta es cambiar la definición de ▷, reemplazando *ox* ++ *oy* por *oy* ++ *ox*. Este cambio es aún más simple que el que requería en las versiones anteriores del programa.

La lección que se desprende de estos ejemplos es que el estilo monádico es muy adecuado cuando el programa que estamos construyendo puede requerir modificaciones debidas a cambios en la especificación. Todos los cambios están localizados en la definición de la mónada y en uno o dos lugares del programa. No es necesaria una reestructuración importante del programa cada vez que añadimos este o aquel complemento. En la Sección 10.4 volveremos a los evaluadores y trataremos la cuestión de cómo combinar sus diferentes características.

Ejercicios

10.2.1 Modificar el evaluador con excepciones para especificar un término alternativo que ha de evaluarse en caso de producirse una excepción. Más concretamente, extender el tipo *Term* incluyendo un nuevo término *Ensayar Term Term*. Para evaluar *Ensayar t u*, primero evaluaremos *t* y si tiene éxito devolveremos su valor; pero si la evaluación provoca una excepción, entonces evaluaremos *u*. He aquí una transcripción de cómo debería comportarse el programa:

```
? eval (Ensayar (Div (Con 1) (Con 0)) (Con 42))
42
```

Definir una operación *ensayar* sobre el tipo *Exc* que facilite la modificación.

10.2.2 En el evaluador que cuenta el número de divisiones, el uso del estado es un tanto excesivo. En lugar de llevar cuenta de un estado en curso, cada cómputo podría simplemente devolver un valor junto con el número de operaciones requeridas para calcularlo:

```
data Contar α   =  (α, Contador)
type Contador   =  Int
```

Modificar el evaluador para usar este nuevo tipo de cálculo.

10.2.3 Modificar el evaluador con estado para poder acceder al estado durante un cómputo (por tanto, la simplificación del ejercicio precedente no funcionará). Más concretamente, extender el tipo *Term* incluyendo un nuevo término *Contador*. El valor de *Contador* es el número de operaciones realizadas hasta el momento, una cantidad que se obtiene accediendo al estado. He aquí una transcripción de cómo debería funcionar el programa:

```
? eval (Div (Div (Div (Con 6) (Con 3)) Contador) Contador)
1
```

Aquí, la respuesta es 1 porque la primera concreción de *Contador* se evalúa a 1 (ya que la división de 6 entre 3 se lleva a cabo primero) y la segunda concreción devuelve 2 (ya que las divisiones de 6 entre 3 y de 2 entre 1 se realizan en primer lugar). Definir una operación correspondiente *contar* en el tipo *Est* que facilite la modificación.

10.2.4 Modificar el evaluador con escritura de modo que solamente se seleccionen trazas de una parte del cómputo. Más concretamente, extender el tipo *Term* con dos términos adicionales, *Trazar Term* y *NoTrazar Term*. Sólo se debería activar la traza para aquellos subtérminos afectados por *Trazar*, y desactivarse para todos los subtérminos afectados *NoTrazar*. Para hacer posible este cambio, deberíamos representar un cómputo mediante una función cuyo argumento sea un valor booleano que indique si la traza está activa. Definir operaciones apropiadas sobre los cómputos para establecer y acceder al estado de este argumento booleano.

10.3 Leyes de las mónadas

Hemos visto varias concreciones de la clase *Monad*, que incluyen *IO*, *Id*, *Exc*, *Est* y *Escri*. En cada caso, las operaciones *return* y ▷ han de satisfacer ciertas leyes. En esta sección estableceremos las leyes de las mónadas de varias maneras diferentes aunque equivalentes. En primer lugar, recordemos los tipos de *return* y ▷:

```
return  ::  α → m α
(▷)     ::  m α → (α → m β) → m β
```

La primera ley establece que *return* es un elemento neutro por la derecha de ▷:

```
p ▷ return  =  p
```

La segunda ley establece que *return* es un "elemento neutro por la izquierda" de ▷ en el siguiente sentido:

```
(return e) ▷ q  =  q e
```

La tercera ley establece que ▷ es "asociativa" en el siguiente sentido:

```
(p ▷ q) ▷ r  =  p ▷ s where s x = ( q x ▷ r)
```

Hemos encontrado ya muchos ejemplos de operaciones cuyo tipo es $\alpha \to \alpha \to \alpha$ que tienen un elemento neutro por la izquierda y por la derecha y que son asociativas; los algebristas llaman *monoides* a estas estructuras. Tal como las hemos introducido, las mónadas difieren de los monoides en que, en lugar de una operación de tipo $\alpha \to \alpha \to \alpha$ para un cierto α, tenemos una operación de vinculación. Sin embargo, como veremos más adelante, también se puede definir ▷ en términos de una operación sobre un monoide. Los teóricos de categorías tienen mala fama por apropiarse de términos tomados de la filosofía, comenzando por el propio término de *categoría* tomado de Kant. La apropiación del término *mónada*, debido a Leibniz, fue motivada y estimulada por el juego de palabras con monoide.

Para todas las mónadas que hemos considerado, es fácil demostrar que se satisfacen las leyes monádicas. La ley del elemento neutro por la izquierda para la mónada de excepciones es inmediata a partir de la definición de ▷. He aquí la demostración de la ley del elemento neutro por la derecha para la mónada de estados. Recordemos que

```
instance Monad Est where
  return x  =  McEst f where f s = (x, s)
```

```
p ▷ q  =  McEst f
          where
          f s  =  aplicar (q x) s'
                  where (x, s') = aplicar p s

aplicar            ::  Est α → Estado → (α, Estado)
aplicar (McEst f) s  =  f s
```

En la demostración utilizamos una modalidad del principio de extensionalidad: $p = q$ si *aplicar p s* = *aplicar q s* para todos los estados *s*:

```
    aplicar (p ▷ return) s
=       {definición de ▷ (en la mónada de estados) y de aplicar}
    aplicar (return x) s'where (x, s') = aplicar p s
=       {definición de return y aplicar}
    (x, s') where (x, s') = aplicar ps
=       {simplificación}
    aplicar p s
```

He aquí la demostración de la ley asociativa para la mónada de escritura. Recordemos que

```
instance Monad Escri where
  return x  =  McEsc ("", x)
  p ▷ q     =  McEsc (ox ++ oy, y)
               where McEsc (ox, x)  =  p
                     McEsc (oy, y)  =  q x
```

En primer lugar, simplificamos el lado izquierdo:

```
    (p ▷ q) ▷ r
=       {definición de ▷ en la mónada de escritura}
    McEsc (ox ++ oy, y)
    where McEsc (ox, x)  =  p ▷ q
                            McEsc (oy, y) = r x
=       {definición de ▷}
    McEsc (ox ++ oy, y)
    where McEsc (ox, x)  =  McEsc (ou ++ ov, v)
                            where McEsc (ou, u)  =  p
                                  McEsc (ov, v)  =  q u
                                  McEsc (oy, y)  =  r x
=       {simplificación}
    McEsc ((ou ++ ov) ++ oy, y)
    where McEsc (ou, u)  =  p
          McEsc (ov, v)  =  q u
          McEsc (oy, y)  =  r v
```

Similarmente, simplificamos el lado derecho:

```
p ▷ s where s x = q x ▷ r
```



```
        {definición de ▷ aplicada dos veces}
    McEsc (ou ++ (ov ++ oy), y)
    where McEsc (ou, u) = p
          McEsc (ov, v) = q u
          McEsc (oy, y) = r v
```

Entonces el resultado se deduce a partir de la asociatividad de ++. Más aún, las leyes del elemento neutro por la izquierda y por la derecha para la mónada de escritura se deducen, respectivamente, de las leyes del elemento neutro por la izquierda y por la derecha para ++.

10.3.1 Leyes de la notación do

También podemos establecer las leyes en términos de la notación **do**. Éstas son:

```
do {B; x ← p; return x}     =  do {B; p}
do {B; x ← return e; C; r}  =  do {B; C[x := e]; r[ x := e]}
do {B; x ← do {C; p}; D; r} =  do {B; C; x ← p; D; r}
```

Aquí, B, C, D toman su valor sobre secuencias de acciones, y $C[x := e]$ significa la secuencia *C* cuyas apariciones libres de *x* se sustituyen por *e*. Dejaremos como ejercicio las demostraciones de que estas tres leyes son equivalentes a las tres establecidas anteriormente.

Hay una regla adicional, llamada la *ley del colapso*, que es una consecuencia de la definición de la notación **do**:

```
do {C; do {D; r}}  =  do {C; D; r}
```

Para mostrar el uso de estas versiones de las leyes monádicas, consideremos una variante del evaluador de la sección precedente, basado en la suma en lugar de en la división:

```
data Term          =  Con Int | Sumar Term Term

eval               ::  Monad m ⇒ Term → m Int
eval (Con x)       =  return x
eval (Sumar t u)   =  do {x ← eval t; y ← eval u; return (x + y)}
```

El objetivo es demostrar que la evaluación de

Sumar t (*Sumar u v*) y de *Sumar* (*Sumar t u*) *v*

produce el mismo resultado. En primer lugar, simplificamos el término izquierdo:

```
    eval (Sumar t (Sumar u v))
=       {definición de eval}
    do {x ← eval t; w ← eval (Sumar u v); return (x + w)}
=       {definición de eval}
    do x ← eval t
       w ← do {y ← eval u; z ← eval v; return (y + z)};
       return (x + w)
=       {ley asociativa}
```

$$
\begin{array}{ll}
 & \textbf{do}\ x \leftarrow eval\ t \\
 & \quad y \leftarrow eval\ u \\
 & \quad z \leftarrow eval\ v \\
 & \quad w \leftarrow return\ (y + z) \\
 & \quad return\ (x + w) \\
= & \quad \{\text{ley del elemento neutro por la izquierda}\} \\
 & \textbf{do}\ \{x \leftarrow eval\ t;\ y \leftarrow eval\ u;\ z \leftarrow eval\ v;\ return\ (x + (y + z))\}
\end{array}
$$

La simplificación del lado derecho es similar:

$$
\begin{array}{ll}
 & eval\ (Sumar\ (Sumar\ t\ u)\ v) \\
= & \quad \{\text{igual que antes}\} \\
 & \textbf{do}\ \{x \leftarrow eval\ t;\ y \leftarrow eval\ u;\ z \leftarrow eval\ v;\ return\ ((x + y) + z)\}
\end{array}
$$

El resultado se deduce a partir de la asociatividad de la suma.

Nótese especialmente que la demostración es válida para *cualquier* mónada, siempre que el programa sea como se ha dicho anteriormente. En la Sección 10.2, modificamos el programa añadiendo llamadas a *tic* en el caso de la mónada de estados. Sin embargo, todavía se cumple la asociatividad, como se puede demostrar utilizando la ley

$$\textbf{do}\ \{tic;\ x \leftarrow p;\ return\ x\} \quad = \quad \textbf{do}\ \{x \leftarrow p;\ tic;\ return\ x\}$$

Esta ley se cumple siempre que *tic* sea la única acción sobre el estado dentro de la mónada *m*. Análogamente, en el caso de la mónada de escritura, modificamos el programa añadiendo llamadas a *linea*. En este caso, los dos términos ya no son equivalentes porque las trazas serán diferentes. Por supuesto, los dos cómputos todavía devolverán el mismo valor final.

10.3.2 Leyes de composición de mónadas

Quizá el mejor modo de recordar las leyes monádicas sea expresarlas en términos de un operador de *composición* de mónadas. Definimos el operador $(\diamond)$ mediante

$$
\begin{array}{lcl}
(\diamond) & :: & (\alpha \to m\ \beta) \to (\beta \to m\ \gamma) \to (\alpha \to m\ \gamma) \\
(f \diamond g)\ x & = & f\ x \rhd g
\end{array}
$$

De modo equivalente, utilizando una sección,

$$f \diamond g \quad = \quad (\rhd g) \cdot f$$

El operador $(\diamond)$ es como la composición de funciones $(\cdot)$, excepto por el hecho de que las funciones componentes tienen cada una tipo $\alpha \to m\ \beta$ para tipos α y β apropiados, y el orden en que se componen las funciones está invertido. De hecho, se puede utilizar $(\diamond)$ para definir $\rhd$:

$$(\rhd g) \quad = \quad id \diamond g$$

Nótese también que $(\diamond)$ y $(\cdot)$ están relacionados por la siguiente ley útil

$$(f \diamond g) \cdot h \quad = \quad (f \cdot h) \diamond g$$

La demostración se deduce inmediatamente de la definición de $(\diamond)$. Nos referiremos a esta ley como la ley del *salto de la rana*.

Reformuladas, las dos leyes monádicas establecen ahora que *return* es un elemento neutro por la izquierda y por la derecha de $(\diamond)$:

$$p \diamond return = p \quad \text{y} \quad return \diamond q = q$$

Y, reformulada, la tercera ley establece que $(\diamond)$ es asociativa:

$$(f \diamond g) \diamond h \quad = \quad f \diamond (g \diamond h)$$

Se deja como ejercicio la demostración de que los dos conjuntos de leyes son equivalentes.

Así, después de todo, una mónada es un monoide. Aunque las leyes de las mónadas se expresan más fácilmente en términos de $(\diamond)$, los programas monádicos son más fáciles de escribir en términos de la notación **do**, que está basada en $\rhd$.

10.3.3 Las funciones *map* y *concat*

Todavía hay otro modo, y muy revelador, en que podemos definir las leyes fundamentales de las mónadas. En primer lugar, consideremos las siguientes definiciones:

$$
\begin{array}{lcl}
mapm & :: & Monad\ m \Rightarrow (\alpha \to \beta) \to (m\ \alpha \to m\ \beta) \\
mapm\ f\ p & = & \textbf{do}\ \{x \leftarrow p;\ return\ (f\ x)\}
\end{array}
$$

$$
\begin{array}{lcl}
concatm & :: & Monad\ m \Rightarrow m\ (m\ \alpha) \to m\ \alpha \\
concatm\ ps & = & \textbf{do}\ \{p \leftarrow ps;\ p\}
\end{array}
$$

La función *mapm* aplica una función al resultado devuelto por un cómputo. Para calcular *mapm f p*, primero calculamos *p*, después vinculamos *x* al resultado, y finalmente devolvemos *f x*. En la definición de *concatm ps*, la variable *ps* denota un cómputo que, cuando se realiza, *él mismo* devuelve un cómputo. Por tanto, *concatm* aplana un intrincado doble nivel de cómputos, en un solo nivel de cómputos elementales.

Podemos expresar *mapm* y *concatm* utilizando el operador de composición monádico $\diamond$. En primer lugar, tenemos

$$
\begin{array}{ll}
 & mapm\ f\ p \\
= & \quad \{\text{definición}\} \\
 & \textbf{do}\ \{x \leftarrow p;\ return\ (f\ x)\} \\
= & \quad \{\text{traduciendo la notación } \textbf{do} \text{ a } \rhd\} \\
 & p \rhd (return \cdot f) \\
= & \quad \{\text{definición de } \diamond \text{ utilizando la función identidad } id\} \\
 & (id \diamond (return \cdot f))\ p
\end{array}
$$

Por tanto $mapm\ f = id \diamond (return \cdot f)$. En segundo lugar,

$$
\begin{array}{ll}
 & concatm\ ps \\
= & \quad \{\text{definición}\} \\
 & \textbf{do}\ \{p \leftarrow ps;\ p\} \\
= & \quad \{\text{traduciendo la notación } \textbf{do} \text{ a } \rhd\} \\
 & ps \rhd id \\
= & \quad \{\text{definición de } \diamond\} \\
 & (id \diamond id)\ ps
\end{array}
$$

Por tanto $concatm = id \diamond id$. El tipo del primer id es $id :: m\ (m\ \alpha) \to m\ (m\ \alpha)$, en tanto que el tipo del segundo id es $id :: m\ \alpha \to m\ \alpha$.

Hay siete leyes que caracterizan *mapm* y *concatm*, todas ellas con un aire familiar:

$$
\begin{array}{lcl}
mapm\ id & = & id \\
mapm\ (f \cdot g) & = & mapm\ f \cdot mapm\ g \\
 & & \\
mapm\ f \cdot return & = & return \cdot f \\
mapm\ f \cdot concatm & = & concatm \cdot mapm\ (mapm\ f) \\
 & & \\
concatm \cdot return & = & id \\
concatm \cdot mapm\ return & = & id \\
concatm \cdot mapm\ concatm & = & concatm \cdot concatm
\end{array}
$$

Las dos primeras leyes establecen que *mapm* es un *funtor*, igual que *map* sobre listas es un funtor (véase la Sección 4.3). Las leyes tercera y cuarta son condiciones de *naturalidad* sobre *return* y *concatm*, igual que las leyes

$$
\begin{array}{lcl}
map\ f \cdot enlistar & = & enlistar \cdot f \\
map\ f \cdot concat & = & concat \cdot map\ (map\ f)
\end{array}
$$

son condiciones de naturalidad sobre las funciones de tratamiento de listas *enlistar* y *concat* (de nuevo, véase la Sección 4.3). Y las tres leyes finales son generalizaciones de leyes simples de *enlistar* y *concat*.

Estas siete leyes son equivalentes a las tres leyes monádicas. Por ejemplo, demostraremos la ley

$$mapm\ f \cdot concatm = concatm \cdot mapm\ (mapm\ f)$$

Para ello, necesitaremos el siguiente hecho:

$$(return \cdot h) \diamond g = g \cdot h$$

La demostración es

$$
\begin{array}{ll}
 & (return \cdot h) \diamond g \\
= & \quad \{\text{ley del salto de la rana} - (f \cdot h) \diamond g = (f \diamond g) \cdot h\} \\
 & (return \diamond g) \cdot h \\
= & \quad \{\text{ley del elemento neutro por la izquierda}\} \\
 & g \cdot h
\end{array}
$$

Para la demostración principal, simplificamos el lado izquierdo:

$$
\begin{array}{ll}
 & mapm\ f \cdot concatm \\
= & \quad \{\text{definiciones de } mapm \text{ y } concatm \text{ en términos de } (\diamond)\} \\
 & (id \diamond (return \cdot f)) \cdot (id \diamond id) \\
= & \quad \{\text{ley del salto de la rana}\} \\
 & (id \diamond id) \diamond (return \cdot f)
\end{array}
$$

De forma parecida, simplificamos el lado derecho:

$$
\begin{array}{ll}
 & concatm \cdot mapm\ (mapm\ f) \\
= & \quad \{\text{definiciones de } mapm \text{ y } concatm \text{ en términos de } (\diamond)\}
\end{array}
$$

$$
\begin{array}{ll}
 & (id \diamond id) \cdot (id \diamond (return \cdot (id \diamond (return \cdot f)))) \\
= & \quad \{\text{ley del salto de la rana}\} \\
 & (id \diamond (return \cdot (id \diamond (return \cdot f)))) \diamond id \\
= & \quad \{\text{asociatividad}\} \\
 & id \diamond (return \cdot (id \diamond (return \cdot f))) \diamond id \\
= & \quad \{\text{por lo anterior e } id \cdot h = h\} \\
 & id \diamond (id \diamond (return \cdot f))
\end{array}
$$

Estos dos resultados son el mismo ya que $\diamond$ es asociativo. Dejaremos el resto de las demostraciones al lector diligente.

10.3.4 Resumen

Hemos visto cuatro modos diferentes de expresar las leyes de una mónada. Cada uno tiene sus ventajas. Las leyes de $\rhd$, y sus equivalentes en términos de la notación **do**, son las más útiles cuando razonamos sobre programas escritos utilizando la notación **do**. Las leyes de ($\diamond$) son las más simples de recordar y muestran que las mónadas son un caso particular de monoides. Las leyes de *mapm* y *concatm*, que son el modo en que los teóricos de categorías introducen las mónadas, son las más reveladoras porque sugieren que se pueden encontrar mónadas en todos los lugares.

Ejercicios

10.3.1 Escribiendo *List* α en lugar de $[\alpha]$, hacer que *List* sea miembro de *Monad*.

10.3.2 Sea M un monoide con elemento neutro e y operación $\oplus$. Hacer que *Mon* sea miembro de la clase *Monad*, donde

$$\textbf{data}\ Mon\ \alpha = McMon\ (M, \alpha)$$

La mónada de escritura es un caso especial de esta mónada, así como la mónada *Contar* implícita en el Ejercicio 10.2.2.

10.3.3 Demostrar la ley $concatm \cdot mapm\ concatm = concatm \cdot concatm$.

10.4 Combinando mónadas

A veces deseamos combinar mónadas. Por ejemplo, podríamos tratar de modificar el evaluador de la Sección 10.2 para utilizar a la vez excepciones y un estado. Una posibilidad es construir otra mónada más, combinando los aspectos de las excepciones y del estado. Pero sería preferible, una vez hecho por separado el trabajo para definir las mónadas de excepciones y las mónadas de estado, que pudiéramos combinarlas de un modo simple.

