

Conceptos, Técnicas y Modelos de Programación

Draft

Draft

Conceptos, Técnicas y Modelos de Programación

por
Peter Van Roy
Seif Haridi

traducción
Juan Francisco Díaz Frias

The MIT Press
Cambridge, Massachusetts
London, England

Draft

©2004 Massachusetts Institute of Technology

Derechos reservados. Ninguna parte de este libro puede ser reproducida en ninguna forma, por medios electrónicos ni mecánicos (incluyendo medios para fotocopiar, grabar, o almacenar y recuperar información) sin permiso escrito del editorial.

Este libro se editó con L^AT_EX 2_ε a partir de la versión en inglés de los autores

Library of Congress Cataloging-in-Publication Data

Van Roy, Peter.

Concepts, techniques, and models of computer programming / Peter Van Roy, Seif Haridi
p. cm.

Includes bibliographical references and index.

ISBN 0-262-22069-5

1. Computer programming. I. Haridi, Seif. II. Title.

QA76.6.V36 2004

005.1—dc22

2003065140

Tabla de Contenido Resumida

Prefacio	xii
Corriendo los programas de ejemplo	xxix
1. Introducción a los conceptos de programación	1
I. MODELOS GENERALES DE COMPUTACIÓN	29
2. El modelo de computación declarativa	31
3. Técnicas de programación declarativa	123
4. Concurrencia declarativa	257
5. Concurrencia por paso de mensajes	379
6. Estado explícito	443
7. Programación orientada a objetos	533
8. Concurrencia con estado compartido	621
II. APÉNDICES	679
A. Ambiente de Desarrollo del Sistema Mozart	681
B. Tipos de Datos Básicos	685
C. Sintaxis del Lenguaje	701
D. Modelo General de Computación	711
Bibliografía	721
Índice alfabético	729

Draft^{vi}

Contenido

Tabla de Contenido

Prefacio	xii
Corriendo los programas de ejemplo	xxix
1. Introducción a los conceptos de programación	1
1.1. Una calculadora	1
1.2. Variables	2
1.3. Funciones	2
1.4. Listas	4
1.5. Funciones sobre listas	7
1.6. Corrección	9
1.7. Complejidad	11
1.8. Evaluación perezosa	12
1.9. Programación de alto orden	14
1.10. Concurrencia	16
1.11. Flujo de datos	16
1.12. Estado explícito	17
1.13. Objetos	18
1.14. Clases	19
1.15. No-determinismo y tiempo	21
1.16. Atomicidad	23
1.17. ¿Para donde vamos?	24
1.18. Ejercicios	24
I. MODELOS GENERALES DE COMPUTACIÓN	29
2. El modelo de computación declarativa	31
2.1. Definición de lenguajes de programación prácticos	33
2.2. El almacén de asignación única	45
2.3. El lenguaje núcleo	52
2.4. La semántica del lenguaje núcleo	60
2.5. Administración de la memoria	78
2.6. Del lenguaje núcleo a un lenguaje práctico	86
2.7. Excepciones	97

Contenido

2.8.	Temas avanzados	105
2.9.	Ejercicios	117
3.	Técnicas de programación declarativa	123
3.1.	Qué es declaratividad?	127
3.2.	Computación iterativa	130
3.3.	Computación recursiva	137
3.4.	Programando recursivamente	140
3.5.	Eficiencia en tiempo y en espacio	183
3.6.	Programación de alto orden	194
3.7.	Tipos abstractos de datos	214
3.8.	Requerimientos no declarativos	231
3.9.	Diseño de programas en pequeño	239
3.10.	Ejercicios	253
4.	Concurrencia declarativa	257
4.1.	El modelo concurrente dirigido por los datos	259
4.2.	Técnicas básicas de programación de hilos	272
4.3.	Flujos	282
4.4.	Utilizando el modelo concurrente declarativo directamente	300
4.5.	Ejecución perezosa	306
4.6.	Programación en tiempo real no estricto	334
4.7.	El lenguaje Haskell	339
4.8.	Limitaciones y extensiones de la programación declarativa	344
4.9.	Temas avanzados	358
4.10.	Notas históricas	370
4.11.	Ejercicios	371
5.	Concurrencia por paso de mensajes	379
5.1.	El modelo concurrente por paso de mensajes	381
5.2.	Objetos puerto	384
5.3.	Protocolos sencillos de mensajes	388
5.4.	Diseño de programas con concurrencia	397
5.5.	Sistema de control de ascensores	402
5.6.	Usando directamente el modelo por paso de mensajes	413
5.7.	El lenguaje Erlang	423
5.8.	Tema avanzado	432
5.9.	Ejercicios	437
6.	Estado explícito	443
6.1.	¿Qué es estado?	446
6.2.	Estado y construcción de sistemas	448
6.3.	El modelo declarativo con estado explícito	452
6.4.	Abstracción de datos	458
6.5.	Colecciones con estado	475

6.6.	Razonando con estado	481
6.7.	Diseño de programas en grande	491
6.8.	Casos de estudio	506
6.9.	Temas avanzados	523
6.10.	Ejercicios	527
7.	Programación orientada a objetos	533
7.1.	Herencia	535
7.2.	Clases como abstracciones de datos completas	537
7.3.	Clases como abstracciones de datos incrementales	547
7.4.	Programación con herencia	565
7.5.	Relación con otros modelos de computación	585
7.6.	Implementando el sistema de objetos	594
7.7.	El lenguaje Java (parte secuencial)	601
7.8.	Objetos activos	607
7.9.	Ejercicios	618
8.	Concurrencia con estado compartido	621
8.1.	El modelo concurrente con estado compartido	624
8.2.	Programación con concurrencia	625
8.3.	Candados	636
8.4.	Monitores	645
8.5.	Transacciones	654
8.6.	El lenguaje Java (parte concurrente)	671
8.7.	Ejercicios	673
II.	APÉNDICES	679
A.	Ambiente de Desarrollo del Sistema Mozart	681
A.1.	Interfaz interactiva	681
A.2.	Interfaz de la línea de comandos	683
B.	Tipos de Datos Básicos	685
B.1.	Números (enteros, flotantes, y caracteres)	685
B.2.	Literales (átomos y nombres)	690
B.3.	Registros y tuplas	691
B.4.	Pedazos (registros limitados)	695
B.5.	Listas	695
B.6.	Cadenas de caracteres	697
B.7.	Cadenas virtuales de caracteres	699
C.	Sintaxis del Lenguaje	701
C.1.	Declaraciones interactivas	702
C.2.	Declaraciones y expresiones	704

C.3.	No terminales para las declaraciones y las expresiones	704
C.4.	Operadores	705
C.5.	Palabras reservadas	707
C.6.	Sintaxis léxica	708
D. Modelo General de Computación		711
D.1.	Principio de extensión creativa	712
D.2.	Lenguaje núcleo	714
D.3.	Conceptos	714
D.4.	Formas diferentes de estado	718
D.5.	Otros conceptos	718
D.6.	Diseño de lenguajes por capas	719
Bibliografía		721
Índice alfabético		729

Prefacio

Seis sabios ciegos se reunieron a discutir su experiencia después de mostrarles un elefante. “Es maravilloso,” dijo el primero, “un elefante es como una cuerda: delgado y flexible.” “No, no, de ninguna manera,” dijo el segundo, “Un elefante es como un árbol: fuertemente plantado sobre el suelo.” “Maravilloso,” dijo el tercero, “un elefante es como una pared.” “Increíble,” dijo el cuarto, “un elefante es un tubo lleno de agua.” “Es una bestia extraña en evolución,” dijo el quinto. “Extraña en realidad,” dijo el sexto, “pero debe tener una armonía subyacente. Investiguemos el tema un poco más.”

— Adaptación libre de un fábula tradicional de India.

Un lenguaje de programación es como un lenguaje natural del hombre, en el sentido que favorece ciertas metáforas, imágenes y formas de pensar.

— Adaptación libre de *Mindstorms: Children, Computers, and Powerful Ideas*, Seymour Papert (1980)

Un enfoque para estudiar la programación es estudiar los lenguajes de programación. Pero el número de lenguajes que existen es tan grande, que es poco práctico estudiarlos todos. ¿Cómo podemos abordar esa inmensidad? Podríamos tomar unos pocos lenguajes que sean representativos de diferentes paradigmas de programación. Pero esto no lleva a un entendimiento profundo de la programación como una disciplina unificada. Este libro utiliza otro enfoque.

Nosotros nos enfocamos en conceptos de programación y en las técnicas para usarlos, y no en los lenguajes de programación. Los conceptos son organizados en términos de modelos de computación. Un modelo de computación es un sistema formal que define cómo se realizan las computaciones. Hay muchas maneras de definir modelos de computación. Como este libro pretende ser práctico, es importante que el modelo de computación sea directamente útil para el programador. Por ello lo definimos en términos de conceptos importantes para los programadores: tipos de datos, operaciones, y un lenguaje de programación. El término *modelo de computación* precisa la noción imprecisa de “paradigma de programación.” El resto del libro trata sobre modelos de computación y no sobre paradigmas de programación. Algunas veces usamos el término “modelo de programación.” Con este nos referimos a lo que el programador necesita: las técnicas de programación y los principios de diseño que el modelo de computación provee.

Cada modelo de computación tiene su propio conjunto de técnicas para programar y razonar sobre los programas. El número de modelos de computación diferentes, que se conocen y se saben útiles, es mucho menor que el número de lenguajes de

Prefacio

programación. Este libro cubre muchos modelos bien conocidos así como algunos modelos menos conocidos. El principal criterio para presentar un modelo es que sea útil en la práctica.

Cada modelo de computación se basa en un lenguaje sencillo fundamental llamado el lenguaje núcleo del modelo. Estos lenguajes se introducen de manera progresiva, añadiendo conceptos uno por uno. Esto nos permite mostrar la profunda relación entre los diferentes modelos. Frecuentemente, añadir un solo concepto nuevo hace una gran diferencia en términos de programación. Por ejemplo, añadir asignación destructiva (estado explícito) a la programación funcional nos permite hacer programación orientada a objetos.

¿Cuándo debemos añadir un concepto a un modelo y cuál concepto debemos añadir? Nosotros abordamos estas preguntas muchas veces. El principal criterio es el *principio de extensión creativa*. De manera general, se añade un nuevo concepto cuando los programas se vuelven complicados por razones técnicas no relacionadas con el problema que se está resolviendo. Añadir un concepto al lenguaje núcleo puede conservar la sencillez de los programas, si el concepto se escoge cuidadosamente. Esto se explica después en el apéndice D. Este principio fundamenta la progresión de los lenguajes núcleo presentados en el libro.

Una propiedad elegante del enfoque de lenguajes núcleo es que nos permite usar al mismo tiempo diferentes modelos en el mismo programa. Esto se suele llamar programación multiparadigma. Esto es algo bastante natural, pues significa simplemente usar los conceptos adecuados para el problema, independientemente del modelo de computación del que proviene el concepto. La programación multiparadigma es una idea vieja; por ejemplo, los diseñadores de Lisp y Scheme han defendido por largo tiempo una visión similar. Sin embargo, este libro la aplica en una forma más amplia y profunda de lo que se ha hecho previamente.

Desde la perspectiva estratégica de los modelos de computación, el libro también presenta nuevas luces sobre problemas importantes en informática. Nosotros presentamos tres de esas áreas, específicamente diseño de interfaces gráficas de usuario, programación distribuida robusta, y programación por restricciones. También mostramos cómo resolver algunos de los problemas de estas áreas por medio del uso juicioso y combinado de diversos modelos de computación.

Lenguajes mencionados

En el libro mencionamos muchos lenguajes de programación y los relacionamos con modelos particulares de computación. Por ejemplo, Java y Smalltalk están basados en un modelo orientado a objetos. Haskell y Standard ML están basados en un modelo funcional. Prolog y Mercury están basados en un modelo lógico. No todos los lenguajes de programación interesantes pueden ser clasificados así. Otros lenguajes los mencionamos por sus propios méritos. Por ejemplo, Lisp y Scheme fueron los pioneros de muchos de los conceptos que se presentan aquí. Erlang es funcional, inherentemente concurrente, y soporta programación distribuida tolerante a fallos.

Escogimos cuatro lenguajes como representantes de modelos de computación im-

portantes: Erlang, Haskell, Java, y Prolog. Identificamos el modelo de computación de cada uno de estos lenguajes en términos del marco uniforme presentado en el libro. Para mayor información sobre estos lenguajes, referimos los lectores a otros libros. Debido a las limitaciones de espacio, no podemos mencionar todos los lenguajes de programación interesantes. La omisión de un lenguaje en particular no implica ninguna clase de juicio de valor sobre el mismo.

Objetivos del libro

Enseñanza de la programación

El principal objetivo del libro es enseñar la programación como una disciplina unificada con fundamento científico útil para quien practica la programación. Miremos más de cerca lo que esto significa.

¿Qué es la programación?

Definimos *la programación*, como una actividad general del hombre, que significa la acción de extender o cambiar la funcionalidad de un sistema. La programación es una actividad de amplio espectro realizada tanto por no especialistas (e.g., consumidores que cambian la configuración de su reloj de alarma o de su teléfono celular) como por especialistas (programadores de computadores, la audiencia de este libro).

Este libro se enfoca en la construcción de sistemas de *software*. En este contexto, la programación es la etapa entre la especificación de un sistema y el programa ejecutable que lo implementa. La etapa consiste en diseñar la arquitectura y las abstracciones del programa y codificarlas en un lenguaje de programación. Esta visión es amplia, tal vez más amplia que la connotación usual ligada a la palabra “programación.” Cubre tanto la programación “en pequeño” como la programación “en grande.” Cubre tanto asuntos arquitecturales (independientes del lenguaje) como asuntos de codificación (dependientes del lenguaje). Está basada en conceptos y su uso, más que en algún lenguaje de programación en particular. Encontramos que esta visión general es natural para enseñar la programación. No está influenciada por las limitaciones de algún lenguaje de programación o de una metodología de diseño en particular. Cuando se usa en una situación específica, la visión general se adapta a las herramientas usadas, teniendo en cuenta sus capacidades y limitaciones.

Tanto ciencia como tecnología

La programación, tal como fue definida anteriormente, consta de dos partes esenciales: la tecnología y su fundamentación científica. La tecnología consiste de herramientas, técnicas prácticas, y estándares, que nos permiten realizar la programación. La ciencia consiste de una teoría amplia y profunda, con poder predictivo, que nos permite entender la programación. Idealmente, la ciencia debe explicar la tecnología de la forma más directa y útil posible.

Si alguna de estas partes no es tenida en cuenta, no podríamos decir que se está programando. Sin la tecnología, haríamos matemáticas puras. Sin la ciencia, haríamos algo artesanal sin un entendimiento profundo de lo realizado. Por lo tanto, enseñar correctamente la programación significa enseñar tanto la

tecnología (herramientas actuales) como la ciencia (conceptos fundamentales). El conocimiento de las herramientas prepara al estudiante para el presente. Por su parte, el conocimiento de los conceptos prepara al estudiante para futuros desarrollos.

Más que una artesanía

A pesar de muchos esfuerzos por introducir un fundamento científico a la programación, ésta es casi siempre enseñada como una artesanía. Usualmente se enseña en el contexto de un (o unos pocos) lenguaje(s) de programación (e.g., Java, complementado con Haskell, Scheme, o Prolog). Los accidentes históricos de los lenguajes particulares escogidos se entrelazan junto con los conceptos fundamentales de manera tan cercana que no pueden ser separados. Existe una confusión entre herramientas y conceptos. Aún más, se han desarrollado diferentes escuelas de pensamiento, basadas en diferentes maneras de ver la programación, llamadas “paradigmas”: orientado a objetos, lógico, funcional, entre otros. Cada escuela de pensamiento tiene su propia ciencia. La unidad de la programación como una sola disciplina se ha perdido.

Enseñar la programación en esta forma es como tener escuelas separadas para la construcción de puentes: una escuela enseña cómo construir puentes de madera mientras otra escuela enseña cómo construirlos de hierro. Los egresados de cualquiera de estas escuelas considerarían la restricción de madera o hierro como fundamental y no pensarían en usar estos materiales juntos.

El resultado es que los programas adolecen de un diseño pobre. Nosotros presentamos un ejemplo basado en Java, pero el problema se presenta en todos los lenguajes en algún grado. La concurrencia en Java es compleja de usar y costosa en recursos computacionales. Debido a estas dificultades, los programadores que aprendieron a serlo usando Java pueden concluir que la concurrencia es un concepto fundamentalmente complejo y costoso. Las especificaciones de los programas se diseñan alrededor de las dificultades, muchas veces de manera retorcida. Pero estas dificultades no son para nada fundamentales. Hay formas de concurrencia bastante útiles en las que la programación es tan sencilla como la programación secuencial (e.g., la programación por flujos¹ ejemplificada por las tuberías Unix²). Además, es posible implementar los hilos, la unidad básica de la concurrencia, casi tan baratos como las invocaciones a procedimientos. Si al programador se le enseña la concurrencia en la forma correcta, entonces él o ella serían capaces de especificar y programar sistemas sin restricciones de concurrencia (incluyendo versiones mejoradas de Java).

1. Nota del traductor: *stream programming*, en inglés.

2. Nota del traductor: *Unix pipes*, en inglés.

El enfoque del lenguaje núcleo

En los lenguajes de programación en la práctica se escriben programas de millones de líneas de código. Para ello, estos lenguajes proveen una sintaxis y un conjunto de abstracciones amplios. ¿Cómo separar los conceptos fundamentales de los lenguajes, sobre los cuales recae su éxito, de sus accidentes históricos? El enfoque del lenguaje núcleo muestra una forma de hacerlo. En este enfoque, un lenguaje práctico es traducido en un lenguaje núcleo consistente de un pequeño número de elementos significativos para el programador. El conjunto de abstracciones y la sintaxis son codificados en el lenguaje núcleo. De esta manera tanto el programador como el estudiante tienen una visión clara de lo que el lenguaje hace. El lenguaje núcleo cuenta con una semántica formal, sencilla, que permite razonar sobre la corrección y complejidad de los programas. De esta manera se proporciona una fundamentación sólida a la intuición del programador y a las técnicas de programación construidas sobre ésta.

Una amplia variedad de lenguajes y paradigmas de programación se pueden modelar a partir de un conjunto de lenguajes núcleo estrechamente relacionados. De allí que el enfoque del lenguaje núcleo es realmente un estudio de la programación, independiente del lenguaje. Como cualquier lenguaje se traduce en un lenguaje núcleo que a su vez es un subconjunto de un lenguaje núcleo más completo, entonces se recupera la unidad subyacente a la programación.

Reducir un fenómeno complejo a sus elementos primitivos es una característica del método científico. Este es un enfoque exitoso utilizado en todas las ciencias exactas. Provee un profundo entendimiento con poder predictivo. Por ejemplo, la ciencia de las estructuras permite diseñar todos los puentes (sean hechos de madera, hierro, ambos o cualquier otro material) y predecir su comportamiento en términos de conceptos simples como fuerza, energía, tensión, y esfuerzo y de las leyes que ellos obedecen [57].

Comparación con otros enfoques

Comparemos el enfoque del lenguaje núcleo con otras tres formas de dotar la programación de una base científica general:

- Un cálculo, como el cálculo λ o el cálculo π , reduce la programación a un número minimal de elementos. Los elementos se escogen para simplificar el análisis matemático, no para ayudar la intuición del programador. Esto es muy útil para los teóricos, pero no lo es tanto para los programadores en la práctica. Los cálculos son muy útiles para estudiar las propiedades fundamentales y los límites de la programación de un computador, pero no para escribir o razonar sobre aplicaciones en general.
- Una máquina virtual define un lenguaje en términos de una implementación sobre una máquina ideal. Una máquina virtual posee una especie de semántica operacional, con conceptos cercanos al *hardware*. Esto es útil para diseñar computadores,

implementar lenguajes, o hacer simulaciones. No es útil para razonar sobre programas y sus abstracciones.

- Un lenguaje multiparadigma es un lenguaje que abarca diversos paradigmas de programación. Por ejemplo, Scheme es a la vez funcional e imperativo [36], y Leda tiene elementos de los paradigmas funcional, orientado a objetos, y lógico [27]. La utilidad de un lenguaje multiparadigma depende de cuán bien integrados están los diferentes paradigmas.

El enfoque del lenguaje núcleo combina características de todos los enfoques anteriores. Un lenguaje núcleo bien diseñado cubre un amplio rango de conceptos, tal como lo hace un lenguaje multiparadigma bien diseñado. Si los conceptos son independientes, entonces se puede dotar al lenguaje núcleo de una semántica formal, sencilla, como en un cálculo. Finalmente, la semántica formal puede ser una máquina virtual en un alto nivel de abstracción. Esto facilita a los programadores razonar sobre los programas.

Diseño de abstracciones

El segundo objetivo del libro es enseñar cómo diseñar abstracciones en la programación. La tarea más difícil para los programadores, y también la más provechosa, no es escribir programas sino diseñar abstracciones. Programar un computador es ante todo diseñar y usar abstracciones para conseguir nuevos objetivos. De manera muy general, definimos una *abstracción* como una herramienta o mecanismo que resuelve un problema particular. Normalmente, la misma abstracción puede ser usada para resolver varios problemas diferentes. Esta versatilidad es una de las propiedades más importantes de las abstracciones.

Las abstracciones están tan profundamente integradas a nuestro diario vivir, que frecuentemente nos olvidamos de ellas. Algunas abstracciones típicas son los libros, las sillas, los destornilladores, y los automóviles.³ Las abstracciones se pueden clasificar en una jerarquía dependiendo de cuán especializadas son (e.g., “lápiz” es más especializado que “instrumento de escritura,” pero ambos son abstracciones).

Las abstracciones son particularmente numerosas dentro de los sistemas de computadores. Los computadores modernos son sistemas altamente complejos consistentes de *hardware*, sistema operativo, *middleware*, y capas de aplicación, cada uno de los cuales está basado en el trabajo de miles de personas durante varias décadas. Estos sistemas contienen un gigantesco número de abstracciones que trabajan juntas de una manera altamente organizada.

Diseñar abstracciones no siempre es fácil. Puede ser un proceso largo y penoso, en el cual se ensayan, se descartan y se mejoran diferentes enfoques. Pero la recompensa es muy grata. No es una exageración decir que la civilización está construida sobre

3. También lo son los lápices, las tuercas y tornillos, los cables, los transistores, las corporaciones, las canciones, y las ecuaciones diferenciales. ¡No tienen que ser entidades materiales!

Prefacio

abstracciones exitosas [125]. Nuevas abstracciones se diseñan cada día. Algunas abstracciones antiguas, como la rueda y el arco, permanecen aún entre nosotros. Algunas abstracciones modernas, como el teléfono celular, rápidamente se vuelven parte de nuestra vida cotidiana.

Nosotros usamos el enfoque descrito a continuación para lograr el segundo objetivo. Empezamos con conceptos de programación, los cuales son la materia prima para construir abstracciones. Introducimos la mayoría de los conceptos relevantes conocidos hoy en día, en particular alcance léxico, programación de alto orden, composicionalidad, encapsulación, cocurrencia, excepciones, ejecución perezosa, seguridad, estado explícito, herencia, y escogencia no-determinística. Presentamos las técnicas para construir abstracciones para cada concepto. Presentamos bastantes ejemplos de abstracciones secuenciales, concurrentes y distribuidas. Presentamos algunas leyes generales para construir abstracciones. Muchas de estas leyes tienen su contraparte en otras ciencias aplicadas, de tal manera que libros como [51], [57], y [63] pueden servir de inspiración a los programadores.

Características principales

Enfoque pedagógico

Hay dos enfoques complementarios para enseñar la programación como una disciplina rigurosa:

- El enfoque basado en computación presenta la programación como una manera de definir ejecuciones sobre máquinas. Este enfoque afianza la intuición del estudiante en el mundo real por medio de ejecuciones actuales sobre sistemas reales. Este enfoque es especialmente efectivo con un sistema interactivo: el estudiante puede crear fragmentos de programas e inmediatamente ver como se comportan. La reducción del tiempo entre pensar “qué pasa si” y ver el resultado es de gran ayuda para entender. La precisión no se sacrifica, puesto que la semántica formal de un programa puede ser definida en términos de una máquina abstracta.
- El enfoque basado en la lógica presenta la programación como una rama de la lógica matemática. La lógica no versa sobre la ejecución sino sobre propiedades de los programas, lo cual es un nivel de abstracción más alto. Los programas son construcciones matemáticas que obedecen leyes lógicas. La semántica formal de un programa se define en términos de una lógica matemática. El razonamiento se realiza a través de afirmaciones lógicas. El enfoque basado en la lógica es más difícil de comprender para los estudiantes si bien es esencial para definir especificaciones precisas de qué hacen los programas.

Tal como el libro *Structure and Interpretation of Computer Programs* [1, 2], nuestro libro usa el enfoque basado en computación en su mayor parte. Los conceptos son ilustrados con fragmentos de programas que pueden ser ejecutados interactivamente

sobre un paquete de *software* que acompaña el libro, *the Mozart Programming System* [120]. Los programas se construyen con un enfoque de construcción por bloques, utilizando abstracciones de bajo nivel para construir las de más alto nivel. Una pequeña dosis de razonamiento lógico se introduce en los últimos capítulos, e.g., para definir especificaciones y para usar invariantes para razonar sobre programas con estado.

Formalismo utilizado

Este libro utiliza un único formalismo para presentar todos los modelos de computación y los programas, a saber, el lenguaje Oz y su modelo de computación. Para ser precisos, los modelos de computación del libro son todos subconjuntos de Oz cuidadosamente escogidos. ¿Por Qué escogimos Oz? La principal razón es que soporta bien el enfoque del lenguaje núcleo. Otra razón es la existencia de *the Mozart Programming System*.

Panorama de los modelos de computación

Este libro presenta una mirada global y amplia de muchos de los modelos de computación más útiles. Los modelos se diseñan no solo pensando en su simplicidad formal (aunque esto es importante), sino sobre la base de cómo puede el programador expresarse y razonar, él mismo, dentro del modelo. Existen muchos modelos de computación prácticos, con diferentes niveles de expresividad, diferentes técnicas de programación, y diferentes formas de razonar sobre ellos. Encontramos que cada modelo tiene sus dominios de aplicación. Este libro explica muchos de estos modelos, cómo se relacionan, cómo programar en ellos, y cómo combinarlos para sacarles el mayor provecho.

Más no es mejor (o peor), solo es diferente

Todos los modelos de computación tienen su lugar. No es cierto que los modelos con más conceptos sean mejores o peores. Un nuevo concepto es como una espada de doble filo. Añadir un concepto a un modelo de computación introduce nuevas formas de expresión, logrando que algunos programas sean más sencillos, pero también se vuelve más difícil razonar sobre los programas. Por ejemplo, añadiendo estado explícito (variables mutables) al modelo de programación funcional, podemos expresar el rango completo de técnicas de programación orientada a objetos. Sin embargo, razonar sobre programas orientados a objetos es más difícil que hacerlo sobre programas funcionales. La programación funcional calcula valores usando funciones matemáticas. Ni los valores ni las funciones cambian en el tiempo. El estado explícito es una forma de modelar cosas que cambian en el tiempo: provee un contenedor cuyo contenido puede ser actualizado. La gran potencia de este concepto hace más difícil razonar sobre él.

*Prefacio**La importancia de reunir diferentes modelos*

Cada modelo de computación fue diseñado originalmente para ser usado aisladamente. Puede entonces parecer una aberración el usarlos juntos en un mismo programa. En nuestro concepto, este no es de ninguna manera el caso, puesto que los modelos no son solamente bloques monolíticos sin nada en común. Por el contrario, los modelos de computación tienen mucho en común; por ejemplo, las diferencias entre los modelos declarativo e imperativo (y entre los modelos concurrente y secuencial) son muy pequeñas comparadas con lo que tienen en común. Por ello, es fácil usar varios modelos a la vez.

Pero aun cuando esto es técnicamente posible, ¿por qué querría uno usar varios modelos en un mismo programa? La respuesta profunda a esta pregunta es simple: porque uno no programa con modelos sino con conceptos de programación y mecanismos para combinarlos. Dependiendo de cuáles modelos se utilizan es posible considerar que se está programando en un modelo en particular. El modelo aparece como una clase de epifenómeno. Ciertas cosas son más fáciles, otras cosas son más difíciles, y razonar sobre el programa se hace de una manera particular. Es muy natural que un programa bien escrito utilice diferentes modelos. En este punto esta respuesta puede parecer críptica. Más adelante en el libro se aclarará.

Un principio importante que veremos en este libro es que los conceptos asociados tradicionalmente con un modelo pueden ser usados con efectos muy positivos en modelos más generales. Por ejemplo, los conceptos de alcance léxico y programación de alto orden, usualmente asociados con la programación funcional, son útiles en todos los modelos. Esto es bien conocido en la comunidad practicante de la programación funcional. Los lenguajes funcionales han sido extendidos con estado explícito (e.g., Scheme [36] y Standard ML [118, 174]) y más recientemente con concurrencia (e.g., Concurrent ML [145] y Concurrent Haskell [137, 135]).

Los límites de los modelos únicos

En nuestra opinión un buen estilo de programación requiere usar conceptos de programación que están normalmente asociados a diferentes modelos de computación. Los lenguajes que implementan un único modelo de computación dificultan esta labor:

- Los lenguajes orientados a objetos fomentan el uso intensivo de los conceptos de estado y herencia. Los objetos tienen estado por defecto. Aunque esto parece simple e intuitivo, realmente complica la programación, e.g., dificulta la concurrencia (ver sección 8.2). Los patrones de diseño, los cuales definen una terminología común para describir buenas técnicas de programación, se explican normalmente en términos del concepto de herencia [54]. En muchos casos, técnicas más simples de programación de alto orden serían suficientes (ver sección 7.4.7). Además, con frecuencia, el concepto de herencia es mal empleado. Por ejemplo, en el desarrollo de interfaces gráficas de usuario en el modelo orientado a objetos, se recomienda usar la herencia

para extender algunas clases genéricas que manejan aparatos⁴ con funcionalidades específicas para la aplicación (e.g., en los componentes Swing de Java). Esto va en contra del principio de separación de responsabilidades.

- Los lenguajes funcionales fomentan el uso intensivo de la programación de alto orden. Algunos ejemplos típicos son los mónadas y la currificación⁵. Los mónadas son usados para codificar el estado pasándolo a través del programa. Esto vuelve los programas más intrincados y no se logran las propiedades de modularidad del verdadero estado explícito (ver sección 4.8). La currificación permite aplicar una función parcialmente, pasándole solamente algunos de sus argumentos. Como resultado, devuelve una función que espera por los argumentos restantes. El cuerpo de la función no se ejecutará hasta que todos los argumentos no hayan llegado. El lado débil es que con inspeccionar el código no queda claro si una función tiene todos sus argumentos o si está aún currificada (“esperando” por el resto).
- Los lenguajes lógicos, siguiendo la tradición de Prolog, fomentan el uso intensivo de la sintaxis de cláusulas de Horn y la búsqueda. Estos lenguajes definen todos los programas como colecciones de cláusulas de Horn, las cuales especifican axiomas lógicos simples en la forma “si-entonces.” Muchos algoritmos se vuelven difíciles de entender cuando son escritos en este estilo. Aunque casi nunca se necesitan, los mecanismos de búsqueda basada en *backtracking* deben ser utilizados siempre (ver [176]).

Estos ejemplos son hasta cierto punto subjetivos; es difícil ser completamente objetivo cuando se está hablando de un buen estilo de programación y de expresividad de un lenguaje. Por lo tanto no deben ser mirados como juicios sobre esos modelos. Más bien, son indicadores de que ninguno de esos modelos es una panacea cuando se usa de manera única. Cada modelo se adapta bien a algún tipo de problemas pero se adapta menos bien a otros. Este libro intenta presentar un enfoque equilibrado, algunas veces usando un modelo de manera aislada pero sin negarse a usar varios modelos juntos cuando sea lo más apropiado.

Enseñar con este libro

A continuación explicamos cómo se adecúa este libro en un currículo en informática y qué cursos se pueden enseñar con él. Por *informática* entendemos el campo completo de la tecnología de la información, incluyendo ciencias de la computación, ingeniería de la computación, y sistemas de información. Algunas veces, la informática es también llamada *computación*.

4. Nota del traductor: *widget* en inglés.

5. Nota del traductor: En inglés, *currying*.

El rol en currículos de informática

Nosotros consideramos la programación como una disciplina independiente de cualquier otro dominio en informática. De acuerdo a nuestra experiencia, la programación se divide naturalmente en tres tópicos centrales:

1. Conceptos y técnicas.
2. Algoritmos y estructuras de datos.
3. Diseño de programas e ingeniería de *software*.

Este libro presenta un minucioso tratamiento del tema (1) y una introducción a los temas (2) y (3). ¿En qué orden se deberían presentar estos temas? Hay una interdependencia muy fuerte entre (1) y (3). La experiencia muestra que el diseño de programas debe ser enseñado temprano, de manera que los estudiantes eviten malos hábitos. Sin embargo, esto es solo una parte de la historia, pues los estudiantes necesitan tener conocimiento sobre los conceptos para expresar sus diseños. Parnas ha utilizado un enfoque que empieza con el tema (3) y usa un modelo de computación imperativo [132]. Debido a que este libro utiliza muchos modelos de computación, recomendamos usarlo para enseñar los temas (1) y (3) concurrentemente, introduciendo a la vez nuevos conceptos y principios de diseño. En el programa de informática de la Universidad Católica de Lovaina en Lovaina la Nueva, Bélgica (UCL), se asignan 8 horas por semestre a cada tema. Esto incluye clases y sesiones de laboratorio. Juntos, estos tres temas conforman un sexto del currículo completo en informática para las formaciones en licenciatura e ingeniería.

Hay otro punto que nos gustaría tratar, y tiene que ver con cómo enseñar programación concurrente. En un currículo tradicional en informática, la concurrencia se enseña extendiendo un modelo con estado, tal como el capítulo 8 extiende el capítulo 6. Es debidamente considerado complejo y difícil programar así. Existen otras formas más sencillas de programación concurrente. La concurrencia declarativa presentada en el capítulo 4 es mucho más simple de usar para programar y puede ser usada con frecuencia en lugar de la concurrencia con estado (ver el epígrafe que comienza el capítulo 4). La concurrencia por flujos⁶, una forma sencilla de concurrencia declarativa, se ha enseñado en cursos de primer año en MIT y otras instituciones. Otra forma sencilla de concurrencia, el paso de mensajes entre hilos, se explica en el capítulo 5. Nosotros sugerimos que tanto la concurrencia declarativa como la concurrencia por paso de mensajes sean parte del currículo estándar y sean enseñadas antes de la concurrencia con estado.

Cursos

Hemos usado este libro como texto para varios cursos ubicados desde segundo año de pregrado hasta cursos de posgrado [144, 178, 179]. En su forma actual, el libro

6. Nota del traductor: *Stream concurrency*, en inglés.

no está orientado como texto de un primer curso de programación, pero el enfoque puede ser ciertamente adaptado para un tal curso.⁷ Los estudiantes deberían tener alguna experiencia básica de programación (e.g., una introducción práctica a la programación y conocimiento de estructuras de datos sencillas como secuencias, conjuntos, y pilas) y algún conocimiento matemático básico (e.g., un primer curso de análisis, matemáticas discretas o álgebra). El libro tiene suficiente material para, por lo menos, cuatro semestres de clases y de sesiones de laboratorio. Algunos de los cursos posibles son:

- Un curso de pregrado sobre conceptos y técnicas de programación. El capítulo 1 presenta una introducción ligera. El curso continúa con los capítulos desde el 2 hasta el 8. Dependiendo de la profundidad del cubrimiento deseado, se puede poner mayor o menor énfasis en algoritmos (para enseñar algoritmos al mismo tiempo que programación), concurrencia (la cual puede ser dejada completamente de lado, si se quiere), o semántica formal (para darle exactitud a las intuiciones).
- Un curso de pregrado sobre modelos de programación aplicados. Esto incluye programación relacional (capítulo 9 (en CTM)), lenguajes de programación específicos (especialmente Erlang, Haskell, Java, y Prolog), programación de interfaces gráficas de usuario (capítulo 10 (en CTM)), programación distribuida (capítulo 11 (en CTM)), y programación por restricciones (capítulo 12 (en CTM)). Este curso es una continuación natural del anterior.
- Un curso de pregrado sobre programación concurrente y distribuida (capítulos 4, 5, 8, y 11 (en CTM)). Los estudiantes deberían tener alguna experiencia en programación. El curso puede empezar con algunas partes de los capítulos 2, 3, 6, y 7 para introducir la programación declarativa y con estado.
- Un curso de posgrado sobre modelos de computación (todo el libro, incluyendo la semántica presentada en el capítulo 13 (en CTM)). El curso puede concentrarse en las relaciones entre los modelos y en la semántica de cada uno de ellos.

El sitio Web del curso tiene más información sobre cursos, incluyendo las transparencias y tareas de laboratorio para algunos de ellos. El sitio Web cuenta con un interpretador animado que muestra cómo el lenguaje núcleo se ejecuta de acuerdo a la semántica de la máquina abstracta. El libro puede ser usado como complemento de otros cursos:

- Parte de un curso de pregrado sobre programación por restricciones (capítulos 4, 9 (en CTM), y 12 (en CTM)).
- Parte de un curso de posgrado sobre aplicaciones colaborativas inteligentes (apartes de todo el libro, con énfasis en la parte II (en CTM)). Si se quiere, el libro puede ser complementado por textos sobre inteligencia artificial (e.g., [148]) o sistemas multiagentes (e.g., [183]).

7. Con mucho gusto ayudaremos a quien esté dispuesto a abordar esta adaptación.

Prefacio

■ Parte de un curso de pregrado sobre semántica. Todos los modelos son definidos formalmente en los capítulos donde se introducen, y su semántica se complementa en el capítulo 13 (en CTM). Así se presenta un caso de estudio, de tamaño real, sobre cómo definir la semántica de un lenguaje de programación moderno y completo.

Aunque el libro tiene un sólido soporte teórico, está orientado a ofrecer una educación práctica en estos aspectos. Cada capítulo cuenta con muchos fragmentos de programas, todos los cuales se pueden ejecutar en el sistema Mozart (ver más abajo). Con estos fragmentos, las clases se pueden ofrecer con demostraciones interactivas de los conceptos. Hemos encontrado que los estudiantes aprecian bastante este tipo de clases.

Cada capítulo termina con un conjunto de ejercicios que normalmente incluyen algo de programación. Ellos se pueden resolver sobre el sistema Mozart. Para la mejor asimilación del material en cada capítulo, animamos a los estudiantes a realizar tantos ejercicios como les sea posible. Los ejercicios marcados (ejercicio avanzado) pueden tomar de varios días a varias semanas para resolverlos. Los ejercicios marcados (proyecto de investigación) son temas abiertos y pueden llevar a realizar contribuciones significativas en investigación en el área.

Software

Una característica práctica del libro es que todos los fragmentos de programa corren sobre una plataforma de *software*, el *Mozart Programming System*. Mozart es un sistema de programación, con todas las características para producir con calidad, acompañado de un sistema de desarrollo incremental y un conjunto completo de herramientas. Mozart compila a un *bytecode* eficiente e independiente de la plataforma que corre sobre muchas variaciones de Unix y Windows, y sobre Mac OS X. Los programas distribuidos pueden propagarse sobre todos estos sistemas. El sitio Web de Mozart, <http://www.mozart-oz.org>, tiene información completa, incluyendo los binarios para descargar, documentación, publicaciones científicas, código fuente, y listas de correo.

El sistema Mozart implementa eficientemente todos los modelos de computación que se cubren en el libro. Esto lo vuelve ideal para usar diferentes modelos al tiempo en el mismo programa y para comparar los modelos escribiendo programas que resuelven un mismo problema en diferentes modelos. Como cada modelo está implementado eficientemente, se pueden escribir programas completos usando solo un modelo. Se pueden utilizar otros modelos después, si se necesita, de manera justificada pedagógicamente. Por ejemplo, se pueden escribir programas enteramente en el estilo orientado a objetos, complementado con pequeños componentes declarativos donde estos son más útiles.

El sistema Mozart es el resultado de un esfuerzo de desarrollo de largo plazo por parte del Mozart Consortium, una colaboración informal de investigación y desarrollo de tres laboratorios. Este sistema ha estado en continuo desarrollo desde 1991. El sistema se ha liberado con todo su código fuente bajo una licencia *Open*

Source. La primera versión pública se liberó en 1995. La primera versión pública con soporte para distribución se liberó en 1999. El libro se basa en una implementación ideal muy cercana a la versión 1.3.0 de Mozart liberada en Abril de 2004. Las diferencias entre la implementación ideal y Mozart están listadas en el sitio Web del libro.

Historia y agradecimientos

Las ideas expresadas en este libro no se produjeron con facilidad. Ellas se producen después de más de una década de discusión, programación, evaluación, deshechar lo malo, adoptar lo bueno y convencer a otros que eso es bueno. Mucha gente contribuyó con ideas, implementaciones, herramientas y aplicaciones. Somos afortunados de haber contado con una visión coherente entre nuestros colegas por un período largo. Gracias a ello, hemos sido capaces de progresar.

Nuestro principal vehículo de investigación y banco de pruebas de nuevas ideas es el sistema Mozart, el cual implementa el lenguaje Oz. Los principales diseñadores y desarrolladores del sistema son (en orden alfabético): Per Brand, Thorsten Brun-klaus, Denys Duchier, Kevin Glynn, Donatien Grolaux, Seif Haridi, Dragan Havelka, Martin Henz, Erik Klintskog, Leif Kornstaedt, Michael Mehl, Martin Müller, Tobias Müller, Anna Neiderud, Konstantin Popov, Ralf Scheidhauer, Christian Schulte, Gert Smolka, Peter Van Roy, y Jörg Würtz. Otros contribuidores importantes son (en orden alfabético): Iliès Alouini, Raphaël Collet, Frej Dreijhammar, Sameh El-Ansary, Nils Franzén, Martin Homik, Simon Lindblom, Benjamin Lorenz, Valentin Mesaros, y Andreas Simon. Agradecemos a Konstantin Popov y Kevin Glynn por administrar la liberación de la versión 1.3.0 de Mozart, la cual se diseñó para acompañar el libro.

También nos gustaría agradecer a los siguientes investigadores y contribuidores indirectos: Hassan Aït-Kaci, Joe Armstrong, Joachim Durchholz, Andreas Franke, Claire Gardent, Fredrik Holmgren, Sverker Janson, Torbjörn Lager, Elie Milgrom, Johan Montelius, Al-Metwally Mostafa, Joachim Niehren, Luc Onana, Marc-Antoine Parent, Dave Parnas, Mathias Picker, Andreas Podelski, Christophe Ponsard, Mahmoud Rafea, Juris Reinfelds, Thomas Sjöland, Fred Spiessens, Joe Turner, y Jean Vanderdonckt.

Agradecemos especialmente a las siguientes personas por su ayuda con material relacionado con el libro. Raphaël Collet por su coautoría de los capítulos 12 (en CTM) y 13 (en CTM), por su trabajo en la parte práctica de LINF1251, un curso ofrecido en la UCL, y por su ayuda con el formato en $\text{\LaTeX} 2\epsilon$. Donatien Grolaux por tres casos de estudio en interfaces gráficas de usuario (utilizados en las secciones 10.4.2 (en CTM)–10.4.4 (en CTM)). Kevin Glynn por escribir la introducción a Haskell (sección 4.7). William Cook por sus comentarios sobre abstracción de datos. Frej Dreijhammar, Sameh El-Ansary, y Dragan Havelka por su ayuda con la parte práctica de Datalogi II, un curso ofrecido en KTH (the Royal Institute of Technology, Stockholm). Christian Schulte por haber pensado y desarrollado com-

Prefacio

pletamente una edición posterior de Datalogi II y por sus comentarios sobre un borrador del libro. Ali Ghodsi, Johan Montelius, y los otros tres asistentes por su ayuda con la parte práctica de esta edición. Luis Quesada y Kevin Glynn por su trabajo sobre la parte práctica de INGI2131, un curso ofrecido en la UCL. Bruno Carton, Raphaël Collet, Kevin Glynn, Donatien Grolaux, Stefano Gualandi, Valentin Mesaros, Al-Metwally Mostafa, Luis Quesada, and Fred Spiessens por sus esfuerzos en leer, probar y verificar los programas ejemplo. Agradecemos muchísimas otras personas, imposible mencionarlas a todas, por sus comentarios sobre el libro. Finalmente, agradecemos a los miembros del *Department of Computing Science and Engineering en UCL*, SICS (*the Swedish Institute of Computer Science, Stockholm*), y el *Department of Microelectronics and Information Technology at KTH*. Pedimos disculpas a todos aquellos que hayamos omitido inadvertidamente.

¿Cómo nos organizamos para conservar un resultado sencillo teniendo esa multitud de desarrolladores trabajando juntos? No es un milagro, sino la consecuencia de una visión fuerte y una metodología de diseño cuidadosamente construida que tomó más de una decada crear y pulir.⁸

Alrededor de 1990, algunos de nosotros regresamos, al mismo tiempo, con fuertes fundamentos tanto teóricos como en construcción de sistemas. Estas personas iniciaron el proyecto ACCLAIM, financiado por la Unión Europea (1991–1994). Por alguna razón, este proyecto se volvió un punto focal.

Entre muchos artículos, los de Sverker Janson y Seif Haridi en 1991 [81] (multiple paradigms in the Andorra Kernel Language AKL), Gert Smolka en 1995 [161] (building abstractions in Oz), y Seif Haridi et al. en 1998 [64] (dependable open distribution in Oz), fueron tres hitos importantes. El primer artículo sobre Oz se publicó en 1993 y ya contaba con muchas ideas importantes [68]. Después del proyecto ACCLAIM, dos laboratorios continuaron trabajando juntos sobre las ideas de Oz: el *Programming Systems Lab (DFKI, Saarland University and Collaborative Research Center SFB 378)*, en Saarbrücken, Alemania, y el *Intelligent Systems Laboratory* en SICS.

El lenguaje Oz fue diseñado originalmente por Gert Smolka y sus estudiantes en el *Programming Systems Lab* [65, 68, 69, 154, 155, 160, 161].

El diseño bien factorizado del lenguaje y la alta calidad de su implementación se deben en gran parte al liderazgo inspirado por Smolka y la experiencia en construcción de sistemas de su laboratorio. Entre los desarrolladores mencionamos a Christian Schulte por su papel coordinando el desarrollo general, Denys Duchier por su activo soporte a usuarios, y Per Brand por su papel coordinando el desarrollo de la implementación distribuida.

En 1996, a los laboratorios alemán y sueco se unió el *Department of Computing Science and Engineering* en UCL cuando el primer autor se fue a trabajar allí.

8. Podemos resumir la metodología en dos reglas (ver [176] para más información). La primera, toda nueva abstracción debe simplificar el sistema o incrementar grandemente su poder expresivo. La segunda, toda nueva abstracción debe tener tanto una implementación eficiente como una formalización sencilla.

Juntos, los tres laboratorios formaron el *Mozart Consortium* con su sitio Web neutral <http://www.mozart-oz.org> de manera que el trabajo no fuera ligado a una única institución.

Este libro fue escrito usando L^AT_EX 2_&, flex, xfig, xv, vi/vim, emacs, y Mozart, al principio sobre un Dell Latitude con Red Hat Linux y KDE, y después sobre un Apple Macintosh PowerBook G4 con Mac OS X y X11. Las fotos de pantalla se tomaron sobre un Sun workstation corriendo Solaris.

El primer autor agradece a la región de Walloon en Bélgica por su generoso soporte al trabajo sobre Oz/Mozart en UCL en los proyectos PIRATES y MILOS.

Comentarios finales

Hemos tratado de hacer que este libro sea útil tanto como libro de texto como de referencia. A Usted de juzgar cuán bien lo hicimos. Debido a su tamaño, es probable que queden algunos errores. Si Usted encuentra alguno, apreciaríamos mucho nos lo hiciera saber. Por favor, envíe estos y todos los comentarios constructivos que pueda tener a la siguiente dirección:

Concepts, Techniques, and Models of Computer Programming
Department of Computing Science and Engineering
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium

Para terminar, nos gustaría agradecer a nuestras familias y amigos por el soporte y ánimo brindados durante los cuatro años que tomó escribir este libro. Seif Haridi agradece especialmente a sus padres Ali y Amina y a su familia Eeva, Rebecca, y Alexander. Peter Van Roy desea agradecer especialmente a sus padres Frans y Hendrika y a su familia Marie-Thérèse, Johan, y Lucile.

Louvain-la-Neuve, Belgium
Kista, Sweden
September 2003

PETER VAN ROY
SEIF HARIDI

Draft

xxviii

Corriendo los programas de ejemplo

Este libro presenta bastantes programas y fragmentos de programa de ejemplo. Todos ellos se pueden correr bajo el *Mozart Programming System*. Para hacerlo se deben tener en cuenta los siguientes puntos:

- El sistema Mozart puede ser descargado sin costo desde el sitio Web del *Mozart Consortium* <http://www.mozart-oz.org>. Existen versiones para diversos sabores de Windows y Unix y para Mac OS X.
- Todos los ejemplos, salvo aquellos previstos como aplicaciones independientes, se pueden correr por medio del ambiente de desarrollo interactivo de Mozart (IDE, por sus siglas en inglés). El apéndice A presenta una introducción a este ambiente.
- Las variables nuevas en los ejemplos interactivos deben ser declaradas con la declaración **declare**. Los ejemplos del capítulo 1 muestran cómo hacerlo. Olvidar estas declaraciones puede producir un error extraño si existen versiones antiguas de las variables. A partir del capítulo 2 la declaración **declare** se omite en el texto cuando es obvio cuáles son las nuevas variables. Sin embargo, la declaración debe ser añadida para correr los ejemplos.
- Algunos capítulos usan operaciones que no forman parte de la versión estándar de Mozart. El código fuente de estas operaciones adicionales (junto con bastante material adicional útil) se colocó en el sitio Web del libro. Recomendamos colocar estas definiciones en su archivo `.ozrc`, de manera que sean cargadas automáticamente desde el arranque del sistema.
- El libro presenta ocasionalmente fotos de la pantalla y medidas de tiempo de programas. A menos que se indique lo contrario, las fotos de la pantalla se tomaron sobre una estación de trabajo de Sun corriendo bajo el sistema operativo Solaris mientras que las medidas de tiempo fueron tomadas a programas corriendo Mozart 1.1.0 sobre la distribución de Linux Red Hat versión 6.1 en un computador portatil Dell Latitude CPx con procesador Pentium III a 500 MHz. La apariencia de las fotos de la pantalla y los tiempos tomados pueden variar sobre su sistema.
- El libro supone una implementación ideal cuya semántica es presentada en los capítulos 13 (en CTM)(modelo de computación general) y 12 (en CTM)(espacios de computación). Existen unas cuantas diferencias entre esta implementación ideal y el sistema Mozart. Ellas se explican en el sitio Web del libro.

1 Introducción a los conceptos de programación

No existe un camino real hacia la geometría.

— Respuesta de Euclides a Ptolomeo, Euclides(*fl. c. 300 A.C.*)

Solo sigue el camino dorado.

— *The Wonderful Wizard of Oz*, L. Frank Baum (1856–1919)

Programar es decirle al computador cómo debería realizar su trabajo. Este capítulo presenta una sencilla y práctica introducción a muchos de los más importantes conceptos en programación. Suponemos que el lector tiene alguna exposición previa a los computadores. Nosotros usamos la interfaz interactiva de Mozart para introducir los conceptos de programación de forma progresiva. Animamos al lector para que ensaye los ejemplos de este capítulo corriéndolos en el sistema Mozart.

Esta introducción solo explora la superficie de los conceptos de programación que veremos en este libro. Los capítulos posteriores profundizan en el entendimiento de estos conceptos y añaden muchos otros conceptos y técnicas.

1.1. Una calculadora

Empecemos por usar el sistema para realizar cálculos. Inicie Mozart digitando:

oz

o haciendo doble clic en el ícono de Mozart. Allí se abre una ventana de edición con dos marcos. En el marco superior, digite la línea siguiente:

{Browse 9999*9999}

Con el ratón seleccione esa línea. Ahora diríjase al menú Oz y seleccione Feed Region. Esto alimenta el sistema con el texto seleccionado. El sistema entonces realiza el cálculo $9999*9999$ y muestra el resultado, 99980001, en una ventana especial llamada el browser. Los corchetes { ... } se usan para hacer una invocación a un procedimiento o a una función. Browse es un procedimiento de un argumento, el cual se invoca {Browse x}. Esta invocación abre el browser, si aún no está abierto, y muestra x dentro de él.

1.2. Variables

Mientras se trabaja con la calculadora, nos gustaría recordar un resultado previo, de manera que se pueda utilizar más tarde sin volverlo a digitar. Esto lo podemos hacer declarando una variable:

```
declare  
v=9999*9999
```

Esto declara `v` y la liga a 99980001. Esta variable la podemos usar más tarde:

```
{Browse v*v}
```

Esto muestra la respuesta 9996000599960001. Las variables son solamente abreviaciones para valores. Ellas no pueden ser asignadas más de una vez. Pero se puede declarar otra variable con el mismo nombre de una previamente declarada. La variable anterior se vuelve inaccesible. Los cálculos previos que la usaban a ella no cambian. Esto se debe a que existen dos conceptos ocultos detrás de la palabra “variable”:

- El identificador. Es lo que se digita. Las variables empiezan con una letra mayúscula y pueden continuar con cualquier número de letras o dígitos. Por ejemplo, la secuencia de caracteres `Var1` puede ser un identificador de variable.
- La variable del almacén. Es lo que el sistema utiliza para calcular. Hace parte de la memoria del sistema, la cual llamamos almacén.

La declaración `declare` crea una nueva variable en el almacén y hace que el identificador de variable tenga una referencia a ella. Los cálculos previos que usan el mismo identificador no se afectan porque el identificador referencia otra variable del almacén.

1.3. Funciones

Realicemos un cálculo un poco más complicado. Suponga que deseamos calcular la función factorial $n!$, la cual se define como $1 \times 2 \times \dots \times (n-1) \times n$. Esta función calcula el número de permutaciones de n elementos, i.e., el número de formas diferentes en que estos elementos se pueden colocar en una fila. Factorial de 10 es:

```
{Browse 1*2*3*4*5*6*7*8*9*10}
```

Esto muestra 3628800. ¿Qué pasa si ahora deseamos calcular el factorial de 100? Nos gustaría que el trabajo tedioso de digitar todos los números de 1 a 100 lo hiciera el sistema. Haremos más: Le diremos al sistema cómo calcular el factorial de cualquier n . Esto lo hacemos definiendo una función:

```
declare
fun {Fact N}
    if N==0 then 1 else N*{Fact N-1} end
end
```

La declaración **declare** crea la nueva variable **Fact**. La declaración **fun** define una función. La variable **Fact** se liga a la función. La función tiene un argumento **N**, el cual es una variable local, i.e., se conoce solamente dentro del cuerpo de la función. Cada vez que se invoca la función se crea una nueva variable local.

Recursión

El cuerpo de la función es una instrucción denominada una expresión **if**. Cuando la función se invoca, la expresión **if** realiza las etapas siguientes:

- Primero comprueba si **N** es igual a 0 realizando la prueba **N==0**.
- Si la prueba tiene éxito, entonces se calcula la expresión que está después del **then**. Esto devuelve simplemente el número 1. Es así debido a que el factorial de 0 es 1.
- Si la prueba falla, entonces se calcula la expresión que está después del **else**. Es decir, si **N** no es igual a 0, entonces se calcula la expresión **N*{Fact N-1}**. ¡Esta expresión utiliza **Fact**, la misma función que estamos definiendo! A esto se le llama recursión. Es perfectamente normal y no es causa de inquietud.

Fact utiliza la siguiente definición matemática de factorial:

$$0! = 1$$

$$n! = n \times (n - 1)! \text{ si } n > 0$$

Esta definición es recursiva debido a que el factorial de **N** es **N** multiplicado por el factorial de **N-1**. Ensayemos la función **Fact**:

```
{Browse {Fact 10}}
```

Esto debería mostrar 3628800 al igual que antes. Así ganamos confianza en que **Fact** está realizando los cálculos correctamente. Ensayemos una entrada un poco más grande:

```
{Browse {Fact 100}}
```

Esto muestra un número enorme (el cual presentamos en grupos de cinco dígitos para mejorar su lectura):

```
933 26215
44394 41526 81699 23885 62667 00490 71596 82643 81621 46859
29638 95217 59999 32299 15608 94146 39761 56518 28625 36979
20827 22375 82511 85210 91686 40000 00000 00000 00000 00000
```

Introducción a los conceptos de programación

Este es un ejemplo de precisión aritmética arbitraria, algunas veces llamada “precisión infinita,” aunque no es infinita. La precisión está limitada por la cantidad de memoria que tenga su sistema. Un computador personal típico, de bajo costo, con 256 MB de memoria puede manejar cientos de miles de dígitos. El lector escéptico preguntará: ¿ese número enorme es verdaderamente el factorial de 100? ¿Cómo podemos asegurarlo? Realizar el cálculo a mano tomaría un tiempo muy largo y probablemente terminaría siendo incorrecto. Más adelante veremos cómo ganar confianza en que el sistema se está comportando correctamente.

Combinaciones

Escribamos una función para calcular el número de combinaciones que se pueden formar con k elementos tomados de un conjunto de n elementos. Esto es lo mismo que calcular el número de subconjuntos, de tamaño k , que se pueden formar a partir de un conjunto de tamaño n . Esto se escribe $\binom{n}{k}$ en notación matemática y se pronuncia “combinatorio $n k$.” Se puede definir como sigue usando la función factorial:

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

lo cual lleva naturalmente a la función siguiente:

```
declare
fun {Comb N K}
  {Fact N} div ({Fact K}*{Fact N-K})
end
```

Por ejemplo, {Comb 10 3} da 120, lo cual corresponde al número de maneras en que se pueden escoger tres elementos de un conjunto de 10 elementos. Esta no es la forma más eficiente de escribir Comb, pero es probablemente la más sencilla.

Abstracción funcional

La definición de Comb utiliza la función Fact, ya existente, en su definición. Siempre es posible utilizar funciones existentes para definir funciones nuevas. Se llama abstracción funcional al hecho de utilizar funciones para construir abstracciones. De esta manera, los programas son como cebollas, con capas de funciones sobre capas de funciones invocando funciones. Este estilo de programación se cubre en el capítulo 3.

1.4. Listas

Ahora podemos calcular funciones sobre los enteros. Pero en realidad no hay mucho que hacer con un entero. Supongamos que deseamos hacer cálculos con

muchos enteros. Por ejemplo, nos gustaría calcular el triángulo de Pascal¹:

$$\begin{array}{ccccccc}
 & & & & & & 1 \\
 & & & & & 1 & \\
 & & & & 1 & 2 & 1 \\
 & & & & 1 & 3 & 3 & 1 \\
 & & & & 1 & 4 & 6 & 4 & 1 \\
 & & & & \cdot & \cdot
 \end{array}$$

Este triángulo debe su nombre al científico y filósofo Blaise Pascal. El triángulo comienza con un 1 en la primera fila. Cada elemento es la suma de los dos elementos justo encima de él a la izquierda y a la derecha (si uno de esos elementos no existe, como en los bordes, se toma como cero). Quisiéramos definir una función que calcule toda la n -ésima fila de un solo golpe. La n -ésima fila contiene n enteros. Podemos calcularla de un solo golpe utilizando listas de enteros.

Una lista no es más que una secuencia de elementos entre paréntesis cuadrados, izquierdo y derecho, como [5 6 7 8]. Por razones históricas, la lista vacía se escribe nil (y no []). Las listas se muestran igual que los números:

```
{Browse [5 6 7 8]}
```

La notación [5 6 7 8] es una abreviación. Realmente, una lista es una cadena de enlaces, donde cada enlace contiene dos cosas: un elemento de la lista y una referencia al resto de la cadena. Las listas siempre se crean elemento por elemento, comenzando con nil y agregando enlaces uno por uno. Un nuevo enlace se escribe H|T, donde H es el elemento nuevo y T es la parte antigua de la cadena. Construyamos una lista. Empezamos con z=nil. Agregamos un primer enlace y=7|z y luego un segundo enlace x=6|y. Ahora x referencia una lista con dos enlaces, una lista que también se puede escribir como [6 7].

El enlace H|T frecuentemente se llama un cons, un término heredado de Lisp.² También lo llamamos un par lista. Crear un nuevo enlace se llama “consear.”³ Si T es una lista, entonces “consear” H y T crea una nueva lista H|T:

-
1. El triángulo de Pascal es un concepto clave en combinatoria. Los elementos de la n -ésima fila son las combinaciones $\binom{n}{k}$, donde k varía de 0 a n . Esto está estrechamente relacionado con el teorema del binomio que dice que $(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^k y^{(n-k)}$ para cualquier entero $n \geq 0$.
 2. La mayoría de la terminología de listas fue introducida con el lenguaje Lisp a finales de la década de los años 1950 y ha permanecido desde ese entonces [111]. Nuestra utilización de la barra vertical es heredada de Prolog, un lenguaje de programación lógica inventado a principios de los años 1970 [37, 163]. En Lisp se escribe el cons como (H . T), lo cual es llamado un par punto.
 3. Nota del traductor: En el libro en inglés dice “consing” lo cual es claramente una palabra inventada para expresar el concepto.

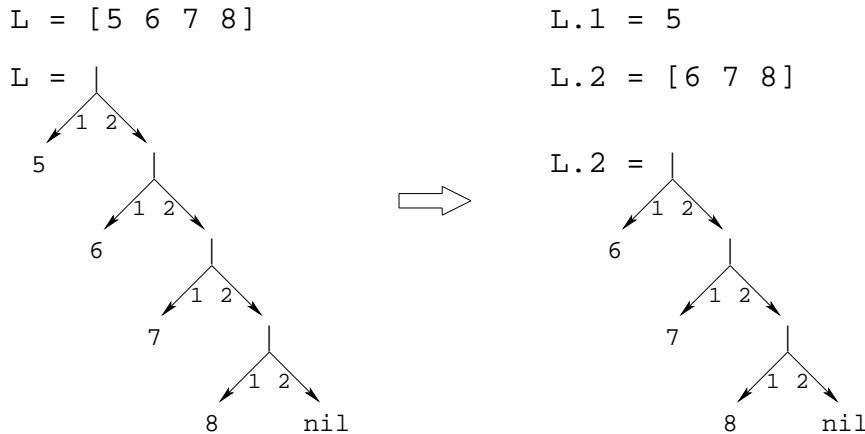


Figura 1.1: Descomposición de la lista [5 6 7 8].

```
declare
H=5
T=[6 7 8]
{Browse H|T}
```

La lista $H|T$ se escribe [5 6 7 8]. Tiene como cabeza 5 y como cola [6 7 8]. El cons $H|T$ puede descomponerse para sacar de allí la cabeza y la cola:

```
declare
L=[5 6 7 8]
{Browse L.1}
{Browse L.2}
```

Aquí se utiliza el operador punto “.”, el cual sirve para seleccionar el primero o el segundo argumento de un par lista. Ejecutar $L.1$ calcula la cabeza de L , el entero 5. Ejecutar $L.2$ calcula la cola de L , la lista [6 7 8]. La figura 1.1 presenta una gráfica: L es una cadena en la cual cada enlace tiene un elemento de la lista y nil marca el final. Ejecutar $L.1$ calcula el primer elemento y ejecutar $L.2$ calcula el resto de la cadena.

Reconocimiento de patrones

Una forma más compacta de descomponer una lista se logra por medio del uso de la instrucción **case**, la cual consigue calcular la cabeza y la cola en una etapa:

```
declare
L=[5 6 7 8]
case L of H|T then {Browse H} {Browse T} end
```

Esto muestra 5 y [6 7 8], igual que antes. La instrucción **case** declara dos variables locales, H y T , y las liga a la cabeza y la cola de la lista L . Decimos que la instrucción **case** realiza reconocimiento de patrones, porque descompone L .

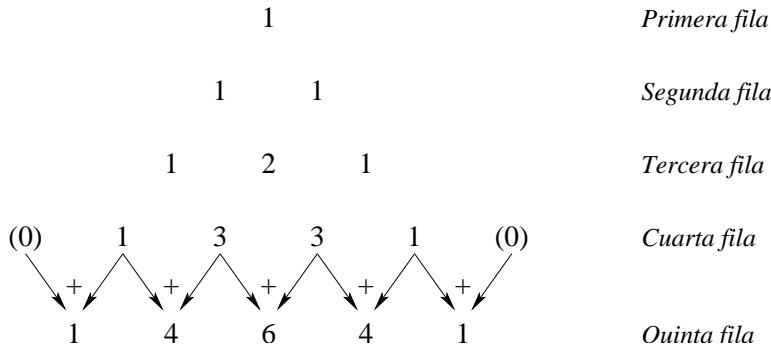


Figura 1.2: Cálculo de la quinta fila del triángulo de Pascal.

de acuerdo al patrón $H|T$. Las variables locales que se declaran con la instrucción **case** son como las variables declaradas con **declare**, salvo que las variables existen solamente en el cuerpo de la instrucción **case**, i.e., entre el **then** y el **end**.

1.5. Funciones sobre listas

Ahora que podemos calcular con listas, definamos una función, `{Pascal N}`, para calcular la n -ésima fila del triángulo de Pascal. Primero, entendamos cómo se hace el cálculo a mano. La figura 1.2 muestra cómo calcular la quinta fila a partir de la cuarta. Miremos cómo funciona esto si cada fila es una lista de enteros. Para calcular una fila, partimos de la anterior. Desplazamos la fila hacia la izquierda una posición y hacia la derecha una posición.⁴ Luego sumamos las dos filas desplazadas. Por ejemplo, tomemos la cuarta fila:

[1 3 3 1]

Desplazemos esta fila a la izquierda y a la derecha y luego sumémoslas elemento por elemento:

$$\begin{array}{ccccc} & [1 & 3 & 3 & 1 & 0] \\ + & [0 & 1 & 3 & 3 & 1] \end{array}$$

Nótese que el desplazamiento a la izquierda añade un cero a la derecha y el desplazamiento a la derecha añade un cero a la izquierda. Realizando la suma se obtiene

[1 4 6 4 1]

lo que corresponde a la quinta fila.

4. Nota del traductor: *shift left* y *shift right*, en inglés.

La función principal

Ahora que entendemos cómo resolver el problema, podemos proceder a escribir la función que realiza esas operaciones. Esta es:

```
declare Pascal SumListas DesplIzq DesplDer
fun {Pascal N}
    if N==1 then [1]
    else
        {SumListas {DesplIzq {Pascal N-1}} {DesplDer {Pascal N-1}}}
    end
end
```

Además de definir `Pascal`, declaramos las variables para las tres funciones auxiliares que falta definir.

Las funciones auxiliares

Para resolver completamente el problema, todavía tenemos que definir tres funciones: `DesplIzq`, la cual realiza el desplazamiento de una posición hacia la izquierda, `DesplDer`, la cual realiza el desplazamiento de una posición hacia la derecha, y `SumListas`, la cual suma dos listas. Estas son `DesplIzq` y `DesplDer`:

```
fun {DesplIzq L}
    case L of H|T then
        H|{DesplIzq T}
    else [0] end
end

fun {DesplDer L} 0|L end
```

`DesplDer` simplemente añade un cero a la izquierda. `DesplIzq` recorre `L` elemento por elemento y construye la salida elemento por elemento. Hemos agregado un `else` a la instrucción `case`. Esto es similar a un `else` en un `if`: se ejecuta si el patrón del `case` no concuerda. Es decir, cuando `L` está vacía, entonces la salida es `[0]`, i.e., una lista que solo contiene al cero.

Esta es `SumListas`:

```
fun {SumListas L1 L2}
    case L1 of H1|T1 then
        case L2 of H2|T2 then
            H1+H2|{SumListas T1 T2}
        end
    else nil end
end
```

Esta es la función más complicada que hemos visto hasta ahora. Utiliza dos instrucciones `case`, una dentro de la otra, debido a que tenemos que descomponer dos listas, `L1` y `L2`. Ahora tenemos completa la definición de `Pascal`. Podemos calcular cualquier fila del triángulo de Pascal. Por ejemplo, invocar `{Pascal 20}` devuelve la vigésima fila:

```
[1 19 171 969 3876 11628 27132 50388 75582 92378  
92378 75582 50388 27132 11628 3876 969 171 19 1]
```

¿Esta respuesta es correcta? ¿Cómo se puede asegurar? Parece correcta: es simétrica (invertir la lista produce la misma lista) y el primero y el segundo argumento son 1 y 19, lo cual es correcto. Al mirar la figura 1.2, es fácil observar que el segundo elemento de la n -ésima fila siempre es $n - 1$ (siempre es uno más que el de la fila previa y empieza en cero para la primera fila). En la próxima sección, veremos cómo razonar sobre corrección.

Desarrollo de software de arriba a abajo

Resumamos la metodología que usamos para escribir Pascal:

- El primer paso consiste en entender cómo realizar el cálculo a mano.
- El segundo paso consiste en escribir una función principal para resolver el problema, suponiendo la existencia de algunas funciones auxiliares (en este caso, `DesplIzq`, `DesplDer`, y `SumListas`).
- El tercer paso consiste en completar la solución escribiendo las funciones auxiliares.

La metodología consistente en escribir primero la función principal y llenar los blancos después se conoce como metodología de desarrollo de *arriba a abajo*.⁵ Es uno de los enfoques más conocidos para el diseño de programas, pero eso es solo una parte de la historia como lo veremos más adelante.

1.6. Corrección

Un programa es correcto si hace lo que nos gustaría que hiciera. ¿Cómo podemos decir si un programa es correcto? Normalmente es imposible duplicar los cálculos del programa a mano. Necesitamos otras maneras. Una manera simple, la cual usamos anteriormente, consiste en verificar que el programa es correcto para las salidas que conocemos. Esto incrementa nuestra confianza en el programa. Pero no llega muy lejos. Para probar la corrección, en general, tenemos que razonar sobre el programa. Esto significa tres cosas:

- Necesitamos un modelo matemático de las operaciones del lenguaje de programación, que defina qué hacen ellas. Este modelo se llama la semántica del lenguaje.
- Necesitamos definir qué quisiéramos que el programa hiciera. Normalmente, esto es una relación matemática entre las entradas con que se alimentará el programa y las salidas que el programa calcula. Esto se llama la especificación del programa.
- Usamos técnicas matemáticas para razonar sobre el programa, utilizando la

5. Nota del traductor: *top-down*, en inglés.

Introducción a los conceptos de programación

semántica. Queremos demostrar que el programa satisface la especificación.

Aunque se haya demostrado que un programa es correcto, éste puede aún producir resultados incorrectos, si el sistema sobre el cual corre no está correctamente implementado. ¿Cómo podemos tener confianza en que el sistema satisface la semántica? Verificar esto es una gran empresa: ¡significa verificar el compilador, el sistema en tiempo de ejecución, el sistema operativo, el *hardware*, y la física sobre la que se basa el *hardware*! Todas estas son tareas importantes pero van más allá del alcance de este libro. Colocamos nuestra confianza en los desarrolladores de Mozart, las compañías de *software*, los fabricantes de *hardware*, y los físicos.⁶

Inducción matemática

Una técnica muy útil es la inducción matemática. Esta consiste en dos etapas. Primero mostramos que el programa es correcto para el caso más simple. Luego mostramos que si el programa es correcto para un caso dado, entonces también lo es para el caso siguiente. Si podemos asegurarnos que todos los casos son finalmente cubiertos, entonces la inducción matemática nos permite concluir que el programa siempre es correcto. Esta técnica se puede aplicar para los enteros y las listas:

- Para los enteros, el caso más simple es 0 y para un entero n dado el caso siguiente es $n + 1$.
- Para las listas, el caso más simple es `nil` (la lista vacía) y para una lista T dada el caso siguiente es $H|T$ (sin condiciones sobre H).

Miremos cómo funciona la inducción para la función factorial:

- `{Fact 0}` devuelve la respuesta correcta, a saber, 1.
- Suponga que `{Fact N-1}` es correcto. Entonces miremos la invocación `{Fact N}`. Vemos que la instrucción `if` toma el caso `else` (pués N no es cero), y calcula $N * {Fact N-1}$. Por hipótesis, `{Fact N-1}` devuelve la respuesta correcta. Por lo tanto, suponiendo que la multiplicación es correcta, `{Fact N}` también devuelve la respuesta correcta.

Este razonamiento utiliza la definición matemática de factorial, a saber, $n! = n \times (n - 1)!$ si $n > 0$, y $0! = 1$. Más adelante en el libro veremos técnicas de razonamiento más sofisticadas. Pero el enfoque básico es siempre el mismo: empezar con la semántica del lenguaje y la especificación del problema, y usar un razonamiento matemático para mostrar que el programa implementa correctamente la especificación.

6. Algunos podrían decir que esto es estúpido. Parafraseando a Thomas Jefferson, ellos podrían decir que el precio de la corrección es la vigilancia eterna.

1.7. Complejidad

La función Pascal que definimos atrás se comporta muy lentamente si tratamos de calcular filas de mayor numeración en el triángulo. Calcular la fila 20 toma un segundo o dos. La fila 30 toma varios minutos.⁷ Si lo ensaya, espere pacientemente el resultado. ¿Por Qué se toma todo este tiempo de más? Miremos de nuevo la función Pascal:

```
fun {Pascal N}
    if N==1 then [1]
    else
        {SumListas {DesplIzq {Pascal N-1}} {DesplDer {Pascal N-1}}}
    end
end
```

Invocar {Pascal N} invocará, a su vez, dos veces a {Pascal N-1}. Por lo tanto, invocar {Pascal 30} invocará {Pascal 29} dos veces, llevando a cuatro invocaciones de {Pascal 28}, ocho de {Pascal 27}, y así sucesivamente doblando el número de invocaciones por cada fila inferior. Esto lleva a 2^{29} invocaciones a {Pascal 1}, lo cual es aproximadamente un billón. No debe sorprender entonces que calcular {Pascal 30} sea lento. ¿Podemos acelerarlo? Sí, existe una forma sencilla: invocar solo una vez a {Pascal N-1} en lugar de hacerlo dos veces. La segunda invocación produce el mismo resultado que la primera. Si tan solo recordáramos el primer resultado, una invocación sería suficiente. Podemos recordarlo usando una variable local. Presentamos la nueva función, PascalRápido, la cual usa una variable local:

```
fun {PascalRápido N}
    if N==1 then [1]
    else L in
        L={PascalRápido N-1}
        {SumListas {DesplIzq L} {DesplDer L}}
    end
end
```

Declaramos la variable local L agregando “**L in**” en la parte **else**. Esto es lo mismo que usar **declare**, salvo que el identificador solo puede ser usado entre el **else** y el **end**. Ligamos L al resultado de {PascalRápido N-1}. Ahora podemos usar L donde lo necesitemos. ¿Cuán rápida es PascalRápido? Ensayémosla calculando la fila 30. Esto toma minutos con Pascal, pero se realiza casi que instantáneamente con PascalRápido. Una lección que podemos aprender de este ejemplo es que usar un buen algoritmo es más importante que tener el mejor compilador posible o la máquina más rápida.

7. Estos tiempos varían según la velocidad de su máquina.

Garantías en el tiempo de ejecución

Como lo muestra este ejemplo, es importante saber algo sobre el tiempo de ejecución de un programa. Conocer el tiempo exacto es menos importante que saber que el tiempo no se ampliará exageradamente con un incremento en el tamaño de la entrada. El tiempo de ejecución de un programa medido como una función del tamaño de la entrada, módulo un factor constante, se llama la *complejidad en tiempo* del programa. Cuál sea la forma de esta función, depende de cómo se mide el tamaño de la entrada. Suponemos que éste se mide de forma que tiene sentido para la manera en que el programa se usa. Por ejemplo, tomamos como tamaño de la entrada de `{Pascal N}` simplemente el entero N (y no, e.g., la cantidad de memoria que se necesita para almacenar N).

La complejidad en tiempo de `{Pascal N}` es proporcional a 2^n . Esto es una función exponencial en n , la cual crece muy rápidamente a medida que n se incrementa. ¿Cuál es la complejidad en tiempo de `{PascalRápido N}`? Se realizan n invocaciones recursivas y cada una de ellas toma tiempo proporcional a n . La complejidad en tiempo es, por tanto, proporcional a n^2 . Esta es una función polinomial en n que crece a una velocidad mucho más lenta que una función exponencial. Los programas cuya complejidad en tiempo es exponencial son poco prácticos salvo para entradas de tamaño pequeño. Los programas cuya complejidad en tiempo son polinomios de bajo orden son prácticos.

1.8. Evaluación perezosa

Las funciones que hemos escrito hasta ahora realizan sus cálculos tan pronto como son invocadas. A esto se le llama evaluación ansiosa. Existe otra forma de evaluar funciones llamada evaluación perezosa.⁸ En la evaluación perezosa, solo se realiza un cálculo cuando su resultado se necesita. Este tema se cubre en el capítulo 4 (ver sección 4.5). A continuación presentamos una función perezosa, sencilla, que calcula una lista de enteros:

```
fun lazy {Ents N}
    N | {Ents N+1}
end
```

Invocar `{Ents 0}` calcula la lista infinita $0 | 1 | 2 | 3 | 4 | 5 | \dots$. Esto parece un ciclo infinito, pero no lo es. La anotación `lazy` asegura que la función sólo sea evaluada cuando se necesite. Esta es una de las ventajas de la evaluación perezosa: podemos calcular con estructuras de datos potencialmente infinitas sin condición alguna de límite de ciclos. Por ejemplo:

```
L={Ents 0}
{Browse L}
```

8. La evaluación ansiosa y la perezosa algunas veces son llamadas evaluación dirigida por los datos y evaluación dirigida por la demanda, respectivamente.

Esto muestra lo siguiente, i.e., absolutamente nada sobre los elementos de L:

```
L<Future>
```

(El browser no hace que las funciones perezosas sean evaluadas.) La anotación “<Future>” significa que L está ligada a una función perezosa. Si se necesita algún elemento de L, entonces esa función se invocará automáticamente. Aquí podemos ver un cálculo que necesita un elemento de L:

```
{Browse L.1}
```

Esto muestra el primer elemento de L, a saber, 0. Podemos realizar cálculos con la lista como si estuviera completamente calculada:

```
case L of A|B|C|_ then {Browse A+B+C} end
```

Esto obliga a que se calculen los primeros tres elementos de L y ni uno más. ¿Qué se muestra en el browser?

Cálculo perezoso del triángulo de Pascal

Realicemos algo útil usando evaluación perezosa. Nos gustaría escribir una función que calcule tantas filas del triángulo de Pascal como se necesiten, pero sin conocer de antemano cuántas. Es decir, tenemos que mirar las filas para decidir cuándo, las que hay, son suficientes. A continuación, una función perezosa que genera una lista infinita de filas:

```
fun lazy {ListaPascal Fila}
  Fila|{ListaPascal
    {SumListas {DesplIzq Fila} {DesplDer Fila}}}
end
```

Invocar esta función y mirar el resultado no mostrará nada:

```
declare
L={ListaPascal [1]}
{Browse L}
```

(El argumento [1] es la primera fila del triángulo) Para mostrar más resultados, las filas deben necesitarse:

```
{Browse L.1}
{Browse L.2.1}
```

Esto muestra la primera y la segunda filas.

En lugar de escribir una función perezosa, podríamos escribir una función que toma N, el número de filas que necesitamos, y calcula directamente estas filas a partir de una fila inicial:

Introducción a los conceptos de programación

```
fun {ListaPascal2 N Fila}
  if N==1 then [Fila]
  else
    Fila|{ListaPascal2 N-1
           {SumListas {DesplIzq Fila} {DesplDer Fila}}}
  end
end
```

Podemos mostrar 10 filas invocando `{Browse {ListaPascal2 10 [1]}}`. ¿Pero qué pasa si más tarde decidimos que se necesitan 11 filas? Tendríamos que invocar de nuevo `ListaPascal2`, con argumento 11. Esto realizaría de nuevo el trabajo de definir las primeras 10 filas. La versión perezosa evita la repetición de este trabajo. Siempre queda lista para continuar donde quedó.

1.9. Programación de alto orden

Hemos escrito una función eficiente, `PascalRápido`, que calcula filas del triángulo de Pascal. Ahora nos gustaría experimentar con variaciones del triángulo de Pascal. Por ejemplo, en lugar de sumar los números de las listas para calcular cada fila, nos gustaría restarlos, o aplicarles el o-exclusivo (calcular solo si los dos números son pares o impares a la vez), o muchas otras alternativas. Una manera de hacer esto es escribir una nueva versión de `PascalRápido` para cada variación. Pero esto se vuelve pesado muy rápidamente. ¿Es posible tener una sola versión que se pueda usar para todas las variaciones? De hecho, es posible. Llamémosla `PascalGenérico`. Siempre que la invoquemos, pasamos la función adecuada a la variación (suma, o-exclusivo, etc.) como argumento. La capacidad de pasar funciones como argumentos se conoce como programación de alto orden.

A continuación se presenta la definición de `PascalGenérico`. Esta cuenta con un argumento adicional `Op` que contiene la función que calcula cada número:

```
fun {PascalGenérico Op N}
  if N==1 then [1]
  else L in
    L={PascalGenérico Op N-1}
    {OpListas Op {DesplIzq L} {DesplDer L}}
  end
end
```

`SumListas` se reemplaza por `OpListas`. El argumento adicional `Op` se pasa a `OpListas`. `DesplIzq` y `DesplDer` no necesitan conocer `Op`, de manera que podemos usar las versiones existentes. La definición de `OpListas` es:

```
fun {OpListas Op L1 L2}
  case L1 of H1|T1 then
    case L2 of H2|T2 then
      {Op H1 H2}|{OpListas Op T1 T2}
    end
  else nil end
end
```

En lugar de realizar la suma `H1+H2`, esta versión realiza `{Op H1 H2}`.

Variaciones sobre el triángulo de Pascal

Definamos algunas funciones para ensayar `PascalGenérico`. Para lograr el triángulo de Pascal original, podemos definir la función de adición:

```
fun {Sum X Y} X+Y end
```

Ahora podemos correr `{PascalGenérico Sum 5}`.⁹ Esto calcula la quinta fila exactamente como antes. Podemos definir `PascalRápido` utilizando `PascalGenérico`:

```
fun {PascalRápido N} {PascalGenérico Sum N} end
```

Definamos otra función:

```
fun {Xor X Y} if X==Y then 0 else 1 end end
```

Esto realiza una operación de *o exclusivo*, la cual se define a continuación:

X	Y	{Xor X Y}
0	0	0
0	1	1
1	0	1
1	1	0

El o-exclusivo nos permite calcular la paridad de cada número en el triángulo de Pascal, i.e., si el número es par o impar. Los números, como tal, no se calculan. Invocar `{PascalGenérico Xor N}` da como resultado:

					1
				1	1
			1	0	1
		1	1	1	1
	1	0	0	0	1
1	1	0	0	1	1
.

Algunas otras funciones se dejan como ejercicio.

1.10. Concurrencia

9. También podemos correr `{PascalGenérico Number.'+' 5}`, pues la operación de adición `'+'` hace parte del módulo `Number`. Pero los módulos no se introducen en este capítulo.

Introducción a los conceptos de programación

Nos gustaría que nuestro programa tuviera varias actividades independientes, cada una de las cuales se ejecute a su propio ritmo. Esto se llama *concurrency*. No debería existir interferencia entre la actividades, a menos que el programador decida que necesitan comunicarse. Así es como el mundo real funciona afuera del sistema. Nos gustaría poder hacer esto dentro del sistema también.

Introducimos la concurrencia por medio de la creación de hilos. Un hilo es simplemente un programa en ejecución como las funciones que vimos anteriormente. La diferencia es que un programa puede tener más de un hilo. Los hilos se crean con la instrucción **thread**. ¿Recuerda como era de lenta la versión original de Pascal? Podemos invocar Pascal dentro de su propio hilo. Esto significa que no impedirá que otros cálculos continúen realizándose. Pueden desacelerarse, si Pascal tiene realmente mucho trabajo por hacer. Esto se debe a que los hilos comparten el mismo computador subyacente. Pero ninguno de los hilos se detendrá. He aquí un ejemplo:

```
thread P in
  P={Pascal 30}
  {Browse P}
end
{Browse 99*99}
```

Esto crea un hilo nuevo. Dentro de este hilo, invocamos `{Pascal 30}` y luego invocamos `Browse` para mostrar el resultado. El hilo nuevo tiene mucho trabajo por hacer. Pero esto no le impide mostrar `99*99` inmediatamente.

1.11. Flujo de datos

¿Qué pasa si una operación trata de usar una variable que no esté ligada aún? Desde un punto de vista puramente estético, sería muy agradable si la operación simplemente esperara. Tal vez otro hilo ligue la variable, y entonces la operación pueda continuar. Este comportamiento civilizado se conoce como flujo de datos. La figura 1.3 muestra un ejemplo sencillo: las dos multiplicaciones esperan hasta que sus argumentos estén ligados y la suma espera hasta que las multiplicaciones se completen. Como veremos más adelante en el libro, hay muy buenas razones para tener acceso a un comportamiento por flujo de datos. Por ahora miremos cómo el flujo de datos y la concurrencia trabajan juntos. Consideremos, e.g.:

```
declare x in
thread {Delay 10000} x=99 end
{Browse inicio} {Browse x*x}
```

La multiplicación `x*x` espera hasta que `x` sea ligada. El primer `Browse` muestra de manera inmediata `inicio`. El segundo `Browse` espera por el resultado de la multiplicación, de manera que no muestra nada todavía. El `{Delay 10000}` invoca una pausa por 10000 ms (i.e., 10 segundos). `x` es ligada sólo después del retraso invocado. Una vez `x` se liga, la multiplicación continúa y el segundo `browse` muestra 9801. Las dos operaciones `x=99` y `x*x` se pueden realizar en cualquier orden con

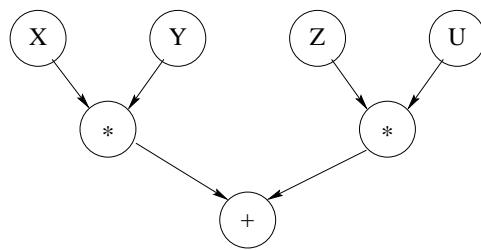


Figura 1.3: Un ejemplo sencillo de ejecución por flujo de datos.

cualquier tipo de retraso; la ejecución por flujo de datos siempre producirá el mismo resultado. El único efecto que puede tener un retraso es que las cosas vayan despacio. Por ejemplo:

```
declare X in
thread {Browse inicio} {Browse X*X} end
{Delay 10000} X=99
```

Esto se comporta exactamente igual que antes: el browser muestra 9801 después de 10 segundos. Esto ilustra dos propiedades agradables del flujo de datos. Primero, los cálculos funcionan correctamente independientemente de cómo se han repartido entre los hilos. Segundo, los cálculos son pacientes: no señalan errores, sino que simplemente esperan.

Agregar hilos y retrasos a un programa puede cambiar radicalmente la apariencia de un programa. Pero siempre que las operaciones se invoquen con los mismos argumentos, los resultados del programa no cambian para nada. Esta es la propiedad clave de la concurrencia por flujo de datos y es la razón por la que presenta la mayoría de las ventajas de la concurrencia sin las complejidades que normalmente se le asocian a ésta. La concurrencia por flujo de datos se cubre en el capítulo 4.

1.12. Estado explícito

¿Cómo podemos lograr que una función aprenda de su pasado? Es decir, nos gustaría que la función tuviera algún tipo de memoria interna que la ayude a realizar su tarea. La memoria se necesita en funciones que pueden cambiar su comportamiento y aprender de su pasado. Este tipo de memoria se llama estado explícito. De la misma manera que la concurrencia, el estado explícito modela un aspecto esencial de la manera como el mundo real funciona. Nos gustaría ser capaces de hacer esto en el sistema también. Más adelante en el libro veremos razones más profundas para tener estado explícito (ver capítulo 6). Por ahora, miremos cómo funciona.

Por ejemplo, quisieramos ver cuán seguido se utiliza la función `PascalRápido`. ¿Hay alguna manera para que `PascalRápido` pueda recordar cuántas veces fue

Introducción a los conceptos de programación

invocada? Podemos hacerlo añadiendo estado explícito.

Una celda de memoria

Hay muchas maneras de definir estado explícito. La manera más simple es definir una celda sencilla de memoria. Esto es como una caja en la cual se puede guardar cualquier contenido. Muchos lenguajes de programación llaman a esto una “variable.” Nosotros lo llamamos una “celda” para evitar confusiones con las variables que usamos antes, las cuales son más parecidas a variables matemáticas, i.e., no más que abreviaciones para valores. Hay tres funciones sobre celdas: `NewCell` crea una celda nueva, `:=` (asignación) coloca un valor nuevo en una celda, y `@` (acceso) toma el valor actualmente almacenado en la celda. Acceso y asignación también se llaman leer y escribir. Por ejemplo:

```
declare
C={NewCell 0}
C:=@C+1
{Browse @C}
```

Esto crea una celda `C` con contenido inicial 0, agrega uno al contenido, y lo muestra.

Agregando memoria a PascalRápido

Con una celda de memoria, podemos lograr que `PascalRápido` cuente cuántas veces es invocada. Primero creamos una celda externa a `PascalRápido`. Luego, dentro de `PascalRápido`, agregamos 1 al contenido de la celda. El resultado es:

```
declare
C={NewCell 0}
fun {PascalRápido N}
  C:=@C+1
  {PascalGenérico Sum N}
end
```

(Para mantenerlo corto, esta definición utiliza `PascalGenérico`.)

1.13. Objetos

Una función con memoria interna se llama normalmente un *objeto*. La versión extendida de `PascalRápido` que definimos en la sección anterior es un objeto. Resulta que los objetos son unos “animales” muy útiles. Presentamos otro ejemplo. Definimos un objeto contador. El contador tiene una celda que mantiene la cuenta actual. El contador tiene dos operaciones, `Incr` y `Leer`, las cuales llamaremos su interfaz. `Incr` agrega 1 a la cuenta y luego devuelve el valor resultante de la cuenta. `Leer` solo devuelve el valor de la cuenta. La definición es:

```
declare
local C in
  C={NewCell 0}
  fun {Incr}
    C:=@C+1
    @C
  end
  fun {Leer}
    @C
  end
end
```

La declaración **local** declara una variable nueva **C** que es visible solo hasta el **end** correspondiente. Hay algo especial para resaltar: la celda se referencia con una variable local de manera que queda completamente invisible desde el exterior. Esto se llama encapsulación. La encapsulación implica que los usuarios no pueden meterse con lo interno del contador. Podemos garantizar que el contador siempre funcionará correctamente sin importar cómo se utiliza. Esto no era cierto para **PascalRápido** extendido debido a que cualquiera podría consultar y modificar la celda.

Por lo tanto, siempre que la interfaz del objeto contador sea la misma, el programa usuario no necesita conocer la implementación. Separar la interfaz de la implementación es la esencia de la abstracción de datos. Esta puede simplificar bastante el programa usuario. Un programa que utiliza un contador funcionará correctamente para cualquier implementación siempre que la interfaz sea la misma. Esta propiedad se llama polimorfismo. La abstracción de datos con encapsulación y polimorfismo se cubre en el capítulo 6 (ver sección 6.4).

Podemos hacer crecer el contador:

```
{Browse {Incr}}
{Browse {Incr}}
```

¿Qué muestra esto? **Incr** se puede usar en cualquier parte en un programa para contar cuántas veces pasa algo. Por ejemplo, **PascalRápido** podría usar **Incr**:

```
declare
fun {PascalRápido N}
  {Browse {Incr}}
  {PascalGenérico Sum N}
end
```

1.14. Clases

En la última sección se definió un objeto contador. ¿Qué pasa si necesitamos más de un contador? Sería agradable tener una “fábrica” que pueda fabricar tantos contadores como necesitemos. Tal fábrica se llama una *clase*. Una forma de definirla es:

Introducción a los conceptos de programación

```
declare
  fun {ContadorNuevo}
    C Incr Leer in
      C={NewCell 0}
      fun {Incr}
        C:=@C+1
        @C
      end
      fun {Leer}
        @C
      end
    end
    contador(incr:Incr leer:Leer)
  end
```

ContadorNuevo es una función que crea una celda nueva y devuelve nuevas funciones `Incr` y `Leer` que utilizan esa celda. Crear funciones que devuelven funciones como resultado es otra forma de programación de alto orden.

Agrupamos las funciones `Incr` y `Leer` juntas dentro de un registro, el cual es una estructura de datos compuesta que permite fácil acceso a sus partes. El registro `contador(incr:Incr leer:Leer)` se caracteriza por su etiqueta `contador` y por sus dos campos, llamados `incr` y `leer`. Creemos dos contadores:

```
declare
  Ctrl1={ContadorNuevo}
  Ctrl2={ContadorNuevo}
```

Cada contador tiene su memoria interna propia y sus funciones `Incr` y `Leer` propias. Podemos acceder a estas funciones utilizando el operador “.” (punto). `Ctrl1.incr` accede a la función `Incr` del primer contador. Hagamos crecer el primer contador y mostremos su resultado:

```
{Browse {Ctrl1.incr}}
```

Hacia la programación orientada a objetos

Hemos presentado un ejemplo de clase sencilla, `ContadorNuevo`, que define dos operaciones, `Incr` y `Leer`. Las operaciones definidas dentro de las clases se llaman métodos. La clase puede ser usada para crear tantos objetos contador como necesitemos. Todos estos objetos comparten los mismos métodos, pero cada uno tiene su memoria interna propia por separado. La programación con clases y objetos se llama programación basada en objetos.

Añadir una idea nueva, la herencia, a la programación basada en objetos lleva a la programación orientada a objetos. La herencia significa que se puede definir una clase nueva en términos de clases ya existentes especificando solamente qué es lo diferente de la clase nueva. Por ejemplo, supongamos que tenemos una clase contador que solo tiene el método `Incr`. Podemos definir una clase nueva, igual que la primera clase salvo que le agrega a ella el método `Leer`. Decimos que la clase nueva hereda de la primera clase. La herencia es un concepto poderoso para estructurar programas. Permite que una clase sea definida incrementalmente,

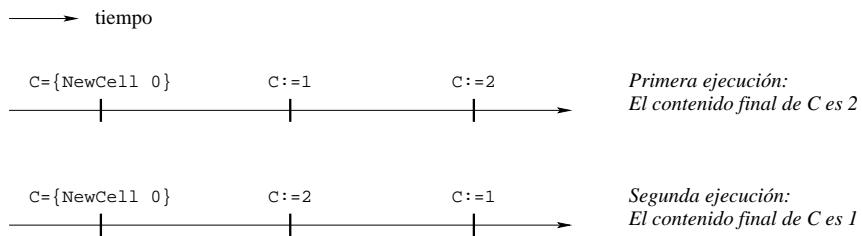


Figura 1.4: Todas las ejecuciones posibles del primer ejemplo no-determinístico.

en diferentes partes del programa. La herencia es un concepto delicado de usar correctamente. Para que la herencia sea fácil de usar, los lenguajes orientados a objetos agregan sintaxis especial. El capítulo 7 cubre la programación orientada a objetos y muestra cómo programar con herencia.

1.15. No-determinismo y tiempo

Hemos visto cómo agregar, separadamente, concurrencia y estado a un programa. ¿Qué pasa cuando un programa tiene ambas? Resulta que tener ambas al mismo tiempo es un asunto delicado, pues el mismo programa puede tener resultados diferentes entre una y otra ejecución. Esto se debe a que el orden en que los hilos tienen acceso al estado puede cambiar de una ejecución a la otra. Esta variabilidad se llama no-determinismo. El no-determinismo existe debido a la imposibilidad de determinar el momento exacto del tiempo en el cual una operación básica se ejecuta. Si conocieramos ese momento exacto, entonces no habría no-determinismo. Pero no se puede conocer ese momento, simplemente porque los hilos son independientes. Como ninguno sabe nada sobre los otros, tampoco saben qué instrucciones ha ejecutado cada uno.

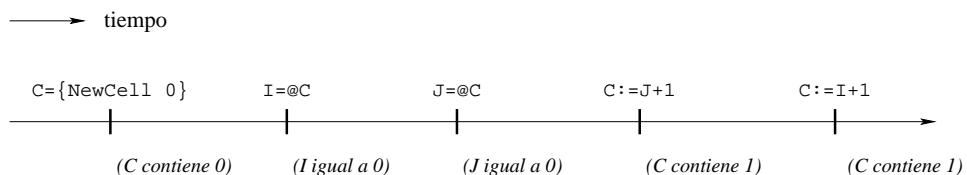
El no-determinismo en sí mismo no es un problema; ya lo tenemos con la concurrencia. Las dificultades aparecen si el no-determinismo sale a la luz en el programa i.e., si es observable. Al no-determinismo observable a veces se le llama condición de carrera. Veamos un ejemplo:

```

declare
C={NewCell 0}
thread
    C:=1
end
thread
    C:=2
end

```

¿Cuál es el contenido de `C` después que se ejecuta este programa? La figura 1.4 muestra las dos posibles ejecuciones de este programa. Dependiendo de cuál se

Introducción a los conceptos de programación**Figura 1.5:** Una ejecución posible del segundo ejemplo no-determinístico.

realice, el contenido final de la celda será 1 o 2. El problema es que no podemos predecir cuál va a ser. Este es un caso sencillo de no-determinismo observable. Las cosas pueden ser mucho más delicadas. Por ejemplo, usemos una celda para almacenar un contador que puede ser incrementado por diversos hilos.

```

declare
  C={NewCell 0}
  thread I in
    I=@C
    C:=I+1
  end
  thread J in
    J=@C
    C:=J+1
  end

```

¿Cuál es el contenido de *C* después que se ejecuta este programa? Al parecer cada hilo simplemente agrega 1 al contenido, dejándolo al final en 2. Pero hay una sorpresa oculta: ¡el contenido final también puede ser 1! ¿Cómo puede suceder? Trate de imaginarse por qué antes de continuar.

Intercalación

El contenido puede ser 1 debido a que la ejecución de los hilos es intercalada. Es decir, en cada turno de ejecución, se ejecuta un pedazo del hilo. Debemos suponer que cualquier posible intercalación puede llegar a ocurrir. Por ejemplo, considere la ejecución mostrada en la figura 1.5. Tanto *I* como *J* están ligados a 0. Entonces, como *I*+1 y *J*+1 son ambos 1, la celda se asigna con 1 dos veces. El resultado final es que el contenido de la celda es 1.

Este es un ejemplo sencillo. Programas mucho más complicados tienen muchas más intercalaciones posibles. Programar con concurrencia y estado juntos es principalmente un asunto de dominio de las intercalaciones. En la historia de la tecnología de computación, muchos errores famosos y peligrosos se debieron a que los diseñadores no se dieron cuenta de lo difícil que es, realmente, esto. La máquina de terapia de radiación Therac-25 es un ejemplo infame. Debido a los errores de programación concurrente, la máquina daba a sus pacientes dosis de radiación que eran miles de veces más grandes que lo normal, resultando en muerte o lesiones graves[104].

Esto nos conduce a una primera lección de programación con estado y concu-

rrencia: ¡de ser posible, no las use al mismo tiempo! Resulta que muchas veces no necesitamos de ambas al tiempo. Cuando un programa definitivamente necesita ambas al tiempo, éste se puede diseñar, casi siempre, de manera que su interacción esté limitada a una parte muy pequeña del programa.

1.16. Atomicidad

Pensemos un poco más sobre cómo programar con concurrencia y estado. Una forma de hacerlo más fácil es usar operaciones atómicas. Una operación es *atómica* si no se pueden observar estados intermedios. Es como si se saltara directamente del estado inicial al estado resultado. La programación con acciones atómicas se cubre en el capítulo 8.

Con operaciones atómicas podemos resolver el problema de intercalación del contador almacenado en la celda. La idea es asegurarse que cada cuerpo de hilo sea atómico. Para hacer esto, necesitamos una manera de construir operaciones atómicas. Para esto, introducimos una nueva entidad del lenguaje llamada un candado. Un candado tiene un interior y un exterior. El programador define las instrucciones que van en el interior. Un candado tiene la propiedad que solamente se puede estar ejecutando un hilo a la vez en su interior. Si un segundo hilo trata de instalarse en su interior, entonces el candado espera a que el primer hilo termine antes de dejarlo entrar. Por lo tanto, lo que pasa en el interior del candado es atómico.

Necesitamos dos operaciones sobre candados. Primero, creamos un nuevo candado invocando la función NewLock. Segundo, definimos el interior del candado con la instrucción **lock L then ... end**, donde L es un candado. Ahora podemos arreglar la celda que almacena el contador:

```
declare
C={NewCell 0}
L={NewLock}
thread
  lock L then I in
    I=@C
    C:=I+1
  end
end
thread
  lock L then J in
    J=@C
    C:=J+1
  end
end
```

En esta versión, el resultado final siempre es 2. Ambos cuerpos de los hilos deben ser asegurados por el mismo candado, pues de otra manera la indeseable intercalación aparece. ¿Logra ver por qué?

1.17. ¿Para donde vamos?

En este capítulo se presentó una mirada rápida a muchos de los más importantes conceptos de programación. Las intuiciones aquí presentadas servirán también en los capítulos que vienen, donde definiremos en forma precisa los conceptos y los modelos de computación de las que ellas hacen parte. En este capítulo se introdujeron los modelos de computación siguientes:

- Modelo declarativo (capítulos 2 y 3). Los programas declarativos definen funciones matemáticas. Son los más sencillos para razonar sobre ellos y para probarlos. El modelo declarativo también es importante porque contiene muchas de las ideas que serán usadas más tarde, en modelos más expresivos.
- Modelo concurrente declarativo (capítulo 4). Al añadir concurrencia por flujo de datos llegamos a un modelo que, siendo aún declarativo, permite una ejecución incremental más flexible.
- Modelo perezoso declarativo (sección 4.5). Añadir evaluación perezosa permite calcular con estructuras de datos potencialmente infinitas. Esto es bueno para la administración de recursos y la estructura de los programas.
- Modelo con estado (capítulo 6). Agregar estado explícito permite escribir programas cuyo comportamiento cambia en el tiempo. Esto es bueno para la modularidad de los programas. Si están bien escritos, i.e., utilizando encapsulación e invariantes, es casi tan fácil razonar sobre estos programas como sobre los programas declarativos.
- Modelo orientado a objetos (capítulo 7). La programación orientada a objetos es un estilo de programación que utiliza abstracciones de datos para programar con estado. Hace sencillo usar técnicas poderosas como el polimorfismo y la herencia.
- Modelo concurrente con estado compartido (capítulo 8). Este modelo añade tanto concurrencia como estado explícito. Si se programa cuidadosamente, utilizando las técnicas para dominar la intercalación, como los monitores y las transacciones, este modelo presenta las ventajas tanto del modelo con estado como del modelo concurrente.

Además de estos modelos, el libro cubre muchos otros modelos útiles tales como el modelo declarativo con excepciones (sección 2.7), el modelo concurrente por paso de mensajes (capítulo 5), el modelo relacional (capítulo 9 (en CTM)), y los modelos especializados de la parte II (en CTM).

1.18. Ejercicios

1. *Una calculadora.* La sección 1.1 utiliza el sistema como una calculadora. Exploraremos las posibilidades:

- a) Calcule el valor exacto de 2^{100} sin utilizar ninguna función nueva. Trate de

pensar en atajos para hacerlo, sin tener que digitar $2*2*2*\dots*2$ con el 2 cien veces. *Sugerencia:* utilice variables para almacenar resultados intermedios.

b) Calcule el valor exacto de $100!$ sin utilizar ninguna función nueva. En este caso, existen atajos posibles?

2. *Calculando combinaciones.* En la sección 1.3 se define la función `Comb` para calcular números combinatorios. Esta función no es muy eficiente, pues puede requerir de calcular factoriales muy grandes. El propósito de este ejercicio es escribir una versión más eficiente de `Comb`.

a) Como un primer paso, utilice la definición alternativa siguiente para escribir una función más eficiente:

$$\binom{n}{k} = \frac{n \times (n - 1) \times \dots \times (n - k + 1)}{k \times (k - 1) \times \dots \times 1}$$

Calcule el numerador y el denominador separadamente y luego divídalos. Asegúrese que el resultado sea 1 cuando $k = 0$.

b) Como un segundo paso, utilice la siguiente identidad:

$$\binom{n}{k} = \binom{n}{n - k}$$

para incrementar aún más la eficiencia. Es decir, si $k > n/2$, entonces realice el cálculo con $n - k$ en lugar de hacerlo con k .

3. *Corrección de programas.* En la sección 1.6 se explican las ideas básicas de corrección de programas y se aplican para mostrar que la función factorial definida en la sección 1.3 es correcta. En este ejercicio, aplique las mismas ideas para mostrar que la función `Pascal` de la sección 1.5 es correcta.

4. *Complejidad de programas.* ¿Qué se dice en la sección 1.7 sobre los programas cuya complejidad en tiempo es un polinomio de alto orden? ¿Son esos programas prácticos o no? ¿Qué piensa usted?

5. *Evaluación perezosa.* En la sección 1.8 se define la función perezosa `Ents` que calcula perzosamente una lista infinita de enteros. Definamos una función que calcule la suma de una lista de enteros:

```
fun {SumList L}
  case L of X|L1 then X+{SumList L1}
  else 0 end
end
```

¿Qué pasa si invocamos `{SumList {Ents 0}}`? ¿Es una buena idea?

6. *Programación de alto orden.* En la sección 1.9 se explica cómo usar la programación de alto orden para calcular variaciones del triángulo de Pascal. El propósito de este ejercicio es explorar esas variaciones.

a) Caclule filas individuales usando substracción, multiplicación, y otras operaciones. ¿Por Qué al usar la multiplicación el triángulo resultante está lleno de ceros? Ensaye el siguiente tipo de multiplicación en su lugar:

```
fun {Mul1 X Y} (X+1)*(Y+1) end
```

Introducción a los conceptos de programación

¿Cómo es la fila 10 cuando se calcula con `Mul1`?

b) La siguiente instrucción cíclica calculará y mostrará diez filas al tiempo:

```
for I in 1..10 do {Browse {PascalGenérico Op I}} end
```

Utilice esta instrucción para explorar más fácilmente las variaciones.

7. *Estado explícito*. Este ejercicio compara variables y celdas. Presentamos dos fragmentos de código. El primero utiliza variables:

```
local X in
X=23
local X in
X=44
end
{Browse X}
end
```

El segundo utiliza celdas:

```
local X in
X={NewCell 23}
X:=44
{Browse @X}
end
```

En el primero, el identificador `X` se refiere a dos variables diferentes. En el segundo, `X` se refiere a una celda. ¿Qué muestra `Browse` en cada fragmento? Explique.

8. *Estado explícito y funciones*. Este ejercicio investiga cómo usar celdas junto con funciones. Definamos una función `{Acumular N}` que acumule todas sus entradas, i.e., que acumule la suma de todos los argumentos de todas las invocaciones. Este es un ejemplo:

```
{Browse {Acumular 5}}
{Browse {Acumular 100}}
{Browse {Acumular 45}}
```

Esto debería mostrar 5, 105, y 150, suponiendo que el acumulador contenía cero al principio. Esta es una manera errada de escribir `Acumular`:

```
declare
fun {Acumular N}
Acc in
Acc={NewCell 0}
Acc:=@Acc+N
@Acc
end
```

¿Qué es lo que está mal con esta definición? ¿Cómo la corregiría?

9. *Almacén de memoria*. Este ejercicio explora otra forma de introducir estado: un almacén de memoria. El almacén de memoria puede ser usado para desarrollar una versión mejorada de `PascalRápido` que recuerde las filas previamente calculadas.

a) Un almacén de memoria es similar a la memoria de un computador. Él cuenta con una serie de celdas de memoria, numeradas de 1 hasta el máximo utilizado hasta el momento. Existen cuatro funciones sobre almacenes de

memoria¹⁰: `NewStore` crea un almacén nuevo, `Put` coloca un valor nuevo en una celda de memoria, `Get` consigue el valor actual almacenado en una celda de memoria, y `Size` devuelve el número de celdas usadas hasta el momento. Por ejemplo:

```
declare
S={NewStore}
{Put S 2 [22 33]}
{Browse {Get S 2}}
{Browse {Size S}}
```

Esto almacena `[22 33]` en la celda de memoria 2, muestra `[22 33]`, y luego muestra 1. Cargue en el sistema Mozart el almacén de memoria como está definido en el archivo de suplementos del sitio Web del libro. Utilice la interfaz interactiva para entender cómo funciona el almacén.

b) Ahora utilice el almacén de memoria para escribir una versión mejorada de `PascalRápido`, llamada `PascalMásRápido`, que recuerde las filas calculadas previamente. Si una invocación requiere una de estas filas, entonces la función puede devolverla directamente sin tener que volverla a calcular. Esta técnica es a veces llamada anotación pues la función “toma nota” de su trabajo previo.¹¹ Esto mejora el desempeño. Miremos cómo funciona:

- Primero cree un almacén `S` disponible para `PascalMásRápido`.
- Para la invocación `{PascalMásRápido N}`, llame `m` el número de filas almacenadas en `S`, i.e., las filas 1 hasta `m` están en `S`.
- Si $n > m$, entonces calcule las filas $m + 1$ hasta n y almacénelas en `S`.
- Devuelva la enésima fila buscándola en `S`.

Visto desde fuera, `PascalMásRápido` se comporta idénticamente a `PascalRápido` solo que es más rápida.

c) Hemos presentado el almacén de memoria como una biblioteca. Resulta que el almacén de memoria se puede definir utilizando una celda de memoria. Ahora haremos un bosquejo de cómo hacerlo y usted escribe las definiciones. La celda guarda el contenido del almacén como una lista de la forma `[N1|X1 ... Nn|Xn]`, donde el cons `Ni|Xi` significa que la celda número `Ni` contiene `Xi`. Esto significa que los almacenes de memoria, mientras sean convenientes, no introducen ningún poder expresivo adicional sobre las celdas de memoria.

d) En la sección 1.13 se define un objeto contador. Cambie su implementación del almacén de memoria de manera que utilice este contador para llevar la cuenta del tamaño del almacén.

10. *Estado explícito y concurrencia.* En la sección 1.15 se presenta un ejemplo de utilización de una celda para almacenar un contador que es incrementado por dos

10. Nota del traductor: puesto que se utilizarán estas definiciones del sitio Web del libro, no se traducen estos nombres de procedimientos.

11. Nota del traductor: *Memoization*, en inglés.

hilos.

- a) Ejecute este ejemplo varias veces. ¿Qué resultados consigue? ¿El resultado fue 1 siempre? ¿Por Qué podría pasar esto?
- b) Modifique el ejemplo agregando invocaciones a `Delay` en cada hilo. Así cambiamos la intercalación de los hilos sin cambiar los cálculos que ellos hacen. ¿Puede usted idear un esquema de manera que el resultado sea siempre 1?
- c) En la sección 1.16 se presenta una versión del contador que nunca da 1 como resultado. ¿Qué pasa si usted utiliza la técnica de los retrasos para tratar de conseguir un 1 de todas maneras?

I MODELOS GENERALES DE COMPUTACIÓN

2

El modelo de computación declarativa

Non sunt multiplicanda entia praeter necessitatem.

No multiplique las entidades más allá de lo que sea necesario.

– La cuchilla de afeitar de Ockham, nombrada así en honor a William Ockham (1285?–1347/49)

La programación abarca tres cosas:

- Primero, un modelo de computación, que es un sistema formal que define un lenguaje y cómo se ejecutan las frases del lenguaje (e.g., expresiones y declaraciones) por parte de una máquina abstracta. Para este libro, estamos interesados en modelos de computación que sean útiles e intuitivos para los programadores. Esto será más claro cuando definamos el primer modelo más adelante en este capítulo.
- Segundo, un conjunto de técnicas de programación y principios de diseño utilizados para escribir programas en el lenguaje del modelo de computación. Algunas veces llamaremos a esto modelo de programación. Un modelo de programación siempre se construye sobre un modelo de computación.
- Tercero, un conjunto de técnicas de razonamiento que le permitan razonar sobre los programas, para incrementar la confianza en que se comportan correctamente, y para calcular su eficiencia.

La anterior definición de modelo de computación es muy general. No todos los modelos de computación definidos de esta manera serán útiles para los programadores. ¿Qué es un modelo de computación razonable? Intuitivamente, diremos que un modelo razonable es aquel que se puede usar para resolver muchos problemas, que cuenta con técnicas de razonamiento prácticas y sencillas, y que se puede implementar eficientemente. Diremos más sobre esta pregunta más adelante. El primer y más sencillo modelo de computación que estudiaremos es la programación declarativa. Por ahora, la definimos como la evaluación de funciones sobre estructuras de datos parciales. Alguna veces esto es llamado programación sin estado, en contraposición a programación con estado (también llamada programación imperativa) la cual es explicada en el capítulo 6.

El modelo declarativo de este capítulo es uno de los modelos de computación más importantes. Abarca las ideas centrales de los dos principales paradigmas declarativos, a saber, programación funcional y lógica. El modelo abarca la programación con funciones sobre valores completos, como en Scheme y Standard ML. También

El modelo de computación declarativa

abarca la programación lógica determinística, como en Prolog cuando no se usa la búsqueda. Y finalmente, este modelo puede hacerse concurrente sin perder sus buenas propiedades (ver capítulo 4).

La programación declarativa es un área rica—cuenta con la mayor parte de las ideas de los modelos de computación más expresivos, al menos en una forma embrionaria. La presentamos por lo tanto en dos capítulos. En este capítulo se definen el modelo de computación y un lenguaje práctico basado en él. El próximo capítulo, capítulo 3, presenta las técnicas de programación de este lenguaje. Los capítulos posteriores enriquecen el modelo básico con muchos conceptos. Algunos de los más importantes son manejo de excepciones, concurrencia, componentes (para programación en grande), capacidades (para encapsulación y seguridad), y estado (conduciendo a objetos y clases). En el contexto de la concurrencia, hablaremos sobre flujo de datos, ejecución perezosa, paso de mensajes, objetos activos, monitores, y transacciones. También hablaremos sobre diseño de interfaces de usuario, distribución (incluyendo tolerancia a fallos), y restricciones (incluyendo búsqueda).

Estructura del capítulo

Este capítulo consta de ocho secciones:

- En la sección 2.1 se explica cómo definir la sintaxis y la semántica de lenguajes de programación prácticos. La sintaxis se define por medio de una gramática independiente del contexto extendida con restricciones del lenguaje. La semántica se define en dos etapas: traduciendo el lenguaje práctico en un lenguaje núcleo sencillo y definiendo luego la semántica del lenguaje núcleo. Estas técnicas, que serán usadas a través de todo el libro, se utilizan en este capítulo para definir el modelo de computación declarativa.
- En las siguientes tres secciones se definen la sintaxis y la semántica del modelo declarativo:
 - En la sección 2.2 se presentan las estructuras de datos: el almacén de asignación única y sus contenidos, los valores parciales y las variables de flujo de datos.
 - En la sección 2.3 se define la sintaxis del lenguaje núcleo.
 - En la sección 2.4 se define la semántica del lenguaje núcleo en términos de una máquina abstracta sencilla. La semántica se diseña para que sea intuitiva y permita razonamientos sencillos sobre corrección y complejidad.
- En la sección 2.5 se utiliza la máquina abstracta para explorar el comportamiento de las computaciones en memoria. Y le damos una mirada a la optimización de última invocación y al concepto de ciclo de vida de la memoria.
- En la sección 2.6 se define un lenguaje de programación práctico sobre un lenguaje núcleo.
- En la sección 2.7 se extiende el modelo declarativo con manejo de excepciones,

lo cual permite a los programas manejar situaciones impredecibles y excepcionales.

- En la sección 2.8 se presentan unos pocos temas avanzados para que los lectores interesados profundicen en su entendimiento del modelo.

2.1. Definición de lenguajes de programación prácticos

Los lenguajes de programación son mucho más simples que los lenguajes naturales, pero aún así pueden tener una sintaxis, un conjunto de abstracciones, y unas bibliotecas, sorprendentemente ricos. Esto es especialmente cierto para los lenguajes que son usados para resolver problemas del mundo real; a estos lenguajes los llamamos lenguajes prácticos. Un lenguaje práctico es como la caja de herramientas de un mecánico experimentado: hay muchas herramientas diferentes para muchos propósitos distintos y todas tienen una razón para estar allí.

En esta sección se fijan las bases del resto del libro explicando cómo se presentará la sintaxis (“gramática”) y la semántica (“significado”) de los lenguajes de programación prácticos. Con estos fundamentos, estaremos listos para presentar el primer modelo de computación del libro, a saber, el modelo de computación declarativa. Continuaremos usando estas técnicas a lo largo del libro para definir modelos de computación.

2.1.1. Sintaxis de un lenguaje

La sintaxis de un lenguaje define cómo son los programas legales, i.e., los programas que se pueden ejecutar efectivamente. En este punto no importa qué es lo que realmente hacen los programas. Esto lo define la semántica y será tratado en la sección 2.1.2.

Gramáticas

Una gramática es un conjunto de reglas que definen cómo construir ‘frases’ a partir de ‘palabras’. Las gramáticas se pueden usar para lenguajes naturales, como el Inglés o el Sueco, así como para lenguajes artificiales, como los lenguajes de programación. En los lenguajes de programación, las ‘frases’ se llaman normalmente ‘declaraciones’ y las ‘palabras’ se llaman normalmente ‘lexemas’¹. Así como las palabras están hechas de letras, los *lexemas* están hechos de caracteres. Esto nos define dos niveles de estructura:

$$\begin{array}{lcl} \text{declaración ('frase')} & = & \text{secuencia de lexemas ('palabras')} \\ \text{lexema ('palabra')} & = & \text{secuencia de caracteres ('letras')} \end{array}$$

1. Nota del traductor:*tokens* en inglés.

El modelo de computación declarativa

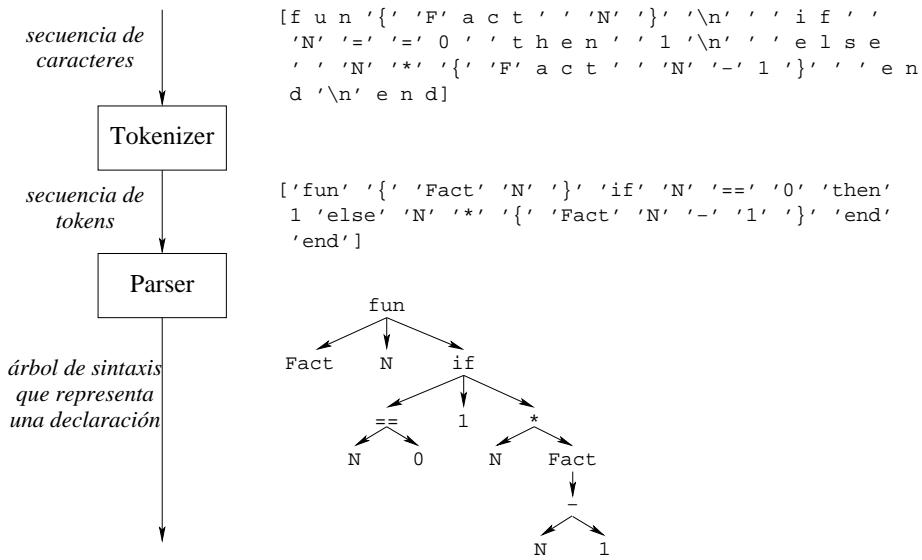


Figura 2.1: Pasando de caracteres a declaraciones.

Las gramáticas son útiles para definir tanto declaraciones como *lexemas*. La figura 2.1 presenta un ejemplo para mostrar cómo los caracteres de entrada se transforman en una declaración. El ejemplo en la figura es la definición de Fact:

```
fun {Fact N}
  if N==0 then 1
  else N*{Fact N-1} end
end
```

La entrada es una secuencia de caracteres donde `' '` representa el espacio en blanco y `'\n'` representa el salto de línea. La entrada se transforma primero en una secuencia de *lexemas* y luego en un árbol de sintaxis. La sintaxis de ambas secuencias en la figura es compatible con la sintaxis de listas que usamos a lo largo del libro. Mientras que las secuencias son “planas”, el árbol de sintaxis captura la estructura de la declaración. Un programa que recibe una secuencia de caracteres y devuelve una secuencia de *lexemas* se llama un analizador léxico. Un programa que recibe una secuencia de *lexemas* y devuelve un árbol de sintaxis se llama un analizador sintáctico.

La notación de Backus-Naur extendida

Una de las notaciones más comunes para definir gramáticas es la notación de Backus-Naur extendida (EBNF, por sus siglas en inglés)², que lleva el nombre de sus

2. Nota del traductor: EBNF viene de *Extended Backus-Naur form*.

inventores John Backus y Peter Naur. La notación EBNF distingue entre símbolos terminales y símbolos no-terminales. Un símbolo terminal es sencillamente un *lexema*. Un símbolo no-terminal representa una secuencia de *lexemas*. Los símbolos no-terminales se definen por medio de una regla gramatical, la cual define cómo expandirlo en *lexemas*. Por ejemplo, la regla siguiente define el no-terminal $\langle \text{digit} \rangle$:

$$\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

La regla indica que $\langle \text{digito} \rangle$ representa uno de los diez *lexemas* 0, 1, …, 9. El símbolo “|” se lee como “o”; significa que se puede escoger una de las alternativas. Las reglas gramáticas pueden, en sí mismas, hacer referencia a otros símbolos no-terminales. Por ejemplo, podemos definir el símbolo no-terminal $\langle \text{ent} \rangle$ para describir cómo se escriben números enteros positivos:

$$\langle \text{ent} \rangle ::= \langle \text{digito} \rangle \{ \langle \text{digito} \rangle \}$$

Esta regla indica que un entero es un dígito seguido de cualquier número de dígitos, incluso de ninguno. Los corchetes “{ … }” se usan para denotar la repetición de lo que esté encerrado entre ellos, cualquier número de veces, incluso ninguna.

Cómo leer gramáticas

Para leer una gramática, empiece con cualquier símbolo no-terminal, por ejemplo $\langle \text{ent} \rangle$. Leer la regla gramatical correspondiente, de izquierda a derecha, produce una secuencia de *lexemas* de acuerdo al esquema siguiente:

- Cada que se encuentra un símbolo terminal, se añade a la secuencia.
- Para cada símbolo no-terminal que se encuentre, se lee su regla gramatical y se reemplaza el no-terminal por la secuencia de *lexemas* en que el no-terminal se expande.
- Cada vez que aparezca una escogencia (con |), tome una de las alternativas.

La gramática puede ser usada tanto para verificar que una declaración es legal como para generar declaraciones.

Gramáticas independientes del contexto y sensitivas al contexto

Llamamos lenguaje formal, o simplemente lenguaje, a cualquier conjunto bien definido de declaraciones. Por ejemplo, el conjunto de todas las declaraciones que se pueden generar con una gramática y un símbolo no-terminal es un lenguaje. Las técnicas para definir gramáticas se pueden clasificar de acuerdo a su expresividad, i.e., qué tipos de lenguajes pueden ellas generar. Por ejemplo, la notación EBNF presentada antes define una clase de gramáticas llamadas gramáticas independientes del contexto. Se llaman así debido a que la expansión de un no-terminal, e.g., $\langle \text{digito} \rangle$, es siempre la misma sin importar el contexto donde se use.

Para la mayoría de los lenguajes prácticos, normalmente no existe una gramática

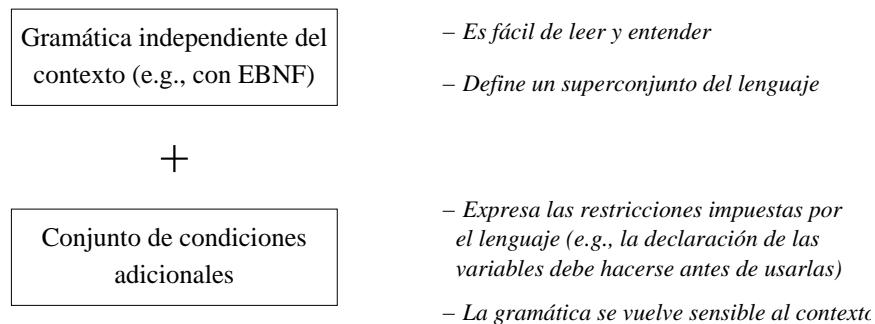


Figura 2.2: El enfoque independiente del contexto para la sintaxis de un lenguaje.



Figura 2.3: Ambigüedad en una gramática independiente del contexto.

independiente del contexto que genere todos los programas legales y solamente estos. Por ejemplo, en muchos lenguajes una variable debe ser declarada antes de ser utilizada. Esta condición no se puede expresar en una gramática independiente del contexto debido a que el no-terminal que representa la variable solo podría usarse con variables ya declaradas. Esta es una dependencia del contexto. Una gramática que contiene un símbolo no-terminal cuya utilización depende del contexto donde se está usando se llama una gramática sensitiva al contexto.

La sintaxis de la mayoría de los lenguajes de programación prácticos se define en consecuencia en dos partes (ver figura 2.2): como una gramática independiente del contexto complementada con un conjunto de condiciones adicionales impuestas por el lenguaje. Las gramáticas independientes del contexto son preferidas en lugar de algunas notaciones más expresivas debido a su facilidad para ser leídas y entendidas. Además, tienen una propiedad de localidad importante: un símbolo no-terminal se puede entender examinando solamente las reglas que lo definen; las reglas que lo usan (posiblemente mucho más numerosas) se pueden ignorar. La gramática independiente del contexto se complementa con la imposición de un conjunto de condiciones adicionales, como la restricción declarar-antes-de-usar que se impone a las variables. Al tomar estas condiciones en cuenta, se tiene una gramática sensible al contexto.

Ambigüedad

Las gramáticas independientes del contexto pueden ser ambiguas, i.e., puede haber varios árboles de sintaxis que correspondan a una misma secuencia de *lexemas*. Por ejemplo, la siguiente es una gramática sencilla para expresiones aritméticas con suma y multiplicación:

```
<exp> ::= <ent> | <exp> <op> <exp>
<op> ::= + | *
```

La expresión $2*3+4$ tiene dos árboles de sintaxis, dependiendo de cómo se leen las dos ocurrencias de $\langle \text{exp} \rangle$. La figura 2.3 muestra los dos árboles. En el uno, el primer $\langle \text{exp} \rangle$ es 2 y el segundo $\langle \text{exp} \rangle$ es $3+4$. En el otro, el primero y el segundo son $2*3$ y 4, respectivamente.

Normalmente, la ambigüedad es una propiedad indeseable en una gramática pues no permite distinguir con claridad cuál programa se escribió. En la expresión $2*3+4$, los dos árboles de sintaxis producen resultados diferentes cuando se evalúan: uno da 14 (el resultado de calcular $2*(3+4)$) y el otro da 10 (el resultado de calcular $(2*3)+4$). Algunas veces las reglas gramaticales se pueden reescribir para remover la ambigüedad, pero esto puede hacer que las reglas se vuelvan más complicadas. Un enfoque más conveniente consiste en agregar condiciones adicionales. Estas condiciones restringen al analizador sintáctico al punto que solo un árbol de sintaxis es posible. Decimos que éstas eliminan la ambigüedad de la gramática.

Para las expresiones con operadores binarios tales como las expresiones aritméticas presentadas atrás, el enfoque típico consiste en agregar dos condiciones, precedencia y asociatividad:

- La precedencia es una condición sobre una expresión con operadores diferentes, como $2*3+4$. A cada operador se le asigna un nivel de precedencia. Los operadores con altos niveles de precedencia se colocan tan profundo como sea posible en el árbol de sintaxis, i.e., tan lejos como sea posible de la raíz. Si * tiene un nivel de precedencia más alto que +, entonces el árbol de sintaxis correspondiente al cálculo $(2*3)+4$ es el escogido. Si * está a mayor profundidad que + en el árbol, entonces decimos que * tiene prioridad sobre +.
- La asociatividad es una condición sobre una expresión con el mismo operador, como $2-3-4$. En este caso, la precedencia no es suficiente para eliminar la ambigüedad pues todos los operadores tienen la misma precedencia. Tenemos que elegir entre los árboles $(2-3)-4$ y $2-(3-4)$. La asociatividad determina si el operador es asociativo hacia la izquierda o hacia la derecha. Si la asociatividad de - se define hacia la izquierda, entonces el árbol escogido es $(2-3)-4$. En cambio, si la asociatividad de - se define hacia la derecha, entonces el árbol escogido es $2-(3-4)$.

La precedencia y la asociatividad son suficientes para eliminar la ambigüedad de todas las expresiones definidas con operadores. El apéndice C presenta la precedencia y la asociatividad de todos los operadores utilizados en el libro.

Notación sintáctica utilizada en el libro

En este capítulo y en el resto del libro, se introduce cada tipo de datos nuevo y cada construcción nueva del lenguaje con un pequeño diagrama de sintaxis que muestra cómo se integran al lenguaje en su totalidad. El diagrama de sintaxis presenta las reglas gramaticales de una gramática de *lexemas*, sencilla e independiente del contexto. La notación se diseña cuidadosamente para satisfacer dos principios básicos:

- Todas las reglas gramaticales funcionan por sí mismas. Ninguna información posterior invalidará una regla gramatical. Es decir, nunca presentaremos una regla gramatical incorrecta solo por “simplificar” la presentación.
- Siempre será claro si una regla gramatical define completamente un símbolo no-terminal o si solo lo define parcialmente. Un definición parcial siempre termina con tres puntos “...”.

Todos los diagramas de sintaxis utilizados en este libro se agrupan y presentan en el apéndice C. Este apéndice también presenta la sintaxis léxica de los *lexemas*, i.e., la sintaxis de los *lexemas* en términos de caracteres. A continuación, un ejemplo de diagrama de sintaxis con dos reglas gramaticales que ilustran nuestra notación:

```
<declaración> ::= skip | <expresión> '=' <expresión> | ...
<expresión> ::= <variable> | <ent> | ...
```

Estas reglas definen parcialmente dos no-terminales, `<declaración>` y `<expresión>`. La primera regla dice que una declaración puede ser la palabra reservada `skip`, o dos expresiones separadas por el símbolo de igualdad `=`, o algo más. La segunda regla dice que una expresión puede ser una variable, un entero, o algo más. La barra vertical `|` establece una alternativa entre diferentes posibilidades en la regla gramatical. Para evitar confusiones con la sintaxis propia para escribir reglas gramaticales, algunas veces se encerrará entre comillas sencillas el símbolo que ocurre literalmente en el texto. Por ejemplo, el símbolo de igualdad se muestra como `'='`. Las palabras reservadas no son encerradas entre comillas sencillas, pues en ese caso no hay confusión posible.

A continuación se presenta un segundo ejemplo que presenta la notación restante:

```
<declaración> ::= if <expresión> then <declaración>
                  { elseif <expresión> then <declaración> }
                  [ else <declaración> ] end | ...
<expresión> ::= '[' { <expresión> } + ']' | ...
<etiqueta> ::= unit | true | false | <variable> | <átomo>
```

La primera regla define la declaración `if`. Esta contiene una secuencia opcional de cláusulas `elseif`, i.e., pueden aparecer cualquier número de veces, incluso ninguna. Esto se denota con los corchetes `{ ... }`. Y luego se tiene un cláusula `else` opcional,

i.e., puede ocurrir una vez o ninguna. Esto se denota con los paréntesis cuadrados [...]. La segunda regla define la sintaxis de listas explícitas. Estas deben contener por lo menos un elemento, i.e., [5 6 7] es válida pero [] no lo es (note el espacio que separa el paréntesis que abre [del que cierra]). Esto se denota con { ... }+. La tercera regla define la sintaxis de las etiquetas de registros. La tercera regla es un definición completa pues no aparecen los tres puntos “...”. No existen sino estas cinco posibilidades y nunca se presentarán más.

2.1.2. Semántica de un lenguaje

La semántica de un lenguaje define lo que hace un programa cuando se ejecuta. Idealmente, la semántica debería ser definida con una estructura matemática simple que nos permita razonar sobre el programa (incluyendo su corrección, tiempo de ejecución, y utilización de memoria) sin introducir ningún detalle irrelevante. ¿Podemos lograr esto para un lenguaje de programación práctico, sin que la semántica se vuelva muy complicada? La técnica que usamos, la cual llamamos el enfoque del lenguaje núcleo, nos permite responder de manera afirmativa a esa pregunta.

Los lenguajes de programación modernos han evolucionado, por más de cinco décadas de experiencia, para lograr construir soluciones a problemas complejos del mundo real.³ Los programas modernos pueden ser bastante complejos, con tamaños medidos en millones de líneas de código, escritas por equipos grandes de programadores a lo largo de muchos años. En nuestro concepto, los lenguajes que escalan a este nivel de complejidad son exitosos en parte gracias a que modelan algunos aspectos esenciales sobre cómo construir programas complejos. En este sentido, estos lenguajes no son solo construcciones arbitrarias de la mente humana, Nos gustaría entonces entenderlos de una manera científica, i.e., explicando su comportamiento en términos de un modelo fundamental y sencillo. Esta es la motivación profunda detrás del enfoque del lenguaje núcleo.

El enfoque del lenguaje núcleo

Este libro utiliza el enfoque del lenguaje núcleo para definir la semántica de los lenguajes de programación. En este enfoque, todas las construcciones del lenguaje se definen en términos de traducciones a un lenguaje básico, conocido como el lenguaje núcleo. Este enfoque consiste de dos partes (ver la figura 2.4):

- Primero, definir un lenguaje muy sencillo, llamado el lenguaje núcleo. Este lenguaje debe facilitar el razonar sobre él y debe ser fiel a la eficiencia en tiempo y espacio de la implementation. Juntos, el lenguaje núcleo y las estructuras de datos

3. La medida de cinco décadas es algo arbitraria. La medimos desde el primer trabajo funcional en un computador, el Manchester Mark I. De acuerdo a documentos de laboratorio, se corrió su primer programa el 21 de Junio de 1948 [159].

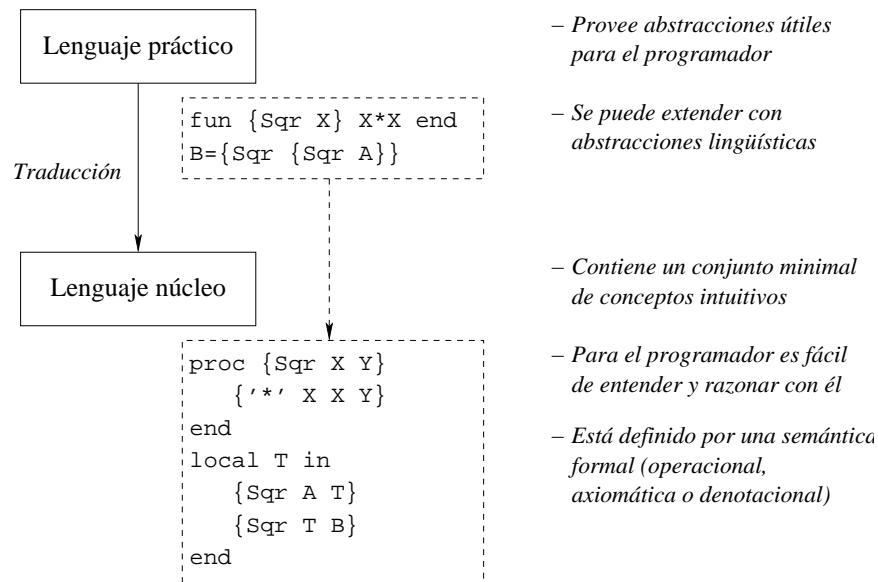


Figura 2.4: El enfoque del lenguaje núcleo en semántica.

que manipula, forman el modelo de computación núcleo.