Sería maravilloso que existiera una única regla que pudiéramos usar para combinar dos mónadas cualesquiera. Lamentablemente no existe tal regla. De hecho, como veremos, a veces hay dos modos distintos en que se pueden combinar dos mónadas. Sin embargo, normalmente es posible definir una regla uniforme para combinar una mónada con otra. La idea básica consiste en definir una operación τ que convierta

constructores de tipos en constructores de tipos, de modo que si m es una mónada, entonces $\tau\ m$ también sea una mónada. A una operación tal se la denomina *transformador de mónadas*.

Cada operación definida se utilizará en varias mónadas diferentes, de modo que podemos definir una clase de tipos para cada operación. Recordemos que tenemos ya una clase de tipos para las dos operaciones básicas sobre mónadas:

```
class Monad m where
  return  :: α → m α
  (▷)     :: m α → ( α → m β) → m β
```

Por definición, una *mónada de excepciones* es aquella que tiene una operación para activar una excepción:

```
class Monad m ⇒ MonadaEx m where
  activar  :: Excepcion → m α
```

Análogamente, una *mónada de estados* es aquella que tenga una operación para incrementar el estado:

```
class Monad m ⇒ MonadaEst m where
  tic :: m ()
```

Finalmente, una *mónada de escritura* es aquella que tiene una operación para mostrar un valor apropiado:

```
class Monad m ⇒ MonadaEscri m where
  showMonad  :: m String → String
```

A diferencia de la clase *Show*, que toma un tipo como argumento, esta versión toma un constructor de tipos.

Por definición, un *transformador de mónadas* es una operación τ que convierte mónadas en mónadas, con la propiedad de que se pueden elevar los cómputos de la mónada subyacente a la mónada transformada:

```
class Transform τ where
  elevar  :: Monad m ⇒ m α → τ m α
```

Aquí α varía sobre tipos, m varía sobre constructores de tipos, que actúan como funciones de tipos en tipos, y τ varía sobre funciones de constructores de tipos a constructores de tipos.

He aquí una versión del evaluador de la Sección 10.2 que a la vez activa una excepción ante una división por cero y cuenta el número de operaciones de división efectuadas. El tipo indica que se utiliza una mónada que tiene a la vez excepciones y estado:

```
eval          :: (ExcMonad m, MonadaEst m) ⇒ Term → m Int
eval (Con x)  = return x
eval (Div t u) = do x ← eval u
                    y ← eval t
                    tic
                    if y == 0
                      then activar "división por cero"
                      else return (xdiv y)
```


10.4.1 Combinando excepciones

Nuestro primer transformador de mónadas combina una mónada arbitraria m con la mónada de excepciones. Seguiremos el convenio de dar a los transformadores de mónadas nombres con todos los caracteres en mayúsculas.

```
newtype EXC m α  =  McEXC (m (Exc α))
data Exc α       =  ActivarExcepcion | Devolver α
type Excepcion   =  String
```

Aquí, *Exc* y *Excepcion* son exactamente iguales que en la Sección 10.2. ¿Por qué hemos elegido escribir m (*Exc* α) en lugar de, por ejemplo, *Exc* ($m\ \alpha$)? Simplemente, porque funciona. Parece imposible escribir una definición apropiada de $\triangleright$ para la otra posibilidad.

Necesitaremos una función auxiliar *recuperar* que recupere el tipo subyacente a partir del nuevo tipo:

```
recuperar          :: EXC m α → m (Exc α)
recuperar (McEXC g) = g
```

El siguiente paso es hacer que *EXC m* sea una mónada, utilizando la suposición de que m es una mónada:

```
instance Monad m ⇒ Monad (EXC m) where
  return x  =  McEXC (return (Devolver x))
  p ▷ q     =  McEXC (recuperar p ▷ r)
               where r (Activar e)  =  return (Activar e)
                     r (Devolver x) =  recuperar (q x)
```

Aquí, *return* y $\triangleright$ de la mónada *EXC m* (las que aparecen en la parte izquierda de las ecuaciones) se definen en términos de *return* y de $\triangleright$ de la mónada subyacente m (la que aparece en la parte derecha de las ecuaciones). Es un ejercicio útil comprobar los tipos:

```
    x :: α
⇒     {definición de Exc}
    Devolver x :: Exc α
⇒     {definición de return en la mónada m}
    return (Devolver x) :: m (Exc α)
⇒     {definición de EXC}
    McEXC (return (Devolver x)) :: EXC m α
```

Análogamente,

```
    p :: EXC m α y q :: α → EXC m β
⇒     {definición de recuperar y ▷ en la mónada m}
    recuperar p :: m (Exc α) y r :: Exc α → m (Exc β)
⇒     {definición de EXC}
    McEXC (recuperar p ▷ r) :: EXC m β
```

También es necesario demostrar que, si *return* y $\triangleright$ en la mónada subyacente m satisfacen las tres leyes monádicas, también lo hacen *return* y $\triangleright$ en la nueva mónada *EXC m*. He aquí la demostración de la ley del elemento neutro por la derecha:

```
        p ▷ return
=          {definición de return y ▷ en la mónada EXC m}
        McEXC (recuperar p ▷ r)
        where r (Activar e) = return (Activar e)
              r (Devolver x) = return (Devolver x)
=          {ya que, por extensionalidad, r = return}
        McEXC (recuperar p ▷ return)
=          {ley del elemento neutro por la derecha de la mónada m}
        McEXC (recuperar p)
=          {definición de recuperar}
        p
```

Las otras dos demostraciones son similares.

Ahora hemos hecho que *EXC m* sea una mónada y el siguiente paso es hacer que *EXC* sea un transformador de mónadas:

```
instance Transform EXC where
  elevar g  =  McEXC (do {x ← g; return (Devolver x)})
```

Esta definición es similar a la de *return* en *Exc*, con un vínculo adicional y con un *return* sobre *m* para hacer que los tipos sean correctos.

La mónada *EXC m* es una mónada de excepciones para cualquier mónada *m*:

```
instance Monad m ⇒ MonadaEx (EXC m) where
  activar e  =  McEXC (return (Activar e))
```

La mónada *EXC m* es una mónada de estados, siempre que *m* lo sea:

```
instance MonadaEst m ⇒ MonadaEst (EXC m) where
  tic  =  elevar tic
```

Aquí *tic* en la mónada *EXC m* (en la parte izquierda) se define en términos de *tic* de la mónada *m* (en la parte derecha).

La mónada *EXC m* es una mónada de escritura, siempre que *m* lo sea:

```
instance MonadaEscri m ⇒ MonadaEscri (EXC m) where
  showMonad p  =  showMonad (recuperar p ▷ q)
                  where
                  q (Activar e)   =  return ("Excepcion: " ++ e)
                  q (Devolver cs) =  return cs
```

En la parte izquierda, *showMonad* :: *EXC m String* → *String*. En la derecha, *showMonad* :: *m String* → *String*. Podemos utilizar *showMonad* para hacer que *EXC m α* sea miembro de la clase *Show*:

```
instance (MonadaEscri m , Show α) ⇒ Show (EXC m α) where
  show  =  showMonad · mapMonad show
```

La función *mapMonad* es la que se definió en la Sección 10.3, donde se la llamó *mapm*.

10.4.2 Combinando el estado

Nuestro segundo transformador de mónadas combina una mónada arbitraria *m* con la mónada de estado:

```
newtype TEST m α = McTEST (Estado → m (α, Estado))
  type Estado  =  Int
```

Nótese que aquí seguimos un esquema ligeramente diferente al de la definición previa. La mónada subyacente *m* se aplica al par formado por el valor y el estado final. Compárese esta idea con otras posibles variantes, tales como *m* (*Estado* → (*α*, *Estado*)), o bien *Estado* → (*m α*, *Estado*). De nuevo escogemos esta definición porque funciona. Parece imposible escribir una definición apropiada de ▷ para las otras posibilidades. Necesitamos una función auxiliar que aplique un cómputo al estado inicial:

```
aplicar              ::  TEST m α → Estado → m (α, Estado)
aplicar (McTEST f)   =   f
```

El siguiente paso es hacer que *TEST m* sea una mónada, utilizando la suposición de que *m* es una mónada:

```
instance Monad m ⇒ Monad (TEST m) where
  return x  =  McTEST f where f s = return (x, s)
  p ▷ q     =  McTEST f
               where
               f s  =  do (x, s') ← aplicar p s
                          aplicar (q x) s'
```

De nuevo es útil comprobar los tipos, una labor que dejamos como ejercicio.

También es necesario demostrar que, si en la mónada *m* subyacente *return* y ▷ satisfacen las tres leyes monádicas, también lo hacen *return* y ▷ en la nueva mónada *TEST m*. He aquí la demostración de la ley del elemento neutro por la derecha:

```
        aplicar (p ▷ return) s
=          {definición de ▷ en la mónada TEST m}
        do {(x, s') ← aplicar p s ; aplicar (return x) s'}
=          {definición de return y aplicar}
        do {(x, s') ← aplicar p s; return (x, s')}
=          {ley de elemento neutro por la derecha para la mónada m}
        aplicar p s
```

Ahora hemos conseguido que *TEST m* sea una mónada y el paso siguiente es hacer que *TEST* sea un transformador de mónadas:

```
instance Transform TEST where
  elevar g  =  McTEST f
               where
               f s  =  do {x ← g; return (x, s)}
```

Esta definición es similar a la de *return* en *Est*, con un vínculo extra y con un *return* sobre *m* para hacer que los tipos sean correctos.

La mónada *TEST m* es una mónada de estados para cualquier mónada *m*:

```
instance Monad m ⇒ MonadaEst (TEST m) where
  tic  =  McTEST f where f s = return ((), s + 1)
```

Esta definición es similar a la de *activar* en *Est*, con un *return* sobre *m* añadido para hacer que los tipos sean correctos.

La mónada *TEST m* es una mónada de excepciones siempre que *m* lo sea:

```
instance MonadaEx m ⇒ MonadaEx (TEST m) where
  activar e  =  elevar (activar e)
```

La mónada *TEST m* es una mónada de escritura siempre que *m* lo sea:

```
instance MonadaEscri m ⇒ MonadaEscri (TEST m) where
  showMonad p
    = showMonad q
      where
      q = do (x, s) ← aplicar p 0
             return ("valor: " ++ x ++ ", contador: " ++ show s)
```

Podemos utilizar *showMonad* para hacer que *TEST m α* sea miembro de la clase *Show*:

```
instance (MonadaEscri m, Show α) ⇒ Show (TEST m α) where
  show  =  showMonad · mapMonad show
```

10.4.3 Poniendo todo junto

Una vez sembradas cuidadosamente las semillas, ya podemos obtener los frutos. Los transformadores de mónadas *EXC* y *TEST* convierten cada uno de ellos una mónada en otra. Necesitamos alguna mónada para empezar, y la mónada identidad *Id* definida en la Sección 10.2 satisface esta necesidad adecuadamente. Recordemos que

```
newtype Id α = McId α
instance Monad Id where
  return x       =  McId x
  (McId x) ▷ q   =  q x
```

Hemos de hacer que *Id* sea una mónada de escritura:

```
instance MonadaEscri Id where
  showMonad (McId cs)  =  cs
```

Obsérvese que la mónada *EXC Id* es equivalente a la mónada *Exc*, y que la mónada *TEST Id* es equivalente a la mónada *Est*.

Hay dos modos diferentes de construir una mónada que soporte a la vez excepciones y estado, dependiendo del orden en que apliquemos nuestros transformadores de estado:

```
evalEstEx  ::  Term → TEST (EXC Id) Int
evalEstEx  =   eval

evalExEst  ::  Term → EXC (TEST Id) Int
evalExEst  =   eval
```


De cualquier modo que lo hagamos, el resultado parece ser el mismo. Considerando

```
prueba = Div (Div (Con 1972) (Con 2)) (Con 23)
```

tenemos

```
? evalEstEx prueba
 valor: 42, contador: 2
? evalExEst prueba
 valor: 42, contador: 2
```

Sin embargo, la similitud es sólo aparente: las respuestas parecen las mismas pero tienen tipos y estructuras diferentes. En el primer caso, la evaluación da como resultado el cómputo

```
McTEST f where f s  =  McEXC (McId (Devolver (42, s + 2)))
```

mientras que, en el segundo caso, la evaluación da como resultado

```
McEXC (McTEST f) where f s  =  (McId (Devolver 42), s + 2)
```

Esta diferencia se hace visible cuando se evalúa un término que provoca una excepción. Recordemos que

```
erroneo = Div (Con 2) (Div (Con 1) (Con 0))
```

Tenemos

```
? evalEstEx erroneo
 excepcion: división por cero

?evalExEst erroneo
 valor: excepcion: división por cero, contador: 1
```

En el primer caso, la evaluación produce el cómputo

```
McTEST f where f s  =  McEXC (McId (Activar "división por cero"))
```

indicando solamente que ha ocurrido un error, mientras que en el segundo caso, la evaluación produce el cómputo

```
McEXC (McTEST f) where f s  =  (McId (Activar "división por cero"), s + 1)
```

indicando que se ha contado un paso de contador antes de que se provocara la excepción.

La segunda de estas dos combinaciones es la que mejor corresponde al modo en que se implementa normalmente el estado, es decir, con una variable global. Con una variable global, siempre podemos recuperar el valor del estado a partir de la variable, incluso después de que se haya provocado una excepción. La primera combinación se parece al caso en el que provocar una excepción hace que el estado dé "marcha atrás" a su valor en el punto en que se invocó el tratamiento de la excepción, y puede resultar considerablemente más costoso de implementar.

La moraleja es clara: las funcionalidades complejas pueden interactuar de formas complejas. No existe una regla fija para combinar mónadas porque no existe una regla fija para combinar funcionalidades. Pero las mónadas ofrecen cierta ayuda para

estructurar esta complejidad, tal como hemos visto aquí, donde han permitido revelar dos modos alternativos de combinar el estado con las excepciones.

Ejercicios

10.4.1 Definimos un transformador *ESCRI* que añade escritura a una mónada dada mediante

$$\textbf{data}\ ESCRI\ m\ \alpha \quad = \quad m\ (\alpha, Traza)$$

Definir concreciones apropiadas de las clases *Monad*, *MonadaEscri* y *Transform*, y definir una clase *MonadaTraza* con una concreción apropiada. Además, definir concreciones para asegurar que si m es una mónada de excepciones, o de estados, entonces también lo es $ESCRI\ m$, y que si m es una mónada de escritura, también lo son $TEST\ m$ y $EXC\ m$.

10.4.2 Escribir un evaluador que utilice tanto *EXC* como *ESCRI* (del Ejercicio 10.4.1) y demostrar que los dos modos de componer los transformadores producen resultados diferentes. ¿Qué combinación se ajusta mejor a la implementación usual de la escritura? Repetir el ejercicio para *TEST* y *ESCRI*.

10.4.3 La función *elevar* debería ser un *morfismo entre mónadas*, en el sentido de que debería preservar la estructura de la mónada:

$$\begin{aligned} elevar\ (return\ x) &= return\ x \\ elevar\ (p \rhd q) &= (elevar\ p) \rhd (elevar \cdot q) \end{aligned}$$

Aquí, *return* y $\rhd$ en la parte izquierda están en la mónada m, y en la parte derecha están en la mónada $\tau\ m$. Demostrar que las operaciones *elevar* definidas en *TEST* y *EXC* son efectivamente morfismos entre mónadas.

10.5 Notas del capítulo

Moggi (1989, 1991) introdujo las mónadas en la computación como un modo de estructurar la semántica denotacional. Independientemente, Spivey (1990) halló que las mónadas suministraban un modo útil de estructurar el tratamiento de excepciones en los programas funcionales. Más adelante, Wadler (1990a, 1995) propuso las mónadas como una técnica general para estructurar programas funcionales.

Las mónadas se han utilizado para estructurar el compilador de Haskell de Glasgow, que está escrito, él mismo, en Haskell; véase Hall, Hammond, Partain, Jones y Wadler (1992). Cada fase del compilador utiliza una mónada para mantener cierta información. Por ejemplo, el comprobador de tipos utiliza una mónada que combina el estado (para mantener la sustitución en curso), un suministrador de nombres (para crear nuevos nombres de variables de tipo) y excepciones. Se han extendido las mónadas de interacción para incluir concurrencia; véase Jones, Gordon y Finne (1996). Los transformadores de mónadas se discuten en Liang, Hudak y Jones (1995).

El programa para el juego del ahorcado es una adaptación de un programa similar en Hutton y Meijer (1996).

En Launchbury (1993) se sugirió la notación **do** y se implementó por primera vez en Jones (1993) para una versión de Gofer. Más adelante, fue adoptada por Haskell.

Capítulo 11

Análisis sintáctico

Un *analizador sintáctico* es un programa que analiza una porción de texto para determinar su estructura lógica. Por ejemplo, el texto puede describir una expresión aritmética, un programa o una base de datos de información. Usualmente, el resultado del análisis es un árbol de algún tipo; por ejemplo, un analizador de expresiones aritméticas podría devolver un árbol conteniendo constantes en las hojas y operadores en los nodos internos. En este capítulo adoptaremos un enfoque monádico para diseñar analizadores. El material se utilizará también en el próximo capítulo.

11.1 Secuenciación

Comenzaremos considerando que un analizador es una función que toma como entrada una cadena de caracteres y devuelve como salida un árbol[1]:

$$\textbf{newtype}\ \mathit{Analiz} \quad = \quad \mathit{Mca}\ (\mathit{String} \rightarrow \mathit{Arbol})$$

El tipo *Analiz* se declara como un tipo nuevo, distinto del (pero isomorfo al) tipo subyacente de las funciones de cadenas en árboles.

Lo que es inadecuado en esta definición es que no proporciona una manera obvia de *secuenciar* analizadores. Por ejemplo, en un analizador de expresiones aritméticas podríamos desear analizar, en secuencia, un número, un operador y otro número. El primer analizador de números consumirá alguna parte inicial de la cadena de entrada, el analizador de operadores consumirá algo más de la entrada y el segundo analizador de números la porción restante. Se deduce que una idea mejor consiste en refinar el tipo *Analiz* para que devuelva como parte del resultado la porción no consumida de la entrada:

$$\textbf{newtype}\ \mathit{Analiz} \quad = \quad \mathit{Mca}\ (\mathit{String} \rightarrow (\mathit{Arbol}, \mathit{String}))$$

Antes de considerar cómo combinar dos analizadores en secuencia, veamos primero otro aspecto del análisis. Puede suceder que un analizador *falle* sobre cierta entrada. No es un error construir analizadores que puedan fallar. Por ejemplo, en un analizador de expresiones aritméticas, podríamos desear buscar o bien un símbolo de operador

[1] N. del T.: Siguiendo el criterio habitual, la constructora *Mca* ha de leerse *Mirar como analizador.*

o bien un paréntesis abierto. Uno de los dos analizadores auxiliares seguramente fallará. Con más generalidad, un analizador puede encontrar varios modos diferentes de estructurar en un árbol una cierta porción inicial de la entrada. Entonces el fallo corresponde al caso particular en que ya no hay más análisis posibles. Para manejar estas posibilidades, refinamos el tipo *Analiz* para que devuelva una lista de pares:

```
newtype Analiz   =  Mca (String → [(Arbol, String)])
```

Según este esquema, un analizador falla devolviendo la lista vacía. Por definición, un analizador es *determinista* si devuelve en todos los casos o bien una lista vacía o bien una lista unitaria. El último refinamiento procede de la observación de que diferentes analizadores devolverán probablemente diferentes tipos de árboles. Para tratar este aspecto, abstraemos el tipo específico *Arbol* y definimos un analizador como un elemento del siguiente tipo:

```
newtype Analiz α   =  Mca (String → [(α, String)])
```

En este punto, parece que un analizador es equivalente a *Est List*, siendo *Est* el transformador de estados del capítulo precedente, donde el estado es una cadena en lugar de un entero. Igual que hicimos en el capítulo anterior, definimos una función *aplicar* para aplicar un analizador:

```
aplicar            ::  Analiz α → String → [(α, String)]
aplicar (Mca f) s  =   f s
```

Utilizando *aplicar*, podemos definir una función *aplicarAnaliz* que devuelva uno de los resultados del análisis:

```
aplicarAnaliz      ::  Analiz α → String → α
aplicarAnaliz p    =   fst · head · aplicar p
```

La función *aplicarAnaliz* devuelve el primer componente del primer elemento de la lista de análisis. Utilizando *aplicar*, podemos hacer ahora que *Analiz* sea una concreción de una mónada:

```
instance Monad Analiz where
  return x   =  Mca f where f s = [(x, s)]
  p ▷ q      =  Mca f
    where f s = [(y, s'') | (x, s') ← aplicar p s, (y, s'') ← aplicar (q x) s']
```

En la definición de $p \triangleright q$, se aplica el analizador p a la cadena de entrada s, produciendo una lista de posibles análisis, cada uno emparejado con la correspondiente entrada remanente; entonces se aplica el analizador q a cada análisis y a su correspondiente entrada para producir, a su vez, una lista de resultados cuya concatenación constituye la respuesta final. Se debe demostrar también que se satisfacen las tres leyes monádicas, una tarea que dejaremos como ejercicio. Una vez conseguido que *Analiz* sea una mónada, podemos combinar analizadores en secuencia utilizando la notación **do**. Para empezar necesitaremos algunos analizadores básicos, uno de los cuales es

```
item   ::  Analiz Char
item   =   Mca f
           where f [ ]    = [ ]
                 f (c:cs) = [(c, cs)]
```


El analizador *item* devuelve el primer carácter de la entrada y falla si se agota dicha entrada. Así, *item* desempeña el mismo papel en los analizadores que *getChar* en la mónada de entrada-salida del capítulo precedente. El segundo analizador básico es aquel que siempre falla:

```
cero   ::  Analiz α
cero   =   Mca f where f s = [ ]
```

Se llama analizador *cero* porque devuelve cero análisis. También, se cumple

```
cero ▷ p          =  cero
p ▷ (const cero)  =  cero
```

Por tanto, *cero* es un elemento cero de ▷, del mismo modo que 0 es un elemento cero de ×. Para ilustrar la utilización de *item* y de *cero*, he aquí un analizador que reconoce un carácter que satisface una propiedad dada:

```
sat    ::  (Char → Bool) → Analiz Char
sat p  =   do {c ← item; if p c then return c else cero}
```

Utilizando *sat* podremos definir otros analizadores:

```
carac     ::  Char → Analiz ()
carac x   =   do {c ← sat (==x); return ()}

cadena          ::  String → Analiz ()
cadena [ ]      =   return ()
cadena (x : xs) =   do {carac x; cadena xs; return ()}

minuscula  ::  Analiz Char
minuscula  =   sat isLower

digito :: Analiz Int
digito = do {d ← sat isDigit; return (ord d − ord '0')}
```

El analizador *carac x* busca el carácter concreto x como siguiente elemento de la cadena de entrada, en tanto que *cadena xs* busca una cadena concreta *xs*; si tienen éxito, ambos analizadores devuelven (). El analizador *minuscula* busca una letra minúscula, mientras que *digito* busca un dígito, devolviendo en caso de éxito el valor numérico asociado.

Ejercicios

11.1.1 Demostrar que las leyes monádicas se satisfacen para la mónada *Analiz*.

11.1.2 Modificar *aplicarAnaliz* para que devuelva un mensaje de error apropiado en caso de que falle el análisis.

11.1.3 Demostrar que *cero* es un elemento cero de ◇, donde ◇ es el operador de composición asociado con la mónada *Analiz*.

11.1.4 Definir un analizador *mayuscula* para reconocer una letra mayúscula. Definir un analizador que reconozca una letra mayúscula seguida de una letra minúscula.

11.2 Alternativa

Para construir analizadores más sofisticados, necesitamos un operador que combine analizadores *alternativos*. Uno de tales operadores es

```
(alt)     :: Analiz α → Analiz α → Analiz α
p alt q   =  Mca f where f s = aplicar p s ++ aplicar q s
```

El analizador *p* **alt** *q* toma una cadena de entrada *s* y devuelve todos los análisis de *p* aplicado a *s*, seguido de todos los análisis de *q* aplicado a *s*.

El operador **alt** satisface ciertas leyes útiles. En primer lugar, forma un monoide tomando *cero* como elemento neutro:

```
cero alt p        =  p
p alt cero        =  p
p alt (q alt r)   =  (p alt q) alt r
```

En segundo lugar, se cumple la ley distributiva

```
(p alt q) ▷ r  =  (p ▷ r) alt (q ▷ r)
```

Esta ley, como las otras, es una consecuencia inmediata de las definiciones.

Para ilustrar el uso de **alt**, he aquí un analizador que reconoce una secuencia de letras minúsculas:

```
minusculas  :: Analiz String
minusculas  =  do {c ← minuscula; cs ← minusculas; return (c : cs)} alt return ""
```

Para ilustrar cómo funciona este analizador, supongamos que la cadena de entrada es "Mayuscula". En este caso, el analizador a la izquierda de **alt** falla porque 'M' no es una letra minúscula. Sin embargo, el analizador de la derecha triunfa; por tanto

```
minusculas "Mayuscula"  =  [( "", "Mayuscula")]
```

Con la cadena de entrada "mMayuscula", el analizador de la izquierda, triunfa, y también lo hace el analizador de la derecha por tanto

```
minusculas "sUpper"  =  [( "m", "Mayuscula"), ("", "mMayuscula")]
```

Finalmente, si se aplica a la cadena de entrada "esMayuscula", *minusculas* devuelve tres análisis:

```
[( "es", "Mayuscula"), ( "e", "sMayuscula"), ( "", "esMayuscula")]
```

Cada análisis en esta lista consume una porción de la cadena de entrada más pequeña que la de sus predecesores. Normalmente, sólo estamos interesados en el análisis que consume la porción inicial más larga posible de la entrada. Una solución es devolver solamente el primer análisis de la lista de posibles resultados, cuidando de que los análisis se realicen en el orden apropiado. Una alternativa mejor consiste en introducir otro combinador de alternativas **obien**, definido mediante

```
(obien)     :: Analiz α → Analiz α → Analiz α
p obien q   =  Mca f
               where f s = if null ps then aplicar q s else ps
                     ps = aplicar p s
```


Dicho con palabras, *p* **obien** *q* devuelve los mismos análisis que *p* excepto si *p* falla, en cuyo caso se devuelven los análisis de *q*. Si *p* y *q* son analizadores deterministas, entonces también lo es *p* **obien** *q*. El operador **obien** satisface las mismas leyes que **alt**, como el lector puede verificar. Para ilustrar el uso de **obien**, supongamos que **alt** se reemplaza por **obien** en la definición de *minusculas*. Entonces tenemos

```
minusculas "esMayuscula"  =  [( "es", "Mayuscula")]
```

En este caso, solamente se devuelve el análisis más largo, por tanto *minusculas* es un analizador determinista. El uso de **obien** en lugar de **alt** requiere cierto cuidado. Por ejemplo, consideremos una forma muy simple de expresión aritmética que conste, o bien de un único dígito, o bien de un dígito seguido de un signo más y de otro dígito. Consideremos los dos analizadores:

```
correcto  :: Analiz Int
correcto  =  digito alt suma

erroneo   :: Analiz Int
erroneo   =  digito obien suma

suma      :: Analiz Int
suma      =  do {m ← digito; carac '+'; n ← digito; return (m+n)}
```

Aplicado a la cadena de entrada "1+2", el analizador *correcto* devuelve dos análisis:

```
[(1, "+2"), (3, "")]
```

Sin embargo, *erroneo* devuelve sólo el primero de estos análisis. El analizador *digito* tiene éxito y por tanto no es invocado el analizador *suma*. Un modo de corregir *erroneo* consiste en reescribirlo de la siguiente forma:

```
mejor  :: Analiz Int
mejor  =  suma obien digito
```

Aplicado a la entrada "1+2", el analizador *mejor* devuelve [(3, "")]. El problema con *mejor* es que es ineficiente: aplicado a la entrada "1", analiza el dígito pero falla al tratar de buscar el signo más, por tanto el argumento izquierdo de **obien** falla. Como resultado, la entrada es analizada de nuevo para buscar un dígito. Esto no supone un problema grave en este pequeño ejemplo, pero lo sería si reemplazáramos *digito* por un analizador que reconozca una secuencia de dígitos. Dicha secuencia sería analizada de nuevo cuando fallase el analizador izquierdo. La mejor solución es *factorizar* el analizador para eliminar el dígito inicial de los dos analizadores componentes, escribiendo

```
loMejor   :: Analiz Int
loMejor   =  do {m ← digito; resto m}
resto m   =  do {carac '+'; n ← digito; return (m+n)} obien return m
```

El analizador *resto* trata el resto de la entrada, caso de haberlo. Toma un argumento *m* cuyo valor es el dígito analizado hasta el momento. Volveremos al tema de la eficiencia en la Sección 11.4, pero la lección aprendida aquí es que es razonable factorizar los analizadores para evitar repetir trabajo innecesario.

Ejercicios

11.2.1 ¿Es conmutativo el operador **alt**?

11.2.2 Definir una variante de *minusculas* que produzca una lista de análisis en la cual cada análisis consuma una porción de la entrada más larga que la de sus predecesores.

11.2.3 Demostrar que **obien** satisface las mismas cuatro leyes que **alt**.

11.3 Repetición

Generalizando la definición de *minusculas* de la sección precedente, podemos definir un combinador de analizadores que repite un análisis cero o más veces:

```
muchos     :: Analiz α → Analiz [α]
muchos p   =  do {x ← p; xs ← muchos p; return (x : xs)} obien return [ ]
```

Por ejemplo, tenemos ahora que *minusculas* = *muchos minuscula*. El siguiente analizador reconoce un identificador, el cual, por definición, es una cadena de letras y dígitos que comienza por una letra minúscula:

```
ident  :: Analiz String
ident  =  do {c ← minuscula; cs ← muchos alfanum; return (c : cs)}
```

La definición de *alfanum* se deja como ejercicio. A veces deseamos repetir un análisis *p* una o más veces en lugar de cero o más veces. Esto se puede hacer mediante un combinador *unoOMas* definido por

```
unoOMas     :: Analiz α → Analiz [α]
unoOMas p   =  do {x ← p; xs ← muchos p; return (x : xs)}
```

Por ejemplo, he aquí un analizador de números naturales:

```
nat  :: Analiz Int
nat  =  do {ds ← unoOMas digito; return (foldl1 (⊕) ds)}
        where m ⊕ n = 10 × m + n
```

El analizador *digito* se definió en la Sección 11.1.

Consideremos ahora cómo definir un analizador de un número entero, el cual, por definición, es una cadena no vacía de dígitos que posiblemente comience con un signo menos. Un primer enfoque consiste en definir

```
ent  :: Analiz Int
ent  =  do {carac '−'; n ← nat; return (−n)} obien nat
```

Otra definición, bastante más sofisticada es

```
ent  =  do {f ← op; n ← nat; return (f n)}
        where op = do {carac '−'; return negate} obien return id
```

El analizador *op* devuelve una función que, o bien es *negate* si el siguiente carácter es un signo menos, o bien es la función identidad.

11.3.1 Repetición con separadores

Consideremos el problema de analizar una lista no vacía de enteros separados por comas y encerrados entre corchetes. Se puede definir cada analizador mediante

```
ents  :: Analiz [Int]
ents  =  do carac '['
            n ← ent
            ns ← muchos (do {carac ','; ent})
            carac ']'
            return (n : ns)
```

Es útil abstraer este patrón bastante común de repetición definiendo un combinador especial *unoOMasCon* que funciona igual que *unoOMas* pero difiere en que las apariciones del primer analizador están separadas por otro analizador cuyo valor devuelto se ignora. La definición es

```
unoOMasCon     :: Analiz β → Analiz α → Analiz [α]
unoOMasCon q p =  do x ← p
                     xs ← muchos (do{q; p})
                     return (x : xs)
```

Para secuencias que puedan ser vacías definimos

```
muchosCon     :: Analiz β → Analiz α → Analiz [α]
muchosCon q p =  unoOMasCon q p obien return [ ]
```

11.3.2 Ignorando el espacio en blanco

En la mayoría de las aplicaciones, para hacer el texto más legible puede aparecer *espacio en blanco* (secuencia de los caracteres blanco, fin de línea y tabulador) entre cada dos *unidades léxicas* (identificadores, números, etc.). El analizador *espacio* reconoce espacios en blanco:

```
espacio  :: Analiz ()
espacio  =  muchos (sat esEspacio) ▷ return ()
```

Se deja como ejercicio escribir una definición apropiada de *esEspacio*. Para tener en cuenta el espacio en blanco, podemos definir un combinador de analizadores que ignora los espacios en blanco a ambos lados de un analizador:

```
unidad    :: Analiz α → Analiz α
unidad p  =  do {espacio; x ← p; espacio; return x}
```

Por ejemplo, el analizador

```
simbolo     :: String → Analiz ()
simbolo xs  =  unidad (cadena xs)
```

busca una cadena *xs* en la entrada, ignorando el espacio en blanco antes y después de la cadena.

Ejercicios

11.3.1 La definición de un analizador encajado entre otros dos analizadores que actúan de paréntesis y cuyos resultados son ignorados, se puede formular como un combinador de analizadores. Dar los detalles.

11.3.2 Rehacer como analizadores monádicos las funciones de procesamiento de texto descritas en la Sección 5.5.

11.4 Eficiencia

Los combinadores descritos hasta ahora son lo suficientemente potentes como para permitirnos traducir directamente cualquier descripción estructurada de una sintaxis en un programa funcional. Sin embargo, a menos que tengamos cuidado, el resultado puede ser un analizador con una eficiencia inesperada en tiempo y en espacio. En esta sección describiremos algunas técnicas sencillas para mejorar la eficiencia de los analizadores.

11.4.1 Recursión a la izquierda

Comenzaremos por construir un analizador para el tipo *Expr* definido mediante

```
data Expr  =  Con Int | Bin Op Expr Expr
data Op    =  Mas | Menos
```

He aquí una gramática para expresiones completamente parentizadas, expresada en lo que se conoce como forma *Backus-Naur*, abreviado BNF[2]:

```
expr        ::=  const | '(' expr opAditivo expr ')'
opAditivo   ::=  '+' | '-'
```

Tal gramática suministra una "descripción estructural natural" de la sintaxis requerida. Cada identificador de esta gramática describe una *categoría sintáctica*, mientras que los símbolos encerrados entre comillas denotan símbolos *terminales*, símbolos que aparecen realmente en el texto. En esta descripción se supone implícitamente que, donde sea aplicable, se pasa por alto el espacio en blanco entre símbolos terminales. La gramática particular para *expr* establece que una expresión es o bien una constante (entera), o bien una expresión compuesta que consta de un paréntesis abierto, seguido por una expresión, seguido por un signo menos o un signo más, seguido por otra expresión y seguido finalmente por un paréntesis cerrado. La gramática se traduce directamente a un analizador de expresiones:

```
expr  ::  Analiz Expr
expr  =   unidad const obien
          do {simbolo "(" ; t ←term; simbolo ")"; return t}
```

[2]N. del T. Del inglés *Backus Naur Form.*


```
const      =  do {n ←ent; return (Con n)}
term       =  do {t ←expr; op ←opAditivo; u ←expr; return (Bin op t u)}
opAditivo  =  do {simbolo "+"; return Mas} obien
              do {simbolo "-"; return Menos}
```

Supongamos ahora que queremos un analizador que también funcione para expresiones que no están completamente parentizadas. En tal caso, (+) y (−) deberían asociar a la izquierda. El modo usual de expresar esta gramática en modo BNF es el siguiente:

```
expr    ::=  expr opAditivo factor | factor
factor  ::=  const | '(' expr ')'
```

Esta gramática también se traduce directamente a un analizador:

```
expr    =  term obien factor
term    =  do {t ←expr; op ←opAditivo; u ←factor; return (Bin op t u)}
factor  =  unidad const obien
           do {simbolo "("; e ← expr; simbolo ")"; return e}
```

Sin embargo, este analizador es ineficiente —tan ineficiente que, de hecho, no proporciona ningún resultado—. El problema es que la primera acción del analizador *expr* es la invocación del analizador *term*, cuya primera acción es la invocación del analizador *expr* otra vez, conduciendo a un bucle infinito. Más aún, como hemos visto en la Sección 11.2, no es correcto reescribir *expr* como

```
expr  =  factor obien expr
```

porque el resultado solamente analiza un factor, ignorando el resto de la expresión. Tampoco funciona reemplazar **obien** por **alt** en la definición de *expr*: el analizador hallará el primer factor pero después entrará en un bucle infinito intentando completar el resto de los análisis. Este problema se conoce como *recursión por la izquierda* y es una dificultad presente en todos los analizadores recursivos, sean funcionales o no. La solución consiste en reescribir la gramática para *expr* del siguiente modo equivalente:

```
expr   ::=  factor resto
resto  ::=  opAditivo factor resto | nada
```

La nueva categoría sintáctica *resto* trata el resto de una expresión, caso de haberla, y *nada* denota la categoría que describe la secuencia vacía de símbolos terminales. El nuevo analizador para *expr* toma la forma

```
expr     =  do {t ←factor; resto t}
resto t  =  do {op ←opAditivo; u ←factor; resto (Bin op t u)}
            obien return t
```

El analizador *resto* toma un argumento que corresponde a la expresión analizada hasta el momento y que actúa como un parámetro acumulador.

11.4.2 Mejorando la pereza

Un programa funcional perezoso típico que consume una lista y produce otra lista funciona leyendo su entrada bajo demanda, calculando la primera parte del resultado tras

haber leído sólo una parte de la entrada. Por ejemplo, las funciones estándar *map* y *filter* tienen este comportamiento. No es razonable esperar un comportamiento similar de un analizador porque, en general, no se puede saber si la entrada será analizada con éxito hasta haberla consumido por completo. Sin embargo, en algunos casos podemos esperar hacerlo mejor. Consideremos por ejemplo, el analizador *muchos p*, cuya definición era

```
muchos p   =  unoOMas p obien return [ ]
unoOMas p  =  do {x ←p; xs ←muchos p; return (x : xs)}
```

Aplicando *aes* = *muchos* (*carac 'a'*) a la entrada "aab", obtenemos como resultado [("aa", "b")]. Bajo evaluación perezosa, se podría esperar que cada letra 'a' de la salida esté disponible después de haber procesado la correspondiente letra de la entrada. Después de todo, está garantizado que *aes* triunfa, produciendo la secuencia más larga posible de aes. Pero no es esto lo que sucede: *aes* lee completamente la entrada antes de generar ninguna salida. Se puede verificar que

```
aplicar aes ⊥                  =  ⊥
aplicar aes ('a' : ⊥)          =  ⊥
aplicar aes ('a' : 'a' : ⊥)    =  ⊥
aplicar aes ('a' : 'a' : 'b' : ⊥)  =  [("aa", 'b' : ⊥)]
```

La salida se produce solamente cuando se lee la letra 'b'.Sería interesante codificar el hecho de que, para cualquier *p*, el analizador *muchos p* siempre triunfa. Éste es el objetivo del analizador *forzar* definido mediante

```
forzar    ::  Analiz α →Analiz α
forzar p  =   Mca f
              where f cs  =  (fst (head rs), snd (head rs) : tail rs
                             where rs = aplicar p cs
```

He aquí la nueva definición de *muchos* que hace uso de *forzar*:

```
muchos p   =  forzar (unoOMas p obien return [ ])
unoOMas p  =  do {x ←p; xs ←muchos p; return (x : xs)}
```

Ahora se puede verificar que con *aes* = *muchos* (*carac 'a'*) se tiene

```
aplicar aes ⊥                  =  (⊥, ⊥) : ⊥
aplicar aes ('a' : ⊥)          =  ('a' : ⊥, ⊥) : ⊥
aplicar aes ('a' : 'a' : ⊥)    =  ('a' : 'a' : ⊥, ⊥) : ⊥
aplicar aes ('a' : 'a' : 'b' : ⊥)  =  [("aa", 'b' : ⊥)]
```

Otros analizadores también se pueden beneficiar del uso juicioso de *forzar*.

11.4.3 Limitando el número de resultados

Una diferencia crucial entre los analizadores alternativos **alt** y **obien** es que, si *p* y *q* son deterministas, también lo es *p* **obien** *q*, en tanto que *p* **alt** *q* es determinista solamente si uno de los dos, o bien *p* o bien *q*, es *cero*. Hay un modo más general de limitar el número de resultados. La idea consiste en proporcionar un modo de hacer explícito que sólo estamos interesados en el primer resultado devuelto por ciertos analizadores. Definimos el combinador *limitar* mediante


```
limitar      ::  Analiz α →Analiz α
limitar p    =   Mca (primero · aplicar p)

primero      ::  [α] →[α]
primero [ ]  =   [ ]
primero (r : rs)  =  [r]
```

Ahora, para analizadores deterministas *p* y *q* tenemos

```
p obien q  =  limitar (palt q)
```

Por otra parte, *limitar* (*p* **alt** *q*) es determinista incluso si *p* y *q* no lo son.

El lector puede preguntarse por qué hemos definido una función *primero*, cuando es equivalente a la expresión *take* 1. La respuesta tiene que ver con la eficiencia en espacio. Tenemos

```
take 1 (r : rs)  =  r : take 0 rs
```

Aunque *take* 0 *rs* se evalúa a [ ], esta evaluación no se lleva a cabo hasta que se requiere realmente el valor. El efecto es que se retiene un puntero al resto de los resultados *rs*, causando por tanto una ineficiencia potencial en espacio. Este problema se evita definiendo *primero* explícitamente.