- Segundo, definir un esquema de traducción de todo el lenguaje de programación al lenguaje núcleo. Cada construcción gramatical en el lenguaje completo se traduce al lenguaje núcleo. La traducción debe ser tan simple como sea posible. Hay dos clases de traducciones, a saber, las abstracciones lingüísticas y el azúcar sintáctico. Ambas se explican más adelante.

El enfoque del lenguaje núcleo se usa a lo largo del libro. Cada modelo de computación tiene su lenguaje núcleo, el cual se construye agregando un concepto nuevo sobre su predecesor. El primer lenguaje núcleo, el cual se presenta en este capítulo, se llama el lenguaje núcleo declarativo. Muchos otros lenguajes núcleo se presentan más adelante en el libro.

Semántica formal

El enfoque del lenguaje núcleo nos permite definir la semántica del lenguaje núcleo en la forma que queramos. Existen cuatro enfoques ampliamente utilizados para definir la semántica de un lenguaje:

- Una semántica operacional muestra cómo se ejecuta una declaración en términos de una máquina abstracta. Este enfoque siempre funciona bien, pues finalmente todos los lenguajes se ejecutan sobre un computador.
- Una semántica axiomática define la semántica de una declaración como la relación

entre el estado de entrada (la situación antes de ejecutarse la declaración) y el estado de salida (la situación después de ejecutarse la declaración). Esta relación se presenta en la forma de una afirmación lógica. Esta es una buena manera de razonar sobre secuencias de declaraciones, pues la afirmación de salida de cada declaración es la afirmación de entrada de la declaración siguiente. Esta semántica funciona bien en modelos con estado, ya que un estado es una secuencia de valores. En la sección 6.6 se presenta una semántica axiomática del modelo con estado del capítulo 6.

- Una semántica denotacional define una declaración como una función sobre un dominio abstracto. Este enfoque funciona bien para modelos declarativos, aunque también puede ser aplicado a otros modelos. En cambio, se vuelve complicado cuando se aplica a lenguajes concurrentes. En las secciones 2.8.1 y 4.9.2 se explica la programación funcional, la cual es particularmente cercana a la semántica denotacional.
- Una semántica lógica define una declaración como un modelo de una teoría lógica. Este enfoque funciona bien para los modelos de computación declarativa y relacional, pero es difícil de aplicar a otros modelos. En la sección 9.3 (en CTM) se presenta una semántica lógica de los modelos de computación declarativa y relacional.

Gran parte de la teoría que fundamenta estos diferentes enfoques de semántica interesa primordialmente a los matemáticos, y no a los programadores. Está por fuera del alcance de este libro presentar esa teoría. La semántica formal que presentamos en este libro es principalmente una semántica operacional, la cual definimos para cada modelo de computación. Esta semántica es suficientemente detallada para poderla utilizar para razonar sobre corrección y complejidad, y suficientemente abstracta para evitar llegar a los detalles irrelevantes. En el capítulo 13 (en CTM) se agrupan todas estas semánticas operacionales en un solo formalismo con una notación compacta y legible.

A lo largo del libro, presentamos una semántica informal para cada construcción nueva del lenguaje y frecuentemente razonamos informalmente sobre los programas. Estas presentaciones informales siempre están basadas en la semántica operacional.

Abstracción lingüística

Tanto los lenguajes de programación como los lenguajes naturales pueden evolucionar para satisfacer sus necesidades. Cuando utilizamos un lenguaje de programación, en algún momento podemos sentir la necesidad de extender el lenguaje, i.e., agregar una nueva construcción lingüística. Por ejemplo, el modelo declarativo de este capítulo no tiene construcciones para ciclos. En la sección 3.6.3 se define una construcción **for** para expresar cierta clase de ciclos que son útiles cuando se escriben programas declarativos. La construcción nueva es a la vez una abstracción y un elemento nuevo de la sintaxis del lenguaje. Por eso la llamamos abstracción lingüística. Un lenguaje de programación práctico posee muchas abstracciones

lingüísticas.

Una abstracción lingüística se define en dos fases. Primero, se define una nueva construcción gramatical. Segundo, se define su traducción al lenguaje núcleo. El lenguaje núcleo no se cambia. Este libro presenta muchos ejemplos de abstracciones lingüísticas útiles, e.g., funciones (**fun**), ciclos (**for**), funciones perezosas (**fun lazy**), clases (**class**), candados (**lock**), y otros.⁴ Algunas de ellas hacen parte del sistema Mozart; las otras se pueden añadir a él con el generador de analizadores sintácticos [93] gump. El uso de esta herramienta está fuera del alcance de este libro.

Algunos lenguajes proveen facilidades para programar directamente las abstracciones lingüísticas en el lenguaje. Un ejemplo simple y poderoso es el macro en Lisp. Un macro en Lisp es como una función que genera código Lisp cuando se ejecuta. Los macros en Lisp y sus sucesores han tenido un éxito extraordinario, en parte gracias a la sintaxis sencilla de Lisp. Lisp cuenta con soporte incorporado para macros, tales como quote (convertir una expresión Lisp en datos) y backquote (realiza lo contrario, a una expresión previamente convertida en dato). Para una discusión detallada de los macros de Lisp e ideas relacionadas, referimos al lector a cualquier buen libro sobre Lisp [59, 162].

Un ejemplo sencillo de abstracción lingüística es la función, distinguida con la palabra reservada **fun**. Ésta se explica en la sección 2.6.2. Ya hemos programado con funciones en el capítulo 1. Pero el lenguaje núcleo de este capítulo solo cuenta con procedimientos, pues todos sus argumentos son explícitos y puede tener múltiples salidas. Hay otras razones más profundas para haber escogido procedimientos; éstas se explican más adelante en este capítulo. Sin embargo, como las funciones son tan útiles, las añadimos como abstracción lingüística.

Definimos una sintaxis tanto para definir funciones como para invocarlas, y una traducción de esta sintaxis en definiciones de procedimientos e invocaciones a ellos. La traducción nos permite responder a todas las preguntas sobre invocación de funciones. Por ejemplo, ¿que significa exáctamente {F1 {F2 X} {F3 Y}} (invocación anidada de funciones)? ¿El orden en que se hacen estas invocaciones está definido? Si es así, ¿cuál es ese orden? Existen muchas alternativas. Algunos lenguajes dejan indefinido el orden de evaluación de los argumentos en una invocación, pero definen que los argumentos se evalúan antes que el cuerpo de la función invocada. Otros lenguajes definen que un argumento se evalúa solo cuando su resultado se necesita, no antes. Entonces, ni siquiera algo tan sencillo como una invocación anidada de funciones tiene una semántica obvia. La traducción clarifica la semántica escogida.

Las abstracciones lingüísticas sirven para algo más que solo incrementar la expresividad de un programa. También sirven para mejorar otras propiedades como corrección, seguridad, y eficiencia. Al ocultar la implementación de la abstracción al programador, el soporte lingüístico impide el uso de la abstracción en forma

4. Compuertas lógicas (**gate**) para descripciones de circuitos, buzones de correo (**receive**) para concurrencia por paso de mensajes, y currificación y listas por comprensión como en los lenguajes de programación modernos, cf. Haskell.

errada. El compilador puede utilizar esta información para producir un código más eficiente.

Azúcar sintáctico

A menudo conviene proveer notación abreviada para modismos que ocurren con frecuencia. Esta notación, llamada azúcar sintáctico, hace parte de la sintaxis del lenguaje y se define con reglas gramaticales. El azúcar sintáctico es análogo a una abstracción lingüística en el sentido que su significado se define precisamente traduciéndolo al lenguaje núcleo. Pero no se debe confundir con una abstracción lingüística: el azúcar sintáctico no provee una abstracción nueva, solo reduce el tamaño del programa y mejora su legibilidad.

Presentamos un ejemplo de azúcar sintáctico basado en la declaración **local**. Las variables locales siempre se definen utilizando la declaración **local x in ... end**. Cuando se necesita usar esta declaración dentro de otra, se hace conveniente contar con azúcar sintáctico que elimine el uso repetido de las palabras reservadas **local** y **end**. En lugar de

```
if N==1 then [1]
else
    local L in
    ...
end
end
```

podemos escribir

```
if N==1 then [1]
else L in
    ...
end
```

lo cual es más corto y legible que la notación completa. En la sección 2.6.1 se presentan otros ejemplos de azúcar sintáctico.

Diseño de lenguajes

Las abstracciones lingüísticas son una herramienta básica en el diseño de lenguajes. Ellas tienen un lugar natural en el ciclo de vida de una abstracción, el cual consta de tres fases. En la primera, definimos la abstracción sin soporte lingüístico, i.e., no existe aún sintaxis diseñada en el lenguaje para poder usarla fácilmente. En la segunda, en algún momento sospechamos que la abstracción es especialmente básica y útil, y decidimos proveerle un soporte lingüístico, convirtiéndola en una abstracción lingüística. Esta fase es exploratoria, i.e., no hay ningún compromiso en que la abstracción lingüística se vuelva parte del lenguaje. Si la abstracción lingüística es exitosa, i.e., simplifica los programas y le es útil a los programadores, entonces si se vuelve parte del lenguaje.

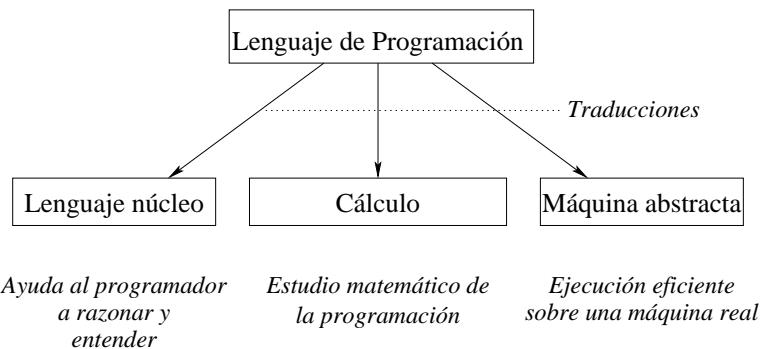


Figura 2.5: Enfoques de traducción en semántica de lenguajes.

Otros enfoques de traducción

El enfoque del lenguaje núcleo es un ejemplo de enfoque de traducción para definir la semántica de lenguajes, i.e., está basado en la traducción de un lenguaje en otro. La figura 2.5 muestra las tres formas en que el enfoque de traducción ha sido utilizado para definir lenguajes de programación:

- El enfoque del lenguaje núcleo, usado a lo largo de este libro, está dirigido al programador. Sus conceptos corresponden directamente a conceptos de programación.
- El enfoque de los cálculos está orientado al matemático. Como ejemplos están la máquina de Turing, el cálculo λ (fundamento de la programación funcional), la lógica de primer orden (fundamento de la programación lógica), y el cálculo π (para modelar concurrencia). Debido a su orientación hacia un estudio matemático formal, estos cálculos tienen la menor cantidad posible de elementos.
- El enfoque de la máquina abstracta está orientado al implementador. Los programas se traducen en una máquina idealizada tradicionalmente llamada *máquina abstracta* o *máquina virtual*.⁵ Es relativamente sencillo traducir código de una máquina idealizada en código de una máquina real.

Este libro utiliza el enfoque del lenguaje núcleo, ya que se enfoca en técnicas de programación prácticas. Los otros dos enfoques tienen el problema que cualquier programa real escrito en ellos está repleto de detalles técnicos con mecanismos

5. Estrictamente hablando, una máquina virtual es una simulación en *software* de una máquina real, ejecutándose sobre la máquina real, es decir casi tan eficiente como la máquina real. Logra su eficiencia ejecutando la mayoría de las instrucciones virtuales directamente como instrucciones reales. El precursor de este concepto fue IBM en el inicio de los años 1960 con el sistema operativo VM. Debido al éxito de Java, el cual usa el término “máquina virtual,” hoy en día también se usa ese término en el sentido de máquina abstracta.

del lenguaje. El enfoque del lenguaje núcleo evita este problema realizando una escogencia cuidadosa de los conceptos.

El enfoque de interpretación

Una alternativa a los enfoques de traducción es el enfoque de interpretación. La semántica del lenguaje se define por medio de un intérprete de éste. Las características nuevas del lenguaje se describen extendiendo el intérprete. Un intérprete es un programa escrito en un lenguaje L_1 que acepta programas escritos en otro lenguaje L_2 y los ejecuta. Este enfoque es utilizado por Abelson, Sussman, y Sussman [2]. En su caso, el intérprete es metacircular, i.e., L_1 y L_2 son el mismo lenguaje L . Agregar características nuevas al lenguaje, e.g., concurrencia y evaluación perezosa, produce un lenguaje nuevo L' el cual se implementa extendiendo el intérprete de L .

El enfoque de interpretación tiene la ventaja que presenta una implementación auto contenida de la implementación de las abstracciones lingüísticas. No utilizamos este enfoque en el libro, debido a que, en general, no preserva la complejidad en tiempo de ejecución de los programas (el número de operaciones realizadas en función del tamaño de la entrada). Una segunda dificultad es que los conceptos básicos interactúan con los otros en el intérprete lo cual los hace más difíciles de entender. El enfoque de traducción facilita el mantener los conceptos separados.

2.2. El almacén de asignación única

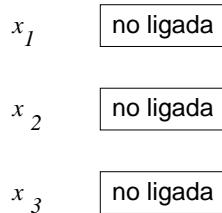
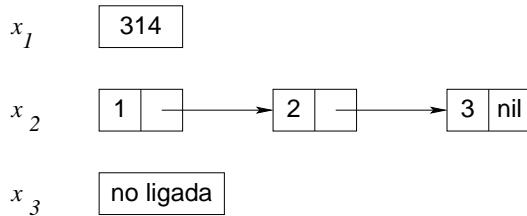
Nosotros introducimos el modelo declarativo explicando primero sus estructuras de datos. El modelo utiliza un almacén de asignación única, el cual consta de un conjunto de variables que inicialmente no están ligadas⁶ y que pueden ser ligadas a un único valor. La figura 2.6 muestra un almacén con tres variables no-ligadas x_1 , x_2 , y x_3 . Denotamos este almacén como $\{x_1, x_2, x_3\}$. Por ahora, supongamos que podemos usar como valores a los enteros, las listas y los registros. La figura 2.7 muestra un almacén donde x_1 está ligado al entero 314 y x_2 está ligado a la lista $[1 \ 2 \ 3]$. Denotamos este almacén como $\{x_1 = 314, x_2 = [1 \ 2 \ 3], x_3\}$.

2.2.1. Variables declarativas

Las variables en el almacén de asignación única se llaman variables declarativas. Usaremos este término donde pueda presentarse confusión con otro tipo de variables. Más adelante en el libro, estas variables se llamarán variables de flujo de datos, debido a su rol en la ejecución con flujos de datos.

Una vez ligada, una variable declarativa permanece ligada a lo largo de toda la

6. Las llamaremos variables no-ligadas.

**Figura 2.6:** Almacén de asignación única con tres variables no-ligadas.**Figura 2.7:** Dos de las variables están ligadas a valores.

computación y, además, es indistinguible de su valor. Esto significa que la variable puede ser usada en los cálculos como si fuera el valor. Hacer la operación $x + y$ es lo mismo que hacer $11 + 22$, si el almacén es $\{x = 11, y = 22\}$.

2.2.2. Almacén de valores

Un almacén en el cual todas las variables están ligadas a valores se llama almacén de valores. Otra forma de decir esto mismo es que un almacén de valores es una aplicación⁷ persistente⁸ de variables en valores. Un valor es una constante en el sentido matemático. Por ejemplo, el entero 314 es un valor. Los valores también pueden ser entidades compuestas, i.e., entidades que contienen uno o más valores. Por ejemplo, la lista [1 2 3] y el registro persona(nombre: "Jorge" edad:25) son valores. La figura 2.8 muestra un almacén de valores, donde x_1 está ligada al entero 314, x_2 está ligada a la lista [1 2 3], y x_3 está ligada al registro persona(nombre: "Jorge" edad:25). Los lenguajes funcionales tales como Standard ML, Haskell, y Scheme cuentan con un almacén de valores pues ellos calculan funciones sobre valores. (Los lenguajes orientados a objetos tales como Smalltalk, C++, y Java cuentan con un almacén de celdas, el cual consta de celdas cuyo contenido puede ser modificado.)

7. Nota del traductor: *mapping*, en inglés.

8. Que permanece en el tiempo.

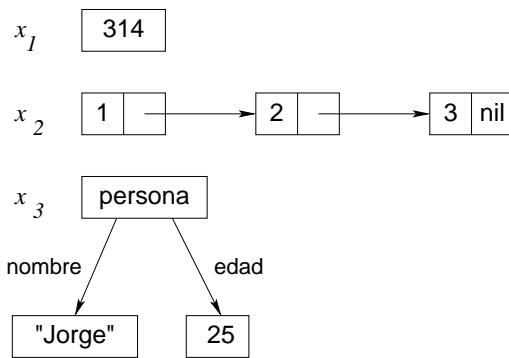


Figura 2.8: Un almacén de valores: todas las variables están ligadas a valores.

En este punto, un lector con alguna experiencia en programación se podrá preguntar por qué introducir un almacén de asignación única, cuando otros lenguajes trabajan con un almacén de valores o de celdas. Hay muchas razones. Una primera razón es que queremos poder calcular con valores parciales. Por ejemplo, un procedimiento puede recibir como argumento una variable no-ligada y ligarla, siendo éste el producto de su trabajo. Una segunda razón es la concurrencia declarativa, sujeto de estudio del capítulo 4. Este tipo de concurrencia se logra gracias al almacén de asignación única. Una tercera razón es que se necesita un almacén de asignación única para la programación relacional (lógica) y la programación por restricciones. Otras razones que tienen que ver con eficiencia (e.g., recursión de cola y listas de diferencias) se verán claras en el próximo capítulo.

2.2.3. Creación de valores

La operación básica de un almacén es la ligadura de una variable con un valor recientemente creado. Escribiremos esto como $x_i = valor$. Aquí x_i se refiere directamente a una variable en el almacén (no es el nombre textual de la variable en el programa) y $valor$ se refiere a un valor, e.g., 314 o $[1 \ 2 \ 3]$. Por ejemplo, la figura 2.7 muestra el almacén de la figura 2.6 después de dos ligaduras:

$$\begin{aligned} x_1 &= 314 \\ x_2 &= [1 \ 2 \ 3] \end{aligned}$$

La operación de asignación única $x_i = valor$ construye el *valor* en el almacén y luego liga la variable x_i a ese valor. Si la variable ya está ligada, la operación verificará si los dos valores son compatibles. Si no lo son, se señala un error (utilizando el mecanismo para manejo de excepciones; ver sección 2.7).

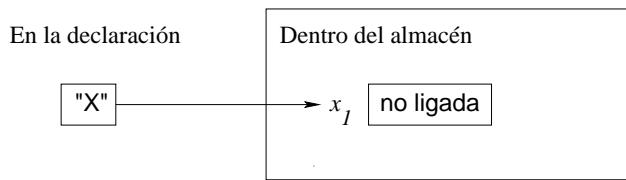


Figura 2.9: Un identificador de variable que referencia una variable no-ligada.

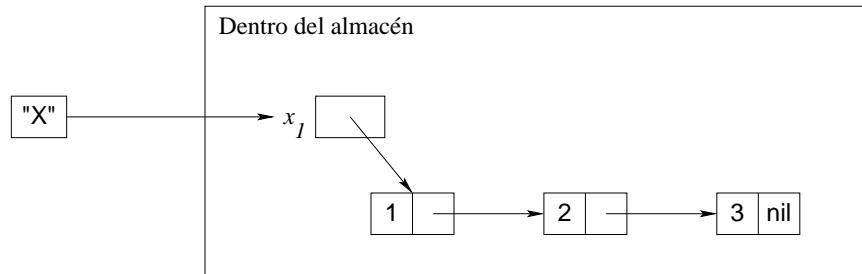


Figura 2.10: Un identificador de variable que referencia una variable ligada.

2.2.4. Identificadores de variables

Hasta aquí, hemos echado un vistazo a un almacén que contiene variables y valores, i.e., entidades del almacén con las que se realizan los cálculos. Sería agradable poder referirse desde afuera del almacén a las entidades del mismo. Este es precisamente el papel de los identificadores de variables. Un identificador de variable es un nombre textual que referencia una entidad del almacén desde afuera de éste. A la aplicación de identificadores de variables en entidades del almacén se le llama un ambiente.

Los nombres de variables en el código fuente de un programa son de hecho identificadores de variables. Por ejemplo, la figura 2.9 muestra un identificador "X" (la letra mayúscula X) que referencia la entidad x_1 del almacén. Esto corresponde al ambiente $\{x \rightarrow x_1\}$. Para referirnos a un identificador cualquiera, usaremos la notación $\langle x \rangle$. Si $\langle x \rangle$ representa x, entonces el ambiente $\{\langle x \rangle \rightarrow x_1\}$ es el mismo que acabamos de mencionar. Como lo veremos más adelante, los identificadores de variables y sus correspondientes entidades del almacén se agregan al ambiente por medio de las declaraciones **local** y **declare**.

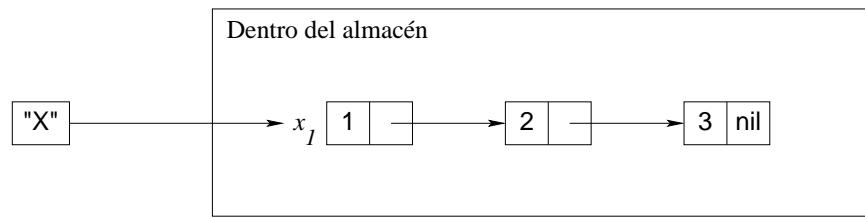


Figura 2.11: Un identificador de variable que referencia un valor.

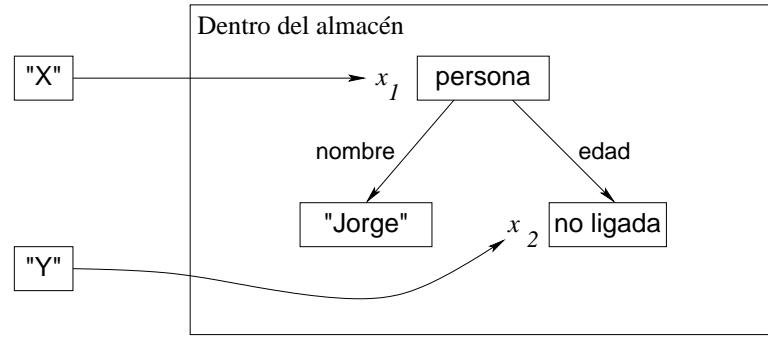


Figura 2.12: Un valor parcial.

2.2.5. Creación de valores con identificadores

Una vez ligada, una variable es indistinguible de su valor. La figura 2.10 muestra lo que pasa cuando x_1 se liga a [1 2 3] en la figura 2.9. Con el identificador de variable x, podemos escribir la ligadura como $x=[1 2 3]$. Este es el texto que un programador escribiría para expresar tal ligadura. Si deseamos referirnos a cualquier identificador, también podemos usar la notación $\langle x \rangle=[1 2 3]$. Para convertir esta notación en un programa legal se debe reemplazar $\langle x \rangle$ por un identificador.

El símbolo de igualdad “=” hace referencia a la operación de ligadura. Después que ésta se completa, el identificador “x” aún referencia a x_1 , pero éste está ligado ahora a [1 2 3]. Esto es indistinguible en la figura 2.11, donde x referencia directamente a [1 2 3]. Al proceso de seguir los enlaces hasta conseguir el valor asociado a un identificador se le llama desreferenciación. Para el programador esto es invisible.

2.2.6. Valores parciales

Un valor parcial es una estructura de datos que puede contener variables no-ligadas. La figura 2.12 muestra el registro $\text{persona}(\text{nombre}: \text{"Jorge"} \text{ edad}: x_2)$, referenciado por el identificador x. Este es un valor parcial pues contiene la variable

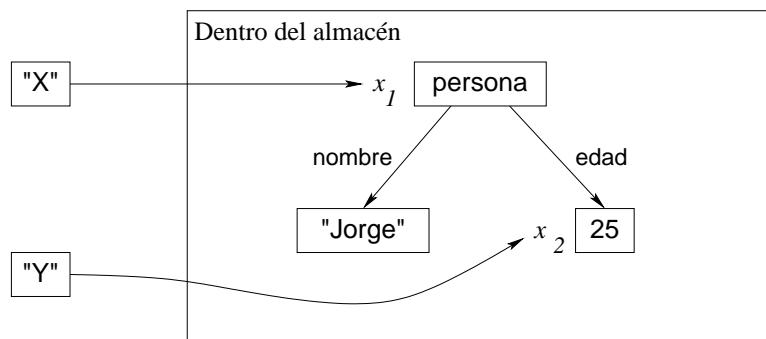


Figura 2.13: Un valor parcial sin variables no-ligadas, i.e., un valor completo.

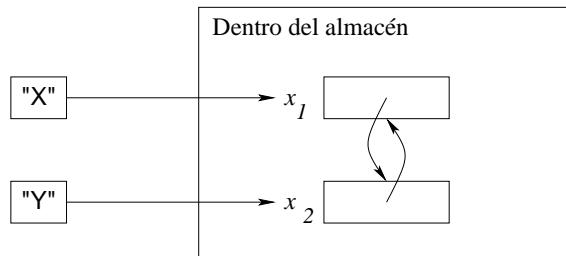


Figura 2.14: Dos variables ligadas entre sí.

no ligada x_2 . El identificador Y referencia a x_2 . La figura 2.13 muestra la situación después que x_2 se liga a 25 (a través de la operación $\text{Y}=25$). Ahora x_1 es un valor parcial sin variables no-ligadas, al cual llamaremos un valor completo. Una variable declarativa puede ser ligada a varios valores parciales siempre y cuando estos sean compatibles dos a dos. Decimos que un conjunto de valores parciales es compatible si las variables no ligadas que se encuentran en ellos se pueden ligar de manera que todos los valores sean iguales. Por ejemplo, `persona(edad:25)` y `persona(edad: x)` son compatibles (si ligamos x a 25), pero `persona(edad:25)` y `persona(edad:26)` no lo son.

2.2.7. Ligadura variable-variable

Una variable puede ser ligada a otra variable. Por ejemplo, considere dos variables no-ligadas x_1 y x_2 referenciadas por los identificadores X y Y . Después de hacer la operación $\text{X}=\text{Y}$, llegamos a la situación mostrada en la figura 2.14. Las dos variables x_1 y x_2 son iguales entre sí. En la figura se muestra esto haciendo que cada variable

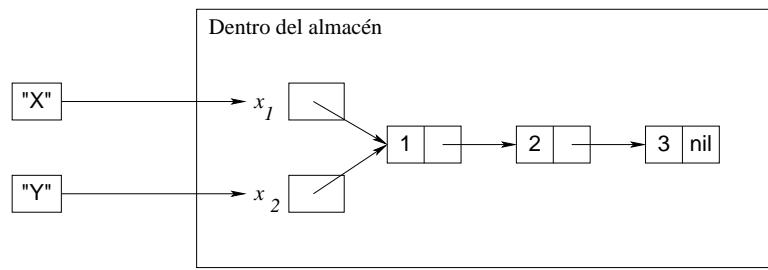


Figura 2.15: El almacén después de ligar una de las variables.

referencie a la otra. Decimos que $\{x_1, x_2\}$ conforman un conjunto de equivalencia.⁹ También escribimos esto como $x_1 = x_2$. Si fueran tres variables ligadas entre sí se escribiría $x_1 = x_2 = x_3$ o $\{x_1, x_2, x_3\}$. Si fueran dibujadas en una figura, estas variables formarían una cadena circular. Cuando una variable en un conjunto de equivalencia se liga, todas las variables ven la ligadura. La figura 2.15 muestra el resultado de hacer $x=[1 2 3]$.

2.2.8. Variables de flujo de datos

En el modelo declarativo la creación de una variable y su ligadura se realizan separadamente. ¿Qué pasa si tratamos de usar una variable antes de que sea ligada? Llamamos esto un error de uso de la variable. Algunos lenguajes crean y ligan las variables en una sola etapa, de manera que no ocurre este error de uso. Este es el caso de los lenguajes de programación funcional. Otros lenguajes permiten que la creación y la ligadura se hagan separadamente. En este caso, cuando hay un error de uso, se presentan las posibilidades siguientes:

1. La ejecución continúa y no se presenta ningún mensaje de error. El contenido de la variable es indefinido, i.e., es “basura”: lo que se encuentre en la memoria. Esto es lo que hace C++.
2. La ejecución continúa y no se presenta ningún mensaje de error. La variable inicia en un valor por defecto una vez declarada, e.g., en 0 para un entero. Esto es lo que hace Java para los campos en objetos y en estructuras de datos, tales como los arreglos. El valor por defecto depende del tipo de la variable.
3. La ejecución se detiene con un mensaje de error (o se lanza una excepción). Esto es lo que hace Prolog para las operaciones aritméticas.
4. La ejecución no es posible porque el compilador detecta que existe un posible camino de ejecución que utiliza una variable que no tiene un valor inicial. Esto es

9. Desde un punto de vista formal, las dos variables forman una clase de equivalencia con respecto a la igualdad.

El modelo de computación declarativa

lo que hace Java para las variables locales.

5. La ejecución se suspende y espera hasta que la variable sea ligada y luego continúa. Esto es lo que hace Oz para soportar programación por flujo de datos.

Estos casos se listan en orden, del menos al más agradable. El primer caso es muy malo, pues ejecuciones diferentes del mismo programa pueden dar resultados diferentes. Aún más, como el error no se señala, el programador no se da cuenta cuándo ocurre esto. El segundo caso es un poco mejor. Si el programa tiene un error de uso, entonces siempre se obtiene el mismo resultado, sin importar si es un resultado errado o no. De nuevo, el programador no se da cuenta cuándo ocurre esto.

El tercer y el cuarto casos pueden ser razonables en ciertas situaciones. En ambos casos, un programa con un error de uso señalará este hecho, ya sea en tiempo de ejecución o en tiempo de compilación. Esto es razonable en sistemas secuenciales, donde esto es realmente un error. El tercer caso no es razonable en un sistema concurrente, pues el resultado se vuelve no-determinístico: dependiendo de cómo se manejen los tiempos de ejecución, algunas veces se señala un error y otras veces no.

En el quinto caso, el programa se suspenderá hasta que la variable sea ligada y luego continuará. El modelo de computación del libro utiliza este caso. Esto no es razonable en un sistema secuencial porque el programa se suspendería para siempre. Pero sí es razonable en un sistema concurrente, donde es parte de la operación normal que un hilo ligue la variable de otro hilo. El quinto caso introduce un nuevo tipo de error de programa, a saber, una suspensión que espera por siempre. Por ejemplo, si un nombre de variable está mal escrito entonces nunca será ligado. Un buen depurador debe detectar cuándo ocurre esto.

Las variables declarativas que hacen que un programa se suspenda hasta que sean ligadas se llaman *variables de flujo de datos*.

El modelo declarativo utiliza variables de flujo de datos porque son supremamente útiles en programación concurrente, i.e., para programas con actividades que se ejecutan independientemente. Si realizamos dos operaciones concurrentes, digamos $A=23$ y $B=A+1$, entonces con el quinto caso siempre se ejecutará correctamente y producirá la respuesta $B=24$. No importa cuál operación, entre $A=23$ y $B=A+1$, se ejecute primero. Con los otros casos, no se puede garantizar la corrección. La propiedad de independencia de orden hace posible la concurrencia declarativa del capítulo 4 y es la razón central de por qué las variables de flujo de datos son una buena idea.

2.3. El lenguaje núcleo

El modelo declarativo define un lenguaje núcleo sencillo. Todos los programas en el modelo se pueden expresar en este lenguaje. Primero definimos la sintaxis y la semántica del lenguaje núcleo; luego, explicamos cómo construir un lenguaje completo soportado en el lenguaje núcleo.

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Invocación de procedimiento

Tabla 2.1: El lenguaje núcleo declarativo.

2.3.1. Sintaxis

La sintaxis del lenguaje núcleo se presenta en las tablas 2.1 y 2.2. La sintaxis ha sido cuidadosamente diseñada para que sea un subconjunto de la sintaxis del lenguaje completo, i.e., todas las declaraciones del lenguaje núcleo son declaraciones válidas del lenguaje completo.

Sintaxis de las declaraciones

En la tabla 2.1 se define la sintaxis de $\langle d \rangle$, el cual denota una declaración. En total son ocho declaraciones que se explicarán más adelante.

Sintaxis de los valores

En la tabla 2.2 se presenta la sintaxis de $\langle v \rangle$, el cual denota un valor. Existen tres tipos de expresiones que denotan valores, a saber, las que denotan números, registros, y procedimientos. En el caso de registros y patrones, todos los argumentos $\langle x \rangle_1, \dots, \langle x \rangle_n$ deben ser identificadores diferentes. Esto asegura que todas las ligaduras variable-variable se escriban como operaciones explícitas del núcleo.

Sintaxis de identificadores de variables

En la tabla 2.1 se utilizan los símbolos no terminales $\langle x \rangle$ y $\langle y \rangle$ para denotar un identificador de variable. También usaremos $\langle z \rangle$ para denotar identificadores. Existen dos formas de escribir un identificador de variable:

- Una letra mayúscula seguida de cero o más caracteres alfanuméricos (letras o dígi-

$\langle v \rangle$::=	$\langle \text{número} \rangle \mid \langle \text{registro} \rangle \mid \langle \text{procedimiento} \rangle$
$\langle \text{número} \rangle$::=	$\langle \text{ent} \rangle \mid \langle \text{flot} \rangle$
$\langle \text{registro} \rangle, \langle \text{patrón} \rangle$::=	$\langle \text{literal} \rangle$ $\langle \text{literal} \rangle (\langle \text{campo} \rangle_1: \langle x \rangle_1 \cdots \langle \text{campo} \rangle_n: \langle x \rangle_n)$
$\langle \text{procedimiento} \rangle$::=	proc { \$ $\langle x \rangle_1 \cdots \langle x \rangle_n$ } d end
$\langle \text{literal} \rangle$::=	$\langle \text{átomo} \rangle \mid \langle \text{bool} \rangle$
$\langle \text{campo} \rangle$::=	$\langle \text{átomo} \rangle \mid \langle \text{bool} \rangle \mid \langle \text{ent} \rangle$
$\langle \text{bool} \rangle$::=	true false

Tabla 2.2: Expresiones que denotan valores en el lenguaje núcleo declarativo.

tos o guión de subíndice:¹⁰ _), e.g., X, X1, or EstaEsUnaVariableLarga_Cierto.

- Cualquier secuencia de caracteres imprimibles encerrada entre dos caracteres ` (comilla inversa), e.g., `esta es una 25\\$variable!`.

Una definición precisa de la sintaxis de un identificador se presenta en el apéndice C.

Todas las variables recién declaradas inician no-ligadas (antes que cualquier declaración se ejecute). Todos los identificadores de variables deben ser declarados explícitamente.

2.3.2. Valores y tipos

Un tipo tipo o tipo de datos es un conjunto de valores junto con un conjunto de operaciones sobre esos valores. Un valor es “de un tipo” si pertenece al conjunto de valores correspondiente a ese tipo. El modelo declarativo es tipado en el sentido que cuenta con un conjunto de tipos bien definido, llamados tipos básicos. Por ejemplo, los programas pueden calcular con enteros o con registros, los cuales son todos de tipo entero o tipo registro, respectivamente. Cualquier intento de usar una operación con valores del tipo errado es detectada por el sistema y se lanzará una condición de error (ver sección 2.7). El modelo no impone ninguna otra restricción en el uso de los tipos.

Debido a que todo uso de los tipos se verifica, es imposible que un programa se comporte por fuera del modelo, i.e., que colapse debido a operaciones indefinidas sobre sus estructuras de datos internas. Sin embargo, todavía se podría lanzar una condición de error, e.g., división por cero. En el modelo declarativo, un programa que lanza una condición de error terminará inmediatamente. No existe nada en el modelo para manejar los errores. En la sección 2.7 extendemos el modelo declarativo con un concepto nuevo, las excepciones, para el manejo de errores. En el modelo extendido, los errores de tipo se pueden manejar dentro del modelo.

10. Nota del traductor: *underscore*, en inglés.

Además de los tipos básicos, los programas pueden definir sus propios tipos. Estos se llaman tipos abstractos de datos, o TADs¹¹. En el capítulo 3 y en capítulos posteriores se muestra cómo definir TADs. Hay otras clases de abstracción de datos, además de los TADs. En la sección 6.4 se presenta una mirada general a las diferentes posibilidades.

Tipos básicos

Los tipos básicos del modelo declarativo son los números (enteros y flotantes), los registros (incluyendo átomos, booleanos, tuplas, listas, y cadenas), y los procedimientos. En la tabla 2.2 se presenta su sintaxis. El símbolo no terminal $\langle v \rangle$ denota un valor construido parcialmente. Más adelante en el libro veremos otros tipos básicos, incluyendo pedazos¹², functors, celdas, diccionarios, arreglos, puertos, clases y objetos. Algunos de estos se explican en el apéndice B.

Tipamiento dinámico

Hay dos enfoques básicos para tipar, a saber, tipamiento dinámico y estático. En el tipamiento estático, todos los tipos de las variables se conocen en tiempo de compilación. En el tipamiento dinámico, el tipo de una variable se conoce solo en el momento en que la variable se liga. El modelo declarativo es dinámicamente tipado. El compilador trata de verificar que todas las operaciones usen los valores del tipo correcto. Pero debido al tipamiento dinámico, algunas verificaciones se dejan necesariamente para el tiempo de ejecución.

La jerarquía de tipos

Los tipos básicos del modelo declarativo se pueden clasificar en una jerarquía. En la figura 2.16 se muestra esta jerarquía, donde cada nodo denota un tipo. La jerarquía está ordenada por inclusión de conjuntos, i.e., todos los valores del tipo correspondiente a un nodo son también valores del tipo correspondiente al nodo padre. Por ejemplo, todas las tuplas son registros y todas las listas son tuplas. Esto implica que todas las operaciones de un tipo son también legales para un subtipo, e.g., todas las operaciones sobre listas funcionan también sobre cadenas. Más adelante en el libro, extenderemos esta jerarquía. Por ejemplo, los literales pueden ser átomos (se explican a continuación) u otro tipo de constantes llamadas nombres (ver sección 3.7.5). En las partes donde la jerarquía está incompleta se ha colocado “...”.

11. Nota del traductor: *ADTs*, en inglés.

12. Nota del traductor: *chunk*, en inglés.

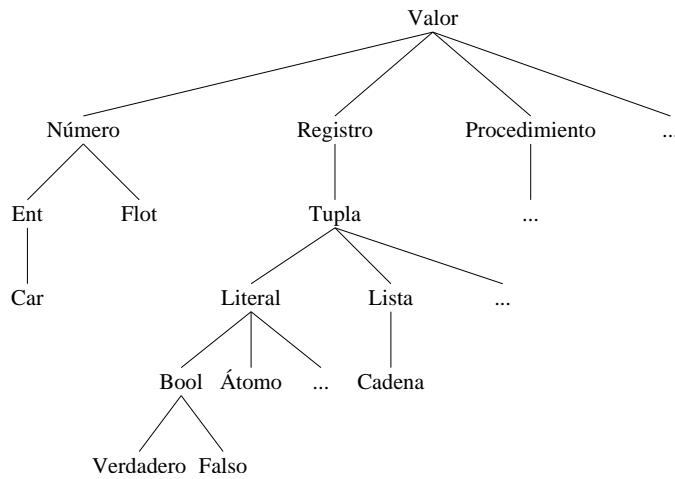


Figura 2.16: La jerarquía de tipos del modelo declarativo.

2.3.3. Tipos básicos

Presentamos algunos ejemplos de tipos básicos y cómo escribirlos. Vea el apéndice B para una información más completa.

- *Números*. Los números son o enteros o de punto flotante. Algunos ejemplos de números enteros son 314, 0, y ~ 10 (menos 10). Note que el signo menos se escribe con una virgulilla “ \sim ”. Algunos ejemplos de números de punto flotante son 1.0, 3.4, 2.0e2, y $\sim 2.0\text{E}^{\sim} 2$.
- *Átomos*. Un átomo es una especie de constante simbólica que puede utilizarse como un elemento indivisible (atómico) en los cálculos. Hay varias maneras diferentes de escribir átomos. Un átomo se puede escribir como una secuencia de caracteres iniciando con una letra minúscula y seguido de cualquier número de caracteres alfanuméricos. También se puede escribir un átomo como una secuencia de caracteres imprimibles encerrados entre comillas sencillas. Algunos ejemplos de átomos son `una_persona`, `donkeyKong3`, y `'#### hola ####'`.
- *Booleanos*. Un booleano es el símbolo `true` o el símbolo `false`.
- *Registros*. Un registro es una estructura de datos compuesta. Esta consiste de una etiqueta seguida de un conjunto de parejas de campos e identificadores de variables. Los campos pueden ser átomos, enteros, o booleanos. Algunos ejemplos de registros son `persona(edad:X1 nombre:X2)` (con campos `edad` y `nombre`), `persona(1:X1 2:X2)`, `'|'(1:H 2:T)`, `'#'(1:H 2:T)`, `nil`, y `persona`. Un átomo es un registro sin campos.
- *Tuplas*. Una tupla es un registro cuyos campos son enteros consecutivos, comenzando desde el 1. Los campos no tienen que escribirse en este caso. Algunos ejemplos

de tuplas son `persona(1:x1 2:x2)` y `persona(x1 x2)`; en este caso ambos significan lo mismo.

■ *Listas*. Un lista es o el átomo `nil` o la tupla `'|'(H T)` (la etiqueta es la barra vertical), donde `T` debe estar no-ligada o ligada a una lista. Esta tupla es llamada un par lista o un cons. Existe un azúcar sintáctico para las listas:

- La etiqueta `'|'` se puede escribir como un operador infijo, de manera que `H|T` significa lo mismo que `'|(H T)`.
- El operador `'|'` es asociativo a la derecha, de manera que `1|2|3|nil` significa lo mismo que `1|(2|(3|nil))`.
- Las listas que terminan en `nil` se pueden escribir entre paréntesis cuadrados `[...]`, de manera que `[1 2 3]` significa lo mismo que `1|2|3|nil`. Estas listas se llaman listas completas.

■ *Cadenas*. Una cadena es una lista de códigos de caracteres. La cadenas se pueden escribir entre comillas dobles, de manera que `"E=mc^2"` significa lo mismo que `[69 61 109 99 94 50]`.

■ *Procedimientos*. Un procedimiento es un valor de tipo procedimiento. La declaración

```
<x> = proc { $ <y>1 ... <y>n } <d> end
```

liga `<x>` a un valor nuevo de tipo procedimiento. Es decir, simplemente se declara un procedimiento nuevo. El símbolo `$` indica que el valor de tipo procedimiento es anónimo, i.e., se crea sin quedar ligado a ningún identificador. Hay una abreviación sintáctica más familiar:

```
proc {<x> <y>1 ... <y>n } <d> end
```

El símbolo `$` se reemplaza por el identificador `<x>`. Esto crea el valor de tipo procedimiento e inmediatamente trata de ligarlo a `<x>`. Tal vez esta abreviación es más fácil de leer, pero oculta la distinción entre crear un valor y ligarlo a un identificador.

2.3.4. Registros y procedimientos

Explicaremos ahora por qué escogimos los registros y los procedimientos como conceptos básicos en el lenguaje núcleo. Esta sección está dirigida a los lectores con alguna experiencia en programación que se preguntan por qué diseñamos el lenguaje núcleo como lo hicimos.

El poder de los registros

Los registros son la forma básica de estructurar datos. Ellos son los cimientos de la mayoría de las estructuras de datos incluyendo listas, árboles, colas, grafos, etc., como lo veremos en el capítulo 3. En algún grado, los registros juegan este papel

El modelo de computación declarativa

en la mayoría de los lenguajes de programación, pero veremos que su poder puede ir mucho más allá. El poder adicional aparece en mayor o menor grado, dependiendo de cuán bien o cuán pobremente se soporten los registros en el lenguaje. Para máximo poder, el lenguaje debería facilitar su creación, su exploración, y su manipulación. En el modelo declarativo, un registro se crea simplemente escribiéndolo, con una sintaxis compacta. Un registro se explora simplemente escribiendo un patrón, también con una sintaxis compacta. Finalmente, hay muchas operaciones para manipular los registros: agregar, borrar, o seleccionar campos; convertirlo a una lista y viceversa, etc. En general, los lenguajes que proveen este nivel de soporte para los registros se llaman lenguajes simbólicos.

Cuando los registros se soportan fuertemente, se pueden usar para incrementar la efectividad de muchas otras técnicas. Este libro se enfoca en tres de ellas en particular: programación orientada a objetos, diseño de interfaces gráficas de usuario (GUI, por su sigla en inglés), y programación basada en componentes. En el capítulo 7, se muestra cómo, en la programación orientada a objetos, los registros pueden representar mensajes y cabeceras de métodos, que es precisamente lo que los objetos utilizan para comunicarse. En el capítulo 10 (en CTM) se muestra cómo, en el diseño de interfaces gráficas de usuario, los registros pueden representar “aparatos,” el cimiento básico de una interfaz de usuario. En la sección 3.9 se muestra cómo, en la programación basada en componentes, los registros pueden representar módulos de primera clase, los cuales agrupan operaciones relacionadas.

Por Qué procedimientos?

Un lector con alguna experiencia en programación puede preguntarse por qué nuestro lenguaje núcleo tiene los procedimientos como una construcción básica. Los fanáticos de la programación orientada a objetos pueden preguntar por qué no usar objetos en su lugar, mientras que los fanáticos de la programación funcional puede preguntar por qué no usar funciones. Hubiéramos podido escoger cualquiera de esas posibilidades pero no lo hicimos. Las razones son bastante sencillas.

Los procedimientos son más apropiados que los objetos porque son más simples. Los objetos son realmente bastante complicados, como se explica en el capítulo 7. Los procedimientos son más adecuados que las funciones porque no definen necesariamente entidades que se comportan como funciones matemáticas.¹³ Por ejemplo, definimos tanto componentes como objetos como abstracciones basadas en procedimientos. Además, los procedimientos son flexibles puesto que no hacen ninguna suposición sobre el número de entradas y salidas. En cambio, una función siempre tiene exáctamente una salida. Un procedimiento puede tener cualquier número

13. Desde un punto de vista teórico, los procedimientos son “procesos” tal como se usan en los cálculos concurrentes como el cálculo π . Los argumentos hacen las veces de canales. En este capítulo usamos procesos compuestos secuencialmente, con canales de un solo tiro. En los capítulos 4 y 5 se muestran otros tipos de canales (con secuencias de mensajes) y se hace composición concurrente de procesos.

Operación	Descripción	Tipo del argumento
<code>A==B</code>	Comparación de igualdad	Valor
<code>A\=B</code>	Comparación de desigualdad	Valor
<code>{ IsProcedure P }</code>	Prueba si es procedimiento	Valor
<code>A=<B</code>	Comparación menor o igual	Número o Átomo
<code>A<B</code>	Comparación menor	Número o Átomo
<code>A>=B</code>	Comparación mayor o igual	Número o Átomo
<code>A>B</code>	Comparación mayor	Número o Átomo
<code>A+B</code>	Suma	Número
<code>A-B</code>	Resta	Número
<code>A*B</code>	Multiplicación	Número
<code>A div B</code>	División	Ent
<code>A mod B</code>	Módulo	Ent
<code>A/B</code>	División	Flot
<code>{Arity R}</code>	Aridad	Registro
<code>{Label R}</code>	Etiqueta	Registro
<code>R.F</code>	Selección de campo	Registro

Tabla 2.3: Ejemplos de operaciones básicas.

de entradas y salidas, incluyendo cero. Veremos que los procedimientos son extremadamente poderosos, cuando hablamos de programación de alto orden en la sección 3.6.

2.3.5. Operaciones básicas

En la tabla 2.3 se presentan las operaciones básicas que usaremos en este capítulo y en el siguiente. Hay un azúcar sintáctico para muchas de estas operaciones de manera que se puedan escribir concisamente como expresiones. Por ejemplo, `X=A*B` es un azúcar sintáctico para `{Number.^*^ A B X}`, donde `Number.^*^` es un procedimiento asociado con el tipo Número.¹⁴ Todas las operaciones se pueden denotar en alguna forma larga, e.g., `Value.^==^`, `Value.^<^`, `Int.^div^`, `Float.^/^`. En la tabla se usa el azúcar sintáctico siempre que exista.

- *Aritmética.* Los números de punto flotante tienen las cuatro operaciones básicas, `+`, `-`, `*`, y `/`, con el significado normal. Los enteros tienen las operaciones básicas `+`, `-`, `*`, `div`, y `mod`, donde `div` es la división entera (se trunca la parte fraccional) y `mod` es el módulo entero, i.e., el residuo después de una división. Por ejemplo, `10 mod 3=1`.

14. Para ser precisos, `Number` es un módulo que agrupa las operaciones del tipo Número y `Number.^*^` selecciona la operación de multiplicación.

El modelo de computación declarativa

- *Operaciones sobre registros.* Las tres operaciones básicas sobre registros son **Arity**, **Label**, y “.” (punto, que significa selección de campo). Por ejemplo, dado

```
X=persona(nombre:"Jorge" edad:25)
```

entonces $\{\text{Arity } X\} = \{\text{edad nombre}\}$, $\{\text{Label } X\} = \text{persona}$, y $X.\text{edad} = 25$. La invocación a **Arity** devuelve una lista que contiene los campos enteros en orden ascendente, y luego los campos que son átomos en orden lexicográfico.

- *Comparaciones.* Las funciones booleanas de comparación incluyen **==** y **\=**, que comparan si cualesquier dos valores son iguales o no, así como las comparaciones numéricas **=<**, **<**, **>=**, y **>**, las cuales pueden comparar dos enteros, dos flotantes, o dos átomos. Los átomos se comparan de acuerdo al orden lexicográfico de sus representaciones para impresión. En el ejemplo siguiente, **Z** está ligado al máximo entre **X** y **Y**:

```
declare X Y Z T in
X=5 Y=10
T=(X>=Y)
if T then Z=X else Z=Y end
```

Hay un azúcar sintáctico de manera que una declaración **if** pueda aceptar una expresión como su condición. El ejemplo anterior se reescribe así:

```
declare X Y Z in
X=5 Y=10
if X>=Y then Z=X else Z=Y end
```

- *Operaciones sobre procedimientos.* Hay tres operaciones básicas sobre procedimientos: definición (con la declaración **proc**), invocación (con la notación con corchetes), y verificación si un valor es un procedimiento con la función **IsProcedure**. La invocación $\{\text{IsProcedure } P\}$ devuelve **true** si **P** es un procedimiento y **false** en cualquier otro caso.

En el apéndice B se presenta un conjunto más completo de operaciones básicas.

2.4. La semántica del lenguaje núcleo

La ejecución del lenguaje núcleo consiste en evaluar funciones sobre valores parciales. Para ver esto, presentamos la semántica del lenguaje núcleo en términos de un modelo operacional sencillo. El modelo se diseña para que el programador pueda razonar en una forma simple tanto sobre la corrección como sobre la complejidad de sus programas. Este es un tipo de máquina abstracta, pero con un alto nivel de abstracción donde se omiten detalles tales como registros y direcciones explícitas de memoria.

2.4.1. Conceptos básicos

Antes de presentar la semántica formal, presentaremos algunos ejemplos para mostrar intuitivamente cómo se ejecuta el lenguaje núcleo. Esto servirá como motivación a la semántica y facilitará su entendimiento.

Una ejecución simple

Durante la ejecución normal, las declaraciones se ejecutan una por una en el orden textual en que aparecen. Miremos una ejecución simple:

```
local A B C D in
  A=11
  B=2
  C=A+B
  D=C*C
end
```

Esta ejecución parece bastante sencilla; en ella se ligará `D` a 169. Miremos exactamente cómo se hace. La declaración `local` crea cuatro variables nuevas en el almacén, y hace que los cuatro identificadores de variables `A`, `B`, `C`, `D` las refieran (por conveniencia, estamos usando una ligera extensión de la declaración `local` de la tabla 2.1). A continuación se realizan las dos ligaduras, `A=11` y `B=2`. La suma `C=A+B` suma los valores de `A` y `B` y liga `C` con el resultado 13. La multiplicación `D=C*C` multiplica el valor de `C` por si mismo y liga `D` con el resultado 169. Es bastante simple.

Identificadores de variables y el alcance léxico

Vimos que la declaración `local` hace dos cosas: crea una variable nueva y coloca al identificador a referenciar esa variable. El identificador solo referencia esa variable dentro de la declaración `local`, i.e., entre el `local` y el `end`. La región del programa en la cual un identificador referencia una variable en particular se llama el *alcance* del identificador. Fuera de su alcance, el identificador no significa la misma cosa. Miremos un poco más de cerca qué implica esto. Considere el siguiente fragmento de programa:

```
local X in
  X=1
  local X in
    X=2
    {Browse X}
  end
  {Browse X}
end
```

¿Qué se ve en el browser? Muestra primero 2 y luego 1. Hay un solo identificador, pero en diferentes puntos de la ejecución, referencia diferentes variables.

El modelo de computación declarativa

Resumamos esta idea. El significado de un identificador como `x` es determinado por la declaración `local` más interna que declara a `x`. El área del programa donde `x` conserva ese significado se llama el alcance de `x`. Podemos averiguar el alcance de un identificador por una simple inspección al texto del programa; no tenemos que hacer nada complicado como ejecutar o analizar el programa. Esta regla para determinar el alcance se llama alcance léxico o alcance estático. Más adelante veremos otro tipo de regla de alcance, el alcance dinámico, el cual es útil algunas veces. Sin embargo, el alcance léxico es, de lejos, el tipo más importante de regla de alcance. Una razón es porque es localizado, i.e., el significado de un identificador se puede determinar mirando sólo un pedazo del programa. Veremos otra razón dentro de poco.

Procedimientos

Los procedimientos son unos de los más importantes cimientos de nuestro lenguaje. Presentaremos un ejemplo sencillo que muestra cómo definir e invocar un procedimiento. El procedimiento siguiente liga `z` con el máximo entre `x` y `y`:

```
proc {Max X Y ?Z}
  if X>=Y then Z=X else Z=Y end
end
```

Para hacer que la definición sea más fácil de leer, marcamos el argumento de salida con un signo de interrogación “?”. Esto no tiene ningún efecto sobre la ejecución; sólo es un comentario. Invocar `{Max 3 5 c}` liga `c` a 5. ¿Cómo funciona exactamente el procedimiento? Cuando `Max` es invocado, los identificadores `x`, `y`, `y` `z` se ligan a 3, 5, y a la variable no-ligada referenciada por `c`. Cuando `Max` liga a `z`, entonces liga esta variable. Como `c` también referencia esa variable, entonces `c` también se liga. A esta manera de pasar los parámetros se le llama paso por referencia. Los procedimientos producen resultados de salida por medio de las referencias a variables no-ligadas, recibidas como argumento, que se ligan dentro del procedimiento. Este libro utiliza paso por referencia la mayoría de las veces, tanto para variables de flujo de datos como para variables mutables. En la sección 6.4.4 se explican otros mecanismos de paso de parámetros.

Procedimientos con referencias externas

Examinemos el cuerpo de `Max`. Es tan solo una declaración `if`:

```
if X>=Y then Z=X else Z=Y end
```

Sin embargo, esta declaración tiene una particularidad: ¡no puede ser ejecutada! Todo porque la declaración no define los identificadores `x`, `y`, y `z`. Estos identificadores indefinidos se llaman identificadores libres. (Algunas veces se llaman variables libres, aunque estrictamente hablando no son variables.) Cuando la declaración se pone dentro del procedimiento `Max`, ésta puede ser ejecutada, pues todos los identificadores libres se han declarado como argumentos del procedimiento.

¿Qué pasa si definimos un procedimiento que sólo declara algunos de los identificadores libres como argumentos? Por ejemplo, definamos el procedimiento `LB` con el mismo cuerpo de procedimiento que `Max`, pero sólo con dos argumentos:

```
proc {LB X ?Z}
  if X>=Y then Z=X else Z=Y end
end
```

¿Qué hace este procedimiento cuando se ejecuta? Aparentemente, toma cualquier número `x` y liga `z` a `x` si `x>=y`, sino la liga a `y`. Es decir, `z` siempre es al menos `y`. ¿Cuál es el valor de `y`? No es ninguno de los argumentos del procedimiento. Tiene que ser el valor de `y` en el momento en que el procedimiento fue definido. Esta es una consecuencia del alcance estático. Si `y=9` en el momento que se definió el procedimiento, entonces invocar `{LB 3 z}` liga `z` a 9. Considere el siguiente fragmento de programa:

```
local Y LB in
  Y=10
  proc {LB X ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  local Y=15 Z in
    {LB 5 Z}
  end
end
```

Con qué valor queda ligado `z` luego de la invocación `{LB 5 z}`? `z` será ligado a 10. La ligadura `Y=15` realizada antes que `LB` sea invocado, es ignorada; la que se tiene en cuenta es la ligadura `Y=10`, vigente al momento de la definición del procedimiento.

Alcance dinámico versus alcance estático

Considere el siguiente ejemplo sencillo:

```
local P Q in
  proc {Q X} {Browse estat(X)} end
  proc {P X} {Q X} end
  local Q in
    proc {Q X} {Browse din(X)} end
    {P hola}
  end
end
```

¿Qué debería mostrar en pantalla, `estat(hola)` o `din(hola)`? El alcance estático establece que se debe mostrar `estat(hola)`. En otras palabras, `P` utiliza la versión de `Q` que existe al momento de la definición de `P`. Pero hay otra solución posible: `P` podría usar la versión de `Q` que existe al invocar a `P`. Esto se llama alcance dinámico.

Tanto el alcance estático como el dinámico han sido utilizados por defecto en lenguajes de programación. Comparemos los dos tipos de alcance, comenzando por ver sus definiciones una al lado de la otra:

El modelo de computación declarativa

- *Alcance estático.* La variable que corresponde a la ocurrencia de un identificador es la que haya sido definida en la declaración textual más interna en las cercanías de la ocurrencia en el programa fuente.
- *Alcance dinámico.* La variable que corresponde a la ocurrencia de un identificador es la que esté en la declaración más recientemente vista durante la ejecución de la declaración actual.

El lenguaje Lisp utilizaba originalmente alcance dinámico. Common Lisp y Scheme, descendientes de Lisp, utilizan alcance estático por defecto. Common Lisp permite declarar variables de alcance dinámico, las cuales son llamadas variables especiales [162]. ¿Cuál es el comportamiento correcto por defecto? El comportamiento correcto por defecto es definir los valores de tipo procedimiento con alcance estático, pues un procedimiento que funciona cuando es definido, seguirá funcionando independientemente del ambiente en el cual sea invocado. Esta es una propiedad importante de la ingeniería de *software*.

El alcance dinámico es útil en algunas áreas bien definidas. Por ejemplo, considere el caso de un procedimiento cuyo código se transfiere a través de la red de un computador a otro. Algunas de estas referencias externas, e.g., invocaciones a operaciones de bibliotecas comunes, pueden usar alcance dinámico. De esta forma, el procedimiento usará el código local para esas operaciones y no código remoto. Esto es mucho más eficiente.¹⁵

Abstracción procedimental

Resumamos lo que hemos aprendido de Max y LB. Tres conceptos juegan un papel importante:

1. Abstracción procedimental. Cualquier declaración puede ser colocada al interior de un procedimiento. Esto se llama abstracción procedimental. También decimos que la declaración se abstrae dentro de un procedimiento.
2. Identificadores libres. Un identificador libre en una declaración es un identificador que no ha sido definido en esa declaración. Normalmente, éste ha sido definido en una declaración que contiene aquella donde está el identificador libre.
3. Alcance estático. Un procedimiento puede tener referencias externas, las cuales son identificadores libres en el cuerpo del procedimiento que no han sido declarados como argumentos. LB tiene una referencia externa mientras que Max no tiene. El valor de una referencia externa es el valor al que está ligada cuando el procedimiento se define. Esta es una consecuencia del alcance estático.

15. Sin embargo, no hay garantía alguna que la operación se comporte de la misma manera en la máquina destino. Luego aún para programas distribuidos, el alcance por defecto debe ser estático.

La abstracción procedimental y el alcance estático, juntos, conforman una de las herramientas más poderosas presentadas en el libro. En la semántica, veremos que se pueden implementar de una manera simple.

Comportamiento por flujo de datos

En el almacén de asignación única, las variables pueden estar no-ligadas. Por otro lado, algunas declaraciones necesitan que ciertas variables estén ligadas, o no podrían ejecutarse. Por ejemplo, qué pasa cuando ejecutamos:

```
local X Y Z in
    X=10
    if X>=Y then Z=X else Z=Y end
end
```

La comparación `X>=Y` devuelve `true` o `false`, si puede decidir en qué caso se encuentra. Pero si `Y` es una variable no-ligada, no se puede decidir. ¿Qué se hace? Devolver `true` o `false` sería incorrecto. Lanzar una condición de error sería una medida drástica, pues el programa no ha realizado nada errado (tampoco nada correcto). Decidimos que el programa, simplemente, suspenderá su ejecución, sin señalar ningún tipo de error. Si alguna otra actividad (a ser determinada más adelante) liga `Y`, entonces la ejecución suspendida puede continuar, como si nada hubiera perturbado su flujo de ejecución normal. Esto se llama comportamiento por flujo de datos. Este comportamiento es el fundamento de una segunda y poderosa herramienta presentada en el libro, a saber, la concurrencia. En la semántica, veremos que el comportamiento por flujo de datos puede ser implementado de una manera sencilla.

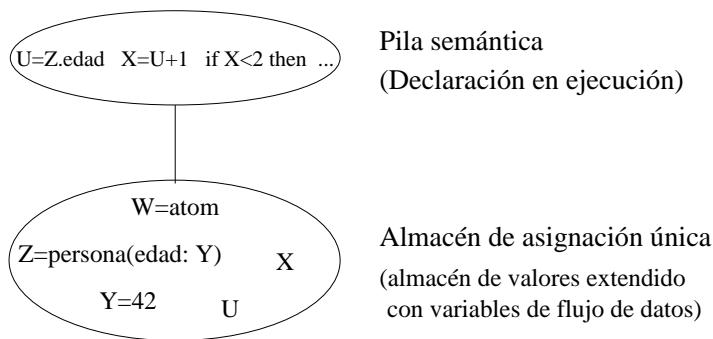
2.4.2. La máquina abstracta

Definimos la semántica del lenguaje núcleo como una semántica operacional, i.e., define el significado del lenguaje núcleo a través de su ejecución en una máquina abstracta. Primero definimos los conceptos básicos de una máquina abstracta: ambientes, declaración semántica, pila de declaraciones, estado de la ejecución, y computación. Luego mostramos cómo ejecutar un programa. Finalmente explicamos cómo calcular con ambientes, lo cual es una operación semántica frecuente.

Definiciones

Un programa en ejecución se define en términos de una computación, la cual es una secuencia de estados de la ejecución. Definamos exáctamente lo que esto significa. Necesitamos los conceptos siguientes:

- Un *almacén de asignación única* σ es un conjunto de variables del almacén. Estas variables se dividen en (1) conjuntos de variables que son iguales pero están no-ligadas, y (2) variables que están ligadas a un número, registro, o procedimiento. Por ejemplo, en el almacén $\{x_1, x_2 = x_3, x_4 = a | x_2\}$, x_1 no está ligada, x_2 y x_3

El modelo de computación declarativa**Figura 2.17:** El modelo de computación declarativa.

son iguales y no-ligadas, y x_4 está ligada a un valor parcial $a|x_2$. Una variable del almacén ligada a un valor es indistinguible de su valor. Es por esto que a una variable del almacén se le conoce también como una entidad del almacén.

- Un *ambiente* E es una aplicación de identificadores de variable en entidades de σ . Esto se explica en la sección 2.2. Escribiremos E como un conjunto de parejas, e.g., $\{x \rightarrow x, y \rightarrow y\}$, donde x, y son identificadores y x, y referencian entidades del almacén.
- Una *declaración semántica* es una pareja $(\langle d \rangle, E)$ donde $\langle d \rangle$ es una declaración y E es un ambiente. La declaración semántica relaciona una declaración con lo que ella referencia en el almacén. El conjunto de las posibles declaraciones se presenta en la sección 2.3.
- Un *estado de la ejecución* es una pareja (ST, σ) donde ST es una pila de declaraciones semánticas y σ es un almacén de asignación única. En la figura 2.17 se presenta una gráfica representando un estado de la ejecución.
- Una *computación* es una secuencia de estados de la ejecución, comenzando a partir de un estado inicial: $(ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow \dots$.

Una transición en una computación se llama una etapa de computación. Una etapa de computación es atómica, i.e., no existen estados intermedios visibles. Es como si la etapa hiciera “todo de una sola vez.” En este capítulo todas las computaciones son secuenciales, i.e., cada estado de la ejecución contiene exactamente una pila semántica, la cual se va transformando por medio de una secuencia lineal de etapas de computación.

Ejecución de un programa

Ejecutemos un programa con esta semántica. Un programa no es más que una declaración $\langle d \rangle$. A continuación mostramos cómo se ejecuta el programa:

- El estado inicial de la ejecución es:

$$([(\langle d \rangle, \emptyset)], \emptyset)$$

Es decir, el almacén inicial está vacío (sin variables, conjunto vacío \emptyset) y el estado inicial de la ejecución tiene sólo una declaración semántica $(\langle d \rangle, \emptyset)$ en la pila ST . La declaración semántica contiene $\langle d \rangle$ y un ambiente vacío (\emptyset) . Usamos paréntesis cuadrados [...] para denotar la pila.

- En cada etapa, se toma el primer elemento de ST (tomarlo implica sacarlo de la pila) y, de acuerdo a su forma, se procede con la ejecución de la etapa.
- El estado final de la ejecución (si lo hay) es un estado en el cual la pila semántica está vacía.

Una pila semántica ST puede estar en uno de tres posibles estados en tiempo de ejecución:

- Ejecutable: ST puede realizar una etapa de computación.
- Terminado: ST está vacía.
- Suspendido: ST no está vacía, pero tampoco puede realizar una etapa de computación.

Calculando con ambientes

En la ejecución de un programa se presenta frecuentemente la necesidad de calcular con ambientes. Un ambiente E es una función que asocia identificadores de variables $\langle x \rangle$ con entidades del almacén (tanto con variables no-ligadas como con valores). La notación $E(\langle x \rangle)$ representa la entidad del almacén asociada con el identificador $\langle x \rangle$ en el ambiente E . Para definir la semántica de las instrucciones de la máquina abstracta, necesitamos dos operaciones comunes entre ambientes, a saber, la extensión y la restricción.

La *Extensión* define un ambiente nuevo agregando asociaciones nuevas a un ambiente ya existente. La notación

$$E + \{ \langle x \rangle \rightarrow x \}$$

denota un ambiente nuevo E' construido a partir de E agregando la asociación $\{ \langle x \rangle \rightarrow x \}$. Esta asociación anula cualquier otra asociación preexistente del identificador $\langle x \rangle$. Es decir, $E'(\langle x \rangle)$ es igual a x , y $E'(\langle y \rangle)$ es igual a $E(\langle y \rangle)$ para todos los identificadores $\langle y \rangle$ diferentes de $\langle x \rangle$. Cuando necesitamos agregar más de una asociación a la vez, escribimos $E + \{ \langle x \rangle_1 \rightarrow x_1, \dots, \langle x \rangle_n \rightarrow x_n \}$.

La *Restricción* define un ambiente nuevo cuyo dominio es un subconjunto del ambiente existente. La notación

$$E|_{\{ \langle x \rangle_1, \dots, \langle x \rangle_n \}}$$

denota un ambiente nuevo E' tal que $\text{dom}(E') = \text{dom}(E) \cap \{ \langle x \rangle_1, \dots, \langle x \rangle_n \}$ y $E'(\langle x \rangle) = E(\langle x \rangle)$ para todo $\langle x \rangle \in \text{dom}(E')$. Es decir, el ambiente nuevo no contiene

identificadores diferentes a aquellos mencionados en el conjunto.

2.4.3. Declaraciones cuya ejecución nunca se suspende

Primero presentaremos la semántica de las declaraciones cuya ejecución nunca se suspenderá.

La declaración skip

La declaración semántica es:

(skip, E)

La ejecución se completa con sólo tomar la pareja de la pila semántica.

Composición secuencial

La declaración semántica es:

$(\langle d \rangle_1 \langle d \rangle_2, E)$

La ejecución consiste de las acciones siguientes:

- Colocar $(\langle d \rangle_2, E)$ en la pila.
- Colocar $(\langle d \rangle_1, E)$ en la pila.

Declaración de variable (la declaración local)

La declaración semántica es:

$(\text{local } \langle x \rangle \text{ in } \langle d \rangle \text{ end}, E)$

La ejecución consiste de las acciones siguientes:

- Crear una variable nueva, x , en el almacén.
- Calcule E' como $E + \{\langle x \rangle \rightarrow x\}$, i.e., E' es lo mismo que E salvo que se le agrega la asociación entre $\langle x \rangle$ y x .
- Coloque $(\langle d \rangle, E')$ en la pila.

Ligadura variable-variable

La declaración semántica es:

$(\langle x \rangle_1 = \langle x \rangle_2, E)$

La ejecución consiste de la acción siguiente:

- Ligue $E(\langle x \rangle_1)$ y $E(\langle x \rangle_2)$ en el almacén.

Creación de valores

La declaración semántica es:

$$(\langle x \rangle = \langle v \rangle, E)$$

donde $\langle v \rangle$ es un valor parcialmente construido de tipo registro, número, o procedimiento. La ejecución consiste de las acciones siguientes:

- Crear una variable nueva, x , en el almacén.
- Construir el valor representado por $\langle v \rangle$ en el almacén y hacer que x lo refiera. Todos los identificadores de $\langle v \rangle$ se reemplazan por sus contenidos en el almacén, de acuerdo al ambiente E .
- Ligar $E(\langle x \rangle)$ y x en el almacén.

Hasta ahora hemos visto cómo construir valores de los tipos número y registro, pero no hemos visto cómo hacerlo para los valores de tipo procedimiento. Para poder explicarlo, primero tenemos que explicar el concepto de alcance léxico.

Ocurrencias libres y ligadas de identificadores

Una declaración $\langle d \rangle$ puede contener múltiples ocurrencias de identificadores de variables. Por cada ocurrencia de un identificador, podemos hacernos la pregunta: ¿Dónde fue definido este identificador? Si la definición está en alguna declaración (dentro o no de $\langle d \rangle$) que rodea (i.e. encierra) textualmente la ocurrencia, entonces decimos que la definición obedece el alcance léxico. Como el alcance está determinado por el texto del código fuente, también se le llama alcance estático.

La ocurrencia de un identificador en una declaración puede ser clasificada como ligada o libre con respecto a esa declaración. Una ocurrencia de un identificador x está ligada con respecto a la declaración $\langle d \rangle$ si x está definido dentro de $\langle d \rangle$, i.e., en una declaración **local**, en el patrón de una declaración **case**, o como argumento de una declaración de un procedimiento. Una ocurrencia de un identificador está libre, si no está ligada. Las ocurrencias libres sólo pueden existir en fragmentos incompletos de programas, i.e., declaraciones que no pueden ejecutarse. En un programa ejecutable, toda ocurrencia de un identificador está ligada con respecto al programa completo.

Ocurrencias ligadas de identificadores y variables ligadas

¡No confunda una ocurrencia ligada de un identificador con una variable ligada! Una ocurrencia ligada de un identificador no existe en tiempo de ejecución; es un nombre textual de variable que aparece textualmente dentro de una construcción que lo define (e.g., un procedimiento o definición de variable). Una variable ligada existe en tiempo de ejecución; es una variable de flujo de datos que está ligada a un valor parcial.

El modelo de computación declarativa

El siguiente es un ejemplo de programa con ocurrencias libres y ligadas de identificadores:

```
local Arg1 Arg2 in
    Arg1=111*111
    Arg2=999*999
    Res=Arg1+Arg2
end
```

En esta declaración, todos los identificadores de variables se definen con alcance léxico. Todas las ocurrencias de los identificadores `Arg1` y `Arg2` están ligadas y la única ocurrencia del identificador `Res` está libre. Esta declaración no puede ejecutarse. Para poder ejecutarla, tiene que hacer parte de una declaración más grande que defina `Res`. La siguiente es una extensión ejecutable:

```
local Res in
    local Arg1 Arg2 in
        Arg1=111*111
        Arg2=999*999
        Res=Arg1+Arg2
    end
    {Browse Res}
end
```

Como el único identificador que ocurre libre, `Browse`, está definido en el ambiente global, esta declaración es ejecutable.

Valores de tipo procedimiento (clausuras)

Miremos cómo construir un valor de tipo procedimiento en el almacén. No es tan simple como uno podría imaginar debido a que los procedimientos pueden tener referencias externas. Por ejemplo:

```
proc {LowerBound x ?z}
    if X>=Y then Z=X else Z=Y end
end
```

En este ejemplo, la declaración `if` tiene seis ocurrencias libres de tres (identificadores de) variables, `x` (dos veces), `y` (dos veces), y `z` (dos veces). Dos de ellas, `x` y `z`, son a su vez parámetros formales (del procedimiento `LowerBound`). La tercera, `y`, no es un parámetro formal; ella tiene que estar definida por el ambiente donde se definió el procedimiento. El mismo valor de tipo procedimiento debe tener una asociación de `y` con el almacén, o sino, no podríamos invocar el procedimiento pues `y` sería una clase de referencia suelta.

Miremos qué pasa en el caso general. Una expresión que define un procedimiento se escribe:

```
proc { $ <y>1 ... <y>n} <d> end
```

La declaración `<d>` puede tener ocurrencias libres de identificadores de variables. Cada ocurrencia libre de un identificador, o corresponde a un parámetro formal, o no. Las primeras se definen nuevamente cada vez que el procedimiento es invocado.

Estas forman un subconjunto de los parámetros formales $\{\langle y \rangle_1, \dots, \langle y \rangle_n\}$. Las segundas se definen una sola vez y para siempre, en el momento en que se define el procedimiento. A estas las llamamos las referencias externas del procedimiento, y las escribimos como $\{\langle z \rangle_1, \dots, \langle z \rangle_k\}$. Entonces, un valor de tipo procedimiento es una pareja:

```
( proc { $ <y>_1 ... <y>_n } <d> end, CE )
```

Aquí CE (el ambiente contextual) es $E|_{\{\langle z \rangle_1, \dots, \langle z \rangle_n\}}$, donde E es el ambiente en el momento en que el procedimiento es definido. La pareja se coloca en el almacén al igual que cualquier otro valor.

A los valores de tipo procedimiento se les llama frecuentemente clausuras o clausuras de alcance léxico, debido a que contienen tanto un ambiente como una definición de procedimiento. Es decir, se encierra (i.e., empaqueta) el ambiente en el momento de la definición del procedimiento. A esto también se le llama captura del ambiente. Cuando el procedimiento es invocado, se utiliza el ambiente contextual para construir el ambiente en que se ejecutará el cuerpo del procedimiento.

2.4.4. Declaraciones cuya ejecución se puede suspender

Nos restan tres declaraciones del lenguaje núcleo por tratar:

```
<d> ::= ...
| if <x> then <d>_1 else <d>_2 end
| case <x> of <patrón> then <d>_1 else <d>_2 end
| '{' <x> <y>_1 ... <y>_n '}'
```

¿Qué debería suceder con estas declaraciones si $\langle x \rangle$ es no-ligada? A partir de la discusión en la sección 2.2.8, sabemos qué debería pasar. Las declaraciones simplemente deberían esperar hasta que $\langle x \rangle$ se ligue. Por eso decimos que éstas son declaraciones cuya ejecución se puede suspender. Todas ellas tienen una *condición de activación*, la cual se define como una condición que se debe cumplir para que la ejecución continúe. La condición es que $E(\langle x \rangle)$ debe estar determinada, i.e., ligada a un número, registro, o procedimiento.

En el modelo declarativo de este capítulo, una vez una declaración se suspende, no continuará nunca. El programa simplemente detiene su ejecución. Esto es así porque no existe otra ejecución que pueda volver cierta la condición de activación. En el capítulo 4, cuando introduzcamos la programación concurrente, tendremos ejecuciones con varias pilas semánticas (más de una). Una pila ST en estado suspendido puede pasar a estado ejecutable si otra pila realiza una operación que haga cierta la condición de activación de ST . Esta es la base de la ejecución por flujo de datos. Por ahora, investiguemos la programación secuencial y quedémonos sólo con una pila semántica.

*El modelo de computación declarativa***Condicional (la declaración if)**

La declaración semántica es:

$(\text{if } \langle x \rangle \text{ then } \langle d \rangle_1 \text{ else } \langle d \rangle_2 \text{ end}, E)$

La ejecución consiste de las acciones siguientes:

- Si la condición de activación es cierta ($E(\langle x \rangle)$ está determinada), entonces realice las acciones siguientes:
 - Si $E(\langle x \rangle)$ no es un booleano (**true** o **false**) entonces lance una condición de error.
 - Si $E(\langle x \rangle)$ es **true**, entonces coloque $(\langle d \rangle_1, E)$ en la pila.
 - Si $E(\langle x \rangle)$ es **false**, entonces coloque $(\langle d \rangle_2, E)$ en la pila.
- Si la condición de activación es falsa, entonces la ejecución no continúa. El estado de la ejecución se queda como está. Decimos que la ejecución se suspende. La suspensión puede ser temporal. Si alguna otra actividad en el sistema hace que la condición de activación sea cierta, entonces la ejecución puede continuar.

Invocación de procedimiento

La declaración semántica es:

$(\{\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n\}, E)$

La ejecución consiste de las acciones siguientes:

- Si la condición de activación es cierta ($E(\langle x \rangle)$ está determinada), entonces realice las acciones siguientes:
 - Si $E(\langle x \rangle)$ no es un valor de tipo procedimiento o es un procedimiento con un número de argumentos diferente de n , entonces lance una condición de error.
 - Si $E(\langle x \rangle)$ tiene la forma $(\text{proc } \{ \$ \langle z \rangle_1 \dots \langle z \rangle_n \} \langle d \rangle \text{ end}, CE)$ entonces coloque $(\langle d \rangle, CE + \{\langle z \rangle_1 \rightarrow E(\langle y \rangle_1), \dots, \langle z \rangle_n \rightarrow E(\langle y \rangle_n)\})$ en la pila.
- Si la condición de activación es falsa, entonces se suspende la ejecución.

Reconocimiento de patrones (la declaración case)

La declaración semántica es:

$(\text{case } \langle x \rangle \text{ of } \langle \text{lit} \rangle (\langle \text{cmp} \rangle_1: \langle x \rangle_1 \dots \langle \text{cmp} \rangle_n: \langle x \rangle_n) \text{ then } \langle d \rangle_1 \text{ else } \langle d \rangle_2 \text{ end}, E)$

(Aquí $\langle \text{lit} \rangle$ y $\langle \text{cmp} \rangle$ son sinónimos de $\langle \text{literal} \rangle$ y $\langle \text{campo} \rangle$.) La ejecución consiste de las acciones siguientes:

- Si la condición de activación es cierta ($E(\langle x \rangle)$ está determinada), entonces realice las acciones siguientes:

- Si la etiqueta de $E(\langle x \rangle)$ es $\langle \text{lit} \rangle$ y su aridad es $\{\langle \text{cmp} \rangle_1, \dots, \langle \text{cmp} \rangle_n\}$, entonces coloque $(\langle d \rangle_1, E + \{\langle x \rangle_1 \rightarrow E(\langle x \rangle).(\langle \text{cmp} \rangle_1, \dots, \langle x \rangle_n \rightarrow E(\langle x \rangle).(\langle \text{cmp} \rangle_n)\})$ en la pila.
- En cualquier otro caso coloque $(\langle d \rangle_2, E)$ en la pila.
- Si la condición de activación es falsa, entonces se suspende la ejecución.

2.4.5. Conceptos básicos (una vez más)

Ahora que hemos visto la semántica del lenguaje núcleo, miremos una vez más los ejemplos de la sección 2.4.1 para ver exáctamente lo que ellos hacen.

Miraremos tres ejemplos; sugerimos que el lector haga los otros como ejercicio.

Identificadores de variables y alcance estático

Anteriormente vimos que la declaración $\langle d \rangle$ presentada a continuación, mostraba primero 2 y luego 1:

$$\langle d \rangle \equiv \left\{ \begin{array}{l} \text{local } x \text{ in} \\ \quad x=1 \\ \quad \langle d \rangle_1 \equiv \left\{ \begin{array}{l} \text{local } x \text{ in} \\ \quad x=2 \\ \quad \{ \text{Browse } x \} \\ \quad \text{end} \end{array} \right. \\ \langle d \rangle_2 \equiv \{ \text{Browse } x \} \\ \quad \text{end} \end{array} \right.$$

El mismo identificador x referencia primero a 2 y luego referencia a 1. Podemos entender mejor lo que pasa ejecutando $\langle d \rangle$ en nuestra máquina abstracta.

1. El estado inicial de la ejecución es:

$$([(\langle d \rangle, \emptyset)], \emptyset)$$

Tanto el ambiente como el almacén están vacíos ($E = \emptyset$ y $\sigma = \emptyset$).

2. Después de ejecutar la declaración **local** más externa y la ligadura $x=1$, tenemos:

$$([(\langle d \rangle_1 \langle d \rangle_2, \{x \rightarrow x\})], \{y = 1, x = y\})$$

Simplificamos $\{y = 1, x = y\}$ a $\{x = 1\}$. El identificador x referencia la variable del almacén x , la cual está ligada a 1. La siguiente declaración es la composición secuencial $\langle d \rangle_1 \langle d \rangle_2$.

El modelo de computación declarativa

3. Después de ejecutar la composición secuencial, tenemos:

$$([(\langle d \rangle_1, \{x \rightarrow x\}), (\langle d \rangle_2, \{x \rightarrow x\})], \\ \{x = 1\})$$

Cada una de las declaraciones $\langle d \rangle_1$ y $\langle d \rangle_2$ tienen su ambiente propio. En este momento, los dos ambientes tienen valores idénticos.

4. Empecemos por ejecutar $\langle d \rangle_1$. La primera declaración en $\langle d \rangle_1$ es una declaración **local**. Ejecutarla da

$$([(x=2 \{ \text{Browse } x \}, \{x \rightarrow x'\}), (\langle d \rangle_2, \{x \rightarrow x\})], \\ \{x', x = 1\})$$

Esto crea una variable nueva x' y calcula el ambiente nuevo $\{x \rightarrow x\} + \{x \rightarrow x'\}$, el cual es $\{x \rightarrow x'\}$. La segunda asociación de x anula la primera.

5. Después de ejecutar la ligadura $x=2$ tenemos:

$$([(\{\text{Browse } x\}, \{x \rightarrow x'\}), (\{\text{Browse } x\}, \{x \rightarrow x\})], \\ \{x' = 2, x = 1\})$$

(Recuerde que $\langle d \rangle_2$ es un `Browse`.) Ahora vemos por qué las dos invocaciones a `Browse` muestran valores diferentes. Note que cada una se evalúa en un ambiente diferente. La declaración **local** interna tiene su ambiente propio, en el cual x referencia otra variable. Esto no afecta la declaración **local** externa, la cual conserva su ambiente sin importar lo que pase en otras instrucciones.

Definición e invocación de procedimientos

Nuestro siguiente ejemplo define e invoca el procedimiento `Max`, el cual calcula el máximo de dos números. Con la semántica podemos ver exáctamente qué pasa durante la definición y la ejecución de `Max`. A continuación está el código fuente del ejemplo, escrito con algunas abreviaciones sintácticas:

```
local Max C in
  proc {Max X Y ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  {Max 3 5 C}
end
```

Traduciéndolo a la sintaxis de lenguaje núcleo y reorganizándolo ligeramente resulta en

```

local Max in
  local A in
    local B in
      local C in
        Max=proc {$ X Y Z}
          local T in
            T=(X>=Y)
            <d>4 ≡ if T then Z=X else Z=Y end
            end
          end
          A=3
          B=5
          <d>2 ≡ {Max A B C}
          end
        end
      end
    end
  end
end

```

Usted puede ver que la sintaxis del programa en el lenguaje núcleo es bastante pródiga, debido a la simplicidad del lenguaje núcleo. Esta simplicidad es importante porque nos permite conservar igualmente simple la semántica. El código fuente original utiliza las tres siguientes abreviaciones sintácticas para legibilidad:

- La definición de más de una variable en una declaración **local**. Esto se traduce en declaraciones **local** anidadas.
- El uso de valores “en línea” en lugar de variables, e.g., **{P 3}** es una abreviación de **local X in X=3 {P X} end**.
- La utilización de operaciones anidadas, e.g., colocar la operación **X>=Y** en lugar del booleano en la declaración **if**.

Utilizaremos estas abreviaciones en los ejemplos de ahora en adelante.

Ejecutemos ahora la declaración $\langle d \rangle$. Por claridad, omitiremos algunas de las etapas intermedias.

1. El estado inicial de la ejecución es:

$([(\langle d \rangle, \emptyset)], \emptyset)$

Tanto el ambiente como el almacén están vacíos ($E = \emptyset$ y $\sigma = \emptyset$).

El modelo de computación declarativa

2. Después de ejecutar las cuatro declaraciones **local**, tenemos:

$$([(\langle d \rangle_1, \{ \text{Max} \rightarrow m, A \rightarrow a, B \rightarrow b, C \rightarrow c \})], \\ \{m, a, b, c\})$$

El almacén contiene las cuatro variables *m*, *a*, *b*, y *c*. El ambiente de $\langle d \rangle_1$ tiene asociaciones con esas variables.

3. Después de ejecutar las ligaduras de **Max**, **A**, y **B**, tenemos:

$$([(\{ \text{Max } A \ B \ C \}, \{ \text{Max} \rightarrow m, A \rightarrow a, B \rightarrow b, C \rightarrow c \})], \\ \{m = (\text{proc } \$ \ x \ y \ z \ \langle d \rangle_3 \ \text{end}, \emptyset), a = 3, b = 5, c\})$$

Ahora, las variables *m*, *a*, y *b* están ligadas a valores. El procedimiento está listo para ser invocado. Note que el ambiente contextual de **Max** está vacío pues no tiene ocurrencias libres de identificadores.

4. Después de ejecutar la invocación del procedimiento, tenemos:

$$([(\langle d \rangle_3, \{ x \rightarrow a, y \rightarrow b, z \rightarrow c \})], \\ \{m = (\text{proc } \$ \ x \ y \ z \ \langle d \rangle_3 \ \text{end}, \emptyset), a = 3, b = 5, c\})$$

Ahora, el ambiente de $\langle d \rangle_3$ tiene las asociaciones de los nuevos identificadores **x**, **y**, y **z**.

5. Después de ejecutar la comparación **x>=y**, tenemos:

$$([(\langle d \rangle_4, \{ x \rightarrow a, y \rightarrow b, z \rightarrow c, t \rightarrow t \})], \\ \{m = (\text{proc } \$ \ x \ y \ z \ \langle d \rangle_3 \ \text{end}, \emptyset), a = 3, b = 5, c, t = \text{false}\})$$

Aquí se agrega el nuevo identificador **t** y su variable *t* se ligó a **false**.

6. La ejecución se completa después de la declaración $\langle d \rangle_4$ (el condicional):

$$([], \{m = (\text{proc } \$ \ x \ y \ z \ \langle d \rangle_3 \ \text{end}, \emptyset), a = 3, b = 5, c = 5, t = \text{false}\})$$

La pila semántica queda vacía y **C** se liga a 5.

Procedimiento con referencias externas (parte 1)

Nuestro tercer ejemplo define e invoca el procedimiento **LowerBound**, el cual asegura que un número nunca tome un valor inferior a una cota inferior dada. El ejemplo es interesante porque **LowerBound** tiene una referencia externa. Miremos cómo se ejecuta el código siguiente:

```
local LowerBound Y C in
    Y=5
    proc {LowerBound X ?Z}
        if X>=Y then Z=X else Z=Y end
    end
    {LowerBound 3 C}
end
```

Este ejemplo es muy parecido al ejemplo de `Max`; tanto, que el cuerpo de `LowerBound` es idéntico al de `Max`. La única diferencia es que `LowerBound` tiene una referencia externa. El valor de tipo procedimiento es:

```
( proc { $ x z } if x>=y then z=x else z=y end end, { y → y } )
```

donde el almacén contiene:

$y = 5$

Cuando el procedimiento se define, i.e., cuando el valor de tipo procedimiento se crea, el ambiente tiene que contener una asociación de `y`. Ahora apliquemos este procedimiento. Suponemos que el procedimiento se invoca con `{LowerBound A C}`, donde `A` está ligada a 3. Antes de la aplicación tenemos:

```
( [({LowerBound A C}, {y → y, LowerBound → lb, A → a, C → c})],
  { lb = (proc { $ x z } if x>=y then z=x else z=y end end, {y → y}),
    y = 5, a = 3, c } )
```

Después de la aplicación tenemos:

```
( [({if x>=y then z=x else z=y end, {y → y, x → a, z → c}}),
  { lb = (proc { $ x z } if x>=y then z=x else z=y end end, {y → y}),
    y = 5, a = 3, c } )
```

El ambiente nuevo se calcula empezando con el ambiente contextual ($\{y \rightarrow y\}$ en el valor de tipo procedimiento), extendiéndolo con las asociaciones de los parámetros formales `x` y `z` con los argumentos actuales `a` y `c`.

Procedimiento con referencias externas (parte 2)

En la ejecución anterior, el identificador `y` referencia a `y` tanto en el ambiente en que se produce la invocación, como en el ambiente contextual de `LowerBound`. ¿Cómo cambiaría la ejecución si en lugar de `{LowerBound 3 C}` se ejecutara la declaración siguiente?:

```
local y in
  Y=10
  {LowerBound 3 C}
end
```

En este caso, `y` no referencia más a `y` en el ambiente de invocación. Antes de mirar la respuesta, por favor deje a un lado el libro, tome una hoja de papel, y trabájelo por su cuenta. Justo antes de la aplicación tenemos la misma situación de antes:

```
( [({LowerBound A C}, {y → y', LowerBound → lb, A → a, C → c})],
  { lb = (proc { $ x z } if x>=y then z=x else z=y end end, {y → y}),
    y' = 10, y = 5, a = 3, c } )
```

El modelo de computación declarativa

El ambiente de la invocación ha cambiado ligeramente: y referencia una nueva variable y' , la cual está ligada a 10. Cuando se realiza la aplicación, el ambiente nuevo se calcula exactamente en la misma forma que antes, a partir del ambiente contextual, extendiéndolo con los parámetros formales. ¡Esto significa que y' es ignorada! Llegamos exactamente a la misma situación que antes en la pila semántica:

```
( [([if X>=Y then Z=X else Z=Y end, {Y → y, X → a, Z → c}]),  
  { lb = (proc {$ X Z} if X>=Y then Z=X else Z=Y end end, {Y → y}),  
    y' = 10, y = 5, a = 3, c } )
```

El almacén aún contendrá la ligadura $y' = 10$. Sin embargo, y' no es referenciada por la pila semántica, de manera que su ligadura no hace ninguna diferencia en la ejecución.

2.5. Administración de la memoria

La máquina abstracta que definimos en la sección anterior es una herramienta poderosa con la cual podemos investigar propiedades de las computaciones. Como una primera exploración, miraremos el comportamiento de la memoria, i.e., cómo evolucionan los tamaños de la pila semántica y el almacén a medida que una computación progresá. Miraremos el principio de optimización de última invocación y lo explicaremos por medio de la máquina abstracta. Esto nos llevará a los conceptos de ciclo de vida de la memoria y de recolección de basura.

2.5.1. Optimización de última invocación

Considere un procedimiento recursivo con una sola invocación recursiva que resulta ser la última invocación en el cuerpo del procedimiento. Tales procedimientos se llaman recursivos por la cola. Mostraremos que la máquina abstracta ejecuta un procedimiento recursivo por la cola utilizando una pila de tamaño constante. Esta propiedad se llama optimización de última invocación o optimización de invocación de cola. El término optimización de recursión de cola se usa algunas veces, pero es menos preciso pues la optimización funciona para cualquier última invocación, no necesariamente invocaciones de recursión de cola (ver Ejercicios, sección 2.9). Considere el procedimiento siguiente:

```
proc {Ciclo10 I}  
  if I==10 then skip  
  else  
    {Browse I}  
    {Ciclo10 I+1}  
  end  
end
```

Invocar {Ciclo10 0} muestra los enteros, sucesivamente, del 0 al 9. Miremos cómo se ejecuta este procedimiento.

- El estado inicial de la ejecución es:

$$([(\{\text{Ciclo10 } 0\}, E_0)], \sigma)$$

donde E_0 es el ambiente donde se realiza la invocación y σ el almacén correspondiente.

- Después de ejecutar la declaración **if**, el estado evoluciona a:

$$([(\{\text{Browse } \texttt{i}\}, \{\texttt{i} \rightarrow i_0\}) (\{\text{Ciclo10 } \texttt{i+1}\}, \{\texttt{i} \rightarrow i_0\})], \{i_0 = 0\} \cup \sigma)$$

- Después de ejecutar el **Browse**, llegamos a la primera invocación recursiva:

$$([(\{\text{Ciclo10 } \texttt{i+1}\}, \{\texttt{i} \rightarrow i_0\})], \{i_0 = 0\} \cup \sigma)$$

- Después de ejecutar la declaración **if** en la invocación recursiva, el estado evoluciona a:

$$([(\{\text{Browse } \texttt{i}\}, \{\texttt{i} \rightarrow i_1\}) (\{\text{Ciclo10 } \texttt{i+1}\}, \{\texttt{i} \rightarrow i_1\})], \{i_0 = 0, i_1 = 1\} \cup \sigma)$$

- Después de ejecutar de nuevo el **Browse**, llegamos a la segunda invocación recursiva:

$$([(\{\text{Ciclo10 } \texttt{i+1}\}, \{\texttt{i} \rightarrow i_1\})], \{i_0 = 0, i_1 = 1\} \cup \sigma)$$

Es claro que la pila en la k -ésima invocación recursiva será siempre de la forma:

$$[(\{\text{Ciclo10 } \texttt{i+1}\}, \{\texttt{i} \rightarrow i_{k-1}\})]$$

Sólo hay una declaración semántica y su ambiente es de tamaño constante. Esto es lo que llamamos optimización de última invocación. Esto muestra que la forma eficiente de programar ciclos en el modelo declarativo es programar el ciclo como un procedimiento recursivo por la cola.

Podemos además ver que los tamaños de la pila semántica y del almacén evolucionan muy diferentemente. El tamaño de la pila semántica está acotado por una constante. Por el otro lado, el almacén se hace más grande en cada llamado. En la k -ésima invocación recursiva, el almacén tiene la forma:

$$\{i_0 = 0, i_1 = 1, \dots, i_{k-1} = k - 1\} \cup \sigma$$

El tamaño del almacén es proporcional al número de invocaciones recursivas. Miremos por qué este crecimiento no es un problema en la práctica. Mire cuidadosamente la pila semántica en la k -ésima invocación recursiva. Ella no necesita las variables $\{i_0, i_1, \dots, i_{k-2}\}$. La única variable que necesita es i_{k-1} . Esto significa que podemos eliminar del almacén las variables no necesitadas, sin cambiar los resultados de la computación. Esto nos lleva a un almacén más pequeño:

$$\{i_{k-1} = k - 1\} \cup \sigma$$

Este almacén más pequeño, es de tamaño constante. Si pudiéramos asegurar, de alguna manera, que las variables no necesitadas sean eliminadas siempre, entonces el programa se podría ejecutar indefinidamente utilizando una cantidad constante de memoria.

Para este ejemplo, podemos resolver el problema almacenando las variables en la pila en lugar de hacerlo en el almacén. Esto es posible porque las variables están ligadas a enteros pequeños que caben en una palabra de la máquina (e.g., 32 bits). Por ello, la pila puede almacenar los enteros directamente en lugar de almacenar referencias a variables del almacén.¹⁶ Este ejemplo es atípico; casi todos los programas reales tienen grandes cantidades de datos antiguos (de larga vida) o compartidos que no se pueden almacenar fácilmente en la pila. Por tanto, aún tenemos que resolver el problema general de eliminación de las variables no necesitadas. En la próxima sección veremos cómo hacerlo.

2.5.2. Ciclo de vida de la memoria

A partir de la semántica de la máquina abstracta se concluye que un programa en ejecución sólo necesita la información de la pila semántica y de la parte del almacén alcanzable desde ésta. Un valor parcial es alcanzable si está referenciado por una declaración de la pila semántica o por otro valor parcial alcanzable. La pila semántica y la parte alcanzable del almacén, juntas, se llaman la memoria activa. Toda la memoria que no es activa puede ser recuperada de manera segura, i.e., puede ser reutilizada en la computación. Vimos que el tamaño de la memoria activa del ejemplo `Ciclo10` está acotada por una constante, lo cual significa que puede permanecer en el ciclo indefinidamente sin llegar a acabar la memoria del sistema.

Ahora podemos introducir el concepto de ciclo de vida de la memoria. Un programa se ejecuta en la memoria principal, la cual consiste de una secuencia de palabras de memoria. En el momento de escribir este libro, los computadores personales de bajo costo tienen palabras de 32 bits, mientras que los computadores de alto desempeño tienen palabras de 64 bits o más largas. La secuencia de palabras se divide en bloques, donde un bloque consiste de una secuencia de una o más palabras utilizadas para almacenar parte de un estado de la ejecución. Un bloque es la unidad básica de asignación de memoria. La figura 2.18 muestra el ciclo de vida de un bloque de memoria. Cada bloque de memoria se mueve continuamente en un ciclo de tres estados: memoria activa, memoria inactiva y memoria libre. La administración de la memoria es la tarea de asegurarse que los bloques circulan correctamente a lo largo de este ciclo. A un programa en ejecución que necesite un

16. Otra optimización que se usa frecuentemente consiste en almacenar parte de la pila en registros de la máquina. Esto es importante pues es mucho más rápido leer y escribir en registros de la máquina que en la memoria principal.

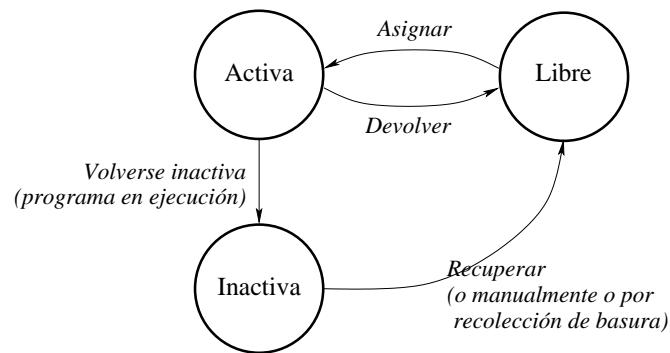


Figura 2.18: Ciclo de vida de un bloque de memoria.

bloque se le asignará uno, tomado de un conjunto de bloques de memoria libre. El bloque entonces se vuelve activo. Durante su ejecución, es posible que el programa no necesite más de algunos de los bloques de memoria que tiene asignados:

- Si el programa puede determinar esto directamente, entonces devuelve los bloques, haciendo que estos se vuelvan libres inmediatamente. Esto es lo que pasa con la pila semántica en el ejemplo de [ciclo10](#).
- Si el programa no puede determinar esto directamente, entonces los bloques se vuelven inactivos. Estos bloques ya no pueden ser alcanzados por el programa en ejecución, pero el programa no lo sabe, por lo cual no puede liberarlos. Esto es lo que pasa con el almacén en el ejemplo de [ciclo10](#).

Normalmente, los bloques de memoria usados para administrar flujo de control (la pila semántica) pueden ser devueltos, mientras que los bloques de memoria usados para estructuras de datos (el almacén) se vuelven inactivos.

La memoria inactiva debe ser finalmente recuperada, i.e., el sistema debe reconocer que está inactiva y devolverla a la memoria libre. De no ser así, el sistema tiene una pérdida de memoria y rápidamente se quedará sin memoria. La recuperación de la memoria inactiva es la parte más difícil de la administración de la memoria, pues reconocer que la memoria es inalcanzable es una condición global que depende del estado completo de la ejecución del programa en ejecución. Los lenguajes de bajo nivel como C o C++ frecuentemente dejan la tarea de recuperación al programador, lo cual es una gran fuente de errores en los programas. Hay dos tipos de errores que pueden ocurrir:

- *Referencia suelta.* Esto sucede cuando un bloque se recupera a pesar de ser alcanzable aún. El sistema reutilizará finalmente este bloque. Esto significa que las estructuras de datos se corromperán de maneras impredecibles, llevando a la terminación abrupta del programa. Este error es especialmente pernicioso pues el efecto (la terminación abrupta) se produce normalmente lejos de la causa (la

recuperación incorrecta). Por esto las referencias sueltas son difíciles de depurar.

■ *Memoria insuficiente*. Esto ocurre cuando no se recuperan bloques a pesar de ser inalcanzables. El efecto es que el tamaño de la memoria activa crece indefinidamente hasta que los recursos en memoria del sistema se agotan. La falta de memoria es menos peligrosa que las referencias sueltas pues los programas pueden continuar ejecutándose por un tiempo, antes que el error los force a detenerse. Los programas de larga vida, como sistemas operativos y servidores, no deben tener pérdidas de memoria.

2.5.3. Recolección de basura

Muchos lenguajes de alto nivel, tales como Erlang, Haskell, Java, Lisp, Prolog, Smalltalk, y otros, realizan la recuperación de forma automática. Es decir, la recuperación es hecha por el sistema independientemente del programa que se esté ejecutando. Esto elimina completamente las referencias sueltas y reduce grandemente las pérdidas de memoria. Además, esto releva al programador de la mayoría de las dificultades de la administración manual de la memoria. A la recuperación automática se le denomina recolección de basura. La recolección de basura es una técnica muy conocida que ha sido utilizada por mucho tiempo. Fue utilizada a principios de los años 1960 en sistemas Lisp. Hasta los años 1990, los principales lenguajes no utilizaron la técnica porque fue juzgada (erróneamente) como demasiado inefficiente. Finalmente, ha sido adoptada, debido a la popularidad del lenguaje Java.

Un recolector de basura típico tiene dos fases. En la primera fase, determina cuál es la memoria activa. Lo hace buscando todas las estructuras de datos alcanzables a partir de un conjunto inicial de punteros denominado el conjunto raíz. El significado original de “puntero” es una dirección en el espacio de direcciones de un proceso. En el contexto de nuestra máquina abstracta, un puntero es una referencia a una variable en el almacén. El conjunto raíz es el conjunto de punteros que son necesitados siempre por el programa. En la máquina abstracta de la sección anterior, el conjunto raíz es simplemente la pila semántica. En general, el conjunto raíz incluye todos los punteros a hilos listos y todos los punteros a estructuras de datos del sistema operativo. Esto lo veremos cuando extendamos la máquina abstracta, en capítulos posteriores, para implementar conceptos nuevos. El conjunto raíz también incluye algunos punteros relacionados con la programación distribuida (a saber, referencias de sitios remotos; ver capítulo 11 (en CTM)).

En la segunda fase, el recolector de basura compacta la memoria. Es decir, toma, por un lado, todos los bloques de memoria activa y, por otro lado, todos los bloques de memoria libre y los coloca, cada uno, en un solo bloque contiguo.

Los algoritmos modernos de recolección de basura son suficientemente eficientes al punto que la mayoría de aplicaciones pueden usarlos con escasas penalidades en espacio y tiempo [83]. Los recolectores de basura más ampliamente utilizados se ejecutan en modo “batch”, i.e., ellos están durmiendo la mayor parte del tiempo y se ejecutan solamente cuando las memorias activa e inactiva alcanzan un umbral predefinido. Mientras el recolector de basura se ejecuta, hay una pausa en la

ejecución del programa. Normalmente la pausa es suficientemente pequeña y no alcanza a ser considerada interruptora.

Existen algoritmos de recolección de basura, denominados recolectores de basura en tiempo real, que pueden ejecutarse continuamente, intercalados con la ejecución del programa. Estos se usan en casos como la programación fuerte en tiempo real, en la cual no puede existir ningún tipo de pausas.

2.5.4. La recolección de basura no es magia

Tener recolección de basura alivia la carga del desarrollador en la administración de memoria, pero no la elimina por completo. Hay dos casos que quedan en responsabilidad del desarrollador: evitar las pérdidas de memoria y administrar los recursos externos.

Evitar pérdidas de memoria

El programador tiene aún alguna responsabilidad en lo que concierne a pérdidas de memoria. Si el programa continúa referenciando una estructura de datos que no se necesita más, entonces la memoria ocupada por esa estructura de datos no podrá ser recuperada nunca. El programa debería tener el cuidado de dejar de hacer referencia a todas las estructuras de datos que no se necesitan más.

Por ejemplo, tome una función recursiva que recorre una lista. Si la cabeza de la lista se pasa en las invocaciones recursivas, entonces la memoria correspondiente a la lista no será recuperada durante la ejecución de la función. A continuación, un ejemplo:

```
L=[1 2 3 ... 1000000]  
  
fun {Sum X L1 L}  
  case L1 of Y|L2 then {Sum X+Y L2 L}  
  else X end  
end  
  
{Browse {Sum 0 L L}}
```

Sum suma los elementos de una lista. Pero también guarda una referencia a L, la lista original, aunque no la necesita para sus cálculos. Esto significa que L permanecerá en la memoria durante toda la ejecución de Sum. Una definición mejor es la siguiente:

```
fun {Sum X L1}  
  case L1 of Y|L2 then {Sum X+Y L2}  
  else X end  
end  
  
{Browse {Sum 0 L}}
```

Aquí, la referencia a L se pierde inmediatamente. Este ejemplo es trivial, pero las cosas pueden ser más sutiles. Por ejemplo, considere una estructura de datos activa S que contiene una lista de otras estructuras de datos D₁, D₂, ..., D_n. Si una de

El modelo de computación declarativa

ellas, digamos `di`, no la necesita más el programa, entonces debería ser eliminada de la lista. De otra manera, la memoria correspondiente nunca será recuperada.

Un programa bien escrito por tanto, tiene que realizar algo de “limpieza” sobre sí mismo, asegurando que no siga referenciando a estructuras de datos que no necesita más. La limpieza debe realizarse en el modelo declarativo, pero esto es incómodo.¹⁷

Administrar los recursos externos

Un programa Mozart utiliza frecuentemente estructuras de datos que son externas a su proceso de operación. LLamamos a tal estructura de datos un recurso externo. Los recursos externos afectan la administración de la memoria de dos maneras. Un estructura de datos interna de Mozart puede referenciar un recurso externo, y viceversa. En ambos casos se necesita la intervención del programador. Consideremos cada caso separadamente.

El primer caso es cuando una estructura de datos de Mozart referencia un recurso externo. Por ejemplo, un registro puede corresponder a una entidad gráfica en una pantalla o a un archivo abierto en un sistema de archivos. Si el registro no se necesita más, entonces la entidad gráfica tiene que ser eliminada o el archivo tiene que ser cerrado. De no ser así, la pantalla o el sistema de archivos generarán memoria perdida. Esto se hace con una técnica llamada finalización, la cual define las acciones que se deben realizar una vez la estructura de datos se vuelve inalcanzable. La técnica de finalización se explica en la sección 6.9.2.

El segundo caso es cuando un recurso externo necesita de una estructura de datos de Mozart. Esto es frecuentemente sencillo de manejar. Por ejemplo, considere un escenario donde el programa Mozart implementa un servidor de una base de datos que es accedida por clientes externos. Este escenario tiene una solución simple: nunca realice recuperación automática de lo almacenado en la base de datos. Otros escenarios pueden no ser tan simples. Una solución general es hacer aparte la parte del programa Mozart que representa el recurso externo. Esta parte debe estar activa (i.e., tiene su hilo propio) de manera que no se recupera caóticamente. Puede ser vista como un “proxy” para el recurso. El proxy guarda una referencia a la estructura de datos Mozart tanto tiempo como el recurso la necesite. El recurso informa al proxy cuando no la necesite más. En la sección 6.9.2 se presenta otra técnica.

2.5.5. El recolector de basura de Mozart

El sistema Mozart realiza administración automática de la memoria. Cuenta, tanto con un recolector de basura local, como con un recolector de basura distribuido. El último se utiliza en programación distribuida y se explica en el capítulo 11

17. Es más eficiente hacerlo con estado explícito (ver capítulo 6).

(en CTM). El recolector de basura local utiliza un algoritmo denominado *copying dual-space algorithm*.

El recolector de basura divide la memoria en dos espacios, de los cuales cada uno toma la mitad del espacio disponible de memoria. En todo momento, el programa en ejecución reside, completamente, en una mitad. La recolección de basura se realiza cuando no hay más memoria libre en esa mitad. El recolector de basura busca todas las estructuras de datos que son alcanzables a partir del conjunto raíz y las copia todas en la otra mitad. Como se copian en bloques de memoria contiguos, se realiza al mismo tiempo la compactación.

La ventaja de un recolector de basura por copias es que su tiempo de ejecución es proporcional al tamaño de la memoria activa, y no al tamaño total de la memoria. En los programas pequeños la recolección de basura será rápida, aún si se están ejecutando en un espacio de memoria amplio. Las dos desventajas de un recolector de basura por copias son que la mitad de la memoria es inutilizable en todo momento y que las estructuras de datos de larga vida (como tablas del sistema) tienen que ser copiadas en cada recolección de basura. Miremos cómo eliminar estas dos desventajas. La copia de datos de larga vida puede ser evitada utilizando un algoritmo modificado llamado un recolector de basura generacional. Éste hace una partición de la memoria en generaciones. Las estructuras de datos de larga vida se colocan en las generaciones más antiguas, las cuales se recolectan menos frecuentemente.

La primera desventaja (memoria inutilizada) sólo es importante si el tamaño de la memoria activa se acerca al máximo tamaño direccionable en la arquitectura subyacente. Actualmente la tecnología de los computadores está en un período de transición de direccionamiento de 32 bits a 64 bits. En un computador con direcciones de 32 bits, el límite se alcanza cuando la memoria activa llega a 1000 MB o más. (El límite no es normalmente 2^{32} bytes, i.e., 4096 MB, debido a las limitaciones en el sistema operativo.) En el momento de escribir el libro, este límite lo alcanzan programas grandes en computadores de alto desempeño. Para tales programas, recomendamos usar un computador con direcciones de 64 bits, las cuales no tienen este problema.

2.6. Del lenguaje núcleo a un lenguaje práctico

El lenguaje núcleo tiene todos los conceptos necesarios para hacer programación declarativa. Pero al tratar de usarlo para esto en la práctica, se muestra demasiado restrictivo (en términos de sintaxis). Los programas en lenguaje núcleo son muy pródigos en sintaxis. La mayor parte de ésta puede ser eliminada añadiendo jocosamente azúcar sintáctico y abstracciones lingüísticas. En esta sección se hace exactamente eso:

- Se define un conjunto de conveniencias sintácticas que llevan a una sintaxis completa más concisa y legible.
- Se define una abstracción lingüística importante, a saber, las funciones, lo que es muy útil para una programación concisa y legible.
- Se explica la interfaz interactiva del sistema Mozart y se muestra cómo se relaciona con el modelo declarativo. Aparece entonces la declaración **declare**, diseñada para uso interactivo, la cual es una variante de la declaración **local**.

El lenguaje resultante es utilizado en el capítulo 3 para explicar las técnicas de programación del modelo declarativo.

2.6.1. Conveniencias sintácticas

El lenguaje núcleo define una sintaxis sencilla para todas sus construcciones y tipos. Para tener una sintaxis más usable, el lenguaje completo tiene las conveniencias sintácticas siguientes:

- Se pueden escribir valores parciales anidados en una forma concisa.
- A las variables se les puede definir y dar un valor inicial en un solo paso.
- Se pueden escribir expresiones en una forma concisa.
- Se pueden anidar las declaraciones **if** y **case** en una forma concisa.
- Se definen los operadores **andthen** y **orelse** para declaraciones **if** anidadas.
- Las declaraciones se pueden convertir en expresiones usando una marca de anidamiento.

Los símbolos no terminales utilizados en la sintaxis y en la semántica del lenguaje núcleo corresponden a los de la sintaxis del lenguaje completo, de la manera que se muestra a continuación:

Sintaxis del núcleo	Sintaxis del lenguaje completo
$\langle x \rangle, \langle y \rangle, \langle z \rangle$	$\langle \text{variable} \rangle$
$\langle d \rangle$	$\langle \text{declaración} \rangle, \langle \text{decl} \rangle$

Valores parciales anidados

En la tabla 2.2, la sintaxis de los registros y los patrones exige que sus argumentos sean variables. En la práctica, se anidan mucho más profundamente que lo que esto permite. Como los valores anidados se usan frecuentemente, presentamos un azúcar sintáctico para ellos. Por ejemplo, extendemos la sintaxis para poder escribir `X=persona(nombre:"Jorge" edad:25)` en lugar de la versión más voluminosa:

```
local A B in A="Jorge" B=25 X=persona(nombre:A edad:B) end
```

donde `X` se liga al registro anidado.

Inicialización implícita de variables

Para lograr programas más cortos y más fáciles de leer, existe un azúcar sintáctico para ligar una variable inmediatamente se define. La idea es colocar una operación de ligadura entre el `local` y el `in`. En lugar de `local x in x=10 {Browse x} end`, en el cual `x` se menciona tres veces, la abreviación nos deja escribir `local x=10 in {Browse x} end`, en el cual `x` se menciona sólo dos veces. Un caso sencillo es el siguiente:

```
local x=<expresión> in <declaración> end
```

En éste se declara `x` y se liga al resultado de `<expresión>`. El caso general es:

```
local <patrón>=<expresión> in <declaración> end
```

donde `<patrón>` es cualquier valor parcial. Éste define primero todas las variables presentes en el `<patrón>` y luego liga `<patrón>` con el resultado de `<expresión>`. La regla general en ambos ejemplos es que los identificadores de variables que aparecen en el lado izquierdo de la igualdad, i.e., `x` o los identificadores en `<patrón>`, son los identificadores que se definen. Los identificadores de variables que aparecen en el lado derecho de la igualdad no se definen.

La inicialización implícita de variables es conveniente para construir una estructura de datos compleja, donde necesitemos tener referencias a variables dentro de la estructura. Por ejemplo, si `T` es no-ligada, entonces la declaración:

```
local árbol(llave:A izq:B der:C valor:D)=T in <declaración> end
```

construye el registro `árbol`, lo liga a `T`, y define `A`, `B`, `C`, y `D` como referencias a partes de `T`. Esto es estrictamente equivalente a:

```
local A B C D in
  T=árbol(llave:A izq:B der:C valor:D) <declaración>
end
```

Es interesante comparar la inicialización implícita de variables con la declaración `case`. Ambas usan patrones e implícitamente definen variables. La primera usa los patrones para construir estructuras de datos mientras que la segunda los usa para

```

⟨expresión⟩ ::= ⟨variable⟩ | ⟨ent⟩ | ⟨float⟩
| ⟨opUnario⟩ ⟨expresión⟩
| ⟨expresión⟩ ⟨opBinEval⟩ ⟨expresión⟩
| `(` `⟨expresión⟩ `)`
| `{` `{⟨expresión⟩ {⟨expresión⟩ } `}`
| ...
⟨opUnario⟩ ::= `~` | ...
⟨opBinEval⟩ ::= `+` | `-` | `*` | `/` | div | mod
| `==` | `!=` | `<` | `=<` | `>` | `=>` | ...

```

Tabla 2.4: Expresiones para calcular con números.

```

⟨declaración⟩ ::= if ⟨expresión⟩ then ⟨declaraciónEn⟩
{ elseif ⟨expresión⟩ then ⟨declaraciónEn⟩ }
[ else ⟨declaraciónEn⟩ ] end
| ...
⟨declaraciónEn⟩ ::= [ { ⟨parteDeclaración⟩ }+ in ] ⟨declaración⟩

```

Tabla 2.5: La declaración **if**.

extraer datos de las estructuras de datos.¹⁸

Expresiones

Una expresión es el azúcar sintáctico para una sucesión de operaciones que devuelven un valor. Es diferente de una declaración, la cual también es una secuencia de operaciones, pero no devuelve un valor. Una expresión puede ser usada en cualquier sitio de una declaración donde se necesite un valor. Por ejemplo, `11*11` es una expresión y `x=11*11` es una declaración. Semánticamente, una expresión se define por una traducción directa en la sintaxis del lenguaje núcleo. En este caso, `x=11*11` se traduce en `{Mul 11 11 x}`, donde `Mul` es un procedimiento de tres argumentos que realiza la multiplicación.¹⁹

18. La inicialización implícita de variables también puede ser usada para extraer datos de las estructuras de datos. Si `T` ya está ligado a un registro `árbol`, entonces sus cuatro campos serán ligados a `A`, `B`, `C`, y `D`. Esto funciona porque la operación de ligadura está realizando realmente una unificación, la cual es una operación simétrica (ver sección 2.8.2). Nosotros no recomendamos este uso.

19. Su nombre real es `Number.^*`, pues hace parte del módulo `Number`.

```

⟨declaración⟩ ::= case ⟨expresión⟩
    of ⟨patrón⟩ [ andthen ⟨expresión⟩ ] then ⟨declaraciónEn⟩
        { ‘[ ]’ ⟨patrón⟩ [ andthen ⟨expresión⟩ ] then ⟨declaraciónEn⟩ }
        [ else ⟨declaraciónEn⟩ ] end
        | ...
⟨patrón⟩     ::= ⟨variable⟩ | ⟨átomo⟩ | ⟨ent⟩ | ⟨flot⟩
                | ⟨string⟩ | unit | true | false
                | ⟨etiqueta⟩ ‘( ‘ { [ ⟨campo⟩ ‘:’ ] ⟨patrón⟩ } [ ‘...’ ] ‘)’
                | ⟨patrón⟩ ⟨consOpBin⟩ ⟨patrón⟩
                | ‘[ ’ { ⟨patrón⟩ } + ‘] ’
⟨consOpBin⟩ ::= ‘#’ | ‘|’

```

Tabla 2.6: La declaración **case**.

En la tabla 2.4 se muestra la sintaxis de las expresiones para calcular con números. Más adelante veremos expresiones para calcular con otros tipos de datos. La expresiones se construyen jerárquicamente, a partir de expresiones básicas (e.g., variables y números) y combinándolas entre ellas. Hay dos formas de combinarlas: usando operadores (e.g., la suma `1+2+3+4`) o usando invocación de funciones (e.g., la raíz cuadrada `{Sqrt 5.0}`).

Declaraciones if y case anidadas

Ahora añadimos azúcar sintáctico para escribir más fácilmente declaraciones **if** y **case** con múltiples alternativas y condiciones complicadas. En la tabla 2.5 se presenta la sintaxis completa de la declaración **if**. En la tabla 2.6 se presenta la sintaxis completa de la declaración **case** y sus patrones. (Algunos de los no terminales en estas tablas están definidos en el apéndice C.) Estas declaraciones se traducen en las declaraciones **if** y **case** del lenguaje núcleo. Aquí presentamos un ejemplo de una declaración **case** en sintaxis completa:

```

case Xs#Ys
of nil#Ys then ⟨d⟩1
[] Xs#nil then ⟨d⟩2
[] (X|Xr) #(Y|Yr) andthen X=<Y then ⟨d⟩3
else ⟨d⟩4 end

```

Consiste de una secuencia de casos alternativos delimitados con el símbolo “[”. Las alternativas se suelen llamar cláusulas. Esta declaración se traduce en la sintaxis del lenguaje núcleo de la manera siguiente:

El modelo de computación declarativa

```

case Xs of nil then ⟨d⟩1
else
  case Ys of nil then ⟨d⟩2
  else
    case Xs of X|Xr then
      case Ys of Y|Yr then
        if X=<Y then ⟨d⟩3 else ⟨d⟩4 end
        ⟨d⟩4 end
      ⟨d⟩4 end
    end
  end

```

La traducción ilustra una propiedad importante de la declaración **case** en sintaxis completa: las cláusulas se prueban secuencialmente empezando con la primera cláusula. La ejecución continúa con una cláusula siguiente sólo si el patrón de la cláusula es inconsistente con el argumento de entrada.

Los patrones se manejan mirando primero los patrones más externos y luego continuando hacia adentro. El patrón anidado $(X|X_r) \# (Y|Y_r)$ tiene un patrón externo de la forma $A\#B$ y dos patrones internos de la forma $A|B$. Los tres patrones son tuplas que están escritas con operadores en sintaxis infija, usando los operadores infijos $\#$ y $|$. Estos patrones se hubieran podido escribir con la sintaxis normal como $\#(A B)$ y $|(A B)$. Cada patrón interno $(X|X_r)$ y $(Y|Y_r)$ se coloca en su propia declaración **case** primitiva. El patrón más externo que usa $\#$ desaparece de la traducción porque aparece también en el argumento de entrada del **case**. Por lo tanto, el reconocimiento del operador $\#$ se puede hacer en el momento de la traducción.

Los operadores andthen y orelse

Los operadores **andthen** y **orelse** son utilizados en cálculos con valores booleanos. La expresión

$\langle \text{expresión} \rangle_1 \text{ andthen } \langle \text{expresión} \rangle_2$

se traduce en

$\text{if } \langle \text{expresión} \rangle_1 \text{ then } \langle \text{expresión} \rangle_2 \text{ else false end}$

La ventaja de usar **andthen** es que $\langle \text{expresión} \rangle_2$ no es evaluada si $\langle \text{expresión} \rangle_1$ es **false**. Existe un operador análogo **orelse**. La expresión

$\langle \text{expresión} \rangle_1 \text{ orelse } \langle \text{expresión} \rangle_2$

se traduce en

$\text{if } \langle \text{expresión} \rangle_1 \text{ then true else } \langle \text{expresión} \rangle_2 \text{ end}$

Es decir, $\langle \text{expresión} \rangle_2$ no es evaluada si $\langle \text{expresión} \rangle_1$ es **true**.

<code><declaración></code>	<code>::= fun {`<variable> {<patrón>} `} `<expresiónEn> end</code>
	<code> ...</code>
<code><expresión></code>	<code>::= fun {`<`\$` {<patrón>} `}` `<expresiónEn> end</code>
	<code> proc {`<`\$` {<patrón>} `}` `<declaraciónEn> end</code>
	<code> `<` {`<expresión> {<expresión>} `}`</code>
	<code> local {<parteDeclaración>} + in <expresión> end</code>
	<code> if <expresión> then <expresiónEn></code>
	<code>{ elseif <expresión> then <expresiónEn> }</code>
	<code>[else <expresiónEn>] end</code>
	<code> case <expresión></code>
	<code>of <patrón> [andthen <expresión>] then <expresiónEn></code>
	<code>{ `<` {<patrón>} [andthen <expresión>] then <expresiónEn> }</code>
	<code>[else <expresiónEn>] end</code>
	<code> ...</code>
<code><declaraciónEn></code>	<code>::= [{<parteDeclaración>} + in] <declaración></code>
<code><expresiónEn></code>	<code>::= [{<parteDeclaración>} + in] [<declaración>] <expresión></code>

Tabla 2.7: Sintaxis de función.

Marcas de anidamiento

La marca de anidamiento “\$” convierte cualquier declaración en una expresión. El valor de la expresión es lo que está en la posición indicada por la marca de anidamiento. Por ejemplo, la declaración `{P x1 x2 x3}` puede ser convertida en `{P x1 $ x3}`, la cual es una expresión cuyo valor es `x2`. Como se evita la definición y el uso del identificador `x2`, el código fuente queda más conciso. La variable correspondiente a `x2` se oculta en el código fuente.

Las marcas de anidamiento pueden lograr que el código fuente sea más fácil de leer para un programador proficiente, mientras que es más difícil para un principiante pues tiene qué pensar en cómo se traduce el código en el lenguaje n\'ucleo. Nosotros los usamos sólo si incrementan grandemente la legibilidad. Por ejemplo, en lugar de escribir

```
local x in {Obj get(x)} {Browse x} end
```

escribiremos `{Browse {Obj get($)}}`. Una vez uno se acostumbra a las marcas de anidamiento, estas se vuelven tanto claras como concisas. Note que la sintaxis de los valores de tipo procedimiento, tal como se explicó en la sección 2.3.3, es consistente con la sintaxis de las marcas de anidamiento.

2.6.2. Funciones (la declaración **fun**)

El modelo declarativo provee una abstracción lingüística para programar con funciones. Este es nuestro primer ejemplo de abstracción lingüística, tal como fue definida en la sección 2.1.2. Ahora definimos la nueva sintaxis para definición e invocación de funciones y mostramos cómo se traducen en el lenguaje núcleo.

Definición de funciones

Una definición de función difiere de una definición de procedimiento en dos cosas: se introduce con la palabra reservada **fun** y el cuerpo debe terminar con una expresión. Por ejemplo, una definición sencilla es:

```
fun {F X1 ... XN} <declaración> <expresión> end
```

Esto se traduce en la siguiente definición de procedimiento:

```
proc {F X1 ... XN ?R} <declaración> R=<expresión> end
```

El argumento adicional **R** se liga a la expresión en el cuerpo del procedimiento. Si el cuerpo de la función es una expresión **if**, entonces cada alternativa de éste puede terminar en una expresión:

```
fun {Max X Y}  
  if X>=Y then X else Y end  
end
```

Esto se traduce a:

```
proc {Max X Y ?R}  
  R = if X>=Y then X else Y end  
end
```

Podemos traducir esto aún más, transformando la expresión **if** a una declaración. Esto produce el resultado final:

```
proc {Max X Y ?R}  
  if X>=Y then R=X else R=Y end  
end
```

Reglas similares se aplican para las declaraciones **local** y **case**, y para otras declaraciones que veremos más adelante. Toda declaración se puede usar como una expresión. Informalmente hablando, cuando en un procedimiento una secuencia de ejecución termina en una declaración, la correspondiente secuencia en una función termina en una expresión. En la tabla 2.7 se presenta la sintaxis completa de las expresiones después de aplicar esta regla. En particular, también existen valores de tipo función, los cuales no son más que valores de tipo procedimiento escritos en sintaxis funcional.

Invocación de funciones

Una invocación a una función $\{F\ X_1 \dots X_N\}$ se traduce en la invocación a un procedimiento $\{F\ X_1 \dots X_N\ R\}$, donde R reemplaza la invocación de función donde se use. Por ejemplo, la siguiente invocación anidada de F :

```
{Q {F X1 ... XN} ...}
```

se traduce en:

```
local R in
  {F X1 ... XN R}
  {Q R ...}
end
```

En general, las invocaciones anidadas de funciones se evalúan antes de la invocación de la función donde están anidadas. Si hay varias, se evalúan en el orden en que aparecen en el programa.

Invocación de funciones en estructuras de datos

Hay una regla más para recordar en cuanto a invocación de funciones. Tiene que ver con una invocación dentro de una estructura de datos (registro, tupla, o lista). Aquí hay un ejemplo:

```
Ys={F X} | {Map Xr F}
```

En este caso, la traducción coloca las invocaciones anidadas después de la operación de ligadura:

```
local Y Yr in
  Ys=Y|Yr
  {F X Y}
  {Map Xr F Yr}
end
```

Esto asegura que la invocación recursiva quede de último. En la sección 2.5.1 se explica por qué esto es importante para la eficiencia de la ejecución. La función `Map` completa se define como sigue:

```
fun {Map Xs F}
  case Xs
  of nil then nil
  [] X|Xr then {F X} | {Map Xr F}
  end
end
```

`Map` aplica la función `F` a todos los elementos de una lista y devuelve como resultado una lista con cada uno de los resultados de esa aplicación. Este es un ejemplo de invocación:

```
{Browse {Map [1 2 3 4] fun {$ X} X*X end}}
```

$\langle \text{declaraciónInter} \rangle ::= \langle \text{declaración} \rangle$ $ \textbf{declare} \{ \langle \text{parteDeclaración} \rangle \} + [\langle \text{declaraciónInter} \rangle]$ $ \textbf{declare} \{ \langle \text{parteDeclaración} \rangle \} + \textbf{in} \langle \text{declaraciónInter} \rangle$ $\langle \text{parteDeclaración} \rangle ::= \langle \text{variable} \rangle \langle \text{patrón} \rangle \sim \langle \text{expresión} \rangle \langle \text{declaración} \rangle$

Tabla 2.8: Sintaxis de las declaraciones interactivas.

Esto muestra [1 4 9 16]. La definición de `Map` se traduce al lenguaje núcleo como sigue:

```
proc {Map Xs F ?Ys}
  case Xs of nil then Ys=nil
  else case Xs of X|Xr then
    local Y Yr in
      Ys=Y|Yr {F X Y} {Map Xr F Yr}
    end
  end end
end
```

La variable de flujo de datos `Yr` se utiliza como receptor del resultado de la invocación recursiva `{Map Xr F Yr}`. Así, la invocación recursiva es la última invocación. En nuestro modelo, esto significa que la recursión se ejecuta con la misma eficiencia en tiempo y en espacio que una construcción iterativa como un ciclo `while`.

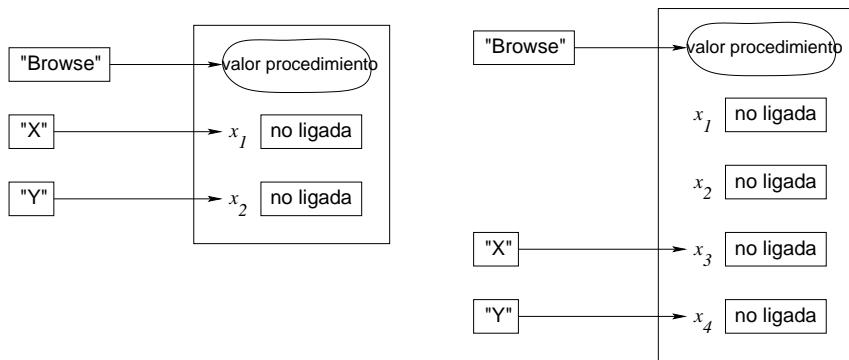
2.6.3. La interfaz interactiva (la declaración `declare`)

El sistema Mozart cuenta con una interfaz interactiva que permite la introducción incremental de fragmentos de programa y ejecutarlos a medida que se van introduciendo. Los fragmentos tienen que respetar la sintaxis de las declaraciones interactivas, la cual se presenta en la tabla 2.8. Una declaración interactiva es o una declaración legal o una nueva forma, la declaración `declare`. Suponemos que el usuario alimenta el sistema con declaraciones interactivas, una por una. (En los ejemplos presentados a lo largo del libro, la declaración `declare` se deja de lado frecuentemente. Ella debería añadirse si el ejemplo declara variables nuevas.)

La interfaz interactiva sirve para mucho más que sólo alimentarla con declaraciones. Ella tiene toda la funcionalidad necesaria para desarrollar *software*. En el apéndice A se presenta un resumen de parte de esta funcionalidad. Por ahora, supondremos que el usuario sólo conoce como alimentarla con declaraciones.

La interfaz interactiva cuenta con un único ambiente global. La declaración `declare` añade asociaciones nuevas en este ambiente. Por ello, `declare` únicamente puede ser usado interactivamente, nunca en aplicaciones autónomas. Al alimentar la interfaz con la definición siguiente:

```
declare X Y
```



Resultado del primer declare X Y Resultado del segundo declare X Y

Figura 2.19: Declaración de variables globales.

se crean dos variables nuevas en el almacén, x_1 y x_2 , y se agregan al ambiente las asociaciones de x y y a ellas. Como las asociaciones se realizan en el ambiente global decimos que x y y son variables globales o variables interactivas. Alimentar la interfaz una segunda vez con la misma definición, causará que x y y se asocien a otras dos variables nuevas, x_3 y x_4 . En la figura 2.19 se muestra lo que pasa. Las variables originales x_1 y x_2 , están aún en el almacén, pero ya no son referenciadas por x y y . En la figura, **Browse** se asocia con un valor de tipo procedimiento que implementa el browser. La declaración **declare** agrega variables y asociaciones nuevas, pero deja inalteradas las variables existentes en el almacén.

Agregar una nueva asociación a un identificador que ya tenía otra asociación con una variable puede ocasionar que la variable quede inalcanzable, si no existen otras referencias a ella. Si la variable hace parte de un cálculo, entonces ella queda aún alcanzable dentro de ese cálculo. Por ejemplo:

```
declare X Y
X=25
declare A
A=persona(edad:X)
declare X Y
```

Justo después de la ligadura $X=25$, x se asocia a 25, pero después del segundo **declare X Y** se asocia a una nueva variable no-ligada. El 25 todavía es alcanzable a través de la variable A , la cual se liga al registro $persona(edad:25)$. El registro contiene 25 porque x se asoció a 25 cuando la ligadura $A=person(age:X)$ fue ejecutada. El segundo **declare X Y** cambia la asociación de x , pero no el registro $persona(edad:25)$ porque éste ya existe en el almacén. Este comportamiento de **declare** se diseñó así para soportar un estilo modular de programación. La ejecución de un fragmento de programa no ocasionará que los resultados de fragmentos ejecutados previamente cambien.

Hay una segunda forma de **declare**:

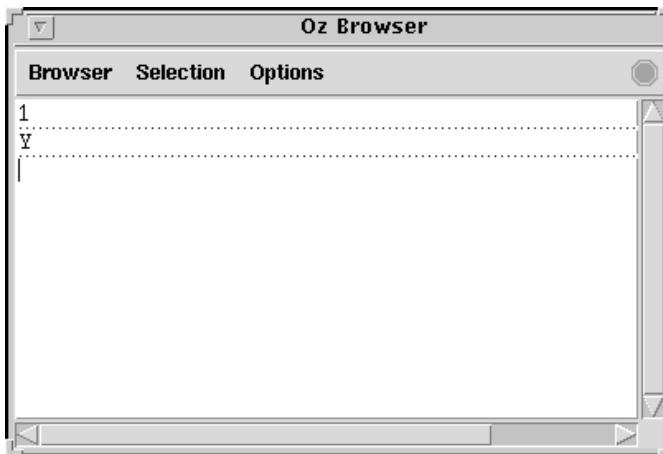


Figura 2.20: El Browser.

```
declare X Y in <decl>
```

en la cual se definen dos variables globales igual que antes, y luego se ejecuta `<decl>`. La diferencia con la primera forma es que `<decl>` no define ninguna variable global (a menos que contenga un `declare`).

El Browser

La interfaz interactiva tiene una herramienta, llamada el *Browser*, que permite observar el almacén. Esta herramienta está disponible para el programador como un procedimiento llamado `Browse`. El procedimiento `Browse` tiene un argumento, y se invoca escribiendo `{Browse <expr>}`, donde `<expr>` es cualquier expresión. El `Browser` puede mostrar valores parciales y los actualiza en la pantalla cada vez que son ligados un poco más. Alimentando la interfaz con:

```
{Browse 1}
```

muestra el entero 1. Alimentándola con:

```
declare Y in
{Browse Y}
```

muestra sólo el nombre de la variable, a saber, `Y`. No se despliega ningún valor. Esto significa que actualmente `Y` es no-ligada. En la figura 2.20 se muestra la ventana del `browser` después de estas dos operaciones. Si `Y` se liga, e.g., haciendo `Y=2`, entonces el `browser` actualizará su ventana para mostrar esta ligadura.