Ejercicios

11.4.1 Demostrar que *p* **obien** *q* = *limitar* (*p* **alt** *q*).

11.5 Notas del capítulo

Para una introducción general a las técnicas prácticas de análisis sintáctico y su aplicación al diseño de compiladores, consúltese Aho, Sethi, y Ullman (1986). El diseño de analizadores funcionales ha sido una aplicación preferida de la programación funcional durante muchos años; por ejemplo Burge (1975) y Wadler (1985). Para un tratamiento más moderno véase Fokker (1995), Hutton (1992) y Hutton y Meijer (1996), el cual se se ha tomado como base para este capítulo. El problema inverso al de análisis sintáctico es el de la *impresión estructurada*[3], o la presentación de un documento estructurado en un formato estéticamente agradable. Véase Hughes (1995) si se desea consultar el diseño de una librería de impresión estructurada.

[3]N. del T. En el original, *pretty printing*.

Capítulo 12

Un demostrador automático

Este capítulo final está dedicado a un proyecto completo de programación: el diseño e implementación de una calculadora funcional para realizar automáticamente ciertas clases de demostraciones. Está basado muy estrechamente en un demostrador de teoremas escrito por Mike Spivey. Aunque la calculadora solamente suministra un pequeño subconjunto de las facilidades que se podría desear tener dentro de un asistente automático de demostraciones y es restrictivo en algunos otros aspectos, todavía será suficientemente potente para demostrar muchas de las leyes no aplicativas descritas en capítulos precedentes.

12.1 Consideraciones básicas

La idea consiste en construir una función *calcular* cuyo tipo es

$$calcular \;::\; [Ley] \rightarrow Expr \rightarrow Computo$$

El primer argumento de *calcular* es una lista de leyes que se pueden aplicar. Cada ley tiene un nombre y toma la forma de una ecuación. El segundo argumento es una expresión de partida y el resultado es un cómputo. Un cómputo consiste en una secuencia de pasos; cada paso se genera aplicando a la expresión en curso la parte izquierda de una de las leyes dadas. El cómputo termina cuando no se pueden aplicar más leyes. El proceso es completamente automático y no requiere intervención por parte de un demostrador de teoremas humano. En la Sección 12.5 se dan ejemplos de cómputos producidos por *calcular*.

Las leyes, expresiones y cómputos son, todos ellos, elementos de tipos de datos apropiados que serán definidos en la sección siguiente. Para describir y presentar los valores de estos tipos de datos de una forma legible, necesitaremos cuatro funciones de conversión:

$$\begin{array}{lcl} analizaLey & :: & String \rightarrow Ley \\ analizaExpr & :: & String \rightarrow Expr \\ imprimeExpr & :: & Expr \rightarrow String \\ imprimeComp & :: & Computo \rightarrow String \end{array}$$

Por ejemplo, podemos utilizar *analizaLey* para introducir una lista de leyes sobre pares:

```
pares
 = map analizaLey [
   "definición fst:        fst.par(f,g) = f",
   "definición snd:        snd.par(f,g) = g",
   "definición prod:       prod(f,g) = par (f.fst,g.snd)",
   "absorción de par:      par(f,g) · h = par (f.h,g.h)"]
```

Cada ley consta de un nombre y una ecuación. El nombre de la ley se termina con dos puntos y la ecuación consta de dos expresiones separadas por un signo igual. Las expresiones se construyen a partir de constantes, como en el caso de *fst* y *par*, y de variables, como en el de *f* y *g*. En la siguiente sección se da la sintaxis precisa que se permite para las expresiones.

Utilizando *analizaExpr* e *imprimeComp*, podemos definir una orden *simplificar* que lleva a cabo un cómputo:

$$\begin{array}{lcl} \textit{simplificar} & :: & [\textit{Ley}] \rightarrow \textit{String} \rightarrow \textit{IO}\ () \\ \textit{simplificar leyes} & = & \textit{putStr} \cdot \textit{imprimeComp} \cdot \textit{calcular leyes} \cdot \textit{analizaExpr} \end{array}$$

La orden *simplificar* utiliza *analizaExpr* para interpretar una cadena de caracteres como una expresión, luego utiliza *calcular* para generar un cómputo, después utiliza *imprimeComp* para convertir el resultado en una cadena y, finalmente, presenta dicha cadena. Por ejemplo,

```
? simplificar pares "prod(f,g).par(h,k)"

  prod (f,g).par(h,k)
=     {definición prod}
  par(f.fst,g.snd).par(h,k)
=     {absorción por un par}
  par(f.fst.par(h,k),g.snd.par(h,k))
=     {definición fst}
  par f.h,g.snd.par(h,k))
=     {definición snd}
  par(f.h,g.h)
```

Los pasos del cómputo se muestran en el formato convencional; el nombre de la ley utilizada en cada paso se muestra entre llaves, separando las dos expresiones a las que se aplica. En cada paso de un cómputo se *encaja* alguna subexpresión de la expresión en curso en el lado izquierdo de alguna regla. Este encaje o ajuste da lugar a una *sustitución* de las variables de la regla. Por ejemplo, en el segundo paso del cómputo anterior, la expresión

$$\textit{par}\ (f \cdot \textit{fst}, g \cdot \textit{snd}) \cdot \textit{par}\ (h, k)$$

se ajusta con éxito al lado izquierdo de la ley de absorción por un par, dando lugar a una sustitución en la que la variable *f* de la ley de absorción por un par se vincula a la expresión $f \cdot \textit{fst}$, la variable *g* se vincula a $g \cdot \textit{snd}$ y *h* se vincula a *par*(*h*, *k*). El resultado de este paso de cómputo es la concreción de la parte derecha de la ley de absorción por un par en la que cada variable se ha reemplazado por la expresión a la cual está vinculada. Nótese que, si bien las expresiones que se manipulan también contienen variables, tales variables no son modificadas por los pasos de un cómputo. Las sustituciones sólo actúan sobre las variables que pertenecen a las leyes. Un aspecto crucial del diseño de *calcular* es que las leyes se aplican exclusivamente de izquierda a derecha. Ello es así para evitar bucles en los cómputos. Si se pudieran aplicar las leyes en ambas direcciones, entonces la calculadora automática podría oscilar siempre aplicando una ley en una dirección y a continuación aplicándola en la dirección contraria. Un argumento similar se aplica a una ley tal como

$$\textit{iterate}\ f \ = \ \textit{cons} \cdot \textit{par}\ (\textit{id}, \textit{iterate}\ f \cdot f)$$

Ésta es la definición de *iterate* expresada en un estilo no aplicativo. La presencia del término *iterate f* en ambos lados de la ley significa que un calculador automático podría aplicar la ley un número infinito de veces, de modo que nunca alcanzaría una conclusión.

A veces queremos aplicar las leyes en ambas direcciones, mientras que otras veces queremos hacer uso de leyes como la de *iterate*. Los bucles potencialmente infinitos se evitan o bien expresando una ley de un modo más general de lo necesario, o bien priorizando de algún modo la aplicación de las leyes.

12.1.1 Demostraciones

A veces se desea demostrar ecuaciones en vez de simplemente calcular formas alternativas para las expresiones. Un método consiste en considerar cada lado de la ecuación por separado y simplificar ambos para obtener el mismo resultado. Se pueden unir los dos cómputos invirtiendo los pasos del segundo. Para ilustrar esta idea, supongamos que hemos introducido la siguiente lista de leyes:

```
leyes  =  filtros ++ cond ++ otros

filtros
 = map analizaLey [
   "definición filter:    filter p = concat.map(caja p)",
   "definición caja:      caja p = if(p,enlistar,nil)"]

cond
 = map analizaLey [
   "if respecto a composición:    if(p,f,g).h = if(p.h,f.h,g.h)",
   "composición respecto a if:    h.if(p,f,g) = if (p,h.f,h.g)"]

otros
 = map analizaLey [
   "nil constante:              nil.f = nil",
   "naturalidad de nil:         map f.nil = nil",
   "naturalidad de enlistar:    map f.enlistar = enlistar · f",
   "naturalidad de concat:      map f.concat = concat · map (map f)",
   "map funtor:                 map f.map g = map (f,g)"]
```

Dada la función *demostrar*, definida más abajo, se tiene

```
? demostrar leyes "filter p.map f = map f.filter(p.f)"

  filter p.map f
=     {definición filter}
  concat.map(caja p).map f
=     {map funtor}
  concat.map(caja p.f)
=     {definición caja}
  concat.map(if(p,enlistar,nil).f)
=     {if respecto a composición}
  concat.map(if(p.f,enlistar.f,nil.f))
=     {nil constante}
  concat.map(if(p.f,enlistar.f,nil))
=     {naturalidad de enlistar}
  concat.map(if(p.f,map f.enlistar,nil))
=     {naturalidad de nil}
  concat.map(if(p.f,map f.enlistar,map f.nil))
=     {composición respecto a if}
  concat.map(map f.if(p.f,enlistar,nil))
=     {definición caja}
  concat.map(map f.caja(p.f))
=     {map funtor}
  concat.map(map f).map(caja(p.f))
=     {concat natural}
  map f.concat.map(caja(p.f))
=     {definición filter}
  map f.filter(p.f)
```

Aunque el cómputo ha sido presentado como una única secuencia de pasos, se lleva a cabo en dos fases distintas. En primer lugar, se simplifica la expresión *filter p · map f* de la parte izquierda, dando como resultado *concat · map (if (p · f, enlistar · f, nil))*. En segundo lugar, se simplifica la expresión *map f · filter (p · f)* de la parte derecha, dando lugar al mismo resultado. Entonces, las dos partes del cómputo se ponen juntas invirtiendo los pasos de la segunda mitad. La principal ventaja de este esquema es su simplicidad; no tenemos que inventar otro formato de demostración, ni tenemos que aplicar las leyes de derecha a izquierda para alcanzar el resultado deseado.

Hay dos desventajas obvias. Una es de estilo, en el sentido de que la mitad del cómputo ha de efectuarse hacia atrás. La otra es que el método puede fallar. No siempre sucede que una ecuación se satisface a causa de que ambos lados pueden simplificarse al mismo resultado (véase el Ejercicio 12.1.3).

Se puede definir la función *demostrar* como sigue:

```
demostrar        :: [Ley] → String → IO ()
demostrar leyes  =  putStr · imprimeComp · demostrarEcn leyes · AnalizaEcn

demostrarEcn :: [Ley] → (Expr, Expr) → Computo
demostrarEcn leyes (izq, der)
  =  unir (calcular leyes izq) (calcular leyes der)
```

Las funciones que quedan pendientes:


```
analizaEcn  :: String → (Expr, Expr)
unir        :: Computo → Computo → Computo
```

se definirán en la sección siguiente.

12.1.2 Consideraciones adicionales

Es improbable que una calculadora automática pueda exhibir la habilidad de un calculador humano. Dado un repertorio cualquiera de ejemplos, es probable que un calculador humano encuentre cómputos más breves o que impliquen expresiones intermedias más pequeñas de lo que una máquina podría esperar conseguir. A diferencia de una máquina, un calculador humano aplicará sin problemas las leyes en cualquier dirección sin caer por ello en un agujero negro.

En el esquema de cómputo automático que estamos diseñando, solamente hay dos grados de libertad: la elección de la regla a aplicar y la elección de la subexpresión que ha de ser reemplazada. Se puede eliminar el primer grado de libertad teniendo en cuenta el orden en que las leyes son presentadas a la calculadora: cuando sean aplicables dos leyes diferentes, se escogerá la que aparezca primero en la lista. El segundo grado de libertad viene representado por el orden en que se presentan las subexpresiones de una expresión dada como candidatas a ser concreciones de leyes: si se pueden aplicar leyes a dos subexpresiones diferentes, entonces se puede escoger la subexpresión que ocurre primero en la enumeración. Aun así, todavía debemos decidir si dar preferencia a las leyes o a las subexpresiones en las demostraciones. En otras palabras, si en cada paso de un cómputo preferimos aplicar una ley particular o cambiar una subexpresión dada.

En términos generales, el foco de interés de un paso de reducción en un cómputo está usualmente en una parte de la expresión, en lugar de la aplicación de una ley concreta. Sin embargo, en los pasos principales del cómputo suelen aplicarse ciertas clases de leyes; éstas son leyes que reducen la complejidad de las expresiones intermedias. Un buen ejemplo es la ley $f \cdot id = f$ (y también $id \cdot f = f$). Es improbable que sea erróneo aplicar esta ley tan pronto como surja la oportunidad. Análogamente, la aplicación temprana de una ley del estilo de $nil \cdot f = nil$ y $fst \cdot par\ (f, g) = f$ puede ayudar a reducir el tamaño de las expresiones intermedias. Por darlas un nombre, las llamaremos leyes *básicas*. Parece entonces razonable dividir la lista de leyes considerada en dos subclases, las leyes básicas y las restantes, y dar prioridad a las leyes básicas en los pasos de un cómputo.

Teniendo esto en cuenta, cambiaremos ligeramente la definición de *simplificar* y *calcular*:

```
simplificar leyes
  =  putStr · imprimeComp · calcular (basicas, otras) · analizaExpr
     where (basicas, otras) = clasifica leyBasica leyes
```

La función *clasifica p* clasifica los elementos de una lista en dos sublistas, aquellos que satisfacen *p* y los restantes. La función *leyBasica :: Ley → Bool* se define en la sección siguiente. Cambiamos el tipo de *calcular* a

```
calcular  :: ([Ley], [Ley]) → Expr → Computo
```

Y de manera similar cambiamos *demostrar*. Los ejemplos de cómputo mostrados anteriomente fueron producidos con las definiciones modificadas.

Ejercicios

12.1.1 Algunas de las leyes que se podría desear utilizar en los cómputos dependen de que se satisfagan ciertas condiciones auxiliares. Dar un ejemplo de tal ley. Estas leyes se pueden utilizar en el esquema propuesto solamente si suponemos que se satisfacen las condiciones auxiliares.

12.1.2 Sugerir una descripción apropiada de una ley básica.

12.1.3 ¿Se puede demostrar la ley $zipp \cdot par\ (map\ f,\ map\ g) = map\ (par\ (f, g))$ simplificando ambos lados al mismo resultado? Se puede demostrar simplificando la expresión $zipp \cdot unzip \cdot map\ (par\ (f, g))$ de dos modos esencialmente diferentes; uno de ellos utiliza la ley $zipp \cdot unzip = id$, y el otro utiliza la definición de *unzip*. Completar los detalles.

12.1.4 Las dos leyes funtoriales de *map* son importantes en muchas demostraciones. ¿Cuál de las dos orientaciones es probable que sea más útil? Utilizando la ley para *zipp* del ejercicio previo, demostrar que $zipp \cdot par\ (id, map\ g) = map\ (par\ (id, g))$. ¿Utiliza su demostración las leyes funtoriales en la dirección preferente?

12.1.5 Hay muchos modos de mejorar el diseño de una calculadora. Por ejemplo, se pueden leer las leyes de uno o más ficheros; la idea de clasificar las leyes en dos categorías se puede generalizar a una noción más abstracta de estrategia de demostración, y también se puede hacer una calculadora interactiva. Dar una lista de otras características o mejoras deseables en una calculadora automática.

12.2 Expresiones, leyes y cómputos

En el corazón de la calculadora se encuentra el tipo de datos *Expr* de las expresiones; la mayoría de los componentes del calculador tienen que ver con reconocer, manipular o imprimir expresiones. En el marco propuesto, toda expresión denota una función. Las expresiones se construyen a partir de variables y constantes (ambas denotan funciones), utilizando la composición funcional como combinador básico. Las variables no toman argumentos, pero las funciones constantes pueden tomar cualquier número de argumentos, que también son expresiones. Por tanto, adoptaremos la siguiente definición de *Expr*:

```
data Expr
  = Var NomVar | Con NomConst [Expr] | Componer [Expr]
    deriving (Eq)

type NomVar   = Char
type NomConst = String
```


Por simplicidad, supondremos que los nombres de una sola letra denotan variables y los nombres de más de una letra denotan constantes. Evitaremos símbolos especiales en los nombres de las constantes; por ejemplo, escribiremos *suma* en lugar de (+) y *cero* en vez de 0. Por tanto, los nombres de las constantes son secuencias de, al menos, dos caracteres alfanuméricos, comenzando por una letra. Nos referiremos colectivamente a las variables y a las constantes como *términos*. Declaramos el tipo *Expr* como una concreción de la clase *Eq* porque necesitaremos comprobar la igualdad de expresiones. He aquí algunos ejemplos de expresiones:

```
fst          ⇒ Con "fst" [ ]
fst · f      ⇒ Componer [Con "fst" [ ], Var 'f']
map f        ⇒ Con "map" [Var 'f']
prod (f, g)  ⇒ Con "prod" [Var 'f', Var 'g']
```

Cada constante toma una tupla de argumentos, que puede ser la tupla vacía (como en el caso de *fst*), una tupla unitaria (como en el caso de *map*), un par, y así sucesivamente. El esquema es muy simple y no permite representar expresiones como *foldr f e*, que utilizan la currificación. Pero siempre podremos representar el término *foldr f e* mediante *foldr (f, e)*. Las dos ecuaciones que definen *foldr* se pueden expresar en forma funcional

```
foldr (f, e) · nil   = e
foldr (f, e) · cons  = f · prod (id, foldr (f, e))
```

En estas ecuaciones, tanto *f* como *cons* son funciones no currificadas que toman un par como argumento.

El diseño pretende que la asociatividad de la composición funcional esté predefinida. Así, tanto $(f \cdot g) \cdot h$ como $f \cdot (g \cdot h)$ se representarán por la misma expresión

```
Componer [Var 'f', Var 'g', Var 'h']
```

Como consecuencia, *Expr* tiene dos invariantes de la representación: (i) la expresión *Componer xs* es válida solamente si la longitud de *xs* es mayor o igual que dos; (ii) ninguna expresión de la forma *Componer xs* contiene un elemento de *xs* que sea a su vez de la forma *Componer ys*. Para mantener estos invariantes, se componen las expresiones mediante la función *componer*:

```
componer      :: [Expr] → Expr
componer xs   = if unitaria xs
                then head xs
                else Componer (concat (map descomponer xs))

descomponer          :: Expr → [Expr]
descomponer (Var v)  = [Var v]
descomponer (Con f xs) = [Con f xs]
descomponer (Componer xs) = xs
```

La función *componer* será aplicada únicamente a listas no vacías. Se define la complejidad de una expresión como el número de sus términos:

```
complejidad          :: Expr → Int
complejidad (Var v)  = 1
```

```
complejidad (Con f xs)      = 1
complejidad (Componer xs)   = length xs
```

Es posible dar una definición más refinada de *complejidad* pero la que hemos dado es suficiente en la práctica. Al imprimir una expresión, pondremos paréntesis encerrando sólo tuplas de longitud mayor o igual que dos o englobando argumentos que no son términos simples:

```
imprimeExpr          :: Expr → String
imprimeExpr (Var v)  = [v]

imprimeExpr (Con f xs)
  | null xs          = f
  | simple xs        = f ++ "␣" ++ imprimeExpr (head xs)
  | otherwise        = f ++ "(" ++
                       unirCon ",␣" (map imprimeExpr xs) ++ ")"

imprimeExpr (Componer xs)  = unirCon "·" (map imprimeExpr xs)

simple      :: [Expr] → Bool
simple xs   = unitaria xs ∧ noArgs (head xs)

noArgs              :: Expr → Bool
noArgs (Var v)      = True
noArgs (Con f xs)   = null xs
noArgs (Componer xs) = False
```

La función *unirCon* se definió en la Sección 5.5. Una definición más sofisticada de *imprimeExpr* debería ser capaz de partir una expresión en puntos adecuados si es demasiado grande para entrar en una línea. Este refinamiento se deja como ejercicio.