Ejecución por flujo de datos

Como vimos antes, las variables declarativas soportan ejecución por flujo de datos, i.e., una operación espera hasta que todos sus argumentos estén ligados antes de ejecutarse. Para programas secuenciales esto no es muy útil, pues el programa esperaría por siempre. Pero, por otro lado, para programas concurrentes, esto es muy útil, pues se pueden estar ejecutando varias secuencias de instrucciones al mismo tiempo. Una secuencia independiente de instrucciones en ejecución se llama un hilo. La programación con más de un hilo se llama programación concurrente y se introduce en el capítulo 4.

Cada fragmento de programa con que se alimenta la interfaz interactiva se ejecuta en su hilo propio. Esto nos permite presentar ejemplos sencillos de programación concurrente en este capítulo. Por ejemplo, alimente la interfaz con la declaración siguiente:

```
declare A B C in
C=A+B
{Browse C}
```

Esto no muestra nada, pues la instrucción `C=A+B` se bloquea (ambos argumentos son no-ligados). Ahora, alimente la interfaz con la declaración siguiente:

```
A=10
```

Esto liga `A`, pero la instrucción `C=A+B` permanece bloqueada pues `B` todavía es no ligado. Finalmente, alimente la interfaz con:

```
B=200
```

Inmediatamente se despliega 210 en el browser. Cualquier operación, no solamente la suma, se bloqueará si no tiene suficiente información para calcular su resultado. Por ejemplo, las comparaciones se pueden bloquear. La comparación de igualdad `x==y` se bloquea si no puede decidir si `x` es igual o diferente a `y`. Esto ocurre, e.g., si una de las variables o ambas son no-ligadas.

Los errores de programación frecuentemente terminan en suspensiones por flujo de datos. Si usted alimenta la interfaz con una declaración que debería mostrar algo y no se muestra nada, entonces lo más seguro es que hay una operación bloqueada. Verifique cuidadosamente que todas las operaciones tengan sus argumentos ligados. Idealmente, el depurador del sistema debería detectar si un programa tiene operaciones bloqueadas que no le dejan continuar la ejecución.

2.7. Excepciones

Primero déjennos encontrar la regla, luego trataremos de explicar las excepciones.
– *El nombre de la rosa*, Umberto Eco (1932–)

¿Cómo manejamos situaciones excepcionales dentro de un programa? ¿Por ejemplo,

El modelo de computación declarativa

una división por cero, tratar de abrir un archivo inexistente, o seleccionar un campo inexistente de un registro? Estas operaciones no ocurren en un programa correcto, luego ellas no deberían perturbar un estilo normal de programación. Por otro lado, estas situaciones ocurren algunas veces. Debería ser posible que los programas las manejaran en una forma sencilla. El modelo declarativo no puede hacer esto sin agregar numerosas verificaciones a lo largo del programa. Una manera más elegante de manejar estas situaciones consiste en extender el modelo con un mecanismo de manejo de excepciones. En esta sección se hace exáctamente eso. Presentamos la sintaxis y la semántica del modelo extendido y explicamos cómo se ven las excepciones en el lenguaje completo.

2.7.1. Motivación y conceptos básicos

En la semántica de la sección 2.4, hablamos de “lanzar un error” cuando una declaración no puede continuar ejecutándose correctamente. Por ejemplo, un condicional lanza un error cuando su argumento no es un valor booleano. Hasta ahora, hemos sido deliberadamente vagos en describir exáctamente qué pasa después de lanzar un error. Ahora seremos más precisos. Definimos un error como una diferencia entre el comportamiento real del programa y el comportamiento deseado. Hay muchas fuentes de error, tanto internas como externas al programa. Un error interno puede resultar de invocar una operación con un argumento de tipo ilegal o con un valor ilegal. Un error externo puede resultar de tratar de abrir un archivo inexistente.

Nos gustaría ser capaces de detectar errores y manejarlos dentro de un programa en ejecución. El programa no debería detenerse cuando esto pasa. Más bien, debería transferir la ejecución, en una forma controlada, a otra parte, llamada el manejador de excepciones, y pasarle a éste un valor que describa el error.

¿Cómo debería ser el mecanismo de manejo de excepciones? Podemos realizar dos observaciones. La primera, el mecanismo debería ser capaz de confinar el error, i.e., colocarlo en cuarentena de manera que no contamine todo el programa. Llamamos a esto el principio de confinamiento del error. Suponga que el programa está hecho de “componentes” que interactúan, organizados en una forma jerárquica. Cada componente se construye a partir de componentes más pequeños. Colocamos “componente” entre comillas porque el lenguaje no necesita tener el concepto de componente. Sólo necesita ser composicional, i.e., los programas se construyen por capas. Entonces, el principio del confinamiento del error afirma que un error en un componente debe poderse capturar en la frontera del componente. Por fuera del componente, el error debe ser invisible o ser reportado en una forma adecuada.

Por lo tanto, el mecanismo produce un “salto” desde el interior del componente hasta su frontera. La segunda observación es que este salto debe ser una sola operación. El mecanismo debe ser capaz, en una sola operación, de salirse de tantos niveles como sea necesario a partir del contexto de anidamiento. En la figura 2.21 se ilustra esto. En nuestra semántica, definimos un contexto como una entrada en la pila semántica, i.e., una instrucción que tiene que ser ejecutada más adelante.

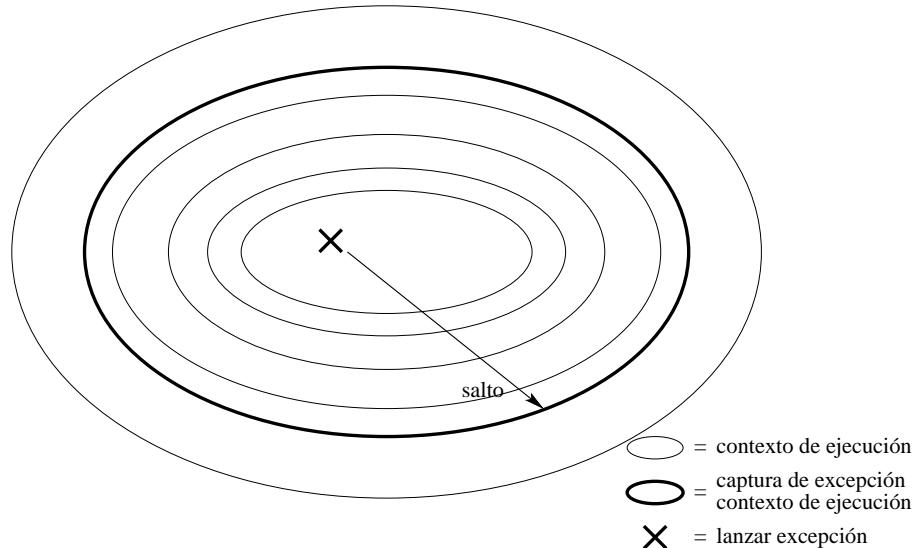


Figura 2.21: Manejo de excepciones.

Los contextos anidados se crean por invocaciones a procedimientos y composiciones secuenciales.

El modelo declarativo no puede dar ese salto en una sola operación. El salto debe ser codificado explícitamente en brincos pequeños, uno por contexto, utilizando variables booleanas y condicionales. Esto vuelve los programas más voluminosos, especialmente si debe añadirse código adicional en cada sitio donde puede ocurrir un error. Se puede mostrar teóricamente que la única forma de conservar los programas sencillos es extender el modelo [92, 94].

Proponemos una extensión sencilla del modelo que satisfaga estas condiciones. Agregamos dos declaraciones: la declaración **try** y la declaración **raise**. La declaración **try** crea un contexto para capturar excepciones junto con un manejador de excepciones. La declaración **raise** salta a la frontera del contexto para capturar la excepción más interna e invoca al manejador de excepciones allí. Declaraciones **try** anidadas crean contextos anidados. Ejecutar **try** $\langle d \rangle$ **catch** $\langle x \rangle$ **then** $\langle d \rangle_1$ **end** es equivalente a ejecutar $\langle d \rangle$, si $\langle d \rangle$ no lanza una excepción. Por otro lado, si $\langle d \rangle$ lanza una excepción, i.e., ejecutando una declaración **raise**, entonces se aborta la ejecución (aún en curso) de $\langle d \rangle$. Toda la información relacionada con $\langle d \rangle$ es sacada de la pila semántica. El control se transfiere a $\langle d \rangle_1$, pasándole una referencia a la excepción mencionada en $\langle x \rangle$.

Cualquier valor parcial puede ser una excepción. Esto significa que el mecanismo de manejo de excepciones puede ser extendido por el programador, i.e., se pueden definir excepciones nuevas a medida que sean necesitadas por el programa. Esto lleva al programador a prever situaciones excepcionales nuevas. Como una excepción

El modelo de computación declarativa

puede ser una variable no-ligada, lanzar una excepción y mirar cuál es se puede hacer de manera concurrente. ¡En otras palabras, una excepción puede ser lanzada (y capturada) antes de conocer de cuál excepción se trata! Esto es bastante razonable en un lenguaje con variables de flujo de datos: podemos saber en algún momento que existe un problema pero no saber todavía cuál es el problema.

Un ejemplo

A continuación presentamos un ejemplo de manejo de excepciones. Considere la función siguiente, la cual evalúa expresiones aritméticas simples y devuelve el resultado:

```
fun {Eval E}
    if {IsNumber E} then E
    else
        case E
        of  suma(X Y) then {Eval X}+{Eval Y}
            [] mult(X Y) then {Eval X}*{Eval Y}
            else raise exprMalFormada(E) end
        end
    end
end
```

Para este ejemplo, decimos que una expresión está mal formada si no es reconocida por `Eval`, i.e., contiene valores diferentes de números, `suma` y `mult`. Al tratar de evaluar una expresión mal formada se lanzará una excepción. La excepción es una tupla, `exprMalFormada(E)`, que contiene la expresión mal formada. A continuación un ejemplo de utilización de `Eval`:

```
try
    {Browse {Eval suma(suma(5 5) 10)}}
    {Browse {Eval mult(6 11)}}
    {Browse {Eval resta(7 10)}}
catch exprMalFormada(E) then
    {Browse "*** Expresión ilegal '#E#' ***"}
end
```

Si una invocación a `Eval` lanza una excepción, entonces el control se transfiere a la cláusula `catch`, la cual muestra un mensaje de error.

2.7.2. El modelo declarativo con excepciones

Extendemos el modelo de computación declarativa con excepciones. En la tabla 2.9 se presenta la sintaxis del lenguaje núcleo extendido. Los programas pueden utilizar dos declaraciones nuevas, `try` y `raise`. Además, existe una tercera declaración, `catch <x> then <d> end`, que se necesita internamente para definir la semántica pero no es permitida en los programas. La declaración `catch` es una “marca” sobre la pila semántica que define la frontera del contexto que captura la excepción. Ahora presentamos la semántica de estas declaraciones.

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Invocación de procedimiento
try $\langle d \rangle_1$ catch $\langle x \rangle$ then $\langle d \rangle_2$ end	Contexto de la excepción
raise $\langle x \rangle$ end	Lanzamiento de excepción

Tabla 2.9: El lenguaje núcleo declarativo con excepciones.

La declaración try

La declaración semántica es:

$(\text{try } \langle d \rangle_1 \text{ catch } \langle x \rangle \text{ then } \langle d \rangle_2 \text{ end}, E)$

La ejecución consiste de las siguientes acciones:

- Coloque la declaración semántica $(\text{catch } \langle x \rangle \text{ then } \langle d \rangle_2 \text{ end}, E)$ en la pila.
- Coloque $(\langle d \rangle_1, E)$ en la pila.

La declaración raise

La declaración semántica es:

$(\text{raise } \langle x \rangle \text{ end}, E)$

La ejecución consiste de las siguientes acciones:

- Saque elementos de la pila hasta encontrar una declaración **catch**.
 - Si se encontró una declaración **catch**, se saca de la pila.
 - Si la pila quedó vacía y no se encontró ninguna declaración **catch**, entonces se detiene la ejecución con el mensaje de error “excepción no capturada”²⁰.
- Sea $(\text{catch } \langle y \rangle \text{ then } \langle d \rangle \text{ end}, E_c)$ la declaración **catch** que se encontró.
- Coloque $(\langle d \rangle, E_c + \{\langle y \rangle \rightarrow E(\langle x \rangle)\})$ en la pila.

20. Nota del traductor: “Uncaught exception”, en inglés.

$\langle \text{declaración} \rangle ::= \text{try } \langle \text{declaraciónEn} \rangle$
$\quad [\text{catch } \langle \text{patrón} \rangle \text{ then } \langle \text{declaraciónEn} \rangle$
$\quad \quad \{ \cdot [] \cdot \langle \text{patrón} \rangle \text{ then } \langle \text{declaraciónEn} \rangle \}]$
$\quad [\text{finally } \langle \text{declaraciónEn} \rangle] \text{ end}$
$\quad \text{ raise } \langle \text{expresiónEn} \rangle \text{ end}$
$\quad \dots$
$\langle \text{declaraciónEn} \rangle ::= [\{ \langle \text{parteDeclaración} \rangle \} + \text{in }] \langle \text{declaración} \rangle$
$\langle \text{expresiónEn} \rangle ::= [\{ \langle \text{parteDeclaración} \rangle \} + \text{in }] [\langle \text{declaración} \rangle] \langle \text{expresión} \rangle$

Tabla 2.10: Sintaxis para el manejo de excepciones.

Miremos cómo se maneja una excepción no capturada en el sistema Mozart. Si la ejecución se realiza mediante la interfaz interactiva, entonces el sistema imprime un mensaje de error en la ventana del emulador de Oz. Por el contrario, si la ejecución se realiza por medio de una aplicación autónoma²¹, la aplicación termina y se envía un mensaje de error a la salida estándar del proceso. Es posible cambiar este comportamiento a algo diferente que sea más deseable para una aplicación particular; esto se hace utilizando el módulo `Property` del sistema.

La declaración catch

La declaración semántica es:

$(\text{catch } \langle x \rangle \text{ then } \langle d \rangle \text{ end}, E)$

Si esta es la declaración semántica encontrada en la pila semántica, es porque la declaración `try` que la colocó en la pila se ejecutó completamente sin lanzar esa excepción. Entonces, ejecutar la declaración `catch` no tiene acciones asociadas, igual que la declaración `skip`.

2.7.3. Sintaxis completa

En la tabla 2.10 se presenta la sintaxis de la declaración `try` en el lenguaje completo. Observe que cuenta con una cláusula `finally`, opcional. La cláusula `catch` también tiene una serie de patrones opcionales. Miremos cómo se definen estas extensiones:

21. Nota del traductor: *Standalone application*, en inglés.

La cláusula finally

Una declaración **try** puede especificar una cláusula **finally** que siempre se ejecutará, ya sea que la ejecución de la declaración lance una excepción o no. La nueva sintaxis

```
try <d>1 finally <d>2 end
```

se traduce al lenguaje núcleo así:

```
try <d>1  
catch X then  
  <d>2  
  raise X end  
end  
<d>2
```

(donde el identificador *x* se escoge de manera que no esté libre en *<d>2*). Es posible definir una traducción en la que *<d>2* sólo ocurra una vez; dejamos esto para los ejercicios.

La cláusula **finally** es útil cuando se trabaja con entidades externas al modelo de computación. Con **finally** podemos garantizar que se realice alguna acción de “limpieza” sobre la entidad, ocurra o no una excepción. Un ejemplo típico es la lectura de un archivo. Suponga que *F* representa un archivo abierto,²² que el procedimiento *ProcesarArchivo* manipula el archivo de alguna manera, y que el procedimiento *CerrarArchivo* cierra el archivo. El programa siguiente asegura que el archivo *F* quede siempre cerrado después que *ProcesarArchivo* termina, se haya lanzado o no una excepción:

```
try  
  {ProcesarArchivo F}  
finally {CerrarArchivo F} end
```

Note que esta declaración **try** no captura la excepción; sólo ejecuta *CerrarArchivo* una vez *ProcesarArchivo* termina. Podemos combinar la captura de excepciones con la ejecución final de una declaración:

```
try  
  {ProcesarArchivo F}  
catch X then  
  {Browse '*** Excepción '#X#' procesando el archivo ***}  
finally {CerrarArchivo F} end
```

Esto se comporta como dos cláusulas **try** anidadas: la más interna con una cláusula **catch** únicamente y la más externa con una cláusula **finally** únicamente.

22. Más adelante veremos cómo se maneja la entrada/salida con archivos.

Reconocimiento de patrones

Una declaración **try** puede usar reconocimiento de patrones para capturar únicamente excepciones consistentes con un patrón dado. Las otras excepciones se pasan a la siguiente declaración **try** que envuelve la actual. La sintaxis nueva:

```
try <d>
  catch <p>1 then <d>1
    [ ] <p>2 then <d>2
    ...
    [ ] <p>n then <d>n
  end
```

se traduce al lenguaje núcleo así:

```
try <d>
  catch X then
    case X
      of <p>1 then <d>1
      [ ] <p>2 then <d>2
      ...
      [ ] <p>n then <d>n
    else raise X end
  end
end
```

Si la excepción no es consistente con ningún patrón, entonces, simplemente, se lanza de nuevo.

2.7.4. Excepciones del sistema

Unas pocas excepciones son lanzadas por el mismo sistema Mozart. Estas se llaman excepciones del sistema. Todas ellas son registros con una de las siguientes tres etiquetas:**failure**, **error**, o **system**.

- **failure**: indica un intento de realizar una operación de ligadura inconsistente (e.g., $1=2$) en el almacén (ver sección 2.8.2.1). También se conoce como una falla en la unificación.
- **error**: indica un error en tiempo de ejecución dentro del programa, i.e., una situación que no debería ocurrir durante la operación normal. Estos errores son errores de tipo o de dominio. Un error de tipo ocurre cuando se invoca una operación con un argumento del tipo incorrecto. e.g., aplicar un valor que no es de tipo procedimiento a algún argumento (`{foo 1}`, donde `foo` es un átomo), o sumar un entero y un átomo (e.g., `X=1+a`). Un error de dominio ocurre cuando se invoca una operación con un argumento por fuera del alcance de su dominio (aún si es del tipo correcto), e.g., sacar la raíz cuadrada de un número negativo, dividir por cero, o seleccionar un campo inexistente de un registro.
- **system**: indica una condición que ocurre, en tiempo de ejecución, en el ambiente del proceso del sistema operativo correspondiente al sistema Mozart, e.g., una

situación imprevisible como un archivo o una ventana cerrados, o una falla al abrir una conexión entre dos procesos Mozart en programación distribuida (ver capítulo 11 (en CTM)).

Lo que se almacena dentro del registro de la excepción depende de la versión del sistema Mozart que se esté utilizando. Por la tanto los programadores deberían basarse sólo en la etiqueta. Por ejemplo,

```
fun {Uno} 1 end
fun {Dos} 2 end
try {Uno}={Dos}
catch
    failure(...) then {Browse fallaCapturada}
end
```

El patrón `failure(...)` captura cualquier registro cuya etiqueta sea `failure`.

2.8. Temas avanzados

En esta sección se presenta información adicional para un entendimiento más profundo del modelo declarativo, sus compromisos, y posibles variaciones.

2.8.1. Lenguajes de programación funcional

La programación funcional consiste en definir funciones sobre valores completos, donde las funciones son verdaderas funciones en el sentido matemático. Un lenguaje en el cual ésta es la única forma de calcular es llamado un lenguaje funcional puro. Examinemos de qué manera se relacionan el modelo declarativo y la programación funcional pura. Como lectura adicional sobre la historia, fundamentos formales, y motivaciones de la programación funcional, recomendamos el artículo de revisión panorámica de Hudak [74].

El cálculo λ

Los lenguajes funcionales puros están basados en un formalismo llamado el cálculo λ .

Existen muchas variantes del cálculo λ . Todas esas variantes tienen en común dos operaciones básicas, a saber, definición y evaluación de funciones. Por ejemplo, el valor de tipo función `fun {$ x} x*x end` es idéntico a la expresión λ , $\lambda x. x * x$. Esta expresión consta de dos partes: la x antes del punto, la cual es el argumento de la función, y la expresión $x * x$, la cual es el resultado de la función. La función `Append`, la cual concatena dos listas en una sola, puede ser definida como un valor de tipo función:

El modelo de computación declarativa

```
Append=fun {$_ Xs Ys}
  if {IsNil Xs} then Ys
  else {Cons {Car Xs} {Append {Cdr Xs} Ys}}
end
```

Esto es equivalente a la expresión λ siguiente:

$$\begin{aligned}append = \lambda xs, ys . & \text{ if } isNil(xs) \text{ then } ys \\& \text{ else } cons(car(xs), append(cdr(xs), ys))\end{aligned}$$

Esta definición de Append utiliza las siguientes funciones auxiliares:

```
fun {IsNil X} X==nil end
fun {IsCons X} case X of _|_ then true else false end end
fun {Car H|T} H end
fun {Cdr H|T} T end
fun {Cons H T} H|T end
```

Restringiendo el modelo declarativo

El modelo declarativo es más general que el cálculo λ en dos formas. La primera es que en el modelo declarativo se definen funciones sobre valores parciales, e.g., con variables no-ligadas. La segunda es que el modelo declarativo utiliza una sintaxis procedimental. Podemos definir un lenguaje funcional puro si colocamos dos restricciones sintácticas al modelo declarativo de manera que siempre calcule funciones sobre valores completos:

- Ligar siempre una variable a un valor al momento de la definición. Es decir, la declaración **local** siempre tiene una de las dos formas siguientes:

```
local <x>=<v> in <d> end
local <x>=<y> <y>1 ... <y>n in <d> end
```

- Utilizar siempre la sintaxis de función y no la de procedimientos. En la invocación de funciones dentro de estructuras de datos, realizar la invocación anidada antes de la creación de la estructura (en lugar de hacerlo después, como se muestra en la sección 2.6.2). Esto evita tener variables no-ligadas dentro de las estructuras de datos.

Con estas restricciones el modelo no necesita variables no-ligadas. El modelo declarativo con estas restricciones es llamado el modelo funcional (estricto). Este modelo es cercano a los lenguajes de programación funcional bien conocidos como Scheme y Standard ML. El rango completo de técnicas de programación de alto orden queda vigente. El reconocimiento de patrones también es posible gracias a la declaración **case**.

Variedades de programación funcional

Exploremos algunos variaciones sobre el tema de la programación funcional:

$\langle \text{declaración} \rangle ::= \langle \text{expresión} \rangle \stackrel{\sim}{=} \langle \text{expresión} \rangle \mid \dots$
$\langle \text{expresión} \rangle ::= \langle \text{expresión} \rangle \stackrel{==}{=} \langle \text{expresión} \rangle$
$\quad \mid \langle \text{expresión} \rangle \stackrel{\sim}{\backslash=} \langle \text{expresión} \rangle \mid \dots$

Tabla 2.11: Igualdad (unificación) y comprobación de igualdad.

- El modelo funcional de este capítulo es dinámicamente tipado como Scheme. Muchos lenguajes funcionales son estáticamente tipados. En la sección 2.8.3 se explican las diferencias entre los dos enfoques. Adicionalmente, muchos de los lenguajes estáticamente tipados, e.g., Haskell y Standard ML, realizan inferencia de tipos, lo cual permite al compilador inferir los tipos de todas las funciones.
- Gracias a las variables de flujo de datos y al almacén de asignación única, el modelo declarativo admite técnicas de programación que no se encuentran en la mayoría de los lenguajes funcionales, incluyendo Scheme, Standard ML, Haskell, y Erlang. Esto incluye ciertas formas de optimización de última invocación y técnicas para calcular con valores parciales como se muestra en el capítulo 3.
- El modelo concurrente declarativo del capítulo 4 agrega la concurrencia conservando todas las propiedades buenas de la programación funcional. Esto es posible gracias a las variables de flujo de datos y al almacén de asignación única.
- En el modelo declarativo, las funciones son ansiosas por defecto, i.e., los argumentos de la función se evalúan antes que se ejecute su cuerpo. Esto también se llama evaluación estricta. Los lenguajes funcionales Scheme y Standard ML son estrictos. Existe otro orden de ejecución útil, la evaluación perezosa, en la cual los argumentos de la función sólo se evalúan si su resultado se necesita. Haskell es un lenguaje funcional perezoso.²³ La evaluación perezosa es una técnica poderosa de control del flujo en programación funcional [76]. Permite programar con estructuras de datos potencialmente infinitas sin ninguna clase de límites explícitos. En la sección 4.5 se explica esto en detalle. Un programa declarativo ansioso puede evaluar funciones y nunca usar sus resultados, realizando así trabajo superfluo. Un programa declarativo perezoso, por su lado, realiza la mínima cantidad de trabajo requerida para conseguir su resultado.
- Muchos lenguajes funcionales soportan una técnica de programación de alto orden denominada currificación, la cual se explica en la sección 3.6.6.

23. Para ser estrictos, Haskell es un lenguaje no estricto. Esto es idéntico a la evaluación perezosa para propósitos prácticos. La diferencia se explica en la sección 4.9.2.

2.8.2. Unificación y comprobación de unificación

En la sección 2.2 vimos cómo ligar variables de flujo de datos a valores parciales y a otras variables, utilizando la operación de igualdad ($=$) como lo muestra la tabla 2.11. En la sección 2.3.5 vimos cómo comparar valores, utilizando las operaciones de prueba de igualdad ($==$ y $\backslash=$). Hasta ahora, sólo hemos visto los casos simples de estas operaciones. Examinemos unos casos más generales.

Ligar una variable a un valor es un caso especial de una operación más general llamada unificación. La unificación $\langle \text{Term1} \rangle = \langle \text{Term2} \rangle$ obliga a que los valores parciales $\langle \text{Term1} \rangle$ y $\langle \text{Term2} \rangle$ sean iguales, si es posible, agregando ligaduras en el almacén. Por ejemplo, la unificación $f(x,y) = f(1,2)$ realiza dos ligaduras: $x=1$ and $y=2$. Si es imposible lograr que los dos términos sean iguales, entonces se lanza una excepción. La unificación existe debido a que calculamos con valores parciales; si sólo existieran valores completos, entonces la unificación no tendría ningún sentido.

Comprobar si una variable es igual a un valor es un caso especial de la comprobación de unificación²⁴ y de no unificación. La comprobación de unificación $\langle \text{Term1} \rangle == \langle \text{Term2} \rangle$ (y su opuesto, la comprobación de no unificación $\langle \text{Term1} \rangle \backslash= \langle \text{Term2} \rangle$) es una función booleana de dos argumentos que se bloquea hasta saber si $\langle \text{Term1} \rangle$ y $\langle \text{Term2} \rangle$ son iguales o no.²⁵ Las comprobaciones de unificación o de no unificación no realizan ninguna ligadura.

2.8.2.1. Unificación (la operación =)

Una buena forma de conceptualizar la unificación es como una operación que agrega información al almacén de asignación única. El almacén es un conjunto de variables de flujo de datos, donde cada variable es no-ligada o ligada a otra entidad del almacén. La información del almacén es justamente el conjunto de todas las ligaduras. Realizar una nueva ligadura, e.g., $x=y$, agregará la información que x y y son iguales. Si x y y ya están ligadas cuando se realiza $x=y$, entonces se deben agregar otras ligaduras al almacén. Por ejemplo, si el almacén tiene $x=\text{foo}(A)$ y $y=\text{foo}(25)$, entonces hacer $x=y$ ligará A a 25. La unificación es un tipo de compilador que recibe información nueva y la “compila en el almacén,” teniendo en cuenta las ligaduras existentes allí. Para entender su funcionamiento, miremos algunas posibilidades.

- Los casos más simples son las ligaduras a valores, e.g., $x=\text{persona}(\text{nombre}:x1 \text{ edad}:x2)$, y las ligaduras variable-variable, e.g., $x=y$. Si x y y son no-ligadas, entonces cada una de estas operaciones agrega una ligadura al almacén.
- La unificación es simétrica. Por ejemplo, $\text{persona}(\text{nombre}:x1 \text{ edad}:x2) = x$ sig-

24. Nota del traductor: El término “comprobación de unificación” es la traducción escogida del término “entailment”, en inglés.

25. El término “comprobación de unificación” viene de la lógica. Es una forma de implicación lógica, pues la igualdad $\langle \text{Term1} \rangle == \langle \text{Term2} \rangle$ es cierta si el almacén, considerado como una conjunción de igualdades, “implica lógicamente” $\langle \text{Term1} \rangle == \langle \text{Term2} \rangle$.

nifica lo mismo que `x=persona(nombre:X1 edad:X2)`.

- Cualesquier dos valores parciales pueden ser unificados. Por ejemplo, la unificación de los dos registros:

```
persona(nombre:X1 edad:X2)
persona(nombre:"Jorge" edad:25)
```

liga `X1` a "Jorge" y `X2` to 25

- Si los valores parciales ya son iguales, entonces la unificación no hace nada. Por ejemplo, unificar `X` y `Y` cuando el almacén contiene los dos registros:

```
X=persona(nombre:"Jorge" edad:25)
Y=persona(nombre:"Jorge" edad:25)
```

no hace nada.

- Si los valores parciales son incompatibles, entonces no pueden ser unificados. Por ejemplo, la unificación de los dos registros:

```
persona(nombre:X1 edad:26)
persona(nombre:"Jorge" edad:25)
```

Los registros tienen valores diferentes en su campo `edad`, a saber, 25 y 26, de manera que no pueden ser unificados. Esta unificación lanzará una excepción, la cual puede ser capturada por una declaración `try`. La unificación podría ligar o no `X1` con "Jorge"; eso depende exactamente del momento en que encuentre la incompatibilidad. Otra forma de conseguir una falla en la unificación es ejecutando la declaración `fail`.

- La unificación es simétrica en los argumentos. Por ejemplo, unificar los dos registros:

```
persona(nombre:"Jorge" edad:X2)
persona(nombre:X1 edad:25)
```

liga `X1` a "Jorge" y `X2` to 25, igual que antes.

- La unificación puede crear estructuras cíclicas, i.e., estructuras que tienen referencias a sí mismas. Por ejemplo, la unificación `x=persona(abuelo:X)`, crea un registro cuyo campo `abuelo` se referencia a sí mismo. Esta situación ocurre en algunas historias locas de viajes a través del tiempo.

- La unificación puede ligar estructuras cíclicas. Por ejemplo, creamos dos estructuras cíclicas, en `X` y `Y`, haciendo `X=f(a:X b:_)` y `Y=f(a:_ b:Y)`. Ahora, al realizar la unificación `X=Y` se crea una estructura con dos ciclos, la cual podemos escribir como `X=f(a:X b:X)`. Este ejemplo se ilustra en la figura 2.22.

2.8.2.2. El algoritmo de unificación

Vamos a presentar una definición precisa de la unificación. Definiremos la operación `unificar(x,y)` que unifica dos valores parciales `x` y `y` en el almacén σ . La unificación es una operación básica de la programación lógica. Cuando se usan en el contexto de la unificación, las variables del almacén se suelen llamar variables lógicas. La programación lógica, también llamada programación relacional, se dis-

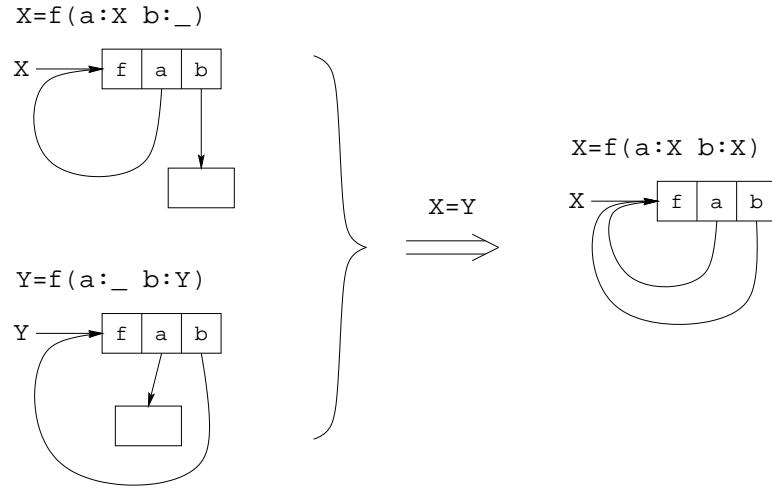


Figura 2.22: Unificación de estructuras cíclicas.

cute en el capítulo 9 (en CTM).

El almacén El almacén consta de un conjunto de k variables, x_1, \dots, x_k , formando una partición como sigue:

- Los conjuntos de variables no-ligadas que son iguales (también llamados conjuntos de variables equivalentes). Las variables en cada conjunto son iguales entre ellas pero no son iguales a ninguna otra que no esté allí.
- Las variables ligadas a un número, registro, o procedimiento (también llamadas variables determinadas).

Un ejemplo es el almacén $\{x_1 = \text{foo}(a:x_2), x_2 = 25, x_3 = x_4 = x_5, x_6, x_7 = x_8\}$ con ocho variables. Este almacén tiene tres conjuntos de variables equivalentes, a saber, $\{x_3, x_4, x_5\}$, $\{x_6\}$, y $\{x_7, x_8\}$; y tiene dos variables determinadas, a saber, x_1 y x_2 .

La operación primitiva de ligadura Definimos la unificación en términos de una operación primitiva de ligadura en el almacén σ . La operación liga todas las variables en un conjunto de variables equivalentes:

- $\text{ligar}(ES, \langle v \rangle)$ liga todas las variables en el conjunto de variables equivalentes ES a el número o registro $\langle v \rangle$. Por ejemplo, la operación $\text{ligar}(\{x_7, x_8\}, \text{foo}(a:x_2))$ modifica el almacén de manera que x_7 y x_8 dejan de pertenecer a un conjunto de variables equivalentes y pasan a ser variables ligadas a $\text{foo}(a:x_2)$.
- $\text{ligar}(ES_1, ES_2)$ mezcla el conjunto de variables equivalentes ES_1 con el conjunto de variables equivalentes ES_2 . Por ejemplo, la operación $\text{ligar}(\{x_3, x_4, x_5\}, \{x_6\})$

modifica el almacén del ejemplo de manera que x_3, x_4, x_5 , y x_6 están en un único conjunto de variables equivalentes, a saber, $\{x_3, x_4, x_5, x_6\}$.

El algoritmo Ahora definimos la operación $\text{unificar}(x, y)$ como sigue:

1. Si x está en el conjunto de variables equivalentes ES_x y y está en el conjunto de variables equivalentes ES_y , entonces se hace ligar(ES_x, ES_y). Si x y y están en el mismo conjunto de variables equivalentes, esto es lo mismo que no hacer nada.
2. Si x está en el conjunto de variables equivalentes ES_x y y está determinada, entonces se hace ligar(ES_x, y).
3. Si y está en el conjunto de variables equivalentes ES_y y x está determinada, entonces se hace ligar(ES_y, x).
4. Si x está ligada a $l(l_1 : x_1, \dots, l_n : x_n)$ y y está ligada a $l'(l'_1 : y_1, \dots, l'_m : y_m)$ con $l \neq l'$ o $\{l_1, \dots, l_n\} \neq \{l'_1, \dots, l'_m\}$, entonces se lanza una excepción de falla de la unificación.
5. Si x está ligada a $l(l_1 : x_1, \dots, l_n : x_n)$ y y está ligada a $l(l_1 : y_1, \dots, l_n : y_n)$, entonces para i de 1 a n se hace $\text{unificar}(x_i, y_i)$.

Manejando los ciclos El algoritmo anterior no maneja la unificación de valores parciales con ciclos. Por ejemplo, suponga que el almacén contiene $x = f(a:x)$ y $y = f(a:y)$. Realizar $\text{unificar}(x, y)$ llevaría a realizar recursivamente $\text{unificar}(x, y)$ que es idéntica a la operación original. ¡El algoritmo se queda en un ciclo infinito! Sin embargo es claro que x y y tienen exactamente la misma estructura: lo que la unificación *debería* hacer es agregar exactamente cero ligaduras al almacén y luego terminar. ¿Cómo podemos resolver este problema?

Una solución sencilla es asegurarse que la operación $\text{unificar}(x, y)$ se realiza a lo sumo una vez por cada posible pareja de variables (x, y) . Como cualquier intento de realizarla otra vez no produce nada nuevo, entonces puede terminar inmediatamente. Si hay k variables en el almacén, esto significa que a lo sumo la operación de unificación se realizará k^2 veces, de manera que está garantizado que el algoritmo termina siempre. En la práctica, el número de unificaciones que se realizan es mucho menor que esto. Podemos implementar la solución con una tabla que almacene todas las parejas para las que se realizó la operación. Esto nos lleva a un nuevo algoritmo $\text{unificar}'(x, y)$:

- Sea M una tabla nueva, vacía.
- Realice $\text{unificar}'(x, y)$.

Ahora la definición de $\text{unificar}'(x, y)$:

- Si $(x, y) \in M$, entonces no hay nada que hacer.
- De lo contrario, inserte (x, y) en M y realice el algoritmo original para $\text{unificar}(x, y)$, en el cual las invocaciones recursivas de la operación de unificación se reemplazan por invocaciones a unify' .

El modelo de computación declarativa

Este algoritmo se puede escribir en el modelo declarativo pasando M como dos argumentos adicionales a `unify'`. Una tabla que recuerda las invocaciones previas de manera que se pueda evitar repetirlas en un futuro se denomina tabla de memorización.

2.8.2.3. Mostrando estructuras cíclicas

Hemos visto que la unificación puede crear estructuras cíclicas. Para mostrarlas en el browser, éste tiene que ser configurado correctamente. En el menú `Options` del browser, escoja la entrada `Representation` y escoja el modo `Graph`. Existen tres modos para mostrar las estructuras, a saber `Tree` (por defecto), `Graph`, y `Minimal Graph`. `Tree` no tiene en cuenta los ciclos ni si se comparten estructuras. `Graph` si maneja esto correctamente, mostrando un grafo. `Minimal Graph` muestra el grafo más pequeño que es consistente con los datos. Daremos algunos ejemplos. considere las dos unificaciones siguientes:

```
local X Y Z in
  f(X b)=f(a Y)
  f(Z a)=Z
  {Browse [X Y Z]}
end
```

En el browser se muestra la lista `[a b R14=f(R14 a)]`, si el browser se ha configurado para mostrar la representación `Graph`. El término `R14=f(R14a)` es la representación textual de un grafo cíclico. El nombre de variable `R14` lo introduce el browser; versiones diferentes de Mozart pueden producir nombres de variables diferentes. Como segundo ejemplo, aliente la interfaz con la unificación siguiente, teniendo el browser configurado en la opción `Graph`, como antes:

```
declare X Y Z in
  a(X c(Z) Z)=a(b(Y) Y d(X))
  {Browse X#Y#Z}
```

Ahora configure el browser en el modo `Minimal Graph` y repita la unificación. ¿Cómo puede usted explicar la diferencia?

2.8.2.4. Comprobaciones de unificación y de no unificación (las operaciones == y \=)

La comprobación de unificación `x==y` es una función booleana que verifica si `x` y `y` son iguales o no. La comprobación contraria, `x\=y`, se llama la comprobación de no unificación. Ambas comprobaciones usan esencialmente el mismo algoritmo.²⁶ La comprobación de unificación devuelve `true` si la información que hay en el almacén implica la información `x=y` en una forma verificable (del almacén “se deduce” `x=y`),

26. Estrictamente hablando, hay un único algoritmo que realiza las dos comprobaciones simultáneamente. Este devuelve `true` o `false` dependiendo de cuál es la comprobación deseada.

y devuelve **false** si de la información del almacén implica que nunca será cierto $x=Y$, de nuevo, en una forma verificable (del almacén “no se deducirá” $x=Y$). La comprobación se bloquea si no puede determinar que x y Y son iguales o que nunca lo podrán ser. La comprobación se define como sigue:

- Devuelve el valor **true** si los grafos que comienzan en los nodos de x y Y tienen la misma estructura, i.e., todos los nodos correspondientes, dos a dos, tienen valores idénticos o son el mismo nodo. Llamamos a esto igualdad estructural.
- Devuelve el valor **false** si los grafos tienen estructura diferente, o algún par de nodos correspondientes tienen valores diferentes.
- Se bloquea cuando analiza un par de nodos correspondientes que son diferentes, pero al menos uno de ellos es no ligado.

Presentamos un ejemplo:

```
declare L1 L2 L3 Cabeza Cola in
L1=Cabeza|Cola
Cabeza=1
Cola=2|nil

L2=[1 2]
{Browse L1==L2}

L3='|'(1:1 2:'|'(2 nil))
{Browse L1==L3}
```

Las tres listas, $L1$, $L2$, y $L3$, son idénticas. Presentamos ahora un ejemplo donde la comprobación de unificación no se puede decidir:

```
declare L1 L2 X in
L1=[1]
L2=[X]
{Browse L1==L2}
```

Al alimentar la interfaz con este ejemplo no se muestra nada en el browser, pues la comprobación de unificación no puede decidir si $L1$ y $L2$ son iguales o no. De hecho, ambas son posibles: si X se liga a 1, entonces las listas son iguales, pero si X se liga a 2, entonces son diferentes. Alimente la interfaz con $X=1$ o $X=2$ para ver qué pasa. Y qué se puede decir sobre el ejemplo siguiente?

```
declare L1 L2 X in
L1=[X]
L2=[X]
{Browse L1==L2}
```

Ambas listas contienen la misma variable no-ligada X . ¿Qué pasará? Piense un poco al respecto antes de leer la respuesta en la nota de pie de página.²⁷ Presentamos un último ejemplo:

27. El browser mostrará **true**, pues $L1$ y $L2$ son iguales sin importar a qué valor pueda ser ligado X .

```
declare L1 L2 X in
L1=[1 a]
L2=[X b]
{Browse L1==L2}
```

El browser mostrará **false**. Mientras que la comparación `1==X` se bloquea, una inspección posterior de los dos grafos muestra que existe una diferencia definitiva, de manera que la comprobación completa debe devolver **false**.

2.8.3. Tipamiento dinámico y estático

La única manera de descubrir los límites de lo posible es aventurarse más allá de ellos, a lo imposible.

– Segunda ley de Clarke, Arthur C. Clarke (1917–)

Es importante que un lenguaje sea fuertemente tipado, i.e., que tenga un sistema de tipos que el lenguaje haga cumplir. (En contraste con un lenguaje débilmente tipado, en el cual la representación interna de un tipo puede ser manipulada por un programa. No hablaremos más de lenguajes débilmente tipados). Existen dos grandes familias de tipamiento fuerte: tipamiento dinámico y tipamiento estático. Hemos introducido el modelo declarativo como un modelo dinámicamente tipado, pero hasta ahora no hemos explicado la motivación de esa decisión de diseño, ni las diferencias entre tipamiento estático y dinámico que lo fundamentan.

- En un lenguaje dinámicamente tipado, las variables pueden ser ligadas a entidades de cualquier tipo; por tanto, en general, su tipo sólo se conoce en tiempo de ejecución.
- Por su lado, en un lenguaje estáticamente tipado, todos los tipos de las variables se conocen en tiempo de compilación. El tipo puede ser definido por el programador o inferido por el compilador.

Cuando se diseña un lenguaje, una de las decisiones más importantes a tomar es si el lenguaje será dinámicamente tipado, estáticamente tipado, o alguna mezcla de ambos. ¿Cuáles son las ventajas y las desventajas de los tipamientos dinámico y estático? El principio básico es que el tipamiento estático coloca restricciones sobre qué programas se pueden escribir, reduciendo la expresividad del lenguaje pero ofreciendo ventajas tales como una capacidad mejorada de captura de errores, eficiencia, seguridad, y verificación parcial de programas. Examinemos esto un poco más de cerca:

- El tipamiento dinámico no coloca restricciones sobre qué programas se pueden escribir. Para ser precisos, todo programa sintácticamente legal se puede ejecutar. Algunos de estos programas lanzarán excepciones, muy posiblemente debido a errores de tipo, las cuales pueden ser capturadas por un manejador de excepciones. El tipamiento dinámico ofrece la más amplia variedad posible de técnicas de programación. El programador invierte menos tiempo ajustando el programa para respetar el sistema de tipos.

- Con tipamiento dinámico se vuelve trivial realizar compilación separada, i.e., los módulos pueden ser compilados sin conocer nada sobre los otros módulos. Esto permite una verdadera programación abierta, en la cual módulos escritos independientemente pueden colocarse juntos en tiempo de ejecución e interactuar entre ellos. El tipamiento dinámico también hace que el desarrollo de programas sea escalable, i.e., los programas extremadamente largos se pueden dividir en módulos que son compilados individualmente sin recompilar otros módulos. Esto es más difícil de hacer con tipamiento dinámico pues la disciplina de tipos se debe cumplir a lo largo de las fronteras de los módulos.
- El tipamiento dinámico acorta el tiempo invertido entre una idea y su implementación, y permite un ambiente de desarrollo incremental como parte del sistema en tiempo de ejecución. También permite la comprobación de programas o fragmentos de programas aún cuando estos estén en un estado incompleto o inconsistente.
- El tipamiento estático permite capturar más errores de un programa en tiempo de compilación. Las declaraciones estáticas de tipos son una especificación parcial del programa, i.e., especifican parte del comportamiento del programa. El comprobador de tipos del compilador verifica que el programa satisfaga esta especificación parcial. Esto puede ser bastante poderoso. Los sistemas estáticos de tipos modernos pueden capturar una sorprendente cantidad de errores semánticos.
- El tipamiento estático permite una implementación más eficiente. Como el compilador tiene más información sobre qué valores puede llegar a contener una variable, entonces puede escoger una representación más eficiente. Por ejemplo, si una variable es de tipo booleano, el compilador puede implementarla con un solo bit. En un lenguaje dinámicamente tipado, el compilador no puede deducir siempre el tipo de una variable. Cuando no puede hacerlo, normalmente debe destinar para ella una palabra completa de memoria, de manera que cualquier valor posible (o un puntero a un valor) pueda ser acomodado allí.
- El tipamiento dinámico puede mejorar la seguridad de un programa. Las abstracciones de datos pueden ser construidas de manera segura basados solamente en la protección ofrecida por el sistema de tipos.

Desafortunadamente, la escogencia entre tipamiento dinámico y estático se basa más frecuentemente en reacciones emocionales (“intestino”), que en un argumento racional. Los defensores del tipamiento dinámico se deleitan con la libertad expresiva y la rapidez de los desarrollos que éste les ofrece y critican la reducida expresividad que ofrece el tipamiento estático. Por su parte, los defensores del tipamiento estático enfatizan en la ayuda que éste les ofrece para escribir programas correctos y eficientes y resaltan que se encuentran muchos errores de programa en tiempo de compilación. Muy pocos datos importantes existen para cuantificar estas diferencias. En nuestra experiencia, las diferencias no son mayores. Programar con tipamiento estático es como utilizar un procesador de texto con corrector ortográfico: un buen escritor puede hacerlo bien sin él, pero con él puede mejorar la calidad del texto.

El modelo de computación declarativa

Cada enfoque tiene su papel en el desarrollo de aplicaciones prácticas. El tipamiento estático se recomienda cuando las técnicas de programación están bien entendidas y cuando la eficiencia y la corrección son de suma importancia. El tipamiento dinámico se recomienda para desarrollos rápidos y cuando los programas deben ser tan flexibles como sea posible, tal como prototipos de aplicaciones, sistemas operativos, y algunas aplicaciones de inteligencia artificial.

La escogencia entre tipamiento estático y dinámico no tiene que ser del tipo “todo o nada”. En cada enfoque, se puede agregar un poco del otro, ganando así algunas de sus ventajas. Por ejemplo, agregar diferentes clases de polimorfismo (donde una variable puede contener valores de diversos tipos diferentes) y flexibilidad a lenguajes funcionales y orientados a objetos estáticamente tipados. El diseño de sistemas de tipos estáticos que capturen tanto como sea posible la flexibilidad de los sistemas dinámicos de tipos, mientras se preservan y fomentan buenos estilos de programación y la verificación en tiempo de compilación, es un área activa de investigación.

Todos los modelos de computación presentados en el libro son subconjuntos del lenguaje Oz, el cual es dinámicamente tipado. Un objetivo de investigación del proyecto Oz es explorar qué técnicas de programación son posibles en un modelo de computación que integre diversos paradigmas de programación. La única forma de lograr este objetivo es con tipamiento dinámico.

Una vez las técnicas de programación sean conocidas, una posible etapa siguiente es el diseño de un sistema estático de tipos. Mientras la investigación para incrementar la funcionalidad y la expresividad de Oz está aún en curso en el *Mozart Consortium*, el proyecto Alice en la Universidad de Saarland en Saarbrücken, Alemania, ha escogido agregar un sistema estático de tipos. Alice es un lenguaje estáticamente tipado que conserva mucho de la expresividad de Oz. Al momento de escribir este libro, Alice interopera con Oz (se pueden escribir programas parcialmente en Alice y en Oz) pues está basado en la implementación de Mozart.

2.9. Ejercicios

1. *Identificadores libres y ligados.* Considere la declaración siguiente:

```
proc {P X}
    if X>0 then {P X-1} end
end
```

¿La segunda ocurrencia del identificador `P` es libre o ligada? Justifique su respuesta.
Sugerencia: Esto es fácil de responder si realiza primero una traducción a la sintaxis del lenguaje núcleo.

2. *Ambiente contextual.* En la sección 2.4 se explica cómo se ejecuta una invocación a un procedimiento. Considere el procedimiento `MulByN` definido a continuación:

```
declare MulByN N in
N=3
proc {MulByN X ?Y}
    Y=N*X
end
```

junto con la invocación `{MulByN A B}`. Suponga que el ambiente al momento de la invocación contiene $\{A \rightarrow 10, B \rightarrow x_1\}$. Cuando el cuerpo del procedimiento se ejecuta, la asociación $N \rightarrow 3$ se agrega al ambiente. ¿Por qué es necesaria esta etapa? ¿En particular, la asociación $N \rightarrow 3$ no existe ya, en alguna parte en el ambiente, al momento de la invocación? ¿No sería esto suficiente para asegurar que el identificador `N` ya está asociado con 3? Dé un ejemplo donde `N` no exista en el ambiente al momento de la invocación. Luego, dé un ejemplo donde `N` exista allí, pero ligada a un valor diferente de 3.

3. *Funciones y procedimientos.* Si el cuerpo de una función tiene una declaración `if` sin la parte `else`, entonces se lanza una excepción si la condición del `if` es falsa. Explique por qué es correcto este comportamiento. Esta situación no ocurre así para los procedimientos. Explique por qué no.

4. *Las declaraciones `if` y `case`.* Este ejercicio explora la relación entre la declaración `if` y la declaración `case`.

- a) Defina la declaración `if` en términos de la declaración `case`. Esto muestra que el condicional no añade nada de expresividad por encima del reconocimiento de patrones, y podría haber sido agregado como una abstracción lingüística.
- b) Defina la declaración `case` en términos de la declaración `if`, utilizando las operaciones `Label`, `Arity`, y `^.^` (selección de campo).

Esto muestra que la declaración `if` es esencialmente una versión más primitiva de la declaración `case`.

5. *La declaración `case`.* Este ejercicio comprueba su entendimiento de la declaración `case` en sintaxis completa. Considere el procedimiento siguiente:

El modelo de computación declarativa

```
proc {Test X}
  case X
    of a|z then {Browse `case'(1)}
    [] f(a) then {Browse `case'(2)}
    [] Y|Z andthen Y==Z then {Browse `case'(3)}
    [] Y|Z then {Browse `case'(4)}
    [] f(Y) then {Browse `case'(5)}
    else {Browse `case'(6)} end
  end
```

Sin ejecutar ningún código, prediga qué pasará cuando alimente la interfaz con: {Test [b c a]}, {Test f(b(3))}, {Test f(a)}, {Test f(a(3))}, {Test f(d)}, {Test [a b c]}, {Test [c a b]}, {Test a|a}, y {Test `|`'(a b c)}. Si es necesario, utilice la traducción al lenguaje núcleo y la semántica para hacer las predicciones. Después de hacer las predicciones, compruebe su entendimiento ejecutando los ejemplos en Mozart.

6. *De nuevo la declaración case.* Dado el procedimiento siguiente:

```
proc {Test X}
  case X of f(a Y c) then {Browse `case'(1)}
  else {Browse `case'(2)} end
end
```

Sin ejecutar ningún código, prediga qué pasará cuando alimente la interfaz con:

```
declare X Y {Test f(X b Y)}
```

Lo mismo para:

```
declare X Y {Test f(a Y d)}
```

Lo mismo para:

```
declare X Y {Test f(X Y d)}
```

Si es necesario, utilice la traducción al lenguaje núcleo y la semántica para hacer las predicciones. Después de hacer las predicciones, compruebe su entendimiento ejecutando los ejemplos en Mozart. Ahora ejecute el ejemplo siguiente:

```
declare X Y
if f(X Y d)==f(a Y c) then {Browse `case'(1)}
else {Browse `case'(2)} end
```

¿El resultado es el mismo o es diferente al del ejemplo anterior? Explique el resultado.

7. *Clausuras con alcance léxico.* Considere el código siguiente:

```
declare Max3 Max5
proc {MaxEspecial Value ?SMax}
  fun {SMax X}
    if X>Value then X else Value end
  end
end
{MaxEspecial 3 Max3}
{MaxEspecial 5 Max5}
```

Sin ejecutar ningún código, prediga qué pasará cuando alimente la interfaz con:

```
{Browse [{Max3 4} {Max5 4}]}
```

Compruebe su entendimiento ejecutando el ejemplo en Mozart.

8. *Abstracción de control.* Este ejercicio explora la relación entre abstracciones lingüísticas y programación de alto orden.

a) Defina la función `AndThen` como sigue:

```
fun {AndThen BP1 BP2}
    if {BP1} then {BP2} else false end
end
```

¿Realizar la invocación

```
{AndThen fun {$} <expresión>1 end fun {$} <expresión>2 end}
```

produce el mismo resultado que `<expresión>1 andthen <expresión>2`? ¿Se evita la evaluación de `<expresión>2` en las mismas situaciones?

b) Escriba una función `OrElse` que sea a `orelse` como `AndThen` es a `andthen`. Explique su comportamiento.

9. *Recursión de cola.* Este ejercicio examina la importancia de la recursión de cola, a la luz de la semántica presentada en el capítulo. Considere las dos funciones siguientes:

```
fun {Sum1 N}
    if N==0 then 0 else N+{Sum1 N-1} end
end

fun {Sum2 N S}
    if N==0 then S else {Sum2 N-1 N+S} end
end
```

Ahora haga lo siguiente:

a) Expanda las dos definiciones en la sintaxis del lenguaje núcleo. Debería ser claro que `Sum2` es recursivo por la cola mientras que `Sum1` no lo es.

b) Ejecute las dos invocaciones `{Sum1 10}` y `{Sum2 10 0}` a mano, utilizando la semántica de este capítulo, poniendo especial atención en cómo evolucionan la pila y el almacén. ¿De qué tamaño es la pila en cada caso?

c) Qué pasaría en el sistema Mozart si se realiza la invocación `{Sum1 100000000}` o `{Sum2 100000000 0}`? ¿Cuál debería funcionar? ¿Cuál no? Ensaye ambas en Mozart y verifique su razonamiento.

10. *Expansión en la sintaxis del lenguaje núcleo.* Considere la función `sMerge` que mezcla dos listas ordenadas:

El modelo de computación declarativa

```
fun {MezclaOrd Xs Ys}
  case Xs#Ys
    of nil#Ys then Ys
    [] Xs#nil then Xs
    [] (X|Xr)#(Y|Yr) then
      if X=<Y then X|{MezclaOrd Xr Ys}
      else Y|{MezclaOrd Xs Yr} end
    end
  end
```

Expanda `MezclaOrd` en la sintaxis del lenguaje núcleo. Note que `X#Y` es una tupla de dos argumentos que también se puede escribir `'#'(X Y)`. El procedimiento resultante debería ser recursivo por la cola, si sigue correctamente las reglas de la sección 2.6.2.

11. *Recursión mutua*. La optimización de última invocación es importante para mucho más que sólo las invocaciones recursivas. Considere la definición siguiente de las funciones mutuamente recursivas `EsImpar` y `EsPar`:

```
fun {EsPar X}
  if X==0 then true else {EsImpar X-1} end
end

fun {EsImpar X}
  if X==0 then false else {EsPar X-1} end
end
```

Decimos que estas funciones son mutuamente recursivas porque cada función invoca a la otra. La recursión mutua se puede generalizar a cualquier número de funciones. Un conjunto de funciones es mutuamente recursivo si pueden colocarse en una secuencia de tal manera que cada función invoca a la siguiente y la última invoca a la primera. En este ejercicio, muestre que las invocaciones `{EsImpar N}` y `{EsPar N}` se ejecutan con el tamaño de la pila limitado por una constante, para todo entero no negativo `N`. En general, si cada función, en un conjunto de funciones mutuamente recursivo, tiene sólo una invocación en su cuerpo, y ésta es la última invocación, entonces todas las funciones en el conjunto se ejecutarán con el tamaño de la pila limitado por una constante.

12. *Excepciones con la cláusula finally*. En la sección 2.7 se muestra cómo definir la declaración `try/finally` traduciéndola en una declaración `try/catch`. En este ejercicio, defina otra traducción de

```
try <d>1 finally <d>2 end
```

en la cual `<d>1` y `<d>2` sólo ocurran una vez. *Sugerencia:* Se necesita una variable booleana.

13. *Unificación*. En la sección 2.8.2 se explica que la operación de ligadura es realmente mucho más general que sólo ligar variables: ella hace que dos valores parciales sean iguales (si son compatibles). Esta operación es llamada unificación. El propósito de este ejercicio es explorar por qué la unificación es interesante. Considere las tres unificaciones `x=[a z]`, `y=[w b]`, y `x=y`. Muestre que las variables `x`, `y`, `z`, y `w`

se ligan a los mismos valores sin importar en qué orden se realicen las unificaciones. En el capítulo 4 veremos que esta independencia del orden es importante para la concurrencia declarativa.

3

Técnicas de programación declarativa

S'il vous plaît... dessine-moi un arbre!

Por favor... dibújame un árbol!

– Adaptación libre de *Le Petit Prince*, Antoine de Saint-Exupéry (1900–1944)

Lo agradable de la programación declarativa es que uno escribe una especificación y la ejecuta como un programa. Lo desagradable de la programación declarativa es que algunas especificaciones claras resultan en programas increíblemente malos. La esperanza de la programación declarativa es que se pueda pasar de una especificación a un programa razonable sin tener que dejar el lenguaje.

– Adaptación libre de *The Craft of Prolog*, Richard O'Keefe (1990)

Considere cualquier operación computacional, i.e., un fragmento de programa con entradas y salidas. Decimos que la operación es declarativa si, cada vez que se invoca con los mismos argumentos, devuelve los mismos resultados independientemente de cualquier otro estado de computación. En la figura 3.1 se ilustra el concepto. Una operación declarativa es independiente (no depende de ningún estado de la ejecución por fuera de sí misma), sin estado (no tiene estados de ejecución internos que sean recordados entre invocaciones), y determinística (siempre produce los mismos resultados para los mismos argumentos). Mostraremos que todos los programas escritos utilizando el modelo de computación del último capítulo son declarativos.

Por Qué es importante la programación declarativa?

La programación declarativa es importante debido a dos propiedades:

- *Los programas declarativos son compositivos.* Un programa declarativo consiste de componentes que pueden ser escritos, comprobados, y probados correctos independientemente de otros componentes y de su propia historia pasada (invocaciones previas).
- *Razonar sobre programas declarativos es sencillo.* Es más fácil razonar sobre programas escritos en el modelo declarativo que sobre programas escritos en modelos más expresivos. Como los programas declarativos sólo pueden calcular valores, se pueden usar técnicas sencillas de razonamiento algebráico y lógico.

Estas dos propiedades son importantes tanto para programar en grande como en pequeño. Sería muy agradable si todos los programas pudieran escribirse en el

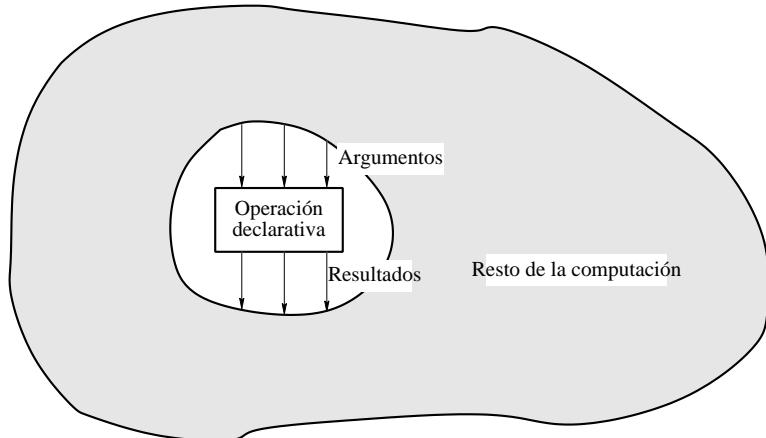


Figura 3.1: Una operación declarativa dentro de una computación general.

modelo declarativo. Desafortunadamente, este no es el caso. El modelo declarativo encaja bien con ciertas clases de programas y mal con otras. En este capítulo y en el siguiente se examinan las técnicas de programación del modelo declarativo y se explica qué clase de programas pueden ser fácilmente escritos en él y cuáles no.

Empezamos mirando más de cerca la primera propiedad. Definimos un *componente* como un fragmento de programa, delimitado precisamente, con entradas y salidas bien definidas. Un componente se puede definir en términos de un conjunto de componentes más simples. Por ejemplo, en el modelo declarativo un procedimiento es una especie de componente. El componente que ejecuta programas es el más alto en la jerarquía. En la parte más baja se encuentran los componentes primitivos los cuales son proveidos por el sistema.

En un programa declarativo, la interacción entre componentes se determina únicamente por las entradas y salidas de cada componente. Considere un programa con un componente declarativo. Este componente puede ser mirado en sí mismo, sin tener que entender el resto del programa. El esfuerzo necesario para entender el programa completo es la suma de los esfuerzos necesitados para entender el componente declarativo y para entender el resto.

Si existiera una interacción más estrecha entre el componente y el resto del programa, entonces ellos no se podrían entender independientemente. Tendrían que entenderse juntos, y el esfuerzo necesario sería mucho más grande. Por ejemplo, podría ser (aproximadamente) proporcional al producto de los esfuerzos necesitados para entender cada parte. En un programa con muchos componentes que interactúan estrechamente, esto explota muy rápido, dificultando o haciendo imposible entenderlo. Un ejemplo de interacción estrecha es un programa concurrente con estado compartido, como se explica en el capítulo 8.

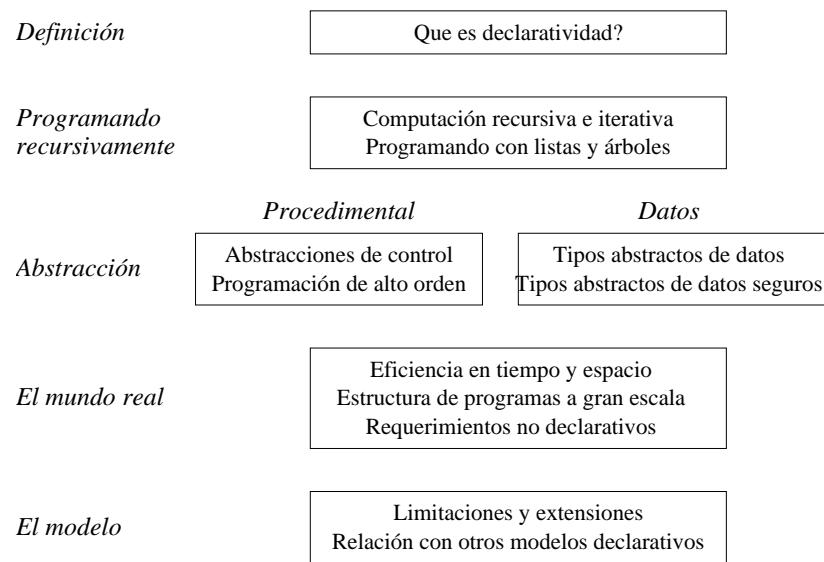


Figura 3.2: Estructura del capítulo.

Las interacciones estrechas son frecuentemente necesarias. Ellas no pueden ser “legalmente dejadas de lado” por programar en un modelo que no las soporte directamente (como se explicará claramente en la sección 4.8). Sin embargo, un principio importante es que sólo deben ser usadas cuando sea necesario y no en cualquier caso. Para soportar este principio, el total de componentes declarativos debería ser el mayor número posible.

Escribiendo programas declarativos

La forma más sencilla de escribir un programa declarativo es utilizar el modelo declarativo del capítulo 2. Las operaciones básicas sobre tipos de datos son declarativas, e.g., las operaciones aritméticas, y las operaciones sobre listas y registros. Si se siguen ciertas reglas, es posible combinar operaciones declarativas para crear operaciones declarativas nuevas. Combinar operaciones declarativas de acuerdo a las operaciones del modelo declarativo resultará en una operación declarativa. Esto se explica en la sección 3.1.3.

La regla estándar del álgebra que dice que “iguales se pueden reemplazar por iguales” es otro ejemplo de combinación declarativa. En los lenguajes de programación, esta propiedad se llama transparencia referencial. Esto simplifica grandemente el razonamiento sobre los programas. Por ejemplo, si sabemos que $f(a) = a^2$, entonces podemos reemplazar $f(a)$ por a^2 en cualquier lugar donde $f(a)$ aparezca. La ecuación $b = 7f(a)^2$ se vuelve $b = 7a^4$. Esto es posible porque $f(a)$ es declarativa: depende solamente de sus argumentos y no de ningún otro estado de la

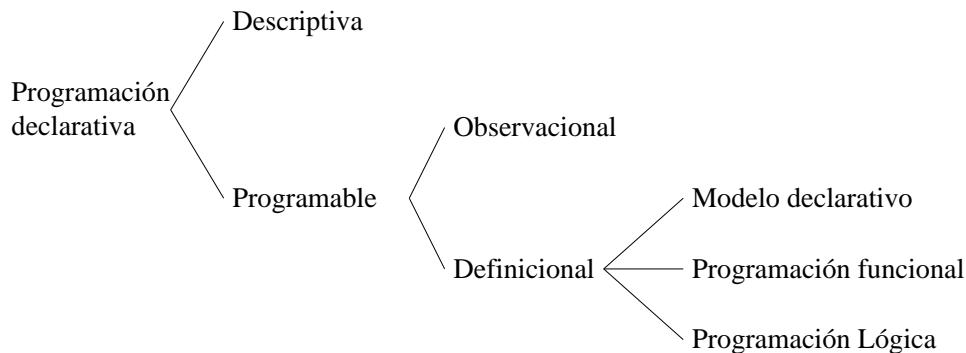


Figura 3.3: Una clasificación de la programación declarativa.

computación.

La técnica básica para escribir programas declarativos es considerar el programa como un conjunto de definiciones de funciones recursivas, utilizando programación de alto orden para simplificar la estructura del programa. Una función recursiva es una función en cuyo cuerpo se hace referencia a sí misma, directa o indirectamente. Recursión directa significa que la función misma es usada en el cuerpo. Recursión indirecta significa que la función hace referencia a otra función que referencia directa o indirectamente la función original. Programación de alto orden significa que las funciones pueden tener otras funciones como argumentos o como resultados. Esta capacidad es el fundamento de todas las técnicas para construir abstracciones que mostraremos en el libro. La programación de alto orden puede compensar de alguna manera la falta de expresividad del modelo declarativo, i.e., facilita el codificar ciertas formas de concurrencia y estado en el modelo declarativo.

Estructura del capítulo

Este capítulo explica cómo escribir programas declarativos prácticos. Este capítulo está organizado aproximadamente en las seis partes que se muestran en la figura 3.2. En la primera parte se define “declaratividad.” En la segunda parte se presenta una mirada global a las técnicas de programación. En la tercera y cuarta partes se explica abstracción procedural y de datos. En la quinta parte se muestra cómo la programación declarativa interactúa con el resto del ambiente de computación. En la sexta parte se reflexiona sobre la utilidad del modelo declarativo y se sitúa este modelo con respecto a otros modelos.

3.1. Qué es declaratividad?

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor

Tabla 3.1: El lenguaje núcleo declarativo descriptivo.

El modelo declarativo del capítulo 2 es una forma especialmente poderosa de escribir programas declarativos, pues todos los programas serán declarativos por el solo hecho de estar escritos en él. Pero esta es sólo una manera, entre muchas, de programar declarativamente. Antes de explicar cómo programar en el modelo declarativo, situaremos este modelo con respecto a otras formas de ser declarativo. También explicaremos por qué los programas escritos en este modelo siempre son declarativos.

3.1.1. Una clasificación de la programación declarativa

Hemos definido declaratividad de una forma particular, de manera que razonar sobre los programas se simplifique. Pero esta no es la única manera de precisar qué es la programación declarativa. Intuitivamente, es programar definiendo el qué (los resultados que queremos lograr) sin explicar el cómo (los algoritmos, etc., necesitados para lograr los resultados). Esta intuición, aunque un poco vaga, cubre muchas ideas diferentes. En la figura 3.3 se presenta una clasificación. El primer nivel de clasificación está basado en la expresividad. Hay dos posibilidades:

- Una declaratividad descriptiva. Esta es la menos expresiva. El “programa” declarativo sólo define una estructura de datos. En la tabla 3.1 se define un lenguaje a este nivel. ¡Con este lenguaje sólo se pueden definir registros! Este lenguaje contiene solamente las primeras cinco declaraciones del lenguaje núcleo de la tabla 2.1. En la sección 3.8.2 se muestra cómo utilizar este lenguaje para definir interfaces gráficas de usuario. Otros ejemplos son un lenguaje de formateo como HTML (Hypertext Markup Language), el cual define la estructura de un documento sin especificar cómo formatearlo, o un lenguaje de intercambio de información como XML (Extensible Markup Language), el cual es usado para intercambiar información en un formato abierto que puede ser leído fácilmente por todos. El nivel descriptivo es demasiado débil para escribir programas generales. Entonces, ¿por qué es interesante? Porque consiste de estructuras de datos con las que es sencillo hacer cálculos. Los registros de la tabla 3.1, los documentos HTML y XML, y las interfaces declarativas de usuario de la sección 3.8.2 pueden ser, todos, creados y transformados fácilmente, por un programa.

Técnicas de programación declarativa

- Una programación declarativa programable. Esta es tan expresiva como una máquina de Turing.¹ Por ejemplo, en la tabla 2.1 se define un lenguaje a este nivel. Vea la introducción al capítulo 6 para conocer un poco más sobre la relación entre los niveles descriptivo y programable.

Existen dos maneras fundamentalmente diferentes de ver la declaratividad programable:

- Una vista definicional, donde la declaratividad es una propiedad de la implementación del componente. Por ejemplo, los programas escritos en el modelo declarativo son declarativos debido a las propiedades del modelo.
- Una vista observacional, donde la declaratividad es una propiedad de la interfaz del componente. La vista observacional cumple el principio de abstracción: para usar un componente es suficiente conocer su especificación sin conocer su implementación. El componente sólo tiene que comportarse declarativamente, i.e., como si fuera independiente, sin estado, y determinístico, sin que necesariamente haya sido escrito en un modelo de computación declarativo.

Este libro utiliza tanto la vista definicional como la observacional. Cuando estamos interesados en mirar por dentro un componente, usamos la vista definicional. Cuando estamos interesados en ver cómo se comporta un componente, usamos la vista observacional.

Hay dos estilos de programación declarativa definicional que se han vuelto particularmente populares: el funcional y el lógico. En el estilo funcional, decimos que un componente definido como una función matemática es declarativo. Lenguajes funcionales como Haskell y Standard ML siguen este enfoque. En el estilo lógico, decimos que un componente definido como una relación lógica es declarativo. Lenguajes lógicos como Prolog y Mercury siguen este enfoque. Aunque es más difícil manipular formalmente programas funcionales o lógicos que programas descriptivos, los primeros todavía cumplen leyes algebraicas sencillas.² El modelo declarativo que se utiliza en este capítulo abarca tanto el estilo funcional como el lógico.

La vista observacional nos permite utilizar componentes declarativos en un programa declarativo aunque hayan sido escritos en un modelo no declarativo. Por ejemplo, una interfaz a una base de datos puede ser una adición valiosa para un lenguaje declarativo. Más aún, la implementación de esta interfaz muy seguramente no será ni lógica ni funcional. Es suficiente que pueda ser definida declarativamente. Algunas veces un componente declarativo estará escrito en un estilo funcional o

1. Una máquina de Turing es un modelo sencillo, formal, de computación, definido inicialmente por Alan Turing, que es tan poderoso como cualquier computador que se pueda construir hasta donde se conoce en el estado actual de las ciencias de la computación. Es decir, cualquier computación que pueda ser programada en cualquier computador también puede ser programada en una máquina de Turing.

2. Esta afirmación se hace para los programas que no utilizan las capacidades no declarativas de estos lenguajes.

lógico, y algunas veces no será así. En capítulos posteriores construiremos componentes declarativos en modelos no declarativos. No seremos dogmáticos al respecto; simplemente consideraremos que un componente es declarativo si se comporta declarativamente.

3.1.2. Lenguajes de especificación

Los proponentes de la programación declarativa afirman algunas veces que ésta les permite prescindir de la implementación, pues la especificación lo es todo. Esto es verdad en un sentido formal, pero no en un sentido práctico. En la práctica, los programas declarativos son muy parecidos a otros programas: requieren algoritmos, estructuras de datos, estructuración, y razonar sobre el orden de las operaciones. La causa de esto es que los lenguajes declarativos sólo pueden usar las matemáticas que se puedan implementar eficientemente. Existe un compromiso entre expresividad y eficiencia. Los programas declarativos son normalmente un poco más grandes que lo que la especificación podría ser. Por tanto, la distinción entre especificación e implementación aún tiene sentido, aún para programas declarativos.

Es posible definir un lenguaje declarativo mucho más expresivo que el que usamos en el libro. Tal lenguaje se llama un lenguaje de especificación. Normalmente es imposible implementar eficientemente lenguajes de especificación. Esto no significa que sean poco prácticos. Por el contrario, son una herramienta importante para pensar sobre los programas. Ellos pueden ser usados junto con un probador de teoremas, i.e., un programa que puede realizar cierto tipo de razonamientos matemáticos. Los probadores de teoremas prácticos no son completamente automáticos; ellos necesitan la ayuda humana. Pero pueden aliviar mucho del trabajo pesado para razonar sobre programas, i.e., la manipulación tediosa de fórmulas matemáticas. Con la ayuda del probador de teoremas, un desarrollador puede probar propiedades muy fuertes sobre su programa. Al hecho de utilizar un probador de teoremas de esta forma se le llama prueba de ingeniería. Hasta ahora, las pruebas de ingeniería son prácticas solamente para programas pequeños. Pero esto es suficiente para usarlo exitosamente cuando la seguridad es un aspecto de importancia crítica, e.g., cuando hay vidas en juego, como en un aparato médico o en el transporte público.

Los lenguajes de especificación están fuera del alcance de este libro.

3.1.3. Implementando componentes en el modelo declarativo

Combinar operaciones declarativas de acuerdo a las operaciones que provee el modelo declarativo siempre dará como resultado una operación declarativa. En esta sección se explica por qué esto es así. Primero vamos a definir más precisamente qué significa que una declaración sea declarativa. Dada cualquier declaración en el modelo declarativo, divide el conjunto de identificadores de variables libres de la declaración en dos conjuntos: el de los identificadores de entrada y el de los de salida. Entonces, dada cualquier ligadura de los identificadores de entrada con valores parciales y dejando los identificadores de salida como variables no-ligadas,

la ejecución de la declaración dará uno de los tres resultados siguientes: (1) alguna ligadura de los identificadores de salida, (2) suspensión, o (3) una excepción. Si la declaración es declarativa, entonces debe pasar que para la misma ligadura de los identificadores de entrada, el resultado siempre será el mismo.

Por ejemplo, considere la declaración `z=x`. Suponga que `x` es el identificador de entrada y `z` el de salida. Para cualquier ligadura de `x` a un valor parcial, ejecutar la declaración ligará `z` a ese mismo valor. Por lo tanto la declaración es declarativa.

Podemos usar este resultado para probar que la declaración

```
if X>Y then Z=X else Z=Y end
```

es declarativa.

Divida los tres identificadores de variables libres de la declaración, `X`, `Y`, `Z`, en dos identificadores de entrada `X` y `Y` y uno de salida `Z`. Entonces, si `X` y `Y` se ligan a cualesquiera valores parciales, la ejecución de la declaración o se bloqueará (no puede resolver si `X>Y`) o ligará `Z` siempre al mismo valor parcial (resuelve `X>Y` siempre de la misma manera). Por lo tanto la declaración es declarativa.

Podemos realizar este mismo razonamiento para todas las operaciones en el modelo declarativo:

- Primero, todas las operaciones en el modelo declarativo son declarativas. Esto incluye todas las operaciones sobre tipos básicos, las cuales se explicaron en el capítulo 2.
- Segundo, combinar operaciones declarativas con las construcciones del modelo declarativo produce una operación declarativa. Las cinco declaraciones compuestas siguientes existen en el modelo declarativo:
 - La declaración de secuencia.
 - La declaración `local`.
 - La declaración `if`.
 - La declaración `case`.
 - La definición de procedimiento, i.e., la declaración `<x>=<v>` donde `<v>` es un valor de tipo procedimiento.

Todas ellas permiten la construcción de declaraciones a partir de otras declaraciones. Todas estas formas de combinar declaraciones son determinísticas (si sus componentes son determinísticos, ellas lo son también) y no dependen de ningún contexto.

3.2. Computación iterativa

Ahora miraremos cómo programar en el modelo declarativo. Empezaremos mirando una clase muy sencilla de programas, aquellos que producen computaciones iterativas. Una computación iterativa es un ciclo que durante su ejecución mantiene el tamaño de la pila acotado por una constante, independientemente del número de

iteraciones del ciclo. Este tipo de computación es una herramienta básica de programación. Hay muchas formas de escribir programas iterativos, y no siempre es obvio determinar cuándo un programa es iterativo. Por eso, comenzaremos presentando un esquema general para construir muchas computaciones iterativas interesantes en el modelo declarativo.

3.2.1. Un esquema general

Una importante cantidad de computaciones iterativas comienzan en un estado inicial S_0 y transforman su estado en etapas sucesivas hasta llegar a un estado final S_{final} :

$$S_0 \rightarrow S_1 \rightarrow \dots \rightarrow S_{\text{final}}$$

Toda computación iterativa de esta forma se puede escribir como un esquema general:

```
fun {Iterar  $S_i$ }
  if {EsFinal  $S_i$ } then  $S_i$ 
  else  $S_{i+1}$  in
     $S_{i+1}=\{\text{Transformar } S_i\}$ 
    {Iterar  $S_{i+1}$ }
  end
end
```

En este esquema, las funciones *EsFinal* y *Transformar* son dependientes del problema. Probemos que cualquier programa que sigue este esquema es iterativo. Para ello, mostraremos que el tamaño de la pila no crece cuando se ejecuta *Iterar*. Por claridad, presentaremos solamente las declaraciones presentes en la pila semántica, dejando por fuera el contenido del ambiente y del almacén:

- Suponga que la pila semántica al principio es [$R=\{\text{Iterar } S_0\}$].
- Suponga que $\{\text{EsFinal } S_0\}$ devuelve **false**. Justo después de ejecutar la declaración **if**, la pila semántica es [$S_1=\{\text{Transformar } S_0\}, R=\{\text{Iterar } S_1\}$].
- Después de ejecutar $\{\text{Transformar } S_0\}$, la pila semántica es [$R=\{\text{Iterar } S_1\}$].

Vemos que la pila semántica tiene sólo un elemento en cada invocación recursiva, a saber, la declaración $R=\{\text{Iterar } S_{i+1}\}$. Este es el mismo argumento que usamos para mostrar la optimización de última invocación en la sección 2.5.1, sólo que presentado más concisamente.

3.2.2. Iteración con números

Un buen ejemplo de computación iterativa es el método de Newton para calcular la raíz cuadrada de un número positivo real x . La idea es comenzar con una adivinanza a de la raíz cuadrada, y mejorar esta adivinanza iterativamente hasta que sea suficientemente buena (cercana a la raíz cuadrada real). La adivinanza mejorada a' es el promedio de a y x/a :

```

fun {Raíz x}
    Adiv=1.0
in
    {RaízIter Adiv x}
end
fun {RaízIter Adiv X}
    if {Buena Adiv X} then Adiv
    else
        {RaízIter {Mejorar Adiv X} x}
    end
end
fun {Mejorar Adiv X}
    (Adiv + X/Adiv) / 2.0
end
fun {Buena Adiv X}
    {Abs X-Adiv*Adiv}/X < 0.00001
end
fun {Abs X} if X<0.0 then ~X else X end end

```

Figura 3.4: Encontrando raíces utilizando el método de Newton (primera versión).

$$a' = (a + x/a)/2.$$

Para ver que la adivinanza mejorada es realmente mejor, estudiemos la diferencia entre la adivinanza y \sqrt{x} :

$$\varepsilon = a - \sqrt{x}$$

Luego la diferencia entre a' y \sqrt{x} es

$$\varepsilon' = a' - \sqrt{x} = (a + x/a)/2 - \sqrt{x} = \varepsilon^2/2a$$

Para que haya convergencia, ε' debe ser menor que ε . Miremos qué condiciones impone esto sobre x y a . La condición $\varepsilon' < \varepsilon$ es la misma que $\varepsilon^2/2a < \varepsilon$, lo cual es lo mismo que $\varepsilon < 2a$. (Suponiendo que $\varepsilon > 0$, pues si no lo es, empezamos con ε' , el cual siempre es mayor que 0.) Substituyendo por la definición de ε , la condición resultante es $\sqrt{x} + a > 0$. Si $x > 0$ y la adivinanza inicial $a > 0$, entonces esta condición siempre es cierta. Por lo tanto el algoritmo siempre converge.

En la figura 3.4 se muestra una forma de definir el método de Newton como una computación iterativa. La función `{RaízIter Adiv x}` invoca `{RaízIter {Mejorar Adiv x} x}` hasta que `Adiv` satisface la condición `{Buena Adiv x}`. Queda claro que esta es una instancia del esquema general, o sea que es una computación iterativa. La adivinanza mejorada se calcula de acuerdo a la fórmula presentada anteriormente. La comprobación de que la solución es suficientemente “buena” es $|x - a^2|/x < 0,00001$, i.e., la raíz cuadrada tiene que ser exacta en sus primeras cinco cifras decimales. Esta comprobación es relativa, i.e., el error se divide por x . Podríamos usar una comprobación absoluta. e.g., algo como $|x - a^2| < 0,00001$, donde la magnitud del error tiene que ser menor a una constante

```

local
  fun {Mejorar Adiv X}
    (Adiv + X/Adiv) / 2.0
  end
  fun {Buena Adiv X}
    {Abs X-Adiv*Adiv}/X < 0.00001
  end
  fun {RaízIter Adiv X}
    if {Buena Adiv X} then Adiv
    else
      {RaízIter {Mejorar Adiv X} X}
    end
  end
in
  fun {Raíz X}
    Adiv=1.0
  in
    {RaízIter Adiv X}
  end
end

```

Figura 3.5: Encontrando raíces utilizando el método de Newton (segunda versión).

dada. ¿Por qué es mejor usar una comprobación relativa cuando se calculan raíces cuadradas?

3.2.3. Usando procedimientos locales

En el programa del método de Newton de la figura 3.4, se definieron varias rutinas “auxiliares”: RaízIter, Mejorar, Buena, y Abs. Estas rutinas son los cimientos de la función principal Raíz. En esta sección discutiremos dónde definir las rutinas auxiliares. El principio básico es que una rutina auxiliar definida solamente como una ayuda para definir otra rutina, no debe ser visible en todas partes. (Usamos la palabra “rutina” tanto para funciones como para procedimientos).

En el ejemplo del método de Newton, RaízIter sólo se necesita dentro de Raíz, Mejorar y Buena sólo se necesitan dentro de RaízIter, y Abs es una función utilitaria genérica que puede ser usada en todas partes. Hay básicamente dos formas de expresar esta visibilidad, con algunas diferencias semánticas. La primera forma se muestra en la figura 3.5: las rutinas auxiliares se definen afuera de Raíz en una declaración **local**. La segunda forma se muestra en la figura 3.6: cada rutina auxiliar se define dentro de la rutina que la necesita.³

En la figura 3.5, existe un compromiso entre legibilidad y visibilidad: Mejorar y Buena podrían ser definidas locales a RaízIter solamente. Esto llevaría a dos

3. Dejamos por fuera la definición de Abs para evitar repeticiones innecesarias.

```
fun {Raíz x}
    fun {RaízIter Adiv x}
        fun {Mejorar Adiv x}
            (Adiv + X/Adiv) / 2.0
        end
        fun {Buena Adiv x}
            {Abs X-Adiv*Adiv}/x < 0.00001
        end
    in
        if {Buena Adiv x} then Adiv
        else
            {RaízIter {Mejorar Adiv x} x}
        end
    end
    Adiv=1.0
in
{RaízIter Adiv x}
end
```

Figura 3.6: Encontrando raíces utilizando el método de Newton (tercera versión).

niveles de declaraciones locales, lo cual es más difícil de leer. Hemos decidido entonces colocar las tres rutinas auxiliares en la misma declaración local.

En la figura 3.6, cada rutina auxiliar ve como referencias externas, los argumentos de la rutina que la encierra. Estos argumentos son precisamente aquellos con los cuales se invocan las rutinas auxiliares. Esto significa que podríamos simplificar al definición eliminando estos argumentos de las rutinas auxiliares. El resultado se muestra en la figura 3.7.

Existe un compromiso entre colocar las definiciones auxiliares por fuera de la rutina que las necesita o colocarlas por dentro:

- Colocarlas por dentro (figuras 3.6 y 3.7) les deja ver los argumentos de las rutinas principales como referencias externas, de acuerdo a la regla de alcance léxico (ver sección 2.4.3). Por lo tanto, se necesitan menos argumentos. Pero cada vez que se invoca la rutina principal, se crean nuevas rutinas auxiliares. Esto significa que se crean nuevos valores de tipo procedimiento.
- Colocarlas por fuera (figuras 3.4 y 3.5) significa que los valores de tipo procedimiento se crean una sola vez para siempre, para todas las invocaciones de la rutina principal. Pero entonces las rutinas auxiliares necesitan más argumentos para que la rutina principal pueda pasárselas la información.

En la figura 3.7, en cada iteración de `RaízIter`, se crean nuevamente los valores de tipo procedimiento correspondientes a `Mejorar` y `Buena`, mientras que la misma `RaízIter` sólo se crea una vez. Esto parece un buen compromiso, donde `RaízIter` es local a `Raíz` y tanto `Mejorar` como `Buena` están por fuera de `RaízIter`. Esto nos lleva a la definición final de la figura 3.8, la cual consideramos la mejor en términos

```

fun {Raíz X}
  fun {RaízIter Adiv}
    fun {Mejorar}
      (Adiv + X/Adiv) / 2.0
    end
    fun {Buena}
      {Abs X-Adiv*Adiv}/X < 0.00001
    end
  in
    if {Buena} then Adiv
    else
      {RaízIter {Mejorar}}
    end
  end
  Adiv=1.0
in
  {RaízIter Adiv}
end

```

Figura 3.7: Encontrando raíces utilizando el método de Newton (cuarta versión).

tanto de eficiencia como de visibilidad.

3.2.4. Del esquema general a una abstracción de control

El esquema general de la sección 3.2.1 le ayuda al programador a diseñar programas eficientes, pero no es visible para el modelo de computación. Vayamos una etapa más allá y proveamos el esquema general como un componente que pueda ser usado por otros componentes. Decimos que el esquema general se comporta como una abstracción de control, i.e., una abstracción que puede ser usada para proveer un flujo de control deseado. Este es el esquema general:

```

fun {Iterar  $S_i$ }
  if {EsFinal  $S_i$ } then  $S_i$ 
  else  $S_{i+1}$  in
     $S_{i+1} = \{Transformar\ S_i\}$ 
    {Iterar  $S_{i+1}$ }
  end
end

```

Este esquema implementa un ciclo **while** general con un resultado calculado. Para que el esquema se vuelva una abstracción de control, tenemos que parametrizarlo extrayendo las partes que varían de uno a otro uso. Hay dos partes de esas: las funciones *EsFinal* y *Transformar*. Colocamos estas dos partes como parámetros de *Iterar*:

```

fun {Raíz x}
  fun {Mejorar Adiv}
    (Adiv + X/Adiv) / 2.0
  end
  fun {Buena Adiv}
    {Abs X-Adiv*Adiv}/X < 0.00001
  end
  fun {RaízIter Adiv}
    if {Buena Adiv} then Adiv
    else
      {RaízIter {Mejorar Adiv}}
    end
  end
  Adiv=1.0
in
  {RaízIter Adiv}
end

```

Figura 3.8: Encontrando raíces utilizando el método de Newton (quinta versión).

```

fun {Iterar S EsFinal Transformar}
  if {EsFinal S} then S
  else S1 in
    S1={Transformar S}
    {Iterar S1 EsFinal Transformar}
  end
end

```

Para utilizar esta abstracción de control, los argumentos `EsFinal` y `Transformar` se presentan como funciones de un argumento. Pasar funciones como argumentos de funciones es parte de un rango de técnicas de programación llamadas programación de alto orden. Estas técnicas se explican después en la sección 3.6. Podemos hacer que `Iterar` se comporte exactamente como `RaízIter` pasándole las funciones `Buena` y `Mejorar`. Esto se puede escribir como sigue:

```

fun {Raíz x}
  {Iterar
    1.0
    fun {$ Ad} {Abs X-Ad*Ad}/X<0.00001 end
    fun {$ Ad} (Ad+X/Ad)/2.0 end}
  end

```

Aquí se utilizan dos valores de tipo función como argumentos de la abstracción de control. Esta es una forma poderosa de estructurar un programa pues separa el flujo de control general de su uso particular. La programación de alto orden es especialmente útil para estructurar programas de esta manera. Si esta abstracción de control se utiliza frecuentemente, una etapa siguiente será proveerla como abstracción lingüística.

3.3. Computación recursiva

Las computaciones iterativas son un caso especial de una clase más general de computación, llamada computación recursiva. Miremos la diferencia entre las dos. Recuerde que una computación iterativa puede ser considerada, simplemente, como un ciclo en el que una cierta acción se repite algún número de veces. En la sección 3.2 se implementa esto en el modelo declarativo introduciendo una abstracción de control, la función `Iterar`. La función comprueba primero si una condición se cumple. Si la condición no se cumple, entonces realiza una acción y se invoca ella misma nuevamente.

La recursión es más general que esto. Una función recursiva puede invocarse a sí misma en cualquier lugar en el cuerpo y puede hacerlo más de una vez. En programación, la recursión se presenta de dos maneras principalmente: en funciones y en tipos de datos. Una función es recursiva si su definición tiene al menos una invocación a sí misma. La abstracción de iteración de la sección 3.2 es un caso sencillo. Un tipo de datos es recursivo si está definido en términos de sí mismo. Por ejemplo, una lista está definida en términos de listas más pequeñas. Las dos formas de recursión están fuertemente relacionadas pues las funciones recursivas se suelen usar para calcular con tipos de datos recursivos.

Vimos que una computación iterativa tiene un tamaño de pila constante, como consecuencia de la optimización de última invocación. Este no es el caso siempre en computación recursiva. El tamaño de la pila puede crecer a medida que el tamaño de la entrada lo hace. Algunas veces es inevitable, e.g., cuando se realizan cálculos con árboles, como lo veremos más adelante. Una parte importante de la programación declarativa es evitar que el tamaño de la pila crezca, siempre que sea posible hacerlo. En esta sección se presenta un ejemplo de cómo hacerlo. Empezamos con un caso típico de computación recursiva que no es iterativa, a saber, la definición ingenua de la función factorial. La definición matemática es:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \text{ si } n > 0 \end{aligned}$$

Esta es una ecuación de recurrencia, i.e., el factorial $n!$ se define en términos del factorial con argumentos más pequeños, a saber, $(n-1)!$. El programa ingenuo sigue esta definición matemática. Para calcular `{Fact N}` hay dos posibilidades, a saber, $N=0$ o $N>0$. En el primer caso, devuelve 1. En el segundo caso, calcula `{Fact N-1}`, lo multiplica por N , y devuelve el resultado. Esto produce el programa siguiente:

```
fun {Fact N}
  if N==0 then 1
  elseif N>0 then N*{Fact N-1}
  else raise errorDeDominio end
  end
end
```

Este programa define el factorial de un número grande en términos del factorial de uno más pequeño. Como todos los números son no negativos, ellos decrecerán hasta

cero y la ejecución terminará.

Note que factorial es una función parcial. No está definida para N negativo. El programa refleja este hecho lanzando una excepción en esos casos. La definición del capítulo 1 tiene un error pues para N negativo se queda en un ciclo infinito.

Hemos hecho dos cosas al escribir `Fact`. Primero, seguimos la definición matemática para llegar a una implementación correcta. Segundo, razonamos sobre la terminación, i.e., mostramos que el programa termina para argumentos de entrada legales, i.e., argumentos dentro del dominio de la función.

3.3.1. Tamaño de la pila creciente

Esta definición de factorial, presenta una computación cuyo máximo tamaño de la pila es proporcional al argumento N de la función. Podemos ver esto usando la semántica. Primero traduzca `Fact` en el lenguaje núcleo:

```
proc {Fact N ?R}
  if N==0 then R=1
  elseif N>0 then N1 R1 in
    N1=N-1
    {Fact N1 R1}
    R=N*R1
  else raise errorDeDominio end
  end
end
```

Desde ya podemos adivinar que el tamaño de la pila crece, pues la multiplicación se realiza después de la invocación recursiva. Es decir, durante la invocación recursiva la pila tiene que guardar información sobre la multiplicación para cuando se regrese de ésta. Sigamos la semántica y calculemos a mano qué pasa cuando se ejecuta la invocación `{Fact 5 R}`. Por claridad, simplifiquemos ligeramente la presentación de la máquina abstracta, substituyendo el valor de una variable del almacén en el ambiente. Es decir, el ambiente $\{\dots, N \rightarrow n, \dots\}$ se escribe como $\{\dots, N \rightarrow 5, \dots\}$ si el almacén es $\{\dots, n = 5, \dots\}$.

- La pila semántica inicial es $\{(\{Fact N R\}, \{N \rightarrow 5, R \rightarrow r_0\})\}$.
- En la primera invocación:

```
[(\{Fact N1 R1\}, \{N1 \rightarrow 4, R1 \rightarrow r_1, \dots\}),
 (R=N*R1, \{R \rightarrow r_0, R1 \rightarrow r_1, N \rightarrow 5, \dots\})]
```

- En la segunda invocación:

```
[(\{Fact N1 R1\}, \{N1 \rightarrow 3, R1 \rightarrow r_2, \dots\}),
 (R=N*R1, \{R \rightarrow r_1, R1 \rightarrow r_2, N \rightarrow 4, \dots\}),
 (R=N*R1, \{R \rightarrow r_0, R1 \rightarrow r_1, N \rightarrow 5, \dots\})]
```

- En la tercera invocación:

$$\begin{aligned} & [(\{\text{Fact N1 R1}\}, \{N1 \rightarrow 2, R1 \rightarrow r_3, \dots\}), \\ & (\text{R=N*R1}, \{R \rightarrow r_2, R1 \rightarrow r_3, N \rightarrow 3, \dots\}), \\ & (\text{R=N*R1}, \{R \rightarrow r_1, R1 \rightarrow r_2, N \rightarrow 4, \dots\}), \\ & (\text{R=N*R1}, \{R \rightarrow r_0, R1 \rightarrow r_1, N \rightarrow 5, \dots\})] \end{aligned}$$

Es claro que la pila aumenta su tamaño en una declaración más por invocación. La última invocación recursiva es la quinta, la cual termina inmediatamente con $r_5 = 1$. Las cinco multiplicaciones se realizan para llegar al resultado final $r_0 = 120$.

3.3.2. Máquina abstracta basada en sustituciones

Este ejemplo muestra que la máquina abstracta del capítulo 2 puede ser bastante incómoda para realizar cálculos a mano. Esto pasa porque la máquina conserva tanto los identificadores de variables como las variables del almacén, usando ambientes para llevar las asociaciones entre unos y otros. Esto es realista; así es cómo está implementada la máquina abstracta en un computador. Pero no es agradable para realizar cálculos a mano.

Podemos hacer un cambio sencillo a la máquina abstracta que la haga mucho más fácil de usar para realizar cálculos a mano. La idea es reemplazar los identificadores que aparecen en las declaraciones por las entidades del almacén a las que ellos refieren. Esta operación es llamada sustitución. Por ejemplo, la declaración R=N*R1 se vuelve $r_2 = 3 * r_3$ cuando se substituye de acuerdo al ambiente $\{R \rightarrow r_2, N \rightarrow 3, R1 \rightarrow r_3\}$.

La máquina abstracta basada en sustituciones no tiene ambientes. Ella sustituye directamente los identificadores en las declaraciones por entidades del almacén. Para el ejemplo del factorial recursivo, las cosas se verían así:

- La pila semántica inicialmente es $[\{\text{Fact } 5\ r_0\}]$.
- A la primera invocación: $[\{\text{Fact } 4\ r_1\}, r_0=5*r_1]$.
- A la segunda invocación: $[\{\text{Fact } 3\ r_2\}, r_1=4*r_2, r_0=5*r_1]$.
- A la tercera invocación: $[\{\text{Fact } 2\ r_3\}, r_2=3*r_3, r_1=4*r_2, r_0=5*r_1]$.

Al igual que antes, vemos que la pila aumenta su tamaño en una declaración más por invocación. Resumimos las diferencias entre las dos versiones de la máquina abstracta:

- La máquina abstracta basada en ambientes, definida en el capítulo 2, es fiel a la implementación sobre un computador real, la cual utiliza ambientes. Sin embargo, los ambientes introducen un nivel adicional de indirección, de modo que son difíciles de usar para realizar cálculos a mano.
- La máquina abstracta basada en sustituciones es más fácil de usar para realizar cálculos a mano, pues hay muchos menos símbolos para manipular. Sin embargo, las sustituciones son costosas de implementar, por lo que generalmente no se usan.

en implementaciones reales.

Ambas versiones hacen las mismas ligaduras en el almacén y las mismas manipulaciones de la pila semántica.

3.3.3. Convirtiendo una computación recursiva en iterativa

La función factorial desarrollada es suficientemente sencilla para reorganizarla y convertirla en una versión iterativa. Miremos cómo hacerlo. Más adelante, presentaremos una manera sistemática de construir computaciones iterativas. Por ahora, sólo presentamos la idea. En el cálculo anterior:

```
R=(5*(4*(3*(2*(1*1))))))
```

Si reorganizamos los números así:

```
R=((((1*5)*4)*3)*2)*1
```

entonces los cálculos se podrían realizar incrementalmente, comenzando con $1*5$. Esto da 5, luego 20, luego 60, luego 120, y finalmente 120. La definición iterativa que realiza los cálculos de esta manera es:

```
fun {Fact N}
  fun {FactIter N A}
    if N==0 then A
    elseif N>0 then {FactIter N-1 A*N}
    else raise errorDeDominio end
    end
  end
in
  {FactIter N 1}
end
```

La función que realmente realiza la iteración, `FactIter`, tiene un segundo argumento `A`. Este argumento es crucial; sin él una versión iterativa de factorial sería imposible. El segundo argumento no es natural de acuerdo a la definición matemática sencilla de factorial que hemos usado. Necesitamos razonar un poco para entender su necesidad y volverlo natural.

3.4. Programando recursivamente

Las computaciones recursivas están en el corazón de la programación declarativa. En esta sección se muestra cómo escribir en este estilo. Mostramos las técnicas básicas de programación con listas, árboles, y otros tipos de datos recursivos. Mostraremos, cuando sea posible, cómo lograr que las computaciones sean iterativas. Esta sección está organizada de la manera siguiente:

- El primer paso es la *definición* de tipos de datos recursivos. En la sección 3.4.1 se presenta una notación simple que nos permite definir los tipos de datos recursivos

más importantes.

- El tipo de dato recursivo más importante es la *lista*. En la sección 3.4.2 se presentan las técnicas básicas de programación con listas.
- Los programas declarativos eficientes tienen que realizar computaciones iterativas. En la sección 3.4.3 se presentan los *acumuladores*, una técnica sistemática para lograr esto.
- Las computaciones construyen con cierta frecuencia estructuras de datos de manera incremental. En la sección 3.4.4 se presentan las *listas de diferencias*, una técnica eficiente para lograr esto, conservando el carácter iterativo de la computación.
- Un tipo de datos, importante, relacionado con la lista es la *cola*. En la sección 3.4.5 se muestra cómo implementar colas eficientemente. También se introduce la idea básica de eficiencia amortizada.
- El segundo tipo de datos recursivo más importante, después de las estructuras lineales como listas o colas, es el *árbol*. En la sección 3.4.6 se presentan las técnicas básicas para programar con árboles.
- En las secciones 3.4.7 y 3.4.8 se presentan dos *casos de estudio* reales, un algoritmo para dibujar árboles y un analizador sintáctico; entre los dos utilizan muchas de las técnicas de esta sección.

3.4.1. Notación de tipos

El tipo lista es un subconjunto del tipo registro. Hay otros subconjuntos del tipo registro que son útiles, e.g., los árboles binarios. Antes de pasar a escribir programas, introducimos una notación sencilla para definir listas, árboles, y otros subtipos de los registros. Esto nos ayudará a escribir funciones sobre estos tipos.

Una lista x_s se define como nil o como $x|x_r$ donde x_r es una lista. Otros subconjuntos del tipo registro también son útiles. Por ejemplo, un árbol binario se puede definir como un nodo hoja hoja o un nodo interior $\text{árbol}(\text{lave}:K \text{ valor}:V \text{ izq}:A_l \text{ der}:A_d)$ donde A_l y A_d son ambos árboles binarios. ¿Cómo podemos escribir estos tipos en una forma concisa? Crearemos una notación basada en la notación de gramáticas independientes del contexto que se usa para definir la sintaxis del lenguaje núcleo. Los símbolos no terminales representan tipos o valores. Usaremos la jerarquía de tipos de la figura 2.16 como una base: todos los tipos de esta jerarquía estarán disponibles como símbolos no terminales predefinidos. Por tanto $\langle\text{Valor}\rangle$ y $\langle\text{Registro}\rangle$ existen ambos, y como representan conjuntos de valores, podemos decir que $\langle\text{Registro}\rangle \subset \langle\text{Valor}\rangle$. Ahora podemos definir las listas:

```
 $\langle\text{Lista}\rangle ::= \text{nil}$ 
|  $\langle\text{Valor}\rangle \cdot | \cdot \langle\text{Lista}\rangle$ 
```

Esto significa que un valor está en el tipo $\langle\text{Lista}\rangle$ si tiene una de las dos formas: O es de la forma $x|x_r$ donde x es del tipo $\langle\text{Valor}\rangle$ y x_r es del tipo $\langle\text{Lista}\rangle$; o es el

átomo `nil`. Esta es una definición recursiva del tipo $\langle \text{Lista} \rangle$. Se puede probar que existe sólo un conjunto $\langle \text{Lista} \rangle$, de hecho el conjunto más pequeño, que satisface esta definición. La prueba está más allá del alcance de este libro, pero se puede encontrar en cualquier libro introductorio sobre semántica, e.g., [186]. Tomamos este conjunto más pequeño como el valor del tipo $\langle \text{Lista} \rangle$. Intuitivamente, el tipo $\langle \text{Lista} \rangle$ se puede construir comenzando con `nil` y la aplicación repetida de la regla gramatical para construir listas cada vez más grandes.

También podemos definir listas cuyos elementos sean de un tipo específico:

```
 $\langle \text{Lista } T \rangle ::= \text{nil}$ 
|  $T^* \mid [ \langle \text{Lista } T \rangle ]$ 
```

Aquí T es una variable de tipo y $\langle \text{Lista } T \rangle$ es una función de tipo. Al aplicar la función de tipo en un tipo cualquiera, el resultado es una lista de ese tipo. Por ejemplo, $\langle \text{Lista } \langle \text{Ent} \rangle \rangle$ da como resultado el tipo lista de enteros. Observe que $\langle \text{Lista } \langle \text{Valor} \rangle \rangle$ es igual a $\langle \text{Lista} \rangle$ (pués las definiciones son idénticas).

Definamos un árbol binario con llaves en forma de literales y elementos del tipo T :

```
 $\langle \text{ArbolBin } T \rangle ::= \text{hoja}$ 
|  $\text{árbol}(\text{llave}: \langle \text{Literal} \rangle \text{ valor}: T$ 
 $\qquad \qquad \qquad \text{izq}: \langle \text{ArbolBin } T \rangle \text{ der}: \langle \text{ArbolBin } T \rangle)$ 
```

El tipo de un procedimiento es $\langle \text{proc } \{ \$ T_1 \dots T_n \} \rangle$, donde T_1, \dots, T_n son los tipos de sus argumentos. El tipo de un procedimiento es llamado algunas veces la firma del procedimiento, porque da información clave del procedimiento en forma concisa. El tipo de una función es $\langle \text{fun } \{ \$ T_1 \dots T_n \} : T \rangle$, lo cual es equivalente a $\langle \text{proc } \{ \$ T_1 \dots T_n T \} \rangle$. Por ejemplo, el tipo $\langle \text{fun } \{ \$ \langle \text{Lista} \rangle \langle \text{Lista} \rangle \} : \langle \text{Lista} \rangle \rangle$ representa los valores de tipo función, que reciben dos valores de tipo lista como argumentos y devuelven un valor de tipo lista como resultado.

Límites de la notación

Esta notación de tipos puede definir muchos conjuntos de valores útiles, pero su expresividad es, definitivamente, limitada. Aquí presentamos algunos casos donde la notación no es suficientemente buena:

- La notación no permite definir los enteros positivos, i.e., el subconjunto de $\langle \text{Ent} \rangle$ cuyos elementos son los mayores que cero.
- La notación no permite definir conjuntos de valores parciales. Por ejemplo, las listas de diferencias no se pueden definir.

Podemos extender la notación para manejar el primer caso, e.g., añadiendo condiciones booleanas.⁴ En los ejemplos que siguen, añadiremos estas condiciones en el texto cuando se necesiten. Esto significa que la notación de tipos es descriptiva: proporciona afirmaciones lógicas sobre el conjunto de valores que puede tomar una variable. No hay exigencia en que los tipos puedan ser comprobados por un compilador. Por el contrario, con frecuencia no son comprobables. Aún algunos tipos que son muy sencillos de especificar, como los enteros positivos, no pueden, en general, ser comprobados por un compilador.

3.4.2. Programando con listas

Los valores de tipo lista se crean y se descomponen de una manera muy concisa, y sin embargo son suficientemente poderosos para codificar cualquier clase de estructura compleja. El lenguaje Lisp original debe gran parte de su poder a esta idea [111]. Gracias a la estructura simple de las listas, la programación declarativa con ellas es fácil y poderosa. En esta sección se presentan las técnicas básicas de programación con listas:

- *Pensando recursivamente.* El enfoque básico es resolver problemas en términos de la solución a versiones más pequeñas del mismo problema.
- *Convirtiendo computaciones recursivas en iterativas.* Los programas ingenuos sobre listas son, con frecuencia, derrochadores, en el sentido que el tamaño de la pila crece a medida que lo hace el tamaño de la entrada. Mostraremos cómo usar transformaciones de estado para volverlos prácticos.
- *Corrección de las computaciones iterativas.* Una manera sencilla y poderosa de razonar sobre computaciones iterativas es el uso de invariantes de estado.
- *Construyendo programas según el tipo de los datos.* Una función que realiza cálculos con elementos de un tipo particular, casi siempre tiene una estructura recursiva que refleja cercanamente la definición de ese tipo.

Terminamos esta sección con un ejemplo más grande, el algoritmo *mergesort*. En secciones posteriores se muestra cómo lograr la escritura de funciones iterativas de una manera más sistemática por medio del uso de acumuladores y de listas de diferencias. Esto nos llevará a escribir funciones iterativas desde un comienzo. En nuestro concepto estas técnicas son “escalables”, i.e., ellas funcionan bien, aún para programas declarativos de gran tamaño.

3.4.2.1. Pensando recursivamente

Una lista es una estructura de datos recursiva, i.e., está definida en términos de versiones más pequeñas de ella misma. Para escribir una función que calcula sobre

4. Esto es similar a la forma de definir la sintaxis del lenguaje en la sección 2.1.1: una notación independiente del contexto con condiciones adicionales donde se necesiten.

Técnicas de programación declarativa

listas tenemos que seguir su estructura recursiva. La función consta de dos partes:

- Un caso básico, para listas de tamaño menor (digamos, de cero, uno o dos elementos). Para este caso la función calcula la respuesta directamente.
- Un caso recursivo, para listas más grandes. En este caso, la función calcula el resultado en términos de sus resultados sobre una o más listas de menor tamaño.

Como nuestro primer ejemplo, tomaremos una función recursiva sencilla que calcula el tamaño o longitud de una lista siguiendo esta técnica:⁵

```
fun {Length Ls}
  case Ls
  of nil then 0
    [] _|Lr then 1+{Length Lr}
  end
end
{Browse {Length [a b c]}}
```

El tipo de esta función es $\langle \text{fun} \; \{ \$ \; \langle \text{Lista} \rangle : \langle \text{Ent} \rangle \} : \langle \text{Ent} \rangle \rangle$, es decir, una función que recibe una lista y devuelve un entero. El caso básico es cuando recibe la lista vacía `nil`, en cuyo caso la función devuelve `0`. El caso recursivo es cuando recibe cualquier otra lista. Si la lista tiene longitud n , es porque su cola tiene longitud $n - 1$. Es decir, la longitud de la lista de entrada es la longitud de su cola más uno. Como la cola es siempre de menor tamaño que la lista original, entonces el programa terminará (en algún momento dejará de invocarse recursivamente).

Nuestro segundo ejemplo es una función que concatena⁶ dos listas `Ls` y `Ms` en una tercera lista, que será el resultado. La pregunta es, ¿sobre cuál lista realiza la recursión? ¿Sobre la primera o sobre la segunda? Nuestra respuesta es que la recursión debe realizarse sobre la primera lista. La función resultante es:

```
fun {Append Ls Ms}
  case Ls
  of nil then Ms
    [] X|Lr then X|{Append Lr Ms}
  end
end
```

Su tipo es $\langle \text{fun} \; \{ \$ \; \langle \text{Lista} \rangle \; \langle \text{Lista} \rangle : \langle \text{Lista} \rangle \} : \langle \text{Lista} \rangle \rangle$. Esta función sigue fielmente las dos propiedades siguientes de la concatenación:

$$\begin{aligned}\text{concatenar}(\text{nil}, m) &= m \\ \text{concatenar}(x|l, m) &= x | \text{concatenar}(l, m)\end{aligned}$$

El caso recursivo siempre invoca a `Append` con un primer argumento de menor tamaño, por lo cual el programa siempre termina.

5. Nota del traductor: Como en Mozart existe un módulo sobre listas que ya contiene estas funciones preconstruidas, se ha dejado en la traducción, el nombre de las funciones en inglés, tal como se llaman en ese módulo.

6. Nota del traductor: `Append`, en inglés.

3.4.2.2. Funciones recursivas y sus dominios

Definamos la función `Nth` para calcular el enésimo elemento de una lista.

```
fun {Nth xs n}
  if n==1 then xs.1
  elseif n>1 then {Nth xs.2 n-1}
  end
end
```

Su tipo es `{fun { $ <Lista> <Ent>} : <Valor>}`. Recuerde que una lista `xs` o es `nil` o es una tupla `x|y` con dos argumentos. `xs.1` da `x` y `xs.2` da `y`. ¿Qué pasa cuando se alimenta la interfaz con lo siguiente?:

```
{Browse {Nth [a b c d] 5}}
```

La lista sólo tiene cuatro elementos. Tratar de averiguar el quinto significa tratar de calcular `xs.1` o `xs.2` con `xs=nil`. Esto lanzará una excepción. También se lanzará una excepción si `n` no es un entero positivo. e.g., cuando `n=0`. Esto pasa porque no existe cláusula `else` en la declaración `if`.

Este es un ejemplo de una técnica general para definir funciones: utilice siempre declaraciones que lancen excepciones cuando los valores están por fuera de sus dominios. Esto maximizará las posibilidades que la función, como un todo, lance una excepción cuando sea invocada con una entrada con un valor por fuera del dominio. No podemos garantizar que siempre se lance una excepción en este caso, e.g., `{Nth 1|2|3 2}` devuelve 2 pero `1|2|3` no es una lista. Tales garantías son difíciles de ofrecer. Se pueden obtener, en algunos casos, en lenguajes estáticamente tipados.

La declaración `case` también se comporta correctamente desde este punto de vista. Al usar una declaración `case` para recurrir sobre una lista, se lanzará una excepción si su argumento no es una lista. Por ejemplo, definamos una función que suma todos los elementos de una lista de enteros:

```
fun {SumList xs}
  case xs
  of nil then 0
  [] x|xr then x+{SumList xr}
  end
end
```

Su tipo es `{fun { $ <Lista <Ent>>} : <Ent>}`. La entrada debe ser una lista de enteros pues `SumList` utiliza internamente el entero 0. La invocación siguiente

```
{Browse {SumList [1 2 3]}}
```

muestra 6. Como `xs` puede ser cualquiera de los dos valores, a saber, `nil` o `x|xr`, es natural usar una declaración `case`. Como en el ejemplo de `Nth`, el no utilizar una cláusula `else` lanzará una excepción si el argumento tiene un valor por fuera del dominio de la función. Por ejemplo:

```
{Browse {SumList 1|foo}}
```

lanza una excepción porque `1 | foo` no es una lista, y la definición de `SumList` supone que su entrada es una lista.

3.4.2.3. Las definiciones ingenuas con frecuencia son lentas

Definamos una función para invertir el orden de los elementos de una lista. Comencemos con una definición recursiva de la inversión de una lista:

- La inversión de `nil` da `nil`.
- La inversión de `x|xs` da `z`, donde
 - `ys` es la inversión de `xs`, y
 - `z` es la concatenación de `ys` y `[x]`.

Esto funciona porque `x` se mueve del primer lugar de la lista al último. De acuerdo a esta definición recursiva, podemos escribir inmediatamente la función `Reverse`:

```
fun {Reverse xs}
  case xs
  of nil then nil
  [] x|xr then
    {Append {Reverse xr} [x]}
  end
end
```

Su tipo es `<fun {$_ <Lista>} : <Lista>`. ¿Qué tan eficiente es esta función? Para responder esta pregunta, tenemos que calcular su tiempo de ejecución dada una entrada de tamaño n . Podemos hacer este análisis rigurosamente, con las técnicas de la sección 3.5. Pero aún sin esas técnicas, podemos ver intuitivamente qué pasa. Habrá n invocaciones recursivas a `Reverse` seguidas por n invocaciones a `Append`. Cada invocación a `Append` se realizará con una lista de tamaño $n/2$ en promedio. El tiempo total de ejecución será, por tanto, proporcional a $n \cdot n/2$, a saber, n^2 . Esto es bastante lento. Esperaríamos que invertir el orden de una lista, lo cual no parece precisamente una operación compleja, tomara tiempo proporcional al tamaño de la lista de entrada y no a su cuadrado.

Este programa tiene un segundo defecto: el tamaño de la pila crece con el tamaño de la lista de entrada, i.e., se define una computación recursiva que no es iterativa. Seguir ingenuamente la definición recursiva de la inversión del orden de una lista, nos ha llevado a un resultado ¡bastante ineficiente! Afortunadamente, existen técnicas sencillas para resolver ambas ineficiencias. Estas técnicas nos permitirán definir computaciones iterativas de complejidad lineal en tiempo, siempre que sea posible. Veremos dos técnicas útiles: transformaciones de estado y listas de diferencias.

3.4.2.4. Convirtiendo computaciones recursivas en iterativas

Miremos cómo convertir computaciones recursivas en iterativas. En lugar de usar `Reverse` como ejemplo, usaremos una función más simple para calcular la longitud (el tamaño) de una lista:

```
fun {Length xs}
  case xs of nil then 0
    [] _|xr then 1+{Length xr}
  end
end
```

Note que la función `SumList` tiene la misma estructura. Esta función es de complejidad lineal en tiempo, pero el tamaño de la pila es proporcional a la profundidad de la recursión, la cual es igual a la longitud de `xs`. Por Qué pasa esto? Porque la suma `1+{Length xr}` es posterior a la invocación recursiva. La invocación recursiva no es lo último, por lo tanto el ambiente de la función no puede ser recuperado antes de ésta.

¿Cómo podemos calcular la longitud de la lista con una computación iterativa en la cual el tamaño de la pila esté acotado por una constante? Para hacer esto, tenemos que formular el problema como una secuencia de transformaciones de estados. Es decir, empezamos con un estado S_0 y lo transformamos sucesivamente en los estados S_1, S_2, \dots , hasta alcanzar un estado final S_{final} , el cual contiene la respuesta. Para calcular la longitud de la lista, podemos tomar, como estado, la longitud i de la parte de la lista que ya ha sido recorrida. Realmente, esta es sólo una parte del estado. El resto del estado es la parte `ys` de la lista que no ha sido recorrida. El estado completo S_i es entonces la pareja (i, ys) . El caso intermedio general para el estado S_i es el siguiente (donde la lista completa `xs` es $[e_1 e_2 \dots e_n]$):

$$\overbrace{e_1 e_2 \cdots e_i}^{\text{xs}} \underbrace{e_{i+1} \cdots e_n}_{\text{ys}}$$

En cada invocación recursiva, i será incrementada en 1 y `ys` reducida en un elemento. Esto nos lleva a la función:

```
fun {IterLength I ys}
  case ys
  of nil then I
    [] _|yr then {IterLength I+1 yr}
  end
end
```

Su tipo es `<fun {$ <Ent> <Lista>} : <Ent>`. Note la diferencia con la definición anterior. En este caso la suma `I+1` se realiza antes que la invocación recursiva a `IterLength`, la cual es la última invocación. Hemos definido, entonces, una computación iterativa.

En la invocación `{IterLength I ys}`, el valor inicial de `I` es 0. Podemos ocultar esta inicialización definiendo `IterLength` como un procedimiento local a `Length`. La definición final de `Length` es por tanto:

```

local
  fun {IterLength I Ys}
    case Ys
      of nil then I
      [] _|Yr then {IterLength I+1 Yr}
    end
  end
in
  fun {Length Xs}
    {IterLength 0 Xs}
  end
end

```

Esto define una computación iterativa para calcular la longitud de una lista. Note que definimos `IterLength` por fuera de `Length`. Esto evita la creación de un nuevo valor de tipo procedimiento cada vez que `Length` sea invocado. No hay ninguna ventaja en definir `IterLength` dentro de `Length`, pues `IterLength` no utiliza el argumento `Xs` de `Length`.

Podemos usar la misma técnica que usamos para `Length`, sobre `Reverse`. El estado es diferente. En el caso de `Reverse`, el estado contiene el inverso de la lista que ya ha sido recorrida en lugar de su longitud. La actualización del estado es fácil: simplemente colocamos un nuevo elemento en la cabeza de la lista. El estado inicial es `nil`. Esto nos lleva a la versión siguiente de `Reverse`:

```

local
  fun {IterReverse Rs Ys}
    case Ys
      of nil then Rs
      [] Y|Yr then {IterReverse Y|Rs Yr}
    end
  end
in
  fun {Reverse Xs}
    {IterReverse nil Xs}
  end
end

```

Esta versión de `Reverse` es de complejidad lineal en tiempo y define una computación iterativa.

3.4.2.5. Corrección con invariantes de estado

Probemos que `IterLength` es correcta. Usaremos una técnica general que funciona igualmente bien para `IterReverse` y para otras computaciones iterativas. La idea es definir una propiedad $P(S_i)$, sobre los estados, que podamos probar que siempre se cumple, i.e., un invariante de estado. Si P se escoge apropiadamente, entonces la corrección de la computación es una consecuencia lógica de $P(S_{\text{final}})$. Para el caso de `IterLength` definimos P como sigue:

$$P((i, \text{ys})) \equiv (\text{longitud(xs)} = i + \text{longitud(ys)})$$

donde $\text{longitud}(\mathbf{L})$ representa la longitud de la lista \mathbf{L} . Note la forma en que se combinan i y \mathbf{ys} (de alguna manera se captura lo realizado y lo que falta por hacer) para proponer el invariante de estado. Para probar que P efectivamente es un invariante de estado usamos inducción:

- Primero probamos $P(S_0)$. Esto se deduce directamente de la definición de $S_0 = (0, \mathbf{xs})$.
- Suponiendo $P(S_i)$ y que S_i no es el estado final, probemos $P(S_{i+1})$. Esto se deduce de la semántica de la declaración **case** y de la invocación a una función. Suponga que $S_i = (i, \mathbf{ys})$. Como S_i no es el estado final, entonces \mathbf{ys} es de longitud diferente de cero. Por la semántica, $i+1$ añade 1 a i y la declaración **case** elimina un elemento de \mathbf{ys} . El nuevo estado S_{i+1} es la pareja $(i+1, \mathbf{yr})$ donde $\text{longitud}(\mathbf{yr}) = \text{longitud}(\mathbf{ys}) - 1$ y, por tanto, $(i+1) + \text{longitud}(\mathbf{yr}) = i + \text{longitud}(\mathbf{ys}) = \text{longitud}(\mathbf{xs})$. En consecuencia, $P(S_{i+1})$ se cumple.

Como \mathbf{ys} se reduce en un elemento en cada invocación, finalmente llegaremos al estado final $S_{\text{final}} = (i, \mathbf{nil})$, y la función devuelve i . Como $\text{longitud}(\mathbf{nil}) = 0$, y $P(S_{\text{final}})$ se cumple, entonces $i = \text{length}(\mathbf{xs})$.

La etapa difícil de la prueba es escoger la propiedad P , la cual tiene que satisfacer dos restricciones. La primera, P tiene que combinar los argumentos de la computación iterativa de manera que el resultado no cambie a medida que la computación progresá. La segunda, P debe ser suficientemente fuerte para que la corrección del resultado sea una consecuencia de saber que $P(S_{\text{final}})$ se cumple. Una manera de imaginarse una propiedad P consiste en ejecutar a mano el programa para unos casos sencillos, y a partir de ellos delinear el caso intermedio general.

3.4.2.6. Construyendo programas según el tipo de los datos

Todos los ejemplos anteriores de funciones sobre listas tienen una propiedad curiosa: todos tienen un argumento de tipo $\langle \text{Lista } T \rangle$, el cual se definió así:

```
 $\langle \text{Lista } T \rangle ::= \text{nil}$ 
|  $T^* | [ \langle \text{Lista } T \rangle ]$ 
```

y todos usan una declaración **case** de la forma:

```
case xs
of nil then <expr> % Caso base
[ ] x|xr then <expr> % Invocación recursiva
end
```

¿Qué está pasando aquí? La estructura recursiva de las funciones sobre listas sigue exactamente la estructura recursiva de la definición del tipo. Esto es cierto en la mayoría de los casos de funciones sobre listas.

Podemos usar esta propiedad para ayudarnos a escribir funciones sobre listas. Esto puede ser de gran ayuda cuando las definiciones de tipos se vuelven complicadas. Por ejemplo, escribamos una función que cuenta los elementos de una lista anidada. Una lista anidada es una lista en la cual cada elemento de la lista puede, a su vez,

Técnicas de programación declarativa

ser una lista, e.g., `[[1 2] 4 nil [[5] 10]]`. Definimos el tipo $\langle \text{ListaAnidada } T \rangle$ así:

```
 $\langle \text{ListaAnidada } T \rangle ::= \text{nil}$ 
|  $\langle \text{ListaAnidada } T \rangle \cdot | \cdot \langle \text{ListaAnidada } T \rangle$ 
|  $T \cdot | \cdot \langle \text{ListaAnidada } T \rangle$ 
```

Para evitar ambigüedades, tenemos que agregar una condición sobre T , a saber, que T , no es ni `nil` ni un `cons`. Ahora escribamos la función $\{\text{LengthL}\}$: $\langle \text{Ent} \rangle$ la cual cuenta el número de elementos en una lista anidada. Siguiendo la definición del tipo llegamos al siguiente esqueleto:

```
fun {LengthL xs}
case xs
of nil then (expr)
[] x|xr andthen {IsList x} then
  (expr) % Invocaciones recursivas para x y xr
[] x|xr then
  (expr) % Invocación recursiva para xr
end
end
```

(El tercer caso no tiene que mencionar $\{\text{Not }\{\text{IsList } x\}\}$ pues se deduce de la negación del segundo caso.) Aquí $\{\text{IsList } x\}$ es una función que comprueba si x es `nil` o un `cons`:

```
fun {IsCons x} case x of _ | _ then true else false end end
fun {IsList x} x==nil orelse {IsCons x} end
```

Rellenando el esqueleto llegamos a la función siguiente:

```
fun {LengthL xs}
case xs
of nil then 0
[] x|xr andthen {IsList x} then
  {LengthL x}+{LengthL xr}
[] x|xr then
  1+{LengthL xr}
end
end
```

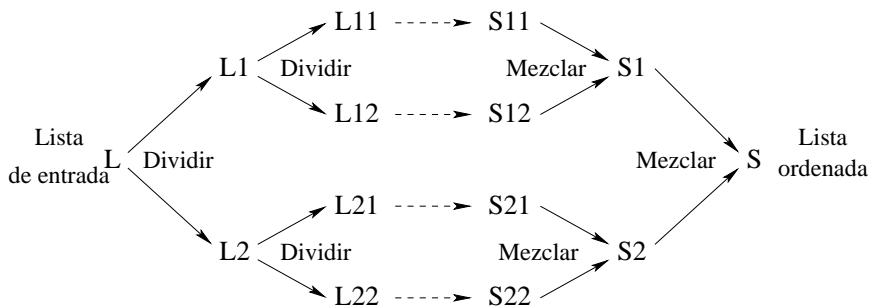
Dos ejemplos de invocaciones son:

```
x=[[1 2] 4 nil [[5] 10]]
{Browse {LengthL x}}
{Browse {LengthL [x x]}}
```

¿Qué aparece en el browser con estas invocaciones?

Si utilizamos una definición diferente del tipo para las listas anidadas, la función de longitud también será diferente. Por ejemplo, definamos el tipo $\langle \text{ListaAnidada2 } T \rangle$ así:

```
 $\langle \text{ListaAnidada2 } T \rangle ::= \text{nil}$ 
|  $\langle \text{ListaAnidada2 } T \rangle \cdot | \cdot \langle \text{ListaAnidada2 } T \rangle$ 
|  $T$ 
```

Figura 3.9: Ordenando con *mergesort*.

Nuevamente, tenemos que agregar la condición que T , no es ni `nil` ni un `cons`. ¡Note la sutil diferencia entre $\langle\text{ListaAnidada } T\rangle$ y $\langle\text{ListaAnidada2 } T\rangle$!

Siguiendo la definición del tipo $\langle\text{ListaAnidada2 } T\rangle$ nos lleva a una función `LengthL2` diferente y más sencilla:

```

fun {LengthL2 xs}
  case xs
  of nil then 0
  [] x|xr then
    {LengthL2 x}+{LengthL2 xr}
  else 1 end
end
  
```

¿Cuál es la diferencia entre `LengthL` y `LengthL2`? Podemos deducirla comparando las definiciones de los tipos $\langle\text{ListaAnidada } T\rangle$ y $\langle\text{ListaAnidada2 } T\rangle$. Una $\langle\text{ListaAnidada } T\rangle$ tienen que ser siempre una lista, mientras que una $\langle\text{ListaAnidada2 } T\rangle$ puede llegar a ser de tipo T . Por ello, la invocación `{LengthL2 foo}` es legal (devuelve 1), mientras que `{LengthL foo}` es ilegal (lanza una excepción). Desde el punto de vista del comportamiento deseado (la entrada debe ser una lista), pensamos que es razonable considerar `LengthL2` errónea y `LengthL` correcta.

Hay una lección importante para aprender en este caso. La definición de un tipo recursivo debe ser realizada antes de escribir una función que la use. De lo contrario es fácil engañarse con una función aparentemente sencilla que es incorrecta. Esto es cierto aún en los lenguajes funcionales que realizan inferencia de tipos, tales como Standard ML y Haskell. La inferencia de tipos puede verificar que un tipo recursivo sea usado correctamente, pero el diseño del tipo recursivo sigue siendo responsabilidad del programador.

3.4.2.7. Ordenando con mergesort

Definimos una función que toma una lista de números o átomos y devuelve una nueva lista con los mismos elementos ordenados ascendente. Se utiliza el operador de comparación `<`, de manera que todos los elementos deben ser del

mismo tipo (flotantes, enteros, o átomos). Utilizamos el algoritmo *mergesort*, el cual es eficiente y puede ser programado fácilmente en el modelo declarativo. El algoritmo *mergesort* se basa en una estrategia sencilla llamada dividir y conquistar:

- Dividir la lista en dos listas de menor tamaño, aproximadamente de la misma longitud.
- Utilizar el *mergesort* recursivamente para ordenar las dos listas de menor tamaño.
- Mezclar las dos listas ordenadas para obtener el resultado final.

En la figura 3.9 se muestra la estructura recursiva. El *mergesort* es eficiente porque las operaciones de dividir y mezclar son ambas computaciones iterativas de complejidad lineal en tiempo. Definimos primero las operaciones de mezclar y dividir y luego si el *mergesort*:

```
fun {Mezclar Xs Ys}
    case Xs # Ys
    of nil # Ys then Ys
    [] Xs # nil then Xs
    [] (X|Xr) # (Y|Yr) then
        if X<Y then X|(Mezclar Xr Ys)
        else Y|(Mezclar Xs Yr)
    end
end
```

El tipo es `fun { $ <Lista T> <Lista T>} : <Lista T>`, donde `T` es `<Ent>`, `<flot>`, o `<Atom>`. Definimos dividir como un procedimiento porque tiene dos salidas. También habría podido ser definido como una función que devuelve una pareja como única salida.

```
proc {Dividir Xs ?Ys ?Zs}
    case Xs
    of nil then Ys=nil Zs=nil
    [] [X] then Ys=[X] Zs=nil
    [] X1|X2|Xr then Yr Zr in
        Ys=X1|Yr
        Zs=X2|Zr
        {Dividir Xr Yr Zr}
    end
end
```

El tipo es `proc { $ <Lista T> <Lista T> <Lista T> } : <Lista T>`. La definición de *mergesort* es:

```
fun {MergeSort Xs}
    case Xs
    of nil then nil
    [] [X] then [X]
    else Ys Zs in
        {Dividir Xs Ys Zs}
        {Mezclar {MergeSort Ys} {MergeSort Zs}}
    end
end
```

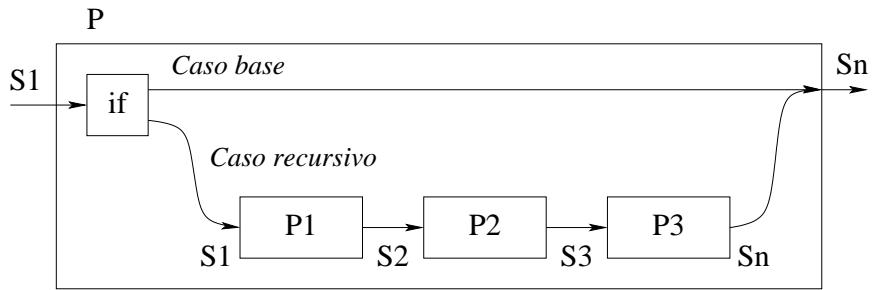


Figura 3.10: Flujo de control con filtración de estados.

Su tipo es `<fun { $ <Lista T>} : <Lista T>` con la misma restricción que en `Mezclar` se tenía sobre `T`. El proceso de división de la lista de entrada se repite hasta llegar a listas de longitud cero o uno, las cuales pueden ser ordenadas inmediatamente.

3.4.3. Acumuladores

Hemos visto cómo programar funciones sencillas sobre listas y cómo volverlas iterativas. La programación declarativa en la realidad se hace, normalmente, de una manera diferente, a saber, escribiendo funciones que son iterativas desde el comienzo. La idea es enviar el estado hacia adelante todo el tiempo y nunca dejar un cálculo para después de la invocación recursiva. Un estado `S` se representa agregando un par de argumentos, `S1` y `Sn`, a cada procedimiento. Este par se llama un *acumulador*. `S1` representa el estado de entrada y `Sn` representa el estado de salida. Cada definición de procedimiento se escribe entonces en un estilo como el siguiente:

```

proc {P X S1 ?Sn}
  if {CasoBase X} then Sn=S1
  else
    {P1 S1 S2}
    {P2 S2 S3}
    {P3 S3 Sn}
  end
end
  
```

El caso base no realiza ningún cálculo, de manera que el estado de salida es el mismo estado de entrada (`Sn=S1`). El caso recursivo filtra el estado a través de cada invocación recursiva (`P1`, `P2`, y `P3`) y finalmente lo regresa a `P`. En la figura 3.10 se presenta una ilustración. Cada flecha representa una variable de estado. El valor del estado se toma en la cola de la flecha y se pasa a su cabeza. Por filtración del estado entendemos que cada salida de un procedimiento es la entrada para el siguiente procedimiento. La técnica de filtrar un estado a través de invocaciones anidadas a procedimientos se llama programación por acumuladores.

La programación por acumuladores es utilizada en las funciones `IterLength` y `IterReverse` que vimos antes. En estas funciones la estructura de acumuladores no es tan evidente, debido a que se definieron como funciones. Lo que está pasando es que los estados de entrada son pasados a la función y el estado de salida es lo que devuelve la función.

Multiples acumuladores

Considere el procedimiento siguiente, el cual recibe una expresión que contiene identificadores, enteros, y operaciones de adición (utilizando un registro con etiqueta `suma`), y calcula dos resultados: traduce la expresión a código de máquina para una máquina sencilla de pila y calcula el número de instrucciones del código traducido.

```
proc {CodifExpr E C1 ?Cn S1 ?Sn}
  case E
    of suma(A B) then C2 C3 S2 S3 in
      C2=sume|C1
      S2=S1+1
      {CodifExpr B C2 C3 S2 S3}
      {CodifExpr A C3 Cn S3 Sn}
    [] I then
      Cn=coloque(I)|C1
      Sn=S1+1
    end
  end
```

Este procedimiento tiene dos acumuladores: uno para construir la lista de instrucciones de máquina (`C1` y `Cn`) y otro para guardar el número de instrucciones (`S1` y `Sn`). Presentamos un ejemplo sencillo de ejecución:

```
declare Código Tamaño in
{CodifExpr suma(suma(a 3) b) nil Código 0 Tamaño}
{Browse Tamaño#Código}
```

Esto muestra en el browser

```
5#[coloque(a) coloque(3) sume coloque(b) sume]
```

Programas más complicados requieren normalmente más acumuladores. Cuando hemos escrito programas declarativos grandes, hemos usado típicamente alrededor de una media docena de acumuladores simultáneamente. El compilador de Aquarius Prolog se escribió en este estilo [177, 175]. Algunos de sus procedimientos tienen hasta doce acumuladores. ¡Esto significa veinticuatro argumentos! Esto es difícil de lograr sin ayuda mecánica. Nosotros usamos un preprocesador DCG extendido⁷ que toma las declaraciones de acumuladores y agrega los argumentos automáticamente [84].

7. DCG (Definite Clause Grammar) es una notación gramatical utilizada para ocultar el filtrado explícito de los acumuladores.

Nosotros no programamos más en ese estilo; nos parece que la programación con estado explícito es más sencilla y eficiente (ver capítulo 6). Es razonable utilizar unos pocos acumuladores en un programa declarativo; es muy raro que un programa declarativo no necesite de unos pocos. Por otro lado, utilizar muchos es una señal de que probablemente quedaría mejor escrito con estado explícito.

Mergesort con un acumulador

En la definición anterior de *mergesort*, invocamos primero la función `Dividir` para dividir la lista de entrada en dos mitades. Hay una manera más sencilla de realizar el *mergesort*, usando un acumulador. El parámetro representa la parte de la lista que no ha sido ordenada aún. La especificación de `MergesortAcc` es:

- `S#L2={MergeSortAcc L1 N}` toma una lista de entrada `L1` y un entero `N`, y devuelve dos resultados: `S`, la lista ordenada de los primeros `N` elementos de `L1`, y `L2`, los elementos restantes de `L1`. Los dos resultados se colocan juntos en una pareja con el constructor de tuplas `#`.

El acumulador lo definen `L1` y `L2`. Esto lleva a la definición siguiente:

```
fun {MergeSort Xs}
  fun {MergeSortAcc L1 N}
    if N==0 then
      nil # L1
    elseif N==1 then
      [L1.1] # L1.2
    elseif N>1 then
      NL=N div 2
      NR=N-NL
      Ys # L2 = {MergeSortAcc L1 NL}
      Zs # L3 = {MergeSortAcc L2 NR}
      in
        {Mezclar Ys Zs} # L3
      end
    end
  in
    {MergeSortAcc Xs {Length Xs}}.1
  end
```

La función `Mezclar` no cambia. Note que este mergesort realiza una división diferente al anterior. En esta versión, la división separa la primera mitad de la lista de entrada de la segunda mitad. En la versión anterior, la división separaba la lista de entrada en la lista de los elementos que estaban en posición par y la lista de los elementos que estaban en posición impar.

Esta versión tiene la misma complejidad en tiempo que la versión previa. Utiliza eso sí menos memoria porque no crea dos listas divididas. Estas listas se definen implícitamente por la combinación del parámetro de acumulación y el número de elementos.

3.4.4. Listas de diferencias

Una lista de diferencias es un par de listas, cada una de las cuales puede tener una cola no-ligada. Las dos listas tienen una relación especial entre ellas: debe ser posible calcular la segunda lista a partir de la primera, solamente por eliminación de cero o más elementos del frente de la lista. Algunos ejemplos son:

```
x#x          % Representa la lista vacía
nil#nil      % idem
[a]#[a]       % idem
(a|b|c|x)#x % Representa [a b c]
(a|b|c|d|x)#[d|x] % idem
[a b c d]#[d] % idem
```

Una lista de diferencias es una representación de una lista estándar. Algunas veces hablaremos de la lista de diferencias como una estructura de datos por sí misma, y otras veces como la representación de una lista estándar. Sea cuidadoso para no confundir estos dos puntos de vista. La lista de diferencias `[a b c d]#[d]` puede contener las listas `[a b c d]` y `[d]`, pero no representa ninguna de estas, sino la lista `[a b c]`.

Las listas de diferencias son un caso especial de estructuras de diferencias. Una estructura de diferencias es un par de valores parciales tal que el segundo está embebido en el primero. La estructura de diferencias representa el valor resultante de eliminar la segunda estructura de la primera. La utilización de estructuras de diferencias facilita la construcción de computaciones iterativas sobre muchos tipos recursivos de datos, e.g., listas o árboles. Las listas y estructuras de diferencias son casos especiales de acumuladores en el cual uno de los argumentos acumulador puede ser una variable no-ligada.

La ventaja de usar listas de diferencias es que cuando la segunda lista es una variable no-ligada, se puede concatenar esta lista de diferencias con otra en tiempo constante. Para concatenar las listas de diferencias `(a|b|c|x)#x` y `(d|e|f|y)#[y]`, sólo hay que ligar `x` con `(d|e|f|y)`. Esto crea la lista de diferencias `(a|b|c|d|e|f|y)#[y]`. Acabamos de concatenar las listas `[a b c]` y `[d e f]` con una única operación de ligadura. Esta es la función que concatena dos listas de diferencias:

```
fun {AppendD D1 D2}
  S1#E1=D1
  S2#E2=D2
  in
    E1=S2
    S1#E2
  end
```

Se puede usar como una concatenación de listas:

```
local x y in {Browse {AppendD (1|2|3|x)#x (4|5|y)#[y]}} end
```

Esto muestra en el browser `(1|2|3|4|5|y)#[y]`. La función estándar de concatenación de listas, definida así:

```
fun {Append L1 L2}
    case L1
        of X|T then X|{Append T L2}
        [] nil then L2
    end
end
```

itera sobre su primer argumento, y por tanto toma tiempo proporcional al tamaño del primer argumento. La concatenación de listas de diferencias es mucho más eficiente: toma tiempo constante.

La limitación al usar listas de diferencias es que no pueden ser concatenadas sino una sola vez. Esto hace que las listas de diferencias sólo pueden ser usadas en circunstancias especiales. Por ejemplo, ellas son la manera natural de escribir programas que construyen grandes listas en términos de muchas listas pequeñas que deben ser concatenadas entre sí.

Las listas de diferencias como se han definido aquí tienen su origen en Prolog y programación lógica [163]. Ellas son la base de muchas técnicas de programación avanzada en Prolog. Como concepto, una lista de diferencias deambula entre el concepto de valor y el concepto de estado. Tiene las buenas propiedades de un valor (los programas que las usan son declarativos), pero también tienen algo del poder del estado debido a que pueden ser concatenadas una sola vez en tiempo constante.

Aplanando una lista anidada

Considere el problema de aplanar una lista anidada, i.e., calcular una lista que tenga los mismos elementos de la lista anidada pero que no estén anidados. Primero mostraremos una solución usando listas y luego mostraremos que es posible obtener una solución mucho mejor usando listas de diferencias. Para escribir la primera solución, razonemos inductivamente en la misma forma que lo hicimos con la función LengthL, basados en el tipo `{ListaAnidada}` definido previamente:

- Aplanar `nil` da `nil`.
- Aplanar `x|xr` si `x` es una lista anidada, da `z` donde
 - Aplanar `x` da `y`,
 - Aplanar `xr` da `yr`, y
 - `z` es la concatenación de `y` y `yr`.
- Aplanar `x|xr` si `x` no es una lista, da `z` donde
 - Aplanar `xr` da `yr`, y
 - `z` es `x|yr`.

Siguiendo este razonamiento llegamos a la definición siguiente:

```
fun {Aplanar Xs}
  case Xs
  of nil then nil
    [] X|Xr andthen {IsList X} then
      {Append {Aplanar X} {Aplanar Xr}}
    [] X|Xr then
      X|{Aplanar Xr}
    end
  end
```

Invocando:

```
{Browse {Aplanar [[a b] [[c] [d]] nil [e [f]]]}}
```

se muestra [a b c d e f] en el browser. Este programa es muy ineficiente pues realiza muchas operaciones de concatenación (ver ejercicios, sección 3.10). Ahora, razonemos en la misma forma, pero con listas de diferencias en lugar de listas estándar:

- Aplanar nil da x#x (lista de diferencias vacía).
- Aplanar x|xr si x es una lista anidada, da y1#y4 donde
 - Aplanar x da y1#y2,
 - Aplanar xr da y3#y4, y
 - y2 y y3 se igualan para concatenar las listas de diferencias.
- Aplanar x|xr si x no es una lista, da (x|y1)#y2 donde
 - Aplanar xr da y1#y2.

Podemos escribir el segundo caso de la siguiente manera:

- Aplanar x|xr si x es una lista anidada, da y1#y4 donde
 - Aplanar x da y1#y2 y
 - Aplanar xr da y2#y4.

Esto lleva al programa siguiente:

```
fun {Aplanar Xs}
  proc {AplanarD Xs ?Ds}
    case Xs
    of nil then Y in Ds=Y#Y
    [] X|Xr andthen {IsList X} then Y1 Y2 Y4 in
      Ds=Y1#Y4
      {AplanarD X Y1#Y2}
      {AplanarD Xr Y2#Y4}
    [] X|Xr then Y1 Y2 in
      Ds=(X|Y1)#Y2
      {AplanarD Xr Y1#Y2}
    end
  end Ys
in {AplanarD Xs Ys#nil} Ys end
```

Este programa es eficiente: realiza una única operación cons para cada elemento de la entrada que no es una lista . Convertimos la lista de diferencias devuelta por

AplanarD en una lista regular ligando su segundo argumento a nil. Escribimos AplanarD como un procedimiento pues su salida hace parte de su último argumento, pero no es el argumento completo (ver sección 2.6.2). Es un estilo común escribir una lista de diferencias en dos argumentos:

```
fun {Aplanar Xs}
  proc {AplanarD Xs ?S E}
    case Xs
      of nil then S=E
      [] X|Xr andthen {IsList X} then Y2 in
        {AplanarD X S Y2}
        {AplanarD Xr Y2 E}
      [] X|Xr then Y1 in
        S=X|Y1
        {AplanarD Xr Y1 E}
    end
  end Ys
  in {AplanarD Xs Ys nil} Ys end
```

Como una simplificación adicional, podemos escribir AplanarD como una función. Para hacerlo, usaremos el hecho que S es la salida:

```
fun {Aplanar Xs}
  fun {AplanarD Xs E}
    case Xs
      of nil then E
      [] X|Xr andthen {IsList X} then
        {AplanarD X {AplanarD Xr E}}
      [] X|Xr then
        X|{AplanarD Xr E}
    end
  end
  in {AplanarD Xs nil} end
```

¿Cuál es el papel de E? Este tiene el “resto” de la salida, i.e., cuando la invocación a AplanarD agota su propia contribución a la salida.

Invirtiendo una lista

Miremos de nuevo el programa ingenuo, de la última sección, para invertir el orden de los elementos de una lista. El problema con este programa es que utiliza una función de concatenación muy costosa. ¿Será más eficiente con la concatenación en tiempo constante de las listas de diferencias? Hagamos la versión ingeniosa de inversión con listas de diferencias:

- Invertir nil da x#x (lista de diferencias vacía).
- Invertir x|xs da z, donde
 - invertir xs da y1#y2 y
 - z es la concatenación de y1#y2 y (x|y)#y.

Realizando la concatenación, el último caso se reescribe:

- Invertir $x|xs$ da $y1#y$, donde
 - invertir xs da $y1#y2$ y
 - $y2$ y $x|y$ se igualan.

Es perfectamente admisible realizar la igualdad antes que la invocación recursiva de la inversión (¿por qué?). Esto nos lleva a:

- Invertir $x|xs$ da $y1#y$, donde
 - invertir xs da $y1#(x|y)$.

Esta es la definición final:

```
fun {Reverse Xs}
  proc {ReverseD Xs ?Y1 Y}
    case Xs
      of nil then Y1=Y
      [] X|Xr then {ReverseD Xr Y1 X|Y}
    end
  end Y1
  in {ReverseD Xs Y1 nil} Y1 end
```

Observe cuidadosamente y verá que esta solución es prácticamente la misma solución iterativa presentada en la última sección. La única diferencia entre `IterReverse` y `ReverseD` es el orden de los argumentos: la salida de `IterReverse` es el segundo argumento de `ReverseD`. Entonces, ¿cuál es la ventaja de utilizar listas de diferencias? Con ellas, llegamos a `ReverseD` sin pensar, mientras que para llegar a `IterReverse` tuvimos que adivinar un estado intermedio que pueda ser actualizado.

3.4.5. Colas

Una cola es una secuencia de elementos con una operación de inserción y otra de eliminación de elementos. La operación de inserción agrega un elemento al final de la cola y la operación de eliminación elimina el elemento del principio. Decimos que la cola tiene un comportamiento FIFO (el primero en entrar es el primero en salir).⁸ Investigamos cómo programar colas en el Modelo declarativo.

Una cola ingenua

Una implementación obvia de colas es usando listas. Si L representa el contenido de la cola, entonces insertar x produce la nueva cola $x|L$ y eliminar x se realiza invocando `{QuitarElUltimo L x L1}`, operación que liga x al último elemento y devuelve la nueva cola en $L1$. `QuitarElUltimo` devuelve el último elemento de L en x y todos los elementos menos el último en $L1$. `QuitarElUltimo` se define así:

8. Nota del traductor: del inglés *first-in, first-out*.

```
proc {QuitarElUltimo L ?X ?L1}
  case L
    of [Y] then X=Y L1=nil
    [] Y|L2 then L3 in
      L1=Y|L3
      {QuitarElUltimo L2 X L3}
    end
  end
```

El problema con esta implementación es que `QuitarElUltimo` es lenta: toma tiempo proporcional al número de elementos en la cola. Nos gustaría, por el contrario, que ambas operaciones, insertar y eliminar, tuvieran una complejidad, en tiempo, constante. Es decir, que realizar una de esas operaciones en cualquier implementación y máquinas dadas, tome menos que un número constante de segundos. El valor de la constante dependerá de la implementación y de la máquina. Que podamos o no lograr el objetivo de complejidad en tiempo constante depende de la expresividad del modelo de computación:

- En un lenguaje estricto de programación funcional, i.e., el modelo declarativo sin variables de flujo de datos (ver sección 2.8.1), no se puede lograr. Lo mejor que se puede hacer es llegar a complejidad amortizada en tiempo constante [129]. Es decir, cualquier secuencia de n operaciones de inserción y eliminación tomará un tiempo total proporcional a alguna constante multiplicada por n . Sin embargo, el costo en tiempo de cada operación individual puede no ser constante.
- En el modelo declarativo, el cual extiende el modelo funcional estricto con variables de flujo de datos, podemos lograr la complejidad en tiempo constante.

Mostraremos cómo definir ambas soluciones. En ambas definiciones, cada operación toma una cola como entrada y devuelve una cola nueva como salida. Apenas una cola es usada como entrada para una operación, ésta no puede volver a ser usada como entrada para otra operación. En otras palabras, existe una sola versión de la cola para ser usada en todo momento. Decimos que la cola es efímera.⁹ Cada versión existe desde el momento en que se crea hasta el momento en que no puede volver a ser usada.

Cola efímera en tiempo constante amortizado

Presentamos la definición de la cola cuyas operaciones de inserción y eliminación tienen límites constantes en tiempo amortizado. La definición viene de [129]:

9. Las colas implementadas con estado explícito (ver capítulos 6 y 7) también son, normalmente, efímeras.

```
fun {ColaNueva} q(nil nil) end

fun {CompCola Q}
  case Q of q(nil R) then q({Reverse R} nil) else Q end
end

fun {InsCola Q X}
  case Q of q(F R) then {CompCola q(F X|R)} end
end

fun {ElimDeCola Q X}
  case Q of q(F R) then F1 in F=X|F1 {CompCola q(F1 R)} end
end

fun {EsVaciaCola Q}
  case Q of q(F R) then F==nil end
end
```

La pareja $q(F\ R)$ representa la cola, donde F y R son listas. F representa el frente de la cola y R representa la parte de atrás de la cola en orden inverso. En cualquier momento, el contenido de la cola está dado por $\{\text{Append } F\ \{\text{Reverse } R\}\}$. Un elemento se puede insertar añadiéndolo al frente de R (es decir de último de la cola). Borrar un elemento (el primero de la cola) significa eliminarlo del frente de F . Por ejemplo, si $F=[a\ b]$ y $R=[d\ c]$, entonces borrar el primer elemento de la cola devuelve a y deja $F=[b]$. Insertar el elemento e hace $R=[e\ d\ c]$. Ambas operaciones se realizan en tiempo constante.

Para que esta representación funcione, cada elemento de R , tarde o temprano, tiene que ser llevado a F . ¿En qué momento se debe hacer esto? Hacerlo elemento por elemento es ineficiente, pues significa reemplazar F por $\{\text{Append } F\ \{\text{Reverse } R\}\}$ cada vez, lo cual toma por lo menos tiempo proporcional al tamaño de F . El truco es hacerlo sólo ocasionalmente. Lo hacemos cuando F se vuelve vacío, de manera que F no es nil si y sólo si la cola no es vacía. Este invariante lo mantiene la función `CompCola`, la cual mueve el contenido de R a F cuando F es nil .

La función `CompCola` realiza una operación de inversión de la lista R . La inversión toma tiempo proporcional al tamaño de R , i.e., al número de elementos que hay que invertir. Cada elemento que entra a la cola se pasa exáctamente una vez de R a F . Repartiendo el tiempo de ejecución de la inversión entre todos los elementos da un tiempo constante por elemento. Es por esto que se dice que el tiempo es constante amortizado.

Cola efímera en tiempo constante en el peor caso

Podemos usar listas de diferencias para implementar colas cuyas operaciones de inserción y eliminación tomen tiempos de ejecución constantes en el peor de los casos. Usamos una lista de diferencias que termina en una variable de flujo de datos no-ligada. Esto nos permite insertar elementos en tiempo constante simplemente ligando la variable de flujo de datos. La definición es:

```

fun {ColaNueva} X in q(0 X X) end

fun {InsCola Q X}
  case Q of q(N S E) then E1 in E=X|E1 q(N+1 S E1) end
end

fun {ElimDeCola Q X}
  case Q of q(N S E) then S1 in S=X|S1 q(N-1 S1 E) end
end

fun {EsVacíaCola Q}
  case Q of q(N S E) then N==0 end
end

```

La tripleta $q(N S E)$ representa la cola. En cualquier momento, el contenido de la cola está representado por la lista de diferencias $S \# E$. N es el número de elementos en la cola. ¿Por qué necesitamos N ? Sin él, no sabríamos cuántos elementos hay en la cola.

Ejemplo de uso

El siguiente ejemplo funciona tanto con la definición en tiempo constante amortizado como con la de tiempo constante en el peor caso:

```

declare Q1 Q2 Q3 Q4 Q5 Q6 Q7 in
Q1={ColaNueva}
Q2={InsCola Q1 pedro}
Q3={InsCola Q2 pablo}
local X in Q4={ElimDeCola Q3 X} {Browse X} end
Q5={InsCola Q4 maría}
local X in Q6={ElimDeCola Q5 X} {Browse X} end
local X in Q7={ElimDeCola Q6 X} {Browse X} end

```

Aquí se insertan tres elementos y luego se eliminan. Cada elemento se inserta antes de ser eliminado. Ahora miremos qué se puede hacer con una definición, que no se pueda hacer con la otra.

Con la definición de tiempo constante en el peor caso podemos borrar un elemento antes de que sea insertado. Esto parece sorprendente, pero es perfectamente natural. Realizar una tal eliminación devolverá una variable no ligada que será ligada al siguiente elemento insertado. Entonces, las últimas cuatro invocaciones del ejemplo pueden ser cambiadas por:

```

local X in Q4={ElimDeCola Q3 X} {Browse X} end
local X in Q5={ElimDeCola Q4 X} {Browse X} end
local X in Q6={ElimDeCola Q5 X} {Browse X} end
Q7={InsCola Q6 maría}

```

Esto funciona gracias a que la operación de ligadura de variables de flujo de datos, la cual se usa tanto para insertar como para eliminar elementos, es simétrica.

Con la definición de tiempo constante amortizado, el mantener simultáneamente múltiples versiones de la cola da resultados correctos, aunque los límites de com-

plejidad amortizada en tiempo constante ya no se cumplen.¹⁰ Este es un ejemplo con dos versiones:

```
declare Q1 Q2 Q3 Q4 Q5 Q6 in
Q1={ColaNueva}
Q2={InsCola Q1 pedro}
Q3={InsCola Q2 pablo}
Q4={InsCola Q2 maría}
local X in Q5={ElimDeCola Q3 X} {Browse X} end
local X in Q6={ElimDeCola Q4 X} {Browse X} end
```

Tanto Q3 como Q4 se calculan a partir de su ancestro común Q2. Q3 contiene pedro y pablo. Q4 contiene pedro y maría. ¿Qué muestran las dos invocaciones a `Browse`?

Colas persistentes

Las dos definiciones anteriores crean colas efímeras. ¿Qué podemos hacer si necesitamos usar múltiples versiones de una cola y aún requerimos tiempo constante de ejecución? Una cola que soporta múltiples versiones simultáneas se dice persistente.¹¹ Algunas aplicaciones requieren de colas persistentes. Por ejemplo, si durante un cálculo le pasamos un valor de tipo cola a otra rutina:

```
...
{AlgúnProc Qa}
Qb={InsCola Qa x}
Qc={InsCola Qb y}
...
```

Suponemos que `AlgúnProc` puede realizar operaciones sobre colas, pero que quien lo invoca no desea ver sus efectos. Por eso podríamos tener dos versiones de una cola. ¿Podemos escribir programas que manejen colas que conserven los límites de tiempo, para este caso? Se puede hacer, si extendemos el modelo declarativo con ejecución perezosa. Entonces, tanto la implementación para el caso amortizado como para el peor caso se pueden volver persistentes. Aplazamos la presentación de esta solución hasta que presentemos el concepto de ejecución perezosa en la sección 4.5.

Por ahora, proponemos un acercamiento sencillo que frecuentemente es todo lo que se necesita para hacer las colas persistentes con complejidad constante en el peor caso. Esta solución depende de que no haya demasiadas versiones simultáneas.

10. Para ver por qué no, considere cualquier secuencia de n operaciones sobre colas. Para que el límite de tiempo constante amortizado valga, el tiempo total de todas las operaciones en la secuencia debe ser proporcional a n . ¿Pero, qué pasa si la secuencia repite una operación “costosa” en muchas versiones? Esto es posible, pues estamos hablando de cualquier secuencia. Y como tanto el tiempo que consume una operación costosa, como el número de versiones, pueden ser proporcionales a n , el límite del tiempo total puede crecer como n^2 .

11. Esta acepción de persistencia no se debe confundir con la persistencia tal como se usa en transacciones y bases de datos (ver secciones 8.5 y 9.6 (en CTM)), el cual es un concepto totalmente diferente.

Definimos una operación `ForkQ` que toma una cola `Q` y crea dos versiones idénticas `Q1` y `Q2`. Antes de esto, definimos un procedimiento `ForkD` que crea dos versiones de una lista de diferencias:

```
proc {ForkD D ?E ?F}
    D1#nil=D
    E1#E0=E {Append D1 E0 E1}
    F1#F0=F {Append D1 F0 F1}
in skip end
```

La invocación `{ForkD D E F}` toma una lista de diferencias `D` y devuelve dos copias frescas de ella, `E` y `F`. `Append` se usa para convertir una lista en una lista de diferencias fresca. Note que `ForkD` consume a `D`, i.e., `D` no puede seguir siendo utilizado pues su cola ya fue ligada. Ahora podemos definir `ForkQ`, el cual usa `ForkD` para crear dos versiones frescas de una cola:

```
proc {ForkQ Q ?Q1 ?Q2}
    q(N S E)=Q
    q(N S1 E1)=Q1
    q(N S2 E2)=Q2
in
    {ForkD S#E S1#E1 S2#E2}
end
```

`ForkQ` consume a `Q` y toma tiempo proporcional al tamaño de la cola. Podemos reescribir el ejemplo utilizando `ForkQ`, así:

```
...
{ForkQ Qa Qa1 Qa2}
{AlgúnProc Qa1}
Qb={InsCola Qa2 x}
Qc={InsCola Qb y}
...
```

Esto trabaja bien si se acepta que `ForkQ` sea una operación costosa.

3.4.6. Árboles

Después de las estructuras de datos lineales como las listas y las colas, los árboles son la estructura de datos recursiva más importante en el repertorio de un programador. Un árbol es un nodo hoja o un nodo que contiene uno o más árboles. Normalmente nos interesamos en árboles finitos, e.g., árboles con un número finito de nodos. Los nodos pueden tener información adicional. Presentamos una posible definición:

$$\begin{aligned} \langle \text{Árbol} \rangle & ::= \text{ hoja} \\ & | \text{ } \text{árbol}(\langle \text{Valor} \rangle \langle \text{Árbol} \rangle_1 \dots \langle \text{Árbol} \rangle_n) \end{aligned}$$

La diferencia básica entre una lista y un árbol es que una lista tiene una estructura lineal mientras que un árbol puede tener una estructura ramificada. Una lista siempre tiene un elemento junto con una lista más pequeña. Un árbol tiene un elemento junto con algún número de árboles más pequeños. Este número puede ser cualquier número natural, i.e., cero para nodos hoja y cualquier número positivo

para los nodos que no son hojas.

Hay muchos variedades diferentes de árboles, con diferentes estructuras de ramificación y contenidos de los nodos. Por ejemplo, una lista es un árbol en el cual los nodos que no son hojas tienen siempre exactamente un subárbol (se puede llamar árbol unario). En un árbol binario, los nodos que no son hojas tienen siempre exactamente dos subárboles. En un árbol ternario los nodos que no son hojas tienen siempre exactamente tres subárboles. En un árbol balanceado, todos los subárboles del mismo nodo tienen el mismo tamaño (i.e., el mismo número de nodos) o aproximadamente el mismo tamaño.

Cada variedad de árbol tiene su propia clase de algoritmos para construirlos, recorrerlos y buscar información en ellos. En este capítulo se utilizan diversas y diferentes variedades de árboles. Presentamos un algoritmo para dibujar árboles binarios de una manera agradable, mostramos cómo utilizar técnicas de programación de alto orden para calcular con árboles, e implementamos diccionarios con árboles binarios ordenados.

En esta sección se sientan las bases para estos desarrollos. Presentamos los algoritmos básicos que subyacen a muchas de las variaciones más sofisticadas. Definimos árboles binarios ordenados y mostramos cómo insertar información, buscar información, y borrar información de ellos.

3.4.6.1. Árbol binario ordenado

Un árbol binario ordenado $\langle \text{ÁrbolBinOrd} \rangle$ es, primero que todo, un árbol binario en el cual cada nodo que no es hoja incluye un par de valores:

$\langle \text{ÁrbolBinOrd} \rangle ::= \text{ hoja}$

Cada nodo que no es hoja incluye los valores `<ValorOrd>` y `<Valor>`. El primer valor `<ValorOrd>` es cualquier subtipo de `<Valor>` que sea totalmente ordenado, i.e., cuenta con funciones booleanas de comparación. Por ejemplo, `<Ent>` (el tipo entero) es una posibilidad. El segundo valor `<Valor>` es cualquier información que se quiera llevar allí; no se impone ninguna condición particular sobre él.

LLamaremos el valor ordenado la llave y el segundo valor la información. Entonces un árbol binario es ordenado si para cada nodo que no es hoja, todas las llaves en el primer subárbol son menores que la llave del nodo, y todas las llaves en el segundo subárbol son mayores que la llave del nodo.

3.4.6.2. Almacenando información en árboles

Un árbol binario ordenado se puede usar como un repositorio de información si definimos tres operaciones: buscar, insertar y borrar información.

Buscar información en un árbol binario ordenado significa buscar si una determinada llave está presente en alguno de los nodos del árbol, y de ser así, devolver la información presente en ese nodo. Con la condición de orden que cumplen los

árboles binarios ordenados, el algoritmo de búsqueda puede eliminar la mitad de los nodos restantes en cada etapa. Este proceso es llamado búsqueda binaria. El número de operaciones que se necesitan es proporcional a la profundidad del árbol, i.e., la longitud del camino más largo de la raíz a una hoja. La búsqueda puede ser programada como sigue:

```
fun {Buscar X T}
  case T
    of hoja then noencontrado
    [] árbol(Y V T1 T2) then
      if X<Y then {Buscar X T1}
      elseif X>Y then {Buscar X T2}
      else encontrado(V) end
    end
  end
```

Invocar {Buscar X T} devuelve encontrado(V) si se encuentra un nodo con llave X, y noencontrado en caso contrario. Otra forma de escribir `Buscar` es utilizando `andthen` en la declaración `case`:

```
fun {Buscar X T}
  case T
    of hoja then noencontrado
    [] árbol(Y V T1 T2) andthen X==Y then encontrado(V)
    [] árbol(Y V T1 T2) andthen X<Y then {Buscar X T1}
    [] árbol(Y V T1 T2) andthen X>Y then {Buscar X T2}
    end
  end
```

Muchos desarrolladores encuentran la segunda versión mucho más legible porque es más visual, i.e., presenta patrones que muestran cómo se ve el árbol en lugar de presentar instrucciones para descomponer el árbol. En una palabra, es más declarativo. Esto facilita verificar que sea correcto, i.e., asegurarse que ningún caso haya sido olvidado. En algoritmos más complicados sobre árboles, el reconocimiento de patrones junto con `andthen` es una clara ventaja sobre las declaraciones `if` explícitas.

Para insertar o borrar información en un árbol binario ordenado, construimos un nuevo árbol, idéntico al original excepto que tiene más o menos información. Esta es la operación de inserción:

```
fun {Insertar X V T}
  case T
    of hoja then árbol(X V hoja hoja)
    [] árbol(Y W T1 T2) andthen X==Y then
      árbol(X V T1 T2)
    [] árbol(Y W T1 T2) andthen X<Y then
      árbol(Y W {Insertar X V T1} T2)
    [] árbol(Y W T1 T2) andthen X>Y then
      árbol(Y W T1 {Insertar X V T2})
    end
  end
```

Invocar {Insertar X V T} devuelve un árbol nuevo que tiene la pareja (X V)

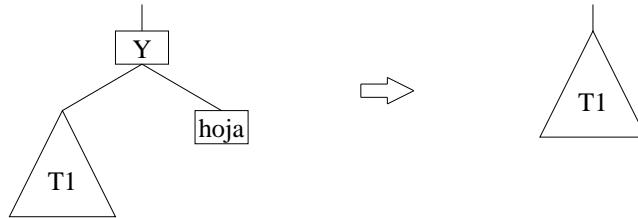


Figura 3.11: Borrando el nodo Y cuando un subárbol es una hoja (caso fácil).

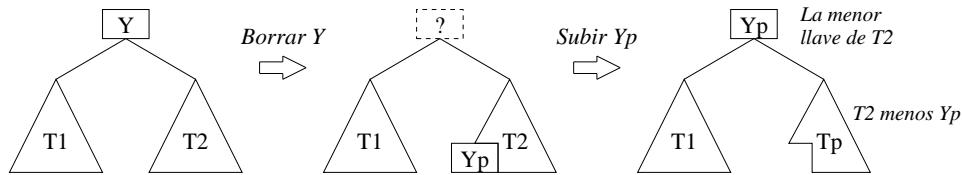


Figura 3.12: Borrando el nodo Y cuando ningún subárbol es una hoja (caso difícil).

insertada en el lugar correcto. Si T ya contiene x, entonces en el nuevo árbol se reemplaza la vieja información por v.

3.4.6.3. Borrado y reorganización del árbol

La operación de borrado guarda una sorpresa. Un primer intento es el siguiente:

```
fun {Borrar x T}
  case T
    of hoja then hoja
    [] árbol(Y W T1 T2) andthen X==Y then hoja
    [] árbol(Y W T1 T2) andthen X<Y then
        árbol(Y W {Borrar X T1} T2)
    [] árbol(Y W T1 T2) andthen X>Y then
        árbol(Y W T1 {Borrar X T2})
  end
end
```

Invocar {Borrar x T} debería devolver un árbol nuevo sin el nodo con llave x. Si T no contiene x, entonces se devuelve T sin cambiarlo. ¿Borrar parece bastante sencillo, pero la anterior definición es incorrecta? ¿Puede ver por qué?

Resulta que Borrar no es tan simple como Buscar o Insertar. El error en la definición anterior es que cuando $X==Y$, se está borrando todo el subárbol en lugar de borrar sólo el nodo en cuestión. Esto es correcto sólo si el subárbol es degenerado, i.e., si tanto T1 como T2 son nodos hoja. La solución no es tan obvia: cuando $X==Y$, hay que reorganizar el árbol de manera que el nodo con llave Y desaparezca, pero

el árbol debe seguir siendo un árbol binario ordenado. Hay dos casos, los cuales se ilustran en las figuras 3.11 y 3.12.

En la figura 3.11 se muestra el caso fácil, cuando un subárbol es una hoja. El árbol reorganizado es simplemente el otro subárbol. En la figura 3.12 se muestra el caso difícil, cuando ninguno de los subárboles es un nodo hoja. Cómo tapamos el “hueco” después de borrar y ? Otro nodo con otra llave debe tomar el lugar de y , tomándolo del interior de alguno de los subárboles. Una idea es tomar el nodo de T_2 con llave más pequeña, llamémoslo y_p , y volverlo la raíz del árbol reorganizado. Los nodos restantes de T_2 forman un subárbol de menor tamaño, llamémoslo T_p , el cual se coloca en el árbol reorganizado. Esto asegura que el árbol reorganizado sigue siendo binario ordenado, pues por construcción todas las llaves de T_1 son menores que y_p , la cual es menor, por la forma como se escogió, que todas las llaves de T_p .

Es interesante ver qué pasa cuando borramos repetidamente la raíz del árbol. Esto va “haciendo un hueco” al árbol desde adentro, borrando más y más la parte izquierda de T_2 . Finalmente, todo el subárbol izquierdo de T_2 será borrado y el subárbol derecho tomará su lugar. Continuando de esta forma, T_2 se achica más y más, pasando a través de estados intermedios en los cuales es un árbol binario ordenado, completo, pero más pequeño. Finalmente, desaparece completamente.

Para implementar la solución, utilizamos una función `{BorrarMenor T2}` que devuelve en una tupla el valor de la llave más pequeña de T_2 , su información asociada, y el nuevo árbol que ya no tiene ese nodo con esa llave. Con esta función podemos escribir una versión correcta de `Borrar` así:

```
fun {Borrar X T}
  case T
    of hoja then hoja
    [] árbol(Y W T1 T2) andthen X==Y then
      case {BorrarMenor T2}
        of nada then T1
        [] Yp#Vp#Tp then árbol(Yp Vp T1 Tp)
        end
    [] árbol(Y W T1 T2) andthen X<Y then
      árbol(Y W {Borrar X T1} T2)
    [] árbol(Y W T1 T2) andthen X>Y then
      árbol(Y W T1 {Borrar X T2})
    end
  end
```

La función `BorrarMenor` devuelve o una tripleta $Yp#Vp#Tp$ o el átomo `nada`. La definimos recursivamente así:

```

fun {BorrarMenor T}
  case T
    of hoja then nada
    [] árbol(Y V T1 T2) then
      case {BorrarMenor T1}
      of nada then Y#V#T2
      [] Yp#Vp#Tp then Yp#Vp#árbol(Y V Tp T2)
      end
    end
  end

```

Uno también podría tomar el nodo con la llave más grande de T1 en lugar del nodo con la llave más pequeña de T2. Esto da el mismo resultado.

La dificultad adicional de `Borrar` comparada con `Insertar` o `Buscar` ocurre con frecuencia con los algoritmos sobre árboles. La dificultad aparece debido a que un árbol ordenado satisface una condición global, a saber, estar ordenado. Muchas variedades de árboles se definen por medio de condiciones globales. Los algoritmos para manejar estos árboles son complejos debido a que tienen que mantener la condición global. Además, los algoritmos sobre árboles son más difíciles que los algoritmos sobre listas pues la recursión tiene que combinar resultados de diferentes problemas más pequeños, y no sólo de uno.

3.4.6.4. Recorrido de árboles

Recorrer un árbol significa realizar una operación sobre sus nodos en algún orden bien definido. Existen muchas formas de recorrer un árbol. Muchas de ellas se derivan de uno de los dos recorridos básicos, llamados recorridos en profundidad o en amplitud. Miremos estos recorridos.

Recorrido en profundidad El recorrido en profundidad es el más sencillo. Para cada nodo, primero se visita el nodo mismo, luego el subárbol izquierdo, y luego el subárbol derecho. Este recorrido es fácil de programar pues es muy cercano a la manera como se ejecuta la invocación de procedimientos anidados. Miremos un recorrido¹² que muestra la llave y la información de cada nodo:

```

proc {DFS T}
  case T
    of hoja then skip
    [] árbol(Llave Val I D) then
      {Browse Llave#Val}
      {DFS I}
      {DFS D}
    end
  end

```

12. Nota del traductor: El procedimiento DFS toma su nombre de sus siglas en inglés, *Depth-First Search*.

Un lector astuto se dará cuenta que este recorrido en profundidad no tiene mucho sentido en el modelo declarativo pues no calcula ningún resultado.¹³ Podemos solucionar esto agregando un acumulador. El siguiente es un recorrido que calcula una lista de todas las parejas llave/valor:

```
proc {CicloDFSAcu T S1 ?Sn}
  case T
    of hoja then Sn=S1
    [] árbol(Llave Val I D) then S2 S3 in
      S2=Llave#Val|S1
      {CicloDFSAcu I S2 S3}
      {CicloDFSAcu D S3 Sn}
  end
end
fun {DFSAcu T} {Reverse {CicloDFSAcu T nil $}} end
```

Aquí se usa un acumulador en la misma forma que vimos antes, con un estado de entrada S_1 y un estado de salida S_n . Como resultado CicloDFSAcu calcula la lista requerida pero en orden inverso; por eso DFSAcu tiene que invocar a Reverse para colocar la lista en el orden correcto. La versión siguiente calcula la lista en el orden correcto directamente:

```
proc {CicloDFSAcu2 T ?S1 Sn}
  case T
    of hoja then S1=Sn
    [] árbol(Llave Val I D) then S2 S3 in
      S1=Llave#Val|S2
      {CicloDFSAcu2 I S2 S3}
      {CicloDFSAcu2 D S3 Sn}
  end
end
fun {DFSAcu2 T} {CicloDFSAcu2 T $ nil} end
```

¿Ve usted en qué difieren las dos versiones? La segunda versión utiliza variables de flujo de datos no-ligadas. Es equivalente a considerar $S_1 \# S_n$ como una lista de diferencias. Esto nos produce un programa más natural que no necesita invocar a Reverse.

Recorrido en amplitud El recorrido en amplitud es el segundo recorrido básico. En éste, primero se recorren todos los nodos de profundidad 0, luego los nodos de profundidad 1, y así sucesivamente, yendo a un nivel más profundo en cada paso. En cada nivel, se recorren los nodos de izquierda a derecha. La profundidad de un nodo es la longitud del camino de la raíz al nodo, sin incluirlo. Para implementar el recorrido en amplitud, necesitamos una cola para guardar todos los nodos de una profundidad determinada. En la figura 3.13 se muestra cómo se hace.¹⁴ Se utiliza el tipo cola que definimos en la sección anterior. El siguiente nodo a visitar proviene

13. Browse no se puede definir en el modelo declarativo.

14. Nota del traductor: El procedimiento BFS toma su nombre de sus siglas en inglés, *Breadth-First Search*.

```
proc {BFS T}
  fun {InsertarÁrbolEnCola Q T}
    if T\=hoja then {InsCola Q T} else Q end
  end

  proc {ColaBFS Q1}
    if {EsVacíaCola Q1} then skip
    else X Q2 Llave Val I D in
      Q2={ElimDeCola Q1 X}
      árbol(Llave Val I D)=X
      {Browse Llave#Val}
      {ColaBFS {InsertarÁrbolEnCola {InsertarÁrbolEnCola Q2 I}
D}}
    end
  end
in
{ColaBFS {InsertarÁrbolEnCola {ColaNueva} T}}
end
```

Figura 3.13: Recorrido en amplitud.

de la cabeza de la cola. Los nodos de los dos subárboles se agregan a la cola (por su cola). El recorrido llegará a vistarlo a ellos una vez todos los otros nodos que hay en la cola hayan sido visitados, i.e., una vez todos los nodos del nivel actual hayan sido visitados.

Al igual que para el recorrido en profundidad, el recorrido en amplitud es útil en el modelo declarativo sólo si es complementado con un acumulador. En la figura 3.14 se presenta un ejemplo de recorrido en amplitud que calcula la lista de todas las parejas llave/valor de un árbol.

El recorrido en profundidad puede ser implementado en una forma similar al recorrido en amplitud, utilizando una estructura de datos explícita para guardar los nodos a visitar. Para el recorrido en profundidad, simplemente usamos una pila en lugar de una cola. En la figura 3.15 se define el recorrido, utilizando una lista para implementar la pila.

¿Cómo se compara la nueva versión de `DFS` con la original? Ambas versiones usan una pila para recordar los subárboles que deben ser visitados. En la versión original, la pila está oculta: es la pila semántica. Hay dos invocaciones recursivas. Durante la ejecución de la primera invocación, la segunda invocación se guarda en la pila semántica. En la versión nueva, la pila es una estructura de datos explícita. Como la versión nueva es recursiva por la cola, igual que `BFS`, la pila semántica no crece. La versión nueva intercambia espacio en la pila semántica por espacio en el almácén.

Miremos cuánta memoria utilizan los algoritmos `DFS` y `BFS`. Supongamos que tenemos un árbol de profundidad n con 2^n nodos hoja y $2^n - 1$ nodos internos (que no son nodos hoja). ¿Cuán grandes llegan a ser la pila y la cola? Podemos probar

```

fun {BFSACu T}
  fun {InsertarÁrbolEnCola Q T}
    if T\=hoja then {InsCola Q T} else Q end
  end

  proc {ColaBFS Q1 ?S1 Sn}
    if {EsVaciaCola Q1} then S1=Sn
    else X Q2 Llave Val I D S2 in
      Q2={ElimDeCola Q1 X}
      árbol(Llave Val I D)=X
      S1=Llave#Val|S2
      {ColaBFS {InsertarÁrbolEnCola {InsertarÁrbolEnCola Q2 I}
D} S2 Sn}
    end
  end
  in
    {ColaBFS {InsertarÁrbolEnCola {ColaNueva} T} $ nil}
  end

```

Figura 3.14: Recorrido en amplitud con acumulador.

```

proc {DFS T}
  fun {InsertarÁrbolEnPila S T}
    if T\=hoja then T|S else S end
  end

  proc {PilaDFS S1}
    case S1
    of nil then skip
    [] X|S2 then
      árbol(Llave Val I D)=X
    in
      {Browse Llave#Val}
      {PilaDFS {InsertarÁrbolEnPila {InsertarÁrbolPiCola S2 D}
I}}
    end
  end
  in
    {PilaDFS {InsertarÁrbolEnPila nil T}}
  end

```

Figura 3.15: Recorrido en profundidad con una pila explícita.

que la pila tendrá a lo sumo $n + 1$ elementos y que la cola tendrá a lo sumo 2^n elementos. Por lo tanto, DFS es mucho más económico: utiliza memoria proporcional a la profundidad del árbol. BFS, por su lado, utiliza memoria proporcional al tamaño del árbol.

3.4.7. Dibujando árboles

Ahora que hemos introducido los árboles y la programación con ellos, escribamos un programa más significativo. Escribiremos un programa para dibujar un árbol binario en una forma estéticamente agradable. El programa calcula las coordenadas de cada nodo. Este programa es interesante porque recorre el árbol por dos motivos: para calcular las coordenadas y para sumar las coordenadas al propio árbol.

Las restricciones para dibujar árboles

Primero definimos un tipo para los árboles:

```
 $\langle \text{Árbol} \rangle ::= \text{árbol}(\text{llave}:\langle \text{Literal} \rangle \text{ val}:\langle \text{Valor} \rangle \text{ izq}:\langle \text{Árbol} \rangle \text{ der}:\langle \text{Árbol} \rangle)$ 
| hoja
```

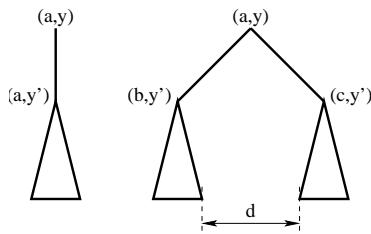
Cada nodo es una hoja o tiene dos hijos. A diferencia de la sección 3.4.6, en ésta usamos un registro con átomos como campos para definir un árbol, en lugar de una tupla. Existe una buena razón para hacer esto, la cual será clara cuando hablemos sobre el principio de independencia. Suponga que tenemos las siguientes restricciones sobre la forma de dibujar un árbol:

1. Existe un espaciamiento mínimo entre los dos subárboles de cada nodo. Para ser precisos, el nodo más a la derecha del subárbol izquierdo debe estar a una distancia, mayor o igual a la mínima permitida, del nodo más a la izquierda del subárbol derecho.
2. Si un nodo tiene dos hijos, entonces su posición horizontal debe ser el promedio aritmético de las posiciones horizontales de ellos.
3. Si un nodo tiene un único nodo hijo, entonces el hijo está directamente debajo de él.
4. La posición vertical de un nodo es proporcional a su nivel en el árbol.

Además, para evitar atiborrar la ventana, sólo se muestran los nodos de tipo `árbol`. En la figura 3.16 se muestran gráficamente las restricciones en términos de las coordenadas de cada nodo. El árbol ejemplo de la figura 3.17 se dibuja como se muestra en la figura 3.19.

Calculando las posiciones de los nodos

El algoritmo para dibujar árboles calcula las posiciones de los nodos recorriendo el árbol, pasando información entre los nodos, y calculando valores en cada nodo.



1. La distancia d entre subárboles tiene un valor mínimo
2. Si hay dos hijos, a es el promedio de b y c
3. Si solo existe un hijo, está directamente debajo del padre
4. La posición vertical y es proporcional al nivel en el árbol

Figura 3.16: The tree-drawing constraints.

```

árbol(llave:a val:111
      izq:árbol(llave:b val:55
                  izq:árbol(llave:x val:100
                                  izq:árbol(llave:z val:56 izq:hoja der:hoja)
                                  der:árbol(llave:w val:23 izq:hoja der:hoja))
                  der:árbol(llave:y val:105 izq:hoja
                                  der:árbol(llave:r val:77 izq:hoja der:hoja)))
      der:árbol(llave:c val:123
                  izq:árbol(llave:d val:119
                                  izq:árbol(llave:g val:44 izq:hoja der:hoja)
                                  der:árbol(llave:h val:50
                                      izq:árbol(llave:i val:5 izq:hoja der:hoja)
                                      der:árbol(llave:j val:6 izq:hoja der:hoja)))
                  der:árbol(llave:e val:133 izq:hoja der:hoja)))

```

Figura 3.17: Un ejemplo de árbol.

El recorrido tiene que ser realizado cuidadosamente de manera que toda la información necesaria esté disponible en el momento preciso. Saber qué recorrido es el adecuado depende de las restricciones que haya. Para las restricciones mencionadas, es suficiente recorrer el árbol en profundidad. En este recorrido, cada subárbol izquierdo es visitado antes que el subárbol derecho correspondiente. Un algoritmo básico para el recorrido se ve así¹⁵:

```

proc {DF Árbol}
  case Árbol
    of árbol(izq:I der:D ...) then
      {DF I}
      {DF D}
      [] hoja then
        skip
      end
    end

```

15. Nota del traductor: DF por las siglas en inglés de *Depth First*.

```

Escala=30
proc {CoordDibujoDF Árbol Nivel LimIzq ?RaízX ?LimDer}
  case Árbol
  of árbol(x:X y:Y izq:hoja der:hoja ...) then
    X=RaízX=LimDer=LimIzq
    Y=Escala*Nivel
  [] árbol(x:X y:Y izq:I der:hoja ...) then
    X=RaízX
    Y=Escala*Nivel
    {CoordDibujoDF I Nivel+1 LimIzq RaízX LimDer}
  [] árbol(x:X y:Y izq:hoja der:D ...) then
    X=RaízX
    Y=Escala*Nivel
    {CoordDibujoDF D Nivel+1 LimIzq RaízX LimDer}
  [] árbol(x:X y:Y izq:I der:D ...) then
    IRaízX ILimDer DRaízX DLimIzq
    in
      Y=Escala*Nivel
      {CoordDibujoDF I Nivel+1 LimIzq IRaízX ILimDer}
      DLimIzq=ILimDer+Escala
      {CoordDibujoDF D Nivel+1 DLimIzq DRaízX LimDer}
      X=RaízX=(IRaízX+DRaízX) div 2
    end
  end
end

```

Figura 3.18: Algoritmo para dibujar árboles.

El algoritmo para dibujar árboles realiza un recorrido en profundidad y calcula las coordenadas (x,y) de cada nodo durante el recorrido. Una tarea previa a la ejecución del algoritmo consiste en extender los nodos árbol con los campos x y y en cada nodo:

```

fun {AgregueXY Árbol}
  case Árbol
  of árbol(izq:I der:D ...) then
    {Adjoin Árbol
      árbol(x:_ y:_ izq:{AgregueXY I} der:{AgregueXY D})}
  [] hoja then
    hoja
  end
end

```

La función AgregueXY devuelve un árbol nuevo con los campos x y y añadidos a todos los nodos. Se utiliza la función Adjoin con la cual se pueden añadir nuevos campos a los registros y ocultar los viejos. Esto se explica en el apéndice B.3.2. El algoritmo para dibujar árboles llenará estos dos campos con las coordenadas de cada nodo. Si los dos campos no existen en ninguna parte del registro, entonces no habrá conflicto con ninguna otra información allí.

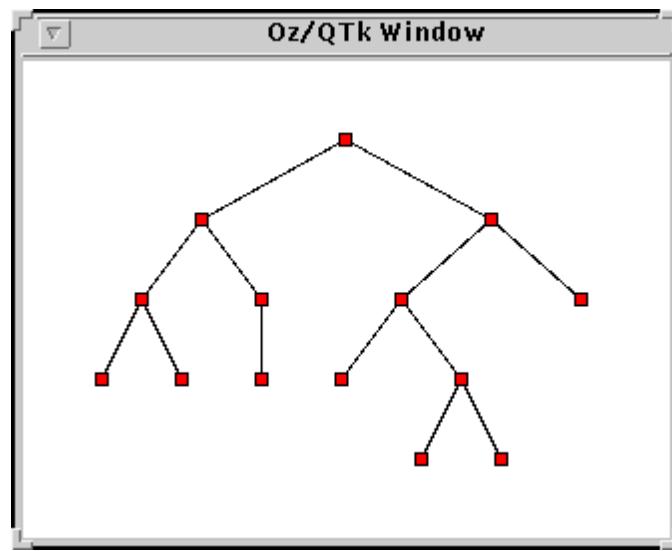


Figura 3.19: El árbol del ejemplo dibujado con el algoritmo para dibujar árboles.

Para implementar el algoritmo para dibujar árboles, extendemos el recorrido en profundidad pasando dos argumentos hacia abajo (a saber, el nivel en el árbol y el límite sobre la posición más a la izquierda del subárbol) y dos argumentos hacia arriba (a saber, la posición horizontal de la raíz del subárbol y la posición más a la derecha del subárbol). El paso de parámetros hacia abajo se llama herencia de argumentos. El paso de parámetros hacia arriba se llama síntesis de argumentos. Con estos argumentos adicionales, tenemos suficiente información para calcular las coordenadas de todos los nodos. En la figura 3.18 se presenta en forma completa el algoritmo para dibujar árboles. El parámetro `Escala` define la unidad básica del dibujo del árbol, i.e., la mínima distancia entre nodos. Los argumentos iniciales son `Nivel=1` y `LimIzq=Escala`. Hay cuatro casos, dependiendo de si un nodo tiene dos, uno (izquierdo o derecho), o cero subárboles. El reconocimiento de patrones de la declaración `case` siempre toma el caso correcto. Esto funciona porque las comprobaciones se realizan de manera secuencial.

3.4.8. Análisis sintáctico

Como segundo caso de estudio de la programación declarativa, escribamos un analizador sintáctico para un lenguaje imperativo pequeño con una sintaxis similar a la de Pascal. Este ejemplo utiliza muchas de las técnicas que hemos visto: en particular, utiliza un acumulador y construye un árbol.

$\langle \text{Prog} \rangle$::=	program $\langle \text{Id} \rangle$; $\langle \text{Decl} \rangle$ end
$\langle \text{Decl} \rangle$::=	begin $\langle \text{Decl} \rangle$ { ; $\langle \text{Decl} \rangle$ } end
		$\langle \text{Id} \rangle := \langle \text{Expr} \rangle$
		if $\langle \text{Comp} \rangle$ then $\langle \text{Decl} \rangle$ else $\langle \text{Decl} \rangle$
		while $\langle \text{Comp} \rangle$ do $\langle \text{Decl} \rangle$
		read $\langle \text{Id} \rangle$
		write $\langle \text{Expr} \rangle$
$\langle \text{Comp} \rangle$::=	$\langle \text{Expr} \rangle \{ \langle \text{OPC} \rangle \langle \text{Expr} \rangle \}$
$\langle \text{Expr} \rangle$::=	$\langle \text{Term} \rangle \{ \langle \text{OPE} \rangle \langle \text{Term} \rangle \}$
$\langle \text{Term} \rangle$::=	$\langle \text{Fact} \rangle \{ \langle \text{OPT} \rangle \langle \text{Fact} \rangle \}$
$\langle \text{Fact} \rangle$::=	$\langle \text{Entero} \rangle \mid \langle \text{Id} \rangle \mid (\langle \text{Expr} \rangle)$
$\langle \text{OPC} \rangle$::=	'==' '!=` '>' '<' '=<' '>='
$\langle \text{OPE} \rangle$::=	'+' '-'
$\langle \text{OPT} \rangle$::=	'*' '/'
$\langle \text{Entero} \rangle$::=	(entero)
$\langle \text{Id} \rangle$::=	(átomo)

Tabla 3.2: El lenguaje de entrada del analizador sintáctico (el cual es una secuencia de *lexemas*).

Qué es un analizador sintáctico?

Un analizador sintáctico es una parte de un compilador. Un compilador es un programa que traduce una secuencia de caracteres que representan un programa, en una secuencia de instrucciones de bajo nivel que pueden ser ejecutadas en una máquina. En su forma más básica un compilador consta de tres partes:

1. *Analizador léxico*. El analizador léxico lee una secuencia de caracteres y devuelve una secuencia de *lexemas*.
2. *Analizador sintáctico*. El analizador sintáctico lee una secuencia de *lexemas* y devuelve un árbol de sintaxis abstracta.
3. *Generador de código*. El generador de código recorre el árbol de sintaxis y genera instrucciones de bajo nivel para una máquina real o una máquina abstracta.

Normalmente esta estructura se extiende con optimizadores para mejorar la generación de código. En esta sección sólo escribiremos el analizador sintáctico. Primero definiremos los formatos de la entrada y la salida del analizador sintáctico.

Los lenguajes de entrada y salida del analizador sintáctico

El analizador sintáctico acepta una secuencia de *lexemas* acorde con la gramática presentada en la tabla 3.2 y devuelve un árbol de sintaxis abstracta. La gramática se diseña cuidadosamente de manera que sea recursiva y determinística. Esto significa

$\langle \text{Prog} \rangle$	$::= \text{prog}(\langle \text{Id} \rangle \langle \text{Decl} \rangle)$
$\langle \text{Decl} \rangle$	$::= \text{`} ; \text{'(} \langle \text{Decl} \rangle \langle \text{Decl} \rangle \text{')}$
	$ \text{`} \text{assign}(\langle \text{Id} \rangle \langle \text{Expr} \rangle)$
	$ \text{`} \text{if}\text{'(} \langle \text{Comp} \rangle \langle \text{Decl} \rangle \langle \text{Decl} \rangle \text{')}$
	$ \text{`} \text{while}(\langle \text{Comp} \rangle \langle \text{Decl} \rangle)$
	$ \text{`} \text{read}(\langle \text{Id} \rangle)$
	$ \text{`} \text{write}(\langle \text{Expr} \rangle)$
$\langle \text{Comp} \rangle$	$::= \langle \text{OPC} \rangle (\langle \text{Expr} \rangle \langle \text{Expr} \rangle)$
$\langle \text{Expr} \rangle$	$::= \langle \text{Id} \rangle \langle \text{Enter} \rangle \langle \text{OP} \rangle (\langle \text{Expr} \rangle \langle \text{Expr} \rangle)$
$\langle \text{OPC} \rangle$	$::= \text{`} == \text{' } \text{`} != \text{' } \text{`} > \text{' } \text{`} < \text{' } \text{`} = < \text{' } \text{`} > = \text{'}$
$\langle \text{OP} \rangle$	$::= \text{`} + \text{' } \text{`} - \text{' } \text{`} * \text{' } \text{`} / \text{'}$
$\langle \text{Enter} \rangle$	$::= (\text{entero})$
$\langle \text{Id} \rangle$	$::= (\text{átomo})$

Tabla 3.3: El lenguaje de salida del analizador sintáctico (el cual es un árbol).

que la escogencia de la regla gramatical está completamente determinada por el *lexema* siguiente. Esto posibilita escribir un analizador sintáctico de arriba hacia abajo¹⁶, de izquierda a derecha, mirando un solo *lexema* hacia adelante.

Por ejemplo, suponga que deseamos analizar sintácticamente un $\langle \text{Term} \rangle$, el cual consiste de una serie no vacía de $\langle \text{Fact} \rangle$ separada por *lexemas* $\langle \text{OPT} \rangle$. Al hacerlo, primero analizamos sintácticamente un $\langle \text{Fact} \rangle$, y luego examinamos el *lexema* siguiente. Si es un $\langle \text{OPT} \rangle$, entonces sabemos que la serie continúa; sino, entonces sabemos que la serie termina allí, i.e., el $\langle \text{Term} \rangle$ se acabó. Para que esta estrategia de análisis sintáctico funcione, necesitamos que no exista superposición entre los *lexemas* $\langle \text{OPT} \rangle$ y los otros posibles *lexemas* que pueden venir después de un $\langle \text{Fact} \rangle$. Inspeccionando las reglas gramaticales, vemos que los otros *lexemas* que pueden venir después de un $\langle \text{Fact} \rangle$ están en $\langle \text{OPE} \rangle$, o en $\langle \text{OPC} \rangle$, o en el conjunto $\{ ;, \text{end}, \text{then}, \text{do}, \text{else}, \}\}$. Confirmamos entonces que no hay superposición entre los *lexemas* de $\langle \text{OPT} \rangle$ y los anteriores.

Hay dos clases de símbolos en la tabla 3.2:terminales y no terminales. Un símbolo no terminal es un símbolo que se expande de acuerdo a una regla gramatical. Un símbolo terminal es un símbolo que corresponde directamente a un *lexema* en la entrada, y que no será expandido. Los símbolos no terminales son $\langle \text{Prog} \rangle$ (programa completo), $\langle \text{Decl} \rangle$ (declaración), $\langle \text{Comp} \rangle$ (comparación), $\langle \text{Expr} \rangle$ (expresión), $\langle \text{Term} \rangle$ (término), $\langle \text{Fact} \rangle$ (factor), $\langle \text{OPC} \rangle$ (operador de comparación), $\langle \text{OPE} \rangle$ (operador entre expresiones), y $\langle \text{OPT} \rangle$ (operador entre términos). Para analizar sintácticamente un programa, se comienza con $\langle \text{Prog} \rangle$ y se expande hasta encontrar una secuencia de *lexemas* que corresponda exactamente con la entrada.

16. Nota del traductor: *top-down*, en inglés.

```

fun {Decl S1 Sn}
T|S2=S1 in
  case T
  of begin then
    {Secuencia Decl fun {$ X} X==';' end S2 `end'|Sn}
    [] `if` then C X1 X2 S3 S4 S5 S6 in
      C={Comp S2 S3}
      S3='then'|S4
      X1={Decl S4 S5}
      S5='else'|S6
      X2={Decl S6 Sn}
      `if'(C X1 X2)
    [] while then C X S3 S4 in
      C={Comp S2 S3}
      S3='do'|S4
      X={Decl S4 Sn}
      while(C X)
    [] read then I in
      I={Id S2 Sn}
      read(I)
    [] write then E in
      E={Expr S2 Sn}
      write(E)
    elseif {IsIdent T} then E S3 in
      S2=':= '|S3
      E={Expr S3 Sn}
      assign(T E)
    else
      S1=Sn
      raise error(S1) end
    end
end

```

Figura 3.20: Analizador sintáctico de arriba hacia abajo, de izquierda a derecha, mirando un solo *lexema* hacia adelante.

La salida del analizador sintáctico es un árbol (i.e., un registro anidado) con la sintaxis presentada en la tabla 3.3. Superficialmente, las tablas 3.2 y 3.3 tienen un contenido similar, pero son realmente bastante diferentes: la primera define una secuencia de *lexemas* y la segunda un árbol. La primera no muestra la estructura del programa de entrada—decimos que es plana. La segunda expone esta estructura—decimos que es anidada. Como este registro anidado expone la estructura del programa, entonces lo llamamos un árbol de sintaxis abstracta. Es abstracta porque la sintaxis está codificada como una estructura de datos en el lenguaje, y ya no en términos de *lexemas*. El papel del analizador sintáctico es extraer la estructura a partir de la entrada plana. Sin esta estructura es extremadamente difícil escribir un generador o unos optimizadores de código.

El programa analizador sintáctico

La manera de ejecutar al analizador sintáctico es con la invocación {Prog S1 Sn}, donde S1 es una lista de *lexemas* de entrada y Sn es lo que queda por analizar de S1. Esta invocación devuelve la salida analizada. Por ejemplo:

```
declare A Sn in
A={Prog
    [program foo `;`
     while a `+` 3 `<` b `do` b `:=` b `+` 1 `end`"]
    Sn}
{Browse A}
```

muestra la salida siguiente del analizador:

```
prog(foo while(`<`(`+`(a 3) b) assign(b `+`(b 1))))
```

Ahora presentamos el código del programa para el analizador sintáctico, anotado. Prog se escribe así:

```
fun {Prog S1 Sn}
    Y Z S2 S3 S4 S5
in
    S1=program|S2    Y={Id S2 S3}    S3=`;`|S4
    Z={Decl S4 S5}  S5=`end`|Sn
    prog(Y Z)
end
```

El acumulador es pasado a través de todos los símbolos terminales y no terminales. Cada símbolo no terminal tiene una función asociada para analizarlo. Las declaraciones son analizadas con la función **Decl**, la cual se muestra en la figura 3.20. Se coloca en T el *lexema* de mirada hacia adelante y se usa en una declaración **case** para encontrar la rama correcta de la regla gramatical de **Decl**.

Las secuencias de declaraciones son analizadas por el procedimiento **Secuencia**, un procedimiento genérico que también maneja secuencias de comparaciones, secuencias de expresiones, y secuencias de términos. **Secuencia** se escribe así:

```
fun {Secuencia NoTerm Sep S1 Sn}
    fun {CicloSecuencia Prefijo S2 Sn}
        case S2 of T|S3 andthen {Sep T} then Prox S4 in
            Prox={NoTerm S3 S4}
            {CicloSecuencia T(Prefijo Prox) S4 Sn}
        else
            Sn=S2 Prefijo
        end
    end
    Primero S2
in
    Primero={NoTerm S1 S2}
    {CicloSecuencia Primero S2 Sn}
end
```

Esta función toma dos funciones de entrada: **NoTerm** (una de las funciones asociadas a un símbolo no terminal) y **Sep** (una función que detecta el símbolo separador de

una secuencia). La sintaxis $T(x_1 \ x_2)$ realiza creación dinámica de registros de acuerdo a una etiqueta que sólo se conocerá en tiempo de ejecución; es un azúcar sintáctico para

```
local R={MakeRecord T [1 2]} in X1=R.1 X2=R.2 R end
```

Las comparaciones, las expresiones, y los términos se analizan como se muestra a continuación con la ayuda de la función Secuencia:

```
fun {Comp S1 Sn} {Secuencia Expr COP S1 Sn} end
fun {Expr S1 Sn} {Secuencia Term EOP S1 Sn} end
fun {Term S1 Sn} {Secuencia Fact TOP S1 Sn} end
```

Cada una de estas tres funciones tiene su correspondiente función para detectar separadores:

```
fun {COP Y}
Y=='<' orelse Y=='>' orelse Y=='=<' orelse
Y=='>=' orelse Y=='===' orelse Y=='!='
end
fun {EOP Y} Y=='+' orelse Y=='-' end
fun {TOP Y} Y=='*' orelse Y=='/' end
```

Finalmente, los factores y los identificadores se analizan de la siguiente manera:

```
fun {Fact S1 Sn}
T|S2=S1 in
if {IsInt T} orelse {IsIdent T} then
S2=Sn
T
else E S2 S3 in
S1='('|S2
E={Expr S2 S3}
S3=')'|Sn
E
end
end

fun {Id S1 Sn} x in S1=X|Sn true={IsIdent X} x end
fun {IsIdent X} {IsAtom X} end
```

Los enteros se representan como valores enteros predefinidos y se detectan utilizando la función predefinida `IsInt`.

Esta técnica de análisis sintáctico funciona para gramáticas para las cuales un *lexema* de mirada hacia adelante es suficiente. Algunas gramáticas, llamadas gramáticas ambigüas, requieren mirar más de un lexema hacia adelante para decidir cuál regla gramatical usar. Una forma sencilla de analizar sintácticamente este tipo de gramáticas es con escogencia no determinística, como se explica en el capítulo 9 (en CTM).

3.5. Eficiencia en tiempo y en espacio

La programación declarativa es en todo caso programación; aunque tenga unas propiedades matemáticas fuertes, la programación declarativa produce programas reales que se ejecutan en computadores reales. Por tanto, es importante pensar en eficiencia computacional. Hay dos aspectos principalmente que tiene que ver con la eficiencia: el tiempo de ejecución (e.g., en segundos) y la utilización de memoria (e.g., en bytes). Mostraremos cómo calcularlos ambos.

3.5.1. Tiempo de ejecución

Usando el lenguaje núcleo y su semántica, podemos calcular el tiempo de ejecución módulo un factor constante. Por ejemplo, para el algoritmo mergesort seremos capaces de decir que el tiempo de ejecución es proporcional a $n \log n$, para listas de entrada de tamaño n .

La complejidad asintótica en tiempo de un algoritmo es la menor cota superior sobre su tiempo de ejecución como una función del tamaño de la entrada, módulo un factor constante. Esto se llama también complejidad en tiempo en el peor caso.

Para encontrar el factor constante, es necesario medir los tiempos de ejecuciones reales de la implementación del programa. Calcular el factor constante a priori es extremadamente difícil, principalmente porque los sistemas de cómputo modernos tienen un *hardware* complejo y una estructura de *software* que introducen mucho ruido y hacen impredecible el tiempo de ejecución: los sistemas realizan administración de la memoria (ver sección 2.5), tienen sistemas de memoria complejos (con memoria virtual y varios niveles de memoria oculta), tienen arquitecturas complejas superescalares e interconectadas (muchas instrucciones se encuentran, simultáneamente, en varios estados de ejecución; el tiempo de ejecución de una instrucción depende frecuentemente de las otras instrucciones presentes), y el sistema operativo hace cambios de contexto en momentos impredecibles. Esta impredecibilidad mejora el desempeño en promedio con el costo de mayores fluctuaciones en el desempeño. Para mayor información sobre medición del desempeño y sus dificultades, recomendamos [80].

La notación \mathcal{O}

Presentaremos el tiempo de ejecución de un programa en términos de la notación $\mathcal{O}(f(n))$. Esta notación nos permite hablar sobre el tiempo de ejecución sin tener que especificar el factor constante. Sea $T(n)$ una función que mide el tiempo de ejecución de algún programa, en función del tamaño n de la entrada. Sea $f(n)$ otra función definida sobre los enteros no negativos. Decimos que $T(n)$ es de orden $f(n)$ si $T(n) \leq c.f(n)$ para alguna constante positiva c , para todo n excepto para valores pequeños $n \leq n_0$. Es decir, a medida que n crece existe un punto después del cual $T(n)$ nunca llega a ser más grande que $c.f(n)$.

Alguna veces esto se escribe $T(n) = \mathcal{O}(f(n))$. ¡Tenga cuidado! Este uso del símbolo igual es un abuso de notación, pues no hay ninguna igualdad implicada en esta definición. Si $g(n) = \mathcal{O}(f(n))$ y $h(n) = \mathcal{O}(f(n))$, entonces no se puede concluir

$\langle d \rangle ::=$	
skip	k
$\langle x \rangle_1 = \langle x \rangle_2$	k
$\langle x \rangle = \langle v \rangle$	k
$\langle d \rangle_1 \langle d \rangle_2$	$T(d_1) + T(d_2)$
local $\langle x \rangle$ in $\langle d \rangle$ end	$k + T(d)$
proc $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \} \langle d \rangle$ end	k
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	$k + \max(T(d_1), T(d_2))$
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	$k + \max(T(d_1), T(d_2))$
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	$T_x(\text{tam}_x(I_x(\{y_1, \dots, y_n\})))$

Tabla 3.4: Tiempo de ejecución de las instrucciones del lenguaje núcleo.

que $g(n) = h(n)$. Una mejor manera de entender la notación \mathcal{O} es en términos de conjuntos y de la relación de pertenencia: $\mathcal{O}(f(n))$ es un conjunto de funciones, y decir que $T(n)$ es $\mathcal{O}(f(n))$ significa simplemente que $T(n)$ pertenece al conjunto.

Calculando el tiempo de ejecución

Usamos el lenguaje núcleo como guía. Cada instrucción del lenguaje núcleo tiene un tiempo de ejecución bien definido, el cual puede ser una función del tamaño de sus argumentos. Suponga que tenemos un programa que consiste de p funciones F_1, \dots, F_p . Nos gustaría calcular las p funciones T_{F1}, \dots, T_{Fp} . Esto se hace en tres etapas:

1. Traducir el programa al lenguaje núcleo.
2. Utilizar los tiempos de ejecución del núcleo para construir una colección de ecuaciones que contengan T_{F1}, \dots, T_{Fp} . Estas ecuaciones se llaman ecuaciones de recurrencia pues definen el resultado para n en función de los resultados para valores menores que n .
3. Resolver las ecuaciones de recurrencia para T_{F1}, \dots, T_{Fp} .

En la tabla 3.4 se presenta el tiempo de ejecución $T(d)$ para cada declaración $\langle d \rangle$ del núcleo. En esta tabla, s es un entero y los argumentos $y_i = E(\langle y \rangle_i)$ para $1 \leq i \leq n$, para el ambiente apropiado E . Cada instancia de k es una constante positiva real diferente. La función $I_x(\{y_1, \dots, y_n\})$ devuelve el subconjunto de los argumentos del procedimiento x que son usados como argumentos de entrada.¹⁷ La función $\text{tam}_x(\{y_1, \dots, y_k\})$ es el “tamaño” de los argumentos de entrada al procedimiento x . Podemos definir libremente el tamaño como mejor nos parezca: si

17. Esto puede diferir de invocación a invocación, e.g., cuando un procedimiento es usado para realizar diferentes tareas en diferentes invocaciones.

quedá mal definido, entonces las ecuaciones de recurrencia no tendrán solución. Para las instrucciones $\langle x \rangle = \langle y \rangle$ y $\langle x \rangle = \langle v \rangle$ existe un caso particular en que pueden tomar más tiempo que una constante, a saber, cuando ambos argumentos están ligados a grandes valores parciales. En este caso, el tiempo es proporcional al tamaño de la parte común de los valores parciales.

Ejemplo: la función Append

Presentamos un ejemplo sencillo para mostrar cómo funciona esto. Considere la función Append:

```
fun {Append xs ys}
  case xs
    of nil then ys
    [] x|xr then x|{Append xr ys}
    end
  end
```

Esto se traduce así al lenguaje núcleo:

```
proc {Append xs ys ?zs}
  case xs
    of nil then zs=ys
    [] x|xr then zr in
      zs=x|zr
      {Append xr ys zr}
    end
  end
```

Utilizando la tabla 3.4, se construye la ecuación de recurrencia siguiente para una invocación de Append:

$$T_{\text{Append}}(\text{tam}(I(\{xs, ys, zs\}))) = k_1 + \max(k_2, k_3 + T_{\text{Append}}(\text{tam}(I(\{xr, ys, zr\}))))$$

(Los subíndices para tam e I no se necesitan acá.) Simplifiquemos esto. Sabemos que $I(\{xs, ys, zs\}) = \{xs\}$ y suponemos que $\text{tam}(\{xs\}) = n$, donde n es la longitud de xs . Esto da

$$T_{\text{Append}}(n) = k_1 + \max(k_2, k_3 + T_{\text{Append}}(n - 1))$$

Simplificando más llegamos a

$$T_{\text{Append}}(n) = k_4 + T_{\text{Append}}(n - 1)$$

Manejamos el caso de base tomando un valor particular de xs para el cual podemos calcular el resultado directamente. Tomemos $xs=\text{nil}$. Esto lleva a

$$T_{\text{Append}}(0) = k_5$$

Resolviendo las dos ecuaciones llegamos a

$$T_{\text{Append}}(n) = k_4 \cdot n + k_5$$

Por lo tanto $T_{\text{Append}}(n)$ es $\mathcal{O}(n)$.

Ecuaciónn	Solución
$T(n) = k + T(n - 1)$	$\Theta(n)$
$T(n) = k_1 + k_2 \cdot n + T(n - 1)$	$\Theta(n^2)$
$T(n) = k + T(n/2)$	$\Theta(\log n)$
$T(n) = k_1 + k_2 \cdot n + T(n/2)$	$\Theta(n)$
$T(n) = k + 2 \cdot T(n/2)$	$\Theta(n)$
$T(n) = k_1 + k_2 \cdot n + 2 \cdot T(n/2)$	$\Theta(n \log n)$
$T(n) = k_1 \cdot n^{k_2} + k_3 \cdot T(n - 1)$	$\Theta(k_3^n)$ (if $k_3 > 1$)

Tabla 3.5: Algunas ecuaciones de recurrencia frecuentes y sus soluciones.

Ecuaciones de recurrencia

Antes de mirar más ejemplos, demos un paso atrás y miremos las ecuaciones de recurrencia en general. Una ecuación de recurrencia tiene una de las dos formas siguientes:

- Una ecuación que define una función $T(n)$ en términos de $T(m_1), \dots, T(m_k)$, donde $m_1, \dots, m_k < n$.
- Una ecuación que presenta $T(n)$ directamente para ciertos valores de n , e.g., $T(0)$ o $T(1)$.

Cuando se calculan tiempos de ejecución, aparecen ecuaciones de recurrencia de diferentes clases. En la tabla 3.5 se presentan algunas ecuaciones de recurrencia que ocurren frecuentemente y sus soluciones. La tabla supone que las k son constantes diferentes de cero. Hay muchas técnicas para derivar estas soluciones. Veremos unas pocas en los ejemplos que siguen. En la caja se explican dos de las más utilizadas.

Ejemplo: PascalRápido

En el capítulo 1, introducimos la función `PascalRápido` y afirmamos, más intuitivamente que otra cosa, que `{PascalRápido N}` es $\Theta(n^2)$. Veamos si podemos derivar esto más rigurosamente. Recordemos la definición de `PascalRápido`:

```
fun {PascalRápido N}
  if N==1 then [1]
  else L in
    L={PascalRápido N-1}
    {SumListas {DesplIzq L} {DesplDer L}}
  end
end
```

Podemos construir las ecuaciones directamente mirando esta definición, sin tener que traducir las funciones en procedimientos. Mirando la definición es fácil ver que `DesplDer` es $\Theta(1)$, i.e., es constante en tiempo. Utilizando un razonamiento similar

Resolviendo ecuaciones de recurrencia

Las técnicas siguientes son frecuentemente utilizadas:

- Una técnica sencilla de tres etapas que casi siempre funciona en la práctica. Primero, conseguir los resultados exactos para entradas de tamaño pequeño (e.g.: $T(0) = k$, $T(1) = k + 3$, $T(2) = k + 6$). Segundo, adivinar la forma del resultado (e.g.: $T(n) = an + b$, para algunos a y b todavía desconocidos). Tercero, reemplace esta forma en las ecuaciones. En nuestro ejemplo esto da $b = k$ y $(an + b) = 3 + (a.(n - 1) + b)$. Esto da $a = 3$, para un resultado final de $T(n) = 3n + k$. La técnica de tres etapas funciona si la forma adivinada es correcta.
- Una técnica mucho más poderosa, llamada funciones generadoras, la cual da un resultado cerrado o resultados asintóticos en una amplia variedad de casos sin tener que adivinar la forma del resultado. Se requiere algún conocimiento técnico de series infinitas y cálculo, pero no más de lo que se ve en un curso universitario inicial sobre estos temas. Vea Knuth [91] y Wilf [185], para consultar buenas introducciones a las funciones generadoras.

al que usamos para Append, podemos inferir que SumListas y Desplizq son $\mathcal{O}(n)$ donde n es la longitud de L. Esto nos lleva a la ecuación de recurrencia siguiente para la invocación recursiva:

$$T_{\text{PascalRápido}}(n) = k_1 + \max(k_2, k_3 + T_{\text{PascalRápido}}(n - 1) + k_4.n)$$

donde n es el valor del argumento n. Simplificando llegamos a

$$T_{\text{PascalRápido}}(n) = k_5 + k_4.n + T_{\text{PascalRápido}}(n - 1)$$

Para el caso base, tomamos n=1. Esto lleva a

$$T_{\text{PascalRápido}}(1) = k_6$$

Para resolver estas dos ecuaciones, primero “adivinamos” que la solución es de la forma:

$$T_{\text{PascalRápido}}(n) = a.n^2 + b.n + c$$

Esta adivinanza viene de un argumento intuitivo como el presentado en el capítulo 1. Ahora reemplazamos esta forma en las dos ecuaciones. Si podemos resolverlas exitosamente para a , b , c , entonces eso significa que nuestra adivinanza era correcta. Al reemplazar en las dos ecuaciones, obtenemos las tres ecuaciones siguientes en a , b , y c :

$$k_4 - 2a = 0$$

$$k_5 + a - b = 0$$

$$a + b + c - k_6 = 0$$

No tenemos que resolver este sistema completamente: es suficiente verificar que $a \neq 0$.¹⁸ Por lo tanto $T_{\text{PascalRápido}}(n)$ es $\Theta(n^2)$.

Ejemplo: MergeSort

En una sección anterior vimos tres versiones del algoritmo mergesort. Todas tienen el mismo tiempo de ejecución con diferente factor constante. Calculemos el tiempo de ejecución de la primera versión. Recordemos la definición:

```
fun {MergeSort xs}
  case xs
  of nil then nil
  [] [x] then [x]
  else Ys Zs in
    {Dividir xs Ys Zs}
    {Mezclar {MergeSort Ys} {MergeSort Zs}}
  end
end
```

Sea $T(n)$ el tiempo de ejecución de $\{\text{MergeSort } xs\}$, donde n es la longitud de xs . Suponga que `Dividir` y `Mezclar` son $\Theta(n)$ en el tamaño de sus entradas. Sabemos que `Dividir` devuelve dos listas de tamaños $\lceil n/2 \rceil$ y $\lfloor n/2 \rfloor$. A partir de la definición de `MergeSort`, definimos las ecuaciones de recurrencia siguientes:

$$T(0) = k_1$$

$$T(1) = k_2$$

$$T(n) = k_3 + k_4 n + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) \text{ if } n \geq 2$$

Se utilizan las funciones techo y piso, que son un poco complicadas. Para salir de ellas, suponga que n es una potencia de 2, i.e., $n = 2^k$ para algún k . Las ecuaciones se vuelven:

$$T(0) = k_1$$

$$T(1) = k_2$$

$$T(n) = k_3 + k_4 n + 2T(n/2) \text{ if } n \geq 2$$

Al expandir al última ecuación llegamos a (donde $L(n) = k_3 + k_4 n$):

$$T(n) = \overbrace{L(n) + 2L(n/2) + 4L(n/4) + \cdots + (n/2)L(2)}^k + 2T(1)$$

Reemplazando $L(n)$ y $T(1)$ por sus valores da

$$T(n) = \overbrace{(k_4 n + k_3) + (k_4 n + 2k_3) + (k_4 n + 4k_3) + \cdots + (k_4 n + (n/2)k_3)}^k + k_2$$

18. Si proponemos $a.n^2 + b.n + c$ y la solución real es de la forma $b.n + c$, entonces obtendremos $a = 0$.

Al realizar la suma llegamos a

$$T(n) = k_4 kn + (n - 1)k_3 + k_2$$

Concluimos que $T(n) = \mathcal{O}(n \log n)$. Para los valores de n que no son potencias de 2, usamos el hecho, demostrado, que $n \leq m \Rightarrow T(n) \leq T(m)$ y así mostramos que el límite \mathcal{O} sigue siendo válido. El límite es independiente del contenido de la lista de entrada. Esto significa que el límite $\mathcal{O}(n \log n)$ también es un límite en el peor caso.

3.5.2. Consumo de memoria

El consumo de memoria no es una figura tan sencilla como el tiempo de ejecución, pues consiste de dos aspectos bastante diferentes:

- El tamaño de la memoria activa instantánea $m_a(t)$, en palabras de memoria. Este número representa la cantidad de memoria que el programa necesita para seguir ejecutándose exitósamente. Un número relacionado es el tamaño máximo de memoria activa, $M_a(t) = \max_{0 \leq u \leq t} m_a(u)$. Este número es útil para calcular cuánta memoria física necesita su computador para ejecutar el programa exitósamente.
- El consumo de memoria instantánea $m_c(t)$, en palabras de memoria por segundo. Este número representa cuánta memoria demanda el programa durante su ejecución. Un valor grande para este número significa que el administrador de memoria tiene mucho trabajo para hacer, e.g., el recolector de basura será invocado más seguido. Un número relacionado es el consumo total de memoria, $M_c(t) = \int_0^t m_c(u) du$, el cual es una medida del trabajo total que el administrador de memoria debe realizar para ejecutar el programa.

No se deben confundir estos dos números. El primero es mucho más importante. Un programa puede solicitar memoria lentamente (e.g., 1 KB/s) y también tener una memoria activa grande (e.g., 100 MB); e.g., una gran base de datos en memoria que maneja sólo consultas sencillas. El opuesto también es posible. Un programa puede consumir memoria a un rata alta (e.g., 100 MB/s) y también tener una memoria activa pequeña ((e.g., 10 KB); e.g., un algoritmo de simulación corriendo en el modelo declarativo.¹⁹

El tamaño de la memoria activa instantánea

El tamaño de la memoria activa se puede calcular en cualquier momento de la ejecución siguiendo todas las referencias de la pila semántica en el almacén y totalizar el tamaño de todas las variables alcanzables y de los valores parciales. Esto es aproximadamente igual al tamaño de todas las estructuras de datos que el

19. Debido a este comportamiento, ¡el modelo declarativo no es bueno para ejecutar simulaciones a menos que tenga un excelente recolector de basura!

$\langle d \rangle ::=$	
skip	0
$\langle x \rangle_1 = \langle x \rangle_2$	0
$\langle x \rangle = \langle v \rangle$	tammem(v)
$\langle d \rangle_1 \langle d \rangle_2$	$M(d_1) + M(d_2)$
local $\langle x \rangle$ in $\langle d \rangle$ end	$1 + M(d)$
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	$\max(M(d_1), M(d_2))$
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	$\max(M(d_1), M(d_2))$
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	$M_x(\text{tam}_x(I_x(\{y_1, \dots, y_n\})))$

Tabla 3.6: Consumo de memoria de las instrucciones del núcleo.

programa necesita para su ejecución.

Consumo total de memoria

El consumo total de memoria se puede calcular con una técnica similar a la usada para el tiempo de ejecución. Cada operación del lenguaje núcleo tiene un consumo de memoria bien definido. En la tabla 3.6 se presenta el consumo de memoria $M(d)$ para cada declaración $\langle d \rangle$ del núcleo. Utilizando esta tabla se pueden construir las ecuaciones de recurrencia para el programa, a partir de las cuales se puede calcular el consumo total de memoria en función del tamaño de la entrada. A este número hay que añadirle el consumo de memoria de la pila semántica. Para la instrucción $\langle x \rangle = \langle v \rangle$ hay un caso especial en el cual el consumo de memoria es menor que $\text{tammem}(v)$, a saber, cuando $\langle x \rangle$ está parcialmente instanciado. En ese caso, sólo la memoria de las nuevas entidades se debe contar. La función $\text{tammem}(v)$ se define como sigue, de acuerdo al tipo y el valor de v :

- Para un entero: 0 para enteros pequeños, de otra forma, proporcional al tamaño del entero. Calcule el número de bits que se necesitan para representar el entero en complemento 2. Si este número es menor que 28, entonces 0. Sino divídalo por 32 y redondéelo al entero más cercano.
- Para un flotante: 2.
- Para un par: 2.
- Para una tupla o un registro: $1 + n$, donde $n = \text{length}(\text{arity}(v))$.
- Para un valor de tipo procedimiento: $k + n$, donde n es el número de referencias externas al cuerpo del procedimiento y k es una constante que depende de la implementación.

Todas las figuras están en número de palabras de memoria de 32-bit y son correctas para Mozart 1.3.0. Para valores anidados, tome la suma de todos los valores. Para valores de tipo registro o procedimiento hay un costo adicional por una sola vez.

Para cada aridad diferente de un registro el costo adicional es aproximadamente proporcional a n (porque la aridad es almacenada una vez en la tabla de símbolos). Para cada procedimiento distinto en el código fuente, el costo adicional depende del tamaño del código compilado, el cual es aproximadamente proporcional al número total de declaraciones e identificadores en el cuerpo del procedimiento. En la mayoría de los casos, estos costos por una sola vez agregan una constante al consumo total de memoria; para los cálculos ellos son ignorados normalmente.

3.5.3. Complejidad amortizada

Algunas veces no estamos interesados en la complejidad de una sola operación, sino en la complejidad total de una secuencia de operaciones. Siempre que la complejidad total sea razonable, puede no importarnos si una operación individual es a veces muy costosa. En la sección 3.4.5 se presenta un ejemplo con colas: siempre que una secuencia de n operaciones de inserción y eliminación tenga un tiempo de ejecución total de $\mathcal{O}(n)$, puede no importar si las operaciones en forma individual son siempre $\mathcal{O}(1)$. Se les permite ocasionalmente ser más costosas, siempre que no suceda con frecuencia. En general, si una secuencia de n operaciones tiene un tiempo total de ejecución de $\mathcal{O}(f(n))$, entonces decimos que tiene una complejidad amortizada de $\mathcal{O}(f(n)/n)$.

Complejidad amortizada versus complejidad en el peor caso

En muchos dominios de aplicación, tener una buena complejidad amortizada es suficientemente bueno. Sin embargo, hay tres dominios de aplicación que necesitan garantías en el tiempo de ejecución de las operaciones individuales. Ellos son los sistemas duros en tiempo real, los sistemas paralelos, y los sistemas interactivos de alto desempeño.

Un sistema duro en tiempo real tiene que satisfacer plazos estrictos para completar los cálculos. Incumplir un plazo puede tener consecuencias graves, incluyendo pérdidas de vidas. Tales sistemas existen, e.g., en los marcapasos y en los sistemas para evitar la colisión de trenes (ver también sección 4.6.1).

Un sistema paralelo ejecuta varios cálculos simultáneamente para lograr acelerar toda una computación. Con frecuencia, la computación como un todo sólo puede avanzar después que todos los cálculos simultáneos se completen. Si uno de estos cálculos toma ocasionalmente mucho más tiempo, entonces la computación completa se ve frenada.

Un sistema interactivo, tal como un juego de computador, debe tener un tiempo de reacción uniforme. Por ejemplo, si un juego de acción multiusuarios retrasa su reacción a una entrada del jugador, entonces la satisfacción del jugador se verá bastante reducida.

El método del banquero y el método del físico

Calcular la complejidad amortizada es un poco más difícil que calcular la complejidad en el peor caso. (Y será aún más difícil cuando introduzcamos ejecución perezosa en la sección 4.5.) Básicamente hay dos métodos, llamados el método del banquero y el método del físico.

El método del banquero cuenta créditos, donde un “crédito” representa una unidad de tiempo de ejecución o de espacio de memoria. Cada operación ahorra o reserva unos créditos. Una operación costosa se permite si se han reservado suficientes créditos para cubrir su ejecución.

El método del físico está basado en encontrar una función potencial, una especie de “altura sobre el nivel del mar”. Cada operación cambia el potencial, i.e., el potencial sube o baja un poco. El costo de cada operación es el cambio en el potencial, a saber, cuánto sube o baja. La complejidad total es una función de la diferencia entre los potenciales inicial y final. Mientras esta diferencia permanezca pequeña, se permiten grandes variaciones en el intermedio.

Para mayor información sobre estos métodos y muchos ejemplos de su uso con algoritmos declarativos, recomendamos el libro escrito por Okasaki [129].

3.5.4. Reflexiones sobre desempeño

Siempre, desde el principio de la era del computador en los años 1940, tanto el tiempo como el espacio se han vuelto más baratos a una tasa exponencial (un factor constante de mejoría cada año). Actualmente, ambos son muy baratos, tanto en términos absolutos como en términos de percepción: un computador personal de bajo costo en el año 2003 tiene típicamente 512MB de RAM y 80GB de almacenamiento en disco con un reloj a una frecuencia de 2 GHz. Esto provee un desempeño máximo sostenido de alrededor de un billón de instrucciones por segundo, donde cada instrucción puede realizar una operación completa de 64 bits, incluyendo las de punto flotante.

Esto es significativamente más grande que un supercomputador Cray-1, el computador más rápido del mundo en 1975. Un supercomputador se define como uno de los computadores más rápidos en un momento particular del tiempo. El primer Cray-1 tuvo una frecuencia del reloj de 80 MHz y podía realizar varias operaciones de punto flotante de 64 bits por ciclo [159]. A costo constante, el desempeño de los computadores personales todavía mejora a una tasa exponencial (se dobla cada cuatro años), y se predice que continuará así al menos durante la primera década del siglo XXI. Esto cumple la ley de Moore.

La ley de Moore

La ley de Moore establece que la densidad de los circuitos integrados se doble aproximadamente cada 18 meses. Esto fue observado primero por Gordon Moore alrededor de 1965 y se ha seguido cumpliendo hasta el día de hoy. Los expertos

creen que esta ley puede desacelerarse pero continuará cumpliéndose substancialmente por otras dos décadas al menos. Una malinterpretación común de la ley de Moore original, es que el desempeño de los computadores se dobla cada dos años aproximadamente. Esta también parece ser substancialmente correcta.

La ley de Moore observa solamente un período pequeño comparado con el tiempo en el cual se ha realizado computación basada en máquinas. Desde el siglo XIX, ha habido al menos cinco tecnologías utilizadas para computación, incluyendo la mecánica, la electromecánica, los tubos al vacío, los transistores discretos, y los circuitos integrados. A través de este largo período se ha visto que el crecimiento del poder de computación siempre ha sido exponencial. De acuerdo a Raymond Kurzweil, quien ha estudiado este patrón de crecimiento, la próxima tecnología será la computación molecular tridimensional [98].²⁰

Debido a esta situación, el desempeño no es, normalmente, un aspecto crítico. Si su problema es tratable, i.e., si existe un algoritmo eficiente para resolverlo, entonces si usted usa buenas técnicas de diseño de algoritmos, el tiempo y el espacio reales, que el algoritmo consuma, serán casi siempre aceptables. En otras palabras, dada una complejidad asintótica razonable de un programa, el factor constante casi nunca es crítico. Esto es cierto incluso para la mayoría de las aplicaciones multimedia (las cuales usan video y audio) debido a las excelentes bibliotecas gráficas que hay.

Problemas intratables

Sin embargo, no todos los problemas son tratables. Hay muchos problemas que son computacionalmente costosos, e.g., en las áreas de optimización combinatoria, investigación de operaciones, computación y simulación científica, aprendizaje de máquinas, reconocimiento de voz y visual, y computación gráfica. Algunos de estos problemas son costosos simplemente porque tienen que hacer una gran cantidad de trabajo. Por ejemplo, los juegos con gráficas reales, los cuales, por definición, están siempre en la frontera de lo posible. Otros problemas son costosos por razones más fundamentales. Por ejemplo, los problemas NP-completos. Estos problemas están en la clase NP, i.e., dada una solución al problema, es fácil comprobar que es correcta.²¹ Pero encontrar una solución puede ser mucho más difícil. Un ejemplo sencillo es el problema de satisfactibilidad de circuitos: ¿Dado un circuito digital combinatorio, el cual consta de compuertas And, Or, y Not, existe un conjunto de valores de entrada que produzca como salida 1? Este problema es NP-completo [39]. Informalmente, un problema NP-completo es un tipo especial de problema NP con la propiedad que si se logra resolver uno de ellos en tiempo polinomial, entonces se pueden resolver todos en tiempo polinomial. Muchos científicos de la computación han tratado, durante varias décadas, de encontrar soluciones en tiempo polinomial a problemas NP-completos, y ninguno ha tenido éxito. Por ello, la mayoría de los

20. Kurzweil afirma que la tasa de crecimiento se está incrementando y llegará a una “singularidad” en algún momento cerca a la mitad del siglo XXI.

21. NP es el acrónimo de “nondeterministic polynomial time”, en inglés.

Técnicas de programación declarativa

científicos de la computación creen que ningún problema NP-completo se puede resolver en tiempo polinomial. En este libro, no hablaremos más sobre problemas computacionalmente costosos. Como nuestro propósito es mostrar cómo programar, entonces nos autolimitamos a trabajar con problemas tratables.

Optimización

En algunos casos, el desempeño de un programa puede ser insuficiente, aún si el problema es teóricamente tratable. Entonces el programa tiene que ser reescrito para mejorar su desempeño. A la tarea de reescribir un programa para mejorarle alguna característica se le llama optimización, aunque el programa nunca es “óptimo” en ningún sentido matemático. Normalmente, un programa se puede mejorar fácilmente hasta cierto punto, a partir del cual el programa es cada vez más complejo y las mejoras cada vez menos importantes. Por tanto, la optimización no debe realizarse a menos que sea necesario. La optimización prematura es la ruina de la computación.

La optimización tiene un lado bueno y un lado malo. El lado bueno es que el tiempo global de ejecución de la mayoría de las aplicaciones es ampliamente determinado por una muy pequeña parte del texto del programa. Por lo tanto, la optimización del desempeño, si es necesaria, siempre se puede lograr reescribiendo, solamente, esta pequeña parte (algunas veces unas pocas líneas son suficientes). El lado malo es que normalmente no es obvio, ni siquiera para programadores expertos, cuál es esa parte a priori. Por tanto, esta parte se debe identificar una vez la aplicación se está ejecutando y sólo si se presenta un problema de este tipo. Si no hay ningún problema, entonces no se debe realizar ninguna optimización del desempeño. La mejor técnica para identificar los “puntos calientes” es el perfilamiento, la cual instrumenta la aplicación para medir sus características de tiempo de ejecución.

Reducir el espacio utilizado por un programa es más fácil que reducir el tiempo de ejecución. El espacio global que utiliza un programa depende de la representación de datos escogida. Si el espacio es un aspecto crítico, entonces una buena técnica es utilizar un algoritmo de compresión sobre los datos cuando estos no hacen parte de una computación inmediata. Esto negocia espacio por tiempo.

3.6. Programación de alto orden

La programación de alto orden es la colección de técnicas de programación disponibles cuando se usan valores de tipo procedimiento en los programas. Los valores de tipo procedimiento se conocen también como clausuras de alcance léxico. El término alto orden viene del concepto de *orden* de un procedimiento. Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden. Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden. Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden. Y así sucesivamente:

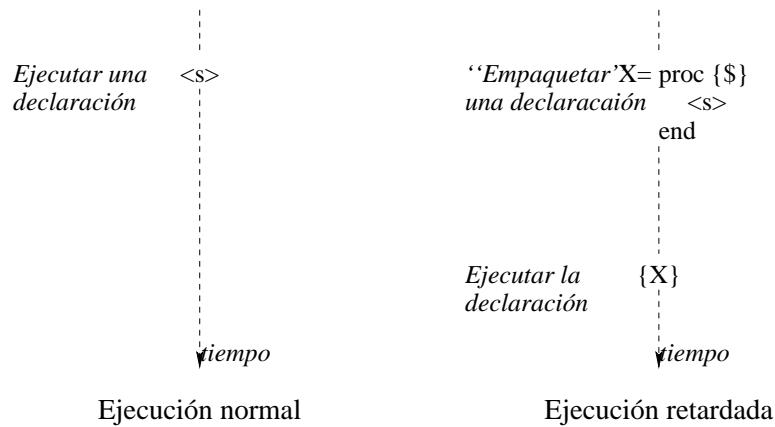


Figura 3.21: Ejecución retardada de un valor de tipo procedimiento.

un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto. La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden. Un lenguaje que permite esto se llama un lenguaje de alto orden.

3.6.1. Operaciones básicas

Existen cuatro operaciones básicas que subyacen a todas las técnicas de programación de alto orden:

1. *Abstracción procedimental*: la capacidad de convertir cualquier declaración en un valor de tipo procedimiento.
2. *Genericidad*: la capacidad de pasar un valor de tipo procedimiento como argumento en una invocación a un procedimiento.
3. *Instanciación*: la capacidad de devolver valores de tipo procedimiento como resultado de una invocación a un procedimiento.
4. *Embebimiento*: la capacidad de colocar valores de tipo procedimiento dentro de estructuras de datos.

Primero, examinemos cada una de estas operaciones en ese orden. Posteriormente, miraremos técnicas más sofisticadas, tales como las abstracciones de ciclos, las cuales usan estas operaciones básicas.

Abstracción procedimental

Ya hemos introducido la abstracción procedimental. Recordemos brevemente la idea central. Cualquier declaración $\langle d \rangle$ puede ser “empaquetada” dentro de un

Técnicas de programación declarativa

procedimiento escribiéndola como **proc** { $\$$ } $\langle d \rangle$ **end**. Esto no ejecuta la declaración, pero en su lugar crea un valor de tipo procedimiento (una clausura). Como el valor de tipo procedimiento contiene un ambiente contextual, ejecutarlo produce exactamente el mismo resultado que ejecutar $\langle d \rangle$. La decisión de ejecutar o no la declaración no se toma donde se define la declaración, sino en otro lugar en el programa. En la figura 3.21 se muestran las dos posibilidades: ejecutar $\langle d \rangle$ inmediatamente o de manera retardada.

Los valores de tipo procedimiento permiten hacer mucho más que solamente retardar la ejecución de una declaración. Ellos pueden tener argumentos, los cuales permiten que algo de su comportamiento sea influenciado por la invocación. Como lo veremos a lo largo del libro, la abstracción procedural es enormemente poderosa. Ella subyace a la programación de alto orden y a la programación orientada a objetos, y es la principal herramienta para construir abstracciones. Presentemos otro ejemplo de abstracción procedural. Considere la declaración:

```
local A=1.0 B=3.0 C=2.0 D SolReal X1 X2 in
  D=B*B-4.0*A*C
  if D>=0.0 then
    SolReal=true
    X1=(~B+{Raíz D})/(2.0*A)
    X2=(~B-{Raíz D})/(2.0*A)
  else
    SolReal=false
    X1=~B/(2.0*A)
    X2={Raíz ~D}/(2.0*A)
  end
  {Browse SolReal#X1#X2}
end
```

Este programa calcula las soluciones de la ecuación cuadrática $x^2 + 3x + 2 = 0$. Se utiliza la fórmula cuadrática $(-b \pm \sqrt{b^2 - 4ac})/2a$, con la cual se calculan las dos soluciones de la ecuación $ax^2 + bx + c = 0$. El valor $d = b^2 - 4ac$ se llama el discriminante. Si este es positivo, entonces hay dos soluciones reales diferentes. Si es cero, entonces hay dos soluciones reales idénticas. En cualquier otro caso, las dos soluciones son números complejos conjugados. La declaración anterior se puede convertir en un procedimiento usándola como el cuerpo de una definición de procedimiento y pasando los identificadores libres como argumentos:

```
proc {EcuaciónCuadrática A B C ?SolReal ?X1 ?X2}
      D=B*B-4.0*A*C
  in
    if D>=0.0 then
      SolReal=true
      X1=(~B+{Raíz D})/(2.0*A)
      X2=(~B-{Raíz D})/(2.0*A)
    else
      SolReal=false
      X1=~B/(2.0*A)
      X2={Raíz ~D}/(2.0*A)
    end
  end
```

Este procedimiento resuelve cualquier ecuación cuadrática. Sólo se invoca con los coeficientes de la ecuación como argumentos:

```
declare RS X1 X2 in
{EcuaciónCuadrática 1.0 3.0 2.0 RS X1 X2}
{Browse RS#X1#X2}
```

Formas restringidas de abstracción procedural

Muchos lenguajes interactivos antiguos tienen una forma restringida de abstracción procedural. Para entender esto, damos una mirada a Pascal y C [82, 88]. En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas). Esto significa que sólo puede existir un valor de tipo procedimiento correspondiente a cada definición de procedimiento. En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance. Estas restricciones imposibilitan en general “empaquetar” una declaración y ejecutarla en otro lugar.

Esto significa que muchas técnicas de programación de alto orden son imposibles. Por ejemplo, es imposible programar abstracciones de control nuevas. En su lugar, Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones). Unas pocas técnicas de programación de alto orden son aún posibles. Por ejemplo, el ejemplo de la ecuación cuadrática funciona porque no tiene referencias externas: se puede definir como un procedimiento global en Pascal y C. Las operaciones genéricas también funcionan frecuentemente por la misma razón (ver a continuación).

Las restricciones de Pascal y C son una consecuencia de la forma en que estos lenguajes realizan la administración de la memoria. En ambos lenguajes, la implementación coloca parte del almacén en la pila semántica. Normalmente, esta parte del almacén guarda las variables locales. La asignación de espacio se realiza utilizando una disciplina de pila. Por ejemplo, a algunas variables locales se les asigna espacio a la entrada al procedimiento, el cual devuelven a la salida correspondiente. Esta es una forma de administración automática de la memoria mucho más sencilla que implementar la recolección de basura. Desafortunadamente, se crean fácilmen-

te referencias sueltas. Es extremadamente difícil depurar un programa grande que se comporta ocasionalmente de manera incorrecta debido a una referencia suelta.

Ahora podemos explicar las restricciones. Tanto en Pascal como en C, crear un valor de tipo procedimiento está restringido de manera que el ambiente contextual no tenga nunca referencias sueltas. Hay algunas técnicas, específicas de un lenguaje, que se pueden usar para aliviar esta restricción. Por ejemplo, en lenguajes orientados a objetos, tales como C++ o Java, es posible que los objetos jueguen el papel de los valores de tipo procedimiento. Esta técnica se explica en el capítulo 7.

Genericidad

Ya hemos visto un ejemplo de programación de alto orden en una sección anterior. Fue introducido tan elegantemente que tal vez usted no haya notado que se estaba haciendo programación de alto orden. Fue con la abstracción de control `Iterar` de la sección 3.2.4, la cual usa dos argumentos de tipo procedimiento, `Transformar` y `EsFinal`.

Hacer una función genérica es convertir una entidad específica (i.e., una operación o un valor) en el cuerpo de la función, en un argumento de la misma. Decimos que la entidad ha sido abstraída hacia afuera del cuerpo de la función. La entidad específica se recibe como entrada cuando se invoca la función. Cada vez que se invoca la función, se envía una entidad específica (la misma u otra).

Miremos un segundo ejemplo de función genérica. Considere la función `SumList`:

```
fun {SumList L}
  case L
  of nil then 0
    [] X|L1 then X+{SumList L1}
  end
end
```

Esta función tiene dos entidades específicas: el número cero (0) y la operación de adición (+). El cero es el elemento neutro de la operación de adición. Estas dos entidades puede ser abstraídas hacia afuera. Cualquier elemento neutro y cualquier operación son posibles. Los pasamos como parámetros. Esto lleva a la siguiente función genérica:

```
fun {FoldR L F U}
  case L
  of nil then U
    [] X|L1 then {F X {FoldR L1 F U}}
  end
end
```

Esta función se suele llamar `FoldR`²² porque asocia a la derecha. Podemos definir `SumList` como un caso especial de `FoldR`:

22. La función `{FoldR L F U}` hace lo siguiente:

`{F X1 {F X2 {F X3 ... {F Xn U} ...}}}`

```
fun {SumList L}
  {FoldR L fun {$ x y} x+y end 0}
end
```

Podemos usar `FoldR` para definir otras funciones sobre listas. Por ejemplo, la función para calcular el producto de todos los elementos de una lista:

```
fun {ProductList L}
  {FoldR L fun {$ x y} x*y end 1}
end
```

La siguiente función devuelve `true` si existe al menos un elemento `true` en la lista:

```
fun {Some L}
  {FoldR L fun {$ x y} x orelse y end false}
end
```

`FoldR` es un ejemplo de abstracción de ciclos. En la sección 3.6.2 se presentan otros tipos de abstracciones de ciclos.

Un ejemplo: mergesort genérico

El algoritmo mergesort que vimos en la sección 3.4.2 está programado para usar `<` como la función de comparación. Podemos hacer un mergesort genérico pasando la función de comparación como un argumento. Cambiamos la función `Mezclar` para referenciar el argumento de tipo función `F` y la función `MergeSort` para referenciar el nuevo `Mezclar`:

Es decir, aplica `F`, función binaria, a cada elemento de la lista con el resultado precedente, comenzando con el último elemento de la lista y el neutro. Es como asociar la operación binaria `F` a la derecha. En inglés esta asociación la llaman Fold; la R de `FoldR` es por asociar a la derecha (*right*, en inglés).

Técnicas de programación declarativa

```

fun {MergeSortGenérico F Xs}
  fun {Mezclar Xs Ys}
    case Xs # Ys
      of nil # Ys then Ys
      [] Xs # nil then Xs
      [] (X|Xr) # (Y|Yr) then
        if {F X Y} then X|{Mezclar Xr Ys}
        else Y|{Mezclar Xs Yr} end
      end
    end
  fun {MergeSort Xs}
    case Xs
      of nil then nil
      [] [X] then [X]
      else Ys Zs in
        {Dividir Xs Ys Zs}
        {Mezclar {MergeSort Ys} {MergeSort Zs}}
      end
    end
  end
in
  {MergeSort Xs}
end

```

Aquí se usa la primera versión de `Dividir`. Hemos colocado las definiciones de `Mezclar` y `MergeSort` dentro de la nueva función `MergeSortGenérico`. Esto evita tener que pasar la función `F` como un argumento a `Mezclar` y `MergeSort`. En su lugar, los dos procedimientos se definen una vez por cada invocación a `MergeSortGenérico`. Podemos definir el mergesort original en términos de `MergeSortGenérico`:

```

fun {MergeSort Xs}
  {MergeSortGenérico fun {$ A B} A<B end xs}
end

```

En lugar de `fun` {\$ A B} A<B `end`, podíamos haber escrito `Number.<` pues la comparación `<` hace parte del módulo `Number`.

Instanciación

Un ejemplo de instanciaión es una función `CrearOrdenamiento` que devuelve una función de ordenamiento. Funciones como `CrearOrdenamiento` se suelen llamar “fábricas” o “generadoras.” `CrearOrdenamiento` toma una función booleana de comparación `F` y devuelve una función de ordenamiento que utiliza a `F` como función de comparación. Miremos cómo construir `CrearOrdenamiento` a partir de una función genérica de ordenamiento `Ordenar`. Suponga que `Ordenar` recibe dos entradas, una lista `L` y una función booleana `F`, y devuelve una lista ordenada. Ahora podemos definir `CrearOrdenamiento`:

```

proc {For A B S P}
  proc {CicloAsc C}
    if C=<B then {P C} {CicloAsc C+S} end
  end
  proc {CicloDesc C}
    if C>=B then {P C} {CicloDesc C+S} end
  end
in
  if S>0 then {CicloAsc A} end
  if S<0 then {CicloDesc A} end
end

```

Figura 3.22: Definición de un ciclo sobre enteros.

```

fun {CrearOrdenamiento F}
  fun {$ L}
    {Ordenar L F}
  end
end

```

Podemos ver a `CrearOrdenamiento` como la especificación de un conjunto de posibles rutinas de ordenamiento. Invocar `CrearOrdenamiento` instancia la especificación, es decir, devuelve un elemento de ese conjunto, el cual decimos que es una instancia de la especificación.

Embebimiento

Los valores de tipo procedimiento se pueden colocar en estructuras de datos. Esto tiene bastantes usos:

- *Evaluación perezosa explícita*, también llamada *evaluacion retardada*. La idea es no construir una estructura de datos en un solo paso, sino construirla por demanda. Se construye sólo una pequeña estructura de datos con procedimientos en los extremos que puedan ser invocados para producir más pedazos de la estructura. Por ejemplo, al consumidor de la estructura de datos se le entrega una pareja: una parte de la estructura de datos y una función nueva para calcular otra pareja. Esto significa que el consumidor puede controlar explícitamente qué cantidad de la estructura de datos se evalúa.
- *Módulos*. Un módulo es un registro que agrupa un conjunto de operaciones relacionadas.
- *Componentes de Software*. Un componente de *software* es un procedimiento genérico que recibe un conjunto de módulos como argumentos de entrada y devuelve un nuevo módulo. Esto puede ser visto como especificar un módulo en términos de los módulos que necesita (ver sección 6.7).

```

proc {ForAll L P}
  case L
  of nil then skip
  [] X|L2 then
    {P X}
    {ForAll L2 P}
  end
end

```

Figura 3.23: Definición de un ciclo sobre listas.

Ciclo sobre enteros

```

{For A B S P}
  {P A      }
  {P A+S   }
  {P A+2*S}
  :
  {P A+n*S}

```

(si $S > 0$: tantas veces como $A + n * S = < B$)
 (si $S < 0$: tantas veces como $A + n * S >= B$)

Ciclo sobre listas

```

{ForAll L P}
  {P X1}
  {P X2}
  {P X3}
  :
  {P Xn}

```

(donde $L = [X_1 \ X_2 \ \dots \ X_n]$)

Figura 3.24: Ciclos sencillos sobre enteros y sobre listas.

3.6.2. Abstracciones de ciclos

Como se muestra en los ejemplos de las secciones previas, los ciclos en el modelo declarativo tienden a ser pródigos en palabras porque necesitan invocaciones recursivas explícitas. Los ciclos se pueden volver más concisos si se les define como abstracciones. Existen diferentes tipos de ciclos que podemos definir. En esta sección, definiremos primero ciclos “para”,²³ sencillos, sobre enteros y sobre listas, y luego les agregaremos acumuladores para volverlos más útiles.

Ciclo sobre enteros

Definamos un ciclo sobre enteros, i.e., un ciclo que repite una operación con una secuencia de enteros. El procedimiento {For A B S P} invoca {P I} para los

23. Nota del traductor: *for-loops*, en inglés.

```

proc {ForAcc A B S P In ?Out}
  proc {CicloAsc C In ?Out}
    Mid in
      if C=<B then {P In C Mid} {CicloAsc C+S Mid Out}
      else In=Out end
    end
  proc {CicloDesc C In ?Out}
    Mid in
      if C>=B then {P In C Mid} {CicloDesc C+S Mid Out}
      else In=Out end
    end
  in
    if S>0 then {CicloAsc A In Out} end
    if S<0 then {CicloDesc A In Out} end
  end

proc {ForAllAcc L P In ?Out}
  case L
  of nil then In=Out
  [] X|L2 then Mid in
    {P In X Mid}
    {ForAllAcc L2 P Mid Out}
  end
end

```

Figura 3.25: Definición de ciclos de acumulación.

enteros I , comenzando por A y continuando hasta B , en pasos de S . Por ejemplo, ejecutar $\{For\ 1\ 10\ 1\ Browse\}$ muestra los enteros $1, 2, \dots, 10$ en el browser. Ejecutar $\{For\ 10\ 1\ ^2\ Browse\}$ muestra $10, 8, 6, 4, 2$. El ciclo For se define en la figura 3.22. Esta definición funciona para pasos tanto positivos como negativos. Se utiliza $CicloAsc$ para S positivo y $CicloDesc$ para S negativo. Gracias al alcance léxico, $CicloAsc$ y $CicloDesc$ sólo necesitan un argumento. Ellos ven B , S , y P como referencias externas.

Ciclo sobre listas

Definamos un ciclo sobre listas, i.e., un ciclo que repite una operación para todos los elementos de una lista. El procedimiento $\{ForAll\ L\ P\}$ invoca $\{P\ X\}$ para todos los elementos X de la lista L . Por ejemplo, $\{ForAll\ [a\ b\ c]\ Browse\}$ muestra a, b, c en el browser. El ciclo $ForAll$ se define en la figura 3.23. En la figura 3.24 se comparan For y $ForAll$ de manera gráfica.

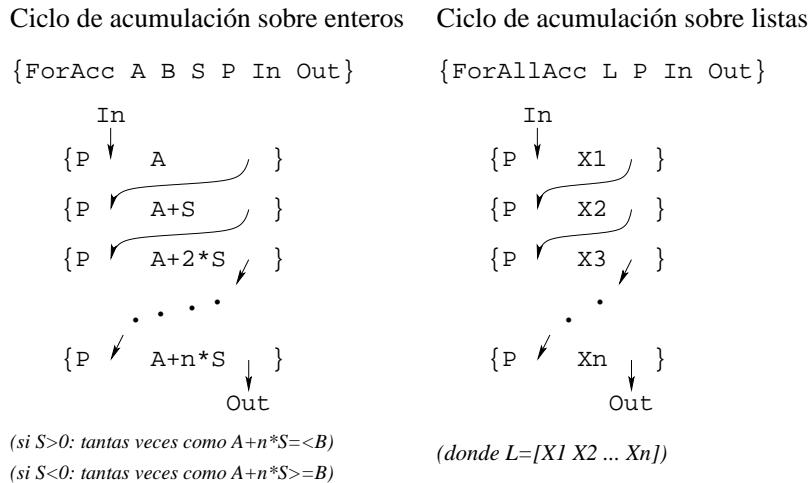


Figura 3.26: Ciclos de acumulación sobre enteros y sobre listas.

Ciclos de acumulación

Los ciclos `For` y `ForAll` sólo repiten una acción sobre diferentes argumentos, pero no calculan ningún resultado. Esto los vuelve poco útiles en el modelo declarativo. Ellos mostrarán su importancia sólo en el modelo con estado del capítulo 6. Para ser útiles en el modelo declarativo, los ciclos tienen que ser extendidos con un acumulador. De esta manera, ellos podrán calcular un resultado. En la figura 3.25 se definen `ForAcc` y `ForAllAcc`, los cuales extienden `For` y `ForAll` con un acumulador.²⁴ `ForAcc` y `ForAllAcc` son los caballos de batalla del modelo declarativo. Ambos se definen con una variable `Mid` que se usa para pasar el estado actual del acumulador al resto del ciclo. En la figura 3.26 se comparan `ForAcc` y `ForAllAcc` de manera gráfica.

Plegando una lista

Existe otra forma de mirar los ciclos de acumulación sobre listas. Estos ciclos se pueden ver como una operación de “plegado” sobre una lista, donde plegar significa insertar un operador infijo entre los elementos de una lista. Considere la lista $l = [x_1 \ x_2 \ x_3 \ \dots \ x_n]$. Plegar l con el operador infijo f produce

$$x_1 \ f \ x_2 \ f \ x_3 \ f \ \dots \ f \ x_n$$

24. En el sistema Mozart, `ForAcc` y `ForAllAcc` se llaman `ForThread` y `FoldL`, respectivamente.

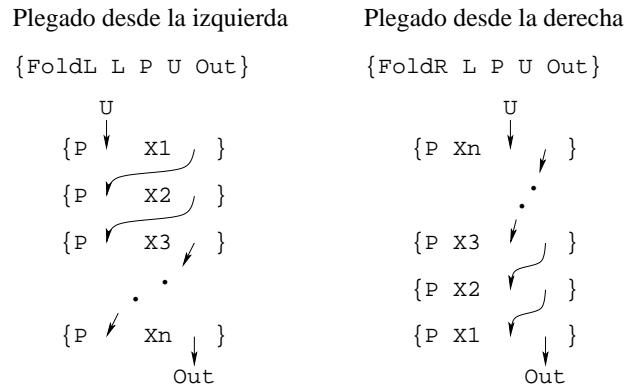


Figura 3.27: Plegando una lista.

Para calcular esta expresión sin ambigüedad tenemos que agregar paréntesis, para lo cual hay dos posibilidades. Podemos hacer primero las operaciones que están más hacia la izquierda (asociar a la izquierda):

$$((\dots((x_1 f x_2) f x_3) f \dots x_{n-1}) f x_n)$$

o hacer primero las operaciones que están más hacia la derecha (asociar a la derecha):

$$(x_1 f (x_2 f (x_3 f \dots (x_{n-1} f x_n) \dots)))$$

Para darle un toque final, modificaremos ligeramente estas expresiones de manera que cada aplicación de f envuelva solamente un nuevo elemento de l . Así es más fácil calcular y razonar con estas expresiones. Para hacerlo, agregamos un elemento neutro u . Esto nos lleva a las siguientes dos expresiones:

$$((\dots(((u f x_1) f x_2) f x_3) f \dots x_{n-1}) f x_n)$$

$$(x_1 f (x_2 f (x_3 f \dots (x_{n-1} f (x_n f u)) \dots)))$$

Para calcular estas expresiones definimos las dos funciones `{FoldL L F U}` y `{FoldR L F U}`. La función `{FoldL L F U}` hace lo siguiente:

$$\{F \dots \{F \{F \{F U X1\} X2\} X3\} \dots Xn\}$$

La función `{FoldR L F U}` hace lo siguiente:

$$\{F X1 \{F X2 \{F X3 \dots \{F Xn U\} \dots\}\}\}$$

En la figura 3.27 se muestran `FoldL` y `FoldR` de forma gráfica. Podemos relacionar `FoldL` y `FoldR` con los ciclos de acumulación que vimos anteriormente. Al comparar la figura 3.26 con la figura 3.27, vemos que `FoldL` es sólo otro nombre para `ForAllAcc`.

Definiciones iterativas de la operación de plegado

En la figura 3.25 se define `ForAllAcc` iterativamente, y por tanto también se define `FoldL`. Esta es la misma definición pero en notación funcional:

```
fun {FoldL L F U}
  case L
  of nil then U
  [] X|L2 then
    {FoldL L2 F {F U X}}
  end
end
```

Esta notación es más concisa que la definición procedimental pero oculta el acumulador, lo cual oscurece su relación con los otros tipos de ciclos. Ser conciso no es siempre una buena cosa.

¿Qué pasa con `FoldR`? La discusión sobre genericidad en la sección 3.6.1 presenta una definición recursiva, y no una iterativa. A primera vista, no parece fácil definir `FoldR` iterativamente. ¿Puede usted escribir una definición iterativa de `FoldR`? La forma de hacerlo es definir un estado intermedio y una función de transformación de estados. Mirando la expresión presentada atrás: ¿Cuál es el estado intermedio? ¿Cómo pasa al siguiente estado? Antes de mirar la respuesta, le sugerimos cerrar el libro y tratar de definir una versión iterativa de `FoldR`. Esta es una alternativa de definición:

```
fun {FoldR L F U}
  fun {Ciclo L U}
    case L
    of nil then U
    [] X|L2 then
      {Ciclo L2 {F X U}}
    end
  end
  end
  in
  {Ciclo {Reverse L} U}
  end
```

Como `FoldR` comienza calculando con x_n , el último elemento de L , la idea es iterar sobre el inverso de la lista L . Ya hemos visto cómo definir la inversa de manera iterativa.

3.6.3. Soporte lingüístico para los ciclos

Como los ciclos son tan útiles, se convierten en candidatos perfectos para convertirlos en abstracciones lingüísticas. En esta sección se define el ciclo declarativo `for`, el cual es una forma de hacer esto. El ciclo `for` está definido como parte del sistema Mozart [47]. El ciclo `for` está fuertemente relacionado con las abstracciones de ciclos de la sección precedente. Utilizar los ciclos `for` es, frecuentemente, más fácil que utilizar abstracciones de ciclos. Cuando escribimos ciclos recomendamos ensayar primero con ciclos `for`.

Iterando sobre enteros

Una operación común es iterar para enteros sucesivos desde un límite inferior i hasta un límite superior j . Sin la sintaxis de ciclo, la forma declarativa estándar de hacer esto utiliza la abstracción {For A B S P}:

```
{For A B S proc {$ i} <d> end}
```

Esto es equivalente al ciclo **for** siguiente:

```
for i in A..B do <d> end
```

cuando el paso S es 1, o:

```
for i in A..B;S do <d> end
```

cuando S es diferente de 1. El ciclo **for** declara el contador de ciclo i , el cual es una variable cuyo alcance se extiende al cuerpo del ciclo $\langle d \rangle$.

Ciclos declarativos versus imperativos

Existe una diferencia fundamental entre un ciclo declarativo y uno imperativo, i.e., un ciclo en un lenguaje imperativo como C o Java. En el último, el contador del ciclo es una variable assignable a la cual se le asigna un valor diferente en cada iteración. El ciclo declarativo es bastante diferente: en cada iteración declara una nueva variable. Todas estas variables son referenciadas por el mismo identificador. No es asignación destructiva en absoluto. Esta diferencia puede tener grandes consecuencias. Por ejemplo, las iteraciones de un ciclo declarativo son completamente independientes las unas de las otras. Por lo tanto es posible ejecutarlas concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for i in A..B do thread <d> end end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de i . Colocar $\langle d \rangle$ dentro de la declaración **thread** ... **end** la ejecuta como una actividad independiente. Este es un ejemplo de concurrencia declarativa, el cual es el tema del capítulo 4. Hacer esto en un ciclo imperativo ocasionaría estragos pues ninguna iteración puede estar segura de acceder al valor correcto de i . Los incrementos del contador del ciclo no seguirían sincronizados con las iteraciones.

Iterando sobre listas

El ciclo **for** se puede extender para iterar sobre listas así como sobre intervalos de enteros. Por ejemplo, la invocación

```
{ForAll L proc {$ x} <d> end end}
```

es equivalente a

```
for x in L do ⟨d⟩ end
```

Así como con ForAll, la lista puede ser un flujo de elementos.

Patrones

El ciclo **for** se puede extender para contener patrones que implícitamente declaran variables. Por ejemplo, si los elementos de L son tripletas de la forma `obj(nombre:N precio:P coordenadas:C)`, entonces podemos iterar sobre ellas de la siguiente manera:

```
for obj(nombre:N precio:P coordenadas:C) in L do
    if P<1000 then {Browse N} end
end
```

En cada iteración, se declaran y ligan las variables nuevas N, P, y C. Su alcance llega hasta el cuerpo del ciclo.

Recoleciendo resultados

Una extensión útil del ciclo **for** es permitir la recolección de resultados. Por ejemplo, hagamos una lista de todos los enteros entre 1 y 1000 que no son múltiplos ni de 2 ni de 3:

```
L=for I in 1..1000 collect:C do
    if I mod 2 \= 0 andthen I mod 3 \= 0 then {C I} end
end
```

El ciclo **for** es una expresión que devuelve una lista. La declaración “`collect:C`” define un procedimiento de recolección C que se puede utilizar en cualquier parte del cuerpo del ciclo. El procedimiento de recolección utiliza un acumulador para recolectar los elementos. Este ejemplo es equivalente a

```
{ForAcc 1 1000 1
  proc {$ ?L1 I L2}
    if I mod 2 \= 0 andthen I mod 3 \= 0 then L1=I|L2
    else L1=L2 end
  end
  L nil}
```

En general, el ciclo **for** es más expresivo que esto, pues el procedimiento de recolección puede ser invocado a cualquier profundidad dentro de ciclos anidados y otros procedimientos sin tener que pasar el acumulado explícitamente. A continuación un ejemplo con dos ciclos anidados:

```
L=for I in 1..1000 collect:C do
    if I mod 2 \= 0 andthen I mod 3 \= 0 then
        for J in 2..10 do
            if I mod J == 0 then {C I#J} end
        end
    end
end
```

¿Cómo hace el ciclo **for** para lograr esto sin pasar el acumulador? Utiliza estado explícito, como se vará más tarde en el capítulo 6.

Otras extensiones útiles

Los ejemplos anteriores presentan algunas de las variaciones de ciclos más utilizadas en una sintaxis declarativa de ciclo. Muchas otras variaciones son posibles. Por ejemplo, salir inmediatamente del ciclo (**break**), salir inmediatamente y devolver un resultado explícito (**return**), continuar inmediatamente con la iteración siguiente (**continue**), iteradores múltiples que avanzan de manera estandarizada, y otros procedimientos de recolección (e.g., **append** y **prepend** para listas y **sum** y **maximize** para enteros) [47]. Para otros diseños de ejemplos de ciclos declarativos recomendamos estudiar el ciclo macro de Common Lisp [162] y el paquete de hilos de estado de SICStus Prolog [84].

3.6.4. Técnicas dirigidas por los datos

Una tarea común consiste en realizar alguna operación sobre una gran estructura de datos, recorrer la estructura y calcular alguna otra estructura de datos con base en el recorrido. Esta idea se utiliza muy frecuentemente con listas y árboles.

3.6.4.1. Técnicas basadas en listas

La programación de alto orden se utiliza frecuentemente junto con las listas. Algunas de las abstracciones de ciclos pueden ser vistas de esta forma, e.g., **FoldL** y **FoldR**. Miremos otras técnicas basadas en listas.

Una operación común sobre listas es **Map**, la cual calcula una lista nueva a partir de una lista antigua, aplicándole una función a cada elemento de esa lista. Por ejemplo, **{Map [1 2 3] fun {\$ I} I*I end}** devuelve **[1 4 9]**. **Map** se puede definir así:

```
fun {Map Xs F}
  case Xs
  of nil then nil
  [] X|Xr then {F X}|{Map Xr F}
  end
end
```

Su tipo es **<fun {\$ <Lista T>} <fun {\$ T>: U>}: <Lista U>>**. **Map** se puede definir con **FoldR**. La lista de salida se construye usando el acumulador de **FoldR**:

```
fun {Map Xs F}
  {FoldR Xs fun {$ I A} {F I}|A end nil}
end
```

¿Qué pasaría si usáramos **FoldL** en lugar de **FoldR**? Otra operación común sobre listas es **Filter**, la cual aplica una función booleana a cada elemento de la lista y devuelve la lista de todos los elementos para los que esta aplicación da **true**. Por ejemplo, **{Filter [1 2 3 4] fun {\$ A B} A<3 end}** returns **[1 2]**. **Filter** se

define así:

```
fun {Filter Xs F}
  case Xs
  of nil then nil
  [] X|Xr andthen {F X} then X|{Filter Xr F}
  [] X|Xr then {Filter Xr F}
  end
end
```

Su tipo es $\langle \text{fun } \{ \$ \langle \text{Lista } T \rangle \langle \text{fun } \{ \$ T T \}: \langle \text{bool} \rangle \}: \langle \text{Lista } T \rangle \rangle$. `Filter` también se puede definir con `FoldR`:

```
fun {Filter Xs F}
  {FoldR Xs fun {\$ I A} if {F I} then I|A else A end end nil}
end
```

Tal parece que `FoldR` es una función sorprendentemente versátil. ¡Esto no debería ser una sorpresa, pues `FoldR` es sencillamente un ciclo “para” con un acumulador! La misma función `FoldR` puede ser implementada en términos del iterador genérico `Iterar` de la sección 3.2:

```
fun {FoldR Xs F U}
  {Iterar
    {Reverse Xs}#U
    fun {\$ S} Xr#A=S in Xr==nil end
    fun {\$ S} Xr#A=S in Xr.2#{F Xr.1 A} end}.2
  end
```

Como `Iterar` es un ciclo “mientras que” con acumulador, se convierte en la abstracción más versátil entre todas ellas. Todas las otras abstracciones de ciclos se pueden programar en términos de `Iterar`. Por ejemplo, para programar `FoldR` sólo tenemos que codificar el estado en la forma correcta con la función de terminación correcta. Aquí codificamos el estado como una pareja `Xr#A`, donde `Xr` es la parte aún no utilizada de la lista de entrada y `A` es el resultado acumulado de `FoldR`. Observe los detalles: la invocación inicial de `Reverse` y el `.2` al final para devolver el resultado acumulado.

3.6.4.2. Técnicas basadas en árboles

Como vimos en la sección 3.4.6 y en otras partes, una operación común sobre un árbol es visitar todos sus nodos en un orden particular y realizar ciertas operaciones mientras se visitan los nodos. Por ejemplo, el generador de código mencionado en la sección 3.4.8 tiene que recorrer los nodos del árbol de sintaxis abstracta para generar código de máquina. El programa para dibujar árboles de la sección 3.4.7, después de calcular las posiciones de los nodos, tiene que recorrer los nodos en orden para dibujarlos. Las técnicas de alto orden se pueden usar para ayudar en estos recorridos.

Consideremos árboles n-arios, los cuales son más generales que los árboles binarios que vimos antes. Un árbol n-ario se puede definir así:

$$\langle \text{Árbol } T \rangle ::= \text{árbol}(\text{nodo}:T \text{ hijos}:\langle \text{Lista } \langle \text{Árbol } T \rangle \rangle)$$

En este árbol, cada nodo puede tener cualquier número de hijos. El recorrido en profundidad de este árbol es tan sencillo como para los árboles binarios:

```
proc {DFS Arbol}
  case Arbol of árbol(nodo:N hijos:Hijos ...) then
    {Browse N}
    for T in Hijos do {DFS T} end
  end
end
```

Podemos “decorar” esta rutina para hacer algo en cada nodo que se visita. Por ejemplo, invoquemos {P T} en cada nodo T. Esto lleva al siguiente procedimiento genérico:

```
proc {VisitarNodos Arbol P}
  case Arbol of árbol(hijos:Hijos ...) then
    {P Arbol}
    for T in Hijos do {VisitarNodos T P} end
  end
end
```

Un recorrido ligeramente más complicado consiste en invocar {P Arbol T} para cada enlace padre-hijo entre un nodo padre Arbol y uno de sus hijos T:

```
proc {VisitarEnlaces Arbol P}
  case Arbol of árbol(hijos:Hijos ...) then
    for T in Hijos do {P Arbol T} {VisitarEnlaces T P} end
  end
end
```

Estos dos procedimientos genéricos fueron usados para dibujar los árboles en la sección 3.4.7 después que se calculaban las posiciones de los nodos. VisitarEnlaces dibujó las líneas entre los nodos y VisitarNodos dibujó los nodos.

Siguiendo el desarrollo de la sección 3.4.6, extendemos estos recorridos con un acumulador. Hay tantas formas de acumular como recorridos posibles. Las técnicas de acumulación pueden ser de arriba hacia abajo²⁵ (el resultado se calcula propagando del padre hacia sus hijos), de abajo hacia arriba²⁶ (de los hijos hacia el padre), o en cualquier otro orden (e.g., a lo ancho del árbol, para un recorrido en amplitud). Haciendo una analogía con el caso de las listas, la técnica de arriba hacia abajo es como FoldL y la técnica de abajo hacia arriba es como FoldR. Realicemos un cálculo de abajo hacia arriba. Primero calculamos un valor para cada nodo. El valor para un parente es función del nodo parente y de los valores de sus hijos. Hay dos funciones: LF, la operación binaria que se debe realizar entre todos los hijos de un parente dado, y TF, la operación binaria entre el nodo parente y el valor proveniente del cálculo sobre los hijos. Esto nos lleva a la función, genérica con acumulador, siguiente:

25. Nota del traductor: *Top-down*, en inglés.

26. Nota del traductor: *Bottom-up*, en inglés.

```
local
  fun {FoldTreeR Hijos TF LF U}
    case Hijos
    of nil then U
    [] S|Hijos2 then
      {LF {FoldTree S TF LF U} {FoldTreeR Hijos2 TF LF U}}
    end
  end
in
  fun {FoldTree Arbol TF LF U}
    case Arbol of árbol(nodo:N hijos:Hijos ...) then
      {TF N {FoldTreeR Hijos TF LF U}}
    end
  end
end
```

El siguiente es un ejemplo de invocación:

```
fun {Sum A B} A+B end
T=árbol(nodo:1
         hijos:[árbol(nodo:2 hijos:nil)
                 árbol(nodo:3 hijos:[árbol(nodo:4 hijos:nil)])))
{Browse {FoldTree T Sum Sum 0}}
```

Esto muestra 10, la suma de todos los valores de los nodos.

3.6.5. Evaluación perezosa explícita

Los lenguajes funcionales modernos tienen una estrategia de ejecución incorporada llamada evaluación perezosa. Aquí mostraremos cómo programar evaluación perezosa explícita con programación de alto orden. En la sección 4.5 se muestra cómo realizar evaluación perezosa implícita, i.e., donde la mecánica que desencadena la evaluación es manejada por el sistema. Como veremos en el capítulo 4, la evaluación perezosa implícita está íntimamente ligada con la concurrencia.

En la evaluación perezosa las estructuras de datos (como las listas) se construyen incrementalmente. El consumidor de la estructura de tipo lista solicita nuevos elementos de la lista cuando los necesita. Este es un ejemplo de ejecución dirigida por los datos, y es bien diferente de la evaluación normal, dirigida por la oferta, donde la lista se calcula completamente independientemente de saber si sus elementos se necesitan o no.

Para implementar la evaluación perezosa, el consumidor debe solicitar elementos nuevos. Una manera de hacerlo es por medio de lo que se llama un disparador programado. Hay dos maneras naturales de expresar disparadores programados en el modelo declarativo: como una variable de flujo de datos o con programación de alto orden. En la sección 4.3.3 se explica cómo hacerlo con una variable de flujo de datos. Aquí explicamos cómo hacerlo con programación de alto orden. El consumidor cuenta con una función a la cual invoca cada que necesita un elemento nuevo de la lista. La invocación a la función devuelve una pareja: el elemento de la lista y una función nueva. La función nueva es el disparador nuevo: invocarla

devuelve el siguiente elemento de la lista y otra función nueva. Y así sucesivamente.

3.6.6. Currificación

La currificación es una técnica que puede simplificar programas que usan intensamente la programación de alto orden. La idea consiste en escribir funciones de n argumentos como n funciones anidadas de un solo argumento. Por ejemplo, la función que calcula el máximo:

```
fun {Max X Y}
    if X>=Y then X else Y end
end
```

se reescribe así:

```
fun {Max X}
    fun {$ Y}
        if X>=Y then X else Y end
    end
end
```

Se conserva el mismo cuerpo de la función original, pero se invoca teniendo en cuenta la nueva definición: $\{\{Max\} 10\} 20$, devuelve 20. La ventaja de usar la currificación es que las funciones intermedias pueden ser útiles en sí mismas. Por ejemplo, la función $\{Max\} 10$ devuelve un resultado que nunca es menor que 10. A la función $\{Max\} 10$ se le llama función parcialmente aplicada. Incluso, podemos darle el nombre `CotaInferior10`:

```
CotaInferior10={Max 10}
```

En muchos lenguajes de programación funcional, particularmente en Standard ML y Haskell, todas las funciones están implícitamente currificadas. Para sacar la mayor ventaja posible del uso de la currificación, estos lenguajes la proveen con una sintaxis sencilla y un implementación eficiente. Ellos definen la sintaxis de manera que se puedan definir funciones currificadas sin necesidad de anidar ninguna clase de palabras reservadas, y que puedan ser invocadas sin paréntesis. Si la invocación `max 10 20` es posible, entonces `max 10` también lo es. La implementación se encarga que la currificación sean tan barata como sea posible. Esto no cuesta nada si no se usa y la construcción de funciones parcialmente aplicadas se evita cuando sea posible.