12.2.1 Análisis sintáctico de expresiones

Queda por definir *analizaExpr*. Por supuesto, utilizaremos los combinadores descritos en el capítulo precedente. La definición es

```
analizaExpr  :: String → Expr
analizaExpr  = aplicarAnaliz expr
```

El analizador sintáctico *expr* reconoce una secuencia no vacía de términos separados por símbolos de composición:

```
expr  :: Analiz Expr
expr  = do {xs ← unoOMasCon (simbolo "·") term; return (componer xs)}
```

El analizador sintáctico *term* reconoce una variable o una constante:

```
term  :: Analiz Expr
```



```
term  = do espacio
           c ← letra
           cs ← muchos alfanum
           if null cs
             then return (Var c)
             else do xs ← argumento
                     return (Con (c : cs) xs)
```

El analizador sintáctico *argumento* reconoce el argumento de una función constante, caso de haber alguno:

```
argumento  :: Analiz [Expr]
argumento  = tupla orelse (notupla orelse return [ ])
```

El analizador sintáctico *tupla* reconoce una expresión entre paréntesis:

```
tupla  :: Analiz [Expr]
tupla  = do simbolo "("
            xs ← unoOMasCon (simbolo "·") expr
            simbolo ")"
            return xs
```

El analizador sintáctico *notupla* reconoce una variable o una constante sin argumentos:

```
notupla  :: Analiza Expr
notupla  = do espacio
              c ← letra
              cs ← muchos alfanum
              if null cs
                then return [Var c]
                else return [Con (c : cs) [ ]]
```

Esto completa la definición de *analizaExpr*. La función *analizaEcn* es similar y aún más simple:

```
analizaEcn  :: String → (Expr, Expr)
analizaEcn  = aplicarAnaliz ecn

ecn  :: Analiz (Expr, Expr)
ecn  = do espacio
          x ← expr
          simbolo "="
          y ← expr
          return (x, y)
```

12.2.2 Leyes

Una ley consta de tres componentes lógicos: un nombre y dos expresiones.

```
type Ley     = (NomLey, Expr, Expr)
type NomLey  = String
```

Por definición, una ley es *básica* si la complejidad de su parte izquierda es mayor que la de su parte derecha:

```
leyBasica                  :: Ley → Bool
leyBasica (nombre, izq, der) = (complejidad izq > complejidad der)
```

Son posibles otras definiciones de una ley básica, pero ésta funciona suficientemente bien en la mayoría de las situaciones. La última función sobre leyes es *analizaLey*:

```
analizaLey  :: String → Ley
analizaLey  = aplicarAnaliz ley

ley  :: Analiz Ley
ley  = do espacio
          nombre ← unoOMas (sat (≠ ':'))
          simbolo ":"
          (x, y) ← ecn
          return (nombre, x, y)
```

12.2.3 Cómputos

Un cómputo consiste en una expresión de partida junto con una secuencia de pasos. Cada paso consiste en el nombre de la ley que se está aplicando junto con la expresión resultante. Por tanto, podemos definir

```
type Computo  = (Expr, [Paso])
type Paso     = (NomLey, Expr)
```

La conclusión de un cómputo es la última expresión del mismo:

```
conclusion            :: Computo → Expr
conclusion (x, pasos) = if null pasos then x else snd (last pasos)
```

La función *unir* para concatenar dos cómputos se puede definir mediante

```
unir          :: Computo → Computo → Computo
unir izq der  = (fst izq, snd izq ++ comprobar x y ++ reordenar der)
                where x = conclusion izq
                      y = conclusion der
```

La función *comprobar* inserta una indicación de fallo si las dos conclusiones son diferentes:

```
comprobar      :: Expr → Expr → [Paso]
comprobar x y  = if x == y then [ ] else [("...??? ...", y)]
```

La función *reordenar* invierte los pasos de un cómputo, convirtiendo

$$(x, [(r_1, y_1), \ldots, (r_n, y_n)]) \quad \text{en} \quad [(r_n, y_{n-1}), \ldots, (r_2, y_1), (r_1, x)]$$

Su definición es

```
reordenar        :: Computo → [Paso]
reordenar (x, ss) = snd (foldl puente (x, [ ]) ss)
                    where puente (x, rs) (r, y) = (y, (r, x) : rs)
```


La función *imprimeComp* se define mediante

```
imprimeComp          :: Computo → String
imprimeComp (x, ss)  = "⌉␣␣" ++ imprimeExpr x ++
                       "⌉" ++ concat (map imprimePaso ss)

imprimePaso          :: Paso → String
imprimePaso (causa, x) = "=␣␣{" ++ causa ++ "}⌉" ++
                         "␣␣" ++ imprimeExpr x ++ "⌉"
```

Ejercicios

12.2.1 Son posibles varias definiciones alternativas de *Expr*. Una de ellas consiste en decir que una expresión es, o bien una variable o una constante (sin ningún argumento), o es la aplicación de una expresión a otra. Formalizar el tipo de datos apropiado y dar la representación de $fst \cdot fold\ f\ e$ como ejemplo. ¿Cuáles son las ventajas y desventajas de esta representación?

12.2.2 Una generalización útil del tipo propuesto para *Expr* consiste en reemplazar la constructora *Componer* por una constructora más general *Asoc NomConst*. Se permiten construcciones tales como *Asoc f xs* siempre que *f* sea una operación asociativa sobre funciones. Dar un ejemplo donde podría ser útil esta generalización.

12.2.3 Dar una definición mejor de *imprimeExpr* que trate la partición de líneas de un modo razonable.

12.2.4 Como alternativa a proporcionar explícitamente una función *imprimeExpr*, se puede hacer que *Expr* sea una concreción de la clase *Show*. Completar los detalles. Mostrar cómo efectuar la impresión de cómputos de un modo similar.

12.3 Ajustes y sustituciones

Ahora mostraremos cómo ajustar una expresión y a un patrón de expresión x. El calculador utiliza ajuste de cuando x es la parte izquierda de alguna ley e y es alguna subexpresión de la expresión en curso en un cómputo parcial. El resultado del ajuste es un *conjunto* de posibles sustituciones. Una sustitución asocia cada variable de x con una expresión tal que cuando se aplique la sustitución a x, el resultado es y. Diremos también que y es una *concreción* de x bajo dicha sustitución. Un ajuste puede fallar, dando como resultado el conjunto vacío de sustituciones, o puede tener éxito de una o más maneras. Por ejemplo, el ajuste de

$$foo\ (f \cdot g, f \cdot g) \quad \text{a} \quad foo\ (a \cdot b \cdot c, a \cdot b \cdot c)$$

tiene éxito de dos modos diferentes, o bien vinculando f a $a \cdot b$ y g a c, o bien vinculando f a a y g a $b \cdot c$. Aunque el calculador automático seleccionará sólo una sustitución en cada paso de cómputo, es importante tener en cuenta los ajustes múltiples en el proceso de determinar las sustituciones válidas. Como veremos, el esquema general

es ajustar expresiones, ajustando las correspondientes subexpresiones y combinando los resultados. Por ejemplo, al ajustar

$$foo\ (f \cdot g,\ bar\ g) \quad \text{a} \quad foo\ (a \cdot b \cdot c,\ bar\ c)$$

la subexpresión $f \cdot g$ se ajusta a $a \cdot b \cdot c$, dando lugar a dos posibles sustituciones. Sólo cuando la expresión *bar g* se ajusta a *bar c* se rechaza una de las dos sustituciones. Una decisión prematura de seleccionar una sola sustitución para el primer par de subexpresiones puede dar lugar a que se pierda una sustitución de éxito.

12.3.1 Sustituciones

Implementaremos las sustituciones como elementos del tipo *Sust*, donde

```
type Sust  =  [(NomVar, Expr)]
```

Dicho con palabras, una sustitución se representa por una lista de vínculos. Una lista vacía no representa la sustitución indefinida, sino más bien una sustitución en la que cada variable *v* se asocia con *Var v*. Así, la lista vacía representa la sustitución identidad. La función *vinculo* devuelve la expresión asociada a una variable dada en una sustitución:

```
vinculo              ::  Sust → NomVar → Expr
vinculo [ ] v        =   Var v
vinculo ((u, x) : s) v  =  if u == v then x else vinculo s v
```

También querremos aplicar una sustitución a una expresión:

```
aplicarSus                  ::  Sust → Expr → Expr
aplicarSus s (Var v)        =   vinculo s v
aplicarSus s (Con f xs) s   =   Con f (map (aplicarSus s) xs)
aplicarSus s (Componer xs)  =   componer (map (aplicarSus s) xs)
```

Finalmente, querremos extender una sustitución con un vínculo adicional:

```
extender :: Sust → (NomVar, Expr) → [Sust]
extender s (v, x)
    | y == x        =  [s]
    | y == Var v    =  [(v, x) : s]
    | otherwise     =  [ ]
    where y = vinculo s v
```

La función *extender* devuelve una lista de sustituciones; esta lista es vacía si el vínculo adicional es incompatible con la sustitución dada; en otro caso es una lista unitaria.

12.3.2 Ajuste

La función *ajustar* toma un par de expresiones y devuelve un conjunto de sustituciones. Por razones de eficiencia, se define *ajustar* en términos de una función más general *xajustar*:

```
ajustar        ::  (Expr, Expr) → [Sust]
ajustar (x, y) =   xajustar [ ] (x, y)
```


El primer argumento de *xajustar* es un parámetro acumulador de tipo *Sust*. La siguiente definición de *xajustar* se derivará formalmente en la Sección 12.5 a partir de una definición directa de *ajustar*; por ahora, simplemente explicaremos la definición cláusula por cláusula. En primer lugar,

```
xajustar                ::  Sust → (Expr, Expr) → [Sust]
xajustar s (Var v, x)   =   extender s (v, x)
```

Ajustar una variable *v* a una expresión arbitraria *x* siempre tiene éxito. Sin embargo, extender la sustitución *s* con (*v*, *x*) puede devolver una lista vacía de sustituciones. Seguidamente,

```
xajustar s (Con f xs, Var v)          =  [ ]
xajustar s (Con f xs, Componer ys)    =  [ ]

xajustar s (Con f xs, Con g ys)
  =  if f == g then xajustarLista s (zip xs ys) else [ ]
```

Ajustar una constante a una variable o a una composición siempre falla. Por otra parte, ajustar una constante a la misma constante tiene éxito si los elementos correspondientes de las dos listas de argumentos ajustan. La función *xajustarLista* se define mediante

```
xajustarLista              ::  Sust → [(Expr, Expr)] → [Sust]
xajustarLista s [ ]        =   [s]
xajustarLista s (xy : xys) =   concat [xajustarLista t xys | t ← xajustar s xy]
```

Ajustar una lista vacía siempre tiene éxito; ajustar una lista (*xy* : *xys*) implica calcular primero *xajustar s xy* para producir una lista de posibles sustituciones, y después refinar cada sustitución *t* de esta lista calculando *xajustarLista t xys*. El resultado es una lista de listas de sustituciones, que se aplana mediante *concat*.

Las tres cláusulas restantes de *xajustar* son

```
xajustar s (Componer xs, Var v)       =  [ ]
xajustar s (Componer xs, Con g ys)    =  [ ]

xajustar s (Componer xs, Componer ys)
  =  concat (map (xajustarLista s) (alinear xs ys))
```

Ajustar una composición a una variable o a una constante falla porque *Componer xs* es una expresión válida solamente si *length xs* ≥ 2. Ajustar dos secuencias de composiciones tiene éxito si lo tiene el ajuste de segmentos alineados adecuadamente. La función *alinear* se define más adelante. Para cada lista *xys* de segmentos adecuadamente alineados, se calcula el valor de *xajustarLista s xys* y se concatenan los resultados.

Para definir *alinear xs ys*, supongamos que *length xs* = *m* y *length ys* = *n*. Si $m > n$, entonces *xs* no se puede ajustar a *ys* porque no hay suficientes elementos en *ys* con los que ajustar. Si $m \leq n$, entonces cada elemento de *xs* se puede ajustar a algún segmento de *ys*. En otras palabras, si $ys_1, ys_2, \ldots, ys_m$ es una partición de *ys* en segmentos contiguos no vacíos, entonces el primer elemento de *xs* se ha de ajustar a ys_1, el segundo elemento a ys_2 y así sucesivamente; por tanto necesitaremos utilizar una

función *partes m* que devuelva una lista con todas las formas posibles de descomponer una secuencia en una lista de *m* componentes. Dada *partes*, podemos definir

```
alinear        :: [Expr] →[Expr] →[[(Expr, Expr)]]
alinear xs ys  =  [zip xs (map componer zs) | zs ←partes (length xs) ys]
```

La definición de *partes* es

```
partes              :: Int →[α] →[[[α]]]
partes 0 [ ]        =  [[ ]]
partes 0 (x : xs)   =  [ ]
partes (n + 1) [ ]  =  [ ]
partes (n + 1) (x : xs) =  map (nuevo x) (partes n xs) ++
                           map (pegar x) (partes (n + 1) xs)
```

La primera ecuación dice qué existe exactamente una forma de descomponer la lista vacía en una lista de cero componentes, a saber, la partición vacía. La segunda ecuación dice que no existe forma alguna de partir una secuencia no vacía en cero componentes. La tercera ecuación dice que, si $m > 0$, no existe forma de descomponer la secuencia vacía en *m* componentes. Finalmente, la cuarta ecuación dice que toda partición de $x : xs$ en $n + 1$ componentes se obtiene, o bien generando una partición *yss* de *xs* de longitud *n* y después añadiendo la [*x*] como primer componente, o bien generando una partición $ys : yss$ de *xs* de longitud $n + 1$ y "añadiendo" después *x* al primer componente *ys*. Las funciones *nuevo* y *pegar* se definen mediante

```
nuevo          :: α →[[α]] →[[α]]
nuevo x yss    =  [x] : yss
pegar          :: α →[[α]] →[[α]]
pegar x (ys : yss) =  (x : ys) : yss
```

Ejercicios

12.3.1 Demostrar que *aplicarSus* [ ] = *id*.

12.3.2 Se puede definir directamente la función *ajustar*. Dos de sus cláusulas son:

```
ajustar (Con f xs, Con g ys)
  = if f == g then ajustarLista (zip xs ys) else [ ]
justar (Componer xs, Componer ys)
  = concat (map ajustarLista (alinear xs ys)
```

Completar las cláusulas que faltan. (La respuesta a este ejercicio se da en la Sección 12.5.)

12.3.3 La función *ajustarLista* del ejercicio precedente se define mediante

```
ajustarLista   :: [(Expr, Expr)] →[Sust]
ajustarLista   =  concat · map unificar · pcLista · map ajustar
```

Esta definición se lee como sigue: primero se calcula la lista de listas que representa las sustituciones parciales (utilizando *map ajustar*), después se combinan todas las sustituciones parciales en una lista de sustituciones candidatas (usando *pcLista*), después

se unifica cada candidato (utilizando *unificar*) y finalmente se reúnen las sustituciones válidas (utilizando *concat*). La función *pcLista* (abreviatura de "producto cartesiano de listas") se define informalmente mediante

$$pcLista\ [xs_1, \ldots, xs_n] \quad = \quad [[x_1, \ldots, x_n] \mid x_1 \leftarrow xs_1, \ldots, x_n \leftarrow xs_n]$$

Dar una definición formal de *pcLista* (la respuesta a este ejercicio se da en la Sección 12.5).

12.3.4 La función *unificar* :: [*Sust*] → [*Sust*] toma una lista de sustituciones parciales y las combina en una única sustitución completa. Por tanto, *unificar* devuelve, o bien una lista vacía si las sustituciones parciales no se pueden combinar, o bien una lista unitaria que contiene la sustitución unificada. Se puede definir mediante

```
unificar       :: [Sust] →[Sust]
unificar [ ]   =  [[ ]]
unificar (s : ss) =  concat [union s t | t ←unificar ss]
```

La función *union* :: *Sust* → *Sust* → [*Sust*] toma dos sustituciones y, si son compatibles, las fusiona en una sustitución única. Dar una definición de *union* (la respuesta a este ejercicio se da en la Sección 12.5).

12.4 Subexpresiones y reescritura

En la sección precedente vimos cómo ajustar una expresión (el lado izquierdo de una ley) a otra expresión. Ahora queremos ajustar una expresión a una subexpresión de una expresión dada. Si el ajuste tiene éxito, entonces la correspondiente concreción del lado derecho de la ley reemplaza a la subexpresión. Para llevar a cabo esta sustitución, es necesario localizar las subexpresiones dentro de la expresión circundante. Por tanto, definimos

```
type SubExpr  =  (Posicion, Expr)
```

Una subexpresión de una expresión *x* puede ser, o bien *toda x*, o un *segmento* de *x* (si *x* es una composición de términos), o una subexpresión de algún termino de *x*. Ello conduce a la siguiente declaración del tipo *Posicion*:

```
data Posicion  =  Toda | Seg Int Int | Pos Int Posicion
```

La posición *Seg j k* denota un segmento que comienza en la posición *j* y se extiende $k > 2$ términos; una posición *Pos j pos* denota la subexpresión ubicada en la posición *pos* de la expresión ubicada en la posición *j*.

Veamos cómo funciona este esquema en la expresión

$$fst \cdot par\ (f \cdot fst,\ g \cdot snd) \cdot zip$$

He aquí una lista completa de las subexpresiones de esta expresión y las posiciones en las cuales aparece cada subexpresión:

```
fst · par (f · fst, g · snd) · zip   en   Toda
fst · par (f · fst, g · snd)         en   Seg 0 2
```

```
par (f · fst, g · snd) · zip   en  Seg 1 2
fst                            en  Pos 0 Toda
par (f · fst, g · snd)         en  Pos 1 Toda
f · fst                        en  Pos 1 (Pos 0 Toda)
f                              en  Pos 1 (Pos 0 (Pos 0 Toda))
fst                            en  Pos 1 (Pos 0 (Pos 1 Toda))
g · snd                        en  Pos 1 (Pos 1 Toda)
g                              en  Pos 1 (Pos 1 (Pos 0 Toda))
snd                            en  Pos 1 (Pos 1 (Pos 1 Toda))
zip                            en  Pos 2 Toda
```

Las subexpresiones anteriores están listadas en el orden en que son generadas por la función *subexprs*:

```
subexprs                :: Expr → [SubExpr]
subexprs (Var v)        =  [(Toda, Var v)]
subexprs (Con f xs)     =  [(Toda, Con f xs)] ++subterms xs
subexprs (Componer xs)  =  [(Toda, Componer xs)] ++segmentos xs ++
                           subterms xs

subterms :: [Expr] → [SubExpr]
subterms xs
  = [(Pos j pos, y)     | j ←[0.. n−1], (pos, y) ←subexprs (xs !! j)]
    where n = length xs

segmentos :: [Expr] → [SubExpr]
segmentos xs
  = [(Seg j k, Componer (take k (drop j xs))) | k ←[2.. n−1], j ←[0.. n−k]]
    where n = length xs
```

La función *subexprs* produce primero segmentos y después subtérminos; probablemente esto es razonable, aunque son posibles otros órdenes de generación más sofisticados. Además, los subtérminos se generan de izquierda a derecha; en la práctica, las manipulaciones se hacen usualmente de derecha a izquierda. Otra opción consiste en hacer que *subexprs* tome un argumento adicional, la posición en que fue cambiada por última vez la expresión en curso, y entonces generar primero las subexpresiones que están cerca de esa posición. Se dejan como ejercicio todos estos refinamientos. Finalmente, necesitamos una función que reemplace subexpresiones por otras expresiones:

```
reemplazar              :: Expr → Posicion → Expr → Expr
reemplazar x Toda y     =  y

reemplazar (Con f xs) (Pos j pos) y
  = Con f (take j xs ++[reemplazar (xs !! j) pos y] ++drop (j + 1) xs)
reemplazar (Componer xs) (Pos j pos) y
  = componer (take j xs ++[reemplazar (xs !! j) pos y] ++drop (j + 1) xs)
reemplazar (Componer xs) (Seg j k) y
  = componer (take j xs ++[y] ++drop (j + k) xs)
```


12.4.1 Reescritura

Finalmente, estamos en condiciones de definir la función *calcular*. Calcular a partir de una expresión significa utilizar repetidamente las leyes para reescribir dicha expresión hasta que no sea posible efectuar más pasos. Por tanto, podemos definir

```
calcular        :: ([Ley], [Ley]) → Expr → Computo
calcular pls x  =  (x, repetir (reescribir pls) x)
```

Recordemos que la lista de leyes está dividida en dos sublistas: las leyes básicas que se intentan aplicar siempre en primer lugar, y las demás.

La función *reescribir* produce todos los posibles modos de reescribir una expresión:

```
reescribir :: ([Ley], [Ley]) → Expr → [Paso]
reescribir (leyesIz, leyesDe) x
  = concat ([reescri ley sx x     | ley ←leyesIz, sx ←subexprs x]
            ++[reescri ley sx x   | sx ←subexprs x, ley ←leyesDe])
```

Las dos clases de leyes se tratan de modo diferente. Primero se intentan las leyes básicas, probando a aplicar cada ley de la secuencia con todas las posibles subexpresiones. En el caso de las otras leyes, el procedimiento es diferente: en cada subexpresión de la secuencia se ensaya con todas las posibles leyes. La función *reescri* produce todos los modos posibles de reescribir una expresión con una sola ley:

```
eescri :: Ley → SubExpr → Expr → [Paso]
reescri (nombre, izq, der) (pos, y) x
  = [(nombre, reemplazar x pos (aplicarSus s der))     | s ←ajustar (izq, y)]
```

Finalmente, *repetir* aplica la primera reescritura de cada paso:

```
repetir :: (Expr→[Paso]) → Expr → [Paso]
repetir fr x
  = if null pasos then [ ] else (n, y) : repetir fr y
    where pasos  =  fr x
          (n, y) =  head pasos
```

El no determinismo se resuelve escogiendo simplemente el primer paso entre los muchos posibles. Por supuesto, se pueden utilizar otros esquemas. Por ejemplo, en un entorno interactivo la elección podría ser tomada por un usuario. Otra posibilidad es explorar el árbol completo de los cómputos posibles y devolver el más corto.

Ejercicios

12.4.1 Modificar la definición de *subexprs* de modo que se puedan producir diferentes órdenes de generación de un modo simple. Ello implicará añadir un argumento extra a *calcular* para indicar cuál de los órdenes debe escogerse.

12.4.2 Reeescribir el calculador para implementar la idea de que *suexprs* tome un argumento adicional, a saber, la posición del último reemplazamiento.

12.4.3 Explorar la idea de hacer interactivo el calculador, de modo que se pueda ofrecer una elección de pasos alternativos al usuario.

12.5 Comprobando el calculador

¿Es muy útil el calculador en la práctica? El único modo de responder a esta pregunta es comprobarlo con un ejemplo no trivial. En esta sección vamos a usar el calculador para generar parte de la definición de *xajustar* dada en la Sección 12.3 a partir de una definición directa de *ajustar*. La parte que hemos elegido automatizar es la más difícil; el resto de la síntesis es suficientemente fácil como para no necesitarse automatismos. La definición directa de *ajustar* (véase el Ejercicio 12.3.2) es la siguiente:

```
ajustar                      :: (Expr, Expr) → [Sust]
ajustar (Var v, x)           =  [[(v, x)]]
ajustar (Con f xs, Var v)    =  [ ]
ajustar (Con f xs, componer ys) = [ ]

ajustar (Con f xs, Con g ys)
  = if f == g then ajustarLista (zip xs ys) else [ ]

ajustar (Componer xs, Var v)     = [ ]
ajustar (Componer xs, Con g ys)  = [ ]

ajustar (Componer xs, Componer ys)
  = concat (map ajustarLista (alinear xs ys))
```

La función *ajustarLista* (véase el Ejercicio 12.3.3) se define mediante:

```
ajustarLista  :: [(Expr, Expr)] → [Sust]
ajustarLista  = concat · map unificar · pcLista · map ajustar
```

La función *ajustarLista* calcula las sustituciones de cada par de expresiones, colecciona todas las combinaciones posibles y unifica cada combinación.

La función *pcLista* se define mediante

```
pcLista          :: [[α]] → [[α]]
pcLista [ ]      = [[ ]]
pcLista (xs : xss) = [y : ys | y ← xs, ys ← pcLista xss]
```

La función *unificar* se define mediante

```
unificar         :: [Sust] → [Sust]
unificar [ ]     = [[ ]]
unificar (s : ss) = concat [union s t | t ← unificar ss]
```

Se puede definir la función *unión* mediante

```
union     :: Sust → Sust → [Sust]
union s t = if compatible s t then [mezclar s t] else [ ]
```

Dos sustituciones son compatibles si y sólo si concuerdan en las variables de su dominio común. Se puede implementar la función *mezclar* mediante (++), o uniendo dos listas de vínculos y eliminando los duplicados. La función *union* no sólo es complicada de definir, sino también costosa de calcular. Una idea mejor, y que motiva la definición eficiente de *xajustar*, es calcular la unión de dos sustituciones sólo cuando una de

ellas es una sustitución que tenga un solo vínculo. Es lo que hace la función *extender*, especificada mediante

```
extender s (v, x) = union s [(Var v, x)]
```

La definición de *extender* dada en la Sección 12.3 implementa esta especificación del modo obvio. Independientemente de cómo se implemente, necesitaremos dos suposiciones acerca de *union*. Primero, *union* [ ] *s* = [*s*] para todo *s*. Dicho con palabras, la sustitución vacía es compatible con cualquier sustitución y es el elemento neutro de *mezclar*. Segundo, definimos *une* mediante

```
une      :: [Sust] → [Sust] → [Sust]
une ss ts = concat [union s t | s ← ss, t ← ts]
```

La segunda suposición es que *une* es asociativa:

```
une ss (une ts us) = une (une ss ts) us
```

Se deduce de la primera sustitución que [[ ]] es el elemento neutro de *une*.

12.5.1 Las generalizaciones

Especificamos la función *xajustar* mediante

```
xajustar     :: Sust → (Expr, Expr) → [Sust]
xajustar s xy = concat [union s t | t ← ajustar xy]
```

Dado que *union* [ ] *t* = [*t*], obtenemos *ajustar* = *xajustar* [ ]. Es muy fácil utilizar la definición de *ajustar* para derivar las siguientes ecuaciones de *xajustar*:

```
xajustar s (Var v, x)            = extender s (v, x)
xajustar s (Con f xs, Var v)     = [ ]
xajustar s (Con f xs, Componer ys) = [ ]
xajustar s (Componer xs, Var v)  = [ ]
xajustar s (Componer xs, Con g ys) = [ ]

xajustar s (Con f xs, Con g ys)
  = if == g then concat [union s t | t ← ajustarLista (zip xs ys)] else [ ]
xajustar s (Componer xs, Componer ys)
  = concat [union s t | t ← concat (map ajustarLista (alinear xs ys))]
```

Se pueden simplificar las dos últimas ecuaciones introduciendo

```
xajustarLista     :: Sust → [(Expr, Expr)] → [Sust]
xajustarLista s xys = concat [union s t | t ← ajustarLista xys]
```

Entonces, tenemos

```
xajustar s (Con f xs, Con g ys)
  = if f == g then xajustarLista s (zip xs ys) else [ ]

xajustar s (Componer xs, Componer ys)
  = concat (map (xajustarLista s) (alinear xs ys))
```

La primera ecuación es inmediata y la segunda se deja como un ejercicio de fácil resolución. En este punto hemos derivado la definición requerida de *xajustar* excepto por el aspecto crucial de que *xajustarLista* se define como anteriormente, y no en la forma dada en la Sección 12.3, a saber

$$\begin{array}{lcl} xajustarLista\ s\ [\ ] & = & [s] \\ xajustarLista\ s\ (xy : xys) & = & concat\ [xajustarLista\ t\ xys \mid t \leftarrow xajustar\ s\ xy] \end{array}$$

Esta definición no necesita la función *union*, que es la razón principal para expresar el ajuste en términos de *xajustar*.

Para demostrar que las dos definiciones de *xajustarLista* son equivalentes, hemos de pasar por una etapa intermedia. Tanto *xajustar* como *xajustarLista* toman una sola sustitución como parámetro acumulador, mientras que las siguientes funciones *yajustar* e *yajustarLista* toman sendas *listas* de sustituciones como argumento:

$$\begin{array}{lcl} yajustar & :: & [Sust] \rightarrow (Expr, Expr) \rightarrow [Sust] \\ yajustar\ ss\ xy & = & concat\ [union\ s\ t \quad \mid s \leftarrow ss, t \leftarrow ajustar\ xy] \end{array}$$

$$\begin{array}{lcl} yajustarLista & :: & [Sust] \rightarrow [(Expr, Expr)] \rightarrow [Sust] \\ yajustarLista\ ss\ xys & = & concat\ [union\ s\ t \quad \mid s \leftarrow ss, t \leftarrow ajustarLista\ xys] \end{array}$$

Es obvio que $xajustar\ s = yajustar\ [s]$ y que $xajustarLista\ s = yajustarLista\ [s]$. De hecho, excepto en un lugar, utilizaremos estas definiciones de *xajustar* y de *xajustarLista* en lugar de las precedentes. Es también obvio que $ajustar = yajustar\ [[\ ]]$. De hecho, un modo de abordar la derivación de un programa eficiente para *ajustar* es primero expresar el programa en términos de *yajustar* y después refinar el resultado para utilizar *xajustar*.

12.5.2 Los combinadores

La primera tarea consiste en reescribir todas las ecuaciones previas en forma funcional no currificada. Para este fin necesitaremos ciertos combinadores especiales. En primer lugar, el combinador *estr*[1] definido mediante

$$\begin{array}{lcl} estr & :: & (\alpha \rightarrow [\beta]) \rightarrow [\alpha] \rightarrow [\beta] \\ estr\ f & = & concat \cdot map\ f \end{array}$$

El uso de *estr* no es absolutamente necesario pero ayudará a abreviar las expresiones. El siguiente grupo de combinadores es *pc*, *pcd* y *pci*[2]:

$$\begin{array}{lcl} pc & :: & ([\alpha], [\beta]) \rightarrow [(\alpha, \beta)] \\ pc\ (xs, ys) & = & [(x, y) \mid x \leftarrow xs, y \leftarrow ys] \end{array}$$

$$\begin{array}{lcl} pcd & :: & (\alpha, [\beta]) \rightarrow [(\alpha, \beta)] \\ pcd\ (x, ys) & = & [(x, y) \quad \mid y \leftarrow ys] \end{array}$$

[1]N. del T.: En el original *star*. El nombre proviene del operador "estrella de Kleisli" utilizado en teoría de categorías.

[2]N. del T.: Respectivamente abreviaturas de *producto cartesiano*, *producto cartesiano por la derecha* y *producto cartesiano por la izquierda*.

$$\begin{array}{lcl} pci & :: & ([\alpha], \beta) \rightarrow [(\alpha, \beta)] \\ pci\ (xs, y) & = & [(x, y) \quad \mid x \leftarrow xs] \end{array}$$

El uso de *pc* es conveniente pero no necesario, porque $pc = estr\ pcd \cdot pci$, como el lector puede comprobar a partir de las leyes de las listas intensionales.

Finalmente, necesitaremos el combinador *asi* (abreviatura de "asociar por la izquierda"), definido mediante

$$\begin{array}{lcl} asi & :: & (\alpha, (\beta, \gamma)) \rightarrow ((\alpha, \beta), \gamma) \\ asi\ (x, (y, z)) & = & ((x, y), z) \end{array}$$

He aquí las definiciones que utilizaremos:

```
ajustes
 = map analizaLey [
   "def ajustarLista: ajustarLista  =  estr unificar.pcLista.map ajustar",
   "def unificar:     unificar.nil  =  enlistar.nil",
   "def unificar:     unificar.cons =  estr union.pcd.prod(id,unificar)",
   "def pcLista:      pcLista.nil   =  enlistar.nil",
   "def pcLista:      pcLista.cons  =  map cons.pc.prod(id,pcLista)"]

yajustes
 = map analizaLey [
   "def yajustarLista: yajustarLista = estr union.pc.prod(id,ajustarLista)",
   "def yajustar:      yajustar      = estr union.pc.prod(id,ajustar)"]

xajustes
 = map analizaLey [
   "def xajustarLista: xajustarLista = yajustarLista.prod(enlistar,id)",
   "def xajustar:      xajustar      = yajustar.prod(enlistar,id)"]
```

Las definiciones anteriores son traducciones directas de las precedentes a la forma funcional. Nótese que no hay una definición de *ajustar* porque no es necesaria en la demostración. Para esta última necesitaremos un número de leyes entre las cuales están las siguientes:

```
leyes1 = estrs ++ cps ++ ids ++ funtors

estrs
 = map analizaLey [
   "estr después de map:     estr f.map g = estr(f,g)",
   "estr después de concat:  estr f.concat = estr(estr  f)",
   "estr después de enlistar estr f.enlistar = f",
   "estr después de nil:     estr f.nil = nil",
   "estr después de estr:    estr(estr f.g) = estr f.estr g"]
```

La ley "estr después de estr" es una consecuencia directa de las leyes "estr después de concat" y "estr después de map". Se puede utilizar el calculador para demostrar estas leyes, tarea que dejamos como ejercicio.

```
cps = pcs ++ pcis ++ pcds

pcs
 = map analizaLey [
   "def  pc:                    pc = estr pcd.pci"]

pcis
 = map analizaLey [
   "pci después de nil:       pci.prod(nil,id) = nil",
   "pci después de enlistar: pci.prod(enlistar,id) = enlistar",
   "pci después de map:       pci.prod(map f,id) = map(prod(f,id)).pci",
   "pci después de concat:    pci.prod(concat,id) = estr  pci.pci",
   "pci después de id:        pci.prod(id,g) = map(prod(id,g)).pci",
   "pci después de estr:      pci.prod(estr f,g) = estrt(pci.prod(f,g)).pci"]

pcds
 = map analizaLey [
   "pcd después de nil:       pcd.prod(nil,id) = nil",
   "pcd después de enlistar: pcd.prod(id,enlistar) = enlistar",
   "pcd después de map:       pcd.prod(id,map g) = map(prod(id,g)).pcd",
   "pcd después de concat:    pcd.prod(id,concat) = estr  pcd.pcd",
   "pcd después de id:        pcd.prod(f,id) = map(prod(f,id)).pcd",
   "pcd después de estr       pcd.prod(f,estr g) = estr(pcd.prod(f,g)).pcd"]
```

Las leyes "pci después de estr" y "pcd después de estr" son consecuencias directas de las correspondientes "después de concat" y "después de map". Las leyes "después de id" son concreciones de las correspondientes "después de map", introducidas simplemente para evitar el uso de la ley *id* = *map id*. Las leyes que faltan son

```
ids
 = map analizaLey [
   "id elem neutro izq:  id.f = f"
   "id elem neutro der:  f.id = f"]

funtors
 = map analizaLey [
   "prod funtor:          prod(f,g).prod(h,k) = prod(f. h,g.k)",
   "prod funtor:          prod(id,id) = id",
   "map funtor:           map f.map g = map(f.g)",
   "map funtor:           map id = id",
   "map después de nil:   map f.nil = nil",
   "map después de cons:  map f.cons = cons.prod(f,map f)"]
```

También necesitaremos algunas leyes adicionales que serán descritas a su debido tiempo. Es justo preguntarse por qué hemos escogido las leyes precedentes y no otras. La respuesta es simplemente que hemos recolectado todas las leyes de *estr*, *pcd*, etc., que podrían resultar útiles. De hecho, muchas de las leyes enumeradas anteriormente no se utilizan en la demostración.

12.5.3 Los cómputos

Nuestro objetivo final es demostrar


```
xajustarLista.prod(id,nil) = enlistar.fst
xajustarLista.prod(id,cons) = estr xajustarLista.pci.prod(xajustar,id).asi
```

Puestas en modo aplicativo estas ecuaciones serían:

$$\begin{aligned} &\textit{xajustarLista}\ (s, [\,]) &&= [s] \\ &\textit{xajustarLista}\ (s, xy : xys) \\ &&&= \textit{concat}\ [\textit{xajustarLista}\ (t, xys) \mid t \leftarrow \textit{xajustar}\ (s, xy)] \end{aligned}$$

Excepto por el hecho de que *xajustarLista* no esta de forma currificada, ésta es una transcripción fiel de la definición de la Sección 12.3. Sería agradable pulsar simplemente un botón para hacer que el cómputo se desarrollara, pero las cosas no son tan sencillas y hemos de proceder más despacio. Esquemáticamente, la demostración se realiza en tres etapas:

- Obtener una definición recursiva de *ajustarLista*.
- Utilizar el resultado para obtener una definición recursiva de *yajustarLista*.
- Utilizar el resultado para obtener la definición requerida de *xajustarLista*.

Para la primera etapa necesitamos una conjetura:

```
conjeturas
 = map analizaLey [
   "conjetura:  estr(pcd.prod(id,g)).pc = pc.prod(id,estr g)"]

? demostrar leyes1 "estr(pcd.prod(id,g)).pc = pc.prod(id,estr g)"

  estr(pcd.prod(id,g)).pc
=   {def. pc}
  estr(pcd.prod(id,g)).estr pcd.pci
=   {estr después de estr}
  estr(estr(pcd.prod(id,g)).pci).pci
=   {pcd después de estr}
  estr(pcd.prod(id,estr g)).pci
=   {estr después de map}
  estr pcd.map(prod(id,estr g)).pci
=   {pci después de id}
  estr pcd.pcip.rod(id,estr g)
=   {def. pc}
  pc.prod(id,estr g)
```

Una vez demostrada la conjetura, podemos proseguir con la tarea de obtener una definición recursiva de *ajustarLista*:

```
leyes2 = ajustes ++ leyes1 ++ conjeturas

? demostrar leyes2 "ajustarLista.nil = enlistar.nil"

  ajustarLista.nil
=    {def. ajustarLista}
```

```
  estr unificar.pcLista.map ajustar.nil
=    {map después de nil}}
  estr unificar.pcLista.nil
=    {def pcLista}
  estr unificar.enlistar.nil
=    {estr después de enlistar}
  unificar.nil
=    {def unificar}
  enlistar.nil
```

La segunda demostración es como sigue:

```
? demostrar leyes2 "ajustarLista.cons = estr
                                        union.pc.prod(ajustar,ajustarLista)"

  ajustarLista.cons
=    {def ajustarLista}
  estr unificar.pcLista.map ajustar.cons
=    {map después de cons}
  estr unificar.pcLista.cons.prod(ajustar,map ajustar)
=    {def pcLista}
  estr unificar.map cons.pc.prod(id,pcLista).prod(ajustar,map ajustar)
=    {estr después de map}
  estr(unificar.cons).pc.prod(id,pcLista).prod(ajustar,map ajustar)
=    prod funtor}
  estr(unificar.cons).pc.prod(id.ajustar,pcLista.map ajustar)
=    {id elem neutro izq}}
  estr(unificar.cons).pc.prod(ajustar,pcLista.map ajustar)
=    {def. unificar}
  estr(estr union.pcd.prod(id,unificar)).pc.prod(ajustar,pcLista.map ajustar)
=    {estr después de estr}
  estr union.estr(pcd.prod(id,unificar)).pc.prod(ajustar,pcLista.map ajustar)
=    {conjetura}
  estr union.pc.prod(id,estr unificar).prod(ajustar,pcLista.map ajustar)
=    {prod funtor}
  estr union.pc.prod(id.ajustar,estr unificar pcLista.map ajustar)
=    {id elem neutro izq}}
  estr union.pc.prod(ajustar,estr unificar.pcLista.map ajustar)
=    {def pc}
  estr union.estr pcd.pci.prod(ajustar,estr unificar.pcLista.map ajustar)
=    {def ajustarLista}
  estr union.estr pcd.pci.prod(ajustar,ajustarLista)
=    {def pc}
  estr union.pc.prod ajustar,ajustarLista)
```

Consideramos los resultados como nuevas definiciones que reemplazan a las anteriores:

```
nuevosAjustes
 = map analizaLey [
   "ajustarLista1: ajustarLista.nil  =  enlistar.nil",
   "ajustarLista2: ajustarLista.cons =  estr
                                        union.pc.prod(ajustar, ajustarLista)"]
```


La siguiente etapa consiste en utilizar las nuevas definiciones para expresar *yajustarLista* recursivamente. Es aquí donde necesitaremos los hechos de que *une* es asociativo y que tiene como elemento neutro a [[ ]], siendo *une* = *estr union* · *pc*:

```
unes
 = map analizaLey [
   "introd de une:  estr union.pc = une",
   "une asoc:       une.prod(id, une.f) = une.prod(une, id).asi.prod(id, f)",
   "idem une:       une.prod(id, enlistar.nil) = fst"]
```

La ley asociativa para *une* es en realidad

```
une.prod(id, une) = une.prod.(une, id).asi
```

Se necesita expresarla de esta forma para evitar utilizar la ley del funtor *prod* en la dirección opuesta, provocando con ello un bucle en los cómputos. Las leyes de *asi* son

```
asis
 = map analizaLey[
   "asi natural:  asi.prod(f,prod(g,h))=prod(prod(f,g),h).asi",
   "asi auxiliar: asi.prod(f,id)=prod(prod(f,id),id).asi"]
```

Se introduce la segunda ley para evitar usar *prod* (*id, id*) = *id* en la dirección opuesta. Estamos listos para la demostración:

```
leyes3 = nuevosAjustes ++ yajustes ++ unes ++ asis ++ leyes1

? demostrar leyes3 "yajustarLista.prod(id,nil) = fst"

  yajustarLista.prod(id,nil)
=     {def yajustarLista}
  estr union.pc.prod(id,ajustarLista).prod(id,nil)
=     introd de une}
  une.prod(id,ajustarLista)prod(id,nil)
=     {prod funtor}
  une.prod(id.id,ajustarLista.nil)
=     {id elem neutro izq}
  une.prod(id,ajustarLista.nil)
=     {ajustarLista1}
  une.prod(id,enlistar.nil)
=     {idem une}
  fst

? demostrar leyes3 "yajustarLista.prod(id,cons) =
                                        yajustarLista.prod(yajustar,id).asi"

   yajustarLista · prod (id,cons)
=     {def yajustarLista}
```

```
  estr union.pc.prod(id,ajustarLista).prod(id,cons)
=     {introd de une}
  une.prod(id,ajustarLista).prod(id,cons)
=     {prod funtor}
  une.prod(id.id,ajustarLista.cons)
=     {id elem neutro izq}
  une.prod(id,ajustarLista.cons)
=     {ajustarLista2}
  une.prod(id,estr union.pc.prod(ajustar,ajustarLista))
=     {introd. de une}
  une.prod(id,uneprod(ajustar,ajustarLista))
=     {une asoc}}
  une.prod(une,id).asi.prod(id,prod(ajustar,ajustarLista))
=     {asi natural}
  une.prod(une,id).prod(prod(id,ajustar),ajustarLista).asi
=     {prod funtor}
  une.prod(une.prod(id,ajustar),id.ajustarLista).asi
=     {id elem neutro izq}
  une.prod(une.prod(id,ajustar) ajustarLista).asi
=     {introd de une}
  une.prod(estr union.pc.prod(id,ajustar),ajustarLista).asi
=     {def yajustar}
  une.prod(yajustar,ajustarLista).asi
=     {id elem neutro der}
  une.prod(yajustar,ajustarLista.id).asi
=     {id elem neutro izq}
  une.prod(id.yajustar,ajustarLista.id).asi
=     {prod funtor}
  une.prod(id,ajustarLista).prod(yajustar,id).asi
=     introd de une}
  estr union.pc.prod(id,ajustarLista).prod(yajustar,id).asi
=     {def yajustarLista}
  yajustarLista.prod(yajustar,id).asi
```

Hemos demostrado en estilo aplicativo

$$\begin{aligned} &\mathit{yajustarLista}\ (\mathit{ss}, [\ ]) &&= \mathit{ss} \\ &\mathit{yajustarLista}\ (\mathit{ss}, \mathit{xy} : \mathit{xys}) &&= \mathit{yajustarLista}\ (\mathit{yajustar}\ (\mathit{ss}, \mathit{xy}), \mathit{xys}) \end{aligned}$$

Y por tanto hemos mostrado que *curry yajustarLista* = *foldl* (*curry yajustar*). En cierto sentido, hemos conseguido el principal objetivo de la optimización de *ajustar*, a saber: derivar un cómputo en el cual no aparece la función *union*. Es fácil utilizar la definición de *ajustar* para obtener una definición de *yajustar* que no haga referencia a *union*. Por tanto ¿por qué preocuparse con el refinamiento extra de reemplazar *yajustar* por *xajustar*? La respuesta tiene que ver con la eficiencia. Si *xajustar* (*s, xy*) falla, devolviendo la lista vacía, entonces *xajustarLista* (*s, xy* : *xys*) falla inmediatamente; pero si *yajustar* (*ss, xy*) falla entonces *yajustarLista* (*ss, xy* : *xys*) no falla hasta que han sido procesados los restantes pares *xys*. En este contexto, véase el Ejercicio 12.5.4.