El modelo de computación declarativo de este capítulo, no tiene ningún soporte especial para la currificación. El sistema Mozart tampoco cuenta con un soporte, ni sintáctico ni de implementación, para soportar esto. La mayoría de los usos de currificación en Mozart son sencillos. Sin embargo, el uso intenso de la programación de alto orden tal como se realiza en los lenguajes funcionales, puede justificar que deba existir tal soporte. En Mozart, las funciones parcialmente definidas tienen que definirse explícitamente. Por ejemplo, la función `max 10` se puede definir así:

```
fun {CotaInferior10 Y}
    {Max 10 Y}
end
```

La definición de la función original no cambia, lo cual es eficiente en el modelo declarativo. Sólo las mismas funciones parcialmente aplicadas se vuelven más costosas.

3.7. Tipos abstractos de datos

Un tipo de datos o simplemente tipo, es un conjunto de valores junto con un conjunto de operaciones sobre estos valores. El modelo declarativo trae consigo un conjunto predefinido de tipos, llamados los tipos básicos (ver sección 2.3). Además de estos, el usuario tiene la libertad de definir tipos nuevos. Decimos que un tipo es abstracto si está completamente definido por su conjunto de operaciones, sin importar la implementación. El término “tipo abstracto de datos” se abrevia TAD. Utilizar un TAD significa que es posible cambiar la implementación del tipo sin cambiar su uso. Investiguemos cómo puede un usuario definir nuevos TADs.

3.7.1. Una pila declarativa

Para comenzar esta sección, presentaremos un ejemplo sencillo de un tipo abstracto de datos, una pila²⁷ $\langle \text{Pila } T \rangle$ cuyos elementos son de tipo T . Suponga que la pila tiene cuatro operaciones, con los tipos siguientes:

```
<fun {PilaNueva} : <Pila T>>
<fun {Colocar <Pila T> T} : <Pila T>>
<fun {Sacar <Pila T> T} : <Pila T>>
<fun {EsVacía <Pila T>} : <Bool>>
```

Este conjunto de operaciones y sus tipos definen la interfaz del TAD. Estas operaciones satisfacen ciertas leyes:

- $\{\text{EsVacía } \{\text{PilaNueva}\}\} = \text{true}$. Toda pila nueva es vacía.
- Para cualquier E y S_0 , se cumple que $S_1 = \{\text{Colocar } S_0 E\}$ y $S_0 = \{\text{Sacar } S_1 E\}$. Colocar un elemento y enseguida sacar uno produce como resultado la misma pila con la que se empezó.
- Para cualquier E no-ligado, $\{\text{Sacar } \{\text{PilaNueva}\} E\}$ lanza un error. No se puede sacar ningún elemento de una pila vacía.

Estas leyes son independientes de cualquier implementación particular, i.e., todas las implementaciones tienen que satisfacer estas leyes. A continuación una implementación de la pila que satisface las leyes:

27. Nota del traductor: *stack*, en inglés.

```
fun {PilaNueva} nil end
fun {Colocar S E} E|S end
fun {Sacar S E} case S of X|S1 then E=X S1 end end
fun {EsVacía S} S==nil end
```

Otra implementación que satisface las leyes es la siguiente:

```
fun {PilaNueva} pilaVacia end
fun {Colocar S E} pila(E S) end
fun {Sacar S E} case S of pila(X S1) then E=X S1 end end
fun {EsVacía S} S==pilaVacia end
```

Un programa que utiliza la pila funcionará con cualquiera de estas implementaciones. Esto es lo que queremos significar cuando decimos que la pila es abstracta.

Una mirada funcional

Los lectores atentos notarán un aspecto inusual de estas dos definiciones: ¡Sacar se escribe usando una sintaxis funcional, pero uno de sus argumentos es una salida! Podríamos haber escrito Sacar así:

```
fun {Sacar S} case S of X|S1 then X#S1 end end
```

en el cual se devuelven las dos salidas en una pareja, pero escogimos no hacerlo así. Escribir {Sacar S E} es un ejemplo de programación con una mirada funcional, la cual usa la sintaxis funcional para las operaciones, aunque no sean necesariamente funciones matemáticas. Consideramos que esto se justifica para los programas que tienen una clara direccionalidad en su flujo de datos. Puede ser interesante resaltar esta direccionalidad aún si el programa no es funcional. En algunos casos esto lleva a un programa más conciso y más legible. Sin embargo, la mirada funcional debe ser usada muy pocas veces, y solamente en los casos en que sea claro que la operación no es una función matemática. Usaremos la mirada funcional ocasionalmente a lo largo del libro, cuando lo consideremos apropiado.

Para el caso de la pila, la mirada funcional nos permite resaltar la simetría entre Colocar y Sacar. Sintácticamente queda claro que ambas operaciones toman una pila y devuelven una pila. Por tanto, e.g., la salida de Sacar puede ser inmediatamente enviada como entrada a Colocar, sin necesidad de una declaración **case** intermedia.

3.7.2. Un diccionario declarativo

Presentamos otro ejemplo, un TAD bastante útil llamado un diccionario. Un diccionario es una asociación finita de elementos de un conjunto de constantes simples con elementos de un conjunto de entidades del lenguaje. Cada constante se asocia a una entidad del lenguaje. Estas constantes se llaman llaves porque, de manera intuitiva, por medio de ellas se accede a la entidad. Usaremos átomos o enteros como constantes. Nos gustaría ser capaces de crear las asociaciones dinámicamente, i.e., agregar llaves nuevas durante la ejecución. Esto lleva al siguiente conjunto

```

fun {DiccionarioNuevo} nil end
fun {Aregar Ds Llave Valor}
  case Ds
  of nil then [Llave#Valor]
    [] (K#V)|Dr andthen K==Llave then
      (Llave#Valor) | Dr
    [] (K#V)|Dr andthen K>Llave then
      (Llave#Valor)|(K#V)|Dr
    [] (K#V)|Dr andthen K<Llave then
      (K#V)|{Aregar Dr Llave Valor}
  end
end
fun {ConsultarCond Ds Llave Defecto}
  case Ds
  of nil then Defecto
    [] (K#V)|Dr andthen K==Llave then
      V
    [] (K#V)|Dr andthen K>Llave then
      Defecto
    [] (K#V)|Dr andthen K<Llave then
      {ConsultarCond Dr Llave Defecto}
  end
end
fun {Dominio Ds}
  {Map Ds fun {$ K#_} K end}
end

```

Figura 3.28: Diccionario declarativo (con lista lineal).

de funciones básicas sobre el nuevo tipo $\langle \text{Dicc} \rangle$:

- $\langle \text{fun } \{\text{DiccionarioNuevo}\} : \langle \text{Dicc} \rangle \rangle$ devuelve un diccionario nuevo vacío.
- $\langle \text{fun } \{\text{Aregar } \langle \text{Dicc} \rangle \langle \text{LLave} \rangle \langle \text{Valor} \rangle\} : \langle \text{Dicc} \rangle \rangle$ recibe un diccionario de entrada y devuelve un diccionario nuevo correspondiente a agregarle la asociación $\langle \text{LLave} \rangle \rightarrow \langle \text{Valor} \rangle$ al diccionario de entrada. Si $\langle \text{LLave} \rangle$ ya existe, entonces en el nuevo diccionario se reemplaza su valor por $\langle \text{Valor} \rangle$.
- $\langle \text{fun } \{\text{Consultar } \langle \text{Dicc} \rangle \langle \text{LLave} \rangle\} : \langle \text{Valor} \rangle \rangle$ devuelve el valor correspondiente a la $\langle \text{LLave} \rangle$ en $\langle \text{Dicc} \rangle$. Si no hay ningún elemento con esa llave, entonces lanza una excepción.
- $\langle \text{fun } \{\text{Dominio } \langle \text{Dicc} \rangle\} : \langle \text{Lista } \langle \text{LLave} \rangle \rangle \rangle$ devuelve una lista de las llaves presentes en $\langle \text{Dicc} \rangle$.

Para este ejemplo definimos el tipo $\langle \text{LLave} \rangle$ como $\langle \text{Atom} \rangle \mid \langle \text{Ent} \rangle$.

```

fun {DiccionarioNuevo} hoja end
fun {Aregar Ds Llave Valor}
    % ... similar a Insertar
end
fun {ConsultarCond Ds Llave Defecto}
    % ... similar a Buscar
end
fun {Dominio Ds}
    proc {DominioD Ds ?S1 Sn}
        case Ds
            of hoja then
                S1=Sn
                [ ] tree(K _ L R) then S2 S3 in
                {DominioD L S1 S2}
                S2=K|S3
                {DominioD R S3 Sn}
            end
        end D
    in
    {DominioD Ds D nil} D
end

```

Figura 3.29: Diccionario declarativo (con árbol binario ordenado).

Implementación basada en listas

En la figura 3.28 se muestra una implementación en la cual el diccionario se representa como una lista de parejas Llave#Valor ordenada por la llave. En lugar de Consultar, definimos una operación de acceso ligeramente más general, ConsultarCond:

- $\langle \text{fun } \{\text{ConsultarCond } \langle \text{Dicc} \rangle \langle \text{LLave} \rangle \langle \text{Valor} \rangle_1\} : \langle \text{Valor} \rangle_2 \rangle$ devuelve el valor corresponde a $\langle \text{LLave} \rangle$ en $\langle \text{Dicc} \rangle$. Si $\langle \text{LLave} \rangle$ no se encuentra, entonces devuelve el último argumento.

ConsultarCond es casi tan fácil de implementar como Consultar y es muy útil como lo veremos en el próximo ejemplo.

Esta implementación es supremamente lenta para diccionarios grandes. Dada una distribución uniforme de las llaves, Agregar necesita mirar, en promedio, la mitad de la lista. ConsultarCond también necesita mirar, en promedio, la mitad de la lista, ya sea que el elemento esté presente o no. Vemos que el número de operaciones es $\mathcal{O}(n)$ para diccionarios con n llaves. Decimos entonces que la implementación hace una búsqueda lineal.

Implementación basada en árboles

Se puede conseguir una implementación más eficiente de diccionarios utilizando un árbol binario ordenado, tal como se definió en la sección 3.4.6. Agregar es sencillamente Insertar y ConsultarCond es muy parecido a Buscar. Esto nos lleva a la definición de la figura 3.29. En esta implementación, las operaciones Agregar y ConsultarCond consumen tiempo y espacio $O(\log n)$ para un árbol con n nodos, suponiendo que el árbol esté “razonablemente balanceado.” Es decir, para cada nodo, los tamaños de los hijos izquierdo y derecho no deben ser “muy diferentes.”

Implementación con estado

Podemos hacer algo aún mejor que la implementación basada en árboles dejando el modelo declarativo atrás y utilizando estado explícito (ver sección 6.5.1). Esto nos lleva a un diccionario con estado, de un tipo ligeramente diferente al diccionario declarativo, pero con la misma funcionalidad. Utilizar estado es una ventaja porque disminuye el tiempo de ejecución de las operaciones Agregar y ConsultarCond hasta tiempo constante amortizado.

3.7.3. Una aplicación sobre frecuencia de palabras

Para comparar nuestras tres implementaciones de diccionario, las usaremos en una aplicación sencilla. Escribimos un programa para contar la frecuencia con que aparecen las palabras en una cadena. Más adelante, mostraremos cómo usar este programa para contar las palabras en un archivo. En la figura 3.30 se define la función FrecPalabras, la cual recibe una lista de caracteres Cs y devuelve una lista de parejas W#N, donde W es una palabra (una secuencia maximal de letras y dígitos) y N es el número de veces que la palabra aparece en Cs. La función FrecPalabras se define en términos de las funciones siguientes:

- {CarDePalabra C} devuelve **true** si y sólo si C es una letra o un dígito.
- {PalAAtomo PW} convierte una lista invertida de caracteres de una palabra en un átomo que contiene esos caracateres. La función StringToAtom se utiliza para crear el átomo.
- {IncPal D W} recibe un diccionario D y un átomo w. Devuelve un diccionario nuevo en el cual el campo W ha sido incremetnado en 1. Miren lo fácil que es escribir esta función con ConsultarCond, la cual tiene cuidado del caso en que W no haga parte del diccionario.
- {CarsAPalabras nil Cs} recibe una lista de caracteres Cs y devuelve una lista de átomos correspondientes a las palabras representadas en Cs. La función Char.toLower se usa para convertir letras mayúsculas a minúsculas, de manera que “El” y “el” se consideran la misma palabra.

```
fun {CarDePalabra C}
    (&a=<C andthen C=<&z) orelse
    (&A=<C andthen C=<&Z) orelse (&0=<C andthen C=<&9)
end

fun {PalAAtom PW}
    {StringToAtom {Reverse PW}}
end

fun {IncPal D W}
    {Agregar D W {ConsultarCond D W 0}+1}
end

fun {CarsAPalabras PW Cs}
    case Cs
    of nil andthen PW==nil then
        nil
    [] nil then
        [{PalAAtom PW}]
    [] C|Cr andthen {CarDePalabra C} then
        {CarsAPalabras {Char.toLowerCase C}|PW Cr}
    [] C|Cr andthen PW==nil then
        {CarsAPalabras nil Cr}
    [] C|Cr then
        {PalAAtom PW}|{CarsAPalabras nil Cr}
    end
end

fun {ContarPalabras D Ws}
    case Ws
    of W|Wr then {ContarPalabras {IncPal D W} Wr}
    [] nil then D
    end
end

fun {FrecPalabras Cs}
    {ContarPalabras {DiccionarioNuevo} {CarsAPalabras nil Cs}}
end
```

Figura 3.30: Frecuencia de palabras (con diccionario declarativo).

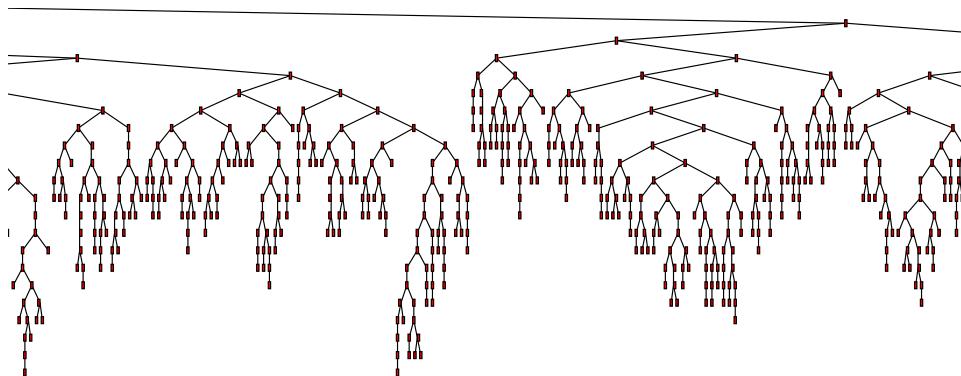


Figura 3.31: Estructura interna del árbol binario que representa el diccionario en `FrecPalabras` (una parte).

- `{ContarPalabras D Ws}` recibe un diccionario vacío y la salida de `CarsAPalabras`. Devuelve un diccionario en el cual cada llave tiene asociado el número de veces que la palabra aparece.

A continuación una ejecución de prueba. La entrada siguiente:

```
declare
T="Oh my darling, oh my darling, oh my darling Clementine.
She is lost and gone forever, oh my darling Clementine."
{Browse {FrecPalabras T}}
```

muestra este conteo de frecuencias de palabras:

```
[she#1 is#1 clementine#2 lost#1 my#4 darling#4 gone#1 and#1
oh#4 forever#1]
```

Hemos ejecutado `FrecPalabras` sobre un texto más sustancial, a saber, sobre un primer borrador del libro. El texto contiene 712626 caracteres, para un total de 110457 palabras de las cuales 5561 son diferentes. Hemos ejecutado `FrecPalabras` con tres implementaciones de diccionarios: utilizando listas (ver ejemplo anterior), utilizando árboles binarios (ver sección 3.7.2), y utilizando estado (la implementación predefinida de diccionarios; ver sección 6.8.2). En la figura 3.31 se muestra parte de la estructura interna del árbol binario que representa el diccionario, dibujado con el algoritmo de la sección 3.8.1. El código que medimos está en la sección 3.8.1. Al ejecutarlo se obtuvieron los tiempos siguientes (error de 10%)²⁸:

28. Utilizando Mozart 1.1.0 sobre un procesador Pentium III a 500 MHz.

Implementación de diccionario	Tiempo de ejecución	Complejidad en tiempo
Con listas	620 segundos	$\mathcal{O}(n)$
Con árboles binarios ordenados	8 segundos	$\mathcal{O}(\log n)$
Con estado	2 segundos	$\mathcal{O}(1)$

El tiempo medido es el utilizado para hacerlo todo, i.e., leer el archivo de texto, ejecutar `FrecPalabras`, y escribir en un archivo las cuentas de cada palabra. La diferencia entre los tiempos se debe por completo a las diferentes implementaciones del diccionario. Comparar los tiempos es un buen ejemplo del efecto práctico que puede tener el usar diferentes implementaciones de un tipo de datos importante. La complejidad muestra cómo el tiempo para insertar o buscar un ítem depende del tamaño del diccionario.

3.7.4. Tipos abstractos de datos seguros

Tanto en el tipo de datos pila como en el tipo de datos diccionario, la representación interna de los valores es visible a los usuarios del tipo. Si los usuarios son programadores disciplinados, entonces esto no será una causa de problemas. Pero éste no es siempre el caso. Un usuario puede estar tentado a mirar dentro de la representación o aún más, a construir valores de esa representación, sin utilizar las operaciones del tipo.

Por ejemplo, un usuario del tipo pila puede usar la operación `Length` para ver cuántos elementos hay en la pila, si la pila ha sido implementada como una lista. La tentación de hacer esto puede ser muy fuerte si no hay otra manera de calcular el tamaño de una pila. Otra tentación es manipular los contenidos de las pilas. Como cualquier lista es un valor legal de pila, el usuario puede construir valores de pila nuevos, e.g., eliminando o añadiendo elementos.

En breve, cualquier usuario puede agregar operaciones nuevas sobre las pilas en cualquier parte en el programa. Esto significa que la implementación del tipo pila queda potencialmente esparsa por todo el programa en lugar de quedar limitada a una pequeña parte de él. Esto es desastroso por dos razones:

- El programa es mucho más difícil de mantener. Por ejemplo, suponga que queremos mejorar la eficiencia de un diccionario reemplazando la implementación basada en listas por una implementación basada en árboles. Tendríamos que revisar el programa completo para encontrar qué partes dependen de que la implementación esté basada en listas. Existe también un problema de confinamiento del error: si el programa tiene errores en una parte, entonces estos pueden propagarse dentro de los TADs volviéndolos incorrectos a ellos también, lo cual contamina entonces otras partes del programa.
- El programa es susceptible de interferencia maliciosa. Este es un problema más sutil y tiene que ver con seguridad. Esto no pasa con programas escritos por gente que confía en los otros. Esto ocurre sobre todo con programas abiertos. Un programa

abierto es aquel que puede interactuar con otros programas que sólo se conocen en tiempo de ejecución. ¿Qué pasa si el otro programa es malicioso y desea interrumpir la ejecución del programa abierto? Debido a la evolución de Internet la proporción de programas abiertos se está incrementando.²⁹

¿Cómo resolver estos problemas? La idea básica es proteger la representación interna de los valores de los TADs. e.g., los valores de tipo pila, de la interferencia no autorizada. El valor que se quiere proteger se coloca dentro de una frontera de protección. Hay dos formas de usar esta frontera:

- **Valor estacionario.** El valor nunca sale de la frontera. Un conjunto bien definido de operaciones puede ingresar por la frontera para calcular con el valor. El resultado del cálculo se queda dentro de la frontera.
- **Valor móvil.** El valor puede salir e ingresar por la frontera. Cuando está afuera, las operaciones se pueden realizar sobre el valor. Las operaciones con autorización adecuada pueden tomar el valor afuera de la frontera y calcular con él. El resultado es colocado luego dentro de la frontera.

Con cualquiera de esas soluciones, razonar sobre la implementación del tipo es más sencillo. En lugar de mirar el programa completo, necesitamos mirar sólo cómo las operaciones del tipo se implementan.

La primera solución es como la banca computarizada. Cada cliente tiene una cuenta con alguna cantidad de dinero. Un cliente puede realizar una transacción que transfiera dinero de su cuenta a otra cuenta. Como algunos clientes normalmente nunca van al banco, el dinero nunca deja el banco. La segunda solución es como una cajilla de seguridad. En esta se almacena el dinero y sólo puede ser abierta por los clientes que tengan la llave. Cada cliente puede tomar dinero de la cajilla o colocar más dinero en ella. Una vez sacado, el cliente puede entregar el dinero a otro cliente. Pero mientras el dinero esté en la cajilla, está seguro.

En la sección siguiente construiremos un TAD seguro utilizando la segunda solución. Esta forma es la más fácil de entender para el modelo declarativo. La autorización que necesitamos para entrar la frontera de protección es una clase de “llave.”

La agregamos como un nuevo concepto al modelo declarativo, y lo llamamos un nombre. En la sección 3.7.7 se explica que una llave es un ejemplo de una idea de seguridad mucho más general, llamada capacidad. En el capítulo 6, en la sección 6.4 se completa la historia sobre TADs seguros mostrando cómo implementar la primera solución y explicando el efecto del estado explícito sobre la seguridad.

29. En el capítulo 11 (en CTM) se muestra cómo escribir programas abiertos en Mozart.

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Aplicación de procedimiento
try $\langle d \rangle_1$ catch $\langle x \rangle$ then $\langle d \rangle_2$ end	Contexto para excepción
raise $\langle x \rangle$ end	Lanzamiento de excepción
$\{ \text{NewName } \langle x \rangle \}$	Creación de nombre
$\langle y \rangle = ! ! \langle x \rangle$	Vista de sólo lectura

Tabla 3.7: El lenguaje núcleo declarativo con tipos seguros.

3.7.5. El modelo declarativo con tipos seguros

El modelo declarativo construido hasta aquí no nos permite construir una frontera de protección. Para hacerlo, necesitamos extender el modelo. Necesitamos dos extensiones: una para proteger valores y otra para proteger variables no ligadas. En la tabla 3.7 se muestra el lenguaje núcleo resultante con estas dos operaciones nuevas, las cuales explicamos ahora.

Protección de valores

Una manera de crear valores seguros es agregando una operación de “envoltura” con una “llave.” Es decir, la representación interna se coloca (envuelve) dentro de una estructura de datos accesible sólo a aquellos que poseen un valor especial, la llave. Conocer la llave permite la creación de envolturas nuevas y mirar el interior de envolturas existentes creadas con la misma llave.

Implementamos esto con un tipo básico nuevo llamado nombre. Un nombre es una constante, como un átomo, salvo que tiene un conjunto de operaciones mucho más restringido. En particular, los nombres no tienen una representación textual: ellos no se pueden imprimir o digitar en un teclado. A diferencia de los átomos, no es posible realizar conversiones entre nombres y cadenas de caracteres. La única manera de conocer un nombre es pasando una referencia a él dentro de un programa. El tipo nombre consta sólo de dos operaciones:

Operación	Descripción
{NewName}	Devuelve un nombre fresco
N1==N2	Compara los nombres N1 y N2

Un nombre fresco es un nombre con la garantía de que es diferente de todos los otros nombres en el sistema. Los lectores atentos notarán que NewName no es declarativo pues invocarlo dos veces devuelve dos resultados diferentes. De hecho, la creación de nombres frescos es una operación con estado. Para garantizar la unicidad se necesita que NewName tenga alguna memoria interna. Sin embargo, si utilizamos NewName sólo para crear TADs declarativos seguros, entonces esto no es un problema. El tipo resultante sigue siendo declarativo.

Para volver un tipo de datos seguro, es suficiente colocarlo dentro de una estructura de datos que sólo pueda ser accedida a través de un nombre. Por ejemplo, tome el valor `s`:

```
s=[a b c]
```

`s` es un estado interno de la pila que definimos antes. Lo volvemos seguro de la manera siguiente:

```
Llave={NewName}  
SS={Chunk.new envolver(Llave:S)}
```

Primero creamos un nombre nuevo en `Llave`. Luego se crea una estructura especial `ss`, llamada un pedazo, que contiene a `s`, de manera que `s` sólo se puede extraer si se conoce `Llave`. Un pedazo es un registro limitado con una sola operación, la operación de selección “.” (ver apéndice B.4). Decimos que esto “envuelve” el valor `s` dentro de `ss`. Si uno conoce `Llave`, entonces acceder a `s` a partir de `ss` es fácil:

```
S=try SS.Llave catch _ then raise error(desenvolver(SS)) end end
```

Decimos que esto “desenvuelve” el valor `s` a partir de `ss`. Si uno no conoce `Llave`, desenvolverlo es imposible. No hay manera de conocer `Llave` si no es porque ha sido pasada explícitamente en el programa. Invocar `ss` con un argumento errado lanzará una excepción. La operación `try` asegura que la llave no se filtra a través de la excepción.

Un empacador

Podemos definir una abstracción de datos para realizar las operaciones mencionadas de envolver y desenvolver. La abstracción define dos operaciones, `Envolver` y `Desenvolver`. `Envolver` recibe cualquier valor y devuelve un valor protegido. `Desenvolver` recibe cualquier valor protegido y devuelve el valor original. Las operaciones `Envolver` y `Desenvolver` vienen en pareja. La única forma de desenvolver un valor envuelto es utilizando la operación correspondiente para desenvolver. Con los nombres podemos definir un procedimiento `NuevoEmpacador` que devuelve una nueva pareja `Envolver/Desenvolver`:

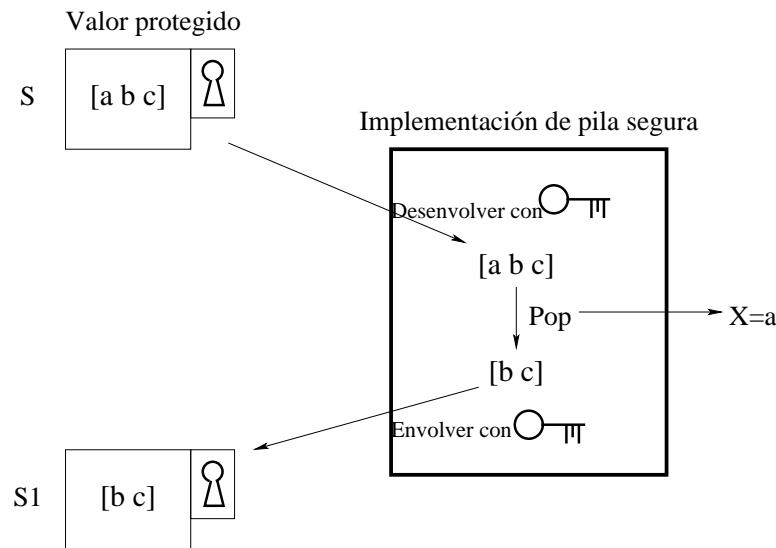


Figura 3.32: Realización de $S1 = \{ \text{Sacar } S \ X \}$ con una pila segura.

```

proc {NuevoEmpacador ?Envolver ?Desenvolver}
Llave={NewName} in
  fun {Envolver X}
    Chunk.new envolver(Llave:X)
  end
  fun {Desenvolver W}
    try W.Llave catch _ then raise error(desenvolver(W)) end
  end
end
end
end

```

Para máxima protección, cada TAD seguro que definamos usará su propia pareja Envolver/Desenvolver. Así cada uno está protegido de los otros, así como del programa principal. Dado el valor S como antes:

$S=[a \ b \ c]$

lo protegemos así:

$SS=\{\text{Envolver } S\}$

Podemos conseguir el valor original así:

$S=\{\text{Desenvolver } SS\}$

Una pila segura

Ahora podemos implementar una pila segura. La idea es desenvolver los valores entrantes y envolver los valores salientes. Para realizar una operación legal sobre un valor seguro del tipo, la rutina desenvuelve el valor seguro, realiza la operación deseada para calcular un valor nuevo, y entonces envuelve ese valor nuevo para garantizar seguridad. Esto nos lleva a la implementación siguiente:

```
local Envolver Desenvolver in
  {NuevoEmpacador Envolver Desenvolver}
  fun {PilaNueva} {Envolver nil} end
  fun {Colocar S E} {Envolver E|{Desenvolver S}} end
  fun {Sacar S E}
    case {Desenvolver S} of X|S1 then E=X {Envolver S1} end
  end
  fun {EsVacía S} {Desenvolver S}==nil end
end
```

En la figura 3.32 se ilustra la operación `Sacar`. La caja con el ojo de la cerradura representa un valor protegido. La llave representa el nombre, el cual es usado internamente por las operaciones `Envolver` y `Desenvolver` para asegurar o desasegurar una caja. El alcance léxico garantiza que las operaciones de envolver y desenvolver sólo sea posible realizarlas dentro de la implementación de la pila. A saber, los identificadores `Envolver` y `Desenvolver` sólo son visibles dentro de la declaración `local`. Por fuera de este alcance, estos identificadores están ocultos. Como `Desenvolver` está oculto, no existe ninguna manera de mirar dentro de un valor de tipo pila. Como `Envolver` está oculto, no hay ninguna manera de “falsificar” valores de tipo pila.

Protección de variables no-ligadas

Algunos veces es útil que un tipo de datos devuelva una variable no ligada. Por ejemplo, un flujo es una lista con una cola no-ligada. Nos gustaría que cualquiera sea capaz de leer el flujo pero que solamente la implementación del tipo de datos sea capaz de extenderlo. Usar las variables no-ligadas estándar, no funciona. Por ejemplo:

```
S=a | b | c | x
```

La variable `x` no es segura pues cualquiera que conozca `S` puede ligar `x`.

El problema es que cualquiera que tenga una referencia a una variable no ligada puede ligar la variable. Una solución es tener una versión restringida de la variable que sólo pueda ser leída, pero no ligada. Llamamos a esto una vista de sólo lectura de una variable. Extendemos el modelo declarativo con una función:

Operación	Descripción
<code>!!x</code>	Devuelve una vista de sólo lectura de <code>x</code>

Cualquier intento de ligar una vista de sólo lectura bloqueará la computación. Cualquier ligadura de x será transferida a la vista de sólo lectura. Para proteger un flujo, su cola debe ser una vista de sólo lectura.

En la máquina abstracta, las vistas de sólo lectura se colocan en un almacén nuevo llamado el almacén de sólo lectura. Modificamos la operación de ligadura de manera que antes de ligar una variable a un valor determinado. se compruebe si la variable está en el almacén de sólo lectura. De ser así, la operación de ligadura se suspende. Cuando la variable se determine, entonces la operación de ligadura puede continuar.

Creación de nombres frescos

Para concluir esta sección, miremos cómo crear nombres frescos en la implementación del modelo declarativo. ¿Cómo podemos garantizar que un nombre es globalmente único? Esto es fácil para programas que se ejecutan en un solo proceso: los nombres se pueden implementar como enteros consecutivos. Desafortunadamente, este enfoque falla para programas abiertos. En este caso, globalmente significa entre todos los programas que se están ejecutando en todos los computadores en el mundo. Hay básicamente dos enfoques para crear nombres que sean únicos globalmente:

1. El enfoque centralizado. Hay una fábrica de nombres en alguna parte en el mundo. Para conseguir un nombre fresco, se debe enviar un mensaje a esta fábrica y la respuesta contendrá un nombre fresco. La fábrica de nombres no tiene que estar físicamente en un lugar; puede estar espaciada sobre muchos computadores. Por ejemplo, el protocolo de Internet (IP) supone una única dirección IP para cada computador conectado a Internet en el mundo. Sin embargo, la dirección IP puede cambiar en el tiempo, e.g., si se realiza un traducción de la dirección de la red o si se usa el protocolo DHCP para asignación dinámica de direcciones IP. Por eso complementamos la dirección IP con un sello de alta resolución marcando el momento preciso de la creación del nombre nuevo. Esto nos lleva a una constante única que puede ser usada para implementar una fábrica local de nombres en cada computador.
2. El enfoque descentralizado. Un nombre fresco es sólo un vector de bits aleatorios. Los bits aleatorios se generan por un algoritmo que depende sobre información externa suficiente, de manera que computadores diferentes no generen el mismo vector. Si el vector es suficientemente grande, entonces la probabilidad de que los nombres no sean únicos es arbitrariamente pequeña. Teóricamente la probabilidad nunca es cero, per en la práctica esta técnica funciona bien.

Ahora que tenemos un nombre único, ¿cómo aseguramos que sea inexpugnable? Esto requiere técnicas de criptografía que están más allá del alcance del libro [153].

3.7.6. Un diccionario declarativo seguro

Ahora miremos cómo hacer un diccionario declarativo seguro. Es bastante fácil. Podemos usar la misma técnica que para la pila, a saber, usar una operación de envolver y una de desenvolver. Esta es la definición nueva:

```
local
    Envolver Desenvolver
    {NuevoEmpacador Envolver Desenvolver}
    % Definiciones previas:
    fun {DiccionarioNuevo2} ... end
    fun {Aregar2 Ds K Valor} ... end
    fun {ConsultarCond2 Ds K Defecto} ... end
    fun {Dominio2 Ds} ... end
in
    fun {DiccionarioNuevo}
        {Envolver {DiccionarioNuevo2}}
    end
    fun {Aregar Ds K Valor}
        {Envolver {Aregar2 {Desenvolver Ds} K Valor}}
    end
    fun {ConsultarCond Ds K Defecto}
        {ConsultarCond2 {Desenvolver Ds} K Defecto}
    end
    fun {Dominio Ds}
        {Dominio2 {Desenvolver Ds}}
    end
end
```

Como `Envolver` y `Desenvolver` sólo se conocen dentro del alcance de la declaración `local`, el diccionario envuelto no puede ser desenvuelto por nadie por fuera de este alcance. Esta técnica funciona para ambas implementaciones de diccionarios: la basada en listas y la basada en árboles.

3.7.7. Habilidades y seguridad

Decimos que una computación es segura si cuenta con propiedades bien definidas y controlables, independientemente de la existencia de otras (posiblemente maliciosas) entidades (computaciones o humanos) en el sistema [3]. Denominamos estas entidades “adversarios.” La seguridad permite la protección tanto de computaciones maliciosas como de computaciones inocentes (pero con errores). La propiedad de ser seguro es global; los “fallos” en un sistema pueden ocurrir a cualquier nivel, desde el *hardware* hasta el *software* y hasta la organización humana que alberga el sistema. Realizar un sistema de computación seguro no envuelve solamente la ciencia de la computación sino muchos otros aspectos de la sociedad humana [4].

Una descripción corta, precisa y concreta de cómo el sistema garantiza su seguridad se llama política de seguridad. Diseñar, implementar, y verificar las políticas de seguridad es crucial en la construcción de sistemas seguros, pero está fuera del alcance de este libro.

En esta sección, consideramos sólo una pequeña parte de la vasta disciplina de seguridad, a saber, el punto de vista del lenguaje de programación. Para implementar una política de seguridad, un sistema utiliza mecanismos de seguridad. A lo largo del libro, discutiremos mecanismos de seguridad que forman parte de un lenguaje de programación, tales como el alcance léxico y los nombres. Nos preguntaremos qué propiedades debe poseer un lenguaje para construir programas seguros, i.e., programas que puedan resistir ataques de adversarios que permanecen dentro del lenguaje.³⁰ Llamamos a un tal lenguaje un lenguaje seguro. Tener un lenguaje seguro es un requerimiento importante para construir programas de computador seguros. Diseñar e implementar un lenguaje seguro es un tema importante en la investigación en lenguajes de programación. Esto envuelve tanto propiedades semánticas como propiedades de la implementación.

Habilidades

Las técnicas de protección que hemos introducido para construir TADs seguros son casos especiales de un concepto de seguridad llamado una habilidad. Las habilidades están en el corazón de la investigación moderna sobre lenguajes seguros. Por ejemplo, el lenguaje seguro E refuerza las referencias a entidades del lenguaje de manera que se comportan como habilidades [115, 164]. Las parejas `Envolver/Desenvolver` que introdujimos previamente son llamadas parejas sellador/quita-sellos en E. En lugar de usar referencias externas para proteger valores, las parejas sellador/quita-sellos codifican y decodifican los valores. En esta vista, el nombre es usado como una llave de codificación y decodificación.

El concepto de habilidad fue inventado en los años 1960 en el contexto de diseño de sistemas operativos [44, 105]. Los sistemas operativos siempre han tenido que proteger a los usuarios de los otros usuarios permitiéndoles realizar su trabajo. Desde este trabajo inicial, ha quedado claro que el concepto hace parte del lenguaje de programación y es generalmente útil para construir programas seguros [116]. Las habilidades se pueden definir de muchas maneras, pero la definición siguiente es razonable para un lenguaje de programación. Una habilidad se define como una entidad inexpugnable del lenguaje que da a su propietario el derecho de realizar un conjunto de acciones. El conjunto de acciones se define dentro de la habilidad y puede cambiar en el tiempo. Por inexpugnable entendemos que no es posible para ninguna implementación, ni siquiera para una implementación íntimamente ligada a la arquitectura del *hardware* tal como las escritas en lenguaje ensamblador, crear una habilidad. En la literatura de E esta propiedad se resume en la frase “La conectividad genera conectividad”: la única forma de conseguir una habilidad es recibiéndola explícitamente a través de una habilidad existente [117].

30. La permanencia dentro del lenguaje se puede garantizar ejecutando siempre los programas sobre una máquina virtual que acepta solamente los binarios de programas legales.

Todos los valores de los tipos de datos son habilidades en este sentido, pues ellos dan a sus propietarios la habilidad de realizar todas las operaciones de ese tipo, pero no más. Un propietario de una entidad del lenguaje es cualquier fragmento de programa que refiera la entidad. Por ejemplo, un registro `R` da a su propietario la habilidad de realizar muchas operaciones, incluyendo la selección de un campo `R.F` y la aridad `{Arity R}`. Un procedimiento `P` da a su propietario la habilidad de invocar `P`. Un nombre da a su propietario la habilidad de comparar su valor con otros valores. Una variable no-ligada da a su propietario la habilidad de ligarla y de leer su valor. Una variable de sólo lectura da a su propietario la habilidad de leer su valor, pero no de ligarlo.

Se pueden definir habilidades nuevas durante la ejecución de un programa como instancias de abstracciones de datos como los TADs. Para los modelos del libro, la forma más simple es utilizar valores de tipo procedimiento. Una referencia a un valor de tipo procedimiento da a su propietario el derecho de invocar el procedimiento, i.e., hacer cualquier acción para la que el procedimiento fue diseñado. Además, una referencia a un procedimiento es inexpugnable. En un programa, la única forma de conocer la referencia es recibirla explícitamente. El procedimiento puede ocultar toda su información sensible en sus referencias externas. Para que esto funcione, el lenguaje debe garantizar que conocer un procedimiento no da automáticamente el derecho a examinar sus referencias externas!

Principio del menor privilegio

Un principio importante de diseño de sistemas seguros es el principio del menor privilegio: cada entidad debería tener la menor autoridad (o “privilegio”) posible para lograr realizar su trabajo. Este también se llama el principio de menor autoridad (POLA, por sus siglas en inglés). Determinar exactamente y en todos los casos cuál es la menor autoridad necesaria es un problema indecidible: no puede existir un algoritmo para resolverlo en todos los casos. Esto se debe a que la autoridad depende de lo que hace la entidad durante su ejecución. Si hubiera un tal algoritmo, sería suficientemente poderoso para resolver el problema de la parada, el cual ya se ha demostrado que es un problema indecidible.

En la práctica, no necesitamos conocer exactamente la menor autoridad. Una seguridad adecuada se puede lograr con aproximaciones. El lenguaje de programación debe facilitar el hacer estas aproximaciones. Las habilidades, como las definimos anteriormente, tienen esta capacidad. Con ellas, se facilita hacer la aproximación tan precisa como se necesite. Por ejemplo, una entidad puede tener la autoridad de crear un archivo con un nombre y un tamaño máximo en un directorio dado. Para el caso de los archivos, una granularidad no tan fina normalmente es suficiente, tal como la autoridad para crear un archivo en un directorio dado. Las habilidades pueden manejar fácilmente tanto el caso de granularidad fina como el caso de granularidad menos fina.

Habilidades y estado explícito

Las habilidades declarativas, i.e., aquellas escritas en un modelo de computación declarativo, adolecen de una propiedad crucial para volverlas útiles en la práctica. El conjunto de acciones que ellas autorizan no puede cambiar en el tiempo. En particular, ninguna de sus acciones puede ser revocada. Para que una habilidad sea revocable, el modelo de computación necesita un concepto adicional, a saber, estado explícito. Esto se explica en la sección 6.4.5.

3.8. Requerimientos no declarativos

La programación declarativa, debido a su visión “funcional pura” de la programación, se separa un tanto del mundo real, en el cual las entidades tienen memoria (estado) y pueden evolucionar independientemente y proactivamente (concurrentia). Para conectar un programa declarativo al mundo real, se necesitan algunas operaciones no declarativas. En esta sección se habla de dos clases de esas operaciones: entrada/salida de archivos, e interfaces gráficas de usuario. Una tercera clase de operaciones, compilación de aplicaciones autónomas, se presenta en la sección 3.9.

Más adelante veremos que las operaciones no declarativas de esta sección encajan dentro de unos modelos de computación más generales que el declarativo, en particular los modelos concurrente y con estado. En un sentido general, esta sección se liga con la discusión sobre los límites de la programación declarativa de la sección 4.8. Algunas de las operaciones manipulan un estado que es externo al programa; este es sólo un caso especial del principio de descomposición del sistema explicado en la sección 6.7.2.

Las operaciones nuevas que introducimos en esta sección vienen colecciónadas en módulos. Un módulo no es más que un registro que agrupa operaciones relacionadas. Por ejemplo, el módulo `List` agrupa muchas operaciones sobre listas, tales como `List.append` y `List.member` (las cuales también pueden ser referenciadas como `Append` y `Member`). En esta sección se introducen los tres módulos `File` (para entrada/salida de archivos de texto), `QTk` para interfaces gráficas de usuario, y `Pickle` (para entrada/salida de cualesquier tipo de valores). Algunos de estos módulos (como `Pickle`) son inmediatamente reconocidos por Mozart cuando arranca. Los otros módulos se pueden cargar invocando `Module.link`. En lo que sigue, mostramos cómo hacer esto para `File` y `QTk`. Información adicional sobre módulos y cómo usarlos, se presenta más adelante en la sección 3.9.

3.8.1. Entrada/Salida de texto por archivo

Una manera sencilla de hacer una interfaz entre la programación declarativa y el mundo real es por medio de archivos. Un archivo es una secuencia de valores almacenados fuera del programa en un medio de almacenamiento permanente como un disco duro. Un archivo de texto es un archivo que contiene una secuencia de

Técnicas de programación declarativa

caracteres. En esta sección mostramos cómo leer y escribir archivos de texto. Esto es suficiente para usar programas declarativos de forma práctica. El patrón básico de acceso es sencillo:

Archivo de entrada $\xrightarrow{\text{leer}}$ Aplicar función $\xrightarrow{\text{escribir}}$ Archivo de salida

Usamos el módulo `File`, el cual se encuentra en el sitio Web del libro. Más adelante haremos operaciones más sofisticadas sobre archivos; por ahora esto es suficiente.

Cargando el módulo File

El primer paso consiste en cargar el módulo `File` en el sistema, tal como se explica en el apéndice A.1.2. Suponemos que Ud. ha compilado una versión del módulo `File`, en el archivo `File.ozf`. Luego ejecuta lo siguiente:

```
declare [File]={Module.link ['File.ozf']}
```

Esto invoca `Module.link` con una lista de caminos a los módulos compilados. Aquí sólo hay uno. El módulo es cargado, enlazado en el sistema, inicializado, y ligado a la variable `File`.³¹ Ahora estamos listos para realizar operaciones sobre archivos.

Leyendo un archivo

La operación `File.readList` lee el contenido total del archivo y lo coloca en una cadena:

```
L={File.readList "foo.txt"}
```

En este ejemplo se lee el archivo `foo.txt` y se almacena en `L`. Esto también lo podemos escribir así:

```
L={File.readList 'foo.txt'}
```

Recuerde que `"foo.txt"` es una cadena (una lista de códigos de caracteres) y `'foo.txt'` es un átomo (una constante con una representación imprimible). El nombre del archivo se puede representar en ambas formas. Hay una tercera forma de representar nombres de archivos: como cadenas virtuales. Una cadena virtual es una tupla con etiqueta `'#'` que representa una cadena. En consecuencia, también hubiéramos podido escribir justo lo siguiente:

```
L={File.readList foo#'.'#txt}
```

La tupla `foo#'.'#txt`, que también podemos escribir como `'#'(foo '.' txt)`, representa la cadena `"foo.txt"`. Utilizando cadenas virtuales se evita la necesidad

31. Para ser precisos, el módulo se carga de manera perezosa: sólo se cargará realmente la primera vez que se use.

de hacer concatenaciones explícitas de cadenas. En Mozart, todas las operaciones predefinidas que esperan cadenas, también funcionan con cadenas virtuales. Las tres formas de cargar `foo.txt` tienen el mismo efecto, es decir, ligan `L` con una lista de códigos de caracteres que están en el archivo `foo.txt`.

Los archivos también pueden ser referenciados por URL. Un URL provee una dirección global conveniente para archivos, pues es ampliamente soportada a través de la infraestructura de la Web. Es tan fácil leer un archivo por medio de su URL como por su nombre:

```
L={File.readList `http://www.mozart-oz.org/features.html`}
```

Eso es todo lo que hay que hacer. Los URLs pueden ser usados para leer archivos, pero no para escribirlos. Esto es debido a que los URLs son manejados por servidores Web, los cuales normalmente son configurados para permitir sólo lecturas.

Mozart tiene otras operaciones que permiten leer un archivo ya sea incrementalmente o perezosamente, en lugar de todo de una vez. Esto es importante para archivos muy grandes que no caben dentro del espacio de memoria de los procesos en Mozart. Para conservar las cosas simples, por ahora, recomendamos que lea los archivos completos de una vez. Más adelante, veremos cómo leer un archivo incrementalmente.

Escribiendo un archivo

Normalmente, un archivo se escribe incrementalmente. añadiendo al archivo una cadena a la vez . El módulo `File` provee tres operaciones. `File.writeOpen` para abrir el archivo, la cual se debe realizar de primero; `File.write` para agregar una cadena al archivo; y `File.writeClose` para cerrar el archivo, lo cual debe realizarse de último. Un ejemplo es el siguiente:

```
{File.writeOpen `foo.txt`}
{File.write `Esto va en el archivo.\n`}
{File.write `El resultado de 43*43 es '#43*43#'.\n`}
{File.write "Las cadenas están bien.\n"}
{File.writeClose}
```

Después de estas operaciones, el archivo '`foo.txt`' tiene tres líneas de texto, a saber:

```
Esto va en el archivo.
El resultado de 43*43 is 1849.
Las cadenas están bien.
```

Ejemplo de ejecución

En la sección 3.7.3 definimos la función `FrecPalabras` que calcula la frecuencia con que se presentan las palabras en una cadena. Podemos usar esta función para calcular esas frecuencias y almacenarlas en un archivo:

```
% 1. Leer archivo de entrada
L={File.readList `libro.txt`}
% 2. Calcular la función
D={FrecPalabras L}
% 3. Escribir archivo de salida
{File.writeOpen `word.frec`}
for X in {Dominio D} do
    {File.write {Get D X}#` apariciones de la palabra '#X#\n`}
end
{File.writeClose}
```

En la sección 3.7.3 se presentan algunas figuras de este código utilizando diferentes implementaciones de diccionario.

3.8.2. Entrada/salida de texto con una interfaz gráfica de usuario

La forma más directa de realizar una interfaz entre los programas y un usuario humano es a través de una interfaz gráfica de usuario (GUI).³² En esta sección se presenta una forma simple pero poderosa de definir interfaces gráficas de usuario, a saber, por medio de especificaciones concisas, principalmente declarativas. Este es un excelente ejemplo de lenguaje descriptivo declarativo, como se definió en la sección 3.1. El lenguaje descriptivo es reconocido por el módulo QTk del sistema Mozart. La interfaz de usuario se especifica como un registro anidado, complementado con objetos y procedimientos. (Los objetos se introducen en el capítulo 7. Por ahora, puede considerarlos como procedimientos con estado interno, como los ejemplos del capítulo 1.)

En esta sección se muestra cómo construir interfaces de usuario para leer y escribir datos textuales a una ventana. Esto es suficiente para muchos programas declarativos. Presentamos un breve vistazo al módulo QTk, suficiente para construir estas interfaces de usuario. Más adelante construiremos interfaces gráficas de usuario más sofisticadas. En el capítulo 10 (en CTM) se presenta una discusión más completa sobre programación declarativa de interfaces de usuario en general y de su desarrollo en QTk.

Especificación declarativa de aparatos

Una ventana en la pantalla está compuesta de un conjunto de aparatos. Un *aparato* es un área rectangular en la ventana que tiene un comportamiento interactivo particular. Por ejemplo, algunos aparatos pueden desplegar texto o información gráfica, y otros aparatos pueden aceptar interacción con el usuario a través de un teclado o de un ratón. Especificamos cada *aparato* de manera declarativa, con un registro cuyas etiqueta y campos definen el tipo del *aparato* y el estado inicial. Específicamente declarativamente la ventana como un registro anidado (i.e., un árbol) que define la estructura lógica de los aparatos en la ventana. Los seis aparatos que

32. Nota del traductor: del inglés, *Graphical user interface*.

usaremos por ahora son:

- El *aparato label* puede desplegar un texto. El *aparato* se especifica por medio del registro:

```
label(text:VS)
```

donde VS es una cadena virtual.

- El *aparato text* se usa para desplegar y recibir grandes cantidades de texto. Puede usar barras de navegación cuando tenga que desplegar más texto del que cabe en una pantalla. Un *aparato* con una barra de navegación vertical (i.e., de arriba hacia abajo) se especifica por el registro:

```
text(handle:H tdscrollbar:true)
```

Cuando la ventana se crea, la variable H se ligará a un objeto usado para controlar el aparato. Llamamos a un tal objeto, un manejador. Usted puede considerar el objeto como un procedimiento de un argumento: {H set(VS)} despliega un texto y {H get(VS)} lee el texto.

- El *aparato button* especifica un botón y una acción a ejecutar cuando se presiona el botón. El *aparato* se especifica así:

```
button(text:VS action:P)
```

donde VS es una cadena virtual, y P es un procedimiento sin argumentos. {P} se invoca en el momento en que el botón sea presionado.³³ Para cada ventana, todas sus acciones son ejecutadas secuencialmente.

- Los aparatos *td* (arriba-abajo) y *lr* (izquierda-derecha) especifican un arreglo de otras widgets ordenadas de arriba a abajo o de izquierda a derecha.

```
lr(W1 W2 ... Wn)  
td(W1 W2 ... Wn)
```

donde W1, W2, ..., Wn son otras especificaciones de aparatos.

Especificación declarativa del comportamiento redimensionable

Cuando se redimensiona una ventana, los aparatos dentro de ella deben comportarse adecuadamente, i.e., o cambian sus dimensiones o las mantienen, dependiendo sobre lo que la interfaz debe hacer. Especificamos declarativamente el comportamiento por redimensión de cada aparato, por medio de un campo opcional *glue* en el registro del aparato. El campo *glue* indica si los bordes del *aparato* deben o no estar “pegados” al *aparato* que lo encierra. El argumento del campo *glue* es un átomo consistente de la combinación de los cuatro caracteres n (norte), s (sur), w

33. Para ser precisos, cuando el botón izquierdo del ratón es presionado y soltado mientras el ratón pase por el botón, Esto permite al usuario corregir cualquier error de click sobre el botón.

```
declare En Sal
A1=proc {$} x in {En get(X)} {Sal set(X)} end
A2=proc {$} {W close} end
D=td(title:"Interfaz sencilla de E/S de texto"
      lr(label(text:"Entrada:")
          text(handle:En tdscrollbar:true glue:nswe)
          glue:nswe)
      lr(label(text:"Salida:")
          text(handle:Sal tdscrollbar:true glue:nswe)
          glue:nswe)
      lr(button(text:"Hágalo" action:A1 glue:nswe)
          button(text:"Salir" action:A2 glue:nswe)
          glue:we))
W={QTk.build D}
{W show}
```

Figura 3.33: Una interfaz sencilla para E/S de texto.

(oeste), e (este), indicando si en cada dirección el borde debe estar pegado o no. Algunos ejemplos son:

- No hay campo `glue`. El *aparato* guarda su tamaño natural y se centra en el espacio asignado a ella, tanto horizontal como verticalmente.
- `glue:nswe` pega los cuatro bordes estirándose para caber tanto horizontal como verticalmente.
- `glue:we` pega horizontalmente los bordes izquierdo y derecho, estirándolos para caber. Verticalmente el *aparato* no se estira pero se centra en el espacio asignado a ella.
- `glue:w` pega el lado izquierdo y no estira.
- `glue:wns` pega verticalmente, arriba y abajo, estirándose para caber verticalmente, y pegándose al eje izquierdo, y no se estira horizontalmente.

Instalando el módulo QTk

La primera etapa es instalar el módulo `QTk` en el sistema. Como `QTk` hace parte de la biblioteca estándar de Mozart, es suficiente conocer el nombre del camino correcto. Lo cargamos en la interfaz interactiva así:

```
declare [QTk]={Module.link ['x-oz://system/wp/QTk.ozf']}
```

Ahora que `QTk` está instalado, podemos usarlo para construir interfaces de acuerdo a las especificaciones de la sección anterior.

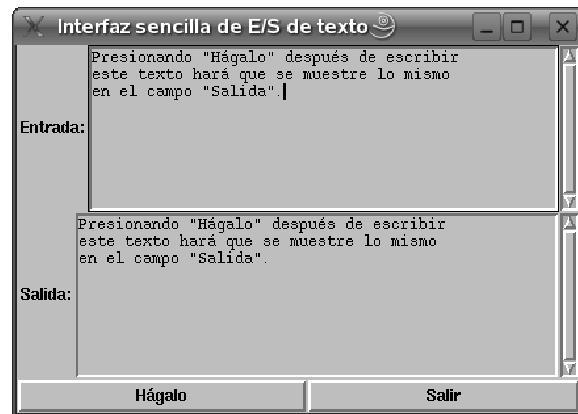


Figura 3.34: Foto de la interfaz.

Construyendo la interfaz

El módulo QTk tiene una función `QTk.build` que toma una especificación de una interfaz, la cual es sólo un registro anidado de aparatos, y construye una ventana nueva que contiene esas aparatos. Construyamos una interfaz sencilla con un botón que despliega `ay` en el browser cada vez que se hace click en él:

```
D=td(button(text:"Presióname"
           action:proc {$} {Browse ay} end))
W={QTk.build D}
{W show}
```

El registro `D` siempre tiene que empezar con `td` o `lr`, aún si la ventana tiene un solo aparato. La ventana empieza oculta. Puede ser desplegada u ocultada de nuevo, invocando `{W show}` o `{W hide}`. En la figura 3.33 se presenta un ejemplo más grande que implementa una interfaz completa de entrada/salida de texto. En la figura 3.34 se muestra la ventana resultante. A primera vista, este programa puede parecer complicado, pero mírello de nuevo: hay seis aparatos (dos `label`, dos `text`, dos `button`) organizadas con aparatos `td` y `lr`. La función `QTk.build` toma la descripción `D`, construye la ventana, y crea los manejadores de objetos `En` y `Sal`. Compare el registro `D` en la figura 3.33 con la foto en la figura 3.34.

Hay dos procedimientos acción, `A1` y `A2`, uno por cada botón. La acción `A1` está pegada al botón “`Hágalo`”. Al hacer click en el botón se invoca `A1`, el cual transfiere texto del primer *aparato* de texto al segundo *aparato* de texto. Esto funciona como se dice a continuación. La invocación `{En get(x)}` obtiene el texto del primer *aparato* de texto y lo liga a `x`. Luego, la invocación `{Sal set(x)}` asigna el texto en la segunda *aparato* de texto con `x`. La acción `A2` está pegada al botón “`Salir`”. Esta acción invoca `{W close}`, lo cual cierra la ventana permanentemente.

Colocar `nswe` en el campo `glue` de casi todo, hace que la ventana se comporte adecuadamente cuando es redimensionada. El *aparato* `lr` con los dos botones sólo

tiene `we` en el campo `glue`, de manera que los botones no se expanden verticalmente. Los aparatos `label` no tienen campo `glue`, de manera que siempre tienen tamaños fijos. El *aparato* `td` al más alto nivel no necesita campo `glue` pues suponemos que siempre está pegada a su ventana.

3.8.3. Entrada/salida de datos sin estado, usando archivos

La entrada/salida de una cadena es sencilla, pues una cadena consiste de caracteres que se pueden almacenar directamente en un archivo. ¿Qué pasa con los otros valores? Sería de gran ayuda para el programador si fuera posible guardar cualquier valor en un archivo y recuperarlo luego. El módulo `Pickle` del sistema provee precisamente esta capacidad. Puede guardar y recuperar cualquier valor completo:

```
{Pickle.save X FN}      % Guarda X en el archivo FN
{Pickle.load FNURL ?X} % Recupera X del archivo (o URL) FNURL
```

Todas las estructuras de datos utilizadas en programación declarativa se pueden guardar y recuperar salvo si contienen variables no-ligadas. Por ejemplo, considere el fragmento de programa siguiente:

```
declare
fun {Fact N}
  if N==0 then 1 else N*{Fact N-1} end
end
F100={Fact 100}
F100Gen1=fun {$} F100 end
F100Gen2=fun {$} {Fact 100} end
FNGen1=fun {$ N} F={Fact N} in fun {$} F end end
FNGen2=fun {$ N} fun {$} {Fact N} end end
```

`F100` es un entero (bastante grande); las otras cuatro entidades son funciones. La operación siguiente guarda las cuatro funciones en un archivo:

```
{Pickle.save [F100Gen1 F100Gen2 FNGen1 FNGen2] `archivofact`}
```

Para ser precisos, esto guarda un valor que consiste de una lista de cuatro elementos en el archivo `archivofact`. En este ejemplo, todos los elementos son funciones. Las funciones han sido escogidas para ilustrar varios grados de evaluación retardada. Los primeros dos devuelven el resultado de calcular $100!$. El primero, `F100Gen1`, conoce el entero y lo devuelve directamente, y el segundo, `F100Gen2`, calcula el valor cada vez que es invocado. El tercero y el cuarto, cuando se invocan con un argumento entero n , devuelven una función que cuando se invoca, devuelve $n!$. El tercero, `FNGen1`, calcula $n!$ cuando lo invocan, de manera que la función que devuelve ya conoce el entero que debe devolver cuando la invoquen. El cuarto, `FNGen2`, no hace ningún cálculo sino que deja ese trabajo para que la función devuelta calcule $n!$ cuando la invoquen.

Para usar el contenido de `archivofact`, se debe cargar el archivo primero:

```
declare [F1 F2 F3 F4]={Pickle.load `archivofact`} in
{Browse {F1}}
{Browse {F2}}
{Browse {{F3 100}}}
{Browse {{F4 100}}}
```

Esto muestra 100! cuatro veces. Por supuesto, también es posible hacer lo siguiente:

```
declare F1 F2 F3 F4 in
{Browse {F1}}
{Browse {F2}}
{Browse {{F3 100}}}
{Browse {{F4 100}}}
[F1 F2 F3 F4]={Pickle.load `archivofact`}
```

Después que el archivo se cargue, se muestra exactamente lo mismo que antes. Esto ilustra de nuevo cómo las variables de flujo de datos hacen posible usar una variable antes de ser ligada.

Enfatizamos en que el valor recuperado es exactamente el mismo que fue guardado. No hay diferencia ninguna entre ellos. Esto es cierto para todos los valores posibles: números, registros, procedimientos, nombres, átomos, listas, y así sucesivamente, incluyendo otros valores que veremos más adelante en el libro. Ejecutar lo siguiente en un proceso:

```
... % Primera declaración (define X)
{Pickle.save X `miarchivo`}
```

y luego esto en un segundo proceso:

```
X={Pickle.load `miarchivo`}
... % Segunda declaración (utiliza X)
```

es rigurosamente idéntico a ejecutar un solo proceso con lo siguiente:

```
... % Primera declaración (define X)
{Pickle.save X `miarchivo`}
_= {Pickle.load `miarchivo`}
... % Segunda declaración (utiliza X)
```

Si se eliminan las invocaciones a Pickle, así:

```
... % Primera declaración (define X)
... % Segunda declaración (utiliza X)
```

entonces sólo hay dos diferencias menores:

- El primer caso crea y lee el archivo `miarchivo`. El segundo caso no.
- El primer caso lanza una excepción si existe algún problema al crear o leer el archivo.

3.9. Diseño de programas en pequeño

Ahora que hemos visto muchas técnicas de programación, la siguiente etapa lógica es usarlas para resolver problemas. Esta etapa es llamada diseño de programas. Comienza a partir de un problema que queremos resolver (normalmente explicado

en palabras, algunas veces de forma no muy precisa); continúa definiendo la estructura del programa a alto nivel, i.e., qué técnicas de programación necesitamos usar y cómo conectarlas entre ellas; y termina con un programa completo que resuelve el problema.

En el diseño de programas, hay una distinción importante entre “programar en pequeño” y “programar en grande.” A los programas resultantes de estos dos ejercicios los llamaremos “programas pequeños” y “programas grandes,” respectivamente. La distinción no tiene nada que ver con el tamaño del programa en términos de líneas de código fuente, sino más bien con cuánta gente estuvo envuelta en su desarrollo. Los programas pequeños son escritos por una persona en un período de tiempo corto. Los programas grandes son escritos por más de una persona en un período de tiempo largo. La misma persona hoy y un año después deben ser consideradas como dos personas diferentes, pues las personas olvidan muchos detalles en un año. Esta sección presenta una introducción a la programación en pequeño; dejamos la programación en grande para la sección 6.7.

3.9.1. Metodología de diseño

Suponga que tenemos un problema que se puede resolver escribiendo un programa pequeño. Miremos cómo diseñar tal programa. Recomendamos la metodología de diseño siguiente, la cual es una mezcla de creatividad y pensamiento riguroso:

- *Especificación informal.* Empezamos por escribir tan precisamente como podamos lo que el programa debe hacer: cuáles son sus entradas y salidas y cómo se relacionan las segundas con las primeras. Esta descripción se llama una especificación informal. Aunque sea precisa la llamaremos “informal” porque está escrita en lenguaje natural (inglés, español, . . .). Las especificaciones “formales” se escriben en una notación matemática.
- *Ejemplos.* Para lograr que la especificación sea perfectamente clara, siempre es buena idea imaginar ejemplos de lo que el programa hace en casos particulares. Los ejemplos deben “presionar” el programa: usarlo en condiciones límites y en las formas más inesperadas que podamos imaginar.
- *Exploración.* Para definir qué técnicas de programación necesitaremos, una buena idea es utilizar la interfaz interactiva para experimentar con fragmentos de programa. La idea es escribir operaciones pequeñas que pensemos que puedan ser necesitadas por el programa. Usamos las operaciones que el sistema provee como una base. Esta etapa nos provee una visión más clara de lo que debe ser la estructura del programa.
- *Estructura y codificación.* En este punto podemos exponer la estructura del programa. Podemos hacer un bosquejo aproximado de las operaciones que se necesitan para calcular las salidas a partir de las entradas y de cómo lograr que estas operaciones se comporten bien juntas. Luego llenamos los blancos escribiendo el código del programa. Las operaciones deben ser sencillas: cada operación debe hacer una sola cosa. Para mejorar la estructura podemos agrupar las operaciones

relacionadas en módulos.

■ *Comprobación y razonamiento.* Ahora que tenemos un programa, debemos verificar que hace lo correcto. Lo ensayamos con una serie de casos de prueba, incluyendo los ejemplos que definimos anteriormente. Corregimos los errores hasta que el programa funcione bien. También podemos razonar sobre el programa y su complejidad, utilizando la semántica formal para las partes que no son claras. La comprobación y el razonamiento son complementarios: es importante hacer ambas para lograr un programa de alta calidad.

■ *Juicio de calidad.* El punto final es volver atrás y juzgar la calidad del diseño. Hay muchos factores envueltos en la calidad: ¿resuelve el diseño el problema planteado, es correcto el diseño, es eficiente, se puede mantener, se puede extender, es sencillo? La simplicidad es especialmente importante, pues hace que muchos de los otros factores sean más fáciles de lograr. Si un diseño es complejo, lo mejor que podemos decir es que no está en su forma definitiva. Cercano a la simplicidad aparece la completitud, es decir, si el diseño tiene (potencialmente) toda la funcionalidad que necesita, de manera que puede ser usado como un componente básico.

Estas etapas no son obligatorias, sino más bien deben ser tomadas como inspiración. Siéntase libre de adaptarlas a sus propias circunstancias. Por ejemplo, cuando esté ideando ejemplos, puede sentir que debe cambiar la especificación. Sin embargo, tenga cuidado de no olvidar la etapa más importante, la cual es la comprobación. Es muy importante porque cierra el ciclo: da retroalimentación de la etapa de codificación a la etapa de especificación.

3.9.2. Ejemplo de diseño de programa

Para ilustrar estas etapas, reconstruyamos el desarrollo de la aplicación de frecuencia de palabras de la sección 3.7.3. Este es un primer intento de especificación informal:

Dado un nombre de archivo, la aplicación abre una ventana y despliega una lista de parejas, donde cada pareja consiste de una palabra y un entero que representa el número de veces que la palabra aparece en el archivo.

¿Es suficientemente precisa esta especificación? ¿Qué pasa con un archivo que contenga una palabra que no es válida en español o con un archivo que contenga caracteres que no son ASCII (American Standard Code for Information Interchange)? Nuestra especificación no es suficientemente precisa: no define qué es una “palabra”. Para ser más precisos, tenemos que conocer el propósito de la aplicación. Digamos que sólo queremos darnos una idea general de las frecuencias de aparición de palabras, independientemente de un lenguaje en particular. Entonces podemos definir una palabra sencillamente como:

Una “palabra” es una secuencia consecutiva maximal de letras y dígitos.

Esto significa que las palabras se separan por lo menos con un carácter que no es ni letra ni dígito. Esto acepta palabras no válidas en español pero no acepta palabras que contengan caracteres que no sean ASCII. ¿Ya es suficientemente buena la especificación? ¿Qué pasa con palabras con guiones (tal como “true-blue” en inglés, por ejemplo) o con expresiones idiomáticas que actúan como unidades semánticas (tal como “prueba y error”)? En el interés de la simplicidad, rechazaremos éstas por ahora. Pero podríamos tener que cambiar la especificación más adelante para aceptarlas, dependiendo del uso que se dé a la aplicación de frecuencia de palabras.

Ahora tenemos nuestra especificación. Note el papel esencial que juegan los ejemplos. Ellos son indicadores importantes cuando avanzamos hacia una especificación precisa. Los ejemplos se diseñaron expresamente para comprobar los límites de la especificación.

La siguiente etapa consiste en diseñar la estructura del programa. La estructura apropiada parece ser un canal o conducto³⁴: primero se lee el archivo en una lista de caracteres y luego se convierte la lista de caracteres en una lista de palabras, donde una palabra se representa como una cadena de caracteres. Para contar las palabras necesitamos una estructura de datos indexada por palabras. El diccionario declarativo de la sección 3.7.2 sería ideal, pero está indexado por átomos. Afortunadamente, hay una operación para convertir cadenas de caracteres en átomos: `StringToAtom` (ver apéndice B). Con ésta podemos escribir nuestro programa. En la figura 3.30 se presenta el corazón: una función `FrecPalabras` que toma una lista de caracteres y devuelve un diccionario. Podemos comprobar este código con varios ejemplos, y especialmente con los ejemplos que usamos para escribir la especificación. A esto le agregaremos el código para leer el archivo y desplegar la salida en una ventana; para ello usamos las operaciones de archivos y de interfaces gráficas de usuario de la sección 3.8. Es importante empaquetar la aplicación limpiamente, como un componente de software. Esto se explica en las dos secciones siguientes.

3.9.3. Componentes de *software*

¿Cuál es una buena manera de organizar un programa? Uno podría escribir el programa como un todo monolítico grande, pero esto no es adecuado. Una mejor manera consiste en dividir el programa en unidades lógicas, cada una de las cuales implementa un conjunto de operaciones relacionadas de alguna manera. Cada unidad lógica consta de dos partes, una interfaz y una implementación. Sólo la interfaz es visible desde afuera de la unidad lógica. Una unidad lógica puede usar otras como parte de su implementación.

Entonces, un programa es, sencillamente, un grafo dirigido de unidades lógicas, donde una arista entre dos unidades lógicas significa que la primera necesita la segunda para su implementación. Estas unidades lógicas son llamadas popularmente

34. Nota del traductor: *pipeline*, en inglés.

```
<declaración> ::= functor <variable>
    [ import{ <variable> [ at <átomo> ]
        | <variable> `(` { (<átomo> | <ent>) [ `:` <variable> ] }+ `)`
        }+ ]
    [ export { [ (<átomo> | <ent>) `:` ] <variable> }+ ]
    define { <parteDeclaración> }+ [ in <declaración> ] end
    | ...
```

Tabla 3.8: Sintaxis de functor.

“módulos” o “componentes,” sin una definición formal de lo que estas palabras significan. En esta sección se introducen los conceptos básicos, se definen precisamente, y se muestra cómo se usan para ayudar a diseñar programas declarativos pequeños. En la sección 6.7 se explica cómo se pueden usar estas ideas para ayudar a diseñar programas en grande.

Módulos y functors

Un módulo agrupa un conjunto de operaciones relacionadas en una entidad que posee una interfaz y una implementación. Representamos los módulos de una manera sencilla:

- La interfaz del módulo es un registro que agrupa entidades relacionadas del lenguaje (normalmente procedimientos, pero cualquier cosa es permitida, incluyendo clases, objetos, etc.).
- La implementación del módulo es un conjunto de entidades de lenguaje a las que se puede acceder por medio de las operaciones de la interfaz pero están ocultas vistas desde el exterior. La implementación se oculta utilizando el concepto de alcance léxico.

Consideraremos las especificaciones de un módulo como entidades distintas de los módulos. Una especificación de un módulo es un tipo de plantilla que crea un módulo cada vez que se instancia. A la especificación de un módulo algunas veces se le llama un componente de software. Desafortunadamente, el término “componente de software” es ampliamente usado con muchos significados diferentes[170]. Para evitar confusiones, llamaremos a las especificaciones de módulos del libro functors. Un functor es una función cuyos argumentos son los módulos que necesita y cuyo resultado es un módulo nuevo. (Para ser precisos, ¡el functor toma interfaces de módulos como argumentos, crea un módulo nuevo, y devuelve la interfaz de ese módulo nuevo!) Debido al rol del functor en la estructuración de programas, lo proveemos como una abstracción lingüística. Un functor tiene tres partes: una parte **import**, que especifica qué otros módulos se necesitan, una parte **export**,

la cual especifica la interfaz del módulo, y una parte **define**, la cual define la implementación del módulo incluyendo la inicialización del código. La sintaxis para la declaración de functors permite su utilización como argumentos o como funciones, al igual que la sintaxis para procedimientos. En la tabla 3.8 se presenta la sintaxis para la definición de functors como declaraciones. Las definiciones de functors también se pueden usar como expresiones si la primera <variable> se reemplaza por una marca de anidamiento “\$”.

En la terminología de la ingeniería de software, un componente de software es una unidad de despliegue independiente, una unidad de desarrollo de un tercero, y no tiene estado persistente (de acuerdo a la definición presentada en [170]). Los functors satisfacen esta definición y son por lo tanto un tipo de componente de software. Con esta terminología, un módulo es una instancia de un componente; es el resultado de instalar un functor en un ambiente de módulos particular. El ambiente de módulos consiste de un conjunto de módulos, cada uno de los cuales puede tener un estado de ejecución.

Los functors en el sistema Mozart son unidades de compilación. Es decir, el sistema tiene soporte para manejar los functors en archivos, tanto en forma de código fuente (i.e., texto legible para el humano) como en forma de código objeto (i.e., forma compilada). El código fuente se puede compilar, i.e., traducir en código objeto. Esto facilita el uso de functors para intercambiar software entre desarrolladores. Por ejemplo el sistema Mozart tiene una biblioteca, denominada MOGUL (Mozart Global User Library), en la cual los terceros pueden colocar cualquier clase de información. Normalmente, estos colocan functors y aplicaciones.

Una aplicación es autónoma si se puede ejecutar sin la interfaz interactiva. La aplicación consiste de un functor principal, el cual se evalúa cuando el programa empieza. La aplicación importa los módulos que necesita, lo cual hace que otros functors sean evaluados. El functor principal se utiliza por el efecto de iniciar la aplicación, y no por el módulo que devuelve, el cual es ignorado silenciosamente. Evaluar, o “instalar,” un functor crea un módulo nuevo en tres etapas. Primero, se crea una variable de sólo lectura. Segundo, cuando el valor de la variable de solo lectura se necesite (i.e. el programa intenta invocar una operación del módulo), entonces el valor del functor se carga en la memoria y se invoca, en un hilo nuevo, con los módulos que él importa, como argumentos. Esto se llama enlace dinámico, contrario al enlace estático en el cual el functor se instala cuando la ejecución comienza. Tercero, la invocación del functor devuelve el módulo nuevo y lo liga a la variable. En cualquier momento, el conjunto de módulos instalados en ese instante se llama el ambiente de módulos.

Implementando módulos y functors

Miremos cómo construir componentes de software por etapas. Primero presentamos un ejemplo de módulo. Luego mostramos cómo convertir este módulo en un componente de software. Finalmente, lo volvemos una abstracción lingüística.

Ejemplo de módulo En general un módulo es un registro, y su interfaz se accede a través de los campos del registro. Construimos un módulo llamado `MyList` que provee la interfaz de procedimientos para concatenación, ordenamiento, y comprobación de membresía en listas. Esto se puede escribir así:

```
declare MyList in
local
    proc {Append ...} ... end
    proc {MergeSort ...} ... end
    proc {Sort ...} ... {MergeSort ...} ... end
    proc {Member ...} ... end
in
    MyList= `export` (append: Append
                      sort: Sort
                      member: Member
                      ...)
end
```

El procedimiento `MergeSort` es inalcanzable afuera de la declaración `local`. No hay acceso directo a los otros procedimientos sino a través de los campos del módulo `MyList`, el cual es un registro. Por ejemplo, a `Append` se accede por medio de `MyList.append`. La mayoría de los módulos de la biblioteca de Mozart, i.e., los módulos Base y System, siguen esta estructura.

Un componente de software Usando la abstracción procedimental, podemos convertir este módulo en un componente de software. El componente de software es una función que devuelve un módulo:

```
fun {MyListFunctor}
    proc {Append ...} ... end
    proc {MergeSort ...} ... end
    proc {Sort ...} ... {MergeSort ...} ... end
    proc {Member ...} ... end
in
    `export` (append: Append
              sort: Sort
              member: Member
              ...)
end
```

Cada vez que se invoca `MyListFunctor`, se crea y se devuelve un nuevo módulo `MyList`. En general, `MyListFunctor` podría tener argumentos, los cuales son los otros módulos necesitados por `MyList`.

A partir de esta definición, queda claro que los functors son sólo valores en el lenguaje. Ellos comparten las propiedades siguientes con los valores de tipo procedimiento:

- Una definición de functor se puede evaluar en tiempo de ejecución, dando como resultado un valor de tipo functor.
- Un functor puede tener referencias externas a otras entidades del lenguaje. Por ejemplo, es fácil construir un functor que contenga datos calculados en tiempo de

ejecución. Esto es útil, e.g., para incluir grandes tablas o datos de tipo imagen en formato fuente.

- Un functor se puede almacenar en un archivo utilizando el módulo `Pickle`. Este archivo puede ser leído por cualquier proceso Mozart. Esto facilita la creación de bibliotecas de functores por terceros, tal como `MOGUL`.
- Un functor es liviano: puede ser usado para encapsular una sola entidad tal como un objeto o una clase, para hacer explícitos los módulos necesitados por la entidad.

Debido a que los functores son valores, es posible manipularlos con el lenguaje en formas muy sofisticadas. Por ejemplo, se puede construir un componente de software que implemente programación basada en componentes, en el cual los componentes determinan en tiempo de ejecución qué otros componentes necesitan y en qué momento enlazarlos. Es aún posible una mayor flexibilidad si se utiliza tipamiento dinámico. Un componente puede enlazar un componente arbitrario en tiempo de ejecución, instalando cualesquier functores e invocándolos de acuerdo a sus necesidades.

Soporte lingüístico Esta abstracción de componente de software es una abstracción razonable para organizar programas grandes. Para hacerla más fácil de usar, para asegurar que no se use incorrectamente, y para hacer más clara la intención del programador (evitar confusión con otras técnicas de programación de alto orden), la convertimos en una abstracción lingüística. La función `MyListFunctor` corresponde a la siguiente sintaxis de functor:

```
functor
export
    append:Append
    sort:Sort
    member:Member
    ...
define
    proc {Append ...} ... end
    proc {MergeSort ...} ... end
    proc {Sort ...} ... {MergeSort ...} ... end
    proc {Member ...} ... end
end
```

Note que la declaración entre `define` y `end` declara variables implícitamente, de la misma forma que la declaración entre `local` e `in`.

Suponga que este functor ha sido compilado y almacenado en el archivo `MyList.ozf` (veremos después cómo compilar un functor). El módulo se puede crear de la manera siguiente desde la interfaz interactiva:

```
declare [MyList]={Module.link ['MyList.ozf']}
```

La función `Module.link` está definida en el módulo `Module` del sistema. Ella toma una lista de functores, los carga desde el sistema de archivos, los enlaza (i.e., los evalúa juntos de manera que cada módulo vea sus módulos importados), y devuelve

la correspondiente lista de módulos. El módulo `Module` permite realizar muchas otras operaciones sobre functors y módulos.

Importando módulos Los componentes de software pueden depender de otros componentes de software. Para ser precisos, la instanciación de un componente de software crea un módulo. La instanciación puede necesitar de otros módulos. En la sintaxis nueva, esto se declara con las definiciones `import`. Para importar un módulo de la biblioteca, es suficiente dar el nombre de su functor. Por el otro lado, para importar un módulo definido por el usuario se requiere declarar el nombre del archivo o el URL del archivo donde el functor está almacenado.³⁵ Esto es razonable, pues el sistema conoce dónde se almacenan los módulos de la biblioteca, pero no conoce dónde se almacenan sus propios functors. Considere el functor siguiente:

```
functor
  import
    Browser
    FO at `file:///home/mydir/FileOps.ozf'
  define
    {Browser.browse {FO.countLines `/etc/passwd`}}
  end
```

La definición `import` importa el módulo `Browser` del sistema y el módulo `FO` definido por el usuario especificado por el functor almacenado en el archivo `/home/mydir/FileOps.ozf`. En el momento en que este functor sea enlazado, se ejecuta la declaración entre `define...` `end`. Esto invoca la función `FO.countLines`, la cual cuenta el número de líneas de un archivo dado, y luego invoca al procedimiento `Browser.browse` para desplegar el resultado. Este functor en particular, se ha definido por su efecto, no por el módulo que él crea. Por ello no exporta ninguna interfaz.

3.9.4. Ejemplo de un programa autónomo

Ahora empaquetaremos la aplicación de frecuencias de palabras utilizando componentes y volviéndola un programa autónomo. En la figura 3.35 se presenta una foto del programa en ejecución. El programa consiste de dos componentes, `Dict` y `WordApp`, los cuales son functors cuyo código fuente están en los archivos `Dict.oz` y `WordApp.oz`. Estos componentes implementan el diccionario declarativo y la aplicación de fercuencias de palabras, respectivamente. Además de importar `Dict`, el componente `WordApp` también importa los módulos `File` y `QTk`, los cuales son utilizados para leer del archivo y para crear una ventana de salida.

El código fuente completo de los componentes `Dict` y `WordApp` se presenta en las figuras 3.36 y 3.37. La diferencia principal entre estos componentes y el código de las secciones 3.7.2 y 3.7.3 es que los componentes están encerrados entre `functor ...`

35. Otros esquemas de nombramiento son posibles, en los cuales los functors tienen algún nombre lógico en un sistema administrador de componentes.

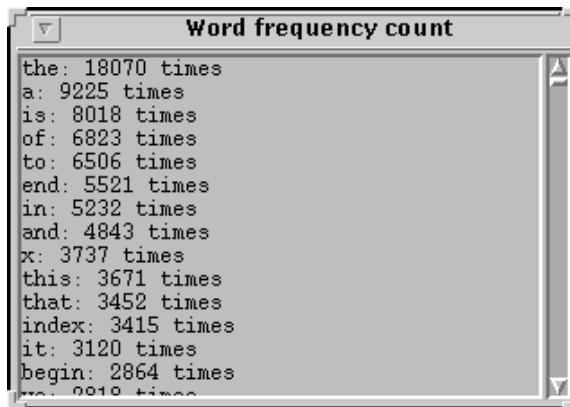


Figura 3.35: Foto de la aplicación de frecuencias de palabras.

end con las cláusulas **import** y **export** adecuadas. En la figura 3.38 se muestran las dependencias entre los módulos. Los módulos Open y Finalize son módulos de Mozart. El componente File se puede encontrar en el sitio Web del libro. El componente QTk está en la biblioteca estándar de Mozart. Se podría usar desde un functor añadiendo la cláusula siguiente:

```
import QTk at `x-oz://system/wp/QTk.ozf`
```

El componente Dict difiere ligeramente del diccionario declarativo de la sección 3.7.2: aquí se reemplaza Dominio por Entradas, la cual devuelve una lista de parejas Llave#Valor en lugar de una lista de solo llaves.

Esta aplicación se puede extender fácilmente de muchas maneras. Por ejemplo, la ventana que muestra la salida en WordApp.oz podría ser reemplazada por la siguiente:

```
H1 H2 Des=td(title:"Word frequency count"
              text(handle:H1 tdscrollbar:true glue:nswe)
              text(handle:H2 tdscrollbar:true glue:nswe))
W={QTk.build Des} {W show}

E={Dict.entradas {FrecPalabras L}}
SE1={Sort E fun {$ A B} A.1<B.1 end}
SE2={Sort E fun {$ A B} A.2>B.2 end}
for X#Y in SE1 do
    {H1 insert(`end` X#: `#Y#` times\n)}
end
for X#Y in SE2 do
    {H2 insert(`end` X#: `#Y#` times\n)}
end
```

Ésta despliega dos marcos, uno en orden alfabético y el otro en orden decreciente por la frecuencia de cada palabra.

```

functor
export nuevo:DiccionarioNuevo agregar:Agregar
consCond:ConsultarCond entradas:Entradas
define
    fun {DiccionarioNuevo} hoja end

    fun {Aregar Ds Llave Valor}
        case Ds
            of hoja then árbol(Llave Valor hoja hoja)
                [] árbol(K _ L R) andthen K==Llave then
                    árbol(K Valor L R)
                [] árbol(K V L R) andthen K>Llave then
                    árbol(K V {Aregar L Llave Valor} R)
                [] árbol(K V L R) andthen K<Llave then
                    árbol(K V L {Aregar R Llave Valor})
            end
        end

    fun {ConsultarCond Ds Llave Defecto}
        case Ds
            of hoja then Defecto
                [] árbol(K V _ _) andthen K==Llave then V
                [] árbol(K _ L _) andthen K>Llave then
                    {ConsultarCond L Llave Defecto}
                [] árbol(K _ _ R) andthen K<Llave then
                    {ConsultarCond R Llave Defecto}
            end
        end

    fun {Entradas Ds}
        proc {EntradasD Ds S1 ?Sn}
            case Ds
                of hoja then
                    S1=Sn
                [] árbol(K V L R) then S2 S3 in
                    {EntradasD L S1 S2}
                    S2=K#V|S3
                    {EntradasD R S3 Sn}
            end
        end
        in {EntradasD Ds $ nil} end
    end

```

Figura 3.36: Implementación autónoma de diccionario (archivo Dict.oz).

```

functor
import
    Dict File
    QTk at `x-oz://system/wp/QTk.ozf`
define
    fun {CarDePalabra C}
        (&a=<C andthen C=<&z) orelse
        (&A=<C andthen C=<&Z) orelse (&0=<C andthen C=<&9) end

    fun {PalAAAtomos PW} {StringToAtom {Reverse PW}} end

    fun {IncPal D W} {Dict.agregar D W {Dict.consCond D W 0}+1} end

    fun {CarsAPalabras PW Cs}
        case Cs
        of nil andthen PW==nil then
            nil
        [] nil then
            [{PalAAAtomos PW}]
        [] C|Cr andthen {CarDePalabra C} then
            {CarsAPalabras {Char.toLowerCase C}|PW Cr}
        [] _|Cr andthen PW==nil then
            {CarsAPalabras nil Cr}
        [] _|Cr then
            {PalAAAtomos PW}|{CarsAPalabras nil Cr}
        end
    end

    fun {ContarPalabras D Ws}
        case Ws of W|Wr then {ContarPalabras {IncPal D W} Wr}
        [] nil then D end
    end

    fun {FrecPalabras Cs}
        {ContarPalabras {Dict.nuevo} {CarsAPalabras nil Cs}} end

L={File.readList stdin}
E={Dict.entradas {FrecPalabras L}}
S={Sort E fun {$ A B} A.2>B.2 end

H Des=td(title:'Word frequency count'
         text(handle:H tdscrollbar:true glue:nswe))
W={QTk.build Des} {W show}
for X#Y in S do {H insert('end' X#: '#Y#' times\n')} end
end

```

Figura 3.37: Aplicación autónoma de frecuencias de palabras (archivo WordApp.oz).

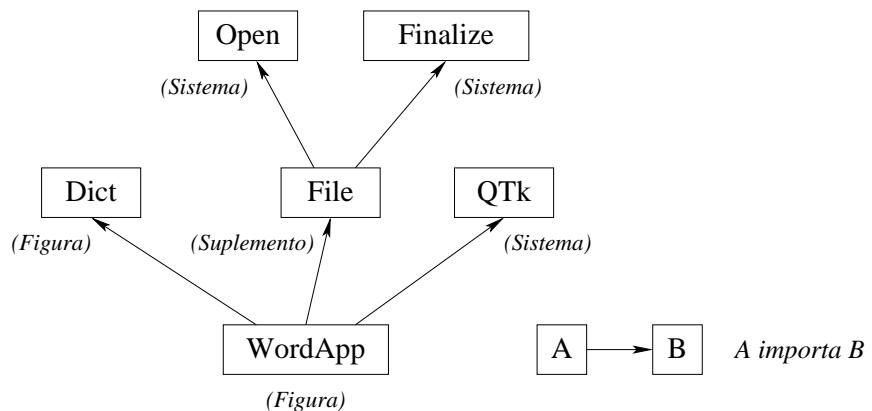


Figura 3.38: Dependencias de los componentes en la aplicación de frecuencias de palabras.

Compilación autónoma y ejecución

Ahora compilemos la aplicación de frecuencias de palabras como una aplicación autónoma. Un functor se puede usar de dos formas: como un functor compilado (el cual se puede importar desde otros functors) o como un programa autónomo (el cual se puede ejecutar directamente desde la línea de comandos). Cualquier functor puede ser compilado para ser un programa autónomo. En ese caso, no se necesita la parte **export**, y la parte **define** define el efecto del programa. Dado el archivo `Dict.oz` conteniendo un functor, el functor compilado `Dict.ozf` se crea con el comando `ozc` a partir de la línea de comandos:

```
ozc -c Dict.oz
```

Dado el archivo `WordApp.oz` conteniendo un functor para ser usado como programa autónomo, el programa autónomo ejecutable `WordApp` se crea con el siguiente comando:

```
ozc -x WordApp.oz
```

Éste se puede ejecutar así:

```
WordApp < book.raw
```

donde `book.raw` es un archivo que contiene un texto. El texto se pasa a la entrada estándar del programa, la cual se ve dentro del programa como un archivo de nombre `stdin`. El functor compilado `Dict.ozf` se enlaza dinámicamente, con el primer acceso al diccionario. También es posible enlazar estáticamente `Dict.ozf` en el código compilado de la aplicación `WordApp`, de manera que no se necesite ningún enlace dinámico. Estas posibilidades están documentadas en el sistema Mozart.

Bibliotecas como módulos

La aplicación de frecuencias de palabras utiliza el módulo `QTK`, el cual hace parte del sistema Mozart. Cualquier lenguaje de programación, para ser útil en la práctica, debe venir acompañado de un amplio conjunto de abstracciones útiles. Éstas se organizan en bibliotecas. Una biblioteca es una colección coherente de una o más abstracciones relacionadas que son útiles en un dominio de problema particular. Dependiendo del lenguaje y de la biblioteca, ésta puede considerarse como parte del lenguaje o como externa al lenguaje. La línea divisoria puede ser algo vaga: en casi todos los casos, muchas de las operaciones básicas de un lenguaje están implementadas, de hecho, en bibliotecas. Por ejemplo, las funciones de mayor dificultad sobre los números reales (seno, coseno, logartimo, etc.) están implementadas, normalmente, en bibliotecas. Como el número de bibliotecas puede ser muy grande, es una buena idea organizarlas como módulos.

Las bibliotecas han venido creciendo en importancia. Han sido estimuladas, por un lado, por la creciente velocidad y capacidad de memoria de los computadores, y por el otro lado, por las crecientes solicitudes de los usuarios. Un lenguaje nuevo que no venga con un conjunto significante de bibliotecas, e.g., para operaciones en red, operaciones gráficas, operaciones sobre bases de datos, etc., o es un juguete, no adecuado para el desarrollo de aplicaciones reales, o es útil sólo en un dominio reducido de problemas. Implementar bibliotecas requiere de un gran esfuerzo. Para aliviar este problema, los lenguajes nuevos vienen, casi siempre, con una interfaz a lenguajes externos. Ésta permite la comunicación con programas escritos en otros lenguajes.

Bibliotecas como módulos en Mozart La biblioteca disponible como módulos en el sistema Mozart consiste de módulos Base y de módulos System. Los módulos Base están disponibles, inmediatamente se inicia el sistema Mozart, tanto desde la interfaz interactiva como desde las aplicaciones autónomas. Estos módulos proveen las operaciones básicas sobre los tipos de datos del lenguaje. Las operaciones sobre números, listas, y registros presentadas en este capítulo hacen parte de los módulos Base. Los módulos System también están disponibles, inmediatamente se inicia el sistema Mozart, desde la interfaz interactiva, pero en las aplicaciones autónomas (i.e., functors) deben ser importados. (Los módulos en la Biblioteca Estándar de Mozart, tales como `QTK`, deben ser instalados explícitamente, siempre, aún en la interfaz interactiva.) Los módulos System proveen funcionalidad adicional tal como E/S por archivos, interfaces gráficas de usuario, programación distribuida, programación lógica y por restricciones, acceso al sistema operativo, entre otras.

Por medio de la interfaz interactiva se puede ver la lista de módulos instalados en el sistema. Simplemente, en el menú interactivo `Oz` escoja la opción `Open Compiler Panel` y luego haga clic sobre la opción `Environment`. Esto muestra todas las variables que están definidas en el ambiente global, incluyendo los módulos.

3.10. Ejercicios

1. *Valor absoluto de números reales.* Nos gustaría definir una función `Abs` que calcule el valor absoluto de un número real. La definición siguiente no funciona:

```
fun {Abs X} if X<0 then ~X else X end end
```

¿Por qué no? ¿Cómo se corregiría? *Sugerencia:* el problema es trivial.

2. *Raíces cúbicas.* En este capítulo se utilizó el método de Newton para calcular raíces cuadradas. El método se puede extender para calcular raíces de cualquier grado. Por ejemplo, el método siguiente calcula raíces cúbicas. Dada una conjetura g para la raíz cúbica de x , una conjetura mejorada se calcula como $(x/g^2 + 2g)/3$. Escriba un programa declarativo para calcular raíces cúbicas usando el método de Newton.

3. *El método del medio intervalo.*³⁶ El método del medio intervalo es una técnica sencilla pero poderosa para encontrar las raíces de una ecuación de la forma $f(x) = 0$, donde f es una función real continua. La idea es que, si tenemos dos números reales a y b tales que $f(a) < 0 < f(b)$, entonces f debe tener al menos una raíz entre a y b . Para encontrar una raíz, sea $x = (a+b)/2$ y calcule $f(x)$. Si $f(x) > 0$, entonces f debe tener una raíz entre a y x . Si $f(x) < 0$, entonces f debe tener una raíz entre x y b . La repetición de este proceso definirá cada vez intervalos más pequeños lo cual convergerá finalmente en una raíz. Escriba un programa declarativo que resuelva este problema usando las técnicas de computación iterativa.

4. *Factorial iterativo.* Este capítulo presenta una definición de factorial cuya profundidad máxima de la pila es proporcional al argumento de entrada. Dé otra definición de factorial que defina una computación iterativa. Use la técnica de transformación de estados, a partir de un estado inicial, como se mostró en el ejemplo de la función `IterLength`.

5. *Un SumList iterativo.* Reescriba la función `SumList` de la sección 3.4.2 de manera que sea iterativa, utilizando las técnicas desarrolladas para `Length`.

6. *Invariantes de estado.* Escriba un invariante de estado para la función `IterReverse`.

7. *Otra función de concatenación.* En la sección 3.4.2 se define la función `Append` haciendo recursión sobre el primer argumento. Qué pasa si tratamos de hacer recursión sobre el segundo argumento? Una posible solución es:

```
fun {Append Ls Ms}
  case Ms
    of nil then Ls
    [] X|Mr then {Append {Append Ls [X]} Mr}
    end
  end
```

¿Este programa es correcto? ¿Termina? ¿Por qué sí o por qué no?

36. Este ejercicio es tomado de [1].

8. *Concatenación iterativa.* Este ejercicio explora el poder expresivo de las variables de flujo de datos. En el modelo declarativo, la siguiente definición de concatenación es iterativa:

```
fun {Append Xs Ys}
  case Xs
    of nil then Ys
    [] X|Xr then X|{Append Xr Ys}
    end
  end
```

Podemos ver esto mirando la expansión:

```
proc {Append Xs Ys ?Zs}
  case Xs
    of nil then Zs=Ys
    [] X|Xr then Zr in
      Zs=X|Zr
      {Append Xr Ys Zr}
    end
  end
```

Aquí se puede hacer una optimización de última invocación debido a que la variable no-ligada `Zr` se puede colocar en la lista `Zs` y ligarla más tarde. Ahora, restrinjamos el modelo de computación, a calcular sólo con valores. ¿Cómo podemos escribir una concatenación iterativa? Un enfoque consiste en definir dos funciones: (1) una inversión iterativa de una lista y (2) una función iterativa que concatena el inverso de una lista con otra lista. Escriba una concatenación iterativa utilizando este enfoque.

9. *Computaciones iterativas y variables de flujo de datos.* El ejercicio anterior muestra que la utilización de variables de flujo de datos simplifica, algunas veces, la escritura de operaciones iterativas sobre listas. Esto nos lleva a la pregunta siguiente. Para cualquier operación iterativa definida con variables de flujo de datos, ¿es posible encontrar otra definición iterativa de la misma operación que no utilice variables de flujo de datos?

10. *Comprobando si algo es una lista.* En la sección 3.4.3 se define una función `LengthL` que calcula el número de elementos en una lista anidada. Para ver si `x` es o no una lista, `LengthL` utiliza la función `Hoja` definida así:

```
fun {Hoja x} case x of _|_ then false else true end end
```

Qué pasa si reemplazamos esto por la siguiente definición?:

```
fun {Hoja x} x\=(_|_) end
```

¿Qué es lo que no funciona si usamos esta versión de `Hoja`?

11. *Limitaciones de las listas de diferencias.* ¿Qué es lo que no funciona bien cuando se trata de concatenar la misma lista de diferencias más de una vez?

12. *Complejidad de aplanar listas.* Calcule el número de operaciones necesitadas por las dos versiones de la función `Aplanar` presentadas en la sección 3.4.4. Si se tienen n elementos y un máximo nivel de anidamiento de k , ¿cuál es la complejidad

de cada versión en el peor caso?

13. *Operaciones sobre matrices.* Suponga que representamos una matriz como una lista de listas de enteros, donde cada lista interna representa una fila de la matriz. Defina las funciones para realizar las operaciones estándar sobre matrices como transposición y multiplicación de matrices.

14. *Colas FIFO.* Considere la cola FIFO definida en la sección 3.4.4. Responda las dos preguntas siguientes:

- a) ¿Qué pasa si usted elimina un elemento de una cola vacía?
- b) Por qué sería incorrecto definir `Esvacía` de la siguiente manera:

```
fun {Esvacía q(N S E)} S==E end
```

15. *Quicksort.* A continuación presentamos un algoritmo para ordenar listas. Su inventor, C.A.R. Hoare, lo llamó quicksort, debido a que fue reconocido como el algoritmo de ordenamiento, de propósito general, más rápido de la época en que se inventó. Este algoritmo utiliza una estrategia de dividir y conquistar y logra una complejidad en tiempo, promedio, de $\Theta(n \log n)$. A continuación se presenta una descripción informal del algoritmo para el modelo declarativo. Dada una lista de entrada `L`, realice las operaciones siguientes:

- a) Tome el primer elemento de `L`, `x`, para usarlo como pivote.
- b) Divida `L` en dos listas, `L1` y `L2`, tal que todos los elementos de `L1` sean menores que `x` y todos los elementos de `L2` sean mayores o iguales que `x`.
- c) Utilice quicksort para ordenar `L1` y `L2`; llamemos `S1` y `S2` los resultados respectivos.
- d) Devuelva la concatenación de `S1` y `S2` como respuesta.

Escriba este programa con listas de diferencias para evitar el costo lineal de la concatenación.

16. (ejercicio avanzado) *Convolución con recursión de cola.*³⁷ En este ejercicio, escriba una función que reciba dos listas $[x_1 \ x_2 \ \dots \ x_n]$ y $[y_1 \ y_2 \ \dots \ y_n]$ y devuelva su convolución simbólica, $[x_1 \# y_n \ x_2 \# y_{n-1} \ \dots \ x_n \# y_1]$. La función debe ser recursiva por la cola y no realizar más de n invocaciones recursivas. *Sugerencia:* la función puede calcular la inversa de la segunda lista y pasársela como un argumento a sí misma. Como la unificación es independiente del orden, esto funciona perfectamente bien.

17. (ejercicio avanzado) *Curificación.* El propósito de este ejercicio es definir una abstracción lingüística para proveer un mecanismo de currificación en Oz. Primero defina un esquema para traducir definiciones de función e invocaciones. Luego, utilice la herramienta para generar analizadores sintáticos, `gump`, para agregar la abstracción lingüística a Mozart.

37. Este ejercicio está inspirado en un ejercicio de Olivier Danvy y Mayer Goldberg.

4

Concurrencia declarativa

Hace veinte años se pensaba que mantener los esquías paralelos al esquiar era una habilidad que sólo se lograba después de muchos años de entrenamiento y práctica. Hoy, esta habilidad se logra en el transcurso de una sola temporada de esquí. ... Todos los objetivos de los padres son alcanzados por los hijos: ... Pero las acciones que ellos realizan para producir esos resultados son bastante diferentes.

– Adaptación libre de *Mindstorms: Children, Computers, and Powerful Ideas*, Seymour Papert (1980)

El modelo declarativo del capítulo 2 nos permite escribir muchos programas y usar técnicas poderosas de razonamiento sobre ellos. Pero, como se explica en la sección 4.8, existen programas útiles que no se pueden escribir fácilmente o eficientemente en ese modelo. Por ejemplo, algunos programas se escriben mejor como un conjunto de actividades que se ejecutan independientemente. Tales programas son llamados concurrentes. La concurrencia es esencial en programas que interactúan con su ambiente, e.g., para agentes, programación de interfaces gráficas de usuario (GUI), interacción de sistemas operativos (OS), y así sucesivamente. La concurrencia también permite organizar los programas en partes que se ejecutan independientemente e interactúan sólo cuando lo necesitan, i.e., programas cliente/servidor y productor/consumidor. Esta es una importante propiedad en ingeniería de software.

La concurrencia puede ser sencilla

Este capítulo extiende el modelo declarativo del capítulo 2 añadiéndole concurrencia mientras continúa siendo declarativo. Es decir, todas las técnicas de programación de razonamiento para programas declarativos siguen siendo válidas. Esta es una propiedad extraordinaria que merece ser más ampliamente conocida, y la exploraremos a lo largo de este capítulo. La intuición subyacente es bastante simple. Está basada en el hecho que una variable de flujo de datos puede ser ligada a un único valor. Esto trae las consecuencias siguientes:

- Qué permanece igual: El resultado de un programa es el mismo sea o no concurrente. Colocar cualquier parte del programa en un hilo no cambia el resultado.
- Qué es nuevo: El resultado de un programa se puede calcular incrementalmente. Si la entrada a un programa concurrente llega de manera incremental, entonces el programa también puede calcular la salida de la misma manera.

Concurrencia declarativa

Presentamos un ejemplo para aclarar esta intuición. Considere el programa secuencial siguiente que calcula una lista de cuadrados sucesivos generando una lista de enteros sucesivos para luego calcularle a cada uno su cuadrado:

```
fun {Gen L H}
  {Delay 100}
  if L>H then nil else L|{Gen L+1 H} end
end

Xs={Gen 1 10}
Ys={Map Xs fun {$ x} x*x end}
{Browse Ys}
```

(La invocación {Delay 100} hace que pasen 100 ms antes de dejar continuar los cálculos.) Podemos volver este programa concurrente realizando la generación y la aplicación, cada una en su propio hilo:

```
thread Xs={Gen 1 10} end
thread Ys={Map Xs fun {$ x} x*x end} end
{Browse Ys}
```

Aquí se utiliza la declaración **thread** *d* **end**, la cual ejecuta *d* concurrentemente. ¿Cuál es la diferencia entre las versiones concurrente y secuencial? El resultado del cálculo es el mismo en ambos casos, a saber, [1 4 9 16 ... 81 100]. En la versión secuencial, Gen calcula toda la lista antes que Map comience. El resultado final se despliega todo de una sola vez cuando el cálculo ha sido terminado, después de un segundo. En la versión concurrente, Gen y Map se ejecutan ambos simultáneamente. Cada vez que Gen agrega un elemento a su lista, Map calcula inmediatamente su cuadrado. El resultado se despliega incrementalmente, a medida que los elementos de la lista son generados, un elemento cada décima de segundo.

Ya veremos que la razón profunda por la que esta forma de concurrencia es tan sencilla es que los programas tienen un no-determinismo no observable. Un programa en el modelo declarativo concurrente siempre tiene esta propiedad, si el programa no trata de ligar la misma variable con valores incompatibles. Esto se explica en la sección 4.1. Otra forma de decir esto es que no existen condiciones de carrera en un programa concurrente declarativo. Una condición de carrera es justamente un comportamiento no-determinístico observable.

Estructura del capítulo

El capítulo se puede dividir en seis partes:

- *Programación con hilos*. En esta parte se explica la primera forma de concurrencia declarativa, a saber, concurrencia dirigida por los datos, también conocida como concurrencia dirigida por la oferta. Para esto se cuenta con cuatro secciones. En la sección 4.1 se define el modelo concurrente dirigido por los datos, el cual extiende el modelo declarativo con hilos. En esta sección también se explica qué significa concurrencia declarativa. En la sección 4.2 se presentan las bases de la programación con hilos. En la sección 4.3 se explica la técnica más popular, la comunicación por

flujos. En la sección 4.4 se presentan otras técnicas, a saber, concurrencia para determinación del orden, corutinas, y composición concurrente.

- *Ejecución perezosa.* En esta parte se explica la segunda forma de concurrencia declarativa, a saber concurrencia dirigida por la demanda, también conocida como ejecución perezosa. En la sección 4.5 se introduce el modelo concurrente perezoso y se presentan algunas de las más importantes técnicas de programación, incluyendo flujos perezosos y listas comprehensivas.
- *Programación en tiempo real no estricto.* En la sección 4.6 se explica cómo programar con tiempo en el modelo concurrente declarativo.
- *El lenguaje Haskell.* En la sección 4.7 se presenta una introducción a Haskell, un lenguaje de programación funcional puro basado en evaluación perezosa. La evaluación perezosa es la forma secuencial de la ejecución perezosa.
- *Limitaciones y extensiones de la programación declarativa.* ¿Cuán lejos puede llegar la programación declarativa? En la sección 4.8 se exploran las limitaciones de la programación declarativa y cómo superarlas. En esta sección se presentan las primeras motivaciones para estado explícito, el cual es introducido y estudiado en los siguientes cuatro capítulos.
- *Temas avanzados e historia.* En la sección 4.9 se muestra cómo extender el modelo concurrente declarativo con excepciones. También se profundiza en varios temas que incluyen las diferentes clases de no-determinismo, ejecución perezosa, variables de flujo de datos, y sincronización (tanto explícita como implícita). Se concluye con la sección 4.10 presentando algunas notas históricas sobre las raíces de la concurrencia declarativa.

La concurrencia es también una parte clave de otros tres capítulos. En el capítulo 5 se extiende el modelo ansioso de este capítulo con un tipo sencillo de comunicación por canal. En el capítulo 8 se explica cómo usar la concurrencia junto con estado, e.g., para programación concurrente orientada a objetos. En el capítulo 11 (en CTM) se muestra cómo hacer programación distribuida, i.e., programar un conjunto de computadores que están conectados por una red. Incluyendo este capítulo, estos cuatro capítulos juntos presentan una introducción comprehensiva a la programación concurrente práctica.

4.1. El modelo concurrente dirigido por los datos

En el capítulo 2 se presentó el modelo de computación declarativa. Este modelo es secuencial, i.e., en todo momento sólo hay una declaración que se ejecuta sobre el almacén de asignación única. Ahora extenderemos el modelo en dos pasos, agregando un solo concepto en cada paso:

- El primer paso es el más importante. Añadimos hilos y la instrucción **thread** *{d}* **end**. Un hilo es sencillamente una declaración en ejecución, i.e., una pila semántica.

Concurrencia declarativa

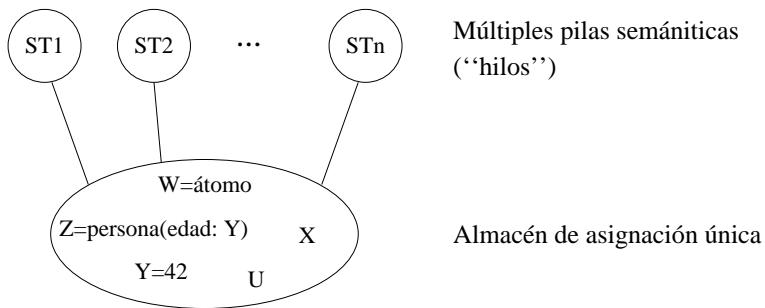


Figura 4.1: El modelo concurrente declarativo.

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Invocación de procedimiento
thread $\langle d \rangle$ end	Creación de hilo

Tabla 4.1: El lenguaje núcleo del modelo concurrente dirigido por los datos.

Esto es todo lo que necesitamos para comenzar a programar con concurrencia declarativa. Como lo veremos, agregar hilos al modelo declarativo preserva todas las propiedades buenas del modelo. Al modelo resultante lo llamamos el modelo concurrente dirigido por los datos.

- El segundo paso extiende el modelo con otro mecanismo de ejecución. Agregamos disparadores por necesidad y la instrucción {ByNeed P X}. Así se adiciona la posibilidad de hacer computación dirigida por la demanda, o ejecución perezosa. Esta segunda extensión también preserva las propiedades buenas del modelo declarativo. Al modelo resultante lo llamamos el modelo concurrente dirigido por la demanda o modelo concurrente perezoso. Se pospone la explicación de la ejecución perezosa hasta la sección 4.5.

En la mayor parte de este capítulo, se dejan las excepciones por fuera del modelo. Esto debido a que con las excepciones el modelo deja de ser declarativo. En la sección 4.9.1 se mira más de cerca la interacción entre concurrencia y excepciones.

4.1.1. Conceptos básicos

Nuestro enfoque de concurrencia no es más que una extensión sencilla del modelo declarativo que permite que haya más de una declaración en ejecución referenciando el almacén. Informalmente hablando, todas estas declaraciones se están ejecutando “al mismo tiempo.” Esto nos lleva al modelo ilustrado en la figura 4.1, cuyo lenguaje núcleo se muestra en la tabla 4.1. El lenguaje núcleo extiende el de la figura 2.1 con una nueva y única instrucción, la declaración **thread**.

Intercalación

Hagamos una pausa para considerar precisamente qué significa “al mismo tiempo.” Hay dos maneras de mirar este asunto, las cuales llamamos el punto de vista del lenguaje y el punto de vista de la implementación:

- El punto de vista del lenguaje es la semántica del lenguaje, tal como la ve el programador. Desde este punto de vista, la suposición más sencilla es dejar que los hilos se ejecuten de manera intercalada; hay una secuencia global de etapas de computación y los hilos van tomando su turno para realizar etapas de computación. Las etapas de computación no se superponen; en otras palabras, cada etapa de computación es atómica. Esto simplifica el razonar sobre los programas.
- El punto de vista de la implementación es cómo los múltiples hilos se implementan, en efecto, en una máquina real. Si el sistema se implementa sobre un único procesador, entonces la implementación podría ser, también, intercalada. Sin embargo, el sistema podría ser implementado sobre múltiples procesadores, de manera que los hilos pueden realizar varias etapas de computación simultáneamente. Así se tomaría ventaja del paralelismo para mejorar el desempeño.

Nosotros usamos la semántica de intercalación a lo largo del libro. Cualquiera que sea la ejecución paralela, siempre existe al menos una intercalación observacionalmente equivalente a ella. Es decir, si observamos el almacén durante la ejecución, siempre podremos encontrar una ejecución intercalada que haga que el almacén evolucione de la misma manera.

Orden causal

Otra forma de ver la diferencia entre ejecución secuencial y concurrente es en términos de un orden definido entre todos los estados de ejecución de un programa cualquiera:

Concurrencia declarativa

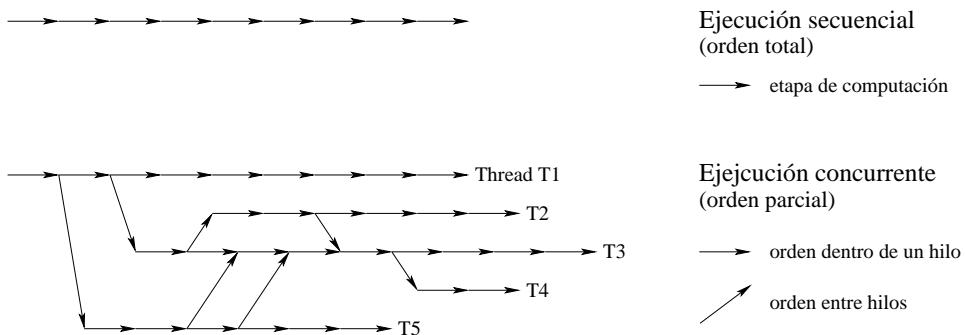


Figura 4.2: Ordenes causales de ejecuciones secuenciales y concurrentes.

Orden causal de las etapas de la computación

Dado un programa cualquiera, todas las etapas de computación forman un orden parcial, llamado el orden causal. Una etapa de computación está antes que otra etapa si en todas las posibles ejecuciones del programa la primera ocurre antes que la segunda. De la misma manera, para una etapa que está después de otra etapa. Algunas veces una etapa no está ni antes ni después de otra. En este caso, decimos que las dos etapas son concurrentes.

En un programa secuencial, todas las etapas de computación están totalmente ordenadas. No hay etapas concurrentes. En un programa concurrente, todas las etapas de computación de un hilo cualquiera están totalmente ordenadas. Las etapas de computación del programa completo forman un orden parcial. Dentro de este orden parcial, dos etapas están ordenadas causalmente si la primera liga una variable de flujo de datos x y la segunda necesita el valor de x.

En la figura 4.2 se muestra la diferencia entre ejecución secuencial y concurrente. En la figura 4.3 se presenta un ejemplo que muestra algunas de las posibles ejecuciones correspondientes a un orden causal particular. Allí el orden causal tiene dos hilos, T1 y T2, donde T1 tiene dos operaciones (I_1 e I_2) y T2 tiene tres operaciones (I_a , I_b , e I_c). Se muestran cuatro posibles ejecuciones. Cada una de ellas respeta el orden causal, i.e., todas las instrucciones relacionadas en el orden causal están relacionadas de la misma manera en la ejecución. ¿Cuántas ejecuciones en total son posibles? *Sugerencia:* no hay muchas en este ejemplo.

No-determinismo

Una ejecución se llama no-determinística si existe un estado de ejecución en el cual haya qué escoger qué hacer en el siguiente paso, i.e., elegir cuál hilo reducir. El no-determinismo aparece naturalmente cuando hay estados concurrentes. Si hay

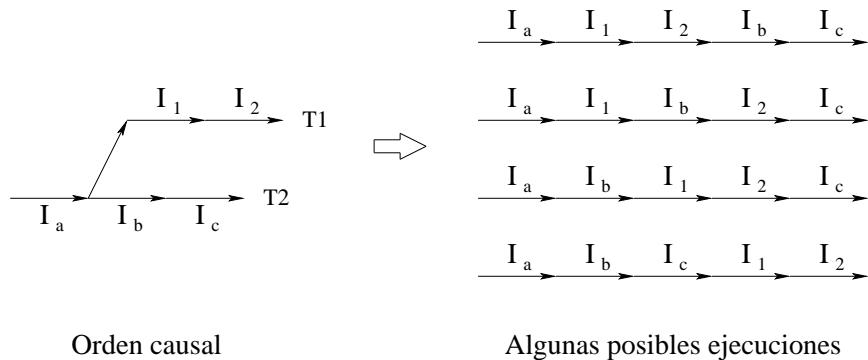


Figura 4.3: Relación entre orden causal y ejecuciones intercaladas.

varios hilos, entonces en cada estado de ejecución el sistema tiene que escoger cuál hilo ejecutar en el siguiente paso. Por ejemplo, en la figura 4.3, después de la primera etapa, la cual siempre realiza I_a , hay que escoger entre I_1 o I_b como la siguiente etapa.

En un modelo concurrente declarativo, el no-determinismo no es visible al programador.¹ Hay dos razones para esto. La primera, las variables de flujo de datos sólo pueden ser ligadas a un valor. El no-determinismo afecta sólo el momento exacto en que cada ligadura tiene lugar; no afecta el hecho de que la ligadura tenga lugar. La segunda, cualquier operación que necesite el valor de una variable no tiene salida diferente a esperar hasta que la variable sea ligada. Si permitimos que las operaciones puedan decidir si esperar o no, el no-determinismo se volverá visible.

Como consecuencia, un modelo declarativo concurrente preserva las propiedades buenas del modelo declarativo del capítulo 2. El modelo concurrente elimina algunas de las limitaciones del modelo declarativo, pero no todas, tal como lo veremos en este capítulo.

Planificación

La decisión de cuál hilo ejecutar en el próximo paso es realizada por una parte del sistema llamada el planificador. En cada etapa de computación, el planificador escoge uno de todos los hilos que están listos, para ejecutarlo en el siguiente paso. Se dice que un hilo está listo, también se dice ejecutable, si su declaración cuenta con toda la información que necesita para ejecutar al menos una etapa de computación. Una vez un hilo está listo, él permanece listo indefinidamente. Se dice que la reducción de un hilo en el modelo concurrente declarativo es monótona. Un hilo

1. Si no hay fallas de unificación, i.e., intentos de ligar la misma variable a valores incompatibles. Normalmente consideramos que una falla de unificación es consecuencia de un error del programador.

Concurrencia declarativa

listo se puede ejecutar en cualquier momento.

Un hilo que no está listo se dice que está suspendido. Su primera declaración no puede continuar debido a que no tiene toda la información que necesita. Se dice que la primera declaración está bloqueada. Este último es un concepto importante sobre el cual volveremos de nuevo.

Decimos que el sistema es imparcial si no deja a ningún hilo listo “con hambre”; i.e., todos los hilos listos se ejecutan finalmente. Esta es una propiedad muy importante para hacer que el comportamiento del programa sea predecible y para simplificar el razonamiento sobre los programas. Esta propiedad está relacionada con la modularidad: la imparcialidad implica que la ejecución de un hilo no depende de la ejecución de ningún otro hilo, a menos que la dependencia sea programada explícitamente. En el resto del libro, supondremos que la ejecución de los hilos se planifica imparcialmente.

4.1.2. Semántica de los hilos

Extendemos la máquina abstracta de la sección 2.4 dejándola ejecutarse con varias pilas semánticas en lugar de una sola. Cada pila semántica corresponde al concepto intuitivo de “hilo.” Todas las pilas semánticas tienen acceso al mismo almacén. Los hilos se comunican a través de este almacén compartido.

Conceptos

Conservamos los conceptos de almacén de asignación única σ , ambiente E , declaración semántica $(\langle d \rangle, E)$, y pila semántica ST . Extendemos los conceptos de estado de ejecución y de computación para tener en cuenta múltiples pilas semánticas:

- Un estado de ejecución es una pareja (MST, σ) donde MST es un multiconjunto de pilas semánticas y σ es un almacén de asignación única. Un multiconjunto es un conjunto en el cual el mismo elemento puede aparecer más de una vez. MST tiene que ser un multiconjunto porque podemos tener dos pilas semánticas diferentes con contenidos idénticos, e.g., dos hilos que ejecutan las mismas declaraciones.
- Una computación es una secuencia de estados de ejecución que comienza en un estado inicial: $(MST_0, \sigma_0) \rightarrow (MST_1, \sigma_1) \rightarrow (MST_2, \sigma_2) \rightarrow \dots$.

Ejecución de un programa

Al igual que antes, un programa es sencillamente una declaración $\langle d \rangle$. Un programa se ejecuta de la manera siguiente:

- El estado de ejecución inicial es

$$\text{declaración} \\ (\{ \underbrace{[\underbrace{(\langle d \rangle, \emptyset)}_{\text{pila}}]}_{\text{multiconjunto}} \}, \emptyset)$$

Es decir, el almacén está vacío al inicio (no hay variables, conjunto vacío \emptyset) y el estado de ejecución inicial tiene una pila semántica que contiene una sola declaración semántica $(\langle d \rangle, \emptyset)$. La única diferencia con el capítulo 2 es que la pila semántica es un multiconjunto.

- En cada etapa, se selecciona una pila semántica ejecutable ST de MST , quedando MST' . Decimos que $MST = \{ST\} \uplus MST'$. (El operador \uplus denota unión de multiconjuntos.) Entonces se realiza una etapa de computación en ST de acuerdo a la semántica del capítulo 2, resultando en

$$(ST, \sigma) \rightarrow (ST', \sigma')$$

Entonces, la etapa de computación de la computación completa es

$$(\{ST\} \uplus MST', \sigma) \rightarrow (\{ST'\} \uplus MST', \sigma')$$

LLamamos esto una semántica de intercalación puesto que existe una secuencia global de etapas de computación. Los hilos, por turnos, van realizando cada uno un poco de trabajo.

- La elección de cuál ST seleccionar es realizada por el planificador de acuerdo a un conjunto bien definido de reglas, llamado el algoritmo de planificación.

Este algoritmo debe tener el cuidado de asegurar que las propiedades buenas, e.g., imparcialidad, se cumplan en cualquier computación. Un planificador real debe tener en cuenta muchas más cosas que sólo la imparcialidad. En la sección 4.2.4 se discuten muchas de estas ideas y se explica cómo funciona el planificador de Mozart.

- Si no hay pilas semánticas ejecutables en MST , entonces la computación no puede continuar:

- Si cada ST de MST está terminado, entonces decimos que la computación terminó.
- Si existe por lo menos un hilo, ST de MST , suspendido, que no puede ser recuperado (ver a continuación), entonces decimos que la computación se bloqueó.

*La declaración **thread***

La semántica de la declaración **thread** se define en términos de cómo su ejecución altera el multiconjunto MST . Una declaración **thread** nunca se bloquea. Si la pila

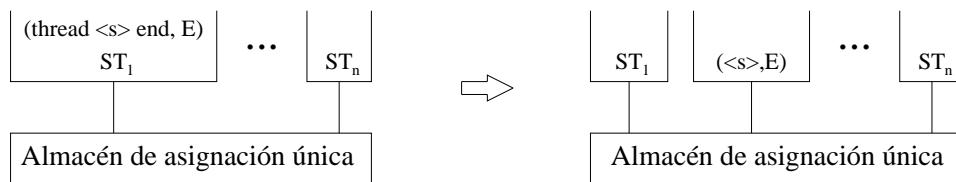


Figura 4.4: Ejecución de la declaración **thread**.

semántica seleccionada ST es de la forma $[(\text{thread } \langle d \rangle \text{ end}, E)] + ST'$, entonces el nuevo multiconjunto es $\{[(\langle d \rangle, E)]\} \uplus \{ST'\} \uplus MST'$. En otras palabras, agregamos una nueva pila semántica $[(\langle d \rangle, E)]$ que corresponda al hilo nuevo. En la figura 4.4 se ilustra esto. Podemos resumir lo anterior en la etapa de computación siguiente: $(\{[(\text{thread } \langle d \rangle \text{ end}, E)] + ST'\} \uplus MST', \sigma) \rightarrow (\{[(\langle d \rangle, E)]\} \uplus \{ST'\} \uplus MST', \sigma)$

Administración de la memoria

La administración de la memoria se extiende a los multiconjuntos como sigue:

- Una pila semántica terminada puede ser desalojada.
- Una pila semántica bloqueada puede ser recuperada como memoria libre si su condición de activación depende de una variable inalcanzable. En ese caso, la pila semántica nunca se volverá ejecutable de nuevo, y por ello eliminarla no cambia nada durante la ejecución.

Esto significa que la sencilla intuición del capítulo 2, que decía que “las estructuras de control son desalojadas y las estructuras de datos son recuperadas,” deja de ser completamente cierta en el modelo concurrente.

4.1.3. Ejemplo de ejecución

El primer ejemplo muestra cómo se crean los hilos y cómo se comunican usando sincronización de flujos de datos. Considere la declaración siguiente:

```
local B in
  thread B=true end
    if B then {Browse si} end
end
```

Por simplicidad, usaremos la máquina abstracta basada en sustituciones, introducida en la sección 3.3.

- Saltamos las etapas de computación iniciales y vamos directamente a la situación

cuando las declaraciones **thread** e **if** están en la pila semántica. El estado es

$$(\{[\mathbf{thread} \ b=\mathbf{true} \ \mathbf{end}], \mathbf{if} \ b \ \mathbf{then} \ \{\mathbf{Browse} \ \mathbf{si}\} \ \mathbf{end}]\}, \\ \{b\} \cup \sigma)$$

donde b es una variable en el almacén. Hay sólo una pila semántica que contiene las dos declaraciones.

- Después de ejecutar la declaración **thread**, el estado es:

$$(\{[b=\mathbf{true}], [\mathbf{if} \ b \ \mathbf{then} \ \{\mathbf{Browse} \ \mathbf{si}\} \ \mathbf{end}]\}, \\ \{b\} \cup \sigma)$$

Ahora hay dos pilas semánticas (“hilos”). La primera, que contiene $b=\mathbf{true}$, está lista. La segunda, que contiene la declaración **if**, está suspendida debido a que su condición de activación (b debe estar determinada) no se cumple.

- El planificador da el turno al hilo listo. Después de ejecutar una etapa, el estado es

$$(\{[], [\mathbf{if} \ b \ \mathbf{then} \ \{\mathbf{Browse} \ \mathbf{si}\} \ \mathbf{end}]\}, \\ \{b = \mathbf{true}\} \cup \sigma)$$

El primer hilo ha terminado (pila semántica vacía). El segundo hilo ahora está listo, pues b está determinada.

- Eliminamos la pila semántica vacía y ejecutamos la declaración **if**. El estado es

$$(\{\{\mathbf{Browse} \ \mathbf{si}\}\}, \\ \{b = \mathbf{true}\} \cup \sigma)$$

Queda todavía un hilo listo. La computación posterior desplegará **si**.

4.1.4. ¿Qué es concurrencia declarativa?

Miremos por qué podemos considerar que el modelo concurrente dirigido por los datos es una forma de programación declarativa. El principio básico de la programación declarativa es que la salida de un programa declarativo debe ser una función matemática de su entrada. En programación funcional, es claro lo que esto significa: el programa se ejecuta con algunos valores de entrada y cuando termina, ha devuelto algunos valores de salida. Los valores de salida son una función de los valores de entrada. ¿Pero qué significa esto en el modelo concurrente dirigido por los datos? Hay dos diferencias importantes con la programación funcional. La primera, las entradas y salidas no son necesariamente valores pues pueden contener variables no-ligadas. ¡Y la segunda, la ejecución podría no terminar pues las entradas pueden ser flujos que crecen indefinidamente! Miremos estos dos problemas, uno a la vez,

Concurrencia declarativa

y luego definimos lo que queremos dar a entender por concurrencia declarativa.²

Terminación parcial

Como un primer paso, consideremos el tema del crecimiento indefinido. Presentaremos la ejecución de un programa concurrente como una serie de etapas, donde cada etapa tiene una finalización natural. A continuación un ejemplo sencillo:

```
fun {Doblar xs}
  case xs of X|Xr then 2*X|{Doblar Xr} end
end
```

```
ys={Doblar xs}
```

El flujo de salida `ys` contiene los elementos del flujo de entrada `xs` multiplicados por 2. A medida que `xs` crece, `ys` crece también. El programa nunca termina. Sin embargo, si el flujo de entrada deja de crecer, entonces el programa dejará finalmente de ejecutarse. Este es un hecho importante. Decimos que el programa termina parcialmente. Si todavía no ha terminado completamente, entonces una ligadura posterior del flujo de entrada, causará que su ejecución continúe (hasta la siguiente terminación parcial!). Pero, si la entrada no cambia, entonces el programa no se ejecutará más.

Equivalencia lógica

Si las entradas se ligan a valores parciales, entonces el programa llegará finalmente a una terminación parcial, y las salidas quedarán ligadas a otros valores parciales. Pero, ¿en qué sentido son las salidas una “función” de las entradas? ¡Tanto las entradas como las salidas pueden tener variables no-ligadas! Por ejemplo, si `xs=1|2|3|xr`, entonces la invocación `ys={Doblar xs}` devuelve `ys=2|4|6|yr`, donde `xr` y `yr` son variables no-ligadas. ¿Qué significa que `ys` es una función de `xs`?

Para responder esta pregunta, tenemos que entender lo que significa que el contenido de un almacén sea “lo mismo” que el de otro. Presentaremos una definición sencilla a partir de principios básicos. (En los capítulos 9 (en CTM) y 13 (en CTM) se presenta una definición más formal basada en la lógica matemática). Antes de presentar la definición, miremos dos ejemplos para entender lo que está sucediendo. El primer ejemplo puede ligar `x` y `y` de dos maneras diferentes:

```
x=1 y=x    % Primer caso
y=x x=1    % Segundo caso
```

En el primer caso, el almacén finaliza con `x=1` y `y=x`. En el segundo caso, el almacén finaliza con `x=1` y `y=1`. En ambos casos, `x` y `y` terminan siendo ligados a 1. Esto significa que el contenido final del almacén en cada caso es el mismo. (Suponemos

2. En el capítulo 13 (en CTM) se presenta una definición formal de concurrencia declarativa que precisa las ideas de esta sección.

que los identificadores denotan las mismas variables del almacén en ambos casos.) Miremos un segundo ejemplo, esta vez con algunas variables no-ligadas:

```
X=foo(Y W) Y=Z    % Primer caso  
X=foo(Z W) Y=Z    % Segundo caso
```

En ambos casos, x queda ligado al mismo registro, salvo que el primer argumento puede ser diferente, y o z . Como $y=z$ (y y z están en la misma clase de equivalencia), esperamos, de nuevo, que el contenido final del almacén sea el mismo en ambos casos.

Ahora definamos lo que significa la equivalencia lógica. Definiremos la equivalencia lógica en términos de variables del almacén. Los ejemplos de arriba usaron identificadores, pero eso sólo fue para poder ejecutarlos. Un conjunto de ligaduras del almacén, como cada uno de los cuatro casos presentados arriba, se llama una restricción. Para cada variable x y restricción c , definimos $\text{valores}(x, c)$ como el conjunto de todos los valores que x puede tomar, de manera que c se cumpla. Entonces definimos:

Dos restricciones c_1 y c_2 son *lógicamente equivalentes* si: (1) contienen las mismas variables, y (2) para cada variable x , $\text{valores}(x, c_1) = \text{valores}(x, c_2)$.

Por ejemplo, la restricción $x = \text{foo}(y w) \wedge y = z$ (donde $x, y, z, y w$ son variables del almacén) es lógicamente equivalente a la restricción $x = \text{foo}(z w) \wedge y = z$. Esto debido a que $y = z$ obliga a que y y z tengan el mismo conjunto de valores posibles, de manera que $\text{foo}(y w)$ define el mismo conjunto de valores que $\text{foo}(z w)$. Note que las variables en un conjunto de equivalencia (como $\{y, z\}$) siempre tienen el mismo conjunto de valores posibles.

Concurrencia declarativa

Ahora podemos definir qué significa que un programa concurrente sea declarativo. En general, un programa concurrente puede tener muchas ejecuciones posibles. El hilo del ejemplo presentado arriba tiene al menos dos posibles ejecuciones, dependiendo del orden en que se realicen las ligaduras $x=1$ y $y=x$.³ El hecho clave es que todas estas ejecuciones tienen que terminar con el mismo resultado. Pero “el mismo” no significa que cada variable tenga que estar ligada a la misma cosa. Sólo significa equivalencia lógica. Esto nos lleva a la definición siguiente:

3. De hecho, hay más de dos, pues la ligadura $x=1$ puede realizarse antes o después de que el segundo hilo sea creado.

Concurrencia declarativa

Un programa concurrente es *declarativo* si, para todas las entradas, se cumple lo siguiente: Todas las ejecuciones con un conjunto dado de entradas tiene uno de los dos resultados siguientes: (1) ninguna termina o (2) todas terminan parcialmente, finalmente, y dan resultados que son lógicamente equivalentes. Como ejecuciones diferentes pueden introducir variables nuevas, suponemos que las variables nuevas en posiciones correspondientes son iguales.

Otra forma de decir esto es decir que existe un no-determinismo no observable. Esta definición es válida tanto para ejecución perezosa como para ejecución ansiosa. Aún más, cuando introduzcamos modelos no declarativos (e.g., con excepciones o estado explícito), usaremos esta definición como un criterio: si una parte de un programa obedece la definición, la consideraremos declarativa para el resto del programa.

Podemos probar que el modelo concurrente dirigido por los datos es declarativo de acuerdo a esta definición. Pero existen modelos declarativos aún más generales. El modelo concurrente dirigido por la demanda de la sección 4.5 también es declarativo. Este modelo es bastante general; tiene hilos y puede realizar ejecución tanto ansiosa como perezosa. El hecho de que sea declarativo es sorprendente.

Falla

Una falla es una terminación anormal de un programa declarativo que ocurre cuando intentamos colocar información inconsistente en el almacén, e.g., si ligáramos x tanto a 1 como a 2. El programa declarativo no puede continuar porque no existe un valor correcto para x .

La falla es una propiedad de todo o nada: si un programa concurrente declarativo resulta en una falla para un conjunto dado de entradas, entonces todas las ejecuciones posibles con ese mismo conjunto de entradas resultará en una falla. Esto tiene que ser así, o de lo contrario la salida no sería una función matemática de la entrada (pues unas ejecuciones podrían llevar a falla y otras no). Considere el ejemplo siguiente:

```
thread X=1 end
thread Y=2 end
thread X=Y end
```

Vemos que todas las ejecuciones alcanzarán finalmente una ligadura inconsistente y por tanto terminará.

La mayor parte de las fallas son debidas a errores del programador. Es muy drástico terminar todo un programa por culpa de un solo error del programador. Frecuentemente, nos gustaría continuar con la ejecución en lugar de terminarla, tal vez para reparar el error o simplemente para reportarlo. Una manera natural de hacer esto es utilizando excepciones. En el punto donde pueda ocurrir una falla lanzamos una excepción en lugar de terminar. El programa puede capturar la excepción y continuar la ejecución. El contenido del almacén será lo que era hasta el momento anterior a la falla.

¡Sin embargo, es importante caer en cuenta que la ejecución, después de lanzada

una excepción, deja de ser declarativa! Esto, porque el contenido del almacén no es el mismo, siempre, para todas las ejecuciones. En el ejemplo de arriba, justo antes de que se presente la falla, hay tres posibilidades para los valores de x & y : 1 & 1, 2 & 2, y 1 & 2. Si el programa continúa su ejecución, entonces podemos observar estos valores. Esto es un no determinismo observable. Decimos entonces que hemos “dejado el modelo declarativo.” A partir del instante en que se lanza una excepción, la ejecución deja de ser parte de un modelo declarativo, y pasa a hacer parte de un modelo más general (no declarativo).

Confinamiento de la falla

Si deseamos que la ejecución vuelva a ser declarativa, después de una falla, entonces tenemos que ocultar el no-determinismo. Esta es una responsabilidad del programador. Para el lector que tiene curiosidad sobre cómo hacer esto, le mostraremos cómo reparar el ejemplo previo. Suponga que x y y son visibles para el resto del programa. Si hay una excepción, nos arreglaremos para que x y y se liguen a valores por defecto. Si no hay excepción, entonces se ligan como antes.

```
declare X Y
local X1 Y1 S1 S2 S3 in
    thread
        try X1=1 S1=ok catch _ then S1,error end
    end
    thread
        try Y1=2 S2=ok catch _ then S2,error end
    end
    thread
        try X1=Y1 S3=ok catch _ then S3,error end
    end
if S1==error orelse S2==error orelse S3==error then
    X=1 % Valor por defecto para X
    Y=1 % Valor por defecto para Y
else X=X1 Y=Y1 end
end
```

Dos cosas tienen que ser reparadas. Primero, capturamos las excepciones de falla con las declaraciones `try`, de manera que la ejecución no termine con un error. (Ver sección 4.9.1 para más información sobre el modelo concurrente declarativo con excepciones.) Se necesita una declaración `try` por cada ligadura, pues cada ligadura puede fallar. Segundo, realizamos las ligaduras sobre las variables locales x_1 y y_1 , las cuales son invisibles para el resto del programa. Realizamos las ligaduras globales sólo cuando estamos seguros que no hubo falla.⁴

4.2. Técnicas básicas de programación de hilos

4. Se supone que $X=X_1$ y $Y=Y_1$ no fallarán.

Concurrencia declarativa

Hay muchas técnicas de programación, nuevas con respecto a las vistas en el modelo secuencial, precisamente porque son apropiadas al modelo concurrente. En esta sección se examinan las más sencillas, las cuales están basadas en un uso sencillo de la propiedad de flujo de datos en la ejecución de los hilos. También miraremos el planificador y veremos qué operaciones son posibles sobre los hilos. En secciones posteriores se explicarán técnicas más sofisticadas, incluyendo comunicación por flujos, concurrencia determinada por el orden, y otras.

4.2.1. Creación de hilos

La declaración **thread** crea un nuevo hilo:

```
thread
  proc {Contar N} if N>0 then {Contar N-1} end end
in
  {Contar 1000000}
end
```

Aquí se crea un hilo nuevo que corre concurrentemente con el hilo principal. La notación **thread ... end** también puede ser usada como una expresión:

```
declare X in
X = thread 10*10 end + 100*100
{Browse X}
```

Lo anterior es simplemente azúcar sintáctico para:

```
declare X in
local Y in
  thread Y=10*10 end
  X=Y+100*100
end
```

Una variable nueva, **Y**, de flujo de datos se crea para comunicarse entre el hilo principal y el hilo nuevo. La suma se bloquea hasta que el cálculo $10*10$ se termina.

Cuando un hilo no tiene más declaraciones para ejecutar, entonces termina. Cada hilo que no está terminado y que no está suspendido, se ejecutará finalmente. Se dice que los hilos se planifican imparcialmente. La ejecución de hilos se implementa con planificación preventiva. Es decir, si hay más de un hilo listo para ejecutarse, entonces cada hilo tendrá tiempo de procesador en intervalos discretos llamados lapsos. No es posible que un hilo se tome todo el tiempo de procesador.

4.2.2. Hilos y el browser

El browser es un buen ejemplo de un programa que funciona bien en un ambiente concurrente. Por ejemplo:

```
thread {Browse 111} end
{Browse 222}
```

¿En qué orden se despliegan los valores 111 y 222? ¡La respuesta es, cualquier orden es posible! ¿Es posible que algo como 112122 se despliegue, o peor, que el browser se comporte erróneamente? A primera vista, puede parecer posible, pues

el browser tiene que ejecutar muchas declaraciones para desplegar cada valor 111 y 222. Si no se toman precauciones especiales, entonces estas declaraciones pueden ejecutarse, incluso, casi en cualquier orden. Pero el browser está diseñado para un ambiente concurrente, y por ello nunca desplegará intercalaciones extrañas. A cada invocación del browser se le asigna su espacio, en la misma ventana del browser, para desplegar su argumento. Si el argumento contiene una variable no-ligada que es ligada posteriormente, entonces la pantalla será actualizada cuando la variable sea ligada. De esta manera, el browser desplegará correctamente múltiples flujos que crecen concurrentemente. Por ejemplo:

```
declare X1 X2 Y1 Y2 in
thread {Browse X1} end
thread {Browse Y1} end
thread X1=todos|los|caminos|x2 end
thread Y1=todos|los|caminos|y2 end
thread X2=conducen|a|roma|_ end
thread Y2=conducen|a|rodas|_ end
```

despliega correctamente los dos flujos

```
todos|los|caminos|conducen|a|roma|_
todos|los|caminos|conducen|a|rodas|_
```

en partes separadas de la ventana del browser. En este capítulo y en capítulos posteriores veremos cómo escribir programas concurrentes que se comporten correctamente, como el browser.

4.2.3. Computación por flujo de datos con hilos

Miremos qué podemos hacer agregando hilos a programas sencillos. Es importante recordar que cada hilo es un hilo de flujo de datos, i.e., se suspende según la disponibilidad de los datos.

Comportamiento sencillo de flujo de datos

Empezamos por observar el comportamiento de flujo de datos en un cálculo sencillo. Considere el programa siguiente:

```
declare x0 x1 x2 x3 in
thread
y0 y1 y2 y3 in
{Browse [y0 y1 y2 y3]}
y0=x0+1
y1=x1+y0
y2=x2+y1
y3=x3+y2
{Browse listo}
end
{Browse [x0 x1 x2 x3]}
```

Concurrencia declarativa

Si se ejecuta este programa, entonces el browser desplegará todas las variables como variables no-ligadas. Observe qué pasa cuando se ejecutan las siguientes declaraciones, una a la vez:

```
x0=0
x1=1
x2=2
x3=3
```

Con cada declaración, el hilo se reanuda, ejecuta una suma, y se suspende de nuevo. Es decir, cuando x_0 se liga, el hilo puede ejecutar $y_0=x_0+1$. Allí se suspende de nuevo porque necesita el valor de x_1 para poder ejecutar $y_1=x_1+y_0$, y así sucesivamente.

Utilizando un programa declarativo en un marco concurrente

Tomemos un programa del capítulo 3 y miremos cómo se comporta cuando se utiliza en un marco concurrente. Considere el ciclo `ForAll`, el cual se definió como sigue:

```
proc {ForAll L P}
  case L of nil then skip
  [] X|L2 then {P X} {ForAll L2 P} end
end
```

¿Qué pasa cuando lo ejecutamos en un hilo?:

```
declare L in
thread {ForAll L Browse} end
```

Si L está no-ligada, entonces el hilo se suspende inmediatamente. Podemos ligar L en otros hilos:

```
declare L1 L2 in
thread L=1|L1 end
thread L1=2|3|L2 end
thread L2=4|nil end
```

¿Cuál es la salida? ¿Tiene alguna diferencia con el resultado de la invocación `{ForAll [1 2 3 4] Browse}`? ¿Cuál es el efecto de utilizar `ForAll` en un marco concurrente?

Una versión concurrente de Map

Presentamos una versión concurrente de la función `Map` definida en la sección 3.6.4:

```
fun {Map Xs F}
  case Xs of nil then nil
  [] X|Xr then thread {F X} end|{Map Xr F} end
end
```

La declaración `thread` se usa en este caso como una expresión. Exploraremos el comportamiento de este programa. Si entramos las declaraciones siguientes:

```
declare F Xs Ys Zs
{Browse thread {Map Xs F} end}
```

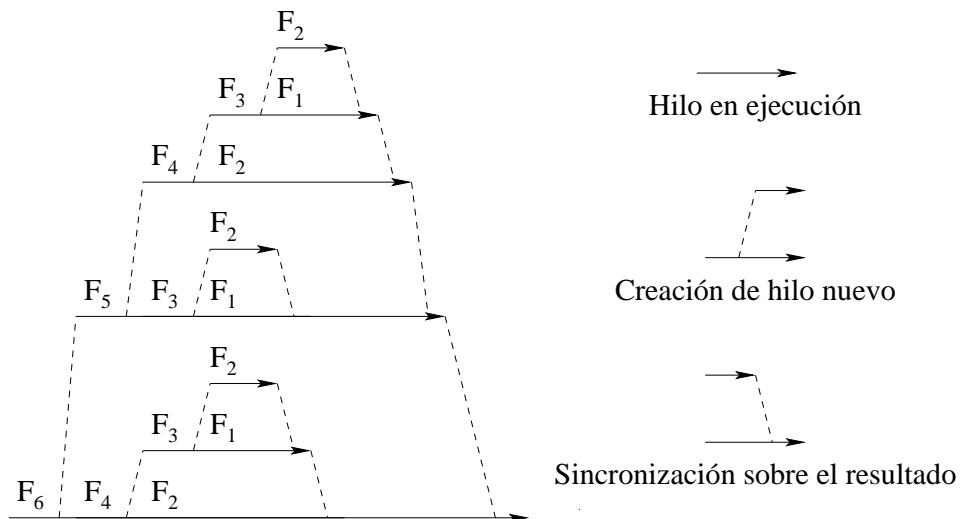


Figura 4.5: Creación de hilos en la invocación {Fib 6}.

entonces se crea un hilo nuevo para ejecutar {Map xs f}. Esta ejecución se suspenderá inmediatamente en la declaración **case** debido a que xs es no-ligada. Si entramos ahora las siguientes declaraciones (¡sin un **declare!**):

```
xs=1|2|ys
fun {f x} x*x end
```

entonces el hilo principal recorrerá la lista, creando dos hilos por cada uno de los dos primeros argumentos de la lista, **thread {f 1} end** y **thread {f 2} end**, y luego se suspenderá de nuevo sobre la cola de la lista ys. Finalmente, al realizar

```
ys=3|zs
zs=nil
```

se creará un tercer hilo con **thread {f 3} end** y terminará la computación del hilo principal. Los tres hilos terminarán, y como resultado final se tendrá la lista [1 4 9]. Note que el resultado es el mismo que en el caso secuencial de Map, sólo que éste puede ser obtenido incrementalmente si la entrada llega incrementalmente. La función Map secuencial se ejecuta en “lote”: el cálculo no da resultado hasta que no se recibe la entrada completa, y entonces devuelve el resultado completo.

Una versión concurrente de la función de Fibonacci

Presentamos un programa concurrente, diseñado con la estrategia de dividir y conquistar, para calcular la función de Fibonacci:

```
fun {Fib x}
  if x=<2 then 1
  else thread {Fib x-1} end + {Fib x-2} end
end
```

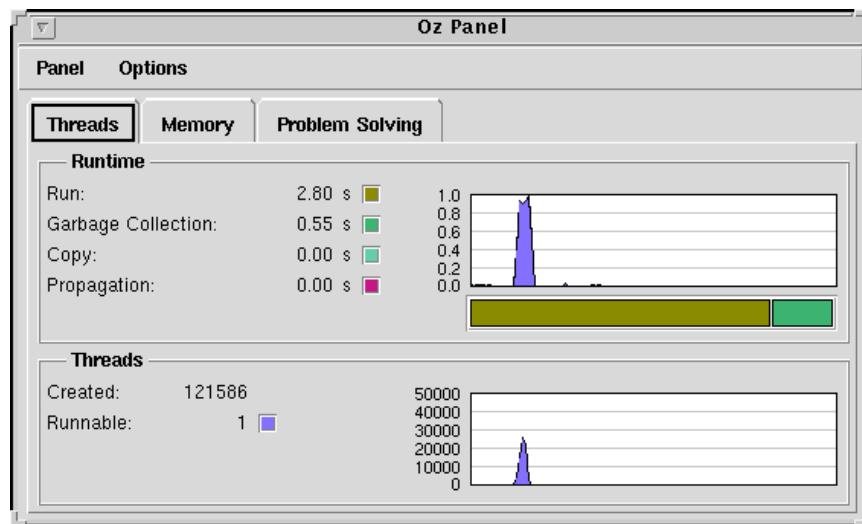


Figura 4.6: El Panel de Oz mostrando la creación de hilos en `X={Fib 26}`.

Este programa está basado en la función Fibonacci secuencial resursiva; la única diferencia es que la primera invocación recursiva se realiza en un hilo propio. ¡Este programa crea un número exponencial de hilos! En la figura 4.5 se muestran todas las creaciones de hilos y sincronizaciones para la invocación `{Fib 6}`. Un total de ocho hilos están envueltos en este cálculo. Se puede usar este programa para probar cuántos hilos se pueden crear en su instalación de Mozart. Por ejemplo, alimento la interfaz con

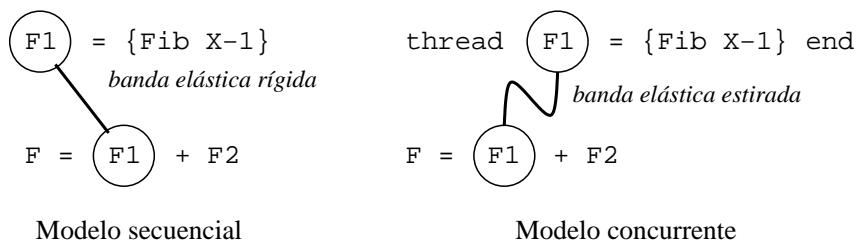
```
{Browse {Fib 26}}
```

mientras observa el Oz Panel para ver cuántos hilos se están ejecutando. Si `{Fib 26}` termina muy rápidamente, ensaye con un argumento más grande. El Oz Panel, mostrado en la figura 4.6, es una herramienta de Mozart que provee información sobre el comportamiento del sistema (tiempo de ejecución, utilización de la memoria, hilos, etc.). Para lanzar el Oz Panel, seleccione la entrada Oz Panel del menú de Oz en la interfaz interactiva.

Flujo de datos y bandas elásticas

Por ahora, es claro que cualquier programa concurrente del capítulo 3 se puede volver concurrente colocando `thread ... end` alrededor de algunas de sus declaraciones y expresiones. Como cada variable de flujo de datos será ligada al mismo valor que antes, el resultado final de la versión concurrente será exactamente el mismo que el de la versión secuencial original.

Una manera de ver esto intuitivamente es por medio de bandas elásticas. Cada

**Figura 4.7:** Flujo de datos y bandas elásticas.

variable de flujo de datos tiene su banda elástica propia. Un borde de la banda está pegado al sitio donde la variable es ligada y el otro borde está ligado al sitio donde se utiliza la variable. En la figura 4.7 se muestra qué pasa en los modelos secuencial y concurrente. En el modelo secuencial, la ligadura y la utilización están, normalmente, cerca la una de la otra, de manera que la banda no se estira mucho. En el modelo concurrente, la ligadura y la utilización pueden estar en hilos diferentes, de manera que la banda se estira. Pero nunca se rompe: el usuario ve siempre el valor correcto.

Concurrencia económica y estructura de un programa

Frecuentemente, la utilización de hilos hace posible mejorar la estructura de un programa, e.g., para hacerlo más modular. La mayoría de los programas grandes tienen muchos lugares en los cuales se pueden usar los hilos para esto. Idealmente, el sistema de programación debería soportar esto con hilos que utilicen pocos recursos computacionales. En este aspecto, el sistema Mozart es excelente. Los hilos son tan económicos que uno puede permitirse el lujo de crearlos en grandes números. Por ejemplo, los computadores personales del año 2003 tienen, típicamente, al menos 256MB de memoria activa, con la cual se pueden soportar más de 100000 hilos activos simultáneos.

Si el uso de la concurrencia lleva a que su programa tenga una estructura más sencilla, entonces úsela sin dudarlo. Pero tenga en mente que, aunque los hilos sean económicos, los programas secuenciales lo son aún más. Los programas secuenciales siempre son más rápidos que los concurrentes que tienen la misma estructura. El programa `Fib` de la sección 4.2.3 es más rápido si la declaración **thread** se elimina. Se deberían crear los hilos sólo cuando el programa los necesite. Por otro lado, no dude en crear un hilo si eso mejora la estructura del programa.

4.2.4. Planificación de hilos

Hemos visto que el planificador debería ser imparcial, i.e., cada hilo listo debe ejecutarse finalmente. Un planificador real hace mucho más que garantizar imparcialidad. Miremos otros temas relacionados y cómo los tiene en cuenta el planificador.

Lapsos de tiempo

El planificador coloca todos los hilos listos en una cola. En cada paso, saca el primer hilo de la cola, lo deja ejecutarse un número de etapas, y luego lo coloca de nuevo en la cola. A este proceso se le llama planificación round-robin. Esta planificación garantiza que el tiempo de procesador es asignado equitativamente a todos los hilos listos.

Sería ineficiente dejar ejecutar una sola etapa de computación a cada hilo antes de ponerlo de nuevo en la cola. El sobrecoste de la administración de la cola (sacar los hilos y volverlos a meter) relativo a la computación real sería muy alto. Por eso, el planificador deja a cada hilo ejecutar muchas etapas de computación antes de devolverlo a la cola. Cada hilo tiene asignado un tiempo máximo de ejecución antes que el planificador lo detenga. Este intervalo de tiempo es llamado un lapso de tiempo o quantum. Una vez se termina el lapso de tiempo a un hilo, el planificador detiene su ejecución y lo coloca en la cola. Detener de esta manera un hilo en ejecución es llamado prevención.

Para asegurarse que a cada hilo se le asigna aproximadamente la misma fracción de tiempo de procesador, los planificadores de hilos tienen dos enfoques. La primera forma consiste en contar las etapas de computación y asignar el mismo número a cada hilo. La segunda forma consiste en usar un temporizador de hardware que asigne el mismo tiempo a cada hilo. Ambos enfoques son prácticos. Comparémoslos:

- El enfoque de conteo tiene la ventaja que la ejecución del planificador es determinística, i.e., la ejecución del mismo programa dos veces realizará la prevención de los hilos exactamente en los mismos instantes. Los planificadores determinísticos se usan con frecuencia para aplicaciones críticas en tiempo real, donde se deben garantizar los tiempos.
- El enfoque del temporizador es más eficiente, debido a que el temporizador es soportado por hardware. Sin embargo, el planificador deja de ser determinístico. Cualquier evento en el sistema operativo, e.g., una operación sobre un disco o sobre la red, cambiará los instantes precisos en que sucede la prevención.

El sistema Mozart utiliza un temporizador de hardware.

Niveles de prioridad

Muchas aplicaciones necesitan tener mayor control sobre la forma en que el tiempo de procesador se comparte entre los hilos. Por ejemplo, en el transcurso de una computación, puede suceder un evento que requiera tratamiento urgente, cambiando el curso “normal” de la computación. Por otro lado, se debe evitar que las computaciones urgentes “maten de hambre” las computaciones normales, i.e., las desacelere desmesuradamente.

Un compromiso que funciona bien en la práctica es tener niveles de prioridad para los hilos. A cada nivel de prioridad se le asigna un porcentaje mínimo de tiempo de procesador. Dentro de cada nivel de prioridad, los hilos comparten el

tiempo de procesador, imparcialmente, como antes. El sistema Mozart utiliza esta técnica, con tres niveles de prioridad: alto, medio, y bajo. Hay tres colas, una por cada nivel de prioridad. Por defecto, el tiempo de procesador se divide entre las prioridades en razones de 100:10:1 para prioridades alta-media-baja. Esto se implementa en una forma muy sencilla: Por cada diez lapsos de tiempo asignados a hilos de prioridad alta, un lapso de tiempo es asignado a hilos de prioridad media. Similarmente, por cada diez lapsos de tiempo asignados a hilos de prioridad media, un lapso de tiempo es asignado a hilos de prioridad baja. Esto significa que los hilos de prioridad alta, si existen, se reparten entre ellos al menos 100/111 (cerca del 90 %) del tiempo de procesador. Similarmente, los hilos de prioridad media, si existen, se reparten entre ellos al menos 10/111 (cerca del 9 %) del tiempo de procesador. Y por último, los hilos de prioridad baja, si existen, se reparten entre ellos al menos 1/111 (cerca del 1 %) del tiempo de procesador. Estos porcentajes son límites inferiores garantizados. Si existen pocos hilos, entonces los porcentajes podrían ser más altos. Por ejemplo, si no hay hilos de prioridad alta, entonces un hilo de prioridad media puede conseguir hasta 10/11 del tiempo de procesador. En Mozart, las razones alta-media y media-baja son 10 por defecto. Éstas pueden ser cambiadas con el módulo `Property`.

Herencia de prioridades

Cuando un hilo crea un hilo hijo, entonces se le asigna al hijo la misma prioridad del padre. Esto es especialmente importante en hilos de prioridad alta. En una aplicación, estos hilos se usan para “administración de urgencias,” i.e., para realizar el trabajo que debe ser realizado primero que el trabajo normal. La parte de la aplicación que realiza la administración de urgencias puede ser concurrente. Si el hilo hijo de un hilo de prioridad alta pudiera tener, digamos, prioridad media, entonces existiría una “ventana” de tiempo corta durante la cual el hilo hijo es de prioridad media, hasta que el padre o el hijo cambien la prioridad del hilo. La existencia de esta ventana de tiempo es suficiente para evitar que, por muchos lapsos de tiempo, el hilo hijo sea planificado, pues el hilo queda en la cola de hilos de prioridad media. Esto puede llevar a errores de coordinación muy difíciles de rastrear. Por lo tanto, un hilo hijo no debe recibir nunca una asignación de prioridad inferior a la de su padre.

Duración de un lapso de tiempo

¿Cuál es el efecto de la duración de un lapso de tiempo? Un lapso corto permite una concurrencia “finamente granulada”: los hilos reaccionan rápidamente a los eventos externos. Pero, si el lapso es demasiado corto, entonces el sobrecosto de los cambios entre hilos se vuelve significativo. Otro asunto es cómo implementar la prevención: ¿el mismo hilo lleva la cuenta de cuánto tiempo ha estado ejecutándose, o esto se hace externamente? Ambas soluciones son viables, pero la segunda es mucho más fácil de implementar. Los sistemas operativos de multitareas modernos,

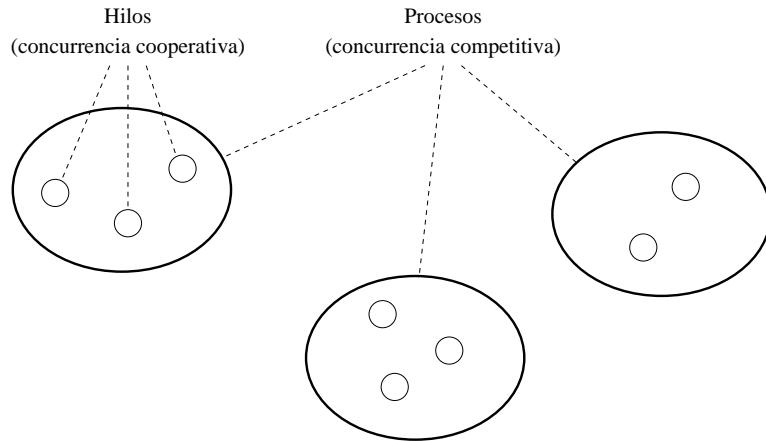
Concurrencia declarativa

Figura 4.8: Concurrencia cooperativa y competitiva.

tales como Unix, Windows 2000/XP, o Mac OS X, cuentan con interrupciones del temporizador que se pueden usar para lanzar la prevención. Estas interrupciones llegan a una frecuencia imparcialmente baja, de 60 a 100 por segundo. El sistema Mozart utiliza esta técnica.

Un lapso de tiempo de 10 ms puede parecer suficientemente corto, pero para algunas aplicaciones es demasiado largo. Por ejemplo, suponga que la aplicación tiene 100000 hilos activos. Entonces a cada hilo se le asignaría un lapso de tiempo cada 1000 segundos. Esto puede ser un tiempo muy largo de espera. En la práctica, hemos encontrado que esto no es un problema. En las aplicaciones con muchos hilos, tales como grandes programas por restricciones (ver capítulo 12 (en CTM)), normalmente los hilos dependen fuertemente de los otros hilos y no del mundo externo. Cada hilo usa sólo una pequeña parte de su lapso de tiempo antes de ceder el paso a otro hilo.

Por otro lado, es posible imaginar una aplicación con muchos hilos, cada uno de los cuales interactúa con el mundo externo independientemente de los otros hilos. Para una aplicación como esa, es claro que Mozart, así como las últimas versiones de los sistemas operativos Unix, Windows, o Mac OS X, es insatisfactorio. El mismo hardware de un computador personal es insatisfactorio. Lo que se necesita es un sistema duro de cómputo en tiempo real, el cual utiliza un tipo especial de hardware junto con un tipo especial de sistema operativo. Este tipo de computación está fuera del alcance de este libro.

4.2.5. Concurrencia cooperativa y competitiva

Los hilos se han previsto para realizar concurrencia cooperativa, y no para realizar concurrencia competitiva. La concurrencia cooperativa tiene que ver con entidades que trabajan juntas por lograr un objetivo global. Los hilos soportan este tipo de

conurrencia, e.g., cualquier hilo puede cambiar las razones de tiempo entre las tres prioridades como lo veremos. Los hilos se han previsto para ser usados por aplicaciones que se ejecutan en un ambiente donde todas las partes confían la una en la otra.

Por otro lado, la concurrencia competitiva tiene que ver con entidades que tienen un objetivo local, i.e., cada una trabaja por sí misma. Las entidades están interesadas solamente en su propio desempeño, no en el desempeño global. Normalmente, la concurrencia competitiva es administrada por el sistema operativo en términos de un concepto llamado un proceso.

Esto significa que las aplicaciones tienen frecuentemente una estructura de dos niveles, como se muestra en la figura 4.8. En el nivel más alto, hay un conjunto de procesos del sistema operativo interactuando entre sí, realizando concurrencia competitiva. Normalmente, los procesos pertenecen a aplicaciones diferentes, con objetivos diferentes, tal vez conflictivos. Dentro de cada proceso, hay un conjunto de hilos interactuando entre ellos, realizando concurrencia cooperativa. Normalmente, los hilos dentro de un proceso pertenecen a la misma aplicación.

La concurrencia competitiva es soportada en Mozart por su modelo de computación distribuida y por el módulo `Remote`. El módulo `Remote` crea un proceso del sistema operativo, independiente, con sus propios recursos computacionales. Una computación competitiva pueden entonces ser colocada dentro de este proceso. Esto es relativamente fácil de programar gracias a que el modelo distribuido es transparente a la red: el mismo programa puede ejecutarse con diferentes estructuras de distribución, i.e., sobre diferentes conjuntos de procesos, y siempre se obtendrá el mismo resultado.⁵

4.2.6. Operaciones sobre hilos

Los módulos `Thread` y `Property` proveen un buen número de operaciones pertinentes sobre hilos. Algunas de estas operaciones se presentan en la tabla 4.2. La prioridad `P` de un hilo es uno de los tres átomos `low`, `medium`, y `high`. El estado de un hilo es uno de los tres átomos `runnable`, `blocked`, y `terminated`. Cada hilo tiene un nombre único, el cual hace referencia al hilo cuando se realizan operaciones sobre él. El nombre del hilo es un valor de tipo `Name`. La única manera de obtener un nombre de un hilo, es por medio del mismo hilo invocando `Thread.this`. No es posible que otro hilo obtenga el nombre sin la cooperación del hilo original. Esto permite controlar rigurosamente el acceso a los nombres de hilos. El procedimiento del sistema

```
{Property.put priorities p(high:X medium:Y)}
```

5. Esto es cierto mientras ningún proceso falle. Vea el capítulo 11 (en CTM) para ejemplos y mayor información.

Operación	Descripción
{Thread.this}	Devuelve el nombre del hilo actual
{Thread.state H}	Devuelve el estado actual de H
{Thread.suspend H}	Suspende H (detiene su ejecución)
{Thread.resume H}	Reanuda H (deshace la suspensión)
{Thread.preempt H}	Previene H
{Thread.terminate H}	Termina H inmediatamente
{Thread.injectException H E}	Lanza la excepción E dentro de H
{Thread.setPriority H P}	Asigna P como la prioridad de H
{Thread.setThisPriority P}	Asigna P como la prioridad del hilo actual
{Property.get priorities}	Devuelve las razones entre las prioridades del sistema
{Property.put priorities p(high:X medium:Y)}	Asigna las razones entre las prioridades del sistema

Tabla 4.2: Operaciones sobre hilos.

asigna la razón de tiempo de procesador a x:1 entre prioridad alta y prioridad media y a y:1 entre prioridad media y prioridad baja. x y y son enteros. Si ejecutamos

```
{Property.put priorities p(high:10 medium:10)}
```

entonces por cada 10 lapsos de tiempo asignados a hilos ejecutables (listos) de prioridad alta, el sistema asignará un lapso de tiempo a hilos de prioridad media, y similarmente entre hilos de prioridad media y prioridad baja. Este es el valor por defecto. Dentro del mismo nivel de prioridad, la planificación es imparcial y round robin.

4.3. Flujos

La técnica más útil para programación concurrente en el modelo concurrente declarativo es la utilización de flujos para la comunicación entre hilos. Un flujo es una lista de mensajes potencialmente ilimitada, i.e., es una lista cuya cola es una variable de flujo de datos no-ligada. Para enviar un mensaje se extiende el flujo con un elemento: se liga la cola a una pareja que contiene el mensaje y una variable nueva no-ligada. Para recibir un mensaje simplemente se lee un elemento del flujo. Un hilo que se comunica a través de flujos es una especie de “objeto activo” que llamaremos un objeto flujo. No hay necesidad de mecanismos de protección con candados o de exclusión mutua pues cada variable es ligada únicamente por un hilo.

La programación por flujos es un enfoque bastante general que se puede aplicar en muchos dominios. Éste es el concepto subyacente en las tuberías Unix. Morrison lo utilizó con buenos resultados en aplicaciones de negocios, en un enfoque que él llama

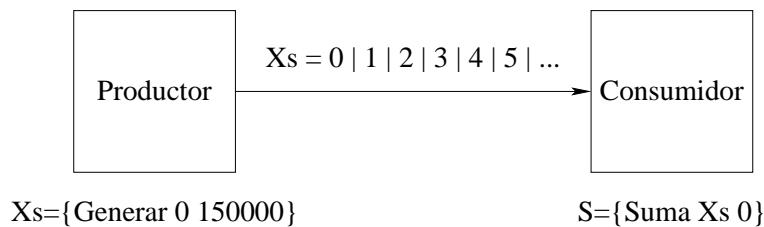


Figura 4.9: Comunicación por flujos productor/consumidor.

“programación basada en flujos” [119]. En este capítulo miramos un caso especial de programación por flujos, a saber, programación determinística por flujos, en el cual cada objeto flujo conoce siempre, para cada entrada, de donde provendrá el siguiente mensaje. Este caso es interesante porque es declarativo, y aún así es bastante útil. Posponemos la mirada a la programación no-determinística por flujos hasta el capítulo 5.

4.3.1. Esquema básico productor/consumidor

En esta sección se explica cómo funcionan los flujos y se muestra cómo programar un esquema productor/consumidor asincrónico con flujos. En el modelo declarativo, un flujo se representa por medio de una lista cuya cola es una variable no-ligada:

```
declare Xs Xs2 in
Xs=0|1|2|3|4|Xs2
```

Un flujo se crea incrementalmente ligando la cola a una lista nueva con una cola nueva:

```
declare Xs3 in
Xs2=5|6|7|Xs3
```

Un hilo, llamado el productor, crea el flujo de esta manera, y otros hilos, llamados los consumidores, leen el flujo. Como la cola del flujo es una variable de flujo de datos, entonces los consumidores leerán el flujo tal como fue creado. El programa siguiente genera asincrónicamente un flujo de enteros y los suma:

```
fun {Generar N Limite}
  if N<Limite then
    N|{Generar N+1 Limite}
  else nil end
end

fun {Suma Xs A}
  case Xs
  of X|Xr then {Suma Xr A+X}
  [] nil then A
  end
end
```

Concurrencia declarativa

```
local Xs S in
  thread Xs={Generar 0 150000} end % Hilo productor
  thread S={Suma Xs 0} end           % Hilo consumidor
  {Browse S}
end
```

En la figura 4.9 se presenta una forma particularmente agradable de definir este patrón, utilizando una notación gráfica precisa. Cada rectángulo denota una función recursiva dentro de un hilo, la flecha continua denota un flujo, y la flecha va dirigida del productor al consumidor. Después de terminados los cálculos, se despliega en el browser 11249925000. El productor, Generar, y el consumidor, Suma, se ejecutan en sus propios hilos. Ellos se comunican a través de la variable compartida xs, la cual es ligada a un flujo de enteros. La declaración **case** dentro de Suma se bloquea mientras xs no esté ligada (no llegan más elementos) y reanuda su ejecución cuando xs se liga (llegan nuevos elementos).

En el consumidor, el comportamiento de flujo de datos de la declaración **case** bloquea la ejecución hasta la llegada del siguiente elemento del flujo. Esto sincroniza el hilo del consumidor con el hilo del productor. Esperar a que una variable de flujo de datos sea ligada es el mecanismo básico de sincronización y comunicación en el modelo concurrente declarativo.

Utilizando un iterador de alto orden

La invocación recursiva de Suma tiene un argumento A que representa la suma de todos los elementos leídos hasta el momento. Este argumento junto con la salida de la función conforman un acumulador como los vistos en el capítulo 3. Podemos deshacernos del acumulador utilizando una abstracción de ciclo:

```
local Xs S in
  thread Xs={Generar 0 150000} end
  thread S={FoldL Xs fun {$ x y} x+y end 0} end
  {Browse S}
end
```

Gracias a las variables de flujo de datos, la función `FoldL` no tiene problemas de funcionamiento en un contexto concurrente. Deshacerse de un acumulador utilizando un iterador de alto orden es una técnica general. El acumulador realmente no ha desaparecido, sino que está oculto dentro del iterador. Pero la escritura del programa es más sencilla pues el programador no tiene que razonar en términos de estado. El módulo `List` cuenta con muchas abstracciones de ciclo y otras operaciones de alto orden que pueden ser utilizadas para ayudar a implementar funciones recursivas.

Múltiples lectores

Podemos introducir múltiples consumidores sin cambiar el programa de ninguna manera. Por ejemplo, aquí se presentan tres consumidores, leyendo el mismo flujo:

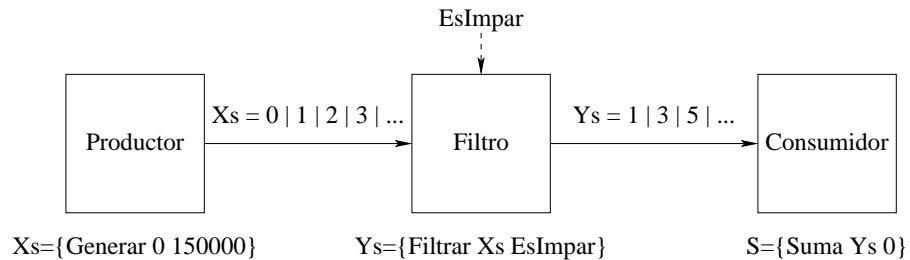


Figura 4.10: Filtrando un flujo.

```

local Xs S1 S2 S3 in
  thread Xs={Generar 0 150000} end
  thread S1={Suma Xs 0} end
  thread S2={Suma Xs 0} end
  thread S3={Suma Xs 0} end
end
  
```

Cada hilo consumidor recibirá los elementos del flujo independientemente de los otros. Los consumidores no interfieren con los otros porque ellos realmente no “consumen” el flujo; sólo lo leen.

4.3.2. Transductores y canales

Podemos colocar un tercer objeto flujo entre el productor y el consumidor. Este objeto flujo lee el flujo del consumidor y crea otro flujo que es leído por el consumidor. A este objeto lo llamamos un transductor. En general, una secuencia de objetos flujo, cada uno de los cuales alimenta al siguiente, es llamada un canal.⁶ Al productor también se le llama la fuente y al consumidor se le llama el receptor. Miremos algunos canales con diferentes tipos de transductores:

Filtering a stream

Uno de los transductores más sencillos es el filtro, el cual devuelve en el flujo de salida sólo aquellos elementos del flujo de entrada que satisfacen una condición específica. Una manera sencilla de hacer un filtro es colocar una invocación a la función `Filter` vista en el capítulo 3, dentro de su propio hilo. Por ejemplo, podemos dejar pasar sólo aquellos elementos que sean enteros impares:

6. Nota del traductor: *pipeline*, en inglés.

Concurrencia declarativa

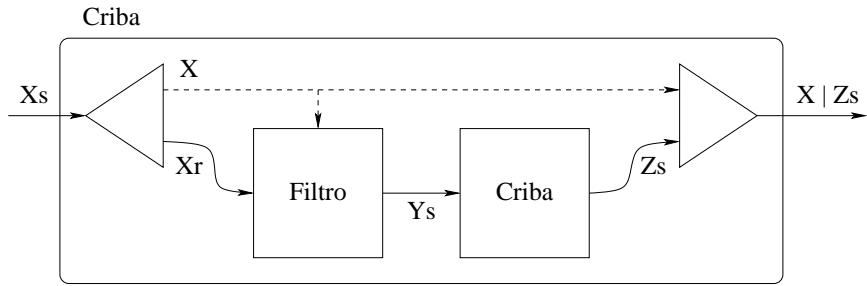


Figura 4.11: Una criba de números primos con flujos.

```

local Xs Ys S in
  thread Xs={Generar 0 150000} end
  thread Ys={Filter Xs EsImpar} end
  thread S={Suma Ys 0} end
  {Browse S}
end

```

donde `EsImpar` es una función booleana de un argumento que devuelve `true` sólo para enteros impares:

```
fun {EsImpar X} X mod 2 \= 0 end
```

En la figura 4.10 se muestra este patrón. Esta figura introduce algo más de notación gráfica, la flecha punteada, la cual denota un único valor (un argumento de la función que no es un flujo).

Criba de Eratóstenes

Como ejemplo de mayor complejidad, definamos un canal que implemente la criba de números primos de Eratóstenes. La salida de la criba es un flujo que sólo contiene números primos. Este programa se llama “criba” pues funciona filtrando sucesivamente los números no primos del flujo, hasta que sólo quedan números primos. Los filtros se crean dinámicamente a medida que se necesitan. El productor genera un flujo de enteros consecutivos a partir del 2. La criba toma el primer elemento y crea un filtro que elimine los múltiplos de ese elemento. Luego se invoca recursivamente sobre el flujo de los elementos que aún quedan. En la figura 4.11 se muestra el funcionamiento de la criba. Se introduce como notación gráfica adicional el triángulo, el cual denota el quitar el primer elemento a un flujo o colocar un nuevo primer elemento a un flujo. Esta es la definición de la criba:

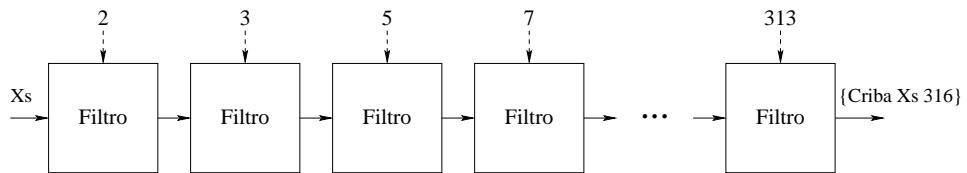


Figura 4.12: Canal de filtros generado por {Criba Xs 316}.

```

fun {Criba Xs}
  case Xs
  of nil then nil
  [] X|Xr then Ys in
    thread Ys={Filter Xr fun {$ Y} Y mod X \= 0 end} end
    X|{Criba Ys}
  end
end
  
```

Esta definición es bastante sencilla, considerando que se está construyendo dinámicamente un canal de actividades concurrentes.

```

local Xs Ys in
  thread Xs={Generar 2 100000} end
  thread Ys={Criba Xs} end
  {Browse Ys}
end
  
```

Este programa despliega los números primos hasta 100000. Este programa es bastante simplista, pues crea demasiados hilos, a saber, uno por número primo. No es necesario un número tan grande de hilos pues es fácil ver que la generación de los números primos hasta n requiere de filtrar los múltiplos sólo hasta \sqrt{n} .⁷ Podemos modificar el programa para crear filtros sólo hasta ese límite:

```

fun {Criba Xs M}
  case Xs
  of nil then nil
  [] X|Xr then Ys in
    if X=<M then
      thread Ys={Filter Xr fun {$ Y} Y mod X \= 0 end} end
    else Ys=Xr end
    X|{Criba Ys M}
  end
end
  
```

Con una lista de 100000 elementos, podemos invocar esta función como {Criba Xs 316} (pues $316 = \lfloor \sqrt{100000} \rfloor$). Así se crea dinámicamente el canal de filtros mostrado en la figura 4.12. Como los factores pequeños son más comunes que los factores grandes, la mayoría de los filtros actuales se hacen en los filtros tempranos.

7. Si el factor f es mayor que \sqrt{n} , entonces existe un factor n/f menor que \sqrt{n} .

4.3.3. Administración de recursos y mejora del rendimiento

¿Qué pasa si el productor genera elementos a mayor velocidad de la que el consumidor puede consumirlos? Si esto dura suficiente tiempo, entonces los elementos no consumidos se van a apilar y a monopolizar los recursos del sistema. Los ejemplos que vimos atrás no hacen nada para evitar esto. Una forma de resolver el problema es limitar la tasa a la cual el productor genera elementos nuevos, de manera que alguna condición global (como una máxima utilización de un recurso) se satisfaga. Esto se llama control del flujo. Para ello se requiere que el consumidor envíe al productor alguna información. Miremos cómo implementar esto.

4.3.3.1. Control del flujo con concurrencia dirigida por la demanda

La forma más sencilla de control del flujo se llama concurrencia dirigida por la demanda, o ejecución perezosa. En esta técnica, el productor sólo genera elementos cuando el consumidor explícitamente lo solicita. (La técnica anterior, donde el productor genera un elemento cada que él lo desea, se llama ejecución dirigida por la oferta, o ejecución ansiosa.) La ejecución perezosa requiere de un mecanismo para que el consumidor le informe al productor cuándo necesita un elemento nuevo. La manera más sencilla de hacer esto es usando una variable de flujo de datos. Por ejemplo, el consumidor puede extender su flujo de entrada cuando necesite un elemento nuevo. Es decir, el consumidor liga el final del flujo a una pareja $x|x_r$, donde x es no ligada. El productor espera por esta pareja y entonces liga x al elemento siguiente. Esto se programa así:

```
proc {DGenerar N Xs}
  case Xs of X|Xr then
    X=N
    {DGenerar N+1 Xr}
  end
end
fun {DSuma ?Xs A Limit}
  if Limit>0 then
    X|Xr=Xs
  in
    {DSuma Xr A+X Limit-1}
  else A end
end
local Xs S in
  thread {DGenerar 0 Xs} end      % Hilo productor
  thread S={DSuma Xs 0 150000} end % Hilo consumidor
  {Browse S}
end
```

Ahora es el consumidor el que controla cuántos elementos se necesitan (150000 es un argumento de `DSuma`, no de `DGenerar`). Esto implementa la ejecución perezosa programándola explícitamente. Este es un ejemplo de un disparador programado

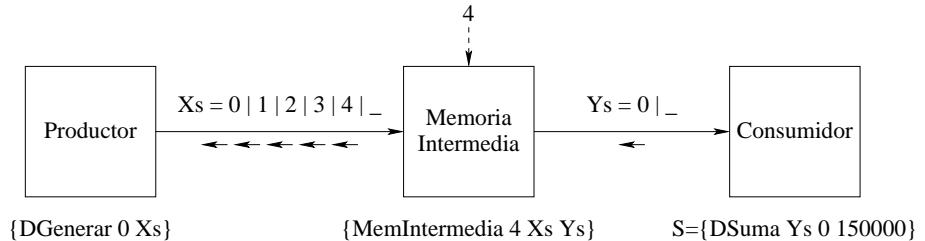


Figura 4.13: Memoria intermedia limitada.

```

proc {MemIntermedia N ?Xs ?Ys}
    fun {Iniciar N ?Xs}
        if N==0 then Xs
        else Xr in Xs=_|Xr {Iniciar N-1 Xr} end
    end

    proc {CicloAtender Ys ?Xs ?Final}
        case Ys of Y|Yr then Xr Final2 in
            Xs=Y|Xr % Envía un elemento de la memoria intermedia
            Final=_|Final2 % Completa la memoria intermedia
            {CicloAtender Yr Xr Final2}
        end
    end

    Final={Iniciar N Xs}
    in
    {CicloAtender Ys Xs Final}
end

```

Figura 4.14: Memoria intermedia limitada (versión concurrente dirigida por los datos).

tal como se definió en la sección 3.6.5.⁸

4.3.3.2. Control del flujo con una memoria intermedia limitada

Hasta ahora hemos visto dos técnicas para administrar la comunicación por flujos, a saber, ejecución ansiosa y perezosa. En la ejecución ansiosa, el productor está completamente libre: no existen restricciones sobre cuán adelantado puede ir

8. Hay otra manera de implementar la ejecución perezosa, a saber, extendiendo el modelo de computación con un concepto nuevo: un disparador implícito. Esto se explica en la sección 4.5. Veremos que los disparadores implícitos son más fáciles de programar que los disparadores programados.

Concurrencia declarativa

sobre los requerimientos del consumidor. En la ejecución perezosa, el productor está completamente restringido: él no puede generar nada sin un requerimiento explícito del consumidor. Ambas técnicas presentan problemas. Hemos visto que la ejecución ansiosa lleva a una explosión en la utilización de recursos. Pero, la ejecución perezosa también tiene un problema serio, pues lleva a una fuerte reducción del rendimiento. Por rendimiento entendemos el número de mensajes que pueden ser enviados por unidad de tiempo. (El rendimiento se contrasta normalmente con la latencia, la cual está definida como el tiempo tomado entre el envío y la llegada de un solo mensaje.) Si el consumidor solicita un mensaje, entonces el productor tiene que calcularlo, y mientras tanto el consumidor espera. Si al productor se le permitiera ir delante del consumidor, entonces éste no tendría que esperar.

¿Existe una manera de lograr lo mejor de los dos mundos, i.e., evitar el problema de recursos sin reducir el rendimiento? Sí; esto es, en efecto, posible. Se puede lograr con una combinación de ejecución ansiosa y perezosa llamada una memoria intermedia limitada. Una memoria intermedia limitada es un transductor que almacena elementos hasta en un número máximo, digamos n . El productor puede ir adelante del consumidor, pero sólo hasta que la memoria intermedia esté llena. Esto limita el uso de recursos extra a n elementos. El consumidor puede tomar elementos de la memoria intermedia, inmediatamente, sin esperar. Esto conserva alto el rendimiento. Cuando la memoria intermedia tiene menos de n elementos, el productor produce más elementos, hasta llenar la memoria intermedia.

En la figura 4.14 se muestra cómo programar la memoria intermedia limitada. En la figura 4.13 se presenta la idea por medio de un dibujo. En el dibujo se introduce un poco más de notación gráfica: las flechas pequeñas en sentido contrario al flujo, las cuales denotan solicitudes de nuevos elementos del flujo (i.e., el flujo es perezoso). Para entender cómo funciona la memoria intermedia, recuerde que tanto `Xs` como `Ys` son flujos perezosos. La memoria intermedia se ejecuta en dos fases:

- La primera fase es la inicialización. Se invoca `Iniciar` para solicitar n elementos al productor. En otras palabras, se extiende `Xs` con n elementos, cada uno no ligado. El productor detecta esto y puede generar esos n elementos.
- La segunda fase es la administración de la memoria intermedia. Se invoca `CicloAtender` para atender las solicitudes del consumidor e iniciar solicitudes al productor. Cuando el consumidor solicita un elemento, `CicloAtender` hace dos cosas: entrega al consumidor un elemento de la memoria intermedia y solicita al productor otro elemento para completar la memoria intermedia.

Un ejemplo sencillo de ejecución es el siguiente:

```
local Xs Ys S in
  thread {DGenerar 0 Xs} end          % Hilo productor
  thread {MemIntermedia 4 Xs Ys} end    % Hilo memoria intermedia
  thread S={DSuma Ys 0 150000} end      % Hilo consumidor
  {Browse Xs} {Browse Ys}
  {Browse S}
end
```

Una forma de ver por uno mismo cómo funciona esto, consiste en desacelerar la ejecución a una escala humana. Esto se puede hacer agregando una invocación `{Delay 1000}` dentro de `DSuma`. De esta forma se puede ver la memoria intermedia: `xs` tendrá siempre cuatro elementos más que `ys`.

El programa de la memoria intermedia limitada es un poco difícil de entender y de escribir. Esto se debe a la cantidad de cosas adicionales que es necesario tener en cuenta para implementar la ejecución perezosa. Estas cosas adicionales existen únicamente por razones técnicas; no tienen ningún efecto sobre cómo se escriben el productor y el consumidor. Este es un buen indicio de que extender el modelo de computación puede ser una buena alternativa para implementar la ejecución perezosa. De hecho, éste es el caso, como lo veremos en la sección 4.5. Será mucho más fácil de programar con la ejecución perezosa implícita introducida allí, que con la ejecución perezosa explícita que usamos aquí.

Hay un defecto en la memoria intermedia limitada que hemos presentado. Se toma $O(n)$ espacio en memoria aún si no hay nada almacenado allí (e.g., cuando el productor es lento). Este espacio extra de memoria es pequeño: consiste de n elementos no ligados en una lista, las cuales son las n solicitudes al productor. Por supuesto, como defensores de la elegancia en la programación, nos preguntamos si es posible evitar este uso de espacio de memoria extra. Una manera sencilla de evitarlo es usando estado explícito, como se define en el capítulo 6. Esto nos permite definir una abstracción de datos que representa una memoria intermedia limitada con dos operaciones, `Put` y `Get`. Internamente, la abstracción de datos puede ahorrar espacio utilizando un entero para contar las solicitudes al productor, en lugar de utilizar elementos en una lista.

Concluimos esta discusión resaltando que tanto la ejecución ansiosa como la perezosa son casos extremos de una memoria intermedia limitada. La ejecución ansiosa se da cuando el tamaño de la memoria intermedia es infinito. La ejecución perezosa, por su lado, se da cuando el tamaño de la memoria intermedia es cero. Cuando el tamaño de la memoria intermedia es un valor finito distinto de cero, entonces el comportamiento está entre esos dos extremos.

4.3.3.3. Control del flujo con prioridades en los hilos

La utilización de una memoria intermedia limitada es la mejor alternativa para implementar el control del flujo, pues funciona para todas las velocidades relativas productor/consumidor sin tener que dar vueltas buscando ningún “número mágico.” Otra manera de hacer el control del flujo, pero menos buena que la memoria intermedia limitada, consiste en cambiar las prioridades relativas entre los hilos productor y consumidor, de manera que los consumidores consuman más rápido que lo que los productores producen. Esta forma es menos buena porque es frágil: su éxito depende de la cantidad de trabajo necesario para que un elemento sea producido t_p y consumido t_c . Se tiene éxito sólo si la razón de velocidad v_c/v_p entre el hilo del consumidor y el hilo del productor es mayor que la razón t_c/t_p . Esta última depende no sólo de las prioridades de los hilos sino también de cuántos otros hilos

Concurrencia declarativa

existen.

Dicho esto, mostraremos cómo implementar esta técnica en todo caso. Asignemos al productor baja prioridad y al consumidor alta prioridad. También asignaremos las razones entre prioridades alta-media y media-baja en 10:1 y 10:1. Usamos las versiones originales, dirigidas por los datos, de Generar y Suma:

```
{Property.put priorities p(high:10 medium:10)}
local Xs S in
    thread
        {Thread.setThisPriority low}
        Xs={Generar 0 150000}
    end
    thread
        {Thread.setThisPriority high}
        S={Suma Xs 0}
    end
    {Browse S}
end
```

Esto funciona en este caso pues el tiempo para consumir un elemento no es 100 veces mayor que el tiempo para producirlo. Pero podría no funcionar para unos productores o consumidores modificados que puedan tomar mayor o menor tiempo. La lección general es que el cambio de las prioridades de los hilos no debería ser usado nunca para lograr que un programa funcione correctamente. El programa debe funcionar correctamente sin importar cuáles son esas prioridades. De esta manera, cambiar las prioridades de los hilos es un asunto de optimización del desempeño, que puede ser usada para mejorar el rendimiento de un programa que ya está funcionando.

4.3.4. Objetos flujo

Ahora demos un paso atrás y reflexionemos sobre lo que se hace realmente en la programación por flujos. Hemos escrito programas concurrentes como redes de hilos que se comunican a través de flujos. Esto introduce un nuevo concepto que llamaremos un objeto flujo: un procedimiento recursivo que se ejecuta en un hilo propio y se comunica con otros objetos flujo a través de flujos de entrada y de salida. El objeto flujo puede mantener un estado interno en los argumentos de su procedimiento, los cuales son acumuladores.

Lo llamamos un objeto porque es una sola entidad que combina las nociones de valor y operación. Esto en contraste con un TAD, en el cual los valores y las operaciones son entidades separadas (ver sección 6.4). Además, cuenta con un estadio interno al cual se accede de forma controlada (por medio de mensajes enviados en los flujos). A lo largo del libro, usaremos el término “objeto” para varias entidades como esta, incluyendo objetos puerto, objetos pasivos, y objetos activos. Estas entidades difieren en la manera como se almacena el estado interno y en la manera como se define el control de acceso a él. El objeto flujo es la primera y la más sencilla de estas entidades.

A continuación se encuentra una manera general de crear objetos flujo:

```
proc {ObjetoFlujo S1 X1 ?T1}
  case S1
    of M|S2 then N X2 T2 in
      {EstadoProximo M X1 N X2}
      T1=N|T2
      {ObjetoFlujo S2 X2 T2}
    [] nil then T1=nil end
  end
declare S0 X0 T0 in
thread
  {ObjetoFlujo S0 X0 T0}
end
```

ObjetoFlujo es una plantilla para crear un objeto flujo. Su comportamiento está definido por EstadoProximo, el cual toma un mensaje de entrada M y un estado X1, y calcula un mensaje de salida N y un estado nuevo X2. Al ejecutar ObjetoFlujo en un hilo nuevo se crea un objeto flujo nuevo con flujo de entrada S0, flujo de salida T0, y estado inicial X0. El objeto flujo lee mensajes del flujo de entrada, realiza cálculos internos, y envía mensajes por el flujo de salida. En general, un objeto puede tener cualquier número fijo de flujos de entrada y de salida.

Los objetos flujo pueden ser enlazados juntos en un grafo, donde cada objeto recibe mensajes de uno o más de los otros objetos y envía mensajes a uno o más de los otros objetos. Por ejemplo, el siguiente es un canal compuesto por tres objetos flujo:

```
declare S0 T0 U0 V0 in
thread {ObjetoFlujo S0 0 T0} end
thread {ObjetoFlujo T0 0 U0} end
thread {ObjetoFlujo U0 0 V0} end
```

El primer objeto recibe de S0 y envía por T0, el cual es recibido por el segundo objeto, y así sucesivamente.

4.3.5. Simulación de la lógica digital

A la programación con un grafo dirigido de objetos flujo se le llama programación sincrónica, puesto que un objeto flujo sólo puede realizar un cálculo después de haber leído un elemento de cada flujo de entrada. Esto implica que todos los objetos flujo en el grafo están sincronizados unos con otros. Es posible que un objeto flujo vaya adelante de sus sucesores en el grafo, pero no puede ir adelante de sus predecesores. (En el capítulo 8 veremos cómo construir objetos activos que se pueden ejecutar, cada uno, completamente independiente de los otros.)

Todos los ejemplos de comunicación por flujos que hemos visto hasta ahora son tipos muy sencillos de grafos, a saber, cadenas lineales. Ahora miraremos un ejemplo donde el grafo no es una cadena lineal. Construiremos un simulador de la lógica digital, i.e., un programa que modele fielmente la ejecución de circuitos electrónicos consistentes de compuertas lógicas interconectadas. Las compuertas se

Concurrencia declarativa

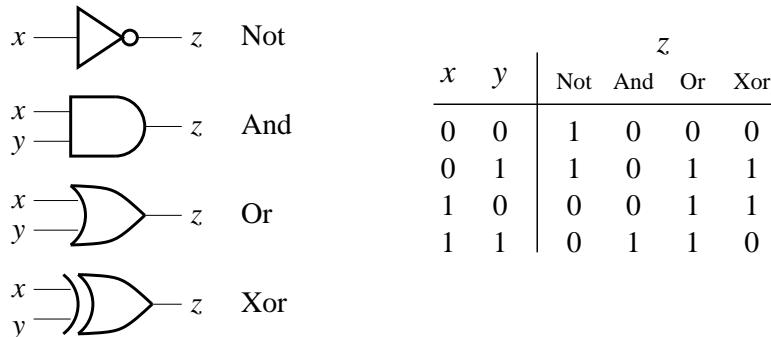


Figura 4.15: Compuertas lógicas digitales.

comunican a través de señales que varían en el tiempo y que sólo pueden tomar valores discretos, como 0 y 1. En la lógica digital sincrónica todo el circuito se ejecuta en etapas seguras. En cada etapa, cada compuerta lógica lee sus cables de entrada, calcula el resultado, y lo coloca en los cables de salida. Las etapas son ejecutadas a una cadencia controlada por un circuito llamado un reloj. La mayor parte de la tecnología electrónica digital actual es sincrónica. Nuestro simulador, también lo será.

¿Cómo modelar señales en un cable y circuitos que leen esas señales? En un circuito sincrónico, las señales varían sólo en etapas discretas de tiempo. Por tanto podemos modelar una señal como un flujo de 0s y 1s. Una compuerta lógica es por tanto un objeto flujo sencillo: un procedimiento recursivo, ejecutándose en un hilo propio, que lee flujos de entrada y calcula flujos de salida. Un reloj es un procedimiento recursivo que produce un flujo inicial a una tasa fija.

4.3.5.1. Lógica combinatoria

Miremos primero, cómo construir compuertas lógicas sencillas. En la figura 4.15 se muestran algunas compuertas típicas con sus símbolos gráficos estándar y las funciones booleanas que ellas definen. La compuerta o-exclusivo se llama normalmente xor. Cada compuerta tiene una o más entradas y una salida. La más sencilla es la compuerta Not, cuya salida es sencillamente la negación de la entrada. Con flujos podemos definirla como sigue:

```
fun {CompuertaNot xs}
  case xs of x|xr then (1-x) | {CompuertaNot xr} end
end
```

Esta compuerta funciona instantáneamente, i.e., el primer elemento del flujo de salida se calcula a partir del primer elemento del flujo de entrada. Esta es una manera razonable de modelar una compuerta real si el período del reloj es mucho más largo que el retraso de la compuerta. Esto nos permite modelar la lógica *combinatoria*, i.e., circuitos lógicos que no tienen memoria interna. Sus salidas son

funciones booleanas de sus entradas, y son totalmente dependientes de las entradas.

¿Cómo conectar varias compuertas juntas? Conectar flujos es fácil: el flujo de salida de una compuerta puede conectarse directamente al flujo de entrada de otra. Como todas las compuertas se pueden ejecutar simultáneamente, cada compuerta necesita ejecutarse dentro de su propio hilo. Esto nos lleva a la definición final de CNot:

```
local
  fun {CicloNot Xs}
    case Xs of X|Xr then (1-X)|{CicloNot Xr} end
  end
in
  fun {CNot Xs}
    thread {CicloNot Xs} end
  end
end
```

Invocar CNot crea una nueva compuerta Not en su hilo propio. Vemos que una compuerta lógica es mucho más que sólo una función booleana; es realmente una entidad concurrente que se comunica con otras entidades concurrentes. Construyamos ahora otros tipos de compuertas. A continuación presentamos una función genérica para construir cualquier tipo de compuerta de dos entradas:

```
fun {ConstruirCompuerta F}
  fun {$ Xs Ys}
    fun {CicloCompuerta Xs Ys}
      case Xs#Ys of (X|Xr)#(Y|Yr) then
        {F X Y}|{CicloCompuerta Xr Yr}
      end
    end
    in
      thread {CicloCompuerta Xs Ys} end
    end
  end
```

Esta función es un buen ejemplo de programación de alto orden: se combina genericidad con instanciación. Con ella podemos construir muchas compuertas:

```
CAnd = {ConstruirCompuerta fun {$ X Y} X*Y end}
COr = {ConstruirCompuerta fun {$ X Y} X+Y-X*Y end}
CNand= {ConstruirCompuerta fun {$ X Y} 1-X*Y end}
CNor = {ConstruirCompuerta fun {$ X Y} 1-X-Y+X*Y end}
CXor = {ConstruirCompuerta fun {$ X Y} X+Y-2*X*Y end}
```

Cada una de estas funciones crea una compuerta cuando se les invoca. Las operaciones lógicas han sido implementadas como operaciones aritméticas sobre los enteros 0 y 1.

Ahora podemos construir circuitos combinados. Un circuito típico es un sumador completo, el cual suma tres números de un bit, dando como resultado un número de dos bits. Los sumadores completos se pueden enlazar unos con otros para construir sumadores de cualquier número de bits. Un sumador completo tiene tres entradas, x , y , z , y dos salidas, c y s , y satisface la ecuación $x + y + z = (cs)_2$. Por ejemplo, si $x = 1$, $y = 1$, y $z = 0$, entonces el resultado es $(10)_2$ en binario, a saber, 2. En

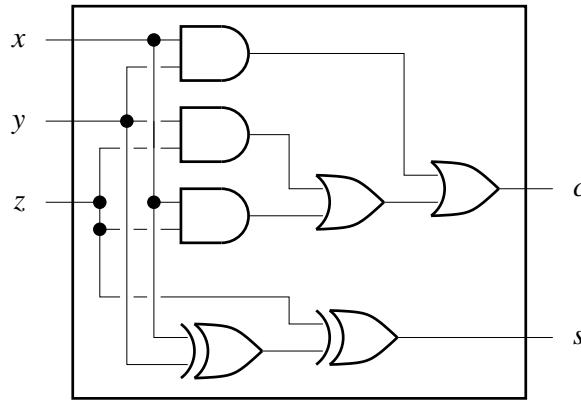


Figura 4.16: Un sumador completo.

la figura 4.16 se define el circuito. Miremos cómo funciona. c es 1 si al menos dos entradas son 1. Hay tres maneras en que esto puede pasar, cada una de las cuales está cubierta por una invocación a `CAnd`. s es 1 si el número de entradas en 1 es impar, lo cual coincide exactamente con la definición del o-exclusivo. A continuación presentamos el mismo circuito, definido en nuestro marco de simulación:

```
proc {SumadorCompleto X Y Z ?C ?S}
  K L M
  in
    K={CAnd X Y}
    L={CAnd Y Z}
    M={CAnd X Z}
    C={Cor K {Cor L M}}
    S={Cxor Z {Cxor X Y}}
  end
```

Utilizamos notación procedimental para `SumadorCompleto` debido a que tiene dos salidas. El siguiente es un ejemplo de utilización del sumador completo:

```
declare
  X=1|1|0|_
  Y=0|1|0|_
  Z=1|1|1|_ C S in
  {SumadorCompleto X Y Z C S}
  {Browse ent(X Y Z)#suma(C S)}
```

Aquí se suman tres conjuntos de bits de entrada.

4.3.5.2. Lógica secuencial

Los circuitos combinados son limitados pues no pueden almacenar información. Seamos un poco más ambiciosos en el tipo de circuitos que deseamos modelar. Modelemos circuitos secuenciales, i.e., circuitos cuyo comportamiento depende de sus

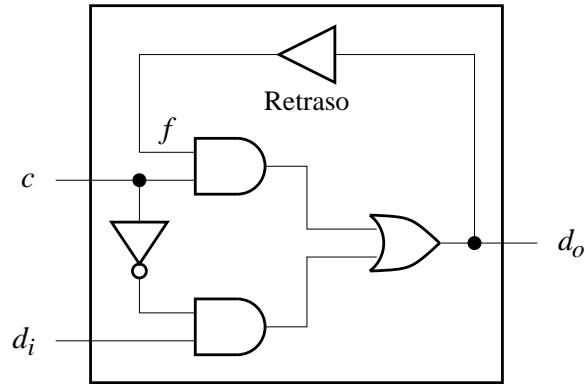


Figura 4.17: Un cerrojo.

salidas anteriores. Sencillamente, esto significa que algunas salidas retroalimentan las mismas entradas. Utilizando esta idea, podemos construir circuitos biestables, i.e., circuitos con dos estados estables. Un circuito biestable es una celda de memoria que puede almacenar un bit de información. Los circuitos biestables son comúnmente llamados flip-flops.

No podemos modelar los circuitos secuenciales con el enfoque de la sección anterior. ¿Qué pasa si lo intentamos? Conectemos una salida a una entrada. Para producir una salida, el circuito tiene que leer una entrada. Pero no hay entrada, luego no se puede producir una salida tampoco. De hecho, esto es una situación de abrazo mortal pues existe una dependencia cíclica: la salida espera por la entrada y la entrada espera por la salida.

Para modelar correctamente los circuitos secuenciales, tenemos que introducir alguna clase de retraso entre las entradas y las salidas. Entonces, el circuito tomará su entrada de la salida anterior. Así, deja de existir el abrazo mortal. Podemos modelar el retraso de tiempo con una compuerta de retraso, la cual simplemente agrega uno o más elementos a la cabeza del flujo:

```
fun {CRetraso xs}
  0 | xs
end
```

Para una entrada $a|b|c|d|\dots$, `CRetraso` devuelve $0|a|b|c|d|\dots$, lo cual es justamente una versión retrasada de la entrada. En la figura 4.17 se define un cerrojo sencillo. El programa es el siguiente:

Concurrencia declarativa

```
fun {Cerrojo C DI}
    DO X Y Z F
    in
        F={CRetraso DO}
        X={CAnd F C}
        Z={CompNot C}
        Y={CAnd Z DI}
        DO={COr X Y}
        DO
    end
```

El cerrojo tiene dos entradas, *C* y *DI*, y una salida, *DO*. Si *C* es 0, entonces la salida le sigue la pista a *DI*, i.e., siempre tiene el mismo valor que *DI*. Si *C* es 1, entonces la salida se congela en el último valor de *DI*. El cerrojo es biestable pues *DO* puede ser 0 o 1. El cerrojo funciona gracias a la retroalimentación retrasada de *DO* a *F*.

4.3.5.3. *El reloj*

Suponga que hemos modelado un circuito complejo. Para simular su ejecución, tenemos que crear un flujo de entrada inicial de valores que son discretizados en el tiempo. Una forma de hacerlo es definiendo un reloj, el cual es una fuente regulada de señales periódicas. Un reloj sencillo es:

```
fun {Reloj}
    fun {Ciclo B}
        B|{Ciclo B}
    end
    in
        thread {Ciclo 1} end
    end
```

Invocar el reloj crea un flujo que crece muy rápidamente, lo cual hace que la implementación vaya a su máxima rata en la implementación Mozart. Podemos desacelerar la simulación a una escala humana visible, agregando un retraso al reloj.

```
fun {Reloj}
    fun {Ciclo B}
        {Delay 1000} B|{Ciclo B}
    end
    in
        thread {Ciclo 1} end
    end
```

La invocación *{Delay N}* hace que el hilo se suspenda para *N* ms para luego volverse un hilo ejecutable de nuevo.

4.3.5.4. *Una abstracción lingüística para compuertas lógicas*

En la mayoría de los ejemplos anteriores, las compuertas lógicas se programan con una construcción que siempre tiene la misma forma. La construcción define un procedimiento con flujos como argumentos y en su corazón existe un procedi-

```

proc {Compuerta X1 X2 ... Xn Y1 Y2 ... Ym}
  proc {P S1 S2 ... Sn U1 U2 ... Um}
    case S1#S2#...#Sn
    of (X1|T1)#{(X2|T2)}...#{(Xn|Tn)} then
      Y1 Y2 ... Ym
      V1 V2 ... Vm
    in
      {EtapaCompuerta X1 X2 ... Xn Y1 Y2 ... Ym}
      U1=Y1|V1
      U2=Y2|V2
      ...
      Um=Ym|Vm
      {P T1 T2 ... Tn V1 V2 ... Vm}
    end
  end
  in
    thread {P X1 X2 ... Xn Y1 Y2 ... Ym} end
end

```

Figura 4.18: Una abstracción lingüística para compuertas lógicas.

miento con argumentos booleanos. En la figura 4.18 se muestra cómo hacer esta construcción sistemáticamente. A partir de un procedimiento `EtapaCompuerta`, se define otro procedimiento `Compuerta`. Los argumentos de `EtapaCompuerta` son booleanas (o enteros) y los argumentos de `Compuerta` son flujos. Distingamos las entradas y salidas de la compuerta. Los argumentos X_1, X_2, \dots, X_n son las entradas de la compuerta. Los argumentos Y_1, Y_2, \dots, Y_m son las salidas de la compuerta. `EtapaCompuerta` define el comportamiento instantáneo de la compuerta, i.e., calcula las salidas booleanas de la compuerta a partir de sus entradas booleanas en un instante dado. `Compuerta` define el comportamiento en términos de flujos. Podemos decir que la construcción eleva un cálculo con booleanos a un cálculo con flujos. Podríamos definir una abstracción que implemente esta construcción. Esto nos lleva a la función `ConstruirCompuerta` que definimos antes. Pero podemos ir más allá y definir una abstracción lingüística, la declaración **gate**:

```
gate input  $\langle x \rangle_1 \dots \langle x \rangle_n$  output  $\langle y \rangle_1 \dots \langle y \rangle_m$  then  $\langle d \rangle$  end
```

Esta declaración encierra adentro la construcción de la figura 4.18. El cuerpo $\langle d \rangle$ corresponde a la definición de `EtapaCompuerta`: se realiza un cálculo booleano con las entradas $\langle x \rangle_1 \dots \langle x \rangle_n$ y las salidas $\langle y \rangle_1 \dots \langle y \rangle_m$. Con la declaración **gate** podemos definir una compuerta And como sigue:

```

proc {CAnd x1 x2 ?x3}
  gate input x1 x2 output x3 then x3=x1*x2 end
end

```

Los identificadores x_1, x_2 , y x_3 se refieren a variables diferentes dentro y fuera de la declaración. Adentro, ellas se refieren a booleanos y afuera ellas se refieren a flujos.

Concurrencia declarativa

```
proc {DepthFirst Árbol Nivel LimIzq ?RaízX ?LimDer}
  case Árbol
  of árbol(x:X y:Y izq:hoja der:hoja ...) then
    X=LimIzq
    RaízX=X
    LimDer=X
    thread Y=Scale*Nivel end
  [] árbol(x:X y:Y izq:I der:hoja ...) then
    X=RaízX
    thread Y=Scale*Nivel end
    {DepthFirst I Nivel+1 LimIzq RaízX LimDer}
  [] árbol(x:X y:Y izq:hoja der:D ...) then
    X=RaízX
    thread Y=Scale*Nivel end
    {DepthFirst D Nivel+1 LimIzq RaízX LimDer}
  [] árbol(x:X y:Y izq:I der:D ...) then
    IRAízX ILimDer DRaízX DLimIzq
    in
      RaízX=X
      thread X=(IRAízX+DRaízX) div 2 end
      thread Y=Scale*Nivel end
      thread DLimIzq=ILimDer+Scale end
      {DepthFirst I Nivel+1 LimIzq IRAízX ILimDer}
      {DepthFirst D Nivel+1 DLimIzq DRaízX LimDer}
    end
  end
```

Figura 4.19: Algoritmo para dibujar árboles con concurrencia para determinación del orden.

Podemos embeber declaraciones **gate** en procedimientos y usarlos para construir circuitos grandes.

Podríamos implementar la declaración **gate** utilizando la herramienta generadora de analizadores sintácticos de Mozart, gump. Muchos lenguajes simbólicos, notablemente Haskell y Prolog, tienen la capacidad de extender su sintaxis, lo cual hace que este tipo de cosas sea fácil de agregar. Esto es muy conveniente para aplicaciones de propósito especial.

4.4. Utilizando el modelo concurrente declarativo directamente

La comunicación por flujos no es la única forma de programar en el modelo concurrente declarativo. En esta sección se exploran otras técnicas. Estas técnicas utilizan el modelo concurrente declarativo directamente, sin sacar ventaja de una abstracción como la de los objetos flujo.

4.4.1. Concurrencia para determinación del orden

En cualquier orden en que se coloquen estas veinticuatro cartas juntas, el resultado será un paisaje perfectamente armonioso.

– Adaptación libre de “*The Endless Landscape*”: 24-piece Myriorama, Leipzig (1830s).

Un uso sencillo de la concurrencia en el modelo declarativo es para encontrar el orden para desarrollar unos cálculos. Es decir, conocemos qué cálculos se deben realizar, pero debido a la dependencia de los datos, no conocemos en qué orden realizarlos. Aún más, el orden puede depender de los valores de los datos, i.e., no existe un orden estático que sea siempre el correcto. En este caso, podemos usar concurrencia por flujo de datos para encontrar el orden automáticamente.

Presentamos un ejemplo de concurrencia para determinación del orden utilizando el algoritmo para dibujar árboles del capítulo 3. Este algoritmo recibe un árbol y calcula las posiciones de todos los nodos del árbol de manera que éste pueda ser dibujado en una forma que sea estéticamente agradable. El algoritmo recorre el árbol en dos direcciones: primero de la raíz hacia las hojas y luego de las hojas hacia la raíz. Durante los dos recorridos, se calculan las posiciones de todos los nodos. Una de las astucias en este algoritmo es el orden en el cual se calculan las posiciones de los nodos. Considere el algoritmo definido en la sección 3.4.7. En esta definición, `Nivel` y `LimIzq` son entradas (se propagan hacia abajo hasta las hojas), `RaízX` y `LimDer` son salidas (se propagan hacia arriba hasta la raíz), y los cálculos deben realizarse en el orden correcto para evitar un abrazo mortal. Existen dos formas de encontrar el orden correcto:

- La primera forma es que el programador deduzca el orden y lo programe de acuerdo a ello. Esto es lo que se hace en la sección 3.4.7. Así se llega a un código más eficiente, pero si el programador comete un error, entonces el programa se bloquea sin dar resultados.
- La segunda forma es que el sistema deduzca el orden dinámicamente. La forma más sencilla de hacerlo consiste en colocar cada cálculo en un hilo diferente. La ejecución por flujo de datos encontrará el orden correcto en tiempo de ejecución.

En la figura 4.19 se presenta una versión del algoritmo para dibujar árboles que utiliza la concurrencia para determinación de orden, para encontrar el orden correcto en tiempo de ejecución. Cada cálculo que puede bloquearse es colocado en un hilo de su propiedad.⁹ El resultado del algoritmo es el mismo de antes. Esto es así porque la concurrencia sólo se usa para cambiar el orden de los cálculos, no para cambiar los cálculos que se realizan. Este es un ejemplo de cómo usar la concurrencia en programación declarativa y permanecer declarativa. En el código anterior, los hilos se crean antes de las invocaciones recursivas. De hecho, los hilos

9. Las operaciones de ligadura no son colocadas en hilos porque ellas nunca se bloquean. ¿Cuál sería la diferencia si cada operación de ligadura se colocara en un hilo propio?

Concurrencia declarativa

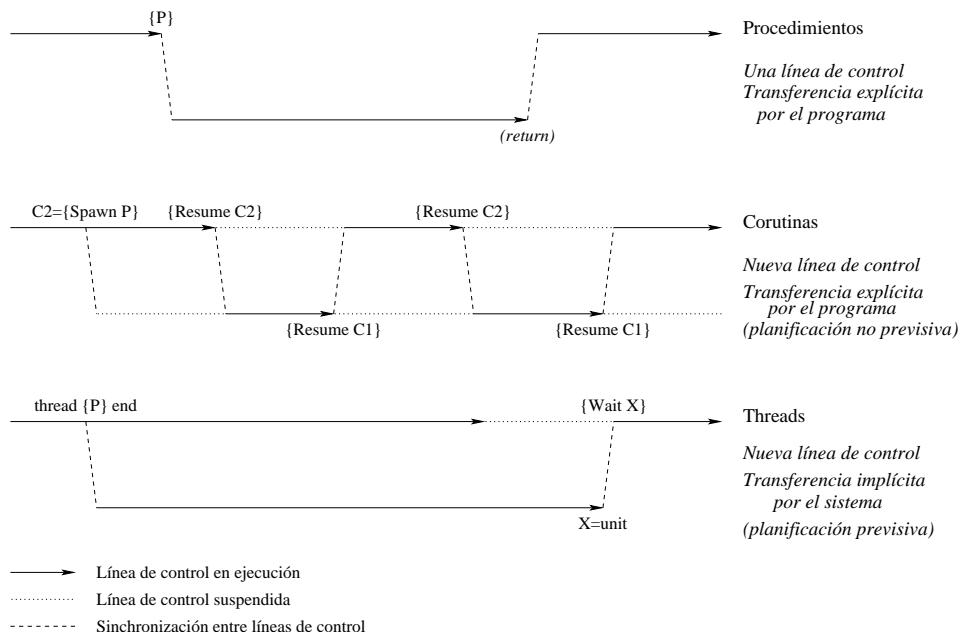


Figura 4.20: Procedimientos, corutinas, e hilos.

se pueden crear en cualquier momento y el algoritmo aún funcionará.

Programación por restricciones

Comparado con el algoritmo secuencial de la sección 3.4.7, el algoritmo de esta sección es más sencillo de diseñar porque desplaza parte de la carga de diseño del programador hacia el sistema. Hay una forma más sencilla aún: utilizar la programación por restricciones. Este enfoque se explica en el capítulo 12 (en CTM).

La programación por restricciones alivia aún más la carga de diseño del programador, con el costo de necesitar algoritmos sofisticados e imparciales para resolver restricciones, los cuales pueden necesitar tiempos de ejecución grandes. La concurrencia para determinación del orden realiza propagación local, en donde las condiciones locales sencillas (e.g., dependencias de flujos de datos) lo determinen. La programación por restricciones va, de manera natural, una etapa más allá: extiende la propagación local realizando, además, búsqueda, en la cual se miran soluciones candidatas hasta que se encuentra una que sea una solución completa.

4.4.2. Corutinas

Una corutina es un hilo que no se puede prevenir. Para explicarlo precisamente, usaremos el término línea de control, el cual se define como una secuencia de

```
fun {Spawn P}
  PID in
    thread
      PID={Thread.this}
      {Thread.suspend PID}
      {P}
    end
    PID
  end

proc {Resume Id}
  {Thread.resume Id}
  {Thread.suspend {Thread.this}}
end
```

Figura 4.21: Implementación de corutinas usando el módulo Thread.

instrucciones en ejecución. En la figura 4.20 se comparan las corutinas con las invocaciones a procedimientos y los hilos. Una invocación a un procedimiento transfiere el control una vez al cuerpo del procedimiento (la invocación) y luego lo recupera (el return). Sólo existe una línea de control en el programa. Una corutina se invoca explícitamente, al igual que un procedimiento, pero cada corutina tiene su propia línea de control, como un hilo. La diferencia con un hilo es que éste último se controla implícitamente: el sistema automáticamente planifica la ejecución de los hilos sin ninguna intervención del programador.

Las corutinas cuentan con dos operaciones, `Spawn` y `Resume`. La función `CId={Spawn P}` crea una nueva corutina y devuelve su identificación `CId`. Esto es similar a la creación de un hilo. La nueva corutina se suspende inicialmente, pero ejecutará el procedimiento de cero argumentos `P` cuando sea reanudada. La operación `{Resume CId}` transfiere el control de la corutina actual a la corutina con identificación `CId`. Cada corutina tiene la responsabilidad de transferir el control suficientemente seguido de manera que las otras tengan oportunidad de ejecutarse. Si esto no se hace correctamente, entonces puede pasar que una corutina nunca tenga la oportunidad de ejecutarse. A esto se le llama inanición y se debe normalmente a un error del programador. (La inanición es imposible con hilos si estos se planifican imparcialmente.)

Como las corutinas no introducen no-determinismo en el modelo, los programas que las usan siguen siendo declarativos. Sin embargo, las corutinas por sí mismas no se pueden implementar en el modelo declarativo pues su implementación requiere usar estado explícito. Las corutinas se pueden implementar usando el modelo concurrente con estado compartido del capítulo 8. En la sección 8.2.2 se explica cómo implementar una versiones sencillas de `Spawn` y `Resume` utilizando este modelo. Otra forma de implementarlas es utilizando el módulo `Thread`.

Implementación usando el módulo Thread

La planificación de hilos se puede controlar de cierta forma que parecen corutinas. Por ejemplo, podemos introducir una operación similar a `Resume` que previene de forma inmediata un hilo, i.e., cambia la ejecución a otro hilo ejecutable (si existe). Esto se hace en Mozart con el procedimiento `Thread.preempt`. También podemos introducir operaciones para controlar si un hilo se puede ejecutar o no. En Mozart, estas operaciones se llaman `Thread.suspend` y `Thread.resume`. Un hilo `T` se puede suspender indefinidamente invocando `{Thread.suspend T}`. Aquí `T` es la identificación del hilo, la cual se obtiene invocando `T={Thread.this}`. El hilo se puede reanudar invocando `{Thread.resume T}`. En la figura 4.21 se muestra cómo implementar `Spawn` y `Resume` en términos de `Thread.this`, `Thread.suspend`, y `Thread.resume`.

4.4.3. Composición concurrente

Hemos visto cómo se bifurcan los hilos utilizando la declaración `thread`. Una pregunta que aparece naturalmente es cómo volver a llevar a un hilo bifurcado al hilo original de control. ¿Es decir, cómo hace el hilo original para esperar hasta que el hilo bifurcado haya terminado su tarea? Este es un caso especial de detección de terminación de múltiples hilos, y hacer que otro hilo espere hasta que ese evento se produzca. El esquema general es bastante fácil cuando se utiliza ejecución por flujo de datos. Suponga que tenemos n declaraciones $\langle \text{dec} \rangle_1, \dots, \langle \text{dec} \rangle_n$. Suponga también que las declaraciones no crean hilos cuando se ejecutan.¹⁰ El código a continuación ejecutará cada declaración en un hilo diferente y esperará hasta que todos ellos hayan terminado:

```
local X1 X2 X3 ... Xn1 Xn in
  thread ⟨dec⟩₁ X1=listo end
  thread ⟨dec⟩₂ X2=X1 end
  thread ⟨dec⟩₃ X3=X2 end
  ...
  thread ⟨dec⟩ₙ Xn=Xn₁ end
  {Wait Xn}
end
```

Esto funciona gracias a la operación de unificación de variables de flujo de datos (ver la sección 2.8.2.1). Cuando el hilo T_i termina, se ligan las variables x_{i-1} y x_i . Se crea una especie de “corto circuito” entre las variables. Cuando todos los hilos hayan terminado, las variables x_1, x_2, \dots, x_n se unificarán (“se mezclarán juntas”) y se ligarán al átomo `listo`. La operación `{wait Xn}` se bloquea hasta que `Xn` sea ligada.

10. El caso general en el que los hilos pueden crear hilos nuevos, y así sucesivamente de forma recursiva, se maneja en la sección 5.6.3.

```

proc {Barrera Ps}
  fun {CicloBarrera Ps L}
    case Ps of P|Pr then M in
      thread {P} M=L end
      {CicloBarrera Pr M}
    [] nil then L
    end
  end
  S={CicloBarrera Ps listo}
  in
    {Wait S}
  end

```

Figura 4.22: Composición concurrente.

Existe una manera diferente de detectar terminación con variables de flujo de datos que no depende de ligar variables con variables, pero utiliza un hilo auxiliar:

```

local X1 X2 X3 ... Xn1 Xn Listo in
  thread <dec>1 X1=listo end
  thread <dec>2 X2=listo end
  thread <dec>3 X3=listo end
  ...
  thread <dec>n Xn=listo end
  thread
    {Wait X1} {Wait X2} {Wait X3} ... {Wait Xn}
    Listo=listo
  end
  {Wait Listo}
end

```

Al contar con estado explícito aparece otro conjunto de alternativas para detectar terminación. Por ejemplo, en la sección 5.6.3 se muestra un algoritmo que funciona aún en el caso en que los hilos creen, ellos mismos, hilos nuevos.

Abstracción de control

En la figura 4.22 se define el combinador `Barrera` que implementa la composición concurrente. Un combinador es simplemente una abstracción de control. El término combinador enfatiza que la operación es composicional, i.e., que los combinadores pueden ser anidados. Esto es cierto para aquellas abstracciones de control basadas en clausuras de alcance léxico, i.e., valores de tipo procedimiento.

`Barrera` recibe una lista de procedimientos de cero argumentos, inicia cada procedimiento en su propio hilo, y termina después que todos los hilos terminan. `Barrera` realiza la detección de terminación utilizando el esquema de unificación de la sección anterior. `Barrera` puede ser la base de una abstracción lingüística, la declaración `conc`:

Concurrencia declarativa

```
conc
  ⟨dec⟩1 [ ] ⟨dec⟩2 [ ] ... [ ] ⟨dec⟩n
end
```

definida como

```
{Barrera
  [proc {$} ⟨dec⟩1 end
   proc {$} ⟨dec⟩2 end
   ...
   proc {$} ⟨dec⟩n end ]}
```

Barrera es más general que la declaración **conc** pues el número de declaraciones a componer concurrentemente no tiene que ser conocido en tiempo de compilación.

4.5. Ejecución perezosa

Todas las cosas surgen sin una palabra hablada,
y crecen sin que nadie pida su producción.

– Tao-te Ching, *Lao-tzu* (6th century B.C.)

“La necesidad es la madre de la invención.”
“Pero, quién es el padre?”
“La pereza!”
– Adaptación libre de un proverbio tradicional.

Hasta ahora, siempre hemos ejecutado las declaraciones, en orden, de izquierda a derecha. En una declaración de secuencia, empezamos por ejecutar la primera declaración. Y, cuando ésta termina, continuamos con la siguiente.¹¹ Este hecho puede parecer demasiado obvio para mencionarlo. ¿Por qué debería ser de otra manera? ¡Pero es un reflejo saludable cuestionarse lo obvio! Muchos descubrimientos significativos los han realizado personas que se cuestionan lo obvio: eso llevó a Newton a descubrir que la luz blanca consiste de un espectro de colores y a Einstein a descubrir que la velocidad de la luz es constante para todos los observadores. Cuestionemos entonces lo obvio y veamos a donde nos lleva.

¿Existen otras estrategias de ejecución para los programas declarativos? Resulta que hay una segunda estrategia, fundamentalmente diferente de la ejecución normal de izquierda a derecha. Llamamos esta estrategia evaluación perezosa o evaluación dirigida por la demanda, en contraste con la estrategia normal, la cual se llama evaluación ansiosa o evaluación dirigida por los datos. En la evaluación perezosa, una declaración sólo se ejecuta cuando su resultado se necesita en alguna otra parte del programa. Por ejemplo, considere el siguiente fragmento de programa:

11. El orden de las declaraciones puede ser determinado estáticamente por la secuencia textual, o dinámicamente por sincronización en el flujo de datos.

```
fun lazy {F1 X} 1+X*(3+X*(3+X)) end
fun lazy {F2 X} Y=X*X in Y*Y end
fun lazy {F3 X} (X+1)*(X+1) end
A={F1 10}
B={F2 20}
C={F3 30}
D=A+B
```

Las tres funciones F1, F2, y F3 son funciones perezosas. Esto se indica con la anotación “*lazy*” en la sintaxis. Las funciones perezosas no se ejecutan en el momento de ser invocadas. Tampoco se bloquean. Lo que pasa es que ellas crean “ejecuciones detenidas” que continuarán sólo cuando se necesiten sus resultados. En nuestro ejemplo, todas las invocaciones A={F1 10}, B={F2 20}, y C={F3 30} crean ejecuciones detenidas. Cuando se invoca la suma D=A+B, entonces se necesitan los valores de A y B. Esto dispara la ejecución de las primeras dos invocaciones. Después que las invocaciones terminan, la suma puede continuar. Como C no se necesitó, la tercera invocación no se ejecutó.

La importancia de la evaluación perezosa

La evaluación perezosa es un concepto poderoso que puede simplificar muchas tareas de programación. Fue descubierta primero en el contexto de la programación funcional, donde tiene una larga y distinguida historia [74]. La evaluación perezosa fue estudiada originalmente como una estrategia de ejecución, útil solamente para programas declarativos. Sin embargo, como lo veremos más adelante, la evaluación perezosa también tiene un papel para jugar en modelos de computación más expresivos que contienen modelos declarativos como subconjuntos.

La evaluación perezosa juega un papel tanto en la programación en grande (para modularización y administración de recursos) como en la programación en pequeño (para diseño de algoritmos). Dentro de la programación en pequeño, la evaluación perezosa puede ayudar en el diseño de algoritmos declarativos que tengan buenos límites de eficiencia en tiempo [129], amortizado o en el peor caso. En la sección 4.5.8 se presentan las principales ideas. Dentro de la programación en grande, la evaluación perezosa puede ayudar a modularizar programas [76]. Por ejemplo, considere una aplicación donde un productor envía un flujo de datos al consumidor. En un modelo ansioso, el productor decide cuándo ha enviado suficientes datos. En el modelo perezoso, es el consumidor quien decide. En las secciones 4.5.3 hasta la 4.5.6 presentamos este ejemplo y otros.

El modelo de computación perezosa del libro es ligeramente diferente de la evaluación perezosa usada en lenguajes funcionales tales como Haskell y Miranda. Como estos lenguajes son secuenciales, la evaluación perezosa utiliza corutinas entre la función perezosa y la función que necesita el resultado. Este libro estudia la evaluación perezosa en un contexto más general de modelos concurrentes. Para evitar confusiones con la evaluación perezosa, la cual siempre es secuencial, usaremos el término ejecución perezosa para cubrir el caso general, el cual puede ser secuencial o concurrente.

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Invocación de procedimiento
thread $\langle d \rangle$ end	Creación de hilo
$\{\text{ByNeed} \langle x \rangle \langle y \rangle\}$	Creación de disparador

Tabla 4.3: El lenguaje núcleo del modelo concurrente dirigido por la demanda.

Estructura de la sección

En esta sección se define el concepto de ejecución perezosa y se presenta un panorama de las nuevas técnicas de programación que lo hacen posible. La estructura de esta sección es la siguiente:

- Las primeras dos secciones presentan los fundamentos de la ejecución perezosa y muestra cómo ésta interactúa con la ejecución ansiosa y la concurrencia. En la sección 4.5.1 se define el modelo concurrente dirigido por la demanda y se presenta su semántica. Este modelo extiende el modelo concurrente dirigido por los datos con el concepto nuevo de pereza. Es un hecho sorprendente que este nuevo modelo sea declarativo. En la sección 4.5.2 mostramos seis diferentes modelos de computación declarativos que son posibles con diferentes combinaciones de pereza, variables de flujo de datos, y concurrencia declarativa. Todos estos modelos son prácticos y algunos de ellos han sido usados como la base de lenguajes de programación funcional.
- En las siguientes cuatro secciones, de la sección 4.5.3 a la 4.5.6, se presentan técnicas de programación utilizando flujos perezosos. Los flujos son el ejemplo más común de uso de la pereza.
- En las tres últimas secciones se presentan usos más avanzados de la pereza. En la sección 4.5.7 se introduce el tema mostrando qué pasa cuando las funciones estándar sobre listas se hacen perezosas. En la sección 4.5.8 se muestra cómo usar la pereza para diseñar estructuras de datos persistentes, con buenas complejidades amortizadas o en el peor de los casos. En la sección 4.5.9 se explican las listas comprehensivas, las cuales representan el más alto nivel de abstracción con el cual mirar los flujos perezosos.

4.5.1. El modelo concurrente dirigido por la demanda

El modelo concurrente dirigido por la demanda extiende el modelo concurrente dirigido por los datos con un solo concepto nuevo, el disparador by-need. Un principio importante en el diseño de este concepto fue que el modelo resultante debía satisfacer la definición de concurrencia declarativa presentada en la sección 4.1.4. En esta sección se define la semántica del disparador by-need y mostramos cómo se pueden expresar funciones perezosas con él.

¿Cómo pueden coexistir la concurrencia dirigida por los datos y la concurrencia dirigida por la demanda en un mismo modelo de computación? Lo que hemos escogido es hacer que la concurrencia dirigida por los datos sea el modelo por defecto, y agregar una operación adicional para introducir la parte dirigida por la demanda. Es razonable que la concurrencia dirigida por los datos sea escogida por defecto pues es mucho más fácil razonar en ella sobre la complejidad en tiempo y espacio, así como implementarla eficientemente. Hemos descubierto que en muchos casos la mejor manera de construir una aplicación es construyéndola en forma dirigida por los datos con un núcleo dirigido por la demanda.

Disparadores by-need

Para hacer concurrencia dirigida por la demanda, agregamos una instrucción, `ByNeed`, al núcleo del lenguaje (ver tabla 4.3). Su operación es extremadamente sencilla. La declaración `{ByNeed P Y}` tiene el mismo efecto que la declaración `thread {P Y} end`, salvo para la planificación. Ambas declaraciones invocan el procedimiento `P` en su propio hilo con argumento `Y`. La diferencia entre las declaraciones es el momento en que la invocación al procedimiento se ejecuta. Para el caso `thread {P Y} end`, sabemos que `{P Y}` se ejecutará finalmente, siempre. Para el caso `{ByNeed P Y}`, sabemos que `{P Y}` se ejecutará sólo si se necesita el valor de `Y`. Si nunca se necesita el valor de `Y`, entonces nunca se ejecutará `{P Y}`. A continuación un ejemplo:

```
{ByNeed proc {$ A} A=111*111 end Y}  
{Browse Y}
```

Esto despliega `Y` sin calcular su valor pues el browser no necesita el valor de `Y`. Al invocar una operación que necesite el valor de `Y`, e.g., `Z=Y+1` o `{Wait Y}`, se disparará el cálculo de `Y`. Esto hace que se despliegue 12321 en el browser.

Semántica de los disparadores by-need

Implementamos `ByNeed`, en el modelo de computación, agregando simplemente un concepto, el disparador by-need. En general, un disparador es una pareja que contiene una condición de activación, la cual es una expresión booleana, y una acción, la cual es un procedimiento. Cuando la condición de activación se cumple, entonces se ejecuta la acción una vez. Cuando esto pasa, decimos que el disparador se activó. Para un disparador by-need la condición de activación es que se necesite

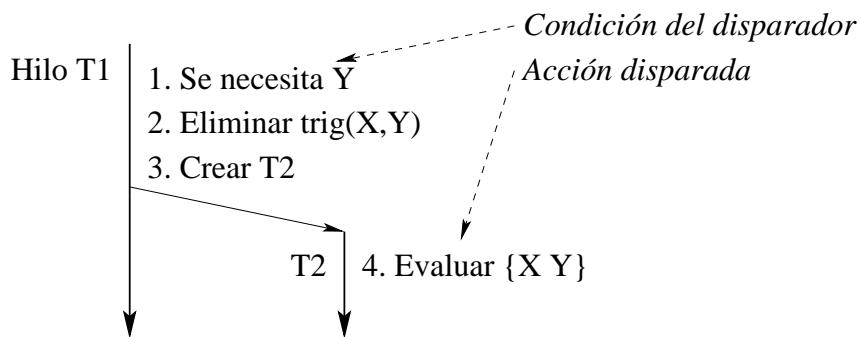


Figura 4.23: El protocolo by-need.

el valor de una variable.

Distinguimos entre disparadores programados, los cuales son escritos explícitamente por el programador, y disparadores implícitos, los cuales hacen parte del modelo de computación. Los disparadores programados se explican en las secciones 3.6.5 y 4.3.3. Un disparador by-need es un tipo de disparador implícito.

Definimos la semántica de los disparadores by-need en tres etapas. Primero agregamos un almacén de disparadores al estado de la ejecución. Luego definimos dos operaciones, creación y activación de un disparador. Finalmente, precisamos lo que significa “necesitar” una variable.

Extensión del estado de la ejecución Un disparador by-need es una pareja $trig(x, y)$ donde y es una variable de flujo de datos, y x es un procedimiento de un argumento. Además del almacén de asignación única σ , añadimos un almacén nuevo τ llamado el almacén de disparadores. El almacén de disparadores contiene todos los disparadores by-need e inicialmente se encuentra vacío. El estado de la ejecución se vuelve una tripleta (MST, σ, τ) .

Creación de disparadores La declaración semántica es

$(\{\text{ByNeed } \langle x \rangle \langle y \rangle\}, E)$

La ejecución consiste de las acciones siguientes:

- Si $E(\langle y \rangle)$ no está determinada, entonces agregue el disparador $trig(E(\langle x \rangle), E(\langle y \rangle))$ al almacén de disparadores.
- Sino, si $E(\langle y \rangle)$ está determinada, entonces cree un hilo nuevo con la declaración semántica inicial $(\{\langle x \rangle \langle y \rangle\}, E)$ (ver sección 4.1 para la semántica de hilos).

Activación de disparadores Si el almacén de disparadores contiene $trig(x, y)$ y se detecta que se necesita y i.e., hay un hilo suspendido esperando que y se determine, o hay un intento de ligar y para determinarla, entonces haga lo siguiente:

- Elimine el disparador del almacén de disparadores.
- Cree un hilo nuevo con la declaración semántica inicial ($\{\langle x \rangle \langle y \rangle\}, \{\langle x \rangle \rightarrow x, \langle y \rangle \rightarrow y\}$) (ver sección 4.1). En esta declaración semántica, $\langle x \rangle$ y $\langle y \rangle$ son dos identificadores diferentes cualesquiera.

Estas acciones se pueden realizar en cualquier momento en el tiempo después que se detecta la necesidad, pues ésta no desaparecerá. La semántica de activación del disparador se llama protocolo by-need, y se ilustra en la figura 4.23.

Administración de la memoria Hay dos modificaciones en la administración de la memoria:

- Se extiende la definición de alcanzabilidad: Una variable x es alcanzable si el almacén de disparadores contiene $trig(x, y)$ y y es alcanzable.
- Recuperación de disparadores: si una variable y se vuelve inalcanzable y el almacén de disparadores contiene $trig(x, y)$, entonces elimine el disparador.

Necesidad de una variable

¿Qué significa que una variable sea necesitada? La definición de necesidad está cuidadosamente diseñada, de manera que la ejecución perezosa sea declarativa, i.e., todas las ejecuciones lleven a almacenes lógicamente equivalentes. Una variable es necesitada por una operación suspendida si la variable debe estar determinada para que la operación continúe. Un ejemplo es el siguiente:

```
thread X={ByNeed fun {$} 3 end} end
thread Y={ByNeed fun {$} 4 end} end
thread Z=X+Y end
```

Para conservar el ejemplo sencillo, supongamos que cada hilo se ejecuta atómicamente. Esto significa que hay seis posibles ejecuciones. Para que la ejecución perezosa sea declarativa, todas estas ejecuciones deben llevar a almacenes lógicamente equivalentes. ¿Es esto cierto? Sí, es cierto, pues la suma esperará hasta que los otros dos disparadores se creen y estos disparadores serán activados luego.

Hay una segunda manera en que se puede necesitar una variable. Una variable se necesita cuando ella es determinada. Si no fuera así, entonces el modelo concurrente dirigido por la demanda no sería declarativo. Miremos un ejemplo:

```
thread X={ByNeed fun {$} 3 end} end
thread X=2 end
thread Z=X+4 end
```

El comportamiento correcto de este programa es que todas sus ejecuciones deben fallar. Si $X=2$ se ejecuta de último, entonces el disparador ya ha sido activado, ligando X con 3, y la falla es clara. Pero si $X=2$ se ejecuta primero, entonces el disparador también necesita ser activado para que se produzca la falla.

Concluimos presentando un ejemplo más sutil:

Concurrencia declarativa

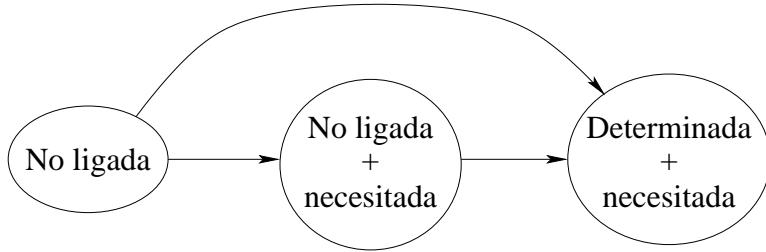


Figura 4.24: Etapas en la vida de una variable en el tiempo.

```

thread X={ByNeed fun {$} 3 end} end
thread X=Y end
thread if X==Y then Z=10 end end
  
```

¿La comparación $X==Y$ debería activar el disparador sobre X ? De acuerdo a nuestra definición la respuesta es no. Si X se determina entonces la comparación aún no se ejecutará (pues Y es no-ligada). Sólo será más tarde, si Y se determina, que se debe activar el disparador sobre X .

Ser necesitada es una propiedad monótona de una variable. Una vez una variable es necesitada, permanece necesitada para siempre. En la figura 4.24 se muestran las etapas en la vida de una variable en el tiempo. Note que una variable determinada siempre es necesitada, sólo por el hecho de estar determinada. La monotonicidad de la propiedad de necesidad es esencial para probar que el modelo concurrente dirigido por la demanda es declarativo.

Utilizando disparadores by-need

Los disparadores by-need se pueden usar para implementar otros conceptos que tienen algo de comportamiento “perezoso” o “dirigido por la demanda”. Por ejemplo, ellos subyacen en las funciones perezosas y en el enlace dinámico. Examinemos cada uno de estos.

Implementando funciones perezosas con disparadores by-need Una función perezosa se evalúa solamente cuando su resultado se necesita. Por ejemplo, la siguiente función genera una lista perezosa de enteros:

```

fun lazy {Generate N} N|{Generate N+1} end
  
```

Esta es una abstracción lingüística definida en términos de `ByNeed`. Se invoca como una función normal:

```

L={Generate 0}
{Browse L}
  
```

Esto no desplegará nada hasta que L sea necesitada. Averigüemos el tercer elemento de L :

{Browse L.2.2.1}

Esto calculará el tercer elemento, 2, y luego lo desplegará. La abstracción lingüística se traduce en el código siguiente usando ByNeed:

```
fun {Generate N}
    {ByNeed fun {$} N|{Generate N+1} end}
end
```

Aquí se utiliza la abstracción procedimental para retrasar la ejecución del cuerpo de la función. El cuerpo ha sido empaquetado en una función de cero argumentos que sólo es invocada cuando el valor de {Generate N} se necesite. Es fácil observar que esto funciona para todas las funciones perezosas. Los hilos son suficientemente económicos en Mozart de manera que esta definición de ejecución perezosa es práctica.

Implementando enlace dinámico con disparadores by-need Brevemente explicamos de qué se trata el enlace dinámico y el papel que juega en la ejecución perezosa. El enlace dinámico es utilizado para implementar un enfoque general para estructurar aplicaciones denominado programación basada en componentes. Este enfoque fue introducido en la sección 3.9 y se explica completamente en los capítulos 5 y 6. Brevemente, el código fuente de una aplicación consiste de un conjunto de especificaciones de componentes, llamadas functors. Una aplicación en ejecución consiste de un conjunto de componentes instanciados denominados módulos. Un módulo está representado por un registro que agrupa en un mismo sitio las operaciones del módulo. Cada campo del registro referencia una operación. Los componentes se enlazan en el momento en que se necesitan, i.e., sus functors se cargan en memoria y se instancian. Mientras el módulo no sea necesitado, el componente no es enlazado. Cuando un programa intenta acceder el campo de un módulo, entonces el componente es necesitado y se usa la ejecución by-need para enlazarlo.

4.5.2. Modelos de computación declarativa

En este punto, hemos definido un modelo de computación que cuenta tanto con pereza como con concurrencia. Es importante darse cuenta que estos dos conceptos son independientes. Con la concurrencia se pueden hacer computaciones en lote de manera incremental. Con la pereza se puede reducir la cantidad de cálculos que se necesitan para conseguir un resultado. Un lenguaje puede tener ambos, alguno, o ninguno de estos dos conceptos. Por ejemplo, un lenguaje con pereza pero sin concurrencia debe utilizar corutinas entre un productor y un consumidor.

Demos ahora una mirada general a todos los modelos de computación declarativos que conocemos. En total, hemos agregado tres conceptos, que conservan la declaratividad mientras incrementan la expresividad, a la programación funcional estricta: variables de flujo de datos, concurrencia declarativa, y pereza. Agregar estos conceptos en diversas combinaciones nos lleva a seis modelos de computación

	<i>secuencial con valores</i>	<i>secuencial con valores y vars. de flujo de datos</i>	<i>concurrente con valores y vars. de flujo de datos</i>
<i>ejecución ansiosa (estricta)</i>	programación funcional estricta (e.g., Scheme, ML) (1)&(2)&(3)	modelo declarativo (e.g., Capítulo 2, Prolog) (1), (2)&(3)	modelo concurrente dirigido por los datos (e.g., Sección 4.1) (1), (2)&(3)
<i>ejecución perezosa</i>	programación funcional perezosa (e.g., Haskell) (1)&(2), (3)	PF perezosa con vars. de flujo de datos (1), (2), (3)	modelo concurrente dirigido por demanda (e.g., Sección 4.5.1) (1), (2), (3)

(1): Declarar una variable en el almacén

(2): Especificar la función para calcular el valor de la variable

(3): Evaluar la función y ligarla a la variable

(1)&(2)&(3): Declaración, especificación, y evaluación coinciden

(1)&(2), (3): Declaración y especificación coinciden; evaluación se realiza después

(1), (2)&(3): Declaración se hace primero; especificación y evaluación se hacen después y coinciden

(1), (2), (3): Declaración, especificación, y evaluación se realizan por separado

Figura 4.25: Modelos de computación declarativos y prácticos.

prácticos, diferentes, como se esquematiza en la figura 4.25.¹² Las variables de flujo de datos son un prerequisito de la concurrencia declarativa, pues son el mecanismo por el cual los hilos se sincronizan y se comunican. Sin embargo, un lenguaje secuencial, como el modelo del capítulo 2, también puede contar con variables de flujo de datos y sacar ventaja de ellas.

Puesto que la pereza y las variables de flujo de datos son conceptos independientes, existen tres momentos especiales en la vida de una variable:

1. La creación de una variable como una entidad del lenguaje, de manera que pueda ser colocada dentro de las estructuras de datos y pasada a, o desde, una función o un procedimiento. La variable no se liga aún a su valor. Tal variable se denomina una “variable de flujo de datos”.
2. La especificación de la invocación a la función o al procedimiento que evaluará el valor de la variable (pero la evaluación no se realiza aún).
3. La evaluación de la función. Cuando el resultado esté disponible, se liga a la variable. La evaluación podría realizarse de acuerdo a un disparador, el cual puede ser implícito, tal como una “necesidad” por el valor. La ejecución perezosa utiliza

12. Este diagrama deja por fuera la búsqueda, la cual lleva a otro tipo de programación declarativa denominada programación relacional. Ésta se explica en el capítulo 9 (en CTM).

la necesidad implícita.

Estos tres momentos pueden ocurrir por separado o al mismo tiempo. Diferentes lenguajes implementan diferentes posibilidades. Esto nos lleva a cuatro variantes del modelo en total. En la figura 4.25 se listan estos modelos así como dos modelos adicionales que resultan de añadir la concurrencia. (En esta figura y en la discusión siguiente, usaremos el símbolo “&” para agrupar los momentos que coinciden.) Por cada una de las variantes, mostramos un ejemplo con una variable *x* que será finalmente ligada al resultado de la computación $11*11$. Estos son los modelos:

- En un lenguaje funcional estricto con valores, tal como Scheme o Standard ML, los momentos (1) & (2) & (3) deben coincidir siempre. Este es el modelo de la sección 2.8.1. Por ejemplo:

```
declare X=11*11 % (1)&(2)&(3) al tiempo
```

- En un lenguaje funcional perezoso con valores, como Haskell, los momentos (1) & (2) siempre coinciden, pero el momento (3) puede hacerse separadamente. Por ejemplo (definiendo primero una función perezosa):

```
declare fun lazy {MultPerezosa A B} A*B end
declare X={MultPerezosa 11 11} % (1)&(2) al tiempo
{Wait X} % (3) por separado
```

Lo que también se puede escribir como

```
declare X={fun lazy {$} 11*11 end} % (1)&(2) al tiempo
{Wait X} % (3) por separado
```

- En un lenguaje estricto con variables de flujo de datos, el momento (1) puede estar separado y los momentos (2) & (3) siempre coinciden. Este es el modelo declarativo, el cual se definió en el capítulo 2. Así también se usa en los lenguajes de programación lógicos como Prolog. Por ejemplo:

```
declare X % (1) por separado
X=11*11 % (2)&(3) al tiempo
```

Si se agrega la concurrencia, llegamos al modelo concurrente dirigido por los datos definido al principio de este capítulo. Esto se usa en los lenguajes de programación lógicos concurrentes. Por ejemplo:

```
declare X % (1) por separado
thread X=11*11 end % (2)&(3) al tiempo
thread if X>100 then {Browse grande} end end % Condicional
```

Como las variables de flujo de datos son de asignación única, el condicional siempre da el mismo resultado.

- En el modelo concurrente dirigido por la demanda de este capítulo, los momentos (1), (2), (3) pueden estar, todos, separados. Por ejemplo:

```
declare X % (1) por separado
X={fun lazy {$} 11*11 end} % (2) por separado
{Wait X} % (3) por separado
```

Cuando la concurrencia se utiliza explícitamente, nos lleva a:

Concurrencia declarativa

```
declare x % (1)
thread x={fun lazy {$} 11*11 end} end % (2)
thread {Wait x} end % (3)
```

Esta es la variante más general del modelo. La única conexión entre los tres momentos es que actúan sobre la misma variable. La ejecución de (2) y (3) es concurrente, con una sincronización implícita entre (2) y (3): (3) espera hasta que (2) haya definido la función.

En todos estos ejemplos, *x* será finalmente ligada a 121. Permitir que los tres momentos se den por separado proporciona máxima expresividad en un marco declarativo.¹³ Por ejemplo, la pereza permite realizar cálculos declarativos con listas potencialmente infinitas. La pereza también permite la implementación de muchas estructuras de datos tan eficientemente como con estado explícito, permaneciendo declarativo (ver, e.g., [129]). Las variables de flujo de datos permiten la escritura de programas concurrentes que sean declarativos. La utilización de ambas permite la escritura de programas concurrentes compuestos de objetos flujo comunicándose a través de flujos potencialmente infinitos.

Por qué la pereza con las variables de flujo de datos debe ser concurrente

En un lenguaje funcional sin variables de flujo de datos, la pereza puede ser secuencial. En otras palabras, los argumentos dirigidos por la demanda de una función perezosa se pueden evaluar secuencialmente (i.e., utilizando corutinas). Si agregamos variables de flujo de datos, ya no se puede hacer lo mismo, pues puede ocurrir un error de abrazo mortal si los argumentos se evalúan secuencialmente. Para resolver el problema, los argumentos tienen que ser evaluados concurrentemente. Miremos el ejemplo siguiente:

```
local
  z
  fun lazy {F1 x}      x+z end
  fun lazy {F2 y}  z=1 y+z end
in
  {Browse {F1 1}+{F2 2}}
end
```

Aquí se definen *F1* y *F2* como funciones perezosas. Al ejecutar este fragmento se despliega 5 en el browser (*¿si ve por qué?*). Si *{F1 1}* y *{F2 2}* se ejecutaran secuencialmente y no concurrentemente, entonces este fragmento caería en un abrazo mortal, pues *x+z* se bloquearía y *z=1* no se alcanzaría nunca. Una pregunta para el lector astuto: ¿cuál de los modelos en la figura 4.25 tiene este problema? La ligadura de *z* realizada por *F2* es una especie de “efecto lateral declarativo,”

13. Una forma de entender la expresividad ganada es pensar que las variables de flujo de datos y la pereza agregan, cada una, una forma débil de estado en el modelo. En ambos casos, las restricciones sobre la utilización del estado aseguran que el modelo se conserve declarativo.

pues F2 cambia sus alrededores a través de un medio separado de sus argumentos. Normalmente, los efectos laterales declarativos son benignos.

Es importante recordar que un lenguaje con variables de flujo de datos y pereza concurrente sigue siendo declarativo. El no-determinismo existente es no observable. $\{F1\ 1\} + \{F2\ 2\}$ siempre produce el mismo resultado.

4.5.3. Flujos perezosos

En el ejemplo de productor/consumidor de la sección 4.3.1, es el productor quien define cuántos elementos de la lista generar, i.e., la ejecución es ansiosa. Esta técnica es razonable si la cantidad total de trabajo es finita y no utiliza muchos recursos del sistema (e.g., memoria o tiempo de procesador). Por otro lado, si el trabajo total utiliza muchos recursos potencialmente, entonces puede ser mejor utilizar ejecución perezosa. Con la ejecución perezosa, el consumidor decide cuántos elementos generar de la lista. Si se necesita un número extremadamente grande o ilimitado de elementos de la lista, entonces la ejecución perezosa utilizará muchos menos recursos del sistema en cualquier momento del tiempo. Los problemas que son poco prácticos con ejecución ansiosa pueden volverse prácticos con la ejecución perezosa. Por otro lado, la ejecución perezosa puede utilizar muchos más recursos en total, debido al costo de su implementación. La necesidad de la pereza debe tomar ambos factores en cuenta.

La ejecución perezosa puede ser implementada de dos maneras en el modelo concurrente declarativo: con disparadores programados o con disparadores implícitos. En la sección 4.3.3 se presenta un ejemplo con disparadores programados. Estos requieren comunicación explícita entre el consumidor y el productor. Los disparadores implícitos, en los que el lenguaje soporta la pereza directamente, son más sencillos. La semántica del lenguaje asegura que una función se evalúa sólo si se necesita su resultado. Esto simplifica la definición de las funciones pues no tienen que hacer todas las cosas adicionales asociadas a los mensajes de los disparadores. En el modelo concurrente dirigido por la demanda presentamos soporte sintáctico para esta técnica: la función puede ser anotada con la palabra clave “`lazy`”. Presentamos cómo realizar el ejemplo anterior con una función perezosa que genere una lista potencialmente infinita:

```
fun lazy {Generar N}
    N | {Generar N+1}
end
fun {Suma Xs A Límite}
    if Límite>0 then
        case Xs of X|Xr then
            {Suma Xr A+X Límite-1}
        end
    else A end
end
```

Concurrencia declarativa

```
local Xs S in
  Xs={Generar 0}          % Productor
  S={Suma Xs 0 150000}   % Consumidor
  {Browse S}
end
```

Al igual que antes, esto despliega 11249925000 en el browser. Note que la invocación a `Generar` no necesita colocarse en un hilo propio, en contraste con la versión ansiosa. Esto se debe a que `Generar` crea un disparador by-need y luego se completa.

En este ejemplo es el consumidor quien decide cuántos elementos de la lista se deben generar. Con la ejecución ansiosa era el productor quien tomaba esa decisión. Dentro del consumidor, la declaración `case` es la que necesita una lista con un elemento por lo menos, lo que dispara implícitamente la generación de un elemento nuevo, `x`, de la lista. Para ver la diferencia en consumo de recursos entre esta versión y la precedente, ensaye ambas con 150000 y luego con 15000000 elementos. Con 150000 elementos no hay problemas de memoria (aun en un computador personal pequeño con 64MB de memoria) y la versión ansiosa es más rápida. Esto se debe al sobrecosto del mecanismo de disparadores implícitos de la versión perezosa. Con 15000000 elementos la situación cambia. La versión perezosa necesita sólo un pequeño espacio de memoria durante la ejecución, mientras que la versión ansiosa necesita una enorme cantidad de memoria. La ejecución perezosa se implementó con la operación `ByNeed` (ver sección 4.5.1).

Declaración de funciones perezosas

En los lenguajes funcionales perezosos, *todas* las funciones son perezosas por defecto. En contraste con esto, el modelo concurrente dirigido por la demanda requiere que la condición de pereza sea declarada explícitamente, con la anotación `lazy`. Creemos que esto facilita las cosas al programador y al compilador, de diversas maneras. La primera manera tiene que ver con eficiencia y compilación. La evaluación ansiosa es muchas veces más eficiente que la evaluación perezosa pues no debe administrar el mecanismo de los disparadores. Para lograr buenos desempeños en programación funcional perezosa, el compilador tiene que determinar qué funciones se pueden implementar de manera segura con evaluación ansiosa. A esta tarea se le denomina análisis de rigidez. La segunda manera tiene que ver con diseño de lenguajes. Es más sencillo extender un lenguaje ansioso con conceptos no declarativos, e.g., excepciones y estado, que hacerlo con un lenguaje perezoso.

Múltiples lectores

El ejemplo de múltiples lectores de la sección 4.3.1 funcionará también con ejecución perezosa. Por ejemplo, suponga que tiene tres consumidores perezosos utilizando las funciones `Generar` y `Suma` definidas en la sección anterior:

```

fun {MemIntermedial Ent N}
    Final={List.drop Ent N}
    fun lazy {Ciclo Ent Final}
        case Ent of E|Ent2 then
            E|{Ciclo Ent2 Final.2}
        end
    end
    in
        {Ciclo Ent Final}
    end

```

Figura 4.26: Memoria intermedia limitada (versión perezosa ingenua).

```

local Xs S1 S2 S3 in
    Xs={Generar 0}
    thread S1={Suma Xs 0 150000} end
    thread S2={Suma Xs 0 100000} end
    thread S3={Suma Xs 0 50000} end
end

```

Cada hilo consumidor solicita elementos del flujo independientemente de los otros. Si uno de los consumidores es más rápido que los otros, entonces los otros no tendrán que solicitar elementos del flujo, pues estos ya habrán sido calculados.

4.5.4. Memoria intermedia limitada

En una sección anterior construimos una memoria intermedia limitada para flujos ansiosos, programando explícitamente la pereza. Construyamos ahora una memoria intermedia limitada utilizando la pereza del modelo de computación. Nuestra memoria intermedia limitada recibirá un flujo perezoso de entrada y devolverá un flujo perezoso de salida.

Definir una memoria intermedia limitada es un buen ejercicio de programación perezosa pues muestra cómo interactúan la ejecución perezosa y la concurrencia dirigida por la demanda. Realizaremos el diseño en etapas. Primero especificaremos su comportamiento. Cuando la memoria sea invocada por primera vez, solicitará n elementos al productor para llenarse a sí misma. Después de esto, cada vez que el consumidor solicite un elemento, la memoria intermedia solicitará a su vez otro elemento al productor. Así, la memoria intermedia siempre contendrá, a lo sumo, n elementos. En la figura 4.26 se muestra la definición resultante. La invocación `{List.drop Ent N}` se salta los primeros N elementos del flujo `Ent`, devolviendo el flujo `Final` como salida. Esto significa que `Final` siempre “mira” N elementos “adelante” con respecto a `Ent`. La función perezosa `Ciclo` se itera cada vez que se necesita un elemento, y devuelve el elemento siguiente `E` pero también solicita al productor un nuevo elemento, invocando `Final.2`. De esta forma, la memoria intermedia siempre contiene, a lo sumo, N elementos.

Sin embargo, la memoria intermedia de la figura 4.26 es incorrecta. El principal

Concurrencia declarativa

```
\begin{ozdisplay}
fun {MemIntermedia2 Ent N}
    Final=thread {List.drop Ent N} end
    fun lazy {Ciclo Ent Final}
        case Ent of E|Ent2 then
            E|{Ciclo Ent2 thread Final.2 end}
        end
    end
in
    {Ciclo Ent Final}
end
```

Figura 4.27: Memoria intermedia limitada (versión perezosa correcta).

problema se debe a la forma en que funciona la ejecución perezosa: el cálculo que necesita el resultado se bloqueará mientras el resultado se calcula. Esto quiere decir, que cuando la memoria intermedia es invocada por primera vez, ella no puede atender ningún requerimiento del consumidor hasta que el productor genere los primeros n elementos. Además, cuando la memoria intermedia esté atendiendo una solicitud del consumidor, no puede enviar la respuesta hasta que el productor haya generado el siguiente elemento para volver a llenarla. Esto es demasiada sincronización: jse enlazan el productor y el consumidor y marchan al mismo tiempo, al mismo paso! Una memoria intermedia utilizable debería, por el contrario, desacoplar el productor y el consumidor. Las solicitudes del conusmidor deberían ser atendidas siempre que la memoria intermedia no esté vacía, independientemente del productor.

No es difícil arreglar este problema. En la definición de `MemIntermedia1`, existen dos sitios donde se generan solicitudes al productor: en la invocación a `List.drop` y en la operación `Final.2`. Colocar un `thread ... end` en ambos sitios resuelve el problema. En la figura 4.27 se muestra la solución.

Ejemplo de ejecución

Miremos cómo funciona esta memoria intermedia. Definimos un productor que genera una lista infinita de enteros consecutivos, pero solamente uno por segundo:

```
fun lazy {Ents N}
    {Delay 1000}
    N|{Ents N+1}
end
```

Ahora creamos esta lista y agregamos una memoria intermedia de cinco elementos:

```
declare
Ent={Ents 1}
Sal={MemIntermedia2 Ent 5}
{Browse Sal}
{Browse Sal.1}
```

La invocación `Sal.1` solicita un elemento. Calcular este elemento toma un segundo. Por lo tanto, el browser desplegará primero `Sal<Future>` y un segundo más tarde añadirá el primer elemento, lo cual actualiza lo que se despliega a `1|_<Future>`. La notación “`_<Future>`” denota una variable de sólo lectura. En el caso de la ejecución perezosa, esta variable tiene un disparador implícito ligado a ella. Ahora esperemos al menos cinco segundos para dejar que la memoria intermedia se llene. Luego entre

```
{Browse Sal.2.2.2.2.2.2.2.2.2}
```

Esto solicita diez elementos. Como la memoria intermedia tiene solamente cinco elementos, ésta se vacía inmediatamente, desplegándose en el browser

```
1|2|3|4|5|6|_<Future>
```

Cada segundo, se añade un elemento más, hasta pasar cuatro segundos. El resultado final es:

```
1|2|3|4|5|6|7|8|9|10|_<Future>
```

En este momento, todas las solicitudes del consumidor se han satisfecho y la memoria intermedia comenzará a llenarse de nuevo a la tasa de un elemento por segundo.

4.5.5. Lectura perezosa de un archivo

La manera más sencilla de leer un archivo es como una lista de caracteres. Sin embargo, si el archivo es muy largo, esto utiliza una cantidad enorme de memoria. Por esto, los archivos se leen, normalmente, por bloques, de manera incremental, un bloque a la vez (donde un bloque es un pedazo contiguo del archivo). El programa tiene el cuidado de guardar en la memoria solamente los bloques que se necesitan. Esto es eficiente en memoria, pero incómodo de programar.

¿Podemos tener lo mejor de los dos mundos: leer el archivo como una lista de caracteres (la cual hace que los programas se conserven sencillos), y leer sólamente las partes que necesitamos (lo cual ahorra memoria)? Con ejecución perezosa la respuesta es sí. Esta es la función `LeerListaPerezosa` que resuelve el problema:

```
fun {LeerListaPerezosa FN}
  {File.readOpen FN}
  fun lazy {LeerSiguiente}
    L T I in
      {File.readBlock I L T}
      if I==0 then T=nil {File.readClose} else T={LeerSiguiente}
    end
    L
  end
  in
    {LeerSiguiente}
  end
```

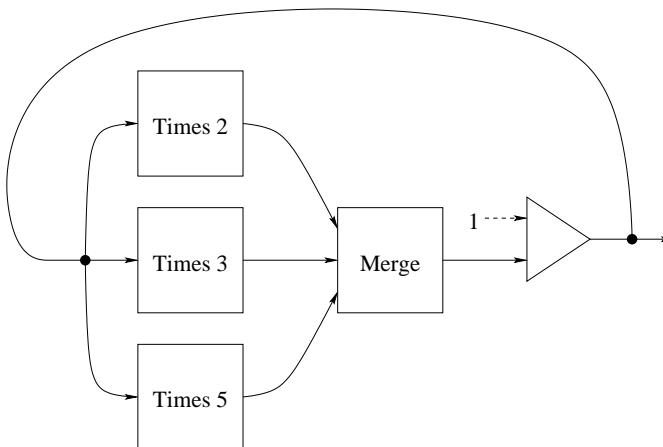


Figura 4.28: Lazy solution to the Hamming problem.

Usamos tres operaciones del módulo `File` (el cual está disponible en el sitio Web del libro): `{File.readOpen FN}`, la cual abre el archivo `FN` para lectura; `{File.readBlock I L T}`, la cual lee un bloque en la lista de diferencias `L#T` y devuelve su tamaño en `I`; y `{File.readClose}`, la cual cierra el archivo.

La función `LeerListaPerezosa` lee un archivo de manera perezosa, un bloque a la vez. Cuando un bloque se acaba, lee el otro automáticamente. Leer bloques es mucho más eficiente que leer simplemente caracteres.

La función `LeerListaPerezosa` es aceptable si el programa lee todo el archivo, pero si sólo lee una parte del archivo, entonces no es suficientemente buena. ¿Ve por qué? ¡Piense cuidadosamente antes de leer la respuesta en la nota de pie de página!¹⁴ En la sección 6.9.2 se muestra la manera correcta de usar la pereza junto con recursos externos tales como los archivos.

4.5.6. El problema de Hamming

El problema de hamming, denominado así en honor a Richard Hamming, es un problema clásico de concurrencia dirigida por la demanda. El problema consiste en generar los primeros n enteros de la forma $2^a3^b5^c$ con $a,b,c \geq 0$. Realmente, Hamming resolvió una versión más general del problema, la cual considera el producto de los primeros k primos. ¡Lo dejamos como un ejercicio! La idea es generar los enteros en orden ascendente en un flujo potencialmente infinito. En todo momento, se conoce una parte finita, h , de este flujo. Para generar el siguiente

14. Esto se debe a que el archivo permanece abierto siempre durante toda la ejecución del programa—esto consume recursos valiosos del sistema, incluyendo un descriptor de archivo y una memoria intermedia de lectura.

elemento de h , tomamos el menor elemento x de h tal que $2x$ es más grande que el último elemento de h . Hacemos lo mismo para 3 y 5, produciendo y y z . Entonces el siguiente elemento es $\min(2x, 3y, 5z)$. Empezamos el proceso inicializando h conteniendo 1 como único elemento. En la figura 4.28 se presenta una gráfica del algoritmo. La manera más sencilla de programar este algoritmo es por medio de dos funciones perezosas. La primera función multiplica todos los elementos de una lista por una constante:

```
fun lazy {MultEscalar N H}
  case H of X|H2 then N*X|{MultEscalar N H2} end
end
```

La segunda función toma dos listas de enteros en orden creciente y las mezcla en una sola lista creciente sin repeticiones:

```
fun lazy {Mezclar Xs Ys}
  case Xs#Ys of (X|Xr)#(Y|Yr) then
    if X<Y then X|{Mezclar Xr Ys}
    elseif X>Y then Y|{Mezclar Xs Yr}
    else X|{Mezclar Xr Yr}
    end
  end
end
```

Cada valor debe aparecer una sola vez en la salida. Por eso, cuando $x==y$, es importante saltar el valor en ambas listas (Xs y Ys). Con estas dos funciones es fácil resolver el problema de Hamming:

```
H=1|{Mezclar {MultEscalar 2 H}
          {Mezclar {MultEscalar 3 H}
                    {MultEscalar 5 H}}}
```

{Browse H}

Construimos una función que mezcla tres argumentos utilizando dos veces la función de mezcla de dos argumentos. Si ejecutamos esto tal cual, lo que despliega el browser es muy poco:

1|_<Future>

No se ha calculado ningún elemento. Para calcular los primeros n elementos de H , debemos solicitar que se calculen. Por ejemplo, podemos definir el procedimiento Tocar:

```
proc {Tocar N H}
  if N>0 then {Tocar N-1 H.2} else skip end
end
```

Este procedimiento recorre N elementos de H , lo cual obliga a que sean calculados. Ahora calculamos 20 elementos invocando Tocar:

{Tocar 20 H}

Esto despliega

1|2|3|4|5|6|8|9|10|12|15|16|18|20|24|25|27|30|32|36|_<Future>

4.5.7. Operaciones perezosas sobre listas

Todas las funciones sobre listas de la sección 3.4 se pueden volver perezosas. Es bastante aleccionador ver cómo este cambio influencia su comportamiento.

Concatenación perezosa

Empezamos con una función sencilla, una versión perezosa de Append:

```
fun lazy {LAppend As Bs}
  case As
  of nil then Bs
  [] A|Ar then A|{LAppend Ar Bs}
  end
end
```

La única diferencia con la versión ansiosa es la anotación “lazy”. La definición perezosa funciona debido a que es recursiva: calcula una parte de la respuesta y luego se invoca a sí misma. La invocación de LAppend con dos listas las concatenará perezosamente:

```
L={LAppend "foo" "bar"}
{Browse L}
```

Decimos que esta función es incremental: al forzar su evaluación sólo se realizan los cálculos suficientes para obtener un elemento adicional de la salida, y entonces se crea otra suspensión. Si “tocamos” elementos sucesivos de L, se desplegará en el browser, consecutivamente, f, o, o, un carácter en cada turno. Sin embargo, después de agotar la entrada “foo”, LAppend se termina y mostrará “bar” en un solo paso. ¿Cómo hacemos una concatenación de dos listas que devuelva una lista totalmente perezosa? Una manera es enviar una lista perezosa como segundo argumento. Primero definamos una función que tome una lista y devuelva una versión perezosa:

```
fun lazy {VolverPerezosa Ls}
  case Ls
  of x|Lr then x|{VolverPerezosa Lr}
  else nil end
end
```

VolverPerezosa funciona iterando sobre la lista de entrada, i.e., igual que en LAppend, se calcula parte de la respuesta y luego se invoca a sí misma. Lo único que cambia esto es el control de flujo: vista como una función entre listas, VolverPerezosa es una identidad. Ahora invoquemos LAppend así:

```
L={LAppend "foo" {VolverPerezosa "bar"}}
{Browse L}
```

Esto enumerará de manera perezosa ambas listas, i.e., sucesivamente devuelve los caracteres f, o, o, b, a, y r.

Map perezoso

Hemos visto la función `Map` en la sección 3.6, la cual evalúa una función sobre todos los elementos de la lista, y devuelve una lista con los resultados. Es fácil definir una versión perezosa de esta función:

```
fun lazy {LMap Xs F}
  case Xs
  of nil then nil
    []
  | x|xr then {F x} | {LMap xr F}
  end
end
```

Esta función toma cualquier lista, perezosa o no, `Xs` y devuelve una lista perezosa. ¿Esta función es incremental?

Listas perezosas de enteros

Definimos la función `{LEntre I J}` que genera una lista perezosa de enteros entre `I` y `J`:

```
fun {LEntre I J}
  fun lazy {LCicloDesde I}
    if I>J then nil else I | {LCicloDesde I+1} end
  end
  fun lazy {LHastaInf I} I | {LHastaInf I+1} end
in
  if J==inf then {LHastaInf I} else {LCicloDesde I} end
end
```

¿Por qué no se anotó a `LEntre` como perezosa?¹⁵ Esta definición permite que `J=inf`, en cuyo caso se genera un flujo infinito, perezoso, de enteros.

Aplanar perezoso

Esta definición muestra que las listas de diferencias perezosas son tan fáciles de generar como las listas perezosas. Tal como ocurrió con las otras funciones perezosas, es suficiente anotar como perezosas todas las funciones recursivas que calculan parte de la solución en cada iteración.

15. Solamente las funciones recursivas deben ser controladas, pues de otra manera podrían llevar a cálculos potencialmente ilimitados.

Concurrencia declarativa

```
fun {LPlanar Xs}
  fun lazy {LPlanarD Xs E}
    case Xs
      of nil then E
      [] X|Xr then
        {LPlanarD X {LPlanarD Xr E}}
      [] X then X|E
    end
  end
in
{LPlanarD Xs nil}
end
```

Hacemos notar que esta definición tiene la misma eficiencia asintótica que la definición ansiosa, i.e., se saca ventaja de la propiedad de concatenación en tiempo constante de las listas de diferencias.

Inversión perezosa de una lista

Hasta ahora, todas las funciones perezosas que hemos introducido son incremenciales, i.e., son capaces de producir elemento por elemento, eficientemente. Algunas veces esto no es posible. Para algunas funciones sobre listas, el trabajo necesario para producir un elemento es suficiente para producirlos todos. Denominamos a esas funciones, funciones monolíticas. Un ejemplo típico es la inversión de una lista. Esta es la definición perezosa:

```
fun {LReverse S}
  fun lazy {Rev S R}
    case S
      of nil then R
      [] X|S2 then {Rev S2 X|R} end
    end
  in {Rev S nil} end
```

Invoquemos esta función:

```
L={LReverse [a b c]}
{Browse L}
```

¿Qué pasa si tocamos el primer elemento de *L*? ¡Se calcula y despliega toda la lista invertida! ¿Por qué pasa esto? Tocar *L* activa la suspensión de *{Rev [a b c] nil}* (recuerde que la propia *LReverse* no está anotada como perezosa). Se ejecuta *Rev* y crea una nueva suspensión para *{Rev [b c] [a]}* (la invocación recursiva), pero no dentro de una lista (de manera que pueda esperar que alguien solicite su valor). Por eso, la nueva suspensión se activa de inmediato. Esto lleva a otra iteración y crea una segunda suspensión, *{Rev [c] [b a]}*. De nuevo, esta suspensión no está dentro de una lista, por lo tanto es activada de inmediato, también. Así se continúa hasta que *Rev* devuelve *[c b a]*. En este momento se tiene una lista, luego la evaluación termina. Como el resultado siempre fue necesitado, se realizó el recorrido completo de la lista. A esto es lo que llamamos una función monolítica. Para el caso de la inversión de una lista, otra forma de entender este comportamiento

es pensar en lo que significa invertir una lista: el primer elemento de la lista de salida es el último elemento de la lista de entrada. Por lo tanto tenemos que recorrer toda la lista de entrada, lo cual nos lleva a construir toda la lista invertida de una sola vez.

Filtro perezoso

Para terminar esta sección, presentamos otro ejemplo de una función incremental, a saber, filtrar una lista de entrada de acuerdo a una condición F :

```
fun lazy {LFilter L F}
  case L
  of nil then nil
  [] X|L2 then
    if {F X} then X|{LFilter L2 F} else {LFilter L2 F} end
  end
end
```

Presentamos esta función porque la necesitaremos para la sección 4.5.9 sobre listas comprehensivas.

4.5.8. Colas persistentes y diseño de algoritmos

En la sección 3.4.5 vimos cómo construir colas con operaciones de inserción y borrado en tiempo constante. Aquellas colas sólo funcionan en el caso efímero, i.e., solamente existe una versión en todo momento. Resulta que podemos usar la pereza para construir colas persistentes con los mismos límites de tiempo. Una cola persistente es aquella que soporta múltiples versiones. Primero mostramos cómo hacer una cola persistente amortizada con operaciones de inserción y borrado en tiempo constante. Luego mostramos cómo lograr tiempo constante en el peor caso.

4.5.8.1. Cola persistente amortizada

Primero abordamos el caso amortizado. La razón por la que la cola de la sección 3.4.5 no es persistente es que `ElimDeCola` hace algunas veces una inversión de una lista, la cual no se hace en tiempo constante. Cada vez que se realiza un `ElimDeCola` sobre la misma versión, se realiza otra inversión de la lista. Esto no permite una complejidad amortizada en caso de haber múltiples versiones.

Podemos recobrar la complejidad amortizada realizando la inversión como parte de la invocación a una función perezosa. Así, al invocar la función perezosa se crea una suspensión en lugar de realizar la inversión de una vez. Un tiempo después, cuando se necesite el resultado de la inversión, la función perezosa realizará la inversión. Con un poco de ingenio, esto puede resolver nuestro problema:

- Entre la creación de la suspensión y la verdadera ejecución de la inversión, nos arreglamos para que existan suficientes operaciones que soporten los costos en que incurre la inversión.

Concurrencia declarativa

- Pero la inversión sólo se puede compensar una vez. ¿Qué pasa si varias versiones desean realizar una inversión? Esto no es problema. La pereza garantiza que la inversión se realiza una única vez, aún si es disparada por más de una versión. La primera versión que la necesite activará el disparador y salvará el resultado. Las otras versiones usarán el resultado sin ningún cálculo adicional.