Para continuar, consideramos que las definiciones recursivas de *yajustarLista* reemplazan a las anteriores:


```
nuevosYAjustes
 = map analizaLey [
   "yajustarLista1: yajustarLista.prod(f,nil) = f.fst",
   "yajustarLista2: yajustarLista.prod(f,cons)
                     = yajustarLista.prod(yajustar,id).asi.prod(f,id)"]
```

Las formas que toman estas ecuaciones son necesarias para evitar la utilización de la ley *prod* (*f, g*) = *prod* (*id, g*) · *prod* (*f, id*). La parte derecha de la primera ecuación es igual a *fst* · *prod* (*f, id*).

Ahora podemos utilizar la definición recursiva *yajustarLista* para acercarnos al resultado que buscamos:

```
leyes4 = nuevosYAjustes ++ xajustes ++ asis ++ leyes1

? demostrar leyes4 "xajustarLista.prod(id,nil) = enlistar.fst"

  xajustarLista.prod(id,nil)
=     {def xajustarLista}
  yajustarLista.prod(enlistar,id).prod(id,nil)
=     {prod funtor}
  yajustarLista.prod(enlistar.id,id.nil)
=     {id elem neutro izq}
  yajustarLista.prod(enlistar.id,nil)
=     {id elem neutro der}
  gmachlist.prod(enlistar,nil)
=     {yajustarLista1}
  enlistar.fst

? demostrar leyes4 "xajustarLista.prod(id,cons) =
                                    yajustarLista.prod(xajustar,id).asi"

  xajustarLista.prod(id,cons)
=     {def xajustarLista}
  yajustarLista.prod(enlistar,id).prod(id,cons)
=     {prod funtor}
  yajustarLista.prod(enlistar.id,id.cons)
=     {id elem neutro izq}
  yajustarLista.prod(enlistar.id,cons)
=     {id elem neutro der}
  gmathlist.prod(enlistar,cons)
=     {yajustarLista2}
  yajustarLista.prod(yajustar,id).asi.prod(enlistar,id)
=     {asi auxiliar}
  yajustarLista.prod(yajustar,id).prod(prod(enlistar,id),id).asi
=     {id elem neutro izq}
  yajustarLista.prod(yajustar.prod(enlistar,id),id.id).asi
=     {id elem neutro izq}
  yajustarLista.prod(yajustar.prod(enlistar,id),id) asi
=     {def xajustar}
  yajustarLista.prod(xajustar,id).asi
```

La comparación de este último resultado con el que buscamos indica que necesitamos un hecho final, en concreto que que *yajustarLista* = *estr xajustarLista* · *pci*. Para demostrar esta ecuación, hemos de volver a las definiciones originales de *xajustarLista* y *yajustarLista*:

```
originales
 = map analizaLey [
    "xajustarLista: xajustarLista star union.pcd.prod(id,ajustarLista)",
    "yajustarLista: yajustarLista estr union.pc.prod(id,ajustarLista)"]

leyes5 = originales ++ leyes1

? demostrar leyes5 "yajustarLista estr xajustarLista.pci"

  yajustarLista
=     {def yajustarLista}
  estr union.pc.prod(id,ajustarLista)
=     {def pc}
  estr union.estr pcd.pci.prod(id,ajustarLista)
=     {pci después de id}
  estr union.estr pcd.map(prod(id,ajustarLista)).pci
=     {estr después de map}
  estr union.estr(pcd.prod(id,mathlist)).pci
=     {estr después de estr}
  estr(estr union.pcd.prod(id,ajustarLista)).pci
=     {def xajustarLista}
  estr xajustarLista.pci
```

12.5.4 Conclusiones

La conclusión positiva de este ejercicio es que realmente se puede obtener un calculador que ayude en la construcción de demostraciones detalladas. Sin embargo, todavía es necesaria la intervención humana para alimentar el proceso, establecer las leyes apropiadas, identificar las conjeturas auxiliares y controlar el orden en que se realizan los cómputos. La principal conclusión negativa es que algunas leyes han de ser incluidas en una forma más complicada de lo que sería deseable para evitar tener que invocar otras leyes a la vez de derecha a izquierda y de izquierda a derecha. A este respecto, las leyes funtoriales son especialmente problemáticas.

Quizá la conclusión más interesante del experimento es que expresar todas las leyes como ecuaciones entre funciones y utilizar la lógica ecuacional simple para demostrar resultados parece una estrategia viable.

Ejercicios

12.5.1 Utilizar el calculador para demostrar las leyes de *estr*.

12.5.2 Utilizar la definición de *ajustar* para obtener un programa para *yajustar* que no necesite *union*.

12.5.3 La demostración de que *ajustarLista* se puede expresar recursivamente de hecho muestra que *ajustarLista* = *foldr* (*une*, [[ ]]). Aplicar el primer teorema de dualidad para obtener otra expresión para *ajustarLista*.

12.5.4 Considérese la evaluación de *foldl* (⊕) *e*. Diremos que *z* es un cero por la izquierda de ⊕ si $z \oplus x = z$ para todo *x*. Demostrar que

$$foldl\ (\oplus)\ z\ xs \quad = \quad z$$

para todas las listas finitas *xs* siempre que *z* sea un cero por la izquierda de ⊕. ¿Cuántos pasos se necesitan para reducir la parte izquierda a *z*?

Sabiendo que

$$foldl\ (\oplus)\ e\ (xs +\!\!+ ys) \quad = \quad foldl\ (\oplus)\ (foldl\ (\oplus)\ e\ xs)\ ys$$

es obvio que, si *foldl* (⊕) *e xs* es un cero por la izquierda de ⊕, entonces

$$foldl\ (\oplus)\ e\ (xs +\!\!+ ys) \quad = \quad foldl\ (\oplus)\ xs$$

Dicho con palabras, se puede truncar la evaluación de *foldl* tan pronto como se encuentre un cero por la izquierda. Definir una función

$$hacerMientras \quad :: \quad (\beta \to bool) \to (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta$$

tal que si *p* es un test efectivo para detectar ceros por la izquierda de ⊕, entonces *hacerMientras p* (⊕) = *foldl* (⊕). Utilizar *hacerMientras* en la definición de *yajustarLista* conduce a un programa con la misma eficiencia que *ajustarLista*.

12.6 Notas del capítulo

El calculador original de Mike Spivey fue escrito en Orwell, un predecesor de Haskell; aunque no se ha documentado el programa en la literatura, Spivey (1990) contiene una discusión sobre ajuste y reescritura de expresiones. En Wilson (1993) se describe otro asistente interactivo con razonamiento ecuacional para Orwell llamado ERA. Los asistentes interactivos de demostraciones y los demostradores automáticos de teoremas constituyen una larga saga, existiendo docenas de sistemas disponibles para el usuario interesado. No intentaremos enumerarlos, tan sólo nos limitamos a dar algunas referencias en el Capítulo 4. Si se desea una referencia fácilmente accesible, consúltese Paulson (1996). Este texto describe una implementación en ML de un demostrador táctico de teoremas llamado Hal. El objetivo oculto de este capítulo ha sido mostrar las ventajas de expresar leyes y demostraciones como ecuaciones entre expresiones funcionales. Los fundamentos matemáticos de esta estrategia son materia de la teoría de categorías (véanse las referencias citadas en el Capítulo 2). Sin embargo, si se desea un verdadero método de especificación y de demostración, se necesita reemplazar las expresiones funcionales por expresiones *relacionales* y el razonamiento ecuacional por el razonamiento *inecuacional*. Para más detalles, véase Bird y de Moor (1997).

Apéndice A

Algunas funciones estándar

Se presenta a continuación un resumen en orden alfabético de las funciones más utilizadas en Haskell, junto con su definición. En algunos casos, esta definición difiere ligeramente de la contenida en el preludio estándar del lenguaje.

1. $(\cdot)$. Composición funcional:

$$\begin{array}{lcl} (\cdot) & :: & (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma) \\ (f \cdot g)\ x & = & f\ (g\ x) \end{array}$$

2. $(++)$. Concatenación de dos listas:

$$\begin{array}{lcl} (++) & :: & [\alpha] \to [\alpha] \to [\alpha] \\ [\,] ++ ys & = & ys \\ (x : xs) ++ ys & = & x : (xs ++ ys) \end{array}$$

3. $(\wedge)$. Conjunción lógica:

$$\begin{array}{lcl} (\wedge) & :: & Bool \to Bool \to Bool \\ True \wedge x & = & x \\ False \wedge x & = & False \end{array}$$

4. $(\vee)$. Disyunción lógica:

$$\begin{array}{lcl} (\vee) & :: & Bool \to Bool \to Bool \\ True \vee x & = & True \\ False \vee x & = & x \end{array}$$

5. $(!!)$. Indexación de una lista:

$$\begin{array}{lcl} (!!) & :: & [\alpha] \to Int \to \alpha \\ [\,] !!\ n & = & error\ \text{"(!!): index too large"} \\ (x : xs) !!\ 0 & = & x \\ (x : xs) !!\ (n+1) & = & xs !!\ n \end{array}$$

6. *and*. Devuelve la conjunción lógica de una lista de booleanos:

```
and  :: [Bool] → Bool
and  =  foldr (∧) True
```

7. *concat*. Concatena todas las listas de una lista de listas:

```
concat  :: [[α]] → [α]
concat  =  foldr (++) [ ]
```

8. *const*. Crea funciones que devuelven un valor constante:

```
const      :: α → β → α
const x y  =  x
```

9. *curry*. Convierte en currificada una función que no lo es:

```
curry      :: ((α, β) → γ) → (α → β → γ)
curry f x y  =  f (x, y)
```

10. *drop*. Borra los *n* primeros elementos de una lista:

```
drop            :: Int → [α] → [α]
drop 0 xs       =  xs
drop (n+1) [ ]  =  [ ]
drop (n+1) (x : xs)  =  drop n xs
```

11. *dropWhile*. Elimina el mayor prefijo de una lista tal que todos los elementos del mismo satisfacen un predicado dado:

```
dropWhile          :: (α → Bool) → [α] → [α]
dropWhile p [ ]    =  [ ]
dropWhile p (x : xs)  =  if p x then dropWhile p xs else x : xs
```

12. *enlistar*. Convierte un valor en una lista unitaria:

```
enlistar    :: α → [α]
enlistar x  =  [x]
```

13. *filter*. Retiene los elementos de una lista que satisfacen un predicado dado:

```
filter          :: (α → Bool) → [α] → [α]
filter p [ ]    =  [ ]
filter p (x:xs) =  if p x then x:filter p xs else filter p xs
```

14. *flip*. Permuta los dos argumentos de una función:

```
flip      :: (β → α → γ) → α → β → γ
flip f x y  =  f y x
```


15. *foldl*. Opera los elementos de una lista de izquierda a derecha:

```
foldl            :: (β → α → β) → β → [α] → β
foldl f e [ ]    =  e
foldl f e (x : xs)  =  strict (foldl f) (f e x) xs
```

16. *foldl1*. Versión de *foldl* para listas no vacías:

```
foldl1           :: (α → α → α) → [α] → α
foldl1 f [ ]     =  error "foldl1: empty list"
foldl1 f (x : xs)  =  foldl f x xs
```

17. *foldr*. Opera los elementos de una lista de derecha a izquierda:

```
foldr            :: (α → β → β) → β → [α] → β
foldr f e [ ]    =  e
foldr f e (x : xs)  =  f x (foldr f e xs)
```

18. *foldr1*. Versión de *foldr* para listas no vacías:

```
foldr1           :: (α → α → α) → [α] → α
foldr1 f [ ]     =  error "foldr1: empty list"
foldr1 f [x]     =  x
foldr1 f (x : xs)  =  f x (foldr1 f xs)
```

19. *fst*. Selecciona el primer elemento de una tupla:

```
fst        :: (α, β) → α
fst (x, y)  =  x
```

20. *head*. Selecciona el primer elemento de una lista no vacía:

```
head        :: [α] → α
head [ ]    =  error "head: empty list"
head (x : xs)  =  xs
```

21. *id*. Función identidad:

```
id    :: α → α
id x  =  x
```

22. *init*. Elimina el último elemento de una lista no vacía:

```
init            :: [α] → [α]
init [ ]        =  error "init: empty list"
init [x]        =  [ ]
init (x : y : xs)  =  x : init (y : xs)
```

23. *iterate*. Produce un lista infinita aplicando sucesivas veces una función a un valor:

```
iterate      :: (α → α) → α → [α]
iterate f x  =  x : iterate f (f x)
```

24. *last*. Selecciona el último elemento de una lista no vacía:

```
last            :: [α] → α
last [ ]        =  error "last: empty list"
last [x]        =  x
last (x : y : xs) =  last (y : xs)
```

25. *length*. Devuelve la longitud de una lista:

```
length          :: [α] → Int
length [ ]      =  0
length (x : xs) =  1 + length xs
```

26. *map*. Aplica una función a todos los elementos de una lista:

```
map             :: (α → β) → [α] → [β]
map f [ ]       =  [ ]
map f (x : xs)  =  f x : map f xs
```

27. *not*. Negación lógica:

```
not        :: Bool → Bool
not True   =  False
not False  =  True
```

28. *null*. Comprueba si una lista es o no vacía:

```
null        :: [α] → Bool
null [ ]    =  True
null (x : xs) =  False
```

29. *or*. Devuelve la disyunción lógica de una lista de booleanos:

```
or  :: [Bool] → Bool
or  =  foldr (∨) False
```

30. *par*. Aplica un par de funciones a un mismo argumento:

```
par        :: (α → β, α → γ) → α → (β,γ)
par (f, g) x  =  (f x, g x)
```

31. *partition*. Descompone una lista en dos según sus elementos satisfagan o no un cierto predicado (comparar con *span* y con *splitAt*):

```
partition          :: (α → Bool) → [α] → ([α],[α])
partition p [ ]    =  ([ ],[ ])
partition p (x : xs) =  if p x then (x : ys, zs) else (ys, x : zs)
                        where (ys, zs) = partition p xs
```


32. *prod*. Aplica un par de funciones a un par de valores:

```
prod       :: (α → γ, β → δ) → (α, β) → (γ, δ)
prod (f, g)  =  par (f · fst, g · snd)
```

33. *reverse*. Invierte una lista finita:

```
reverse  :: [α] → [α]
reverse  =  foldl (flip ( : )) [ ]
```

34. *scanl*. Aplica *foldl* a todos los prefijos de una lista:

```
scanl        :: (β → α → β) → β → [α] → [β]
scanl f e xs =  e : scanl' f e xs
                where scanl' f a [ ]       =  [ ]
                      scanl' f a (y : ys)  =  scanl f (f a y) ys
```

35. *scanl1*. Aplica *foldl1* a todos los prefijos no vacíos de una lista no vacía:

```
scanl1          :: (α → α → α) → [α] → [α]
scanl1 f [ ]    =  error "scanl1: empty list"
scanl1 f (x : xs) =  scanl f x xs
```

36. *scanr*. Aplica *foldr* a todos los sufijos de una lista:

```
scanr          :: (α → β → β) → β → [α] → [β]
scanr f e [ ]  =  [e]
scanr f e (x : xs) =  f x (head ys):ys
                      where ys = scanr f e xs
```

37. *scanr1*. Aplica *foldr1* a todos los sufijos no vacíos de una lista no vacía:

```
scanr1          :: (α → α → α) → [α] → [α]
scanr1 f [ ]    =  error "scanr1: empty list"
scanr1 f [x]    =  [x]
scanr1 f (x : y : xs) =  f x (head zs) : zs
                         where zs = scanr1 f (y : xs)
```

38. *span*. Divide una lista en dos, de modo que la lista izquierda sea el mayor prefijo cuyos elementos satisfacen un predicado dado:

```
span          :: (α → Bool) → [α] → ([α],[α])
span p [ ]    =  ([ ], [ ])
span p (x : xs) =  if p x then (x : ys, zs) else ([ ], x : xs)
                   where (ys, zs) = span p xs
```

39. *splitAt*. Divide una lista en dos según una longitud dada:

```
splitAt          :: Int → [α] → ([α],[α])
```

```
splitAt 0 xs          = ([ ], xs)
splitAt (n+1) [ ]     = ([ ], [ ])
splitAt (n+1) (x : xs) = (x : ys, zs)
                         where (ys, zs)=splitAt n xs
```

40. *snd*. Selecciona el segundo elemento de una tupla:

```
snd       :: (α, β) → β
snd (x, y) = y
```

41. *tail*. Elimina el primer elemento de una lista no vacía:

```
tail       :: [α] → [α]
tail [ ]    = error "tail: empty list"
tail (x : xs) = xs
```

42. *take*. Selecciona el prefijo de una longitud dada de una lista:

```
take          :: Int → [α] → [α]
take 0 xs      = [ ]
take (n+1) [ ]  = [ ]
take (n+1) (x : xs) = x : take n xs
```

43. *takeWhile*. Selecciona de una lista dada el mayor prefijo cuyos elementos satisfagan un predicado dado:

```
takeWhile          :: (α → Bool) → [α] → [α]
takeWhile p [ ]     = [ ]
takeWhile p (x : xs) = if p x then x : takeWhile p xs else [ ]
```

44. *uncurry*. Convierte en no currificada una función currificada:

```
uncurry       :: (α → β → γ) → (α, β) → γ
uncurry f xy   = f (fst xy) (snd xy)
```

45. *unitaria*. Comprueba si una lista es unitaria:

```
unitaria     :: [α] → Bool
unitaria xs   = not (null xs) ∧ null (tail xs)
```

46. *until*. Aplica una función a un valor el menor número posible de veces, hasta alcanzar un valor que satisface un predicado dado:

```
until       :: (α → Bool) → (α → α) → α → α
until p f x  = if p x then x else until p f (f x)
```

47. *unzip*. Desempareja una lista de pares y devuelve las dos listas resultantes:

```
unzip  :: [(α, β)] → ([α], [β])
unzip   = foldr f ([ ], [ ])
          where f (x, y) = prod ((x :), (y :))
```


48. *zip*. Empareja dos listas elemento a elemento y devuelve una lista de pares:

```
zip              :: [α] → [β] → [(α, β)]
zip [ ] ys        = [ ]
zip (x : xs) [ ]   = [ ]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
```

49. *zipp*. Versión no currificada de *zip*:

```
zipp  :: ([α], [β]) → [(α, β)]
zipp   = uncurry zip
```

Bibliografía

Adelson-Velski, G. M. y Landis, E. M. (1962). An algorithm for the organisation of information *Soviet Mathematics - Doklady*, *3*(5), 1259–1263. English translation in *Doklady Akademia Nauk SSSR*, 146, 263–266.

Aho, A., Sethi, R., y Ullman, J. (1986). *Compilers – Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., USA.