Esto suena bien, pero depende de ser capaces de crear la suspensión suficientemente lejos del momento de la inversión real. ¿Podemos hacerlo? En el caso de una cola, podemos. Representemos la cola como una cuádrupla:

```
q(LonF F LonR R)
```

F y R son las listas del frente y de la parte de atrás, tal como en el caso efímero. Añadimos los enteros LonF y LonR, los cuales representan las longitudes de F y R. Necesitamos estos enteros para determinar cuándo es el momento de crear la suspensión. En algún instante mágico, moveremos los elementos de R a F. La cola entonces se convierte en

```
q(LonF+LonR {LAppend F {fun lazy {$} {Reverse R} end}} 0 nil)
```

En la sección 3.4.5 hacemos esto (ansiosamente) cuando F se vuelve vacía, de manera que Append no toma tiempo. Pero esto sería demasiado tarde en este caso, para conservar la complejidad amortizada, porque el costo de la inversión puede que no se alcance a compensar (e.g., puede ser que R sea una lista muy grande). Nótese que la inversión queda evaluada, en todos los casos, cuando la invocación a LAppend haya terminado, i.e., después que se hayan eliminado $|F|$ elementos de la cola. ¿Podemos arreglar que los elementos de F compensen el costo de la inversión? Podemos, si creamos la suspensión cuando $|R| \approx |F|$. Entonces, la eliminación de cada elemento de F compensa parte de la inversión. En el momento en que debamos efectuar la inversión, ésta ya estará completamente compensada. Utilizar la concatenación perezosa hace que la compensación sea incremental. Esto nos lleva a la implementación siguiente:

```

fun {ColaNueva} q(0 nil 0 nil) end

fun {CompCola Q}
  case Q of q(LonF F LonR R) then
    if LonF>=LonR then Q
    else q(LonF+LonR {LAppend F {fun lazy {$} {Reverse R} end}})
          0 nil) end
  end
end

fun {InsCola Q X}
  case Q of q(LonF F LonR R) then
    {CompCola q(LonF F LonR+1 X|R)}
  end
end

fun {ElimDeCola Q X}
  case Q of q(LonF F LonR R) then F1 in
    F=X|F1 {CompCola q(LonF-1 F1 LonR R)}
  end
end

```

Tanto `InsCola` como `ElimDeCola` invocan la función `CompCola`, la cual escoge el momento para realizar la invocación perezosa. Como `InsCola` aumenta $|R|$ y `ElimDeCola` disminuye $|F|$, finalmente $|R|$ se vuelve más grande que $|F|$. Cuando $|R| = |F| + 1$, `CompCola` realiza la invocación perezosa `{LAppend F {fun lazy {$} {Reverse R} end}}`. La función `LAppend` se definió en la sección 4.5.7.

Resumamos esta técnica. Reemplazamos la invocación original a la función ansiosa por una invocación a una función perezosa. La invocación perezosa es parcialmente incremental y parcialmente monolítica. Al momento en que la parte monolítica se alcanza, deben haberse realizado suficientes etapas incrementales de manera que se compense el costo de la parte monolítica. En consecuencia, el resultado es de complejidad constante en tiempo amortizado.

Para una discusión más profunda de esta técnica, incluyendo su aplicación en otras estructuras de datos y una prueba de corrección, recomendamos [129].

4.5.8.2. Cola persistente en tiempo constante en el peor caso

La razón por la que la definición anterior no es tiempo constante en el peor caso es porque `Reverse` es monolítico. Si pudiéramos reescribirlo para ser incremental, entonces tendríamos una solución con comportamiento en tiempo constante en el peor caso. Pero la inversión de una lista no se puede hacer incremental, de manera que esta idea no funciona. Ensayemos otro enfoque.

Miremos el contexto de la invocación a `Reverse`, la cual se hace junto con una concatenación perezosa:

```
{LAppend F {fun lazy {$} {Reverse R} end}}
```

Primero se ejecuta la concatenación incrementalmente. Cuando todos los elementos

Concurrencia declarativa

de F se han pasado a la salida, entonces se ejecuta la inversión de manera monolítica. El costo de la inversión es amortizado por las etapas de la concatenación.

En lugar de amortizar el costo de la inversión, podríamos, tal vez, realizar la inversión real al mismo tiempo que se realizan las etapas de la concatenación. Así, cuando la concatenación haya terminado, habrá terminado la inversión también. Este es el corazón de la solución. Para implementarla, comparemos las definiciones de la concatenación y la inversión. La inversión utiliza la función recursiva Rev :

```
fun {Reverse R}
  fun {Rev R A}
    case R
      of nil then A
      [] X|R2 then {Rev R2 X|A} end
    end
  in {Rev R nil} end
```

Rev recorre R , acumula una solución en A , y luego devuelve la solución. ¿Podemos hacer ambos, Rev y LAppend en un solo ciclo? Este es LAppend :

```
fun lazy {LAppend F B}
  case F
  of nil then B
  [] X|F2 then X|{LAppend F2 B}
  end
end
```

Esta función recorre F y devuelve B . En la invocación recursiva, B se pasa sin ningún cambio. ¡Cambiemos esto, y utilicemos B para acumular el resultado de la inversión! Esto produce la siguiente función combinada:

```
fun lazy {LAppRev F R B}
  case F#R
  of nil#[Y] then Y|B
  [] (X|F2)#[Y|R2) then X|{LAppRev F2 R2 Y|B}
  end
end
```

LAppRev recorre tanto F como R . En cada iteración, se calcula un elemento de la concatenación y se acumula un elemento de la inversión. Esta definición sólo funciona si R tiene exactamente un elemento más que F , lo cual es cierto para nuestra cola. La invocación original

```
{LAppend F {fun lazy {$} {Reverse R} end}}
```

se reemplaza por

```
{LAppRev F R nil}
```

lo cual produce exactamente el mismo resultado, salvo que LAppRev es totalmente incremental. La definición de CompCola queda entonces:

```
fun {CompCola Q}
  case Q of q(LonF F LonR R) then
    if LonR=<LonF then Q
    else q(LonF+LonR {LAppRev F R nil} 0 nil) end
  end
end
```

Un análisis cuidadoso mostrará que la complejidad de esta cola en el peor caso es $O(\log n)$, y no $O(1)$ como nuestra intuición podría esperarlo. Esta complejidad es mucho mejor que $O(n)$, pero no es constante. Vea los ejercicios (sección 4.11) para una explicación y una sugerencia de cómo lograr una complejidad constante.

Si a un programa se le agrega la pereza de manera ingenua como lo hemos hecho, su complejidad en el peor caso, será una cota para su complejidad amortizada. Esto se debe a que la pereza cambia en dónde se ejecutan las invocaciones a funciones, pero no hace más que eso (el caso ansioso es una cota superior). La definición de esta sección es extraordinaria porque hace justo lo opuesto: empieza con una cota amortizada y utiliza la pereza para producir la misma cota en el peor caso.

4.5.8.3. Lecciones para el diseño de algoritmos

La pereza es capaz de barajar los cálculos alrededor, difundirlos hacia afuera o agruparlos sin cambiar el resultado final. Por ello es una herramienta poderosa para diseñar algoritmos declarativos. Sin embargo, debe ser utilizada cuidadosamente. Si se usa de manera inepta, la pereza puede perfectamente destruir buenos límites de complejidad en el peor caso, volviéndolos límites amortizados. Utilizada sabiamente, la pereza puede mejorar algoritmos amortizados: algunas veces puede volver el algoritmo persistente y algunas veces puede transformar un orden de complejidad amortizado en un orden de complejidad en el peor caso.

Podemos bosquejar un esquema general. Comience con un algoritmo A que tiene un orden amortizado $O(f(n))$ cuando se usa de manera efímera. Por ejemplo, la primera cola de la sección 3.4.5 tiene un orden amortizado $O(1)$. Podemos usar la pereza para convertirla de efímera a persistente, conservando ese orden de complejidad. Aquí se presentan dos posibilidades:

- Frecuentemente podemos llegar a un algoritmo modificado A' que conserva el orden $O(f(n))$ amortizado, cuando se usa de manera persistente. Esto es posible cuando las operaciones costosas pueden ser diseminadas de manera que la mayoría se vuelven incrementales, y unas pocas permanencen como operaciones monolíticas.
- En unos pocos casos, podemos ir más allá y llegar a un algoritmo modificado A'' con orden de complejidad $O(f(n))$ en el peor caso, cuando se usa de manera persistente. Esto es posible cuando las operaciones costosas se pueden diseminar y convertir completamente en operaciones incrementales.

En esta sección se desarrollaron ambas posibilidades con la primera cola de la sección 3.4.5. Los algoritmos persistentes así obtenidos son, con frecuencia, bastante eficientes, especialmente si son utilizados por aplicaciones que realmente necesitan

la persistencia. Estos algoritmos se comparan favorablemente con los algoritmos en modelos con estado.

4.5.9. Listas por comprehensión

Las listas por comprehensión no son más que una conveniencia notacional para definir flujos. Los flujos pueden ser limitados, i.e., ellos son listas, o potencialmente ilimitados. En el último caso, es bastante útil calcularlos perezosamente. Las listas por comprehensión pueden ser útiles para todas esas posibilidades. Las introducimos aquí, en el contexto de la ejecución perezosa, pues ellas son especialmente interesantes con flujos perezosos. Las listas por comprehensión permiten la especificación de flujos en una forma que se parece mucho a la notación matemática para definición de conjuntos por comprehensión. Por ejemplo, la notación matemática $\{x * y \mid 1 \leq x \leq 10, 1 \leq y \leq x\}$ especifica el conjunto $\{1 * 1, 2 * 1, 2 * 2, 3 * 1, 3 * 2, 3 * 3, \dots, 10 * 10\}$, i.e. $\{1, 2, 3, 4, 5, \dots, 100\}$. Convertimos esta notación en una herramienta de programación práctica, modificándola, ya no para especificar conjuntos, sino flujos perezosos. Así se logra una notación muy eficiente para programar, conservando un alto nivel de abstracción. Por ejemplo, la lista por comprehensión $[x * y \mid 1 \leq x \leq 10, 1 \leq y \leq x]$ especifica la lista $[1 * 1 \ 2 * 1 \ 2 * 2 \ 3 * 1 \ 3 * 2 \ 3 * 3 \dots 10 * 10]$ (en este orden), i.e., la lista $[1 \ 2 \ 4 \ 3 \ 6 \ 9 \dots 100]$. La lista se calcula perezosamente. Gracias a la pereza, la lista por comprehensión puede generar un flujo potencialmente ilimitado, y no solamente una lista finita.

Las listas por comprehensión tienen la forma básica siguiente:

$$[f(x) \mid x \leftarrow \text{generador}(a_1, \dots, a_n), \text{condición}(x, a_1, \dots, a_n)]$$

El generador $x \leftarrow \text{generador}(a_1, \dots, a_n)$ calcula una lista perezosa cuyos elementos son sucesivamente asignados a x . La condición $\text{condición}(x, a_1, \dots, a_n)$ es una función booleana. La lista de comprehensión especifica una lista perezosa que contiene los elementos $f(x)$, donde f es cualquier función y x toma los valores del generador para los cuales la condición sea cierta. En el caso general, puede haber cualquier número de variables, generadores, y condiciones. Un generador típico es *desde*:

$$x \leftarrow \text{desde}(a, b)$$

Aquí x toma los valores enteros $a, a + 1, \dots, b$, en ese orden. Los cálculos se hacen de izquierda a derecha. Los generadores, tomados de izquierda a derecha, son considerados ciclos: el generador más a la derecha corresponde al ciclo más interno.

Hay una conexión muy cercana entre las listas por comprehensión y la programación relacional del capítulo 9 (en CTM). Ambas proveen interfaces perezosas para secuencias infinitamente largas y facilitan la escritura de programas “generar-y-comprobar”. Ambas permiten especificar las secuencias de forma declarativa.

Aunque las listas por comprehensión se consideran, normalmente, perezosas, se pueden programar tanto en versiones ansiosas como perezosas. Por ejemplo, la lista por comprehensión:

$$z = [x \# x \mid x \leftarrow \text{desde}(1, 10)]$$

se puede programar de dos maneras. Una versión ansiosa es

```
z={Map {Entre 1 10} fun {$ x} x#x end}
```

Para la versión ansiosa, el modelo declarativo del capítulo 2 es suficientemente bueno. Se utiliza la función `Map` de la sección 3.6.3 y la función `Entre` que genera una lista de enteros. Una versión perezosa es

```
z={LMap {LEntre 1 10} fun {$ x} x#x end}
```

La versión perezosa utiliza las funciones `LMap` y `LEntre` definidas en la sección precedente. Este ejemplo y la mayoría de los ejemplos de esta sección se pueden hacer en versión ansiosa o perezosa. Utilizar la versión perezosa siempre es correcto. Utilizar la versión ansiosa es una optimización del desempeño. Ésta es varias veces más rápida si no se cuenta el costo de calcular los elementos de la lista. La optimización sólo es posible si la lista completa cabe en la memoria. En el resto de esta sección, siempre utilizaremos la versión perezosa.

A continuación presentamos una lista de comprehensión con dos variables:

$$z = [x \# y \mid x \leftarrow \text{desde}(1, 10), y \leftarrow \text{desde}(1, x)]$$

Ésta se puede programar como

```
z={LAplanar
    {LMap {LEntre 1 10} fun {$ x}
        {LMap {LEntre 1 x} fun {$ y}
            x#y
            end}
        end}}}
```

Hemos visto `LAplanar` en una sección anterior; `LAplanar` convierte una lista de listas a una lista perezosa “plana”, i.e., una lista perezosa que contiene todos los elementos, pero no listas. Necesitamos `LAplanar` porque de otra forma tendríamos una lista de listas. Podemos colocar `LAplanar` dentro de `LMap`:

```
fun {FMap L F}
    {LAplanar {LMap L F}}
end
```

Esto simplifica el programa:

```
z={FMap {LEntre 1 10} fun {$ x}
    {LMap {LEntre 1 x} fun {$ y}
        x#y
        end}
    end}
```

Ahora un ejemplo con dos variables y una condición:

$$z = [x \# y \mid x \leftarrow \text{desde}(1, 10), y \leftarrow \text{desde}(1, 10), x + y \leq 10]$$

Esto produce la lista de todas las parejas $x \# y$ tal que la suma $x + y$ es máximo 10. Se puede programar como

Concurrencia declarativa

```
Z={LFilter
    {FMap {LEntre 1 10} fun {$ x}
        {LMap {LEntre 1 10} fun {$ y}
            X#Y
            end}
        end}
    fun {$ x#y} X+Y=<10 end}
```

Aquí se utiliza la función `LFilter` definida en la sección anterior. Podemos reformular este ejemplo para hacerlo más eficiente. La idea es generar tan pocos elementos como sea posible. En el ejemplo de arriba, se generan 100 ($=10^2$) elementos. De $2 \leq x + y \leq 10$ y $1 \leq y \leq 10$, podemos inferir que $1 \leq y \leq 10 - x$. Esto produce la solución siguiente:

$$z = [x \# y \mid x \leftarrow \text{desde}(1, 10), y \leftarrow \text{desde}(1, 10 - x)]$$

El programa se vuelve entonces:

```
Z={FMap {LEntre 1 10} fun {$ x}
    {LMap {LEntre 1 10-x} fun {$ y}
        X#Y
        end}
    end}
```

Esto produce la misma lista de antes, pero sólo genera cerca de la mitad de todos los elementos.

4.6. Programación en tiempo real no estricto

4.6.1. Operaciones básicas

El módulo `Time` contiene una buena cantidad de operaciones útiles para la programación en tiempo real no estricto. Una operación en tiempo real tiene un conjunto de plazos (tiempos específicos) en los cuales deben completarse ciertos cálculos. Una operación en tiempo real no estricto sólo requiere que los plazos de tiempo real se respeten la mayoría de las veces. Esto en contraste con tiempo real estricto, cuyos plazos son estrictos, i.e., deben ser respetados todo el tiempo, sin ninguna excepción. El tiempo real estricto se requiere cuando hay vidas en juego, e.g., en equipos médicos y en control de tráfico aéreo. El tiempo real no estricto se usa en otros casos, e.g., telefonía y electrónica de consumo. El tiempo real estricto requiere técnicas especiales tanto en software como en hardware. Los computadores personales estándar no son adecuados para el tiempo real estricto debido a retrasos imprevisibles en el hardware (e.g., memoria virtual, planificación de procesos, caching). El tiempo real no estricto es mucho más fácil de implementar y, la mayoría de las veces, es suficiente. Tres de las operaciones que provee el módulo `Time` son:

1. `{Delay I}`: suspende el hilo en ejecución por al menos `I` ms y luego continúa.

```

local
  proc {Ping N}
    if N==0 then {Browse 'ping terminado'}
    else {Delay 500} {Browse ping} {Ping N-1} end
  end
  proc {Pong N}
    {For 1 N 1
      proc {$ I} {Delay 600} {Browse pong} end
    {Browse 'pong terminado'}
  end
in
  {Browse 'comienzo del juego'}
  thread {Ping 50} end
  thread {Pong 50} end
end

```

Figura 4.29: Un programa sencillo de ‘Ping Pong’.

2. {Alarm I U}: crea un hilo nuevo que liga U to **unit** después de pasados I ms al menos.
3. {Time.time}: devuelve un número entero que representa los segundos que han pasado desde la iniciación del año actual.

La semántica de **Delay** es sencilla: él comunica al planificador que el hilo debe ser suspendido por un período de tiempo dado. Después de pasado ese tiempo, el planificador marca, de nuevo, el hilo como ejecutable. El hilo no se ejecuta, necesariamente, de inmediato. Si existen muchos otros hilos ejecutables, puede pasar un tiempo antes que el hilo realmente se ejecute.

Ilustramos el uso de **Delay** por medio de un ejemplo sencillo que muestra la ejecución intercalada de dos hilos. El programa se llama ‘Ping Pong’ y se define en la figura 4.29. El programa inicia dos hilos. Un hilo despliega ping periódicamente cada 500 ms y el otro despliega pong cada 600 ms. Como los pongs se producen más lentamente que los pings, es posible que se lleguen a desplegar dos pings sin pongs en el medio. ¿Puede ocurrir lo mismo con dos pongs? Es decir, ¿se pueden desplegar dos pongs sin pings en la mitad? Suponga que el hilo de Ping no ha terminado aún. De otra forma el asunto sería demasiado fácil. Piense cuidadosamente antes de leer la respuesta en la nota de pie de página.¹⁶

16. El lenguaje puede de hecho permitir que dos pongs se desplieguen sin intervención de pings debido a que la definición **Delay** sólo especifica el tiempo mínimo de suspensión. Un hilo suspendido por 500 ms puede estar suspendido, ocasionalmente, por un tiempo mayor, e.g., por 700 ms. Pero esto ocurre rara vez en la práctica pues depende de eventos externos en el sistema operativo o en otros hilos.

Concurrencia declarativa

```
functor
import
    Browser(browse:Browse)
define
    proc {Ping N}
        if N==0 then {Browse `ping terminado`}
        else {Delay 500} {Browse ping} {Ping N-1} end
    end
    proc {Pong N}
        {For 1 N 1
            proc {$ I} {Delay 600} {Browse pong} end }
        {Browse `pong terminado`}
    end
in
    {Browse `comienzo del juego`}
    thread {Ping 50} end
    thread {Pong 50} end
end
```

Figura 4.30: Un programa autónomo de ‘Ping Pong’.

Una aplicación autónoma sencilla

En la sección 3.9 se muestra cómo producir aplicaciones autónomas en Oz. Para crear el programa ‘Ping Pong’ como aplicación autónoma, lo primero que hay que hacer es un functor del programa, como se muestra en la figura 4.30. Si el código fuente está almacenado en el archivo `PingPong.oz`, entonces el programa se puede compilar con el comando siguiente:

```
ozc -x PingPong.oz
```

Digite `PingPong` en su consola para comenzar el programa. Para terminarlo en una consola Unix, teclee CTRL-C.

El programa de la figura 4.30 no termina limpiamente cuando terminan los hilos del `Ping` y del `Pong`. De hecho, no se detecta cuándo terminan los hilos. Podemos resolver este problema usando las técnicas de la sección 4.4.3. En la figura 4.31 se agrega una detección de terminación que termina el hilo principal sólo cuando terminan los hilos del `Ping` y del `Pong`. También podríamos usar directamente la abstracción `Barrera`. Después de detectar la terminación, utilizamos la invocación `{Application.exit 0}` para salir limpiamente de la aplicación.

4.6.2. Producción de pulsos

Nos gustaría poder invocar una acción (e.g., enviar un mensaje a un objeto flujo, invocar un procedimiento, etc.) exáctamente una vez por segundo, teniendo la hora local como argumento. Tenemos tres operaciones a nuestra disposición:

```

functor
import
    Browser(browse:Browse)
    Application
define
    ...
    X1 X2
in
    {Browse `comienzo del juego`}
    thread {Ping 50} X1=listo end
    thread {Pong 50} X2=listo end
    {Wait X1} {Wait X2}
    {Application.exit 0}
end

```

Figura 4.31: Un programa autónomo de ‘Ping Pong’ que termina limpiamente.

{Delay D}, la cual produce un retraso de al menos D ms, {Time.time}, la cual devuelve el número de segundos transcurridos desde que comenzó el año actual, y {OS.localTime}, la cual devuelve un registro con la hora local, con precisión de un segundo. ¿Qué hace la función siguiente?:

```

fun {ProductorDePulsos}
    fun {Ciclo}
        X={OS.localTime}
        in
            {Delay 1000}
            X|{Ciclo}
        end
    in
        thread {Ciclo} end
    end

```

Esta función crea un flujo que crece en un elemento por segundo. Para ejecutar una acción cada segundo, se crea un hilo que lee el flujo y realiza la acción:

```
thread for X in {ProductorDePulsos} do {Browse X} end end
```

Cualquier cantidad de hilos puede leer el mismo flujo. El problema es que esta solución no es del todo correcta. El flujo se extiende casi exactamente una vez por segundo. El problema es el “casi.” Cada cierto tiempo, se pierde un segundo, i.e., elementos consecutivos en el flujo muestran una diferencia de dos segundos. Sin embargo, hay un buen punto: el mismo segundo no puede ser enviado dos veces, pues {Delay 1000} garantiza un retraso de al menos 1000 ms, a lo cual se añade la ejecución de las instrucciones en Ciclo. Esto nos lleva a un retraso total de al menos $1000 + \varepsilon$ ms, donde ε es una fracción de un microsegundo.

¿Cómo podemos corregir este problema? Una manera sencilla es comparar el resultado actual de OS.localTime con el resultado previo, y agregar un elemento al flujo sólo cuando el tiempo local haya cambiado. Esto nos lleva a:

Concurrencia declarativa

```
fun {ProductorDePulsos}
    fun {Ciclo T}
        T1={OS.localTime}
    in
        {Delay 900}
        if T1\=T then T1|{Ciclo T1} else {Ciclo T1} end
    end
    in
        thread {Ciclo {OS.localTime}} end
    end
```

Esta versión garantiza que se envía exactamente un tick por segundo, si {Delay 900} retrasa todo siempre menos de un segundo. Esta última condición es cierta si no hay demasiados hilos activos y la recolección de basura no toma mucho tiempo. Una manera de garantizar la primera condición es asignar al hilo de Ciclo una alta prioridad y a todos los otros hilos una prioridad media o baja. Para garantizar la segunda condición, el programa debe asegurar que no existen muchos datos activos, pues el tiempo de recolección de basura es proporcional a la cantidad de datos activos.

Esta versión tiene un problema menor y es que “duda” cada nueve segundos. Es decir, puede pasar que {os.localTime} produzca el mismo resultado dos veces seguidas, pues las invocaciones están separadas solamente por algo más de 900 ms. Esto significa que el flujo no será actualizado por 1800ms. Otra forma de ver este problema es que se necesitan diez intervalos de 900 ms para cubrir nueve segundos, lo cual significa que durante alguno de los intervalos no pasa nada. ¿Cómo podemos evitar esta duda? Una manera sencilla es hacer el retraso más pequeño. Con un retraso de 100ms, la duda nunca será mayor de 100 ms más el tiempo de recolección de basura.

Una mejor manera de evitar la duda es utilizar relojes sincronizados. Es decir, creamos un contador de libre ejecución que se ejecuta aproximadamente a la velocidad de un segundo por tick, y ajustamos su velocidad de manera que permanezca sincronizada con el tiempo del sistema operativo. Esto se hace así:

```
fun {ProductorDePulsos}
    fun {Ciclo N}
        T={Time.time}
    in
        if T>N then {Delay 900}
        elseif T<N then {Delay 1100}
        else {Delay 1000} end
        N|{Ciclo N+1}
    end
    in
        thread {Ciclo {Time.time}} end
    end
```

El ciclo tiene un contador, n , que se incrementa siempre en 1. Comparamos el valor del contador con el valor del resultado de `{Time.time}`.¹⁷ Si el contador va más lento ($T > N$), lo aceleramos. Si por el contrario, el contador va más rápido ($T < N$) lo desaceleraremos. Los factores de aceleración y desaceleración son pequeños (10 % en el ejemplo), lo cual hace que la duda sea imperceptible.

4.7. El lenguaje Haskell

Presentamos una breve introducción a Haskell, un popular lenguaje de programación soportado por varios interpretadores y compiladores [74, 136].¹⁸ Este lenguaje es, tal vez, el intento más exitoso por definir un lenguaje completamente declarativo y práctico. Haskell es un lenguaje de programación funcional fuertemente tipado, no estricto, que soporta la currificación y el estilo de programación monádico. Por “fuertemente tipado” se entiende que los tipos de todas las expresiones se calculan en tiempo de compilación y todas las aplicaciones de función deben ser correctas en términos de tipos.

El estilo monádico es un conjunto de técnicas de programación de alto orden que se usan para reemplazar el estado explícito en muchos casos. Con el estilo monádico se puede hacer mucho más que sólo simular el estado; esto no lo explicaremos en esta breve introducción, pero haremos referencia a cualquiera de los muchos artículos y tutoriales que se han escrito sobre este tema [75, 187, 126].

Antes de presentar el modelo de computación, empezaremos con un ejemplo sencillo. Podemos escribir la función factorial en Haskell como sigue:

```
factorial :: Integer -> Integer
factorial 0          = 1
factorial n | n > 0 = n * factorial (n-1)
```

La primera línea es la firma del tipo de la función. Con ella se especifica que `factorial` es una función que espera un argumento de tipo `Integer` y devuelve un resultado de tipo `Integer`. Haskell realiza inferencia de tipos, i.e., el compilador es capaz de inferir automáticamente las firmas de los tipos para casi todas las funciones.¹⁹ Esto sucede aún cuando se provee la firma del tipo de la función: el compilador, en este caso, comprueba que la firma sea correcta. Las firmas de tipo de las funciones proveen documentación útil.

Las siguientes dos líneas corresponden al código de `factorial`. En Haskell una definición de función puede consistir de muchas ecuaciones. Para aplicar una función a un argumento debemos hacer reconocimiento de patrones; examinamos

17. ¿Cómo corregiríamos `ProductorDePulsos` para funcionar correctamente cuando `Time.time` da la vuelta, i.e., vuelve y tome el valor 0?

18. El autor de esta sección es Kevin Glynn.

19. Salvo para unos pocos casos muy especiales que están más allá del alcance de esta sección, tales como recursión polimórfica.

las ecuaciones, una por una, de arriba hacia abajo hasta encontrar la primera cuyo patrón corresponde con el argumento. La primera línea de `factorial` solamente corresponde con el argumento 0; en este caso la respuesta es inmediata, a saber, 1. Si el argumento es diferente de cero tratamos de ver si corresponde con la segunda ecuación. Esta ecuación tiene una condición booleana la cual debe ser cierta para que el reconocimiento tenga éxito. La segunda ecuación corresponde con todos los argumentos que sean mayores que cero; en ese caso evaluamos `n * factorial (n-1)`. ¿Qué pasa si aplicamos `factorial` a un argumento negativo? Ninguna de las ecuaciones corresponde y el programa produce un error en tiempo de ejecución.

4.7.1. Modelo de computación

Un programa en Haskell consiste de una sola expresión. Esta expresión puede contener muchas subexpresiones reducibles. ¿En qué orden deberían ser evaluadas? Haskell es un lenguaje no estricto, de manera que ninguna expresión debería ser evaluada a menos que su resultado definitivamente se necesite. Intuitivamente, entonces, deberíamos reducir primero la expresión más a la izquierda hasta que sea una función, substituir los argumentos en el cuerpo del procedimiento (`jsin evaluarlos!`) y entonces reducir la expresión resultante. A este orden de evaluación se le denomina orden normal. Por ejemplo, considere la expresión siguiente:

```
(if n >= 0 then factorial else error) (factorial (factorial n))
```

Aquí se usa `n` para escoger cual función, `factorial` o `error`, aplicarle al argumento `(factorial (factorial n))`. No tiene sentido evaluar el argumento hasta que no hayamos evaluado la declaración `if then else`. Una vez ésta es evaluada, podemos substituir `factorial (factorial n)` en el cuerpo de `factorial` o de `error` según sea lo apropiado, y continuar con la evaluación.

Expliquemos en una forma más precisa cómo se reducen las expresiones en Haskell. Imagine toda expresión como un árbol.²⁰ Haskell evalúa primero la subexpresión más a la izquierda hasta que esta evaluación resulta en un constructor de datos o en una función:

- Si resulta en un constructor de datos, entonces la evaluación termina. Ninguna de las subexpresiones restantes se evalúa.
- Si resulta en una función y ésta no está aplicada a ningún argumento, entonces la evaluación termina.
- De otra manera, si resulta en una función que está aplicada a uno o varios argumentos, entonces aplica la función al primer argumento (sin evaluarlo), substituyéndolo en el cuerpo de la función, y vuelve a evaluar.

20. Por razones de eficiencia, la mayoría de las implementaciones de Haskell representan las expresiones como grafos, i.e., las expresiones compartidas se evalúan una sola vez.

Algunas funciones predefinidas como la suma y el reconocimiento de patrones obligan a que los argumentos se evalúen antes de evaluar su cuerpo. El orden de evaluación normal tiene una muy agradable propiedad: todo programa declarativo que termine con algún otro orden de evaluación, también termina con este orden de evaluación.

4.7.2. Evaluación perezosa

Como los argumentos de las funciones no se evalúan automáticamente antes de la invocación, decimos que la invocación a funciones en Haskell es flexible²¹. Aunque el lenguaje Haskell no lo obliga, la mayoría de las implementaciones de Haskell son de hecho perezosas, i.e., aseguran que las expresiones se evalúan a lo sumo una vez. Las diferencias entre evaluación perezosa y flexible se explican en la sección 4.9.2.

Los compiladores de Haskell, en su proceso de optimización, realizan un análisis denominado análisis de flexibilidad para determinar cuándo no se necesita la pereza para terminación o control de recursos. Las funciones que no necesitan la pereza se compilan como funciones ansiosas (“estrictas”), lo cual es mucho más eficiente.

Como un ejemplo de uso de la pereza, reconsideremos el cálculo de una raíz cuadrada por el método de Newton presentado en la sección 3.2. La idea es crear primero una lista “infinita” que contenga cada vez mejores aproximaciones de la raíz cuadrada. Luego recorremos la lista hasta encontrar la primera aproximación que sea suficientemente exacta y la devolvemos. Gracias a la pereza sólo se agregarán a la lista tantas aproximaciones como se necesiten.

```
raíz x = head (dropWhile (not . buena) aproxRaíz)
where
    buena adiv = (abs (x - adiv*adiv))/x < 0.00001
    mejorar adiv = (adiv + x/adiv)/2.0
    aproxRaíz = 1:(map mejorar aproxRaíz)
```

Las definiciones después de la palabra clave `where` son definiciones locales, i.e., ellas sólo son visibles dentro de `raíz`. La función `buena` se evalúa a cierto si la aproximación actual está suficientemente cerca de la real. La función `mejorar` recibe una aproximación y devuelve otra, mejor. La función `aproxRaíz` produce la lista infinita de aproximaciones. El símbolo `:` es el constructor de listas, equivalente a `|` en Oz. La primera aproximación es 1. Las siguientes aproximaciones se calculan aplicando la función `mejorar` a la lista de aproximaciones. La función `map` aplica una función a todos los elementos de una lista, similar a `Map` en Oz.²² De esta manera, el segundo elemento de `aproxRaíz` será `mejorar 1`, el tercer elemento será `mejorar (mejorar 1)`, y así sucesivamente. Para calcular el n -ésimo elemento de la lista, evaluamos `mejorar` sobre el $n - 1$ -ésimo elemento de la lista.

21. Nota del traductor: *nonstrict*, en inglés.

22. Note que sus argumentos, la función y la lista, aparecen en un orden distinto en las versiones de Haskell y Oz.

Concurrencia declarativa

La expresión `dropWhile (not . buena) approxRaíz` quita del frente de la lista las aproximaciones que no estén suficientemente cerca. La expresión `(not . buena)` es una composición de funciones; se aplica `buena` a la aproximación y la función booleana `not` al resultado. Entonces, `(not . buena)` es una función que devuelve cierto si `buena` devuele falso.

Finalmente, `head` devuelve el primer elemento de la lista resultante, el cual corresponde a la primera aproximación que está suficientemente cerca de la raíz real. Note cómo se ha separado el cálculo de las aproximaciones del cálculo que escoge la respuesta apropiada.

4.7.3. Currificación

De acuerdo a las reglas de reducción vemos que una función que espera múltiples argumentos, realmente es aplicada a cada uno, uno a la vez. De hecho, la aplicación de una función de n argumentos en un solo argumento, resulta en una función de $(n - 1)$ argumentos, especializada en el valor del primer argumento. Este proceso se denomina currificación (ver también la sección 3.6.6). Podemos escribir una función que multiplica por dos todos los elementos de una lista invocando `map` sólo con un argumento:

```
doblarLista = map (\x -> 2*x)
```

La notación `\x -> 2*x` es la forma de escribir en Haskell una función anónima, i.e., una expresión λ . El carácter ASCII \ representa el símbolo λ . En Oz la misma expresión se escribiría `fun { $ x } 2*x end`. Miremos cómo se evalúa `doblarLista`:

```
doblarLista [1,2,3,4]
=> map (\x -> 2*x) [1,2,3,4]
=> [2,4,6,8]
```

Note que los elementos de la lista están separados por comas en Haskell.

4.7.4. Tipos polimórficos

Toda expresión en Haskell tienen un tipo determinado estáticamente. Sin embargo, no estamos limitados a tipos predefinidos en Haskell. Un programa puede introducir tipos nuevos. Por ejemplo, podemos introducir el tipo nuevo `ÁrbolBin` para árboles binarios:

```
data ÁrbolBin a = Vacío | Nodo a (ÁrbolBin a) (ÁrbolBin a)
```

Un `ÁrbolBin` es `Vacío` o es un `Nodo` consistente de un elemento y de dos subárboles. `Vacío` y `Nodo` son constructores de datos: ellos construyen estructuras de datos de tipo `ÁrbolBin`. En la definición, `a` es una variable de tipo y representa un tipo arbitrario, el tipo de los elementos contenidos en el árbol. Entonces, `ÁrbolBin Integer` representa el tipo de los árboles binarios de enteros. Note cómo se restringen el elemento de un `Nodo` y sus subárboles a tener el mismo tipo. Podemos escribir una

función `tamaño` que devuelve el número de elementos en un árbol binario, de la manera siguiente:

```
tamaño :: ÁrbolBin a -> Integer
tamaño Vacío          = 0
tamaño (Nodo val ai ad) = 1 + (tamaño ai) + (tamaño ad)
```

La primera línea corresponde a la firma del tipo de la función. Se puede leer así: “Para todo tipo `a`, `tamaño` recibe un argumento de tipo `ÁrbolBin a` y devuelve un `Integer`. La función `tamaño` es polimórfica, pues funciona sobre árboles que contienen cualquier tipo de elementos. El código de la función consiste de dos líneas. La primera línea reconoce los árboles que son vacíos; su tamaño es 0. La segunda línea reconoce los árboles que no son vacíos; su tamaño es 1 más el tamaño de sus subárboles izquierdo y derecho.

Escribamos una función `buscar` para árboles binarios ordenados. El árbol contiene tuplas consistentes de una llave entera y un valor en forma de cadena de caracteres. Su tipo es `ÁrbolBin (Integer,String)`. La función `buscar` devuelve un valor de tipo `Maybe String`. Este valor será `Nothing` si la llave no existe en el árbol; y será `Just val` si `(k,val)` está en el árbol:

```
buscar :: Integer -> ÁrbolBin (Integer,String) -> Maybe String
buscar k Vacío = Nothing
buscar k (Nodo (nk,nv) ai ad) | k == nk = Just nv
buscar k (Nodo (nk,nv) ai ad) | k < nk  = buscar k ai
buscar k (Nodo (nk,nv) ai ad) | k > nk  = buscar k ad
```

A primera vista, la firma del tipo de `buscar` puede parecer extraña. ¿Por qué existe una `->` entre el tipo `Integer` y el tipo `ÁrbolBin (Integer,String)`? Esto es debido a la currificación. Cuando aplicamos `buscar` a una llave entera, conseguimos una función nueva, que busca esa llave en cada árbol binario al que se le aplique esa función.

4.7.5. Clases de tipos

Una desventaja de la definición anterior de `buscar` es que el tipo de la función es muy restrictivo. Nos gustaría que fuera polimórfico así como el tipo de la función `tamaño`. Entonces se podría usar el mismo código para buscar en árboles que contengan tuplas de casi cualquier tipo. Sin embargo, debemos restringir el primer argumento de la tupla a ser de un tipo que soporte las operaciones de comparación `==`, `<`, y `>` (e.g., no queremos las funciones como llaves, pues no existe un ordenamiento computable para funciones).

Para soportar esto, Haskell tiene clases de tipos. Una clase de tipos proporciona un nombre a un grupo de funciones. Si un tipo soporta estas funciones, entonces decimos que ese tipo es un miembro de esa clase de tipos. En Haskell hay una clase de tipos predefinida llamada `Ord` la cual soporta las operaciones `==`, `<`, y `>`. La firma de tipo siguiente especifica que el tipo de las llaves del árbol debe pertenecer a la clase de tipos `Ord`:

Concurrencia declarativa

```
buscar :: (Ord a) => a -> ÁrbolBin (a,b) -> Maybe b
```

y de hecho, este es el tipo que Haskell inferirá para `buscar`. Las clases de tipos permiten que los nombres de función sean *sobre cargados*. El operador `<` para `Integers` no es el mismo operador `<` de las cadenas de caracteres. Como un compilador de Haskell conoce los tipos de todas las expresiones, éste puede substituir la operación apropiada al tipo específico en cada uso. Las clases de tipos las soportan los lenguajes funcionales tales como Clean y Mercury. (Mercury es un lenguaje lógico con soporte para la programación funcional). Otros lenguajes, incluyendo Standard ML y Oz, logran un efecto similar de sobre carga usando functors.

Los programadores pueden agregar sus propios tipos a las clases de tipos. Por ejemplo, podríamos agregar el tipo `ÁrbolBin` a la clase de tipos `Ord` proveyendo definiciones apropiadas para los operadores de comparación. Si creamos un tipo para los números complejos, entonces lo podríamos hacer miembro de la clase de tipos numéricos `Num` proveyendo los operadores numéricos apropiados. La firma de tipo más general para la función `factorial` es

```
factorial :: (Num a, Ord a) => a -> a
```

Así, `factorial` podría ser aplicada a un argumento de cualquier tipo pero que soporte operaciones numéricas y de comparación, devolviendo un valor del mismo tipo.

4.8. Limitaciones y extensiones de la programación declarativa

La principal ventaja de la programación declarativa es que simplifica considerablemente la construcción de sistemas. Los componentes declarativos se pueden construir y depurar independientemente de los otros componentes. La complejidad de un sistema es la suma de las complejidades de sus componentes. Una pregunta natural para hacerse es, ¿cuán lejos puede ir la programación declarativa? ¿Todo puede ser programado en una forma declarativa, de manera que los programas sean a la vez naturales y eficientes? Esto sería una gran bendición para la construcción de sistemas. Decimos que un programa es *eficiente* si su desempeño difiere sólo en un factor constante del desempeño del programa en lenguaje ensamblador que resuelve el mismo problema.

Decimos que un programa es *natural* si requiere muy poco código para resolver problemas técnicos no relacionados con el problema original. Consideramos separadamente los asuntos correspondientes a eficiencia y naturalidad. Hay tres asuntos asociados a la naturalidad: modularidad, no-determinismo, e interfaces con el mundo real.

Recomendamos utilizar el modelo declarativo de este capítulo o la versión secundaria del capítulo 2, excepto cuando alguno de los asuntos arriba mencionados sea crítico. Esto facilita la escritura de componentes correctos y eficientes.

4.8.1. Eficiencia

¿Es eficiente la programación declarativa? Existe un desacuerdo fundamental entre el modelo declarativo y un computador estándar, tal como se presenta en [134]. El computador está optimizado para modificar datos en el sitio, mientras que el modelo declarativo nunca modifica los datos sino que siempre crea datos nuevos. Este no es un problema tan serio como parece a primera vista. El modelo declarativo puede tener un gran consumo de memoria inherente pero el tamaño de su memoria activa permanece pequeño. Queda como tarea, sin embargo, implementar el modelo declarativo con asignación en el sitio. Esto depende, primero que todo, en la sofisticación del compilador.

¿Puede un compilador traducir efectivamente programas declarativos en un computador estándar? Parafraseando al autor de ciencia ficción y futurólogo Arthur C. Clarke, podemos decir que “cualquier compilador suficientemente avanzado es indistinguible de la magia” [35].²³ Es decir, es irreal esperar que el compilador reescriba su programa. Aún después de varias décadas de investigación, no existe ningún compilador así para programación de propósito general. Lo más lejos que se ha llegado es a escribir compiladores que pueden reescribir el programa en casos particulares. El científico de la computación Paul Hudak los llama compiladores “smart-aleck”. Debido a sus impredecibles optimizaciones, son difíciles de usar. Por lo tanto, para el resto de la discusión, suponemos que el compilador realiza una traducción directa del código fuente en el código objetivo, en el sentido en que las complejidades en tiempo y espacio del código compilado se puede derivar de manera sencilla a partir de la semántica del lenguaje.

Ahora podemos responder a la pregunta de si la programación declarativa es eficiente. Suponiendo que contamos con un compilador directo, la respuesta pedante a la pregunta es no. Pero de hecho, en la práctica, la respuesta es sí, con una salvedad: la programación declarativa es eficiente si a uno se le permite reescribir el programa a uno menos natural. Tres ejemplos típicos son:

1. Un programa que realiza modificaciones incrementales de estructuras de datos grandes, e.g., una simulación que modifica grafos grandes (ver sección 6.8.4), en general no puede ser compilado eficientemente. Aún después de décadas de investigación, no existe un compilador directo que pueda tomar un tal programa e implementarlo eficientemente. Sin embargo, si a uno se le permite reescribir el programa, entonces hay un truco sencillo que, normalmente, es suficiente en la práctica. Si el estado está hilado al código (e.g., guardado en un acumulador) y el programa tiene cuidado de nunca acceder a un estado viejo, entonces el acumulador se puede implementar con asignación destructiva.
2. Una función que realiza memorización no se puede programar sin cambiar su interfaz. Suponga que tenemos una función que utiliza muchos recursos compu-

23. Tercera ley de Clarke: “Cualquier tecnología suficientemente avanzada es indistinguible de la magia.”

Concurrencia declarativa

tacionales. Para mejorar su desempeño, podríamos agregarle memorización, i.e., una memoria interna oculta con los resultados calculados previamente, indexada por los argumentos de la función. En cada invocación a la función, primero comprobamos si el resultado ya está calculado en esa memoria. Esta memoria interna no se puede agregar sin reescribir el programa, agregando un acumulador en todas los sitios donde la función se invoca. En la sección 10.4.2 (en CTM) se presenta un ejemplo.

3. Una función que implementa un algoritmo complejo, frecuentemente requiere un código complicado. Es decir, aunque el programa pueda ser escrito declarativamente con la misma eficiencia que en el modelo con estado, hacerlo resultará más complejo. Esto se debe a que el modelo declarativo es menos expresivo que el modelo con estado. En la sección 6.8.1 se muestra un ejemplo: un algoritmo de clausura transitiva escrito en ambos modelos, el declarativo y el modelo con estado. Ambas versiones tienen la misma eficiencia $O(n^3)$. El algoritmo con estado es más simple de escribir que el algoritmo declarativo.

Concluimos que la programación declarativa no puede ser, siempre, eficiente y natural a la vez. Miremos ahora los asuntos correspondientes a la eficiencia.

4.8.2. Modularidad

Decimos que un programa es modular con respecto a un cambio en una parte específica si el cambio se puede realizar sin cambiar el resto del programa. En general, la modularidad no se puede lograr con un modelo declarativo, pero se puede lograr si el modelo es extendido con estado explícito. Presentamos dos ejemplos en los que los programas declarativos no son modulares:

1. El primer ejemplo es la memorización en memoria oculta que vimos antes. Agregar esta memoria a una función no es modular, pues se hace necesario hilar un acumulador en muchos lugares por fuera de la función.
2. Un segundo ejemplo es la instrumentación de un programa. Nos gustaría conocer cuántas veces se invoca alguno de sus subcomponentes. Nos gustaría agregar contadores a esos subcomponentes, preferiblemente sin cambiar ni las interfaces de los subcomponentes ni el resto del programa. Si el programa es declarativo, esto es imposible, pues la única manera de hacerlo es hilando un acumulador a través del programa.

Miremos más de cerca el segundo ejemplo, para entender exactamente por qué es inadecuado el modelo declarativo. Suponga que estamos usando el modelo declarativo para implementar un gran componente declarativo. La definición del componente tiene el aspecto siguiente:

```

fun {SC ...}
  proc {P1 ...}
  ...
  end
  proc {P2 ...}
  ...
  {P1 ...}
  {P2 ...}
  end
  proc {P3 ...}
  ...
  {P2 ...}
  {P3 ...}
  end
in
  `export'(p1:P1 p2:P2 p3:P3)
end

```

Invocar SC crea una instancia del componente: devuelve un módulo con tres operaciones, P1, P2, y P3. Nos gustaría instrumentar el componente contando el número de veces que se invoca al procedimiento P1. Los valores consecutivos de la cuenta son un estado. Podemos codificar estos estados como un acumulador, i.e., agregando dos argumentos a cada procedimiento. Agregada esta instrumentación, el aspecto del componente sería el siguiente:

```

fun {SC ...}
  proc {P1 ... S1 ?Sn}
    Sn=S1+1
    ...
  end
  proc {P2 ... T1 ?Tn}
  ...
  {P1 ... T1 T2}
  {P2 ... T2 Tn}
  end
  proc {P3 ... U1 ?Un}
  ...
  {P2 ... U1 U2}
  {P3 ... U2 Un}
  end
in
  `export'(p1:P1 p2:P2 p3:P3)
end

```

A cada procedimiento definido en SC se le cambió su interfaz: se añaden dos argumentos adicionales, que juntos conforman un acumulador. El procedimiento P1 se invoca {P1 ... Sen Ssa1}, donde Sen es el contador cuando inicia la invocación y Ssa1 es el contador cuando termina. El acumulador tiene que ser pasado entre las invocaciones a procedimientos. Esta técnica requiere que tanto SC como el módulo que lo invoque realicen una buena cantidad de trabajo adicional, pero esto funciona.

Otra solución es escribir el componente en un modelo con estado. Tal modelo se define en el capítulo 6; por ahora suponga que contamos con una entidad

Concurrencia declarativa

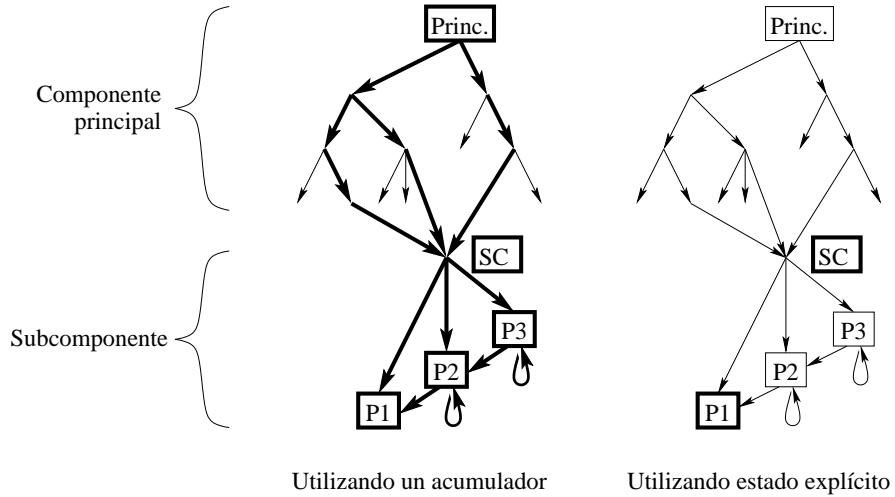


Figura 4.32: Cambios requeridos para la instrumentación del procedimiento P1.

nueva en el lenguaje, denominada “celda,” la cual podemos asignar y acceder (con los operadores `:=` y `@`), similar a una variable assignable de los lenguajes de programación imperativos. Las celdas se introdujeron en el capítulo 1. Entonces, la definición del componente tendría el aspecto siguiente:

```

fun {SC ...}
  Ctr={NewCell 0}
  proc {P1 ...}
    Ctr:=@Ctr+1
    ...
  end
  proc {P2 ...}
    ...
    {P1 ...}
    {P2 ...}
  end
  proc {P3 ...}
    ...
    {P2 ...}
    {P3 ...}
  end
  fun {Cont} @Ctr end
in
  `export'(p1:P1 p2:P2 p3:P3 cont:Cont)
end

```

En este caso, la interfaz del componente tiene una función adicional, `Cont`, y las interfaces de `P1`, `P2`, y `P3` quedan intactas. El módulo que invoca `SC` no tiene ningún trabajo adicional a realizar. El contador se inicia automáticamente en cero, apenas el componente se instancia. El módulo invocador puede invocar `Cont` en cualquier

momento para averiguar el valor actual del contador. El módulo invocador también puede ignorar completamente a `Cont`, si así lo desea, en cuyo caso el componente tendrá exactamente el mismo comportamiento que antes (salvo por una muy ligera diferencia en desempeño).

En la figura 4.32 se comparan los dos enfoques. La figura muestra el grafo de invocación de un programa con un componente `Princ` que invoca el subcomponente `SC`. Un grafo de invocación es un grafo dirigido donde cada nodo representa un procedimiento y hay una arista de cada procedimiento al procedimiento que él invoca. En la figura 4.32, `SC` se invoca de tres sitios en el componente principal. Ahora, instrumentemos `SC`. En el enfoque declarativo (a la izquierda), se debe agregar un acumulador a cada procedimiento en el camino de `Princ` a `P1`. En el modelo con estado (a la derecha), los únicos cambios son la operación adicional `Cont` y el cuerpo de `P1`. En ambos casos, los cambios se muestran con líneas gruesas. Comparemos los dos enfoques:

- El enfoque declarativo no es modular con respecto a la instrumentación de `P1`, pues cada definición de procedimiento e invocación en el camino de `Princ` a `P1` requiere dos argumentos adicionales. Las interfaces de `P1`, `P2`, y `P3` se cambiaron todas. Esto significa que otros componentes que invoquen a `SC` también deberán ser cambiados.
- El enfoque con estado es modular pues la celda sólo es mencionada donde se necesita, en la inicialización de `SC` y en `P1`. En particular, las interfaces de `P1`, `P2`, y `P3` permanecen iguales en el enfoque con estado. Como la operación adicional `Cont` puede ser ignorada, entonces otros componentes que invoquen a `SC` no tienen que ser cambiados.
- El enfoque declarativo es más lento pues realiza mucho trabajo adicional pasando argumentos. Todos los procedimientos se desaceleran por culpa de uno. El enfoque con estado es eficiente; sólo gasta tiempo cuando es necesario.

¿Cuál enfoque es más sencillo: el primero o el segundo? El primero tiene un modelo más sencillo pero el programa es más complejo. El segundo tiene un modelo más complejo pero un programa más sencillo. Desde nuestro punto de vista, el enfoque declarativo no es natural. El enfoque con estado, por ser modular, es claramente el más sencillo en conjunto.

La falacia del preprocesador

Tal vez exista una forma en que podamos tener nuestra torta y comerla. Definamos un preprocesador que agregue los argumentos de manera que no tengamos que escribirlos en ninguna parte. Un preprocesador es un programa que recibe otro código de programa fuente como entrada, lo transforma de acuerdo a unas reglas sencillas, y devuelve el resultado de esa transformación. Definimos entonces un preprocesador que recibe como entrada un programa con la sintaxis del enfoque con estado y lo transforma en un programa con el aspecto del enfoque declarativo.

Concurrencia declarativa

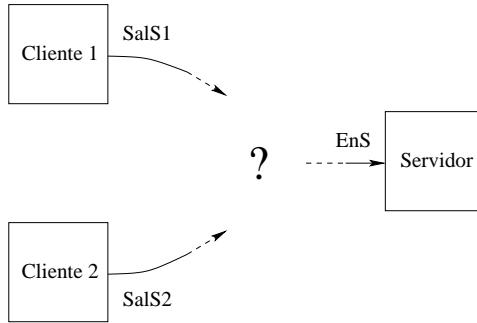


Figura 4.33: How can two clients send to the same server? They cannot!.

¡Voilà! Pareciera que ahora podemos programar con estado en el enfoque declarativo. Hemos superado una limitación del modelo declarativo. ¿Pero, lo hemos hecho? De hecho, no hemos logrado nada de ese tipo. Todo lo que hemos logrado hacer es construir una implementación ineficiente de un modelo con estado. Miremos por qué:

- Cuándo utilizamos el preprocesador, vemos solamente programas que parecen de la versión con estado, i.e., programas con estado. Esto nos obliga a razonar en el modelo con estado. Por tanto, hemos extendido de facto el modelo declarativo con estado explícito.
- El preprocesador transforma estos programas con estado en programas con estado hilado, los cuales son ineficientes debido a todo el trabajo de paso de argumentos.

4.8.3. No-determinismo

El modelo concurrente declarativo parece ser bastante poderoso para construir programas concurrentes. Por ejemplo, fácilmente podemos construir un simulador de circuitos digitales electrónicos. Sin embargo, a pesar de este poder aparente, el modelo tiene una limitación que lo deja cojo para muchas aplicaciones concurrentes: el modelo siempre se comporta determinísticamente. Si un programa tiene no-determinismo observable, entonces no es declarativo. Esta limitación está fuertemente relacionada con la modularidad: los componentes que son verdaderamente independientes se comportan de manera no determinística con respecto a los otros. Para mostrar que esta nos es una pura limitación teórica, presentamos dos ejemplos reales: una aplicación cliente/servidor y una aplicación de despliegue de video.

La limitación se puede eliminar agregando una operación no-determinística al modelo. El modelo extendido deja de ser declarativo. Hay muchas operaciones no determinísticas que podríamos agregar. En los capítulos 5 y 8 se explican las posibilidades en detalle. Aquí las trataremos brevemente:

- Una primera solución es agregar una operación de espera no determinística, tal como `WaitTwo` la cual espera a que una de las dos variables se ligue, e indica una de las ligadas. Su definición se presenta en el archivo de suplementos en el sitio Web del libro. `WaitTwo` es apropiada para la aplicación cliente/servidor.
- Una segunda solución consiste en agregar `IsDet`, una función booleana que comprueba inmediatamente si una variable de flujo de datos está ligada o no. Esto permite usar las variables de flujo de datos como una forma débil de estado. `IsDet` es apropiada para la aplicación de despliegue de video.
- Una tercera solución es agregar estado explícito al modelo, e.g., en la forma de puertos (canales de comunicación) o celdas (variables mutables).

¿Cómo se comparan las tres soluciones con respecto a la expresividad? `waitTwo` se puede programar en el modelo concurrente con estado explícito. Por tanto, parece que el modelo más expresivo sólo necesita estado explícito y la función `IsDet`.

4.8.3.1. Una aplicación cliente/servidor

Analicemos una aplicación sencilla cliente/servidor. Suponga que hay dos clientes independientes. Ser independientes implica que son concurrentes. ¿Qué pasa si ellos se comunican con el mismo servidor? Por ser independientes, el servidor puede recibir información de los dos clientes en cualquier orden. Este es un comportamiento no-determinístico observable.

Examinemos esto más de cerca y miremos por qué no se puede expresar en el modelo declarativo concurrente. El servidor tiene un flujo de entrada a partir del cual lee los mensajes (órdenes de servicio). Supongamos que un cliente envía una orden al servidor; esto funciona perfectamente. ¿Cómo se puede conectar un segundo cliente al servidor? El segundo cliente tiene que obtener una referencia al flujo, de manera que pueda ligarlo y ser leído por el servidor. ¡El problema es que tal flujo no existe! Sólo hay un flujo, entre el primer cliente y el servidor. El segundo cliente no puede ligar el flujo, pues entraría en conflicto con las ligaduras del primer cliente.

¿Cómo podemos resolver este problema? Aproximémonos a él ingenuamente y miremos si podemos encontrar una solución. Un enfoque podría ser que el servidor tuviera dos flujos de entrada, así:

```
fun {Servidor EnS1 EnS2}
  ...
end
```

¿Pero, cómo lee el servidor los flujos? ¿Lee primero un elemento de `EnS1` y luego un elemento de `EnS2`? ¿Lee simultáneamente un elemento de ambos flujos? Ninguna de estas soluciones es correcta. De hecho, no es posible escribir una solución en el modelo concurrente declarativo. Lo único que podemos hacer es tener dos servidores independientes, uno para cada cliente. Pero estos servidores no se pueden comunicar entre ellos, pues de ser así tendríamos el mismo problema de nuevo.

En la figura 4.33 se ilustra el problema: `Ens` es el flujo del servidor y `sals1` y

Concurrencia declarativa

Sals2 son los dos flujos de salida de los clientes. ¿Cómo se pueden hacer llegar al servidor los mensajes en los flujos de ambos clientes? ¡La respuesta sencilla es que en el modelo concurrente declarativo no se puede! En el modelo concurrente declarativo, un objeto activo tiene que conocer, siempre, de cuál flujo va a realizar la lectura siguiente.

¿Cómo resolver este problema? Si los clientes se ejecutan de forma coordinada, de manera que el servidor siempre sepa cuál cliente enviará la siguiente orden, entonces el programa es declarativo. Pero esto es irreal. Para escribir una verdadera solución, tenemos que agregar una operación no-determinística al modelo, como la operación WaitTwo mencionada atrás. Con WaitTwo, el servidor puede esperar una orden de cualquiera de los dos clientes. En el capítulo 5 se presenta una solución utilizando WaitTwo, en el modelo concurrente no-determinístico (ver sección 5.8).

4.8.3.2. Una aplicación de proyección de video

Demos una mirada a una aplicación sencilla de proyección de video. Esta consiste en una pantalla que recibe un flujo de imágenes y las proyecta. Las imágenes llegan a una velocidad particular, i.e., algún número de imágenes por segundo. Por diferentes razones, esta velocidad puede fluctuar: las imágenes tienen diferentes resoluciones, se puede realizar algún tipo de procesamiento sobre ellas, o el ancho de banda y la latencia de la red de transmisión es variable.

Debido a la velocidad variable de llegada, la pantalla no puede proyectar siempre todas las imágenes. Algunas veces tiene que saltarse unas imágenes. Por ejemplo, la pantalla puede tener que saltar rápidamente hasta la última imagen enviada. Este tipo de administración de flujos no se puede realizar en el modelo concurrente declarativo, pues no hay manera de detectar el fin de un flujo. Esto se puede hacer extendiendo el modelo con una operación nueva, IsDet. La comprobación booleana {IsDet xs} mira inmediatamente si xs ya está ligada o no (devolviendo **true** o **false**), y no espera si no lo está. Utilizando IsDet, podemos definir la función Saltar que recibe un flujo y devuelve su cola no ligada:

```
fun {Saltar xs}
  if {IsDet xs} then
    case xs of _|xr then {Saltar xr} [] nil then nil end
  else xs end
end
```

Este itera sobre el flujo hasta que encuentra una cola no-ligada. A continuación, presentamos una versión ligeramente diferente, que siempre espera hasta que exista al menos un elemento.

```
fun {Saltar1 xs}
  case xs of x|xr then
    if {IsDet xr} then {Saltar1 xr} else xs end
  [] nil then nil end
end
```

Con Saltar1 , podemos escribir un proyector de video que, después de proyectar su imagen, salte inmediatamente a la última imagen transmitida:

```
proc {Proyectar Xs}
  case {Saltar1 Xs}
    of X|Xr then
      {ProyectarMarco X}
      {Proyectar Xr}
    [] nil then skip
  end
end
```

Esto funcionará bien aun existiendo variaciones entre la velocidad de llegada de las imágenes y el tiempo que toma proyectar una imagen.

4.8.4. El mundo real

El mundo real no es declarativo, pues tiene tanto estado (las entidades tienen una memoria interna), como concurrencia (las entidades evolucionan independientemente).²⁴ Como los programas declarativos interactúan con el mundo real, directa o indirectamente, entonces hacen parte de un ambiente que contiene estos conceptos. Esto tiene dos consecuencias:

1. Problemas de interfaces: A los componentes declarativos les hace falta expresividad para comunicarse con los componentes no declarativos. Estos últimos están omnipresentes, e.g., los periféricos y las interfaces de usuario son inherentemente concurrentes y con estado (ver sección 3.8). Los sistemas operativos también hacen uso de la concurrencia y el estado para sus propósitos, por las razones mencionadas previamente. Uno puede pensar que estas propiedades no declarativas podrían ser enmascaradas o codificadas de alguna manera, pero esto nunca funciona. La realidad siempre está ahí, furtivamente.
2. Problemas de especificación: Debido a que las especificaciones de los programas se realizan para el mundo real, éstas frecuentemente mencionan estado y concurrencia. Si el programa es declarativo, entonces tienen que codificar esto de alguna manera. Por ejemplo, una especificación de una herramienta colaborativa puede requerir que los usuarios protejan con candados lo que ellos están trabajando para prevenir conflictos durante el acceso concurrente. En la implementación, los candados tienen que ser codificados de alguna manera. Utilizar los candados directamente en un modelo con estado lleva a una implementación más cercana a la especificación.

24. De hecho, el mundo real es paralelo, pero esto se modela dentro de un programa con la concurrencia. La concurrencia es un concepto del lenguaje que expresa computaciones lógicamente independientes. El paralelismo es un concepto de implementación que expresa actividades que ocurren simultáneamente. En un computador, el paralelismo solamente se utiliza para mejorar el desempeño.

4.8.5. Escogiendo el modelo correcto

Existen muchos modelos de computación que difieren en cuán expresivos son y en cuán difícil es razonar sobre los programas escritos en ellos. El modelo declarativo es uno de los más sencillos de todos. Sin embargo, como lo hemos explicado, tiene serias limitaciones para algunas aplicaciones. Hay modelos más expresivos que superan esas limitaciones; el costo es que algunas veces razonar sobre los programas escritos en esos modelos es más complicado. Por ejemplo, la concurrencia se necesita con frecuencia cuando se interactúa con el mundo exterior. Cuando tales interacciones son importantes, entonces se debe usar un modelo concurrente, en lugar de tratar de solucionarlo todo sólo con el modelo declarativo.

Los modelos más expresivos no son “mejores” que los otros, pues no siempre llevan a programas más sencillos, y razonar sobre ellos es normalmente más difícil.²⁵ De acuerdo a nuestra experiencia, todos los modelos tienen su lugar y se pueden usar juntos, de manera beneficiosa, en el mismo programa. Por ejemplo, en un programa con estado concurrente, muchos componentes pueden ser declarativos. A la inversa, en un programa declarativo, algunos componentes (e.g., algoritmos sobre grafos) necesitan estado para implementarse bien. Resumimos nuestra experiencia en la regla siguiente:

Regla de mínima expresividad

Cuando se programe un componente, el modelo de computación correcto es el menos expresivo que resulte en un programa natural.

La idea es que cada componente debería ser programado en su modelo “natural.” Utilizar un modelo menos expresivo nos llevaría a programas más complejos y utilizar un modelo más expresivo nos llevaría a un programa más sencillo, pero haciendo que sea más difícil razonar sobre él.

El problema con esta regla es que, realmente, no hemos definido “natural.” Esto porque en algún grado, la naturalidad es una propiedad subjetiva. Debido a sus diferentes conocimientos y formaciones, para unas personas un modelo es más fácil de usar, mientras que para otras es más fácil otro. El asunto no es la definición precisa de “natural,” sino el hecho de que tal definición exista para cada persona, aunque pueda ser diferente de la una a la otra. Entonces cada persona tiene la tarea de aplicar la regla de manera consistente.

4.8.6. Modelos extendidos

Ahora tenemos una idea de las limitaciones del modelo declarativo y una intuición inicial sobre cómo superar esas limitaciones con la ayuda de modelos extendidos con estado y concurrencia. Este es entonces un buen momento para presentar una breve

25. Otra razón por la que no son mejores tiene que ver con la programación distribuida y la conciencia de red, las cuales se explican en el capítulo 11 (en CTM).

mirada global a estos modelos, comenzando con el modelo declarativo:

- *Modelo secuencial declarativo* (ver capítulos 2 y 3). Este modelo abarca la programación funcional estricta y la programación lógica determinística. Extiende el primero con valores parciales (utilizando variables de flujo de datos, las cuales son llamadas también “variables lógicas”) y el último con procedimientos de alto orden. Razonar con este modelo se basa en cálculos algebráicos con valores. Las igualdades pueden ser substituidas por igualdades pudiéndose, entonces, aplicar las identidades alejorálicas. El comportamiento de un componente es independiente del momento en que se ejecuta o de lo que pasa en el resto de la computación.
- *Modelo concurrente declarativo* (en este capítulo; definido en las secciones 4.1 y 4.5). Este es el modelo declarativo extendido con hilos explícitos y computación by-need. Este modelo, siendo verdaderamente concurrente, conserva la mayoría de las buenas propiedades del modelo declarativo, e.g., razonar es casi tan sencillo. Este modelo puede hacer tanto concurrencia dirigida por los datos, como concurrencia dirigida por la demanda. Contiene la programación funcional perezosa y la programación lógica concurrente y determinística. Los componentes interactúan a través del uso y la ligadura de variables de flujo de datos compartidas.
- *Modelo declarativo con excepciones* (definido en las secciones 2.7.2 y 4.9.1). El modelo concurrente declarativo con excepciones deja de ser declarativo, pues se pueden escribir programas que exponen su no-determinismo (o sea con un no-determinismo observable).
- *Modelo concurrente por paso de mensajes* (ver capítulo 5). Este es el modelo declarativo extendido con comunicación por canales (puertos). Esto elimina la limitación del modelo declarativo concurrente de no poder implementar programas con algún no-determinismo, e.g., una aplicación cliente/servidor donde varios clientes hablan a un servidor. Esta es una generalización útil del modelo concurrente declarativo, fácil de programar con ella, y que permite restringir el no-determinismo a pequeñas partes del programa.
- *Modelo con estado* (ver capítulos 6 y 7; definido en la sección 6.3). Este es el modelo declarativo extendido con estado explícito. En este modelo se puede expresar la programación secuencial orientada a objetos tal como es conocida normalmente. Un estado es una secuencia de valores que se extiende a medida que la computación avanza. Tener estado explícito significa que un componente no da siempre el mismo resultado cuando es invocado con los mismos argumentos. El componente puede “recordar” información de una invocación a otra. Esto permite al componente llevar una “historia,” la cual lo deja interactuar con su ambiente de manera significativa, adaptándose y aprendiendo de su pasado. Razonar con este modelo requiere razonar sobre la historia.
- *Modelo concurrente con estado compartido* (ver capítulo 8; definido en la sección 8.1). Este es el modelo declarativo extendido con estado explícito y con hilos. Este modelo contiene la programación concurrente orientada a objetos. La concurrencia es más expresiva que en el modelo concurrente declarativo pues puede

Concurrencia declarativa

usar el estado explícito para esperar simultáneamente la ocurrencia de uno o varios eventos (esto se llama escogencia no-determinística). Razonar con este modelo es muy complejo pues pueden existir múltiples historias interactuando en formas impredecibles.

- *Modelo relacional* (ver capítulo 9 (en CTM); definido en la sección 9.1 (en CTM)). Este es el modelo declarativo extendido con búsqueda (lo cual se llama algunas veces “no-determinismo no sé,”²⁶ aunque el algoritmo de búsqueda es casi siempre determinístico). En el programa, la búsqueda se expresa como una secuencia de escogencias. El espacio de búsqueda se explora haciendo diferentes escogencias hasta que el resultado sea satisfactorio. Este modelo permite la programación con relaciones, abarcando la programación lógica no-determinística en el estilo de Prolog. Este modelo es un precursor de la programación con restricciones, la cual se introduce en el capítulo 12 (en CTM).

Más adelante, dedicamos capítulos completos a cada uno de estos modelos para explicar para qué son buenos, cómo programar en ellos, y cómo razonar con ellos.

4.8.7. Usando diferentes modelos a la vez

Típicamente, cualquier programa bien escrito, de tamaño razonable, tiene diferentes partes escritas en diferentes modelos. Hay muchas maneras de usar diferentes modelos a la vez. En esta sección se presenta una técnica, particularmente útil, la cual llamamos adaptación de impedancias, que lleva, de una manera natural, a utilizar diferentes modelos a la vez en el mismo programa.

La adaptación de impedancias es una de las maneras, más poderosas y prácticas, de implementar el principio general de separación de asuntos. Considere dos modelos de computación, **Grande** y **Pequeño**, tal que el modelo **Grande** es más expresivo que el modelo **Pequeño**, pero es más fácil de razonar en el segundo que en el primero. Por ejemplo, el modelo **Grande** podría ser con estado y el modelo **Pequeño** ser el declarativo. Con la adaptación de impedancias, podemos escribir un programa en el modelo **Pequeño** que pueda vivir en el ambiente computacional del modelo **Grande**.

La adaptación de impedancias funciona construyendo una abstracción en el modelo **Grande** parametrizada por un programa en el modelo **Pequeño**. El corazón de esta técnica está en encontrar e implementar la abstracción correcta. El trabajo difícil sólo se realiza una vez; después sólo queda el trabajo fácil de usar la abstracción. Tal vez parezca sorprendente, pero resulta que casi siempre es posible encontrar en implementar la abstracción apropiada. Algunos ejemplos típicos de adaptación de impedancias son:

- Utilizar un componente secuencial en un modelo concurrente. Por ejemplo, la abstracción puede ser un serializador que acepta solicitudes concurrentes, las vuelve secuenciales, y devuelve correctamente las respuestas (rélicas). En la figura 4.34

26. Nota del traductor: del inglés, “don’t know nondeterminism.”

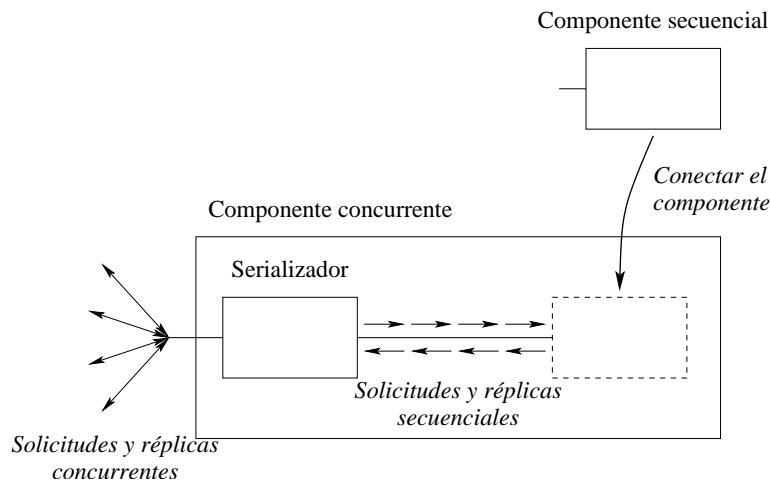


Figura 4.34: Adaptación de impedancias: ejemplo de un serializador.

se ilustra el ejemplo. Conectar un componente secuencial en el serializador produce un componente concurrente.

- Utilizar un componente declarativo en un modelo con estado. Por ejemplo, la abstracción puede ser un administrador de almacenamiento que pasa su contenido al programa declarativo y almacena el resultado como su nuevo contenido.
- Utilizar un componente centralizado en un modelo distribuido. Un modelo distribuido se ejecuta sobre dos o más procesos del sistema operativo. Por ejemplo, la abstracción puede ser un recolector que acepta solicitudes de cualquier sitio y las pasa a un solo sitio.
- Utilizar un componente realizado para un modelo seguro, en un modelo inseguro. Un modelo inseguro es aquel que supone la existencia de entidades maliciosas que pueden perturbar los programas en formas bien definidas. Un modelo seguro supone que tales entidades no existen. La abstracción puede ser un protector que aisla una computación verificando todas las solicitudes de otras computaciones. La abstracción maneja todos los detalles que tienen que ver enfrentar la presencia de adversarios maliciosos. Un cortafuegos es un ejemplo de protector.
- Utilizar un componente realizado para un modelo cerrado en un modelo abierto. Un modelo abierto es aquel que permite que las computaciones independientes se encuentren con otras e interactúen. Un modelo cerrado supone que todas las computaciones se conocen desde el principio. La abstracción puede ser un conector que acepta solicitudes de una computación para conectarse a otra, utilizando un esquema abierto de direccionamiento.
- Utilizar un componente que suponga un modelo confiable en un modelo con falla parcial. Una falla parcial ocurre en un sistema distribuido cuando falla parte del sistema. Por ejemplo, la abstracción puede ser un replicador que implementa la tole-

Concurrencia declarativa

rancia a fallos por medio de replicación activa entre diversos sitios y administrando la recuperación cuando uno de ellos falla.

Estos casos son ortogonales. Como lo muestran los ejemplos, frecuentemente es una buena idea implementar varios casos en una abstracción. Este libro presenta abstracciones que ilustran todos estos casos y más. Normalmente, la abstracción coloca una condición menor sobre el programa escrito en el modelo Pequeño. Por ejemplo, un replicador muchas veces supone que la función que está replicando es determinística.

La adaptación de impedancias se usa extensivamente en el proyecto en Ericsson [9]. Una abstracción típica en Erlang toma un programa declarativo escrito en un lenguaje funcional y lo vuelve con estado, concurrente, y tolerante a fallos.

4.9. Temas avanzados

4.9.1. El modelo concurrente declarativo con excepciones

En la sección 2.7 agregamos excepciones a la programación secuencial declarativa. Miremos ahora qué pasa cuando agregamos excepciones a la programación concurrente declarativa. Primero explicamos cómo interactúan las excepciones con la concurrencia. Luego explicamos cómo interactúan las excepciones con la computación by-need.

Excepciones y concurrencia

Hasta el momento hemos ignorado las excepciones en la programación concurrente declarativa. Hay una razón muy sencilla para ello: si un componente lanza una excepción en el modelo concurrente declarativo, entonces ¡el modelo deja de ser declarativo! Agreguemos excepciones al modelo concurrente declarativo y miremos qué pasa. Para el modelo dirigido por los datos, el lenguaje núcleo resultante se presenta en la tabla 4.4. Esta tabla contiene las instrucciones **thread** y **ByNeed**, las declaraciones **try** y **raise**, y también una operación nueva, **FailedValue**, la cual maneja la interacción entre las excepciones y la computación by-need. Primero explicamos la interacción entre la concurrencia y las excepciones; dejamos **FailedValue** para la sección siguiente.

Investiguemos cómo es que las excepciones vuelven el modelo no declarativo. Hay básicamente dos maneras. La primera es que por ser declarativo un componente tiene que ser determinístico. Si las declaraciones $x=1$ y $x=2$ se ejecutan concurrentemente, entonces la ejecución deja de ser determinística: una de ellas tendrá éxito y la otra lanzará una excepción. En el almacén, x será ligada a 1 o a 2; ambos casos son posibles. Este es un caso claro de no-determinismo observable. La excepción es un testigo de esto; ella es lanzada por una falla de unificación, lo cual significa que potencialmente existe un no-determinismo observable. La excepción no es una

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Invocación de procedimiento
thread $\langle d \rangle$ end	Creación de hilo
$\{ \text{ByNeed} \langle x \rangle \langle y \rangle \}$	Creación de disparador
try $\langle d \rangle_1$ catch $\langle x \rangle$ then $\langle d \rangle_2$ end	Contexto de la excepción
raise $\langle x \rangle$ end	Lanzamiento de excepción
$\{ \text{FailedValue} \langle x \rangle \langle y \rangle \}$	Valor fallido

Tabla 4.4: El lenguaje núcleo concurrente declarativo con excepciones.

garantía de esto; e.g., ejecutar $x=1$ y $x=2$ en orden en el mismo hilo lanzará una excepción, aunque x quede ligada siempre a 1. Pero si no hay excepciones, entonces la ejecución es, seguramente, determinística y por tanto declarativa.

Una segunda forma en que se puede lanzar una excepción es cuando una operación no puede completarse normalmente. Esto puede ser debido a razones internas, e.g., los argumentos están fuera del dominio de la operación (tal como dividir por cero), o por razones externas, e.g., el ambiente externo tiene un problema (tal como tratar de abrir un archivo que no existe). En ambos casos, la excepción indica que se trató de realizar una operación violando su especificación. Cuando esto suceda, se acabarán todas las apuestas, por así decir. Desde el punto de vista de la semántica, no hay garantía sobre cuál operación fue realizada; pudo haber sucedido cualquier cosa. De nuevo, la operación se ha vuelto potencialmente no-determinística.

Resumiendo, cuando una excepción se lanza, esto indica una ejecución no-determinística o una ejecución que viola una especificación. En cualquier caso, el componente deja de ser declarativo. Decimos que el modelo concurrente declarativo es declarativo módulo las excepciones. Resulta que el modelo concurrente declarativo con excepciones es similar al modelo concurrente con estado compartido del capítulo 8. Esto se explica en la sección 8.1.

¿Entonces qué hacemos cuando ocurre una excepción? ¿Estamos completamente impotentes para escribir programas declarativos? No del todo. En algunos casos, el componente puede “arreglar las cosas” de manera que sea aún declarativo cuando es visto desde afuera. El problema básico es hacer el componente determinístico. Todas las fuentes de no-determinismo tienen que estar ocultas cuando son vistas desde afuera. Por ejemplo, si un componente ejecuta $x=1$ y $x=2$ concurrentemente, entonces lo mínimo que hay que hacer es (1) capturar la excepción colocando una

Concurrencia declarativa

declaración **try** alrededor de cada ligadura, y (2) encapsular **x** de manera que su valor no sea observable desde afuera. Ver el ejemplo de confinamiento de la falla en la sección 4.1.4.

Excepciones y computación by-need

En la sección 2.7, agregamos excepciones al modelo declarativo como una forma de manejar condiciones anormales sin llenar el código con líneas de comprobaciones de errores. Si una ligadura falla, se lanza una excepción, la cual puede ser capturada y manejada por otra parte de la aplicación.

Miremos cómo podemos extender esta idea a la computación by-need. ¿Qué pasa si la ejecución de un disparador by-need no se puede completar normalmente? En ese caso no se calcula un valor. Por ejemplo:

```
x={ByNeed fun {$} A=foo(1) B=foo(2) in A=B A end}
```

¿Qué debería pasar si un hilo necesita **x**? Al disparar el cálculo se producirá una falla cuando se ejecute la ligadura **A=B**. Es claro que **x** no se puede ligar a un valor, pues la computación by-need no se pudo completar. Por otro lado, no se puede simplemente dejar **x** sin ligar pues el hilo que necesita **x** está esperando un valor. La solución correcta para ese hilo es lanzar una excepción. Para asegurarse de ello, podemos ligar **x** a un valor especial llamado un valor fallido. Cualquier hilo que necesite un valor fallido lanzará una excepción.

Extendemos el lenguaje núcleo con la operación **FailedValue**, la cual crea un valor fallido:

```
x={FailedValue cannotCalculate}
```

Su definición se presenta en el archivo de suplementos en el sitio Web del libro. **FailedValue** crea un valor fallido que encapsula la excepción **nosepuedeCalcular**. Cualquier hilo que trate de usar **x** lanzará la excepción **nosepuedeCalcular**. Cualquier valor parcial puede ser encapsulado dentro del valor fallido.

Con **FailedValue** podemos definir una versión “robusta” de **ByNeed** que cree automáticamente un valor fallido cuando una computación by-need lanza una excepción:

```
proc {ByNeed2 P x}
  {ByNeed
    proc {$ x}
      try Y in {P Y} X=Y
      catch E then X={FailedValue E} end
      end X
    end
  end
```

ByNeed2 se invoca de la misma manera que **ByNeed**. Si hay alguna posibilidad de que la computación by-need vaya a lanzar una excepción, entonces **ByNeed2** encapsulará la excepción en un valor fallido.

En la tabla 4.4 se presenta el lenguaje núcleo para el modelo concurrente

declarativo completo, incluyendo tanto computación by-need como excepciones. El lenguaje núcleo contiene las operaciones `ByNeed` y `FailedValue` así como las declaraciones `try` y `raise`. La operación `{FailedValue <x> <y>}` encapsula la excepción `<x>` en el valor fallido `<y>`. Cada vez que un hilo necesite `<y>`, la declaración `raise <x> end` se ejecuta en el hilo.

La implementación del enlace dinámico es un ejemplo importante de uso de valores fallidos. Recuerde que la computación by-need se usa para cargar y enlazar módulos a medida que se necesitan. Si el módulo no se puede encontrar, entonces la referencia al módulo se liga a un valor fallido. Entonces, cuando un hilo trata de usar el módulo inexistente, se lanza una excepción.

4.9.2. Más sobre ejecución perezosa

Existe una vasta literatura sobre ejecución perezosa. En la sección 4.5 sólo se tocó la punta del iceberg. Continuamos ahora la discusión sobre ejecución perezosa. Planteamos dos temas:

- *Asuntos de diseño de lenguajes.* Cuando se diseña un nuevo lenguaje, ¿cuál es el rol de la pereza? Discutiremos brevemente los asuntos relacionados con esto.
- *Orden de reducción y paralelismo.* Los lenguajes de programación funcional modernos, como se exemplificó con Haskell, usan con frecuencia una variante de la pereza denominada evaluación flexible. Presentamos un vistazo breve de este concepto y analizamos por qué es útil.

Asuntos de diseño de lenguajes

¿Un lenguaje declarativo debe ser perezoso o ansioso o contar con las dos opciones? Esto hace parte de una pregunta más amplia: ¿Un lenguaje declarativo debe ser un subconjunto de un lenguaje extendido, no declarativo? Limitar un lenguaje a un modelo de computación permite optimizar la sintaxis y la semántica para ese modelo. Esto lleva a código sorprendentemente conciso y legible, para los programas que “encajan bien” con el modelo. Haskell y Prolog son ejemplos particularmente notables de este enfoque de diseño de lenguajes [20, 163]. Haskell utiliza evaluación perezosa todo el tiempo y Prolog utiliza resolución de cláusulas de Horn todo el tiempo. Vea las secciones 4.7 y 9.7 (en CTM), respectivamente, para mayor información sobre estos dos lenguajes. FP, un lenguaje funcional tempranero, influyente, llevó esto al extremo con un conjunto especial de caracteres, lo cual redujo, pardójicamente, la legibilidad [12]. Sin embargo, como lo vimos en la sección 4.8, muchos programas requieren más de un modelo de computación. Esto también es cierto para ejecución ansiosa versus ejecución perezosa. Miremos por qué:

- Para la programación en pequeño, e.g., el diseño de algoritmos, el ser ansioso es importante si la complejidad es un asunto que nos interesa. La evaluación ansiosa facilita el diseño y el razonamiento sobre algoritmos con una complejidad deseada

Concurrencia declarativa

en el peor caso. La evaluación perezosa dificulta esto; aún los expertos se confunden. Por otro lado, la pereza es importante en el diseño de algoritmos persistentes, i.e., que pueden tener múltiples versiones coexistiendo. En la sección 4.5.8 se explica por qué esto es así y se presenta un ejemplo. Creemos que un buen enfoque es usar evaluación ansiosa por defecto y colocar la evaluación perezosa, explícitamente, exactamente donde se necesita. Esto es lo que hace Okasaki con una versión del lenguaje funcional ansioso Standard ML, extendido con pereza explícita [129].

- Para la programación en grande, ambos tipos de evaluaciones, la ansiosa y la perezosa, juegan un papel importante cuando se diseñan las interfaces de los componentes. Por ejemplo, considere un canal de comunicación entre un componente consumidor y otro componente productor. Hay dos maneras básicas de controlar esta ejecución: o el productor decide cuando calcular nuevos elementos (estilo “push”) o el consumidor solicita elementos a medida que los necesita (estilo “pull”). Un estilo “push” implica una ejecución ansiosa y un estilo “pull” implica una ejecución perezosa. Ambos estilos pueden ser útiles. Por ejemplo, una memoria intermedia limitada obliga un estilo “push” cuando no está llena y un estilo “pull” cuando está llena.

Concluimos que un lenguaje declarativo orientado a la programación de propósito general debe soportar tanto ejecución ansiosa como perezosa, con la primera por defecto y la segunda disponible a través de una declaración. Si no se cuenta con una de las dos, la otra siempre podrá ser codificada pero esto lleva a programas innecesariamente complejos.

Orden de reducción y paralelismo

Vimos que la evaluación perezosa evaluará los argumentos de una función solamente cuando estos sean necesitados. Técnicamente hablando, esto se denomina orden normal de reducción. Cuando se ejecuta un programa declarativo, el orden de reducción normal siempre escoge reducir primero la expresión más a la izquierda. Después de realizar una etapa de reducción, se escoge nuevamente la expresión más a la izquierda, y así sucesivamente. Miremos un ejemplo para ver cómo funciona esto. Considere la función F1 definida de la siguiente manera:

```
fun {F1 A B}
  if B then A else 0 end
end
```

Evaluemos la expresión $\{F1 \{F2 x\} \{F3 y\}\}$. La primera etapa de reducción aplica F1 a sus argumentos. Esto sustituye los argumentos en el cuerpo de F1, dando como resultado `if {F3 y} then {F2 x} else 0 end`. La segunda etapa empieza la evaluación de F3. Si esta devuelve `false`, entonces F2 no se evalúa nunca. Podemos ver, intuitivamente, que el orden de reducción normal evalúa las expresiones cuando son necesitadas.

Hay muchos órdenes posibles de reducción, pues en cada etapa de ejecución se debe decidir por cuál función continuar la siguiente reducción. Con la concurrencia

declarativa, muchos de estos órdenes pueden aparecer durante la ejecución. Esto no hace diferencia en el resultado de la evaluación: decimos que hay un no-determinismo no observable.

Además del orden de reducción normal, hay otro orden interesante denominado orden de reducción aplicativo. Este siempre evalúa los argumentos de la función antes de evaluar la función. Esto es lo mismo que evaluación ansiosa. En la expresión $\{F1 \{F2 x\} \{F3 y\}\}$, se evalúan tanto $\{F2 x\}$ como $\{F3 y\}$ antes de evaluar $F1$. Con el orden de reducción aplicativo, si $\{F2 x\}$ o $\{F3 y\}$ caen en un ciclo infinito, entonces la computación completa caerá en un ciclo infinito. Esto sucederá así los resultados de $\{F2 x\}$ o $\{F3 y\}$ no lleguen a ser necesitados por el resto de la computación. Decimos entonces que el orden de reducción aplicativo es estricto o rígido.

Para todo programa declarativo, podemos probar que todo orden de reducción que termina produce el mismo resultado. Este resultado es una consecuencia del teorema de Church-Rosser, el cual muestra que la reducción en el cálculo λ es confluente, i.e., las reducciones que comienzan a partir de la misma expresión y siguen diferentes caminos siempre se pueden encontrar de nuevo más adelante. Podemos decir esto de otra manera: cambiar el orden de reducción sólo afecta el hecho de que un programa termine o no, pero no cambia su resultado (cuando termina). Podemos probar también que con el orden de reducción normal se logra el menor número de etapas de reducción comparado con cualquier otro orden de reducción.

Evaluación flexible Un lenguaje de programación funcional cuyo modelo de computación termina, siempre que el orden de reducción normal termina, se denomina un lenguaje flexible. Mencionamos la evaluación flexible porque ella se usa en Haskell, un lenguaje funcional popular. La diferencia entre evaluación flexible y perezosa es sutil. Un lenguaje perezoso realiza el mínimo número de etapas de reducción. Un lenguaje flexible podría realizar más etapas, pero aún garantiza que termina en aquellos casos en que la evaluación perezosa termina. Para ver mejor la diferencia entre evaluación perezosa y flexible, considere el ejemplo siguiente:

```
local x={F 4} in x+x end
```

En un lenguaje flexible $\{F 4\}$ puede ser computada dos veces. En un lenguaje perezoso $\{F 4\}$ será computada exáctamente una sola vez, cuando x sea necesario por primera vez; luego, el resultado será reutilizado cada que se necesite x . Un lenguaje perezoso es flexible, pero no siempre es cierto a la inversa.

La diferencia entre la evaluación perezosa y la flexible se vuelve importante en un procesador paralelo. Por ejemplo, durante la ejecución de $\{F1 \{F2 x\} \{F3 y\}\}$ se podría empezar ejecutando $\{F2 x\}$ en un procesador disponible, aún antes de saber si se necesitará realmente o no. Esto se denomina ejecución especulativa. Si más tarde se encuentra que $\{F2 x\}$ se necesita, entonces ya hemos comenzado su ejecución. Si, por el contrario, $\{F2 x\}$ no se necesita, entonces se aborta la ejecución tan pronto como se pueda. Esto significa un trabajo adicional innecesario, pero como

Concurrencia declarativa

	Asincrónica	Sincrónica
Remisión	ligar una variable	esperar hasta que la variable sea necesitada
Recepción	utiliza la variable inmediatamente	esperar hasta que la variable sea ligada

Tabla 4.5: Variables de flujo de datos como canales de comunicación.

se realiza en otro procesador no causará una desaceleración. Un lenguaje flexible se puede implementar con ejecución especulativa.

La flexibilidad es problemática cuando deseamos extender un lenguaje con estado explícito (como lo veremos en el capítulo 6). Un lenguaje flexible es difícil de extender con estado explícito porque la flexibilidad introduce una impredecibilidad fundamental en la ejecución del lenguaje. Nunca podemos estar seguro de cuántas veces se evalúa una función. En un modelo declarativo esto no es un problema serio pues el resultado de la computación no cambia. Se vuelve un problema serio cuando añadimos estado explícito. Las funciones con estado explícito pueden tener resultados impredecibles. La evaluación perezosa tiene el mismo problema pero en un grado menor: el orden de evaluación es dependiente de los datos, pero al menos sabemos que la función será evaluada a lo sumo una vez. La solución usada en el modelo concurrente declarativo es hacer que la evaluación ansiosa se realice por defecto y que la evaluación perezosa requiera una declaración explícita. La solución usada en Haskell es más complicada: evitar estado explícito y en su lugar utilizar un tipo de acumulador denominado un mónada. El enfoque monádico utiliza la programación de alto orden para hilar el estado implícitamente. Los argumentos adicionales son parte de las entradas y salidas de la función; ellos son hilados definiendo un nuevo operador de composición de funciones.

4.9.3. Variables de flujo de datos como canales de comunicación

En el modelo concurrente declarativo, los hilos se comunican a través de variables de flujo de datos compartidas. Existe una correspondencia estrecha entre variables de flujo de datos y operaciones sobre un canal de comunicación. Consideramos una variable de flujo de datos como una especie de canal de comunicación y un hilo como una especie de objeto. Entonces, ligar una variable es un especie de operación de remisión por el canal, y esperar hasta que una variable de flujo de datos sea ligada es una especie de operación de recepción. El canal tiene la propiedad que solamente se puede enviar un mensaje por él, pero el mensaje se puede recibir muchas veces. Investiguemos esta analogía un poco más.

Considere una comunicación que involucra dos partes, un remitente y un receptor. Como parte del acto de comunicación, cada parte realiza una operación: una remisión o una recepción. Una operación asincrónica se completa inmediatamente, sin ninguna interacción con la otra parte. Una operación sincrónica continúa sólo después que se produce una interacción exitosa con la otra parte. Sobre un canal

de comunicación, las operaciones de remisión y recepción pueden ser, cada una, sincrónica o asincrónica. Esto produce cuatro posibilidades en total. ¿Podemos expresar estas posibilidades con variables de flujo de datos? Dos de las posibilidades son directas pues corresponden al uso estándar en la ejecución de flujo de datos:

- Ligar una variable corresponde a una remisión asincrónica. La ligadura se puede realizar independientemente de si los hilos han recibido el mensaje.
- Esperar hasta que una variable sea ligada corresponde a una recepción sincrónica. La ligadura debe existir para que el hilo pueda continuar la ejecución.

¿Qué se puede decir sobre una recepción asincrónica y una remisión sincrónica? De hecho, ambas son posibles:

- Una recepción asincrónica significa sencillamente utilizar una variable antes de que sea ligada. Por supuesto, cualquier operación que necesite el valor de la variable tendrá que esperar hasta que el valor llegue.
- Una remisión sincrónica significa esperar con la ligadura hasta que el valor de la variable sea recibido. Consideramos que un valor es recibido si es necesario por alguna operación. Entonces la remisión sincrónica se puede implementar con disparadores by-need:

```
proc {RemSinc X M}
Sinc in
  {ByNeed proc {$ _} X=M Sinc=listo end X}
  {Wait Sinc}
end
```

Invocar {RemSinc X M} envía M por el canal X y espera hasta que éste haya sido recibido.

En la tabla 4.5 se resumen las cuatro posibilidades.

Los canales de comunicación cuentan algunas veces con operaciones no bloqueantes de remisión y recepción. Esto no es lo mismo que operaciones asincrónicas. La característica que define una operación no bloqueante es que devuelve inmediatamente un resultado booleano diciendo si la operación fue exitosa o no. Con variables de flujo de datos, una operación de remisión es trivialmente no bloqueante pues siempre tiene éxito. En cambio, una operación no bloqueante de recepción es más interesante. Ésta consiste en verificar si la variable está ligada o no, y devolver **true** o **false** de acuerdo a ello. Esto se puede implementar con la función **IsDet**. {**IsDet** X} devuelve inmediatamente **true** si X está ligada y **false** si no lo está. Para ser precisos, **IsDet** devuelve **true** si X está determinada, i.e., ligada a un número, registro, o procedimiento. No hay necesidad de decirlo, **IsDet** no es una operación declarativa.

4.9.4. Más sobre sincronización

Hemos visto que los hilos se pueden comunicar a través de variables compartidas de flujo de datos. Cuando un hilo necesita el resultado de un cálculo realizado por

	Dirigido por la oferta	Dirigido por la demanda
Implícita	ejecución por flujo de datos	ejecución perezosa
Explícita	candados, monitores, etc.	disparador programado

Tabla 4.6: Clasificación de la sincronización.

otro hilo, entonces espera hasta que este resultado esté disponible. Decimos que hay sincronización sobre la disponibilidad del resultado. La sincronización es uno de los conceptos fundamentales en programación concurrente. Ahora, investiguemos este concepto un poco más.

Primero definiremos precisamente el concepto básico de punto de sincronización. Considere los hilos T1 y T2, cada uno ejecutando una secuencia de etapas de computación. T1 hace $\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots$ y T2 hace $\beta_0 \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow \dots$.

Realmente, los hilos se ejecutan juntos en una computación global. Esto significa que existe una secuencia global de etapas de computación que contiene las etapas de cada hilo, intercaladas: $\alpha_0 \rightarrow \beta_0 \rightarrow \beta_1 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots$. Hay muchas formas en que dos computaciones se pueden intercalar. Pero, hay intercalaciones que no pueden ocurrir en una computación real:

- Debido a la imparcialidad, no es posible que se produzca una secuencia infinita de etapas α sin algunas etapas β . La imparcialidad es una propiedad global que el sistema hace cumplir.
- Si los hilos dependen de los resultados de los otros de alguna manera, entonces existen restricciones adicionales denominadas puntos de sincronización. Un punto de sincronización relaciona dos etapas de computación β_i y α_j . Decimos que β_i se sincroniza sobre α_j si en cada intercalación que puede ocurrir en una computación real, β_i ocurre *después* de α_j . La sincronización es una propiedad local que la hacen cumplir las operaciones que ocurren en los hilos.

¿Cómo especifica un programa cuándo realizar la sincronización? Hay dos enfoques amplios:

- *Sincronización implícita.* En este enfoque, las operaciones de sincronización no son visibles en el texto del programa; ellas hacen parte de la semántica operacional del lenguaje. Por ejemplo, usar una variable de flujo de datos se sincronizará sobre el momento en que la variable sea ligada a un valor.
- *Sincronización explícita.* En este enfoque, las operaciones de sincronización son visibles en el texto del programa; ellas son operaciones explícitas colocadas allí por el programador. Por ejemplo, en la sección 4.3.3 se muestra un productor/consumidor dirigido por la demanda que utiliza un disparador programado. Más adelante en el libro, veremos otras formas de realizar la sincronización explícita, e.g., utilizando candados o monitores (ver capítulo 8).

Existen dos orientaciones de la sincronización:

- *Sincronización dirigida por la oferta (ejecución ansiosa)*. Tratar de ejecutar la operación hace que la operación espere hasta que sus argumentos estén disponibles. En otras palabras, la operación se sincroniza sobre la disponibilidad de los argumentos. Esta espera no tiene ningún efecto sobre el cálculo o no de los argumentos; si algún hilo no los calcula, entonces la operación esperará indefinidamente.
- *Sincronización dirigida por la demanda (ejecución perezosa)*. Tratar de ejecutar la operación dispara el cálculo de los argumentos. En otras palabras, el cálculo de los argumentos se sincroniza sobre la operación que los necesita.

En la tabla 4.6 se muestran las cuatro posibilidades que existen. Las cuatro son prácticas y existen en sistemas reales. La sincronización explícita es el mecanismo primario en la mayoría de los lenguajes que están basados en un modelo con estado, e.g., Java, Smalltalk, y C++. Este mecanismo se explica en el capítulo 8. La sincronización implícita es el mecanismo primario en la mayoría de los lenguajes que están basados en un modelo declarativo, e.g., lenguajes funcionales tales como Haskell que usa evaluación perezosa, y lenguajes lógicos tales como Prolog, o lenguajes lógicos concurrentes, los cuales usan ejecución por flujo de datos. Este mecnismo se presenta en este capítulo.

Estas cuatro posibilidades se pueden usar eficientemente en los modelos de computación del libro. Esto nos permite comparar su expresividad y facilidad de uso. En nuestra opinión, la programación concurrente es más sencilla con sincronización implícita que con sincronización explícita. En particular, encontramos que la programación con ejecución por flujo de datos hace que los programas concurrentes sean más sencillos. Aún en un modelo con estado, como el del capítulo 8, la ejecución por flujo de datos es ventajosa. Después de comparar lenguajes con sincronización implícita y explícita, Bal, Steiner, y Tanenbaum llegaron a la misma conclusión: las variables de flujo de datos son “espectacularmente expresivas” en la programación concurrente, comparadas con la sincronización explícita, aún sin estado explícito [14]. Esta expresividad es una de las razones por las cuales enfatizamos en la sincronización implícita en el libro. Ahora, examinemos un poco mejor la utilidad de la ejecución por flujo de datos.

4.9.5. Utilidad de las variables de flujo de datos

En la sección 4.2.3 se muestra cómo se usa la ejecución por flujo de datos para la sincronización en el modelo declarativo concurrente. Hay muchos otros usos de la ejecución por flujo de datos. En esta sección se resumen estos usos. Presentamos referencias a ejemplos a lo largo del libro para ilustrarlos. La ejecución por flujo de datos es útil porque:

- Es una primitiva poderosa para la programación concurrente (ver este capítulo y el capítulo 8). La ejecución por flujo de datos se puede usar para sincronización y comunicación entre computaciones concurrentes. Cuando se usan variables de flujo de datos, muchas técnicas de programación concurrente se simplifican y otras técnicas nuevas aparecen.

Concurrencia declarativa

- Con ella se eliminan las dependencias de orden entre las partes de un programa (ver este capítulo y el capítulo 8). Para ser precisos, la ejecución por flujo de datos reemplaza las dependencias estáticas (decididas por el programador) por dependencias dinámicas (decididas por los datos). Esta es la razón básica por la cual la computación por flujo de datos es útil para la programación paralela. La salida de una parte puede ser pasada, directamente, como entrada a la parte siguiente, independientemente del orden en el cual las partes se ejecuten. Cuando las partes se ejecuten, la segunda se bloqueará sólo si es necesario, i.e., sólo si ella necesita el resultado de la primera y éste no está disponible aún.
- Es una primitiva poderosa para programación distribuida (ver capítulo 11 (en CTM)). La ejecución por flujo de datos mejora la tolerancia de la latencia y la independencia de terceras partes. Una variable de flujo de datos se puede pasar entre los sitios de manera arbitraria. En todo momento, ella “recuerda sus orígenes,” i.e., cuando el valor se sepa, entonces la variable lo recibirá. La comunicación que se necesita para ligar la variable hace parte de la variable y no del programa que manipula la variable.
- Posibilita la realización de cálculos declarativos con información parcial. Esto fue explotado en el capítulo 3 con las listas de diferencias. Una forma de mirar los valores parciales es como valores completos que sólo se conocen parcialmente. Esta idea es poderosa y se explota, posteriormente, en programación por restricciones (ver capítulo 12 (en CTM)).
- Permite soportar, en el modelo declarativo, la programación lógica (ver sección 9.3 (en CTM)). Es decir, es posible asociar una semántica lógica a muchos programas declarativos. Esto nos permite razonar sobre esos programas a un muy alto nivel de abstracción. Desde un punto de vista histórico, las variables de flujo de datos fueron descubiertas originalmente en el contexto de la programación lógica concurrente, donde se les conoce como variables lógicas.

Una manera intuitiva de entender las variables de flujo de datos consiste en verlas como una posición intermedia entre tener y no tener estado:

- Una variable de flujo de datos tiene estado, pues ella puede cambiar su estado (i.e., ser ligada a un valor), pero sólo puede ser ligada una vez en su vida. Este aspecto puede ser utilizado para lograr algunas de las ventajas de la programación con estado (como se explica en el capítulo 6), permaneciendo en el modelo declarativo. Por ejemplo, las listas de diferencias pueden ser concatenadas en tiempo constante, lo cual no es posible con las listas en un modelo puramente funcional.
- Una variable de flujo de datos no tiene estado, pues la ligadura es monótona. Por monótona entendemos que se puede añadir más información a la ligadura, pero ninguna información puede ser cambiada o eliminada. Suponga que la variable está ligada a un valor parcial. Más adelante, ese valor parcial pueden ser ligado una y otra vez, ligando aquellas variables no-ligadas que se encuentran dentro del valor parcial. Pero estas ligaduras no pueden ser cambiadas o deshechas.

El aspecto sin estado se puede usar para lograr algunas de las ventajas de la

programación declarativa dentro de un modelo no declarativo. Por ejemplo, se puede agregar concurrencia al modelo declarativo, dando lugar al modelo concurrente declarativo presentado en este capítulo, precisamente porque los hilos se comunican a través de variables compartidas de flujo de datos.

Futuros y estructuras-I

Las variables de flujo de datos son sólo una técnica para implementar la ejecución por flujo de datos. Otra técnica, bastante popular, se basa en un concepto ligeramente diferente, la variable de asignación única, una variable mutable que puede ser asignada una sola vez. Ésta difiere de una variable de flujo de datos en que esta última puede ser asignada (tal vez muchas veces) a muchos valores parciales, mientras los valores parciales sean compatibles unos con otros. Dos de las más conocidas instancias de la variable de asignación única son los futuros y las estructuras-I. El propósito de los futuros y de las estructuras-I es incrementar el paralelismo potencial de un programa, eliminando las dependencias no esenciales entre los cálculos. Con ellas se logra concurrencia entre una computación que calcula un valor y una que lo usa. Esta concurrencia se puede explotar en una máquina paralela. Ahora definimos los futuros y las estructuras-I y los comparamos con variables de flujo de datos.

Los futuros fueron introducidos por primera vez en Multilisp, un lenguaje orientado a la escritura de programas paralelos [62]. Multilisp introduce la invocación (**future E**) (en sintaxis Lisp), donde *E* es una expresión cualquiera. Esto hace dos cosas: devuelve inmediatamente un contenedor para el resultado de *E* e inicia una evaluación concurrente de *E*. Cuando se necesita el valor de *E*, i.e., una computación trata de acceder al contenedor, entonces la computación se bloquea hasta que el valor esté disponible. En el modelo concurrente declarativo, modelamos esto así: (donde *E* es una función sin argumentos):

```
fun {Future E}
  x in
    thread x={E} end
      !x
  end
```

Un futuro sólo puede ser ligado por la computación concurrente que se crea junto con él. Esto se hace cumplir devolviendo una variable de sólo lectura. Multilisp tiene también un constructor **delay** que no inicia ninguna evaluación hasta que la ejecución by-need la utilice. Esto implica que su argumento sólo se evalúa cuando se necesita el resultado.

Una estructura-I (por “estructura incompleta”) es un arreglo de variables asignación única. Se puede acceder a todos los elementos individuales antes que se computen todos los elementos. Las estructuras-I se introdujeron como una construcción del lenguaje para escribir programas paralelos sobre máquinas de flujo de datos, e.g., en el lenguaje de flujo de datos Id [11, 77, 122, 180]. Las estructuras-I tambien se usan en pH (“parallel Haskell”), un lenguaje de diseño reciente que

Concurrencia declarativa

extiende Haskell con paralelismo implícito [123, 124]. Una estructura-I permite la concurrencia entre una computación que calcula los elementos del arreglo y una computación que utiliza sus valores. Cuando se necesita el valor de un elemento, entonces la computación se bloquea hasta que ese valor esté disponible. Igual que un futuro y que una variable de sólo lectura, un elemento de una estructura-I sólo puede ser ligado por la computación que lo calcula.

Hay una diferencia fundamental entre las variables de flujo de datos, por un lado, y los futuros y las estructuras-I, por el otro. Las últimas sólo pueden ser ligadas una vez, mientras que las variables de flujo de datos pueden ser ligadas varias veces, siempre que las ligaduras sean consistentes unas con otras. Dos valores parciales son consistentes si se pueden unificar. Una variable de flujo de datos se puede ligar muchas veces a diferentes valores parciales, siempre y cuando los valores parciales sean unificables. En la sección 4.3.1 se presenta un ejemplo de esto, cuando se realiza comunicación por flujos con múltiples lectores. A cada lector se le permite ligar la cola de la lista, mientras la ligue de una manera consistente.

4.10. Notas históricas

La concurrencia declarativa tiene una historia larga y respetable. Presentamos algunos de los hechos más destacados. En 1974, Gilles Kahn definió un lenguaje sencillo al estilo Algol, con hilos que se comunicaban por canales que se comportan como colas FIFO con espera bloqueante y remisión no bloqueante [85]. El llamó a este modelo programación paralela determinada.²⁷ En el modelo de Kahn, un hilo podía esperar mensajes sólo por un canal a la vez, i.e., cada hilo siempre conoce por cuál canal vendrá el siguiente mensaje. Además, sólo un hilo puede enviar por cada canal. Esta última restricción es realmente un poco fuerte. El modelo de Kahn podría ser extendido para ser como el modelo concurrente decalarativo. Por un canal puede enviar mensajes más de un hilo, siempre que las remisiones estén ordenadas determinísticamente. Por ejemplo, dos hilos podrían, a su turno, enviar mensajes por el mismo canal.

En 1977, Kahn y David MacQueen extendieron el modelo original de Kahn de manera significativa [86]. El modelo extendido es orientado por la demanda, soporta la reconfiguración dinámica de la estructura de la comunicación, y permite múltiples lectores por un mismo canal.

En 1990, Saraswat, Rinard, y Panangaden generalizaron el modelo original de Kahn a restricciones concurrentes [151]. Esto añadió valores parciales al modelo y plantea los canales de comunicación como flujos. Saraswat et al. definieron primero un lenguaje concurrente por restricciones determinado, el cual es esencialmente el mismo que el modelo dirigido por los datos presentado en este capítulo. Con

27. Por “paralelismo” él entendía concurrencia. En aquellos tiempos el término paralelismo cubría ambos conceptos.

él se generaliza el modelo original de Kahn, y posibilita el uso de técnicas de programación tales como reconfiguración dinámica, canales con múltiples lectores, mensajes incompletos, estructuras de diferencias, y concatenación por recursión de cola.

Saraswat et al. definieron el concepto de punto de reposo, el cual está estrechamente relacionado con la terminación parcial, tal como se definió en la sección 13.2 (en CTM). Un punto de reposo de un programa es un almacén σ que satisface la propiedad siguiente. Cuando el programa se ejecuta con este almacén, nunca se agrega información (el almacén no cambia). El almacén existente en el momento en que un programa ha terminado parcialmente, es un punto de reposo.

Los modelos concurrentes declarativos de este libro están fuertemente relacionados con los artículos citados arriba. El concepto básico de concurrencia determinada fue definido por Kahn. La existencia del modelo dirigido por la demanda está implícita en el trabajo de Saraswat et al. El modelo dirigido por la demanda está relacionado con el modelo de Kahn y MacQueen. La contribución del libro consiste en colocar estos modelos en un marco uniforme que los incluye a todos. En la sección 4.5 se define un modelo dirigido por la demanda agregando sincronización by-need al modelo dirigido por los datos. La sincronización by-need está basada en el concepto de variable necesitada. Como la necesidad está definida como una propiedad monótona, esto nos lleva a un modelo declarativo bastante general que tiene tanto concurrencia como pereza.

4.11. Ejercicios

1. *Semántica de los hilos.* Considere la variación siguiente de la declaración que usamos en la sección 4.1.3 para ilustrar la semántica de los hilos:

```
local B in
    thread B=true end
    thread B=false end
    if B then {Browse si} end
end
```

En este ejercicio, haga lo siguiente:

- a) Enumere todas las ejecuciones posibles de esta declaración.
 - b) Todas las ejecuciones hacen que el programa termine anormalmente. Realice un cambio pequeño al programa para evitar esas terminaciones anormales.
2. *Hilos y recolección de basura.* Este ejercicio examina cómo se comporta la recolección de basura con hilos y variables de flujo de datos. Considere el programa siguiente:

Concurrencia declarativa

```
proc {B _}
    {Wait _}
end

proc {A}
    Recolectable={DiccionarioNuevo}
in
    {B Recolectable}
end
```

¿Después de que se haga la invocación {A}, la variable Recolectable se vuelve basura? Es decir, ¿la memoria que ocupaba Recolectable será recuperada? Responda pensando en la semántica. Verifique que el sistema Mozart se comporta de esa manera.

3. *Fibonacci concurrente*. Considere la definición secuencial de la función de Fibonacci:

```
fun {Fib x}
    if X=<2 then 1
    else {Fib X-1}+{Fib X-2} end
end
```

y compárela con la definición concurrente presentada en la sección 4.2.3. Ejecute ambas en el sistema Mozart y compare sus desempeños. ¿Cuánto más rápida es la definición secuencial? ¿Cuántos hilos crea la invocación concurrente {Fib N} en función de N?

4. *Concurrencia para determinación del orden*. Explique qué pasa cuando ejecuta el siguiente fragmento:

```
declare A B C D in
thread D=C+1 end
thread C=B+1 end
thread A=1 end
thread B=A+1 end
{Browse D}
```

¿En qué orden se crean los hilos? ¿En qué orden se realizan las sumas? ¿Cuál es el resultado final? Compárelo con el siguiente fragmento:

```
declare A B C D in
A=1
B=A+1
C=B+1
D=C+1
{Browse D}
```

Aquí sólo hay un hilo. ¿En qué orden se realizan las sumas? ¿Cuál es el resultado final? ¿Qué puede concluir?

5. *La operación Wait*. Explique por qué la operación {Wait x} podría ser definida así:

```
proc {Wait x}
    if X==listo then skip else skip end
end
```

Utilice su comprensión del comportamiento de flujo de datos de la declaración **if** y de la operación **==**.

6. *Planificación de hilos.* En la sección 4.8.3.2 se muestra cómo llegar a la cola de un flujo, saltándose los elementos ya calculados en él. Si usamos esta técnica para sumar los elementos del flujo de enteros de la sección 4.3.1, el resultado es mucho menor que 11249925000, el cual es la suma de los enteros en el flujo. ¿Por qué es este resultado mucho menor? Explique este resultado en términos de la planificación de los hilos.

7. *Disparadores programados usando programación de alto orden.* Los disparadores programados se pueden implementar utilizando programación de alto orden, en lugar de concurrencia y variables de flujo de datos. El productor le pasa una función **F** sin argumentos al consumidor. Cuando el consumidor necesita un elemento, invoca a la función. Ésta devuelve una pareja **X#F2** donde **X** es el siguiente elemento en el flujo y **F2** es una función que tiene el mismo comportamiento de **F**. Modifique el ejemplo de la sección 4.3.3 para utilizar esta técnica.

8. *Comportamiento de flujo de datos en un contexto concurrente.* Considere la función **{Filter En F}**, que devuelve los elementos de **En** para los cuales la función booleana **F** devuelva **true**. Esta es una posible definición de **Filter**:

```
fun {Filter En F}
  case En
    of X|En2 then
      if {F X} then X|{Filter En2 F}
      else {Filter En2 F} end
    else
      nil
    end
  end
```

Al ejecutar la instrucción siguiente:

```
{Show {Filter [5 1 2 4 0] fun {$ x} x>2 end}}
```

se despliega

```
[5 4]
```

(Usamos el procedimiento **Show**, el cual despliega el valor instantáneo de su argumento en la ventana del emulador de Mozart. Es diferente al **Browse**, pues la salida de **Show** no se actualiza si el argumento se liga posteriormente.) Entonces **Filter** se comporta como se espera, en el caso de una ejecución secuencial y que todos los valores de entrada están disponibles. Ahora exploremos el comportamiento de flujo de datos de **Filter**.

a) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare A
{Show {Filter [5 1 A 4 0] fun {$ x} x>2 end}}
```

Uno de los elementos de la lista es una variable **A** que no ha sido ligada aún a ningún valor. Recuerde que las declaraciones **case** e **if** suspenderán el hilo en

Concurrencia declarativa

que ellas se ejecutan, hasta que puedan decidir qué alternativa tomar.

b) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare Sal A
thread Sal={Filter [5 1 A 4 0] fun {$ x} x>2 end} end
{Show Sal}
```

Recuerde que invocar Show despliega su argumento tal como esté al momento de la invocación. Se pueden desplegar diversos resultados. ¿Cuáles y por qué? ¿La función Filter es determinística? ¿Por qué sí o por qué no?

c) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare Sal A
thread Sal={Filter [5 1 A 4 0] fun {$ x} x>2 end} end
{Delay 1000}
{Show Sal}
```

Recuerde que la invocación {Delay n} suspende su hilo por n ms al menos. Durante este tiempo, otros hilos listos pueden ser ejecutados.

d) ¿Qué pasa cuando se ejecuta lo siguiente?:

```
declare Sal A
thread Sal={Filter [5 1 A 4 0] fun {$ x} x>2 end} end
thread A=6 end
{Delay 1000}
{Show Sal}
```

¿Qué se despliega y por qué?

9. *Simulación de la lógica digital.* En este ejercicio diseñaremos un circuito para sumar números de n -bits y simularlo usando la técnica de la sección 4.3.5. Dados dos números binarios de n -bits, $(x_{n-1}\dots x_0)_2$ y $(y_{n-1}\dots y_0)_2$, construiremos un circuito que sume estos números usando una cadena de sumadores completos, tal como se realiza una suma grande a mano. La idea es sumar cada par de bits por separado y pasar lo que se lleva a la siguiente pareja. Empezamos con los bits de más bajo orden x_0 y y_0 . Se conectan a un sumador completo con una tercera entrada $z = 0$. Esto produce un bit de la suma s_0 y de lo que se lleva c_0 . Ahora conectamos x_1, y_1 , y c_0 a un segundo sumador completo. Éste produce un nuevo bit de la suma s_1 y de lo que se lleva c_1 . Continuamos así para los n bits. El resultado final de la suma es $(s_{n-1}\dots s_0)_2$. En este ejercicio, programe el circuito que realiza la suma utilizando sumadores completos. Verifique que funciona correctamente, realizando varias sumas.

10. *Fundamentos de la pereza.* Considere el fragmento de programa siguiente:

```
fun lazy {Tres} {Delay 1000} 3 end
```

El cálculo {Tres}+0 devuelve 3 después de 1000 ms de retraso. Esto es lo esperado, pues la suma necesita el resultado de {Tres}. Ahora calcule {Tres}+0 tres veces consecutivas. Cada cálculo espera 1000 ms. ¿Por qué pasa esto si se supone que Tres es perezosa? ¿No debería calcularse una sola vez su resultado?

11. *Pereza y concurrencia.* Este ejercicio explora un poco más el comportamiento concurrente de la ejecución perezosa. Ejecute lo siguiente:

```

fun lazy {HagaX} {Browse x} {Delay 3000} 1 end
fun lazy {HagaY} {Browse y} {Delay 6000} 2 end
fun lazy {HagaZ} {Browse z} {Delay 9000} 3 end

X={HagaX}
Y={HagaY}
Z={HagaZ}

{Browse (X+Y)+Z}

```

Esto despliega x y y inmediatamente, z después de seis segundos, y el resultado, 6, después de 15 segundos. Explique este comportamiento. ¿Qué pasa si reemplazamos $(X+Y)+Z$ por $X+(Y+Z)$ o por **thread** $X+Y$ **end** + Z ? ¿En cuál forma se presenta más rápido el resultado final? ¿Cómo programaría la suma de n números enteros i_1, \dots, i_n , sabiendo que el entero i_j estará disponible sólo después de t_j ms, de manera que el resultado final se produzca lo más rápido posible?

12. *Pereza y creación incremental de flujos.* Comparemos la forma de creación incremental de flujos que se consigue a partir de la pereza y la concurrencia. En la sección 4.3.1 se presentó un ejemplo de productor/consumidor usando concurrencia. En la sección 4.5.3 se presentó el mismo ejemplo de productor/consumidor usando la pereza. En ambos casos, el flujo de salida se generaba incrementalmente. ¿Cuál es la diferencia? ¿Qué pasa si se usa tanto concurrencia como pereza en el mismo ejemplo de productor/consumidor?

13. *Pereza y funciones monolíticas.* Considere las dos definiciones siguientes para invertir de forma perezosa una lista:

```

fun lazy {Reverse1 S}
  fun {Rev S R}
    case S of nil then R
      [] X|S2 then {Rev S2 X|R} end
    end
  in {Rev S nil} end
fun lazy {Reverse2 S}
  fun lazy {Rev S R}
    case S of nil then R
      [] X|S2 then {Rev S2 X|R} end
    end
  in {Rev S nil} end

```

¿Cuál es la diferencia de comportamiento entre {Reverse1 [a b c]} y {Reverse2 [a b c]}? ¿Las dos definiciones calculan el mismo resultado? ¿Tienen el mismo comportamiento perezoso? Explique su respuesta en cada caso. Finalmente, compare la eficiencia de la ejecución de las dos definiciones. ¿Cuál definición usaría en un programa perezoso?

14. *Pereza y computación iterativa.* En el modelo declarativo, una ventaja de las variables de flujo de datos es que la definición directa de Append es iterativa. En este ejercicio, considere la versión perezosa directa de Append sin variables de flujo de datos, como se definió en la sección 4.5.7. ¿Esta versión es iterativa? ¿Por qué sí o por qué no?

Concurrencia declarativa

15. *Desempeño de la pereza.* En este ejercicio, tome algunos de los programas declarativos que haya escrito y vuélvalos perezosos declarando todas las rutinas como perezosas. Utilice versiones perezosas de todos los operadores predefinidos, e.g., la suma se vuelve `Sumar`, la cual se define como `fun lazy {Sumar X Y} X+Y end`. Compare el comportamiento de los programas ansiosos originales con los nuevos programas perezosos. ¿Cuál es la diferencia en eficiencia? Algunos lenguajes funcionales, tales como Haskell y Miranda, consideran implícitamente que todas las funciones son perezosas. Para lograr un desempeño razonable, estos lenguajes realizan un análisis de rigidez, el cual trata de encontrar el mayor número posible de funciones que puedan ser compiladas de manera segura como funciones ansiosas.

16. *Ejecución by-need.* Defina una operación que requiera el cálculo de `x` pero que no espere por su valor.

17. *El problema de Hamming.* El problema de Hamming presentado en la sección 4.5.6 es realmente un caso especial del problema original, el cual calcula los n primeros enteros de la forma $p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$ con $a_1, a_2, \dots, a_k \geq 0$ usando los primeros k primos p_1, \dots, p_k . En este ejercicio, escriba un programa que resuelva el problema para cualquier n dado k .

18. *Concurrencia y excepciones.* Considere la abstracción de control siguiente para implementar `try-finally`:

```
proc {TryFinally S1 S2}
  B Y in
    try {S1} B=false catch X then B=true Y=X end
    {S2}
    if B then raise Y end end
  end
```

Utilizando la semántica de la máquina abstracta como guía, determine los diferentes resultados posibles del programa siguiente:

```
local U=1 V=2 in
  {TryFinally
    proc {$}
      thread
        {TryFinally proc {$} U=V end
          proc {$} {Browse ping} end}
      end
    end
    proc {$} {Browse pong} end
  end
```

¿Cuántos resultados diferentes son posibles? ¿Cuántas ejecuciones diferentes son posibles?

19. *Limitaciones de la concurrencia declarativa.* En la sección 4.8 se afirma que la concurrencia declarativa no puede modelar aplicaciones clinete/servidor, porque el servidor no puede leer órdenes de más de un cliente. Sin embargo, la función declarativa `Mezclar` de la sección 4.5.6 lee tres flujos de entrada para generar un flujo de salida. ¿Cómo puede explicar esto?

20. (ejercicio avanzado) *Complejidad en el peor caso y pereza*. En la sección 4.5.8 se explica cómo diseñar una cola con complejidad en tiempo en el peor caso de $O(\log n)$. El logaritmo aparece debido a que la variable F puede tener un número logarítmico de suspensiones asociadas a ella. Miremos cómo pasa esto. Considere una cola vacía a la que, repetidamente, se le añaden elementos nuevos. La tupla $(|F|, |R|)$ empieza en $(0, 0)$. Luego se vuelve $(0, 1)$, lo cual inicia inmediatamente una computación perezosa que hará que la tupla pase a ser, finalmente, $(1, 0)$. (Note que F queda no-ligada y tiene una suspensión asociada.) Cuando se agregan dos elementos más, la tupla se vuelve $(1, 2)$, y se inicia una segunda computación perezosa que hará que la tupla pase a ser, finalmente, $(3, 0)$. Cada vez que se invierte R y se concatena a F , se asocia una nueva suspensión a F . El tamaño de R , que es lo que dispara la computación perezosa, se dobla en cada iteración. Esto es lo que causa la cota logarítmica. En este ejercicio, investigue cómo escribir una cola con complejidad constante en el peor caso. Un enfoque que funciona consiste en usar la idea de una planificación, como se define en [129].
21. (ejercicio avanzado) *Listas por comprensión*. Defina una abstracción lingüística (tanto ansiosa como perezosa) para las listas por comprensión y agréguela al sistema Mozart. Utilice la herramienta generadora de analizadores sintácticos **gump** documentada en [93].
22. (proyecto de investigación) *Control de la concurrencia*. El modelo concurrente declarativo cuenta con tres operaciones primitivas que afectan el orden de ejecución sin cambiar el resultado de una computación: composición secuencial (orden total, dirigido por la oferta), ejecución perezosa (orden total, dirigido por la demanda), y concurrencia (orden parcial, determinado por las dependencias de los datos). Estas operaciones se pueden usar para “afinar” el orden en el cual un programa acepta una entrada y produce un resultado, e.g., ser más, o menos, incremental. Este es un buen ejemplo de separación de asuntos. En este ejercicio, investigue sobre este tema un poco más y responda las preguntas siguientes. ¿Estas tres operaciones son completas? ¿Es decir, todos los posibles órdenes de ejecución parcial se pueden especificar con ellas? ¿Cuál es la relación con las estrategias de reducción en el cálculo λ (e.g., orden de reducción aplicativo, orden de reducción normal)? ¿Son esenciales las variables de flujo de datos o las de asignación única?
23. (proyecto de investigación) *Implementación paralela de lenguajes funcionales*. En la sección 4.9.2 se explica que la evaluación flexible permite sacar ventaja de la ejecución especulativa cuando se implementa un lenguaje funcional paralelo. Sin embargo, utilizar la evaluación flexible dificulta usar estado explícito. En este ejercicio, analice este compromiso. ¿Puede un lenguaje paralelo sacar ventaja tanto de la ejecución especulativa como del estado explícito? Diseñe, implemente y evalúe un lenguaje para verificar sus ideas.

5

Concurrencia por paso de mensajes

Sólo entonces Atreyu se dió cuenta que el monstruo no era un cuerpo único y sólido, sino que estaba hecho de innumerables insectos pequeños de acero azul, que zumbaban como avispones enfadados. Su enjambre compacto era lo que le permitía tomar, continuamente, diferentes formas.

– Adaptación libre de *The Neverending Story*, Michael Ende (1929–1995)

El paso de mensajes es un estilo de programación en el cual un programa consiste de entidades independientes que interactúan enviándose mensajes entre ellas, de manera asincrónica, i.e., sin esperar por una respuesta. Este estilo de programación fue estudiado primero por Carl Hewitt en el modelo actor [70, 71]. El paso de mensajes es importante en tres áreas:

- Es el marco fundamental para sistemas multiagentes, una disciplina que mira los sistemas complejos como un conjunto de “agentes” interactuando entre ellos. Los agentes son entidades independientes que trabajan para lograr sus objetivos propios, locales. Si la interacción se diseña adecuadamente, entonces los agentes pueden lograr objetivos globales. Por ejemplo, la asignación de recursos puede ser realizada eficientemente por agentes egoístas que interactúan de acuerdo a mecanismos inspirados por una economía de mercado [148, 183].
- Es el estilo natural para un sistema distribuido, i.e., un conjunto de computadores que se pueden comunicar entre ellos a través de una red. Es natural porque refleja la estructura del sistema y sus costos. Los sistemas distribuidos se están volviendo ubícuos debido a la continua expansión del Internet. Las tecnologías antiguas para los sistemas de programación distribuida, tales como RPC, CORBA, y RMI, están basadas en comunicación sincrónica. Las tecnologías más nuevas, tales como servicios Web, son asincrónicas. Las técnicas de este capítulo se aplican directamente a las tecnologías asincrónicas. (Las particularidades de los sistemas de programación distribuida se exploran más adelante en el capítulo 11 (en CTM).)
- Se presta muy bien para la construcción de sistemas altamente confiables. Como las entidades en el paso de mensajes son independientes, si alguna falla las otras pueden continuar su ejecución. En un sistema diseñado adecuadamente, las otras entidades se reorganizan para continuar proveyendo el servicio. Esta idea es utilizada por el lenguaje Erlang, el cual se usa en telecomunicaciones, y redes de alta velocidad (ver sección 5.7).

Concurrencia por paso de mensajes

Definimos un modelo computacional para el paso de mensajes como una extensión del modelo concurrente declarativo. Luego usamos este modelo para mostrar cómo programar con paso de mensajes.

Extendiendo el modelo concurrente declarativo

El modelo concurrente declarativo del último capítulo no puede tener un no-determinismo observable. Esto limita los tipos de programas que podemos escribir en el modelo. Por ejemplo, vemos que es imposible escribir un programa cliente/servidor donde el servidor no conozca cuál cliente le enviará el siguiente mensaje.

El modelo concurrente por paso de mensajes extiende el modelo concurrente declarativo agregando solamente un concepto nuevo, un canal asincrónico de comunicación. Esto significa que cualquier cliente puede, en cualquier momento, enviar mensajes al canal, y el servidor puede leer todos los mensajes de ese canal. Esto elimina la limitación sobre qué tipo de programas podemos escribir. Un programa cliente/servidor puede dar resultados diferentes en ejecuciones diferentes debido a que el orden de los envíos del cliente no está determinado. Esto significa que el modelo de paso de mensajes es no determinístico y, por tanto, deja de ser declarativo.

Utilizamos un tipo sencillo de canal, denominado puerto, el cual tiene un flujo asociado a él. El envío de un mensaje al puerto hace que el mensaje aparezca en el flujo del puerto. Una técnica de programación útil consiste en asociar un puerto con un objeto flujo. Llamamos a la entidad resultante un objeto puerto. Un objeto puerto lee todos sus mensajes del objeto flujo, y envía mensajes a otros objetos puerto a través de sus puertos. Cada objeto puerto está definido por un procedimiento recursivo que es declarativo. Esto conserva algunas de las ventajas del modelo declarativo.

Estructura del capítulo

El capítulo consiste de las partes siguientes:

- En la sección 5.1 se define el modelo concurrente por paso de mensajes. Allí se define el concepto de puerto y el lenguaje núcleo.
- En la sección 5.2 se introduce el concepto de objetos puerto, el cual se consigue combinando puertos con objetos flujo.
- En la sección 5.3 se muestra cómo hacer ciertos protocolos sencillos de mensajes con objetos puerto.
- En la sección 5.4 se explica cómo diseñar programas con componentes concurrentes. Allí se definen los conceptos básicos y se presenta una metodología para administrar la concurrencia.
- En la sección 5.5 se presenta un caso de estudio de esa metodología. Allí se usan objetos puerto para construir un sistema de control de ascensores.
- En la sección 5.6 se muestra cómo usar directamente el modelo de paso de

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Invocación de procedimiento
thread $\langle d \rangle$ end	Creación de hilo
$\{ \text{NewName } \langle x \rangle \}$	Creación de nombre
$\langle y \rangle = ! ! \langle x \rangle$	Vista de sólo lectura
try $\langle d \rangle_1$ catch $\langle x \rangle$ then $\langle d \rangle_2$ end	Contexto de la excepción
raise $\langle x \rangle$ end	Lanzamiento de excepción
$\{ \text{NewPort } \langle y \rangle \langle x \rangle \}$	Creación de puerto
$\{ \text{Send } \langle x \rangle \langle y \rangle \}$	Remisión al puerto

Tabla 5.1: El lenguaje núcleo con concurrencia por paso de mensajes.

mensajes, sin utilizar la abstracción del objeto puerto. Puede ser más difícil de razonar sobre los programas así que utilizando los objetos puerto, pero algunas veces es útil.

- En la sección 5.7 se presenta una introducción a Erlang, un lenguaje de programación basado en objetos puerto, usado para construir sistemas altamente confiables.
- En la sección 5.8 se explica un tema avanzado: el modelo concurrente no-determinístico, el cual es un modelo intermedio, en términos de expresividad, entre el modelo concurrente declarativo y el modelo por paso de mensajes de este capítulo.

5.1. El modelo concurrente por paso de mensajes

El modelo concurrente por paso de mensajes extiende el modelo concurrente declarativo por medio de la adición de puertos. En la tabla 5.1 se muestra el lenguaje núcleo. Los puertos son una especie de canal de comunicación. Los puertos no son declarativos pues permiten un no-determinismo observable: muchos hilos pueden enviar mensajes a un puerto pero su orden no está determinado. Sin embargo, la parte de la computación que no utiliza puertos puede seguir siendo declarativa. Esto significa que teniendo cuidado podemos utilizar aún muchas de las técnicas de razonamiento del modelo concurrente declarativo.

Concurrencia por paso de mensajes

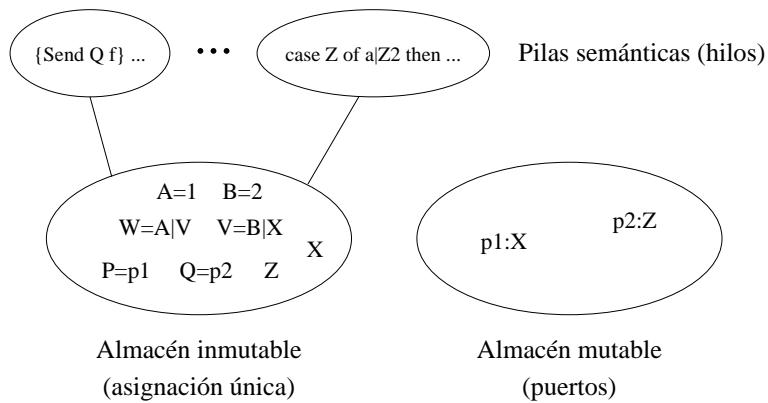


Figura 5.1: El modelo concurrente por paso de mensajes.

5.1.1. Puertos

Un puerto es un TAD con dos operaciones, a saber, la creación de un canal y el envío de algo por él:

- `{NewPort ?S ?P}`: crea un puerto nuevo con punto de entrada `P` y flujo `S`.
- `{Send P x}`: concatena `x` al flujo correspondiente al punto de entrada `P`.

Las remisiones consecutivas desde el mismo hilo aparecen en el flujo en el mismo orden en que fueron ejecutadas. Esta propiedad implica que un puerto es un canal de comunicación FIFO, asincrónico. Por ejemplo:

```
declare S P in
{NewPort S P}
{Browse S}
{Send P a}
{Send P b}
```

Esto despliega en el browser el flujo `a|b|_`. Realizar más envíos extenderá el flujo. Si el extremo actual del flujo es `S`, entonces realizar el envío `{Send P a}` ligará `S` a `a|s1`, y `s1` se convierte en el nuevo extremo del flujo. Internamente, el puerto siempre recuerda el extremo actual del flujo. El extremo del flujo es una variable de sólo lectura. Esto significa que un puerto es un TAD seguro.

Por asincrónico queremos decir que un hilo que envía un mensaje no se queda esperando por una respuesta. Tan pronto como el mensaje es colocado en el canal de comunicación, el hilo puede continuar su ejecución. Esto quiere decir que el canal de comunicación puede contener muchos mensajes pendientes, los cuales están esperando ser atendidos.

5.1.2. Semántica de los puertos

La semántica de los puertos es bastante simple. Para definirla, primero extendemos el estado de ejecución del modelo declarativo agregando un almacén mutable como se muestra en la figura 5.1. Luego definimos las operaciones `NewPort` y `Send` en términos de ese almacén.

Extensión del estado de ejecución

Además del almacén de asignación única σ (y del almacén de disparadores τ , si la pereza es importante) agregamos un almacén nuevo, μ , denominado el almacén mutable. Este almacén contiene puertos, los cuales son parejas de la forma $x : y$, donde x y y son variables del almacén de asignación única. El almacén mutable se encuentra inicialmente vacío. La semántica garantiza que x siempre esté ligada a un valor de tipo nombre que representa un puerto y que y siempre sea no-ligada. Usamos valores de tipo nombre para identificar los puertos porque estos valores son constantes únicas e inexpugnables. El estado de ejecución es ahora una tripleta (MST, σ, μ) (o una cuádrupla (MST, σ, μ, τ) si se considera el almacén de disparadores).

La operación NewPort

La declaración semántica $(\{\text{NewPort } \langle x \rangle \langle y \rangle\}, E)$ hace lo siguiente:

- Crea un nombre fresco para el puerto, n .
- Liga $E(\langle y \rangle)$ y n en el almacén.
- Si la ligadura es exitosa, entonces agrega la pareja $E(\langle y \rangle) : E(\langle x \rangle)$ al almacén mutable μ .
- Si la ligadura falla, entonces lanza una condición de error.

La operación Send

La declaración semántica $(\{\text{Send } \langle x \rangle \langle y \rangle\}, E)$ hace lo siguiente:

- Si la condición de activación es cierta ($E(\langle x \rangle)$ está determinada), entonces se realizan las acciones siguientes:
 - Si $E(\langle x \rangle)$ no está ligada al nombre de un puerto, entonces se lanza una condición de error.
 - Si el almacén mutable contiene $E(\langle x \rangle) : z$, entonces se realizan las acciones siguientes:
 - Crear una variable nueva, z' , en el almacén.
 - Actualizar el almacén mutable de manera que en lugar de la pareja $E(\langle x \rangle) : z$, aparezca la pareja $E(\langle x \rangle) : z'$.

Concurrencia por paso de mensajes

- Ligar z en el almacén con la lista nueva $E(\langle y \rangle) | z'$.
- Si la condición de activación es falsa, entonces se suspende la ejecución.

Esta semántica está ligeramente simplificada con respecto a la semántica completa de los puertos. En un puerto correcto, el extremo del flujo siempre debe ser una vista de sólo lectura. Esto implicaría hacer una extensión directa a la semántica de las operaciones `NewPort` y `Send`, lo cual dejamos como un ejercicio para el lector.

Administración de la memoria

Se requieren dos modificaciones a la administración de la memoria, debido al almacén mutable:

- Extender la definición de alcanzabilidad: Una variable y es alcanzable si el almacén mutable contiene la pareja $x : y$ y x es alcanzable.
- Recuperación de puertos: si una variable x se vuelve inalcanzable, y el almacén mutable contiene la pareja $x : y$, entonces se elimina esa pareja.

5.2. Objetos puerto

Un objeto puerto es una combinación de uno o más puertos y un objeto flujo. Esto extiende los objetos flujo en dos formas. Primero, se hace posible la comunicación muchos-a-uno: muchos hilos pueden referenciar un objeto puerto dado y enviarle mensajes a él de manera independiente. Esto no es posible con un objeto flujo porque éste tiene que conocer de dónde viene el próximo mensaje. Segundo, los objetos puerto se pueden embeber dentro de estructuras de datos (incluyendo los mensajes). Esto no es posible con un objeto flujo pues éste está referenciado por un flujo que sólo puede ser extendido por un único hilo.

El concepto de objeto puerto tiene muchas variaciones populares. Algunas veces se usa la palabra “agente” para cubrir una idea similar: una entidad activa con la cual uno puede intercambiar mensajes. El sistema Erlang tiene el concepto de “proceso,” el cual es como un objeto puerto salvo que allí se agrega un buzón de correo adjunto que permite filtrar los mensajes entrantes usando reconocimiento de patrones. Otro término frecuentemente usado es “objeto activo.” Éste es similar a un objeto puerto salvo que está definido de forma orientada a objetos, por una clase (como lo veremos en el capítulo 7). En este capítulo sólo usaremos objetos puerto.

En el modelo por paso de mensajes, un programa consiste de un conjunto de objetos puerto que envían y reciben mensajes. Los objetos puerto pueden crear nuevos objetos puerto. También pueden enviar mensajes que contengan referencias a otros objetos puerto. Esto significa que el conjunto de objetos puerto forma un grafo que puede evolucionar durante la ejecución.

Los objetos puerto se pueden usar para modelar sistemas distribuidos. Un objeto puerto modela un computador o un proceso del sistema operativo. Un algoritmo

distribuido no es más que un algoritmo entre objetos puerto.

Un objeto puerto tiene la estructura siguiente:

```
declare P1 P2 ... Pn in
local S1 S2 ... Sn in
  {NewPort S1 P1}
  {NewPort S2 P2}
  ...
  {NewPort Sn Pn}
  thread {PR S1 S2 ... Sn} end
end
```

El hilo contiene un procedimiento recursivo PR que lee los flujos del puerto y realiza ciertas acciones por cada mensaje recibido. Enviar un mensaje a un objeto puerto es simplemente enviar un mensaje a uno de sus puertos. Presentamos un ejemplo de un objeto puerto, con un puerto, que despliega todos los mensajes que se reciben:

```
declare P in
local S in
  {NewPort S P}
  thread {ForAll S Browse} end
end
```

Recuerde que Browse es un procedimiento de un argumento. Con la sintaxis de ciclo **for**, esto se puede escribir así:

```
declare P in
local S in
  {NewPort S P}
  thread for M in S do {Browse M} end end
end
```

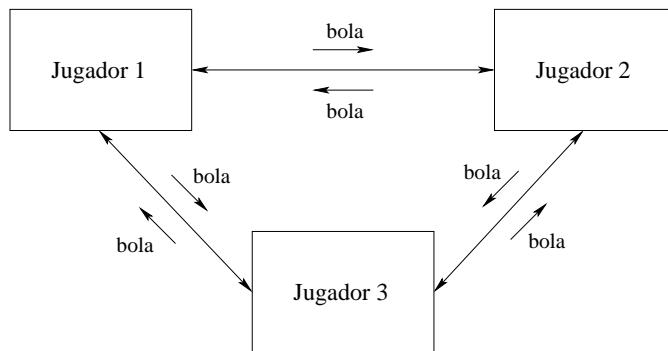
Al invocar {Send P hola} se desplegará, finalmente, hola. Podemos comparar esto con los objetos flujo del capítulo 4. La diferencia es que los objetos puerto permiten la comunicación muchos-a-uno, i.e., cualquier hilo que refiera el puerto puede enviar un mensaje al objeto puerto en cualquier momento. Esto contrasta con los objetos flujo, donde el objeto siempre conoce de cuál hilo procederá el próximo mensaje.

5.2.1. La abstracción NuevoObjetoPuerto

Podemos definir una abstracción para facilitar la programación con objetos puerto. Definamos una abstracción para el caso en que el objeto puerto tiene un solo puerto. Para definir el objeto puerto, sólo tenemos que pasar el estado inicial Inic y la función de transición de estados Fun. El tipo de esta función es **fun** {\$ T_s T_m} : T_s donde T_s es el tipo del estado y T_m es el tipo del mensaje.