Andersson, A. (1993). Binary trees made simple In Dehne, F., Sack, J.-R., Santoro, N., y Whitesides, S.editores, *Workshop on Algorithm Design and Data Structures*, volumen 709 of *Lecture Notes in Computer Science*. Springer-Verlag.

Barr, M. y Wells, C. (1995). *Category Theory for Computing Science* (second edition). Prentice Hall International, Hemel Hempstead, Herts, UK.

Bellman, R. E. (1957). *Dynamic Programming.* Princeton University Press, Princeton, USA.

Ben-Ari, M. (1993). *Mathematical Logic for Computing Science.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Bentley, J. R. (1987). *Programming Pearls.* Addison-Wesley, Reading, Mass., USA.

Bird, R. y de Moor, O. (1997). *The Algebra of Programming.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Bird, R. S. (1980). Tabluation techniques for recursive programs *ACM Computing Surveys*, *12*(4), 403–417.

Bird, R. S. (1984). The promotion and accumulation strategies in functional programming *ACM Transactions on Programming Languages and Systems*, *6*(4), 487–504.

Bird, R. S. (1986). Transformational Programming and the paragraph problem *Science of Computer Programming*, *6*(2), 159–189.

Bird, R. S. (1989). Algebraic identities for program calculation *Computer Journal*, *32*(2), 122–126.

Bird, R. S. (1991). Meertens' Number In Jeuring, J.editor, *Lambert Meertens, Liber Amicorum*, págs. 4–10 Amsterdam, The Netherlands. CWI.

Bjerner, B. y Holmström, S. (1989). A compositional approach to time analysis of first order lazy functional programs In *Conference on Functional Programming Languages and Computer Architecture*, págs. 157–165. ACM.

Boole, G. (1847). *The Mathematical Analysis of Logic.* Cambridge. Reprinted, 1948, by Basil Blackwell, Oxford, UK.

Boyer, R. S. y Moore, J. S. (1979). *A Computational Logic.* Academic Press, London, UK.

Burge, W. H. (1975). *Recursive Programming Techniques.* Addison-Wesley, Reading, Mass., USA.

Burke, E. y Foxley, E. (1996). *Logic and its Applications.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Burstall, R. M. y Darlington, J. (1977). A transformational system for developing recursive programs *Journal of the ACM, 24*(1), 44–67.

Church, A. (1941). *The Calculi of Lamda Conversion*, volumen 6 of *Annals of Mathematical Studies.* Princeton University Press, Princeton, USA.

Clack, C., Myers, C., y Poon, E. (1995). *Programming with Miranda.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Cormen, T. H., Leiserson, C. E., y Rivest, R. L. (1990). *Introduction to Algorithms.* MIT Press, Cambridge, Mass., USA.

Davey, B. y Priestley, H. (1990). *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, UK.

de Moor, O. (1994). Categories, Relations and Dynamic Programming *Mathematical Structures in Computer Science, 4*, 33–69.

Dielissen, V. J. y Kaldewaij, A. (1995). A simple, efficient, and flexible implementation of flexible arrays In Möller, B.editor, *Third International Conference on the Mathematics of Program Construction*, volumen 947 of *Lecture Notes in Computer Science*, págs. 232–241. Springer-Verlag.

Fokker, J. (1995). Functional parsers In Jeuring, J. y Meijer, E.editores, *Advanced Functional Programming*, volumen 925 of *Lecture Notes in Computer Science*, págs. 1–23. Springer-Verlag.

Fokkinga, M. M. (1992). *Law and Order in Algorithmics.* Ph.D. thesis, Technical University of Twente, The Netherlands.

Gibbons, J. (1991). *Algebras for Tree Algorithms.* Ph.D. thesis, Programming Research Group, Oxford University, UK. Technical Monograph PRG-94.

Gödel, K. (1990). *Kurt Gödel, Collected Works*, volumen I. Oxford University Press, Oxford, UK. Edited by S. Feferman *et al.*

Gordon, A. (1993). *Functional Programming and Input/Output.* Ph.D. thesis, Computing Laboratory, Cambridge, UK. Published as a Distinguished Dissertation by Cambridge University Press.

Gordon, M. J., Milner, R., y Wadsworth, C. P. (1979). *Edinburgh LCF*, volumen 78 of *Lecture Notes in Computer Science.* Springer-Verlag.

Gordon, M. J. C. (1979). *The Denotational Description of Programming Languages, An Introduction.* Springer-Verlag.

Gordon, M. J. C. (1994). *Programming Language Theory and its Implementation.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Graham, R. L., Knuth, D. E., y Patashnik, O. (1990). *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, Reading, Mass., USA.

Gries, D. y Schneider, F. B. (1995). *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag.

Gunter, C. (1992). *Semantics of Programming Languages.* Foundations of Computing. MIT Press, Cambridge, Mass., USA.

Guttag, J. V. y Horning, J. J. (1978). The algebraic specification of abstract data types *Acta Informatica, 10*(1), 27–52.

Hall, C., Hammond, K., Partain, W., Jones, S. P., y Wadler, P. (1992). The Glasgow Haskell Compiler: a retrospective In *Proceedings of the 1992 Glasgow Workshop in Functional Programming*, Springer-Verlag Workshops in Computing, págs. 134–143. Springer-Verlag.

Harrison, R. (1989). *Abstract Data Types in Modula-2.* John Wiley and Sons, Chichester, UK.

Hoare, C. A. R. (1962). Quicksort *Computer Journal, 5*(1), 10–15.

Hofstadter, D. R. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid.* Harvester Press, Hassocks, Essex, UK.

Hoogerwoord, R. (1991). A logarithmic implementation of flexible arrays In Bird, R. S., Morgan, C. C., y Woodcock, J. P. C.editores, *Second International Conference on the Mathematics of Program Construction*, volumen 669 of *Lecture Notes in Computer Science*, págs. 191–207. Springer-Verlag.

Hu, T. C. (1982). *Combinatorial Algorithms.* Addison-Wesley, Reading, Mass., USA.

Hudak, P., Fasel, J., y Peterson, J. (1996). A gentle introduction to Haskell Informe técnico YALEU/DCS/RR-901, Yale University, USA.

Huffman, D. A. (1952). A method for the construction of minimum redundancy codes *Proceedings of the IRE, 40*, 1098-1101.

Hughes, R. J. M. (1984). *The Design and Implementation of Programming Languages.* Ph.D. thesis, Programming Research Group, Oxford University, Oxford, UK.

Hughes, R. J. M. (1995). The design of a pretty-printing library In Jeuring, J. y Meijer, E.editores, *Advanced Functional Programming*, volumen 925 of *Lecture Notes in Computer Science*, págs. 53-96. Springer-Verlag.

Hutton, G. (1992). Higher-order functions for parsing *Journal of Functional Programming, 2*, 323-343.

Hutton, G. y Meijer, E. (1996). Monadic parser combinators Draft manuscript.

ISO (1989). A Character Set for Western European languagesISO Standard 8859-1.

Jeuring, J. (1993). *Theories for Algorithm Calculation.* Ph.D. thesis, University of Utrecht, The Netherlands.

Jones, M. P. (1992). *Qualified Types: Theory and Practice.* Ph.D. thesis, Programming Research Group, Oxford University, Oxford, UK.

Jones, M. P. (1993). Release notes for Gofer 2.28 User manual for Gofer.

Jones, M. P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism *Journal of Functional Programming, 5*(1), 1-35.

Jones, S. P., Gordon, A., y Finne, S. (1996). Concurrent Haskell In *23rd ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida.* ACM.

Kaldewaij, A. (1990). *Programming: The Derivation of Algorithms.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Knuth, D. E. (1973a). *The Art of Computer Programming: Fundamental Algorithms* (second edition)., volumen 1. Addison-Wesley, Reading, Mass., USA.

Knuth, D. E. (1973b). *The Art of Computer Programming: Sorting and Searching*, volumen 3. Addison-Wesley, Reading, Mass., USA.

Knuth, D. E. (1981). *The Art of Computer Programming: Seminumerical Algorithms* (second edition)., volumen 2. Addison-Wesley, Reading, Mass., USA.

Knuth, D. E. y Plass, M. F. (1981). Breaking paragraphs into lines *Software, Practice and Experience, 11*, 1119-1184.

Launchbury, J. (1993). Lazy imperative programming In *Proceedings of the ACM SigPlan Workshop on State in Programming Languages.* Also University of Yale Research Report, YALEU/DCS/RR-968, New Haven, USA.

Liang, S., Hudak, P., y Jones, M. (1995). Monad transfomers and modular interpreters In *Proceedings of the 22nd ACM Conference on Principles of Programming Languages.*

Malcolm, G. R. (1990). *Algebraic Data Types and Program Transformation.* Ph.D. thesis, University of Groningen, Groningen, The Netherlands.

Martin, J. J. (1986). *Data Types and Data Structures.* Prentice Hall International, Hemel Hempstead, Herts. UK.

Martin, U. y Nipkow, T. (1990). Automating Squiggol In Broy, M. y Jones, C. B.editores, *Programming Concepts and Methods*, págs. 223-236 Amsterdam, The Netherlands. North-Holland.

Meertens, L. G. L. T. (1987). First steps towards the theory of Rose Trees Draft Report, CWI, Amsterdam, The Netherlands.

Meijer, E. (1992). *Calculating Compilers.* Ph.D. thesis, University of Nijmegen, The Netherlands.

Moggi, E. (1989). Computational lambda-calculus and monads In *IEEE Symposium on Logic in Computer Science* Asilomar, Calif., USA.

Moggi, E. (1991). Notions of computation and monads *Information and Control, 93*, 55-92.

Morgan, C. (1996). *Programming from Specifications* (Second edition). Prentice Hall International, Hemel Hempstead, Herts, UK.

Okasaki, C. (1995). Simple and efficient purely functional queues and deques *Journal of Functional Programming, 5*(4), 583-592.

Okasaki, C. (1996). *Purely Functional Data Structures.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA.

Paulson, L. C. (1983). Rewriting in Cambridge LCF *Science of Computer Programming, 3*(2), 119-149.

Paulson, L. C. (1996). *ML for the Working Programmer* (second edition). Cambridge University Press, Cambridge, UK.

Penrose, R. (1994). *Shadows of the Mind.* Oxford University Press, Oxford, UK.

Pettorossi, A. (1984). *Methodologies for transformations and memoing in applicative languages.* Ph.D. thesis, University of Edinburgh, Edinburgh, UK.

Peyton Jones, S. (1987). *The Implementation of Functional Programming Languages.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Peyton Jones, S. y Lester, D. (1991). *Implementing Functional Languages.* Prentice Hall International, Hemel Hempstead, Herts, UK.

Pierce, B. C. (1991). *Basic Category Theory for Computer Scientists.* Foundations of Computing. MIT Press, Cambridge, Mass., USA.

Pitts, A. y Dyber, P.editores. (1997). *Semantics and Logics of Computation.* Publications of the Newton Institute, Cambridge University Press, Cambridge, UK.

Runciman, C. y Röjemo, N. (1996). New dimensions in heap profiling *Journal of Functional Programming*, *6*(4), 587–620.

Sands, D. (1995). A naive time analysis and its theory of cost equivalence *Journal of Logic and Computation*, *5*(4), 495–541.

Schoenmakers, B. (1992). *Data Structures and Amortized Complexity in a Functional Setting*. Ph.D. thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

Scott, D. (1976). Data types as lattices *SIAM Journal of Computing*, *5*, 522–587.

Scott, D. (1982). Domains for denotational semantics In *International Colloquium on Automata, Languages and Programming*, volumen 140 of *Lecture Notes in Computer Science*, págs. 577–613. Springer-Verlag.

Sijtsma, B. (1988). *Verification and Derivation of Infinite-list Programs*. Ph.D. thesis, University of Groningen, The Netherlands.

Spivey, M. (1990). A functional theory of exceptions *Science of Computer Programming*, *14*, 25–42.

Stoy, J. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Mass., USA.

Thompson, S. (1995). *Miranda: The Craft of Functional Programming*. Addison-Wesley, London, UK.

Thompson, S. (1996). *Haskell: The Craft of Functional Programming*. Addison-Wesley, London, UK.

Wadler, P. (1984). *Listlessness is better than Laziness*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA.

Wadler, P. (1985). How to replace failure by a list of successes In *Second International Conference on Functional Programming Languages and Computer Architecture*,, volumen 201 of *Lecture Notes in Computer Science*. Springer-Verlag.

Wadler, P. (1987). Strictness analysis aids time analysis In *Proceedings of the 14th ACM Symposium on the Principles of Programming Languages*, págs. 307–313.

Wadler, P. (1989). Theorems for free! In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, págs. 347–359.

Wadler, P. (1990a). Comprehending monads In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, págs. 347–359. Also in *Mathematical Structures in Computer Science*, 2, 461–493, 1992.

Wadler, P. (1990b). Deforestation: transforming programs to eliminate trees *Theoretical Computer Science*, *73*, 231–248.

Wadler, P. (1995). Monads for functional programming In Jeuring, J. y Meijer, E.editores, *Advanced Functional Programming*, volumen 925 of *Lecture Notes in Computer Science*. Springer-Verlag.

Wadler, P. y Blott, S. (1989). How to make *ad hoc* polymorphism less *ad hoc* In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, págs. 60–76.

Wilson, S. P. (1993). Equational Reasoning Support for Orwell Informe técnico PRG Monographs, PRG-104, Oxford University Computing Laboratory, Oxford, UK.

Índice analítico

Símbolos

(·), 349
(++), 349
(∨), 349
(∧), 349

A

ajustarDer, 126
ajustarIzq, 126
ajuste de patrones, 27
análisis asintótico, 190
analizador sintáctico, 307
  de expresiones, 326
  factorizar, 311
and, 350
aplicar, 9
árbol
  altura de un, 155
  binario, 153
    de búsqueda, 160
    etiquetado, 160
    montículo, 164
    perfecto, 155
  casi perfecto, 242
  diestro, 242
  equilibrado, 229
  general, 183
  perfecto, 241
    por la derecha, 242
    por la izquierda, 241
  tamaño de un, 154
  zurdo, 236
argumentos, 9
asociación, 13
asociatividad, 13

B

booleano, 15
búsqueda
  binaria, 70
  lineal, 70

C

cadena, 34
cadenas, 45
categoría sintáctica, 314
clase
  declaración de concreción, 29
  declaración de una, 29
  definiciones por defecto, 30
  métodos de una, 29
  subclase de una, 30
cola de prioridad, 235
complemento con signo, 132
completitud de cadena, 255
composición funcional, 13
concat, 85, 350
concreción, 329
condiciones de naturalidad, 95
conexión de Galois, 234
conjunción, 28
const, 158, 350

constructoras, 35
constructoras ocultas, 235
contexto, 3, 160
coste amortizado, 220
cuadrado, 2–4
currificación, 10
curry, 11, 20, 350

D

declaración
de clase de tipos, 29
de concreción de una clase, 29
de tipos de datos, 27
recursiva, 51
definición
local, 17
recursiva, 16
deforestación, 207
deriving
cláusula, 36
desbordamiento de los enteros, 66
diferenciación finita, 207
disyunción, 28
divide y vencerás, 127
dropWhile, 350
drop, 89, 350

E

ecuaciones con guardas, 16
Either, 41
encaje, 320
enlistar, 350
entorno, 3
error, 17, 20
error
de sintaxis, 19
de tipos, 19
especificación, 22
especificación algebraica, 214
estado, 285
estilo
aplicativo, 10
no aplicativo, 10
punto a punto, 10
evaluación, 4
impaciente, 187
perezosa, 5, 187

F

fact, 16
filter, 95, 350
flip, 350
foldl1, 105, 351
foldl, 104, 351
foldn, 62
foldr1, 105, 351
foldr, 103, 351
fold
familia de funciones, 102, 206
leyes de, 109
forma normal, 4
forma canónica, 4
forma normal débil, 204
fromEnum, 35
fst, 37, 351
fuga de espacio, 209
función, 8
continua, 254
estricta, 8
monótona, 254
no estricta, 8
función estricta, 215
funtor, 94, 296
fusión, 111, 207
fold-concat, 112
fold-map, 112
fold-scan, 113
ley de, 63

G

generador
regla del, 96
guarda, 16
regla de la, 96
guión, 2

H

head, 88, 351
heapsort, 166

I

id, 351
implementación, 22
importación, 223
impresión estructurada, 317
inducción
estructural, 55
exhaustiva, 59
hipótesis de, 56
infinito, 55
init, 88, 351
invariante de la representación, 217
iterate, 352

L

last, 88, 352
length, 87, 352
ley del colapso, 293
ley del contable, 113
ley del salto de la rana, 295
librería, 3
listas, 79
concatenar dos, 82
indexación de, 92
intensionales, 96
parciales, 83
llenado
de párrafos, 148

M

magnitud con signo, 132
map, 93, 352
mayor suma de segmentos, 114
mcd, 69
método de la congruencia lineal, 268
métodos de una clase, 29
minimo, 2
módulo
referido a números, 67
mónada, 275, 278
composición de, 294
de escritura, 289, 298
de estado, 288, 298
de excepciones, 288, 298
identidad, 287
morfismo entre, 304
transformador de, 298

N

nodo
hoja, 153, 167
interno, 153, 167
not, 27, 352
null, 352
números
finitos, 55
infinitos, 54
de Meertens, 178
parciales, 55

O

operación
parcial, 53
total, 53
operador, 12
operadores
de igualdad, 29
de orden, 29
sobrecargados, 29
or, 352
orden
de aproximación, 252
parcial, 252
completo, 253
plano, 253
ordenación
algoritmos de, 126

P

par, 352
parámetro acumulador, 137, 170, 194, 209
partition, 352
precedencia, 9, 13
preludio, 3
principio de extensionalidad, 9
principio de optimalidad, 201
problemas de cierre, 260
prod, 353

programación monádica, 275
propiedad intensional, 10
puntero, 187

Q

quicksort, 161
quicksort, 208, 212

R

recolector de basura, 188
recorrido
    en anchura, 171
    en profundidad, 170
recursión a la izquierda, 314
redex
    más externo, 186
    más interno, 186
reducción, 4
    más externa, 186
    más interna, 186
reducción de grafos, 187
reescritura, 333, 335
regla de Horner, 104
renombramiento, 42
representación canónica
    de un racional, 68
resultados, 9
reverse, 85, 105, 353
rosadelfa, 167
    anchura de una, 167
    perfecta, 168

S

scanl1, 108, 353
scanl, 106, 353
scanr1, 108, 353
scanr, 107, 353
show, 46
secciones, 12
sesión, 2
signatura, 214
signo, 16
símbolo terminal, 314
simplificación, 4
síntesis de programas, 24, 60
snd, 37, 354
span, 127, 353
splitAt, 91, 353
strict, 204
subclase, 30
sustitución, 320

T

tabulación, 200
tail, 88, 354
take, 89, 354
takeWhile, 354
techo de un número decimal, 156
teorema de fusión
    para *foldl*, 111
    para *foldr*, 111
teoremas de dualidad
    primero, 109
    segundo, 109
    tercero, 111
teoría de dominios, 252
teoría de categorías, 49
tipo abstracto de datos, 213
tipos, 2, 19
    clases de, 21
    constructoras de, 35
    disciplina de, 19
    polimórficos, 20
    signaturas de, 2
    variables de, 20
toEnum, 35
transformador de estados, 285
tratamiento de excepciones, 284
tuplamiento, 198
    técnica de, 148
tuplas, 37
    operadores de comparación, 40
    orden lexicográfico, 40
twice, 11

U

uncurry, 11, 15, 354
unidad léxica, 313
unitaria, 354
unlines, 126

until, 354
unzip, 101, 354

V

valor, 6
vectores flexibles, 238
vínculo, 3

Z

zipp, 355
zip, 99, 355

Richard Bird

Introducción a la Programación Funcional con Haskell

Segunda edición

Después del éxito de la primera edición, esta segunda de **Introducción a la Programación Funcional con Haskell** ha sido revisada y actualizada exhaustivamente, proporcionando unos fundamentos sólidos y completos sobre las técnicas y los principios que inspiran la programación funcional.

Esta segunda edición utiliza el popular lenguaje Haskell para presentar los algoritmos funcionales. Se han incluido nuevos capítulos sobre optimización de programas, tipos abstractos de datos en un marco funcional y programación monádica. Hay, además, casos de estudio completamente nuevos y muchos ejercicios adicionales.

Al igual que en la primera edición, se hace hincapié en las técnicas de razonamiento sobre programas funcionales y en la derivación sistemática de los mismos a partir de sus especificaciones.

El libro es autocontenido, ya que no supone conocimientos previos de programación y es adecuado como texto introductorio para estudiantes de primer o segundo año.

Richard Bird es *Professor of Computation* en la Universidad de Oxford y miembro del Lincoln College de Oxford.

ISBN 84-8322-176-4

9 788483 221761

Pearson Educación