```
fun {NuevoObjetoPuerto Inic Fun}
Sen Ssal in
  thread {FoldL Sen Fun Inic Ssal} end
  {NewPort Sen}
end
```

Concurrencia por paso de mensajes

**Figura 5.2:** Tres objetos puerto jugando con la bola.

Aquí se usa FoldL para implementar el ciclo acumulador. Este es un ejemplo excelente de concurrencia declarativa. Cuando el flujo de entrada se termina, el estado final aparece en `Ssal`. Algunos objetos puerto son puramente reactivos, i.e., no tienen estado interno. La abstracción es aún más sencilla para ellos, pues no se necesita ningún acumulador:

```

fun {NuevoObjetoPuerto2 Proc}
  Sen in
    thread for Msj in Sen do {Proc Msj} end end
    {NewPort Sen}
  end
  
```

No hay función de transición de estados, sino simplemente un procedimiento que es invocado con cada mensaje.

5.2.2. Un ejemplo

Suponga que hay tres jugadores colocados en un círculo, lanzándose una bola entre ellos. Cuando un jugador recibe la bola, él o ella escoje uno de los otros dos al azar para lanzarle la bola. Podemos modelar esta situación con objetos puerto. Considere tres objetos puerto, donde cada objeto tiene una referencia a los otros. Existe una bola que se envía entre los objetos. Cuando un objeto puerto recibe la bola, inmediatamente la envía a otro, seleccionado al azar. En la figura 5.2 se muestran los tres objetos y qué mensajes puede enviar cada objeto y a dónde. Tal diagrama se denomina un diagrama de componentes. Para programar esto, primero definimos un componente que crea un jugador nuevo:

```
fun {Jugador Otros}
    {NuevoObjetoPuerto2
proc {$ Msg}
    case Msg of bola then
        Ran={OS.rand} mod {Width Otros} + 1
    in
        {Send Otros.Ran bola}
    end
end}
```

Otros es una tupla que contiene los otros jugadores. Ahora podemos configurar el juego:

```
J1={Jugador otros(J2 J3)}
J2={Jugador otros(J1 J3)}
J3={Jugador otros(J1 J2)}
```

En este programa, Jugador es un componente y J1, J2, J3 son sus instancias. Para comenzar el juego, lanzamos una bola a uno de los jugadores:

```
{Send J1 bola}
```

Esto inicia un juego tremadamente rápido de lanzarse la bola. Para desacelerarlo, podemos agregar un {Delay 1000} en cada jugador.

5.2.3. Razonando con objetos puerto

Considere un programa consistente de objetos puerto enviándose mensajes entre ellos. Probar que el programa es correcto consiste de dos partes: probar que cada objeto puerto es correcto (cuando se considera por si mismo) y probar que los objetos puerto trabajan juntos correctamente. La primera etapa consiste en mostrar que cada objeto puerto es correcto. Cada objeto puerto define una abstracción de datos. La abstracción debe tener una afirmación invariante, i.e., una afirmación que es cierta cuando una operación abstracta haya terminado y antes de que la operación siguiente haya comenzado. Para mostrar que la abstracción es correcta es suficiente mostrar que la afirmación es invariante. Ya mostramos cómo hacerlo para el modelo declarativo en el capítulo 3. Como el interior de un objeto puerto es declarativo (es una función recursiva que lee un flujo), podemos usar las técnicas que mostramos allí.

Como el objeto puerto tiene un solo hilo, sus operaciones se ejecutan secuencialmente. Esto significa que podemos usar la inducción matemática para mostrar que la afirmación es invariante. Tenemos que probar dos cosas:

- Cuando el objeto puerto se crea, la afirmación es cierta.
- Si la afirmación es cierta antes de que un mensaje sea procesado, entonces la afirmación es cierta después que el mensaje haya sido procesado.

La existencia del invariante muestra que el objeto puerto por si mismo es correcto. La siguiente etapa consiste en mostrar que el programa que utiliza los objetos puerto

es correcto. Esto requiere todo un conjunto de técnicas diferentes.

Un programa en el modelo por paso de mensajes es un conjunto de objetos puerto enviándose mensajes entre ellos. Para mostrar que esto es correcto, tenemos que determinar cuáles son las posibles secuencias de mensajes que cada objeto puerto puede recibir. Para determinar esto, empezamos por clasificar todos los eventos en el sistema. Los eventos son de tres clases: envío de mensajes, recepción de mensajes, y cambios del estado interno de un objeto puerto. Podemos entonces, definir la causalidad entre eventos (si un evento ocurre antes que otro). Considerando el sistema de objetos puerto como un sistema de transición de estados, entonces podemos razonar sobre el programa completo. Explicar esto en detalle está más allá del alcance de este capítulo. Los lectores interesados pueden referirse a los libros sobre sistemas distribuidos, tales como Lynch [108] o Tel [171].

5.3. Protocolos sencillos de mensajes

Los objetos puerto trabajan en conjunto intercambiando mensajes de manera coordinada. Es interesante estudiar qué tipos de coordinación son importantes. Esto nos lleva a definir un protocolo como una secuencia de mensajes, entre dos o más partes, que puede ser entendida a un nivel de abstracción más alto que sólo como mensajes individuales. Demos una mirada más detallada a los protocolos de mensajes y miremos cómo implementarlos con objetos puerto.

La mayoría de los protocolos más conocidos son bastante complicados tales como los protocolos de Internet (TCP/IP, HTTP, FTP, etc.) o los protocolos LAN (redes de área local) tales como Ethernet, DHCP (Dynamic Host Connection Protocol), entre otros [97]. En esta sección mostramos cómo implementar algunos protocolos más sencillos utilizando objetos puerto. Todos los ejemplos utilizan `NuevoObjetoPuerto2` para crear objetos puerto.

En la figura 5.3 se muestran los diagramas de mensajes de muchos de los protocolos sencillos (¡dejamos los otros diagramas al lector!). Estos diagramas muestran los mensajes intercambiados entre un cliente (denotado `c`) y un servidor (denotado `s`). El tiempo fluye hacia abajo. En la figura se tiene el cuidado de distinguir los hilos desocupados (que están disponibles para requerimientos de servicio) de los hilos suspendidos (los cuales no se encuentran disponibles).

5.3.1. RMI (Invocación Remota de Métodos)

Tal vez el protocolo sencillo más popular es el protocolo RMI.¹ Por medio de él, un objeto puede invocar a otro en un proceso diferente del sistema operativo, ya sea en la misma máquina o en otra máquina conectada por una red [169]. Históricamente,

1. Nota del traductor: del inglés, *Remote Method Invocation*.

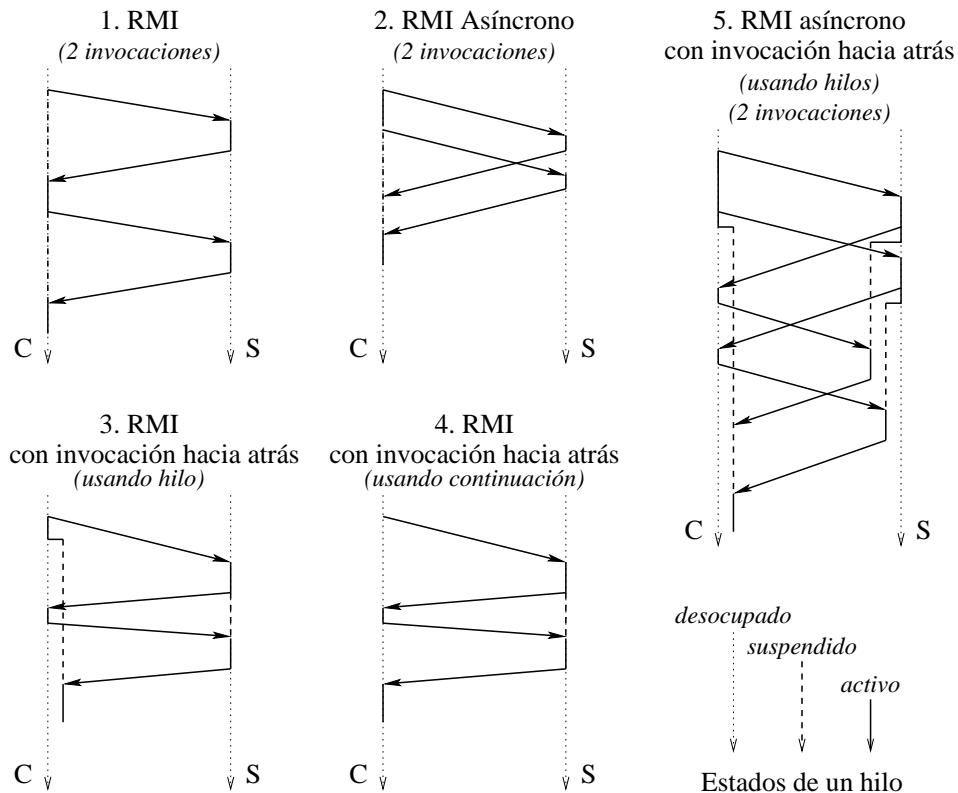


Figura 5.3: Diagramas de mensajes de protocolos sencillos.

RMI es un descendiente de RPC,² el cual fue inventado en los inicios de la década de 1980, antes de que la programación orientada a objetos se volviera popular [21]. El término RMI se volvió popular una vez los objetos comenzaron a reemplazar a los procedimientos como las entidades que se invocaban de manera remota. Aplicamos el término RMI de manera algo vaga a los objetos puerto, aunque ellos no tienen métodos en el sentido de la programación orientada a objetos (ver el capítulo 7 para mayor información sobre los métodos). Por ahora, supongamos que un método no es más que lo que un objeto puerto hace cuando recibe un mensaje particular.

Desde el punto de vista del programador, los protocolos RMI y RPC son bastante sencillos: un cliente envía una solicitud a un servidor y espera que el servidor le envíe una respuesta. (Este punto de vista abstrae detalles de implementación tales como la forma en que las estructuras de datos se pasan de un espacio de direcciones a otro.) Presentamos un ejemplo sencillo. Primero definimos el servidor como un objeto puerto:

2. Nota del traductor: del inglés, *Remote Procedure Call*.

Concurrencia por paso de mensajes

```
proc {ProcServidor Msj}
  case Msj
    of calc(X Y) then
      Y=X*X+2.0*X+2.0
    end
  end
  Servidor={NuevoObjetoPuerto2 ProcServidor}
```

Este servidor en particular, no tiene un estado interno. El segundo argumento de calc, Y, es ligado por el servidor. Suponemos que el servidor realiza un cálculo complejo, lo cual lo modelamos con el polinomio $X^2 + 2.0 \cdot X + 2.0$. Definimos el cliente:

```
proc {ProcCliente Msj}
  case Msj
    of valor(Y) then Y1 Y2 in
      {Send Servidor calc(10.0 Y1)}
      {Wait Y1}
      {Send Servidor calc(20.0 Y2)}
      {Wait Y2}
      Y=Y1+Y2
    end
  end
  Cliente={NuevoObjetoPuerto2 ProcCliente}
  {Browse {Send Cliente valor($)}}}
```

Note que usamos una marca de anidamiento, “\$”. Recordamos que la última línea es equivalente a

```
local x in {Send Cliente valor(x)} {Browse x} end
```

Los marcadores de anidamiento son una forma conveniente de convertir declaraciones en expresiones. Hay una diferencia interesante entre las definiciones del cliente y del servidor. La definición del cliente referencia al servidor directamente, mientras que la definición del servidor no sabe quienes son sus clientes. El servidor recibe una referencia indirecta al cliente, a través del argumento Y. Esta es una variable de flujo de datos que el servidor liga con la respuesta. El cliente espera hasta recibir la respuesta antes de continuar.

En este ejemplo, todos los mensajes se ejecutan secuencialmente por el servidor. En nuestra experiencia, esta es la mejor manera de implementar RMI. Así es sencillo programar y razonar sobre los programas. Algunas implementaciones de RMI hacen las cosas un poco diferente, pues permiten que múltiples invocaciones de clientes diferentes se procesen concurrentemente. Esto se hace permitiendo múltiples hilos del lado del servidor que aceptan solicitudes para el mismo objeto. El servidor deja de resolver las solicitudes secuencialmente. Así es mucho más difícil de programar: se requiere que el servidor proteja sus datos sobre el estado interno. Este caso lo examinaremos más adelante en el capítulo 8. Cuando se programa en un lenguaje que provea RMI o RPC, tal como C o Java, es importante conocer si el servidor atiende los mensajes secuencialmente o no.

En este ejemplo, tanto el cliente como el servidor se escribieron en el mismo

lenguaje y se ejecutan en el mismo proceso del sistema operativo. Así es para todos los programas de este capítulo. Cuando se programa un sistema distribuido, ya no es así. Por ejemplo, dos procesos del sistema operativo que ejecuten programas Java pueden comunicarse por medio de Java RMI. Por otro lado, dos procesos que ejecutan programas escritos en diferentes lenguajes se pueden comunicar utilizando CORBA o Servicios Web. Las técnicas generales de programación presentadas en este capítulo sirven aún en estos casos, con algunas modificaciones debido a la naturaleza de los sistemas distribuidos. Esto se explica en el capítulo 11 (en CTM).

5.3.2. RMI asincrónico

Otro protocolo útil es el RMI asincrónico. Este protocolo es similar a RMI, salvo que el cliente continúa la ejecución inmediatamente después de enviar la solicitud al servidor. Al cliente se le informa cuando la respuesta llegue. Con este protocolo, se pueden realizar dos solicitudes sucesivas rápidamente. Si las comunicaciones entre el cliente y el servidor son lentas, entonces, con este protocolo, se tendrá una gran ventaja en desempeño con respecto a RMI. En RMI, solamente podemos enviar la segunda solicitud después de que la primera se ha completado, i.e., después de una ronda del cliente al servidor. Este es el cliente:

```
proc {ProcCliente Msj}
  case Msj
    of valor(?Y) then Y1 Y2 in
      {Send Servidor calc(10.0 Y1)}
      {Send Servidor calc(20.0 Y2)}
      Y=Y1+Y2
    end
  end
  Cliente={NuevoObjetoPuerto2 ProcCliente}
  {Browse {Send Cliente valor($)}}}
```

Los envíos de los mensajes se solapan. El cliente espera por los resultados Y_1 y Y_2 antes de realizar la suma.

Note que el servidor es el mismo que para RMI estándar. El servidor aún recibe los mensajes uno por uno y los ejecuta secuencialmente. Las solicitudes son manejadas por el servidor en el mismo orden en que llegan y las respuestas se envían en ese orden también.

5.3.3. RMI con invocación de vuelta (usando hilos)

El protocolo RMI con invocación de vuelta es como RMI salvo que el servidor necesita invocar al cliente para poder completar la solicitud. Veamos un ejemplo. En este caso, el servidor hace una invocación de vuelta (al cliente) para conseguir el valor de un parámetro especial llamado `delta`, el cual sólo es conocido por el cliente:

Concurrencia por paso de mensajes

```
proc {ProcServidor Msj}
  case Msj
    of calc(X ?Y Cliente) then X1 D in
      {Send Cliente delta(D)}
      X1=X+D
      Y=X1*X1+2.0*X1+2.0
    end
  end
  Servidor={NuevoObjetoPuerto2 ProcServidor}
```

El servidor conoce la referencia al cliente porque es un argumento del mensaje calc. No colocamos {Wait D} pues está implícito en la suma X+D. A continuación, vemos un cliente que invoca al servidor de la misma forma que en RMI:

```
proc {ProcCliente Msj}
  case Msj
    of valor(?Z) then Y in
      {Send Servidor calc(10.0 Y Client)}
      Z=Y+100.0
    [] delta(?D) then
      D=1.0
    end
  end
  Cliente={NuevoObjetoPuerto2 ProcCliente}
  {Browse {Send Cliente valor($)}}}
```

(Igual que antes, wait está implícito.) Desafortunadamente esta solución no funciona. Se queda en un abrazo mortal durante la invocación {Send Client valor(\$)}. ¿Ve Usted por qué? Dibuje un diagrama de mensajes para ver por qué.³ Esto muestra que un protocolo RMI sencillo no es el concepto correcto para realizar invocaciones de vuelta.

La solución a este problema es que el cliente no espere por la respuesta del servidor. El cliente debe continuar inmediatamente después de realizar su invocación, de manera que esté listo para aceptar la invocación de vuelta. Cuando la respuesta, finalmente, llegue, el cliente debe manejarla correctamente. Presentamos una forma de escribir un cliente correcto:

```
proc {ProcCliente Msj}
  case Msj
    of valor(?Z) then Y in
      {Send Servidor calc(10.0 Y Cliente)}
      thread Z=Y+100.0 end
    [] delta(?D) then
      D=1.0
    end
  end
  Cliente={NuevoObjetoPuerto2 ProcCliente}
  {Browse {Send Cliente valor($)}}}
```

3. Es porque el cliente queda suspendido cuando invoca al servidor, de manera que el cliente no puede atender la invocación de vuelta del servidor.

En lugar de esperar a que el servidor ligue a y , el cliente crea un nuevo hilo donde espera ese hecho. El nuevo cuerpo de ese hilo consiste en el trabajo a realizar una vez y sea ligada. Cuando la respuesta, finalmente, llegue, el hilo nuevo hará el trabajo y ligará z .

Es interesante ver qué sucede cuando invocamos este cliente desde afuera. Por ejemplo, realicemos la invocación `{Send Cliente valor(z)}`. Cuando esta invocación termine, normalmente z no estará ligada aún. Normalmente esto no es un problema, pues la operación que utiliza a z se bloqueará hasta que z sea ligada. Si este comportamiento no es deseable, entonces la misma invocación al cliente puede ser tratada como un RMI:

```
{Send Cliente valor(z)}  
{Wait z}
```

Esto traslada la sincronización del cliente a la aplicación que utiliza el cliente. Esta es la forma correcta de manejar este problema. El problema con la solución errada original, era que la sincronización se realizaba en el lugar equivocado.

5.3.4. RMI con invocación de vuelta (usando un registro de continuación)

La solución del ejemplo anterior crea un hilo nuevo por cada invocación al cliente. Esto supone que los hilos son económicos. ¿Cómo resolver el problema si no se nos permite crear un hilo nuevo? La solución es que el cliente pase, lo que se denomina, una (señal de) continuación al servidor. Después de que el servidor termine su trabajo, envía la continuación de vuelta al cliente, de manera que éste pueda continuar. De esta forma, el cliente nunca espera y se evita el abrazo mortal. Esta es la definición del servidor:

```
proc {ProcServidor Msj}  
    case Msj  
        of calc(X Cliente Cont) then X1 D Y in  
            {Send Cliente delta(D)}  
            X1=X+D  
            Y=X1*X1+2.0*X1+2.0  
            {Send Cliente Cont#Y}  
    end  
end  
Servidor={NuevoObjetoPuerto2 ProcServidor}
```

Después de terminar su trabajo, el servidor envía $Cont\#Y$ de vuelta al cliente. Se agrega Y a la continuación pues ¡el cliente necesita a Y !

Concurrencia por paso de mensajes

```
proc {ProcCliente Msj}
  case Msj
    of valor(?Z) then
      {Send Servidor calc(10.0 Cliente cont(Z))}
    [] cont(Z)#Y then
      Z=Y+100.0
    [] delta(?D) then
      D=1.0
    end
  end
  Cliente={NuevoObjetoPuerto2 ProcCliente}
  {Browse {Send Cliente valor($)}}}
```

La parte de valor después de invocar al servidor quedó colocada en un nuevo registro, cont. El cliente envía al servidor la continuación cont(z); el servidor calcula y y luego deja al cliente continuar con su trabajo enviándole cont(z)#y.

Cuando el cliente es invocado desde afuera, la solución, a las invocaciones de vuelta, basada en continuaciones, se comporta de la misma manera que la solución basada en hilos. A saber, z, normalmente, no estará ligada aún, cuando la invocación al cliente termine. Manejamos esto de la misma manera que con la solución basada en hilos, trasladando la sincronización del cliente a quien lo invoca.

5.3.5. RMI con invocación de vuelta (usando un procedimiento de continuación)

El ejemplo anterior se puede generalizar de manera poderosa enviando, como continuación, un procedimiento en lugar de un registro. Cambiamos el cliente como sigue (el servidor no cambia):

```
proc {ProcCliente Msj}
  case Msj
    of valor(?Z) then
      C=proc {$ Y} Z=Y+100.0 end
    in
      {Send Servidor calc(10.0 Cliente cont(C))}
    [] cont(C)#Y then
      {C Y}
    [] delta(?D) then
      D=1.0
    end
  end
  Cliente={NuevoObjetoPuerto2 ProcCliente}
  {Browse {Send Cliente valor($)}}}
```

La continuación contiene el trabajo que el cliente debe realizar una vez la invocación al servidor termine. Como la continuación es un valor de tipo procedimiento, es auto contenida: puede ser ejecutada por cualquiera sin conocer su interior.

5.3.6. Reporte de errores

Todos los protocolos que hemos cubierto hasta este momento suponen que el servidor siempre hará su trabajo correctamente. ¿Qué se debe hacer si este no es el caso, i.e., si el servidor puede cometer un error ocasionalmente? Por ejemplo, lo puede cometer debido a un problema de red entre el cliente y el servidor, o porque el proceso servidor dejó de ejecutarse. En cualquier caso, el cliente debe ser notificado de que ha ocurrido un error. La forma natural de notificar al cliente es lanzando una excepción. Así se modifica el servidor para hacer esto:

```
proc {ProcServidor Msj}
  case Msj
    of raizc(X Y E) then
      try
        Y={Sqrt X}
        E=normal
      catch Exc then
        E=exception(Exc)
      end
    end
  end
  Servidor={NuevoObjetoPuerto2 ProcServidor}
```

El argumento extra `E` indica si la ejecución fue normal o no. El servidor calcula raíces cuadradas. Si el argumento es negativo, `Sqrt` lanza una excepción, la cual es capturada y enviada al cliente.

Este servidor puede ser invocado por protocolos tanto sincrónicos como asincrónicos. En un protocolo sincrónico, el cliente puede invocar el servidor así:

```
{Send Server raizc(X Y E)}
  case E of exception(Exc) then raise Exc end end
```

La declaración `case` bloquea al cliente hasta que `E` sea ligada. De esta manera, el cliente se sincroniza sobre una de las dos cosas que pueden suceder: un resultado normal o una excepción. Si se lanza una excepción en el servidor, entonces la excepción es lanzada de nuevo en el cliente. Esto garantiza que `Y` no vaya a ser usada a menos que sea ligada a un resultado normal. En un protocolo asincrónico no hay garantías. Es responsabilidad del cliente comprobar `E` antes de usar `Y`.

Este ejemplo supone que el servidor puede capturar la excepción y enviársela de vuelta al cliente. ¿Qué sucede si el servidor falla o el enlace de comunicación entre el cliente y el servidor se rompe o es demasiado lento para que el cliente espere? Estos casos se manejan en el capítulo 11 (en CTM).

5.3.7. RMI asincrónico con invocación de vuelta

Los protocolos se pueden combinar para crear unos más sofisticados. Por ejemplo, podríamos querer hacer dos RMIs asincrónicos donde cada RMI realiza una invocación de vuelta. Este es el servidor:

Concurrencia por paso de mensajes

```

proc {ProcServidor Msj}
  case Msj
    of calc(X ?Y Cliente) then X1 D in
      {Send Cliente delta(D)}
      thread
        X1=X+D
        Y=X1*X1+2.0*X1+2.0
      end
    end
  end

```

Este es el cliente:

```

proc {ProcCliente Msj}
  case Msj
    of valor(?Y) then Y1 Y2 in
      {Send Servidor calc(10.0 Y1 Cliente)}
      {Send Servidor calc(20.0 Y2 Cliente)}
      thread Y=Y1+Y2 end
      [ ] delta(?D) then
        D=1.0
      end
    end

```

¿Cuál es el diagrama de mensajes para la invocación {Send Cliente valor(Y)}?
 ¿Qué sucedería si el servidor no crea un hilo para realizar el trabajo posterior a la invocación de vuelta?

5.3.8. Doble invocación de vuelta

Algunas veces el servidor realiza una primera invocación de vuelta al cliente, el cual, a su vez, realiza una segunda invocación de vuelta al servidor. Para manejar esto, tanto el cliente como el servidor deben continuar inmediatamente y no esperar el resultado de vuelta. Este es el servidor:

```

proc {ProcServidor Msj}
  case Msj
    of calc(X ?Y Cliente) then X1 D in
      {Send Cliente delta(D)}
      thread
        X1=X+D
        Y=X1*X1+2.0*X1+2.0
      end
      [ ] servidordelta(?S) then
        S=0.01
      end
    end

```

Este es el cliente:

```
proc {ProcCliente Msj}
  case Msj
    of valor(Z) then Y in
      {Send Servidor calc(10.0 Y Cliente)}
      thread Z=Y+100.0 end
    [] delta(?D) then S in
      {Send Servidor servidordelta(S)}
      thread D=1.0+S end
    end
  end
```

La invocación `{Send Cliente valor(Z)}` invoca al servidor, el cual invoca el método `delta(D)` del cliente, el cual invoca el método `servidordelta(S)` del servidor. Una pregunta para un lector atento: ¿Por qué se coloca también, la última declaración, `D=1.0+S`, en un hilo?⁴

5.4. Diseño de programas con concurrencia

En esta sección se presenta una introducción a la programación basada en componentes, con componentes concurrentes.

En la sección 4.3.5 vimos cómo realizar diseño lógico utilizando el modelo concurrente declarativo. Allí definimos las compuertas lógicas como componentes básicos de un circuito y mostramos cómo componerlas para construir circuitos cada vez más grandes. Cada circuito contaba con entradas y salidas, las cuales se modelaron como flujos.

En esta sección se continúa esa discusión en un contexto más general, pues la colocamos en el contexto de la programación basada en componentes. Gracias a la concurrencia por paso de mensajes ya no tenemos las limitaciones de la ejecución sincrónica “al mismo paso” que presentamos en el capítulo 4.

Primero introduciremos los conceptos básicos de diseño concurrente. Luego presentaremos un ejemplo práctico, un sistema de control de ascensores. Mostramos cómo diseñar e implementar ese sistema utilizando diagramas de componentes y diagramas de estado, de alto nivel. Empezamos por explicar estos conceptos.

5.4.1. Programación con componentes concurrentes

Para diseñar una aplicación concurrente, la primera etapa consiste en modelarla como un conjunto de actividades concurrentes que interactúan en formas bien definidas. Cada actividad concurrente se modela exactamente con un componente concurrente. A un componente concurrente también se le conoce con el nombre de “agente.” Los agentes pueden ser reactivos (no tienen estado interno) o tener un estado interno. La ciencia de la programación con agentes también es conocida con

4. Estrictamente hablando, no se necesita en este ejemplo. Pero en general, ¡el cliente no sabe si el servidor realizará otra invocación de vuelta!

Concurrencia por paso de mensajes

el nombre de sistemas multiagentes, frecuentemente abreviados como MAS.⁵ En los MAS, se han ideado muchos protocolos diferentes con complejidades variables. En esta sección se tocan brevemente estos protocolos. En la programación basada en componentes, los agentes son considerados, normalmente, entidades bastante simples con poca inteligencia incorporada. En la comunidad de inteligencia artificial, se considera que los agentes, normalmente, realizan el mismo tipo de razonamiento.

Definamos un modelo sencillo para la programación con componentes concurrentes. El modelo tiene componentes primitivos y formas de combinar componentes. Los componentes primitivos se usan para crear objetos puerto.

Un componente concurrente

Definimos un modelo sencillo, para programación basada en componentes, basado en objetos puerto y ejecutado con concurrencia por paso de mensajes. En este modelo, un componente concurrente es un procedimiento con entradas y salidas. Cuando se invoca, el componente crea una instancia del componente, la cual es un objeto puerto. Una entrada es un puerto cuyo flujo es leído por el componente. Una salida es un puerto por el cual el componente puede enviar mensajes.

Por diferentes razones, los procedimientos son el concepto correcto para modelar componentes concurrentes. Ellos son compositacionales y pueden tener un número arbitrario de entradas y salidas. Cuando se componen los subcomponentes, ellos permiten la visibilidad de las entradas y las salidas que deben ser controladas, e.g., algunos pueden tener visibilidad restringida al interior del componente.

Interfaz

Un componente concurrente interactúa con su ambiente a través de su interfaz. La interfaz consiste del conjunto de sus entradas y salidas, las cuales se conocen colectivamente como sus cables. Un cable conecta una o más salidas a una o más entradas. El modelo por paso de mensajes de este capítulo provee dos tipos básicos de cables: de un tiro y de muchos tiros. Los cables de un tiro se implementan con variables de flujo de datos. Ellos se usan para valores que no cambian o para mensajes por una sola vez (como acuses de recibo). Solamente se puede pasar un mensaje, y sólo se puede conectar una salida a una entrada dada. Los cables de muchos tiros se implementan con puertos. Ellos se utilizan para flujos de mensajes. Se puede enviar cualquier número de mensajes, y cualquier número de salidas pueden escribir en una entrada dada.

El modelo concurrente declarativo del capítulo 4 cuenta, también, con cables de uno y de muchos tiros, pero los últimos están restringidos en que sólo una salida puede escribir a una entrada dada.⁶

5. Nota del traductor: de la sigla en inglés, *multi-agent systems*.

6. Para ser precisos, muchas salidas pueden escribir a una entrada dada, pero deben escribir información compatible.

Operaciones básicas

Hay cuatro operaciones básicas en la programación basada en componentes:

1. Instanciación: creación de una instancia de un componente. Por defecto, cada instancia es independiente de las otras. En algunos casos, las instancias pueden tener una dependencia sobre una instancia compartida.
2. Composición: construcción de un nuevo componente a partir de otros componentes. Los últimos se denominan subcomponentes para enfatizar su relación con el componente nuevo. Suponemos por defecto, que los componentes que deseamos componer son independientes. ¡Esto quiere decir que son concurrentes! De manera sorprendente, tal vez, los componentes compuestos en un sistema secuencial tienen dependencias aún si no comparten argumentos. Esto se debe a que la ejecución es secuencial.
3. Acoplamiento: Combinación de instancias de componentes, por medio de la conexión de entradas y salidas de unos y otros. Hay diferentes tipos de conexiones (o enlaces): de un tiro o de muchos tiros; las entradas pueden ser conectadas a una salida solamente o a muchas salidas; las salidas pueden ser conectadas a una entrada solamente o a muchas entradas. Normalmente, las conexiones de un tiro van de una salida a muchas entradas. Todas las entradas ven el mismo valor cuando esté disponible. Las conexiones de muchos tiros van de muchas salidas a muchas entradas. Todas las entradas ven el mismo flujo de valores de entrada.
4. Restricción: como su nombre lo dice, restricción de la visibilidad de las entradas o las salidas dentro de un componente compuesto. Restringir significa limitar algunas de los cables de la interfaz de los subcomponentes al interior del componente nuevo, i.e., esos cables no aparecen en la interfaz del componente nuevo.

Presentamos un ejemplo para ilustrar estos conceptos. En la sección 4.3.5 mostramos cómo modelar circuitos lógicos digitales como componentes. Definimos los procedimientos CAnd, COr, CNot, y CRetraso para implementar compuertas lógicas. Al ejecutar uno de estos procedimientos se crea una instancia del componente. Estas instancias son objetos flujo pero hubieran podido ser objetos puerto. (Un ejercicio sencillo consiste en generalizar las compuertas lógicas para que se conviertan en objetos puerto). Definimos también un cerrojo, como un componente compuesto en términos de compuertas, como sigue:

```
proc {Cerrojo C DI ?DO}
    X Y Z F
    in
        {CRetraso DO F}
        {And F C X}
        {CNot C Z}
        {CAnd Z DI Y}
        {COr X Y DO}
    end
```

Concurrencia por paso de mensajes

El componente cerrojo tiene cinco subcomponentes. Ellos se acoplan entre ellos conectando salidas y entradas. Por ejemplo, la salida x de la primera compuerta And se conecta como una entrada a la compuerta Or. Solamente los cables C , DI , y DO son visibles desde el exterior del cerrojo. Los cables x , y , z , y F están restringidos al interior del componente.

5.4.2. Metodología de diseño

Diseñar un programa concurrente es más difícil que diseñar un programa secuencial, pues, normalmente, existen muchas más interacciones potenciales entre las diferentes partes. Para tener confianza en que un programa concurrente es correcto, necesitamos seguir una secuencia de reglas de diseño que no sean ambiguas. De acuerdo a nuestra experiencia, las reglas de diseño de esta sección producen buenos resultados si se siguen con algún rigor.

- *Especificación informal.* Escribir una especificación, posiblemente informal pero precisa, de lo que el sistema debería hacer.
- *Componentes.* Enumerar todas las formas diferentes de actividad concurrente en la especificación. Cada actividad se volverá un componente. Dibujar un diagrama de bloques del sistema que muestre todas las instancias de componentes.
- *Protocolos de mensajes.* Decida qué mensajes enviarán los componentes y diseñe los protocolos de mensajes entre ellos. Dibujar el diagrama de componentes con todos los protocolos de mensajes.
- *Diagramas de estado.* Para cada entidad concurrente, escribir su diagrama de estados. Para cada estado, verificar que se reciben y se envían todos los mensajes apropiados con las condiciones y acciones correctas.
- *Implementar y planificar.* Codificar el sistema en el lenguaje de programación favorito. Decidir cuál algoritmo de planificación se utilizará para implementar la concurrencia entre los componentes.
- *Probar e iterar.* Probar el sistema y reiterar hasta que se satisfaga la especificación inicial.

Usamos estas reglas para diseñar el sistema de control de ascensores que presentaremos más adelante.

5.4.3. Componentes funcionales básicos como patrones de concurrencia

La programación con componentes concurrentes trae como consecuencia una gran cantidad de protocolos de mensajes. Algunos protocolos sencillos se ilustran en la sección 5.3. También hay protocolos mucho más complicados. Debido a la cercanía entre la concurrencia por paso de mensajes y la concurrencia declarativa, muchos de estos protocolos se pueden programar como operaciones sencillas sobre listas.

Todas las operaciones estándar sobre listas (e.g., del módulo `List`) se pueden interpretar como patrones de concurrencia. Veremos que esta es una forma poderosa

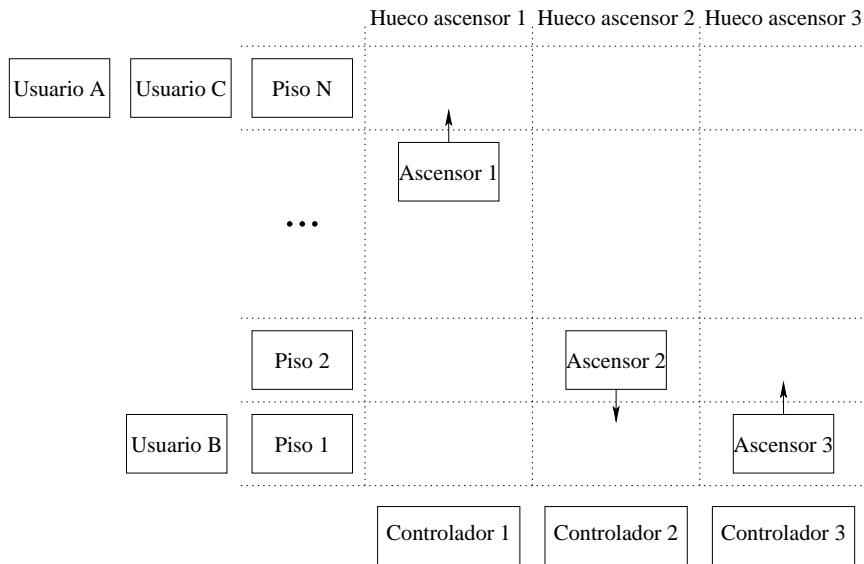


Figura 5.4: Vista esquemática de la edificación con ascensores.

de escribir programas concurrentes. Por ejemplo, la función `Map` estándar se puede usar como un patrón que transmite solicitudes y recolecta las respuestas en una lista. Considere una lista `LP` de puertos, cada uno de los cuales es el puerto de entrada de un objeto puerto. Nos gustaría enviar el mensaje `solic(foo Res)` a cada objeto puerto, el cual ligará finalmente `Res` con la respuesta. Utilizando `Map` podemos enviar todos los mensajes y recolectar las respuestas, en una sola línea:

```
LR={Map LP fun {$ P} Res in {Send P solic(foo Res)} Res end}
```

Las solicitudes se envían de manera asíncrona y las respuestas aparecerán finalmente en la lista `LR`. Podemos simplificar la notación, aún más, utilizando el marcador de anidamiento `$` con la operación `Send`. Esto evita por completo mencionar la variable `Res`:

```
LR={Map LP fun {$ P} {Send P solic(foo $)} end}
```

Podemos realizar cálculos con `LR` como si las respuestas estuvieran allí; los cálculos esperarán automáticamente cuando se necesite. Por ejemplo, si las respuestas son enteros positivos, entonces podemos calcular el máximo en la misma forma que en un programa secuencial:

```
M={FoldL LR Max 0}
```

En la sección 5.2.1 se muestra otra forma de usar `FoldL` como un patrón de concurrencia.

5.5. Sistema de control de ascensores

Los ascensores hacen parte de nuestro día a día.⁷ Sin embargo, alguna vez se ha preguntado cómo funcionan? ¿Cómo se comunican los ascensores con los pisos y cómo decide un ascensor a qué piso ir? Hay muchas maneras de programar un sistema de control de ascensores.

En esta sección diseñaremos un sistema sencillo de control de ascensores, en forma de programa concurrente, utilizando la metodología presentada en la sección anterior. Nuestro primer diseño será bastante sencillo. A pesar de ello, como lo verá, el programa concurrente resultante será bastante complejo. Por lo tanto, tendremos mucho cuidado en seguir la metodología de diseño presentada anteriormente.

Modelaremos la operación de un sistema hipotético de control de ascensores de una edificación, con un número fijo de ascensores, un número fijo de pisos entre los cuales viajan los ascensores, y los usuarios. En la figura 5.4 se presenta una vista abstracta de cómo se ve la edificación. Hay pisos, ascensores, controladores del movimiento de los ascensores, y usuarios que van y vienen. Modelaremos lo que pasa cuando un usuario llama el ascensor para ir a otro piso. Nuestro modelo se enfocará en la concurrencia y la coordinación, para mostrar correctamente cómo interactúan las actividades concurrentes en el tiempo. Pero pondremos suficiente atención para lograr un programa que funcione.

La primera tarea es la especificación. En este caso, estaremos satisfechos con una especificación parcial del problema. Hay un conjunto de pisos y un conjunto de ascensores. Cada piso cuenta con un botón, que los usuarios pueden presionar, para llamar el ascensor. El botón de llamado no especifica ninguna dirección hacia arriba o hacia abajo. El piso escoge aleatoriamente el ascensor que le va a prestar el servicio. Cada ascensor tiene una serie de botones, piso(I), numerados para todos los pisos I, para solicitarle que se detenga en un piso dado. Cada ascensor tiene un plan, el cual consiste en la lista de los pisos que debe visitar, en el orden en que debe hacerlo.

El algoritmo de planificación que usaremos se denomina FCFS:⁸ un piso nuevo siempre se coloca al final del plan. Esto es lo mismo que la planificación FIFO. Tanto el botón para llamar el ascensor desde un piso, como los botones piso(I) planifican con FCFS. Este no es el mejor algoritmo de planificación para ascensores pero tiene la ventaja de su sencillez. Más adelante veremos cómo mejorarlo. Cuando un ascensor llega a un piso planificado, las puertas se abren y permanecen abiertas por un tiempo fijo antes de cerrarse de nuevo. El ascensor toma un tiempo fijo para moverse de un piso al siguiente.

El sistema de control de ascensores se ha diseñado como un conjunto de componentes concurrentes que interactúan entre ellos. En la figura 5.5 se muestra el

7. Los elevadores son útiles para aquellos que viven en departamentos, de la misma manera que los ascensores son útiles para aquellos que viven en apartamentos.

8. Nota del traductor: de las siglas en inglés, *first-come, first-served*.

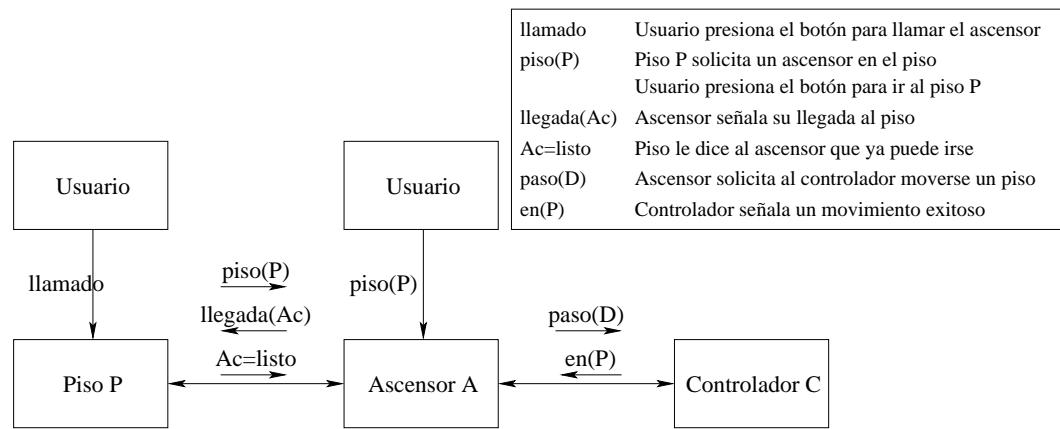


Figura 5.5: Diagrama de componentes del sistema de control de ascensores.

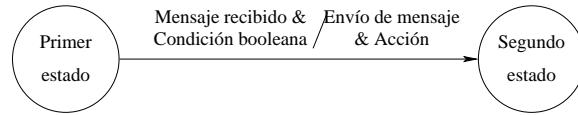


Figura 5.6: Notación para diagramas de estado.

diagrama de bloques de sus interacciones. Cada rectángulo representa una instancia de un componente concurrente. En nuestro diseño, hay cuatro tipos de componentes, a saber pisos, ascensores, controladores, y temporizadores. Todas las instancias de componentes son objetos puerto. Los controladores se usan para manejar el movimiento de los ascensores. Los temporizadores manejan el aspecto de tiempo real del sistema.

Debido a la planificación FCFS, los ascensores se moverán con cierta frecuencia más lejos de lo necesario. Si un ascensor ya está en un piso, entonces solicitar un ascensor para ese piso, de nuevo, puede llamar a otro ascensor. Si un ascensor está en camino de un piso a otro, entonces solicitar un piso intermedio no hará que el ascensor se detenga allí. Podemos evitar esos problemas agregando un poco de inteligencia al planificador. Una vez hayamos determinado la estructura de toda la aplicación, será más claro cómo realizar ésta y otras mejoras.

5.5.1. Diagramas de transición de estados

Una buena forma de diseñar un objeto puerto es empezar por enumerar los estados en que puede estar y los mensajes que puede enviar y recibir. Esto facilita el

Concurrencia por paso de mensajes

comprobar que todos los mensajes se manejen apropiadamente en todos los estados. Revisaremos los diagramas de transición de estados de cada componente, pero primero introduciremos la notación para estos diagramas (algunas veces llamados diagramas de transición, en breve).

Un diagrama de transición de estados es un autómata de estados finitos; por tanto consiste de un conjunto finito de estados y un conjunto de transiciones entre estados. En cada instante, el componente está en un estado en particular. Inicialmente, el componente se encuentra en un estado inicial, y evoluciona realizando transiciones. Una transición es una operación atómica que realiza lo siguiente. La transición está habilitada cuando se recibe el mensaje apropiado y se cumple una condición booleana sobre el mensaje y el estado. Entonces, la transición puede enviar un mensaje y cambiar el estado. En la figura 5.6 se muestra la notación gráfica. Cada círculo representa un estado, y las flechas entre los círculos representan transiciones.

Los mensajes se pueden enviar de dos maneras: a un puerto o ligando una variable de flujo de datos. Los mensajes se pueden recibir por el flujo del puerto o esperando una ligadura. Las variables de flujo de datos se usan como canales livianos por los cuales sólo se puede enviar un mensaje (un “cable de un tiro”). Para modelar retrasos en el tiempo, usamos un protocolo temporizador: el invocador `Pid` envía el mensaje `iniciotemporizador(N Pid)` a un agente temporizador para solicitar un retraso de `N ms`. El invocador continúa inmediatamente después de enviado el mensaje. Cuando el tiempo ha pasado, el agente temporizador envía el mensaje de vuelta, `fintemporizador`, al invocador. (El protocolo temporizador es similar a la operación `{Delay N}`, reformulada en el estilo de componentes concurrentes.)

5.5.2. Implementación

Presentamos la implementación del sistema de control de ascensores mostrando cada parte de manera separada, a saber, el controlador, el piso, y el ascensor. Definiremos las funciones para crearlos:

- `{Piso Num Inic Ascensores}` devuelve una instancia del componente piso `IdPiso`, con número `Num`, estado inicial `Inic`, y ascensores `Ascensores`.
- `{Ascensor Num Inic IdC Pisos}` devuelve una instancia del componente ascensor `IdA`, con número `Num`, estado inicial `Inic`, controlador `IdC`, y pisos `Pisos`.
- `{Controlador Inic}` devuelve una instancia del componente controlador `IdC`.

Para cada función, explicamos cómo funciona y presentamos el diagrama de estados y el código fuente. Luego creamos una edificación con un sistema de control de ascensores y mostramos cómo interactúan los componentes.

El controlador

El controlador es el más fácil de explicar. Tiene dos estados, motor detenido y motor trabajando. En el estado motor detenido, el controlador puede recibir el

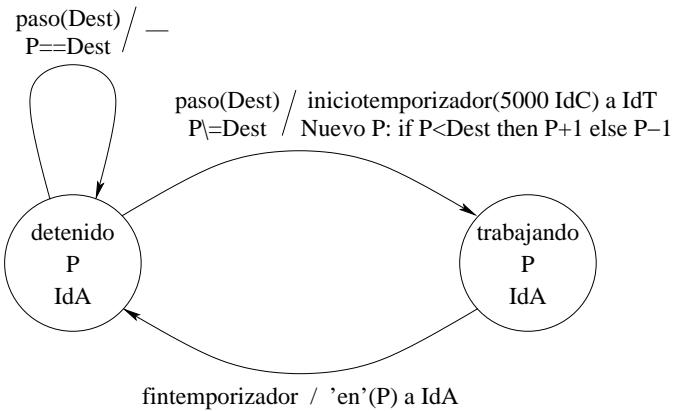


Figura 5.7: Diagrama de estados de un controlador de un ascensor.

mensaje `paso(Dest)` de parte del ascensor, donde `Dest` es el número del piso de destino. Entonces, el controlador pasa al estado motor trabajando. Dependiendo del piso actual y el piso destino, el controlador se mueve un piso hacia arriba o un piso hacia abajo. Con la ayuda del protocolo temporizador, el controlador pasa del estado motor trabajando al estado motor detenido luego de transcurrido un tiempo fijo. Este es el tiempo que se necesita para que el ascensor se mueva un piso (hacia arriba o hacia abajo). En el ejemplo, suponemos que este tiempo es 5000 ms. El protocolo temporizador modela una implementación real la cual tendría un sensor en cada piso. Cuando el ascensor llega al piso `P`, el controlador envía al ascensor el mensaje `'en'(P)`. En la figura 5.7 se presenta el diagrama de estados del controlador `IdC`.

El código fuente del temporizador y el controlador se presenta en la figura 5.8. Es interesante comparar el código del controlador con el diagrama de estados. El temporizador definido aquí, también se usa en el componente piso.

Los lectores atentos notarán que el controlador realmente tiene más de dos estados, pues estrictamente hablando el número del piso hace parte del estado. Para conservar el diagrama de estados sencillo, parametrizamos los estados motor detenido y motor trabajando con el número del piso. Esta representación, de varios estados como uno solo con variables internas, es una especie de azúcar sintáctico para diagramas de estado. Esto nos permite representar diagramas muy grandes en una forma compacta. También usaremos esta técnica para los diagramas de estado de los componentes piso y ascensor.

El piso

Los pisos son más complicados pues pueden estar en uno de tres estados: no se ha solicitado ningún ascensor para el piso, se ha solicitado un ascensor para el piso

Concurrencia por paso de mensajes

```

fun {Temporizador}
    {NuevoObjetoPuerto2
        proc {$ Msj}
            case Msj of iniciotemporizador(T IdC) then
                thread {Delay T} {Send IdC fintemportador} end
            end
        end
    end

fun {Controlador Inic}
    IdT={Temporizador}
    IdC={NuevoObjetoPuerto Inic
        fun {$ estado(Motor P IdA) Msj}
            case Motor
            of trabajando then
                case Msj
                of fintemportador then
                    {Send IdA `en'(P)}
                    estado(detenido P IdA)
                end
            [] detenido then
                case Msj
                of paso(Dest) then
                    if P==Dest then
                        estado(detenido P IdA)
                    elseif P<Dest then
                        {Send IdT iniciotemporizador(5000 IdC)}
                        estado(trabajando P+1 IdA)
                    else % P>Dest
                        {Send IdT iniciotemporizador(5000 IdC)}
                        estado(trabajando P-1 IdA)
                    end
                end
            end
        end
    in IdC end

```

Figura 5.8: Implementación de los componentes temporizador y controlador.

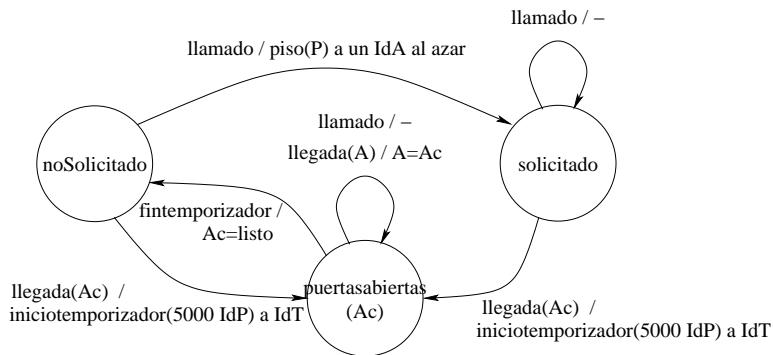


Figura 5.9: Diagrama de estados de un piso.

pero no ha llegado, y un ascensor está en el piso con las puertas abiertas. En la figura 5.9 se presenta el diagrama de estados del piso IdPiso. Cada piso puede recibir un mensaje `llamado` de un usuario, un mensaje `llegada(Ac)` de un ascensor, y un mensaje interno del temporizador. El piso puede enviar un mensaje `piso(P)` a un ascensor.

El código fuente del componente piso se muestra en la figura 5.10. Allí se utiliza la función que genera un número aleatorio `OS.rand`, para escoger un ascensor al azar. También se utiliza `Browse` para desplegar en el browser cuándo se solicita un ascensor y cuándo se abren y se cierran las puertas del ascensor en el piso. Se supone que el tiempo total necesario para abrir y cerrar las puertas es 5000 ms.

El ascensor

El componente ascensor es el más complicado de todos. En la figura 5.11 se presenta el diagrama de estados del ascensor IdA. Cada ascensor puede estar en uno de cuatro estados: sin plan y detenido (desocupado), con plan y moviéndose hacia un piso dado, en un piso donde lo solicitaron y esperando la señal de cierre de puertas, esperando el cierre de puertas cuando lo solicitaron en el piso donde estaba desocupado. La mejor manera de entender esta figura es seguirla a través de algunos escenarios de ejecución. Por ejemplo, considere el siguiente escenario sencillo. Un usuario presiona el botón de solicitud del ascensor, en el piso 1. Entonces, el piso envía el mensaje `piso(1)` a un ascensor (al azar). El ascensor recibe este mensaje y envía `paso(1)` al controlador. Supongamos que el ascensor está actualmente en el piso 3. Entonces, el controlador envía el mensaje `en(2)` al ascensor, el cual envía de nuevo el mensaje `paso(1)` al controlador. Ahora, el controlador envía el mensaje `en(1)` al ascensor, el cual entonces envía el mensaje `llegada(Ac)` al piso 1, y espera a que el piso le confirme que ya cerró las puertas y pueda volver a moverse.

Cada ascensor puede recibir mensajes de la forma `piso(N)` o de la forma

Concurrencia por paso de mensajes

```

fun {Piso Num Inic Ascensores}
  IdT={Temporizador}
  IdPiso={NuevoObjetoPuerto Inic
    fun {$ estado(EstP) Msj}
      case EstP
        of nosolicitado then AsAzar in
          case Msj
            of llegada(Ac) then
              {Browse `Ascensor en el piso '#Num#: apertura
puertas`}
              {Send IdT iniciotemporizador(5000 IdPiso)}
              estado(puertasabiertas(Ac))
            [ ] llamado then
              {Browse `Piso '#Num# solicita un ascensor!`}
              AsAzar=Ascensores.(1+{OS.rand} mod {Width
Ascensores})
              {Send AsAzar piso(Num)}
              estado(solicitado)
            end
            [ ] solicitado then
              case Msj
                of llegada(Ac) then
                  {Browse `Ascensor en el piso '#Num#: apertura
puertas`}
                  {Send IdT iniciotemporizador(5000 IdPiso)}
                  estado(puertasabiertas(Ac))
                [ ] llamado then
                  estado(solicitado)
                end
                [ ] puertasabiertas(Ac) then
                  case Msj
                    of fintemporizador then
                      {Browse `Ascensor en el piso '#Num#: cierre
puertas`}
                      Ac=listo
                      estado(nosolicitado)
                    [ ] llegada(A) then
                      A=Ac
                      estado(puertasabiertas(Ac))
                    [ ] llamado then
                      estado(puertasabiertas(Ac))
                    end
                  end
                end
              end
            in IdPiso end

```

Figura 5.10: Implementación del componente piso.

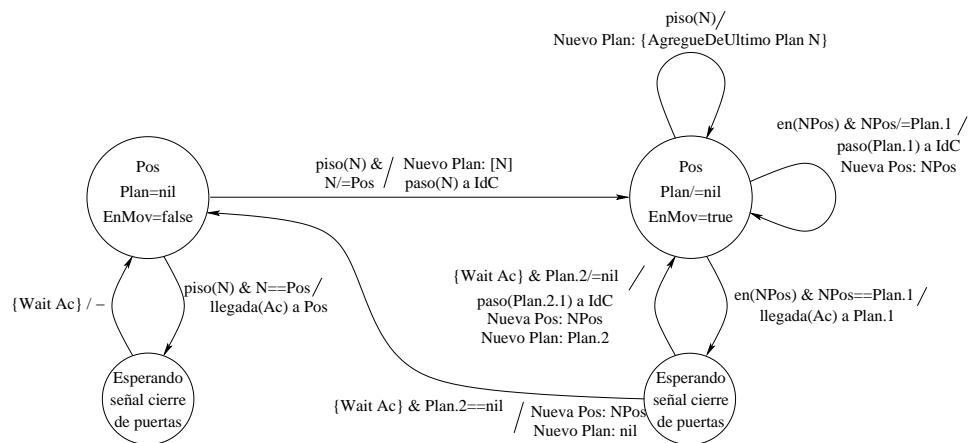


Figura 5.11: Diagrama de estados de un ascensor.

‘en’ (N). El ascensor, a su vez, puede enviar un mensaje `llegada(Ac)` a un piso o un mensaje `paso(Dest)` a su controlador. Después de enviar el mensaje `llegada(Ac)`, el ascensor espera a que el piso le confirme que ya abrió y cerró las puertas. Esta confirmación se realiza usando la variable de flujo de datos `Ac=listo`; el ascensor espera por medio de la invocación `{Wait Ac}`.

En la figura 5.12, se muestra el código fuente del componente ascensor. Allí se utiliza una serie de declaraciones `if` para implementar las condiciones de las diferentes transiciones. También se usa `Browse` para desplegar cuándo un ascensor se dirige a un piso solicitado y cuándo llega a un piso solicitado. La función `{AgregueDeUltimo L N}` implementa el planificador: agrega N al final del plan L (si el piso no estaba planificado aún) y devuelve el nuevo plan.

La edificación

Ya que hemos especificado el sistema completo, resulta instructivo seguir la ejecución a mano, observando el flujo de control en los pisos, ascensores, controladores, y temporizadores. Por ejemplo, supongamos que hay diez pisos y dos ascensores. Ambos ascensores se encuentran en el piso 1, y solicitan un ascensor en el piso 9 y en el piso 10. ¿Cuáles son las ejecuciones posibles del sistema? Definamos un componente compuesto que crea una edificación con NP pisos y NA ascensores:

Concurrencia por paso de mensajes

```

fun {AgregueDeUltimo L N}
    if L\=nil andthen {List.last L}==N then L
    else {Append L [N]} end
end

fun {Ascensor Num Iinic IdC Pisos}
    {NuevoObjetoPuerto Iinic
        fun {$ estado(Pos Plan EnMov) Msj}
            case Msj
            of piso(N) then
                {Browse `Ascensor '#Num#` solicitado en el piso '#N}
                if N==Pos andthen {Not EnMov} then
                    {Wait {Send Pisos.Pos llegada($)}}
                    estado(Pos Plan false)
                else Plan2 in
                    Plan2={AgregueDeUltimo Plan N}
                    if {Not EnMov} then
                        {Send IdC paso(N)} end
                        estado(Pos Plan2 true)
                    end
                [] `en`{NewPos} then
                    {Browse `Ascensor '#Num#` en el piso '#NewPos}
                    case Plan
                    of S|Plan2 then
                        if NewPos==S then
                            {Wait {Send Pisos.S llegada($)}}
                            if Plan2==nil then
                                estado(NewPos nil false)
                            else
                                {Send IdC paso(Plan2.1)}
                                estado(NewPos Plan2 true)
                            end
                        else
                            {Send IdC paso(S)}
                            estado(NewPos Plan EnMov)
                        end
                    end
                end
            end
        end
    end
}

```

Figura 5.12: Implementación del componente ascensor.

```

proc {Edificación NP NA ?Pisos ?Ascensores}
    Ascensores={MakeTuple ascensores NA}
    for I in 1..NA do IdC in
        IdC={Controlador estado(detenido 1 Ascensores.I)}
        Ascensores.I={Ascensor I estado(1 nil false) IdC Pisos}
    end
    Pisos={MakeTuple pisos NP}
    for I in 1..NP do
        Pisos.I={Piso I estado(nosolicitado) Ascensores}
    end
end

```

Aquí usamos `MakeTuple` para crear una tupla nueva de variables no-ligadas. Cada instancia de componente se ejecutará en su propio hilo. Esta es una ejecución sencilla:

```

declare P A in
{Edificación 10 2 P A}
{Send P.9 llamado}
{Send P.10 llamado}
{Send A.1 piso(4)}
{Send A.2 piso(5)}

```

El primer ascensor es enviado al piso 4 y el segundo al piso 5.

Razonando sobre el sistema de control de ascensores

Para mostrar que el componente ascensor funciona correctamente, podemos razonar sobre sus propiedades invariantes. Por ejemplo, solamente se puede recibir un mensaje `'en'(_)` cuando `Plan\=nil`. Este es un invariante sencillo que se puede probar fácilmente del hecho que los mensajes `'en'` y `paso` suceden en parejas. Por inspección es fácil ver que un mensaje `paso` se realiza siempre que el ascensor pasa a un estado donde `Plan\=nil`, y que la única transición por fuera de este estado (disparada por un mensaje `llamado`) preserva el invariante. Otro invariante es que los elementos consecutivos de un plan son diferentes siempre (¿puede probarlo?).

5.5.3. Mejoras al sistema de control de ascensores

El sistema de control de ascensores de la sección anterior es, en alguna medida, ingenuo. En esta sección indicaremos cinco maneras en que puede ser mejorado: utilizando composición de componentes para hacerlo jerárquico, mejorando cómo se abren y se cierran puertas, utilizando la negociación para encontrar el mejor ascensor para atender una solicitud, mejorando la planificación para reducir la cantidad de movimiento de los ascensores, y manejando las fallas (ascensores que dejan de funcionar). Dejamos las últimas tres mejoras como ejercicios para el lector.

Concurrencia por paso de mensajes

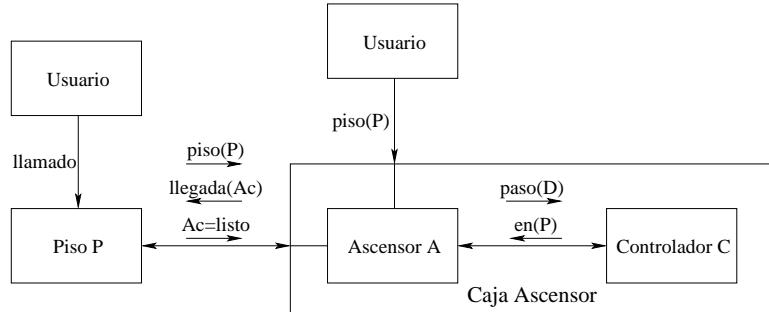


Figura 5.13: Diagrama jerárquico de componentes del sistema de control de ascensores.

Organización jerárquica

Mirando el diagrama de componentes de la figura 5.5, vemos que cada controlador se comunica únicamente con su ascensor correspondiente. Esto también se ve en la definición de `Edificación`. Esto significa que podemos mejorar la organización combinando los componentes controlador y ascensor en un componente compuesto, que llamaremos la caja del ascensor. En la figura 5.13 se muestra el diagrama de componentes actualizado con un componente caja del ascensor. Implementamos esto, definiendo el componente `CajaAscensor` así:

```

fun {CajaAscensor I estado(P S M) Pisos}
    IdC={Controlador estado(detenido P IdA)}
    IdA={Ascensor I estado(P S M) IdC Pisos}
    in IdA end

```

Entonces el procedimiento `Edificación` se puede simplificar:

```

proc {Edificación NP NA ?Pisos ?Ascensores}
    Ascensores={MakeTuple ascensores NA}
    for I in 1..NA do IdC in
        Ascensores.I={CajaAscensor I estado(1 nil false) Pisos}
    end
    Pisos={MakeTuple pisos NP}
    for I in 1..NP do
        Pisos.I={Piso I estado(nosolicitado) Ascensores}
    end
end

```

La encapsulación que provee `CajaAscensor` mejora la modularidad del programa. Podemos cambiar la organización interna de una caja de ascensor sin cambiar su interfaz.

Administración mejorada de la puerta

Nuestro sistema abre todas las puertas en un piso cuando el primer ascensor llega, y las cierra un tiempo fijo después. Entonces, ¿qué pasa si un ascensor llega a un

piso cuando las puertas están aún abiertas? Las puertas pueden estar justo a punto de cerrarse. Este comportamiento es inaceptable en un ascensor real. Necesitamos mejorar nuestro sistema de control de ascensores de manera que cada ascensor tenga su propio conjunto de puertas.

Negociación mejorada

Podemos mejorar nuestro sistema de control de ascensores de manera que el piso solicite el ascensor más cercano en lugar de solicitar uno al azar. La idea es que los pisos envíen mensajes a todos los ascensores preguntándoles cuánto tiempo estiman que les tomará llegar al piso. Entonces, el piso puede solicitar el ascensor con el menor tiempo estimado. Este es un ejemplo de un protocolo sencillo de negociación.

Planificación mejorada

Podemos mejorar la planificación del ascensor. Por ejemplo, suponga que el ascensor se está moviendo del piso 1 al piso 5 y actualmente está en el piso 2. Una solicitud del ascensor en el piso 3 debería hacer que el ascensor se detuviera en ese piso en su camino, en lugar de ir primero al piso 5 y luego devolverse al 3, como lo hace la planificación ingenua actual. El algoritmo mejorado se mueve en una dirección hasta que no hay más pisos, en ese camino, dónde detenerse; entonces, cambia de dirección. Algunas variantes de este algoritmo, denominado el algoritmo del elevador, por obvias razones, son usadas para planificar el movimiento de una cabeza de un disco duro. Con este planificador podemos tener dos botones de solicitud de ascensores, uno para solicitarlo para subir y otro para bajar.

Tolerancia a fallos

¿Qué pasa si una parte del sistema deja de funcionar? Por ejemplo, un ascensor puede estar fuera de servicio, ya sea porque está en mantenimiento, o porque se ha dañado o simplemente, porque alguien está bloqueando sus puertas en un piso en particular. Los pisos también pueden estar “fuera de servicio,” e.g., un ascensor puede tener deshabilitado el detenerse en un piso por alguna razón. Podemos extender el sistema de control de ascensores para seguir proveyendo al menos una parte del servicio en esos casos. Las ideas básicas se explican en los ejercicios (sección 5.9).

5.6. Usando directamente el modelo por paso de mensajes

El modelo por paso de mensajes se puede utilizar de otras formas diferentes a sólo la programación con objetos puerto. Una forma es programar directamente con hilos, procedimientos, puertos y variables de flujo de datos. Otra forma es utilizar otras abstracciones. En esta sección presentamos algunos ejemplos.

Concurrencia por paso de mensajes

```
proc {NuevosObjetosPuerto ?AregarObjetoPuerto ?Invocar}
    Sen P={NewPort Sen}

    proc {CicloMsjs S1 Procs}
        case S1
        of agregar(I Proc Sinc)|S2 then Procs2 in
            Procs2={AdjoinAt Procs I Proc}
            Sinc=listo
            {CicloMsjs S2 Procs2}
        [] msj(I M)|S2 then
            try {Procs.I M} catch _ then skip end
            {CicloMsjs S2 Procs}
        [] nil then skip end
    end
in
    thread {CicloMsjs Sen procs} end

proc {AregarObjetoPuerto I Proc}
    Sinc in
        {Send P agregar(I Proc Sinc)}
        {Wait Sinc}
    end

proc {Invocar I M}
    {Send P msj(I M)}
end
end
```

Figura 5.14: Definiendo objetos puerto que comparten un hilo.

5.6.1. Objetos puerto que comparten un hilo

Es posible ejecutar muchos objetos puerto en un solo hilo, si el hilo serializa todos sus mensajes. Esto puede ser más eficiente que utilizar un hilo por cada objeto puerto. De acuerdo a David Wood de Symbian Ltd., esta solución se usó en el sistema operativo de los computadores Psion Series 3 palmtop, donde la memoria es muy solicitada[188]. La ejecución es eficiente pues no se necesita ninguna planificación de hilos. Los objetos pueden acceder a los datos compartidos sin precauciones particulares pues todos los objetos se ejecutan en el mismo hilo. La principal desventaja es que la sincronización es más difícil. La ejecución no puede esperar dentro de un objeto, a que se termine un cálculo realizado por otro objeto; intentar esto bloqueará el programa. Esto significa que los programas deben escribirse en un estilo particular. El estado debe ser global, o estar almacenado en los argumentos de los mensajes, y no en los objetos. Los mensajes son una especie de continuación, i.e., no hay retorno. Cada ejecución de un objeto finaliza enviando un mensaje.

En la figura 5.14 se define la abstracción NuevosObjetosPuerto. Ésta configura

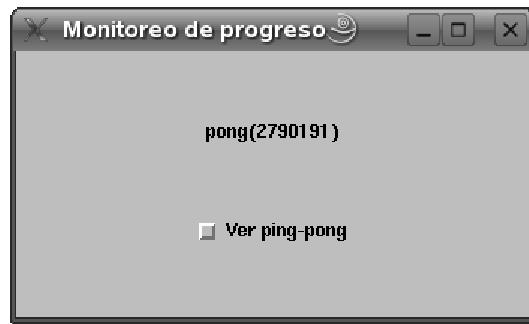


Figura 5.15: Toma en pantalla del programa Ping-Pong.

el único hilo y devuelve dos procedimientos, `AgregarObjetoPuerto` e `Invocar`:

- `{AgregarObjetoPuerto OP Proc}` agrega un objeto puerto nuevo con nombre `OP` al hilo. El nombre debe ser un literal o un número. Se puede agregar al hilo cualquier número de objetos puerto.
- `{Invocar OP Msj}` envía el mensaje `Msj` al objeto puerto `OP`, de manera asincrónica. Todas las ejecuciones de los mensajes de todos los objetos puerto se realizan en un único hilo. Las excepciones que se lancen durante la ejecución de mensajes, sencillamente, se ignoran.

Note que la abstracción almacena los procedimientos de los objetos puerto en un registro etiquetado con el átomo `procs`, y utiliza `AdjoinAt` para extender el registro cuando se agrega un objeto puerto nuevo.

El programa Ping-Pong

En la figura 5.15 se presenta una toma de pantalla de un programa concurrente pequeño, Ping-Pong, el cual utiliza objetos puerto que comparten un hilo. En la figura 5.16 se presenta el código fuente del programa Ping-Pong. Se utiliza `NewProgWindow`, un sencillo monitor de progreso definido en la sección 10.4.1 (en CTM). Inicialmente se crean dos objetos, `objping` y `objpong`. Cada objeto entiende dos mensajes, `ping(N)` y `pong(N)`. El objeto `objping` envía de manera asincrónica un mensaje `pong(N)` al objeto `objpong` y viceversa. Cada mensaje se ejecuta desplegando un texto en la ventana, continuando luego con el envío de un mensaje al otro objeto. El argumento entero `N` cuenta el número de mensajes; por eso se incrementa en cada invocación. La ejecución empieza con la invocación inicial `{Invocar objping ping(0)}`.

Cuando el programa empieza, se crea una ventana que despliega un término de la forma `ping(123)` o `pong(123)`, donde el entero muestra el valor del contador de mensajes. Esto monitorea el progreso de la ejecución. Si el botón de control está habilitado, entonces cada término se despliega por 50 ms. Pero, si el botón de

Concurrencia por paso de mensajes

```
declare AgregarObjetoPuerto Invocar
{NuevosObjetosPuerto AgregarObjetoPuerto Invocar}

MsjInfo={NewProgWindow "Ver ping-pong"}

fun {ProcPingPong Otro}
  proc {$ Msj}
    case Msj
      of ping(N) then
        {MsjInfo "ping(\"#N#\")"}
        {Invocar Otro pong(N+1)}
      [] pong(N) then
        {MsjInfo "pong(\"#N#\")"}
        {Invocar Otro ping(N+1)}
      end
    end
  end

{AgregarObjetoPuerto objping {ProcPingPong objpong}}
{AgregarObjetoPuerto objpong {ProcPingPong objping}}
{Invocar objping ping(0)}
```

Figura 5.16: El programa Ping-Pong: utilizando objetos puerto que comparten un hilo.

```
fun {ColaNueva}
  Inserción PuertoInserción={NewPort Inserción}
  Atención PuertoAtención={NewPort Atención}
in
  Inserción=Atención
  cola(ins:proc {$ X} {Send PuertoInserción X} end
        attn:proc {$ X} {Send PuertoAtención X} end)
end
```

Figura 5.17: Cola (versión ingenua con puertos).

control está deshabilitado, entonces los mensajes fluyen internamente a una rata mucho más rápida, limitada solamente por la velocidad del sistema en tiempo de ejecución de Mozart.⁹

9. Al utilizar una versión previa a la versión 1.3.0 de Mozart, sobre un procesador PowerPC a 1 GHz (PowerBook G4 con sistema operativo Mac OS X), la rata estuvo alrededor de 300000 invocaciones de métodos asincrónicos por segundo.

5.6.2. Una cola concurrente con puertos

El programa que se muestra en la figura 5.17 define un hilo que actúa como una cola FIFO. La función ColaNueva devuelve una cola nueva Q , la cual es un registro $\text{cola}(\text{ins:ProcIns } \text{atn:ProcAtn})$ que contiene dos procedimientos, uno para insertar un elemento en la cola y uno para atender (ir a traer) un elemento de la cola. La cola se implementa con dos puertos. La utilización de variables de flujo de datos vuelve la cola insensible al orden real de llegada de las solicitudes $Q.\text{atn}$ y $Q.\text{ins}$. Por ejemplo, pueden llegar solicitudes $Q.\text{atn}$ aún cuando la cola esté vacía. Para insertar un elemento x , se invoca $\{Q.\text{ins } x\}$. Para traer un elemento en y , se invoca $\{Q.\text{atn } y\}$.

El programa de la figura 5.17 es casi correcto, pero no funciona porque los flujos de los puertos son variables de sólo lectura. Para ver esto, ensaye la secuencia de declaraciones siguiente:

```
declare Q in
thread Q={ColaNueva} end
{Q.ins 1}
{Browse {Q.atn $}}
{Browse {Q.atn $}}
{Browse {Q.atn $}}
{Q.ins 2}
{Q.ins 3}
```

El problema es que `Inserción=Atención` trata de imponer la igualdad entre dos variables de sólo lectura, i.e., ligarlas. Pero una variable de sólo lectura, sólo puede ser leída, nunca ligada. Entonces el hilo que define la cola se suspenderá en la declaración `Inserción=Atención`. Podemos resolver este problema definiendo un procedimiento `Emparejar` y ejecutándolo en su propio hilo, como se muestra en la figura 5.18. Usted puede verificar que la secuencia de instrucciones anterior, funciona ahora.

Miremos un poco más de cerca para ver por qué esta versión es correcta.

```
{Q.ins I0} {Q.ins I1} ... {Q.ins In}
```

agrega incrementalmente los elementos I_0, I_1, \dots, I_n , al flujo `Inserción`, resultando lo siguiente:

```
I0|I1|...|In|F1
```

donde F_1 es una variable de sólo lectura. En la misma forma, se realizan una serie de operaciones de atención:

```
{Q.atn X0} {Q.atn X1} ... {Q.atn Xn}
```

la cual agrega los elementos X_0, X_1, \dots, X_n al flujo `Atención`, resultando en

```
X0|X1|...|Xn|F2
```

donde F_2 es otra variable de sólo lectura. La invocación `{Emparejar Inserción}`

Concurrencia por paso de mensajes

```

fun {ColaNueva}
  Inserción PuertoInserción={NewPort Inserción}
  Atención PuertoAtención={NewPort Atención}
  proc {Emparejar Xs Ys}
    case Xs # Ys
      of (X|Xr) # (Y|Yr) then
        X=Y {Emparejar Xr Yr}
        [] nil # nil then skip
      end
    end
  in
    thread {Emparejar Inserción Atención} end
    cola(ins:proc {$ x} {Send PuertoInserción x} end
         attn:proc {$ x} {Send PuertoAtención x} end)
  end

```

Figura 5.18: Cola (versión correcta con puertos).

`Atención}` liga los `xi`'s a los `ii`'s y se bloquea de nuevo para `F1=F2`.

Esta cola concurrente es completamente simétrica con respecto a la inserción y atención de elementos. Es decir, `Q.ins` y `Q.atn` se definen exactamente en la misma forma. Además, debido al uso de variables de flujo de datos para referenciar los elementos de la cola, estas operaciones nunca se bloquean. Esto le aporta a la cola la extraordinaria propiedad de que puede ser usada para insertar y recuperar elementos antes de que sean conocidos. Por ejemplo, si se invoca `{Q.atn x}` cuando no hay elementos en la cola, entonces se devuelve una variable no-ligada en `x`. El siguiente elemento que sea insertado será ligado a `x`. Para hacer una recuperación bloqueante, i.e., una que espere a que no existan elementos en la cola, la invocación a `Q.atn` debe venir seguida de un `Wait`:

```

{Q.atn x}
{Wait x}

```

Similarmente, si se invoca `{Q.ins x}` cuando `x` es no-ligada, i.e., cuando no hay un elemento para insertar, entonces la variable no-ligada `x` será colocada en la cola. Ligar `x` hará que el elemento se conozca. Para realizar una inserción sólo si se conoce el elemento, la invocación a `Q.ins` debería estar precedida de un `Wait`:

```

{Wait x}
{Q.ins x}

```

Hemos capturado así la asimetría esencial entre la inserción y la atención: está en la operación `wait`. Otra forma de ver esto es que las operaciones de inserción y atención reservan lugares en la cola. Esta reserva se realiza independientemente de que los valores de los elementos se conozcan o no.

Los lectores atentos verán que existe una solución, más sencilla aún, al problema del programa de la figura 5.17. El procedimiento `Emparejar` no es realmente necesario. Es suficiente ejecutar `Inserción=Atención` en un hilo propio. Esto funciona debido a que el algoritmo de unificación hace exactamente lo que hace

Emparejar.¹⁰

5.6.3. Una abstracción de hilo con detección de terminación

“Damas y caballeros, en poco tiempo estaremos llegando a la estación Bruselas Midi, donde este tren termina su recorrido.”

– Anuncio, Tren de alta velocidad Thalys, línea Paris–Bruselas, Enero de 2002

La creación de hilos con la declaración **thread** $\langle d \rangle$ **end** puede crear por ella misma hilos nuevos durante la ejecución de $\langle d \rangle$. Nos gustaría detectar el momento en que todos estos hilos hayan terminado. Esto no parece fácil: los hilos nuevos pueden pos sí mismos crear hilos nuevos, y así sucesivamente. Se necesita un algoritmo de detección de terminación como el de la sección 4.4.3. El algoritmo de esa sección requiere el paso explícito de variables entre los hilos. Requerimos una solución que sea encapsulada, i.e., que no contenga esta incomodidad. Para ser precisos, requerimos un procedimiento **NewThread** con las propiedades siguientes:

- La invocación $\{\text{NewThread } P \text{ SubThread}\}$ crea un hilo nuevo que ejecuta el procedimiento sin argumentos P , y devuelve SubThread , un procedimiento de un argumento.
- Durante la ejecución de P se pueden crear hilos nuevos invocando $\{\text{SubThread } P_1\}$, donde el procedimiento sin argumentos P_1 corresponde al cuerpo del hilo. Denominamos a estos hilos creados así, subhilos. SubThread se puede invocar recursivamente, es decir, dentro de P y dentro del argumento P_1 de cualquier invocación a SubThread .
- La invocación a **NewThread** termina su ejecución después que el hilo nuevo y todos sus subhilos hayan terminado las suyas.

Es decir, hay tres formas de crear un hilo nuevo:

```
thread <d> end
  {NewThread proc {$} <d> end ?SubThread}
    {SubThread proc {$} <d> end}
```

Todas tienen un comportamiento concurrente idéntico salvo en el caso de **NewThread**, cuya invocación tiene un comportamiento de terminación diferente. **NewThread** se puede definir utilizando el modelo por paso de mensajes tal como se muestra en la figura 5.19. Esta definición utiliza un puerto. Cuando se crea un subhilo, se envía $+1$ al puerto, y cuando se termina la ejecución de un subhilo, se envía -1 al puerto. El procedimiento **TerminarEnCero** acumula el número total de subhilos en ejecución. Si este número es cero, entonces todos los subhilos han terminado y, por tanto, **TerminarEnCero** termina.

Podemos probar que esta definición es correcta utilizando afirmaciones invariantes. Considere la afirmación siguiente: “La suma de los elementos en el flujo del puerto es mayor o igual al número de hilos activos.” Si esta suma es cero, entonces

10. Este diseño de la cola FIFO fue presentado primero por Denys Duchier.

Concurrencia por paso de mensajes

```
local
  proc {TerminarEnCero N Is}
    case Is of I|Ir then
      if N+I\=0 then {TerminarEnCero N+I Ir} end
    end
  end
in
  proc {NewThread P ?SubThread}
    Is Pt={NewPort Is}
  in
    proc {SubThread P}
      {Send Pt 1}
      thread
        {P} {Send Pt ~1}
      end
    end
    {SubThread P}
    {TerminarEnCero 0 Is}
  end
end
```

Figura 5.19: Una abstracción de hilo con detección de terminación.

la afirmación implica que el número de hilos activos es cero también. Podemos usar inducción para mostrar que la afirmación es cierta en cada parte de cada ejecución posible, empezando desde la invocación a `NewThread`. Esto es claramente cierto cuando comienza `NewThread`, pues ambos números son cero. Durante una ejecución hay cuatro acciones relevantes: enviar `+1`, enviar `-1`, el comienzo de un hilo, y la terminación de un hilo. Una inspección al programa permite ver que estas acciones mantienen cierta la afirmación. (Podemos suponer, sin pérdida de generalidad, que la terminación de un hilo se produce justo antes de que se envíe `-1` al puerto, pues en ese momento el hilo ya no ejecuta ninguna parte del programa del usuario.)

Esta definición de `NewThread` tiene dos restricciones. Primero, `P` y `P1` siempre deben invocar `SubThread` para crear hilos, nunca ninguna otra operación (tales como `thread ... end` o un `SubThread` creado en otra parte). Segundo, `SubThread` no debería ser invocado desde ninguna otra parte del programa. La definición puede ser extendida para relajar estas restricciones o comprobarlas. Dejamos estas tareas como ejercicio para el lector.

Semántica del envío de mensajes y sistemas distribuidos

Sabemos que la operación `Send` es asíncrona, i.e., se completa inmediatamente. El algoritmo de detección de terminación se basa en otra propiedad de `Send`: que `{Send Pt 1}` (en el hilo padre) llegue antes que `{Send Pt ~1}` (en el hilo hijo). ¿Podemos suponer que envíos de mensajes en hilos diferentes se comportan de esta manera? Sí podemos, si nos aseguramos que la operación `Send` reserva un espacio

en el flujo del puerto. Revisemos la semántica que definimos para los puertos al principio del capítulo: la operación `Send` coloca, de hecho, su argumento en el flujo del puerto. A esto lo denominamos la semántica de reserva de espacio de `Send`.¹¹

Desafortunadamente, esta semántica no es la correcta en general. Realmente deseamos un semántica de reserva de espacio *final*, donde la operación `Send` pudiera no reservar inmediatamente el espacio, pero asegurarnos que lo hará finalmente. ¿Por qué esta semántica es “correcta”? Porque ese es el comportamiento natural de un sistema distribuido, donde un programa es diseminado en muchos procesos y los procesos pueden estar en máquinas diferentes. Una operación `Send` puede ejecutarse en un proceso diferente a donde está construido el flujo del puerto. ¡Entonces, realizar una operación `Send` no reserva inmediatamente un espacio porque ese espacio podría estar en una máquina diferente (recuerde que la operación `Send` debe completarse inmediatamente)! Todo lo que podemos decir es que realizar una operación `Send` finalmente reservará un espacio.

Con la semántica “correcta” para `Send`, nuestro algoritmo de terminación es incorrecto pues `{Send Pt ~1}` puede llegar antes que `{Send Pt 1}`. Podemos resolver este problema definiendo un puerto que reserva el espacio en términos de un puerto que reserva finalmente el espacio:

```
proc {NewSport ?S ?SSend}
S1 P={NewPort S1} in
  proc {SSend M} X in {Send P M#X} {Wait X} end
    thread S={Map S1 fun {$ M#X} X=listo M end} end
  end
```

`NewSport` se comporta como `NewPort`. Si `NewPort` define un puerto que reserva finalmente el espacio, entonces `NewSport` definirá un puerto que reserva el espacio. El utilizar `NewSport` en el algoritmo de detección de terminación asegurará que sea correcto en el caso en que usemos la semántica “correcta” de `Send`.

5.6.4. Eliminación de dependencias secuenciales

Examinemos cómo eliminar dependencias secuenciales innútiles entre diferentes partes de un programa. Tomamos como ejemplo el procedimiento `{Filter L F L2}`, el cual toma una lista `L` y una función booleana de un argumento `F`, y devuelve una lista `L2` que contiene los elementos `x` de `L` para los cuales `{F x}` es `true`. Esta es una función de la biblioteca (es parte del módulo `List`) que se puede definir declarativamente así:

11. Algunas veces esto se denomina `Send` sincrónico, porque sólo se completa cuando el mensaje es entregado al flujo. Evitaremos este término porque el concepto de “entrega” no está claro. Por ejemplo, podríamos querer hablar de entrega de un mensaje a una aplicación (un proceso) en lugar de un flujo.

```

proc {ConcFilter L F ?L2}
  Send Close
in
  {NewPortClose L2 Send Close}
  {Barrera
    {Map L
      fun {$ X}
        proc {$}
          if {F X} then {Send X} end
        end
      end}}
  {Close}
end

```

Figura 5.20: Un filtro concurrente sin dependencias secuenciales.

```

fun {Filter L F}
  case L
  of nil then nil
  [] X|L2 then
    if {F X} then X|{Filter L2 F} else {Filter L2 F} end
  end
end

```

o, de manera equivalente, utilizando la sintaxis de ciclo:

```

fun {Filter L F}
  for X in L collect:C do
    if {F X} then {C X} end
  end
end

```

Esta definición es eficiente, pero introduce dependencias secuenciales: $\{F X\}$ sólo puede ser calculada después de que haya sido calculada para todos los elementos de L que se encuentran antes que x en la lista. Estas dependencias aparecen porque todos los cálculos se realizan secuencialmente en el mismo hilo. Pero estas dependencias no son realmente necesarias. Por ejemplo, en la invocación

```
{Filter [A 5 1 B 4 0 6] fun {$ X} X>2 end Sal}
```

es posible deducir inmediatamente que 5, 4, y 6 estarán en la salida, sin esperar a que A y B sean ligadas. Más adelante, si algún hilo hace $A=10$, entonces 10 podría ser agregado inmediatamente al resultado.

Podemos escribir una versión nueva de `Filter` que evite esas dependencias. Esta versión construye la salida incrementalmente, a medida que la información de entrada va llegando. Utilizamos dos conceptos fundamentales:

- La composición concurrente (ver sección 4.4.3). El procedimiento `Barrera` implementa la composición concurrente: crea una tarea concurrente para cada elemento de la lista y espera a que se terminen.

- Los canales asincrónicos (puertos; ver atrás en este capítulo). El procedimiento `NewPortClose` implementa un puerto con una operación de envío de mensajes y una de cierre del puerto. Su definición se presenta en el archivo suplementario en el sitio Web del libro. La operación de cierre termina el flujo del puerto con `nil`.

En la figura 5.20 se presenta la definición. Primero se crea un puerto cuyo flujo es la lista de salida. Luego, se invoca `Barrera` con una lista de procedimientos, cada uno de los cuales agrega `x` a la salida de la lista si `{F X}` es `true`. Finalmente, cuando todos los elementos de la lista han sido tenidos en cuenta, la lista de salida se finaliza cerrando el puerto.

¿`ConcFilter` es declarativo? Tal como está escrito, ciertamente que no, pues la lista de salida puede aparecer en cualquier orden (dando lugar a un no-determinismo observable). Se puede hacer declarativo evitando su no-determinismo, e.g., ordenando la lista de salida. Hay otra forma, utilizando las propiedades de abstracción de datos. Si el resto del programa no depende del orden (e.g., la lista representa una estructura de datos tipo conjunto), entonces `ConcFilter` se puede tratar como si fuera declarativo. Esto es fácil de ver: si la lista estuviera en efecto oculta bajo una abstracción de conjunto, entonces `ConcFilter` sería determinístico y declarativo.

5.7. El lenguaje Erlang

El lenguaje Erlang fue desarrollado por Ericsson para aplicaciones en telecomunicaciones, en particular para telefonía [9, 184]. Su implementación, la Ericsson OTP (Open Telecom Platform), se caracteriza por una concurrencia de granularidad fina (hilos eficientes), confiabilidad extrema (*software* de alto desempeño con tolerancia a fallos), y capacidad de reemplazo de código en caliente (actualización del *software* cuando el sistema está en ejecución). Erlang es un lenguaje de alto nivel que oculta la representación interna de los datos y administra automáticamente la memoria. Este lenguaje ha sido usado exitosamente en varios productos de Ericsson.

5.7.1. Modelo de computación

El modelo de computación de Erlang tiene una estructura elegante por capas. Primero explicaremos el modelo y luego mostraremos cómo se extiende para soportar distribución y tolerancia a fallos.

El modelo de computación de Erlang consiste de entidades concurrentes llamadas “procesos.” Un proceso consiste de un objeto puerto y un buzón de correo. El lenguaje se puede dividir en dos capas:

- *Núcleo funcional*. Los procesos se programan en un lenguaje funcional estricta y dinámicamente tipado. Cada proceso contiene un objeto puerto definido por una función recursiva. Un proceso que genera un proceso nuevo, especifica cuál función debe ejecutarse inicialmente dentro de él.

Concurrencia por paso de mensajes

- *Extensión para el paso de mensajes.* Los procesos se comunican enviando mensajes, de manera asincrónica, a otros procesos, en un orden FIFO. Los mensajes se colocan en el buzón de los procesos de destino. Los mensajes pueden contener cualquier valor, incluidos valores de tipo función. Cada proceso tiene un identificador de proceso, su PID, el cual es una constante única que identifica el proceso. El PID puede ser embebido en estructuras de datos y en mensajes. Un proceso puede leer mensajes de su buzón. La recepción de los mensajes puede ser bloqueante o no bloqueante. Los procesos receptores utilizan reconocimiento de patrones para esperar y para eliminar mensajes de su buzón, sin interrumpir los otros mensajes. Esto significa que los mensajes no se atienden necesariamente en el orden en que son enviados.

Una propiedad importante de los procesos en Erlang, es que son independientes por defecto. Es decir, lo que pasa en un proceso no tiene efectos sobre los otros procesos, a menos que el efecto haya sido programado explícitamente. Esto implica que los mensajes enviados entre los procesos se copian. No existen, nunca, referencias compartidas entre procesos. Esta es también una razón poderosa para que la primitiva de comunicación sea el envío (de mensajes) asincrónico. La comunicación sincrónica crea una dependencia, pues el proceso remitente espera por una respuesta del proceso de destino. La independencia de procesos facilita la construcción de sistemas altamente confiables [8].

Extensiones para distribución y tolerancia a fallos

El modelo centralizado se extiende para distribución y tolerancia a fallos:

- *Distribución transparente.* Los procesos pueden estar en la misma máquina o en diferentes máquinas. Un ambiente de máquina individual se denomina un nodo en la terminología de Erlang. En un programa, la comunicación entre objetos locales o remotos se escribe exactamente de la misma manera. El PID encapsula el destino y permite al sistema decidir, en tiempo de ejecución, si debe hacer una operación local o remota. Los procesos son estacionarios; esto significa que una vez se crea un proceso en un nodo, el proceso permanece allí por todo su tiempo de vida. El envío de un mensaje a un proceso remoto necesita exactamente de una operación de red, i.e., no hay nodos intermedios involucrados. Los procesos se pueden crear en nodos remotos. Los programas son transparentes a la red, i.e., producen el mismo resultado sin importar en qué nodo se coloquen los procesos. Los programas tienen conciencia de la red pues el programador tiene completo control sobre la localización de los procesos y puede optimizarla de acuerdo a las características de la red.
- *Detección de fallos.* Un proceso puede configurarse para realizar una acción cuando otro proceso falle. En la terminología de Erlang esto se denomina *enlazar* los dos procesos. Una posible acción es que cuando el segundo proceso falle, se envíe un mensaje al primero. Esta capacidad de detectar fallos permite que muchos mecanismos de tolerancia a fallos se programen completamente en Erlang. Como los procesos son independientes, este estilo de programación es sencillo.

- *Persistencia.* El sistema en tiempo de ejecución de Erlang viene con una base de datos, llamada Mnesia, que ayuda a construir aplicaciones altamente disponibles. Mnesia es replicada para lograr alta disponibilidad.

En resumen, podemos decir que el modelo de computación de Erlang, i.e., objetos puero independientes, está fuertemente optimizado para construir sistemas distribuidos tolerantes a fallos. La base de datos Mnesia compensa la no existencia de un almacén mutable general. Un ejemplo de un producto exitoso construido usando Erlang es el switch ATM AXD 301, de Ericsson, el cual provee telefonía sobre una red ATM. El switch AXD 301 maneja de 30 a 40 millones de llamadas por semana con una disponibilidad de 99.999999% de acuerdo a Ericsson (aproximadamente 30 ms de tiempo no disponible por año) y contiene 1.7 millones de líneas de Erlang [7, 8].

5.7.2. Introducción a la programación en Erlang

Para tener una primera impresión de Erlang, presentamos algunos programas pequeños en Erlang y mostramos cómo hacer lo mismo en los modelos de computación del libro. Los programas son tomados, en su mayoría, del libro de Erlang [9]. Mostraremos cómo escribir funciones y programas concurrentes por paso de mensajes. Para mayor información sobre la programación en Erlang, reomendamos indudablemente el libro de Erlang.

Una función sencilla

El núcleo de Erlang es un lenguaje funcional estricto con tipamiento dinámico. A continuación presentamos una definición sencilla de la función factorial:

```
factorial(0) -> 1;  
factorial(N) when N>0 -> N*factorial(N-1).
```

Este ejemplo introduce las convenciones sintácticas básicas de Erlang. Los nombres de funciones se escriben en minúsculas y los identificadores de variables con Mayúsculas. Los identificadores de variables son ligados a valores cuando se definen, lo cual significa que Erlang tiene un almacén de valores. Una ligadura de un identificador no se puede cambiar; es asignación única tal y como sucede en el modelo declarativo. Estas convenciones son heredadas de Prolog, lenguaje en el que se escribió la primera implementación (un intérprete) de Erlang.

Las funciones en Erlang se definen con cláusulas; una cláusula tiene una cabeza (con un patrón y una guarda opcional) y un cuerpo. Los patrones se comprueban en orden, comenzando por la primera cláusula. Si se reconoce un patrón, sus variables se ligan y el cuerpo de la cláusula se ejecuta. La guarda opcional es una función booleana que tiene que devolver `true`. Todos los identificadores de variables del patrón deben ser diferentes. Si un patrón no es reconocido, entonces se ensaya con la cláusula siguiente. Podemos traducir la función factorial al modelo declarativo de la forma siguiente:

Concurrencia por paso de mensajes

```
fun {Factorial N}
    case N
    of 0 then 1
    [] N andthen N>0 then N*{Factorial N-1}
    end
end
```

La declaración **case** realiza el reconocimiento de patrones exactamente como lo hace Erlang, con una sintaxis diferente.

Reconocimiento de patrones con tuplas

La siguiente es una función que realiza reconocimiento de patrones con tuplas:

```
área({cuadrado, Lado}) ->
    Lado*Lado;
área({rectángulo, X, Y}) ->
    X*Y;
área({círculo, Radio}) ->
    3.14159*Radio*Radio;
área({triángulo, A, B, C}) ->
    S=(A+B+C)/2;
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

Aquí se utiliza la función raíz cuadrada `sqrt` definida en el módulo `math`. Esta función calcula el área de una forma plana. La forma es representada por medio de una tupla que identifica la forma y provee su tamaño. Las tuplas en Erlang se escriben con corchetes: `{cuadrado, Lado}` se escribiría `cuadrado(Lado)` en el modelo declarativo. En el modelo declarativo, la función se puede escribir así:

```
fun {Área T}
    case T
    of cuadrado(Lado) then Lado*Lado
    [] rectángulo(X Y) then X*Y
    [] círculo(Radio) then 3.14159*Radio*Radio
    [] triángulo(A B C) then S=(A+B+C)/2.0 in
        {Sqrt S*(S-A)*(S-B)*(S-C)}
    end
end
```

Concurrencia y paso de mensajes

En Erlang, los hilos se crean junto con un objeto puerto y un buzón. Esta combinación se denomina un proceso. Hay tres primitivas:

1. La operación `spawn` (se escribe `spawn(M,F,A)`) crea un proceso nuevo y devuelve un valor (denominado “identificador de proceso”) que se puede usar para enviarle mensajes al proceso mismo. Los argumentos de `spawn` proveen la función de invocación para iniciar el proceso, identificada por el módulo `M`, el nombre de función `F`, y una lista `A`. Note que los módulos son entidades de primera clase en Erlang.

2. La operación `send` (se escribe `Pid!Msj`), envía de manera asincrónica el mensaje `Msj` al proceso con identificador `Pid`. Los mensajes se colocan en el buzón del receptor.

3. La operación `receive` se utiliza para eliminar mensajes del buzón. Esta operación utiliza reconocimiento de patrones para eliminar los mensajes que cumplen un patrón dado. En la sección 5.7.3 se explica esto en detalle.

Tomemos la función `área` y coloquémosla dentro de un proceso. Aquí la colocamos dentro de un servidor que puede ser invocado por cualquier otro proceso.

```
-module(servidorárea).  
-export([inicio/0, ciclo/0]).
```

```
inicio() -> spawn(servidorárea, ciclo, []).
```

```
ciclo() ->  
receive  
    {De, Forma} ->  
        De!área(Forma),  
        ciclo()  
end.
```

Aquí se definen las dos operaciones `inicio` y `ciclo` en el módulo nuevo `servidorárea`. Estas dos operaciones se exportan al exterior del módulo. Es necesario definirlas al interior de un módulo porque la operación `spawn` requiere el nombre del módulo como un argumento. La operación `ciclo` lee repetidamente un mensaje (una tupla de dos argumentos `{De, Forma}`) y lo responde invocando `área` y enviando la respuesta al proceso `De`. Ahora, iniciemos un servidor nuevo e invoquémoslo:

```
Pid=servidorárea:inicio(),  
Pid!{self(), {cuadrado, 3.4}},  
receive  
    Res -> ...  
end,
```

Aquí `self()` es una operación del lenguaje que devuelve el identificador de proceso correspondiente al proceso actual. Esto permite al servidor devolver una respuesta. Escribimos esto mismo en el modelo concurrente con estado, así:

```
fun {Inicio}  
S ServidorÁrea={NewPort S} in  
    thread  
        for msj(Res Forma) in S do  
            Res={Área Forma}  
        end  
    end  
    ServidorÁrea  
end
```

Así se inicia un nuevo servidor y se invoca:

Concurrencia por paso de mensajes

```
Pid={Inicio}
local Res in
    {Send Pid msj(Res cuadrado(3.4)) }
    {Wait Res}
    ...
end
```

En este ejemplo se usa la variable de flujo de datos `Res` para colocar la respuesta. Así se simula el envío de mensaje a través de `De` que realiza Erlang. Para hacer exáctamente lo que hace Erlang, necesitamos traducir la operación `receive` en un modelo de computación del libro. Esto es un poco más complicado y se explica en la sección siguiente.

5.7.3. La operación `receive`

Gran parte del sabor único y de la expresividad de la programación concurrente en Erlang se debe a los buzones y a la forma como se administran. Los mensajes se sacan de los buzones con la operación `receive`. Se utiliza reconocimiento de patrones para tomar los mensajes deseados, dejando los otros mensajes sin cambios. Al utilizar `receive` se produce un código particularmente compacto, legible, y eficiente. En esta sección, implementamos `receive` como una abstracción lingüística. Mostramos cómo traducirla en los modelos de computación del libro. Existen dos razones para realizar la traducción. Primero, se provee una semántica precisa para `receive`, lo cual ayuda en la comprensión de Erlang. Segundo, se muestra cómo hacer programación al estilo Erlang en Oz.

Debido al núcleo funcional de Erlang, `receive` es una expresión que devuelve un valor. La expresión `receive` tiene la forma [9] general siguiente:

```
receive
    Patrón1 [when Guarda1] -> Cuerpo1;
    ...
    PatrónN [when GuardaN] -> CuerpoN;
    [ after Expr -> CuerpoT; ]
end
```

las guardas (cláusulas `when`) y el tiempo de espera (cláusula `after`) son opcionales. Esta expresión se bloquea hasta que llegue un mensaje al buzón del hilo actual que cumpla uno de los patrones. En ese momento se elimina ese mensaje, se ligan las variables correspondientes del patrón, y se ejecuta el cuerpo. Los patrones son muy similares a los patrones de la declaración `case`: ellos introducen variables nuevas de asignación única cuyo alcance es el cuerpo correspondiente. Por ejemplo, el patrón de Erlang `{rectángulo, [X,Y]}` corresponde al patrón `rectángulo([X Y])` de Oz. Los identificadores que comienzan con letra minúscula corresponden a átomos, y los que comienzan con letra mayúscula corresponden a variables, tal y como sucede en la notación del libro. Los términos compuestos se encierran entre los corchetes `{ y }` y corresponden a tuplas.

La cláusula opcional `after` define un tiempo de espera; si no llega ningún mensaje

```

 $T(\text{receive} \dots \text{end } \text{Sen } \text{Ssal}) \equiv$ 
  local
    fun {Ciclo S T#E Ssal}
      case S of M|S1 then
        case M
          of T(Patrón1) then E=S1 T(Cuerpo1 T Ssal)
          ...
          [] T(PatrónN) then E=S1 T(CuerpoN T Ssal)
          else E1 in E=M|E1 {Ciclo S1 T#E1 Ssal}
        end
      end
    end T
  in
    {Ciclo Sen T#T Ssal}
  end

```

Figura 5.21: Traducción de `receive` sin tiempo de espera.

que cumpla un patrón, después de un número de milisegundos especificado por el resultado de la evaluación de la expresión `Expr`, entonces se ejecuta el cuerpo de la cláusula `after`. Si se han especificado cero milisegundos y no hay mensajes en el buzón, entonces la cláusula `after` se ejecuta inmediatamente.

Comentarios generales

Cada proceso de Erlang se traduce en un hilo con un puerto. Enviar un mensaje al proceso significa enviar un mensaje al puerto; esta operación agrega el mensaje al flujo del puerto, el cual representa el contenido del buzón. Todas las formas de la operación `receive`, una vez terminadas, o sacan exactamente un mensaje del buzón o dejan el buzón sin cambios. Modelamos esto asociando a cada traducción de `receive` un flujo de entrada y un flujo de salida. Todas las traducciones tienen dos argumentos, `Sen` y `Ssal`, que referencian los flujos de entrada y de salida, respectivamente. Estos flujos no aparecen en la sintaxis de Erlang. Después de ejecutar una operación `receive`, hay dos posibilidades para el valor del flujo de salida. O es el mismo flujo de entrada o tiene un mensaje menos que éste. Lo último sucede si el mensaje cumple algún patrón.

Distinguimos tres diferentes formas de `receive` que llevan a tres traducciones diferentes. En cada forma, la traducción se puede insertar directamente en el programa y se comportará como el respectivo `receive`. La primera forma se traduce usando el modelo declarativo. La segunda forma tiene un tiempo de espera; en ésta se utiliza el modelo concurrente no-determinístico (ver sección 8.2). La tercera forma es un caso particular de la segunda, donde el retraso es cero, lo cual hace que la traducción sea más sencilla.

Concurrencia por paso de mensajes

```

 $T(\text{receive} \dots \text{end} \text{ Sen } \text{Ssal}) \equiv$ 
  local
    Cancel = {Alarm  $T(\text{Expr})$ }
    fun {Ciclo S T#E Ssal}
      if {WaitTwo S Cancel} == 1 then
        case S of M|S1 then
          case M
            of  $T(\text{Patrón}1)$  then E=S1  $T(\text{Cuerpo}1 \text{ T } \text{Ssal})$ 
            ...
            []  $T(\text{Patrón}N)$  then E=S1  $T(\text{Cuerpo}N \text{ T } \text{Ssal})$ 
            else E1 in E=M|E1 {Ciclo S1 T#E1 Ssal} end
          end
        else E=S  $T(\text{Cuerpo}T \text{ T } \text{Ssal})$  end
      end T
    in
      {Ciclo Sen T#T Ssal}
    end

```

Figura 5.22: Traducción de `receive` con tiempo de espera.

Primera forma (sin tiempo de espera)

La primera forma de la expresión `receive` es:

```

receive
  Patrón1 -> Cuerpo1;
  ...
  PatrónN -> CuerpoN;
end

```

La operación se bloquea hasta que llegue un mensaje que cumpla uno de los patrones. Los patrones se comprueban en orden del Patrón1 al PatrónN. Dejamos por fuera las guardas para evitar atiborrar el código. Agregarlas es sencillo. Un patrón puede ser cualquier valor parcial; en particular una variable no-ligada siempre cumplirá un patrón. Los mensajes que no cumplen ningún patrón se colocan en el flujo de salida y no hacen que la operación `receive` se termine.

En la figura 5.21 se presenta la traducción de la primera forma, la cual escribiremos $T(\text{receive} \dots \text{end} \text{ Sen } \text{Ssal})$. El flujo de salida contiene los mensajes que permanecen después de que la expresión `receive` ha eliminado el mensaje que ella necesitaba. Note que la traducción $T(\text{Cuerpo} \text{ T } \text{Ssal})$ de un cuerpo que *no* contiene una expresión `receive` debe ligar $\text{Ssal}=\text{T}$.

La función Ciclo se usa para manejar la recepción de los mensajes no esperados: si se recibe un mensaje M que no cumple ningún patrón, entonces se coloca en el flujo de salida y se invoca a Ciclo recursivamente. Ciclo utiliza una lista de diferencias para manejar el caso en que una expresión `receive` contenga una expresión `receive`.

```

 $T(\text{receive} \dots \text{end} \text{ Sen} \text{ Ssal}) \equiv$ 
  if {IsDet Sen} then
    case Sen of M|S1 then
      case M
        of T(Patrón1) then  $T(\text{Cuerpo1} \text{ S1} \text{ Ssal})$ 
        ...
        [] T(PatrónN) then  $T(\text{CuerpoN}) \text{ S1} \text{ Ssal}$ 
        else  $T(\text{CuerpoT} \text{ Sen} \text{ Ssal})$  end
      end
    else Ssal=Sen end
  
```

Figura 5.23: Traducción de `receive` con tiempo de espera cero..

Segunda forma (con tiempo de espera)

La segunda forma de la expresión `receive` es:

```

receive
  Patrón1 -> Cuerpo1;
  ...
  PatrónN -> CuerpoN;
  after Expr -> CuerpoT;
end

```

Cuando la expresión `receive` se ejecuta, primero se evalúa `Expr`, dando como resultado el entero n . Si no se ha reconocido ningún mensaje después de n ms, entonces se ejecuta la acción asociada al tiempo de espera. En la figura 5.22 se presenta la traducción.

La traducción utiliza una interrupción de tiempo implementada con `Alarm` y `WaitTwo`. `{Alarm N}`, explicada en la sección 4.6, garantiza esperar al menos n ms y luego liga la variable no-ligada `Cancel` a `unit`. `{WaitTwo S Cancel}`, la cual se define en el archivo suplementario en el sitio Web del libro, espera de manera simultánea por uno de los dos eventos: un mensaje (`S` es ligada) y un tiempo de espera (`Cancel` es ligada). Puede devolver 1 si el primer argumento está ligado y 2 si su segundo argumento está ligado.

La semántica de Erlang es ligeramente más complicada que como se definió en la figura 5.22. Se debe garantizar que el buzón se comprueba al menos una vez, aún si el tiempo de espera es cero o ha expirado en el momento en que el buzón está siendo comprobado. Podemos implementar esta garantía estipulando que `WaitTwo` favorece su primer argumento, i.e., siempre devuelve 1 si su primer argumento se determina. La semántica de Erlang también garantiza que la operación `receive` termina rápidamente una vez el tiempo de espera expira. Aunque esto se garantiza fácilmente por la actual implementación, no es garantizado para la figura 5.22, pues `Ciclo` puede quedarse dando vueltas para siempre, si los mensajes llegan más rápido que la velocidad a la que itera el ciclo. Dejamos al lector modificar la figura 5.22 para agregar esta garantía.

Tercera forma (con tiempo de espera cero)

La tercera forma de la expresión `receive` es como la segunda salvo que el tiempo de espera es cero. Sin retraso, la operación `receive` no se bloquea. Se puede hacer una traducción más sencilla comparada con el caso de tiempo de espera diferente de cero. En la figura 5.23 se presenta la traducción. Por medio de `IsDet`, primero se comprueba si existe un mensaje que cumpla alguno de los patrones. `{IsDet S}`, explicada en la sección 4.9.3, comprueba inmediatamente si `S` está ligada o no y devuelve `true` o `false`. Si ningún mensaje concuerda con ningún patrón (e.g., si el buzón está vacío), entonces se ejecuta la acción por defecto `CuerpoT`.

5.8. Tema avanzado**5.8.1. El modelo concurrente no-determinístico**

En esta sección se explica el modelo concurrente no-determinístico, el cual, en términos de su expresividad, se sitúa en un lugar intermedio entre el modelo concurrente declarativo y el modelo concurrente por paso de mensajes. El modelo concurrente no-determinístico es menos expresivo que el modelo por paso de mensajes, pero a cambio cuenta con una semántica lógica (ver capítulo 9 (en CTM)).

El modelo concurrente no-determinístico es el modelo utilizado por la programación lógica concurrente [158]. Algunas veces es llamado el modelo de procesos de la programación lógica, pues modela los predicados como computaciones concurrentes. Es un modelo interesante, tanto por razones históricas, como por los elementos que ofrece para comprender la programación concurrente en la práctica. Primero introducimos el modelo concurrente no determinístico y mostramos cómo resolver con él, el problema de comunicación por flujos de la sección 4.8.3. Luego mostramos cómo implementar la escogencia no-determinística en el modelo concurrente declarativo con excepciones, mostrando que este último es al menos tan expresivo como el modelo no-determinístico.

En la tabla 5.2 se presenta el lenguaje núcleo del modelo concurrente no-determinístico. Se añade sólo una operación al modelo concurrente declarativo: una escogencia no-determinística que espera por la ocurrencia de uno de dos eventos, y de manera no-determinística devuelve cuándo ha sucedido uno de los dos con una indicación de cuál.

Limitación del modelo concurrente declarativo

En la sección 4.8.3 vimos una limitación fundamental del modelo concurrente declarativo: los objetos flujo deben leer los flujos de entrada en un patrón fijo. Un objeto flujo no puede ser alimentado independientemente por dos flujos. ¿Cómo podemos resolver este problema? Considere el caso de dos objetos cliente y un objeto servidor. Podemos tratar de resolverlo colocando un nuevo objeto flujo, un

$\langle d \rangle ::=$	
skip	Declaración vacía
$\langle d \rangle_1 \langle d \rangle_2$	Declaración de secuencia
local $\langle x \rangle$ in $\langle d \rangle$ end	Creación de variable
$\langle x \rangle_1 = \langle x \rangle_2$	Ligadura variable-variable
$\langle x \rangle = \langle v \rangle$	Creación de valor
if $\langle x \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Condicional
case $\langle x \rangle$ of $\langle \text{patrón} \rangle$ then $\langle d \rangle_1$ else $\langle d \rangle_2$ end	Reconocimiento de patrones
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Invocación de procedimiento
thread $\langle d \rangle$ end	Creación de hilo
{WaitTwo $\langle x \rangle \langle y \rangle \langle z \rangle \}$	Escogencia no-determinística

Tabla 5.2: El lenguaje núcleo concurrente no-determinístico.

mezclador de flujos, entre los dos clientes y el servidor. El mezclador de flujos tendría dos flujos de entrada y uno de salida. Todos los mensajes que llegaran a cada uno de los flujos de entrada se colocarían en el flujo de salida. En la figura 5.24 se ilustra la solución. Esto parece resolver nuestro problema: cada cliente envía mensajes al mezclador de flujos, y el mezclador de flujos los reenvía al servidor. El mezclador de flujos se puede definir así:

```
fun {MezcladorFlujos SSa11 SSa12}
    case SSa11#SSa12
        of (M|NuevoS1)#SSa12 then
            M|{MezcladorFlujos NuevoS1 SSa12}
        [] SSa11#(M|NuevoS2) then
            M|{MezcladorFlujos SSa11 NuevoS2}
        [] nil#SSa12 then
            SSa12
        [] SSa11#nil then
            SSa11
        end
    end
```

El mezclador de flujos se ejecuta en su propio hilo. Esta definición maneja el caso de terminación, i.e., cuando alguno de los clientes, o ambos, termina. Esta solución tiene todavía una dificultad básica: ¡no funciona! ¿Por qué no? Piénselo cuidadosamente antes de leer la respuesta en la nota de pie de página.¹²

12. No funciona porque la declaración **case** comprueba sólo un patrón a la vez, y sólo va al siguiente patrón cuando el anterior ha fallado. Mientras esté esperando algo por el flujo SSa11, no puede aceptar una entrada por el flujo SSa12, y viceversa.

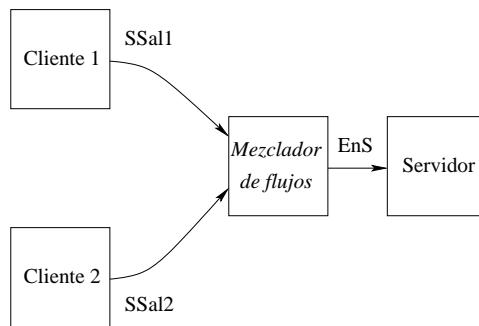
Concurrencia por paso de mensajes

Figura 5.24: Conectando dos clientes utilizando un mezclador de flujos.

Agregando escogencia no-determinística

Pero esta solución fracasada tiene las semillas de una solución exitosa. El problema es que la declaración **case** sólo espera por *una* condición a la vez. Por tanto, una posible solución es extender el modelo concurrente declarativo con una operación que permita esperar concurrentemente sobre más de una condición. Denominamos esta operación escogencia no-determinística. Una de las maneras más sencillas consisten en agregar una operación que espera concurrentemente la ligadura de una de dos variables de flujo de datos. Denominamos esta operación **WaitTwo** porque generaliza la operación **wait**. La invocación a esta operación `{WaitTwo A B}` devuelve 1 o 2 cuando A o B sean ligados. Puede devolver 1 cuando A sea ligado y 2 cuando lo sea B. Una definición sencilla en Mozart se presenta en el archivo de suplementos del sitio Web del libro. El modelo concurrente declarativo extendido con **WaitTwo** se denomina el modelo concurrente no determinístico.

Programación lógica concurrente

El modelo concurrente no-determinístico es el modelo básico de la programación lógica concurrente, tal como fue desarrollada desde sus inicios por IC-Prolog, Parlog, Concurrent Prolog, FCP (Flat Concurrent Prolog), GHC (Guarded Horn Clauses), y Flat GHC [32, 33, 34, 156, 157, 173]. Además, este modelo es el modelo principal usado por el proyecto japonés de quinta generación y muchos otros proyectos substanciales en los años 1980 [53, 158, 172]. En el modelo concurrente no determinístico es posible escribir un mezclador de flujos. Su definición tiene el siguiente aspecto:

```
fun {MezcladorFlujos SSal1 SSal2}
    F={WaitTwo SSal1 SSal2}
in
    case F#SSal1#SSal2
    of 1#(M|NewS1)#SSal2 then
        M|{MezcladorFlujos SSal2 NuevoS1}
    [] 2#SSal1#(M|NewS2) then
        M|{MezcladorFlujos NuevoS2 SSal1}
    [] 1#nil#SSal2 then
        SSal2
    [] 2#SSal1#nil then
        SSal1
    end
end
```

Este estilo de programación es exáctamente lo que se hace con la programación lógica concurrente. Una sintaxis típica para esta definición, en un lenguaje lógico concurrente al estilo Prolog, sería la siguiente:

```
mezcladorFlujos([M|NuevoS1], SSal2, EnS) :- true |
    EnS=[M|NuevoS],
    mezcladorFlujos(SSal2, NuevoS1, NuevoS).
mezcladorFlujos(SSal1, [M|NuevoS2], EnS) :- true |
    EnS=[M|NuevoS],
    mezcladorFlujos(NuevoS2, SSal1, NuevoS).
mezcladorFlujos([], SSal2, EnS) :- true |
    EnS=SSal2.
mezcladorFlujos(SSal1, [], EnS) :- true |
    EnS=SSal1.
```

Esta definición consiste de cuatro cláusulas, cada una de las cuales define una escogencia no-determinística. Recuerde que sintácticamente Prolog utiliza [] por nil y [H|T] por H|T. Cada cláusula consiste de una guarda y un cuerpo. La barra vertical | separa la guarda del cuerpo. Una guarda sólo realiza comprobaciones, y se bloquea si no puede decidir una comprobación. Una guarda debe cumplirse para que la cláusula sea elegible. El cuerpo sólo se ejecuta si la cláusula es elegida. El cuerpo puede ligar variables de salida.

El mezclador de flujos invoca primero a `WaitTwo` para decidir cuál flujo escuchar. La declaración `case` se ejecuta solamente después de que `WaitTwo` devuelve un valor. Gracias al argumento `F`, las alternativas que no aplican se saltan. Note que cada invocación recursiva intercambia los dos argumentos de tipo flujo. Esto ayuda a garantizar la imparcialidad entre los flujos en los sistemas en donde la declaración `WaitTwo` favorece al uno o al otro (lo cual es frecuente en una implementación). Un mensaje que llega a un flujo de entrada, finalmente aparecerá en un flujo de salida, independientemente de lo que suceda con el otro flujo de entrada.

¿Es práctico?

¿Qué podemos decir sobre la programación en la práctica en este modelo? Supongamos que hay clientes nuevos que llegan durante la ejecución. Cada cliente desea comunicarse con el servidor. ¡Esto significa que se debe crear un nuevo mezclador de flujos para cada cliente! El resultado final será un árbol de mezcladores de flujos alimentando el servidor. ¿Es práctica esta solución? Ella tiene dos problemas:

- Es ineficiente. Cada mezclador de flujos se ejecuta en su propio hilo. El árbol de mezcladores de flujos se extiende en tiempo de ejecución cada vez que un objeto nuevo referencia al servidor. Además, el árbol no es necesariamente balanceado. Tomará un tiempo adicional el balancearlo.
- Le falta expresividad. Es imposible referenciar directamente al servidor. Por ejemplo, es imposible colocar la referencia a un servidor en una estructura de datos. La única manera que tenemos de referenciar el servidor es referenciando a uno de sus flujos. Podemos colocar éste en una estructura de datos, pero solamente un cliente puede usar esa referencia. (Recuerde que las estructuras de datos declarativas no se pueden modificar.)

¿Cómo podemos resolver estos dos problemas? El primer problema podría, hipotéticamente, resolverse por medio de un compilador muy inteligente que reconoce el árbol de mezcladores de flujos y lo reemplaza, en la implementación, por una comunicación directa de muchos-a-uno. Sin embargo, después de dos décadas de investigación en esta área, tal compilador no existe [172]. Algunos sistemas resuelven el problema de otra manera: agregando una abstracción para mezclas de múltiples flujos, cuya implementación se realiza por fuera del modelo. Esto equivale a extender el modelo con puertos. El segundo problema se puede resolver parcialmente (ver ejercicios, sección 5.9), pero la solución es aún incómoda.

Al parecer, hemos encontrado una limitación del modelo concurrente no determinístico. Después de una revisión más profunda, el problema parece ser que no existe una noción de estado explícito en el modelo, donde el estado explícito asocia un nombre con una referencia en el almacén. Tanto el nombre como la referencia al almacén son inmutables; solamente se puede cambiar la asociación. Existen muchas formas equivalentes de introducir estado explícito. Una manera es agregar el concepto de celda, como lo mostraremos en el capítulo 6. Otra forma es agregando el concepto de puerto, como lo hicimos en este capítulo. Los puertos y las celdas son equivalentes en un lenguaje concurrente: existen implementaciones sencillas de cada uno en términos del otro.

```
fun {WaitTwo A B}
x in
  thread {Wait A} try X=1 catch _ then skip end end
  thread {Wait B} try X=2 catch _ then skip end end
  X
end
```

Figura 5.25: Escogencia no-determinística simétrica (utilizando excepciones).

```
fun {WaitTwo A B}
U in
  thread {Wait A} U=unit end
  thread {Wait B} U=unit end
  {Wait U}
  if {IsDet A} then 1 else 2 end
end
```

Figura 5.26: Escogencia no-determinística asimétrica (utilizando IsDet).

Implementación de la escogencia no-determinística

La operación `WaitTwo` se puede definir en el modelo concurrente declarativo extendido con excepciones.¹³ En la figura 5.25 se presenta una definición sencilla. `WaitTwo` devuelve 1 o 2, dependiendo de si `A` está ligada o `B` está ligada. Esta definición es simétrica, pues no favorece ni a `A` ni a `B`. Podemos escribir una versión asimétrica que favorezca a `A`, utilizando `IsDet`, tal como se muestra en la figura 5.26.¹⁴

5.9. Ejercicios

1. *Objetos puerto que comparten un hilo.* En la sección 5.6.1 se presentó un programa pequeño, denominado ping-pong, que cuenta con dos objetos puerto. Cada objeto ejecuta un método `y` de manera asincrónica invoca al otro. Una vez se inserta un mensaje inicial en el sistema, se produce una secuencia infinita de mensajes ping-pong, yendo `y` viniendo, entre los objetos. ¿Qué pasa si se insertan dos (o

13. Para efectos prácticos, sin embargo, recomendamos la definición presentada en el archivo de suplementos del sitio Web del libro.

14. Ambas definiciones tienen un pequeño defecto: si una variable nunca es ligada, pueden dejar, por siempre, hilos “pendientes alrededor.” Las definiciones se pueden corregir para terminar todos los hilos pendientes. Dejamos esas correcciones como un ejercicio para el lector.

Concurrencia por paso de mensajes

más) mensajes iniciales? Por ejemplo, ¿qué pasa si se envían estos dos mensajes iniciales?:

```
{ Invocar objping ping(0) }
{ Invocar objpong pong(10000000) }
```

Los mensajes continuarán en ping-pong indefinidamente, pero ¿cómo? ¿Cuáles mensajes se enviarán y cómo se intercalarán las ejecuciones de los objetos? ¿La intercalación será sincronizada como en una marcha militar (alternándose estrictamente entre los objetos), libre (sujeta a las fluctuaciones causadas por el planificador de hilos), o algo entre estos dos extremos?

2. *Sistema de control de ascensores.* En la sección 5.5 se presentó el diseño de un sistema sencillo de control de ascensores. Explorémoslo:

- a) El diseño actual tiene un objeto controlador por ascensor. Para economizar costos, el desarrollador decide cambiar esto y, entonces, conservar un solo controlador para todo el sistema. Cada ascensor se comunica con este controlador. La definición interna del controlador permanece igual. ¿Esta es una buena idea? ¿Cómo cambia el comportamiento del sistema de control de ascensores?
- b) En el diseño actual, el controlador avanza, hacia arriba o hacia abajo, un piso a la vez. El controlador se detiene en todos los pisos por donde pasa, aún si el piso no fue solicitado. Cambie los objetos ascensor y controlador para evitar este comportamiento, de manera que el ascensor se detenga únicamente en los pisos solicitados.

3. *Tolerancia a fallos para el sistema de control de ascensores.* Pueden ocurrir dos tipos de fallas: los componentes pueden ser bloqueados temporalmente o pueden estar fuera de servicio permanentemente. Miremos cómo manejar algunos problemas comunes que suelen presentarse a causa de estos fallos:

- a) Un ascensor es bloqueado. Extienda el sistema para seguir funcionando cuando un asensor está temporalmente bloqueado en un piso por un usuario malicioso. Primero extienda el objeto piso para restablecer el temporizador de las puertas en el momento en que se solicita el mismo piso mientras las puertas están abiertas. Entonces, el plan del ascensor debería ser enviado a otros ascensores y los pisos no deberían solicitar más ese ascensor en particular. Una vez el ascensor funcione de nuevo, los pisos deberían poder solicitar ese ascensor de nuevo. Esto se puede hacer con tiempos de espera.
- b) Un ascensor está fuera de servicio. El primer paso consiste en agregar primitivas genéricas para detección de fallos. Podemos necesitar detección tanto sincrónica como asincrónica. En la detección sincrónica, cuando un componente sale de servicio, suponemos que cualquier mensaje enviado a él recibe una respuesta inmediata `fueradeservicio(Id)`, donde `Id` identifica el componente. En la detección asincrónica, “enlazamos” un componente con otro cuando todavía están funcionando. Entonces, cuando el segundo componente sale de servicio, el mensaje `fueradeservicio` se envía, inmediatamente, al primero. Ahora extienda el sistema para que continúe funcionando cuando un

ascensor está fuera de servicio.

c) Un piso está fuera de servicio. Extienda el sistema para que continúe funcionando cuando un piso está fuera de servicio. El sistema debe reconfigurarse por sí mismo para seguir funcionando en una edificación con un piso de menos.

d) Mantenimiento de un ascensor. Extienda el sistema de manera que un ascensor pueda ser detenido temporalmente para mantenimiento y vuelto a poner en funcionamiento más adelante.

e) Interacciones. ¿Qué pasa si varios pisos y ascensores quedan fuera de servicio simultáneamente? ¿Su sistema funcionará adecuadamente?

4. *Detección de terminación.* Reemplace la definición de SubThread de la sección 5.6.3 por:

```
proc {SubThread P}
  thread
    {Send Pt 1} {P} {Send Pt ~1}
  end
end
```

Explique por qué el resultado no es correcto. Muestre una ejecución tal que exista un punto donde la suma de los elementos en el flujo del puerto sea cero, pero aún no hayan terminado todos los hilos.

5. *Filtro concurrente.* En la sección 5.6.4 se definió una versión concurrente de Filter, denominada ConcFilter, que calcula cada elemento de la salida de manera independiente, i.e., sin esperar que los elementos previos sean calculados.

a) Qué pasa cuando se ejecuta lo siguiente:

```
declare Sal
{ConcFilter [5 1 2 4 0] fun {$ x} x>2 end Sal}
{Show Sal}
```

¿Cuántos elementos son desplegados por Show? ¿En qué orden se despliegan esos elementos? Si hay otras formas en que se puedan desplegar esos elementos, muéstrelas todas. ¿La ejecución de ConcFilter es determinística? ¿Por qué o por qué no?

b) Qué pasa cuando se ejecuta lo siguiente:

```
declare Sal
{ConcFilter [5 1 2 4 0] fun {$ x} x>2 end Sal}
{Delay 1000}
{Show Sal}
```

Ahora, ¿qué despliega Show? Si hay otras formas en que se puedan desplegar esos elementos, muéstrelas todas.

c) Qué pasa cuando se ejecuta lo siguiente:

```
declare Sal A
{ConcFilter [5 1 A 4 0] fun {$ x} x>2 end Sal}
{Delay 1000}
{Show Sal}
```

¿Qué se despliega ahora? ¿En qué orden se despliegan los elementos? Si, después de lo anterior, A se liga a 3, entonces ¿qué le pasa a la lista Sal?

Concurrencia por paso de mensajes

- d) Si la lista de entrada tiene n elementos, ¿cuál es la complejidad (en notación Θ) del número de operaciones de ConcFilter? Analice la diferencia en tiempo de ejecución entre Filter y ConcFilter.
6. *Semántica de la operación receive de Erlang.* En la sección 5.7.3 se mostró cómo traducir la operación `receive` de Erlang. La segunda forma de esta operación, con tiempo de espera, es la más general. Mirémosla un poco más de cerca.
- Verifique que la segunda forma reduce a la tercera forma cuando el tiempo de espera es cero.
 - Verifique que la segunda forma reduce a la primera forma cuando el tiempo de espera tiende a infinito.
 - Otra forma de traducir la segunda forma podría ser insertando un mensaje único (utilizando un nombre) después de n ms. Esto requiere cierto cuidado para asegurar que el mensaje único no aparezca luego en el flujo de salida. Escriba otra traducción de la segunda forma que utilice esta técnica. ¿Cuáles son las ventajas y desventajas de esta traducción con respecto a la del libro?
7. *La operación receive de Erlang como una abstracción de control.* En este ejercicio, implemente la operación `receive` de Erlang definida en la sección 5.7.3, como la abstracción de control siguiente:
- `C={Buzón.new}` crea un buzón nuevo, C.
 - `{Buzón.send C M}` envía el mensaje M al buzón C.
 - `{Buzón.receive C [P1#E1 P2#E2 ... Pn#En] D}` realiza una operación `receive` sobre el buzón C y devuelve el resultado. Cada `Pi` es una función booleana de un argumento, `fun {$ M} {expr} end`, que representa un patrón y su guarda. La función devuelve `true` si y sólo si el mensaje M es reconocido por un patrón y cumple la respectiva guarda. Cada `Ei` es una función de un argumento, `fun {$ M} {expr} end`, que representa el cuerpo. Esta función de invoca cuando el mensaje M se recibe con éxito; su resultado se devuelve como el resultado de `Buzón.receive`. El último argumento D representa el tiempo de espera. Puede ser el átomo `infinito`, el cual significa que no hay tiempo de espera, o la pareja `T#E`, donde T es un entero, en milisegundos, que define el retraso y E es una función sin argumentos.
8. *Limitaciones de la comunicación por flujos.* En este ejercicio, exploramos los límites de la comunicación por flujos en el modelo concurrente no-determinístico. En la sección 5.8.1 afirmamos que podemos resolver parcialmente el problema de colocar referencias a un servidor en una estructura de datos. ¿Cuán lejos podemos llegar? Considere el objeto activo siguiente:
- ```
declare SN
thread {ServidorDeNombres SN nil} end
```
- donde `ServidorDeNombres` se define como sigue (junto con la función auxiliar `Reemplazar`):

```
fun {Reemplazar LEn A ViejoS NuevoS}
 case LEn
 of B#S|L1 andthen A=B then
 Viejos=S
 A#NuevoS|L1
 [] E|L1 then
 E|{Reemplazar L1 A ViejoS NuevoS}
 end
 end
proc {ServidorDeNombres SN L}
 case SN
 of registrar(A S)|SN1 then
 {ServidorDeNombres SN1 A#S|L}
 [] obtflujo(A S)|SN1 then L1 Viejos NuevoS in
 L1={Reemplazar L A ViejoS NuevoS}
 thread {MezcladorFlujos S NuevoS ViejoS} end
 {ServidorDeNombres SN1 L1}
 [] nil then
 skip
 end
 end
```

El objeto `ServidorDeNombres` reconoce dos mensajes. Suponga que `S` es un flujo de entrada de un servidor y `foo` es el nombre que deseamos colocarle al servidor. Dada una referencia `SN` al flujo de entrada del servidor de nombres, hacer `SN=registrar(foo S)|SN1` agrega la pareja `foo#S` a su lista interna `L`. Por su lado, hacer `SN=obtflujo(foo S1)|SN1` creará un flujo de entrada fresco, `S1`, para el servidor cuyo nombre es `foo`, el cual ha sido almacenado por el servidor de nombres en la lista interna `L`. Como `foo` es una constante, podemos colocar referencias a servidores en una estructura de datos, definiendo un servidor de nombres. ¿Esta solución es práctica? ¿Por qué o por qué no? Piense antes de leer la respuesta en la nota de pie de página.<sup>15</sup>

---

15. ¡No es posible nombrar el servidor de nombres! Éste tiene que ser añadido a todos los procedimientos como un argumento adicional. Eliminar este argumento necesita estar explícito.

Draft  
<sub>442</sub>

## 6

## Estado explícito

L'état c'est moi.  
*Yo soy el estado.*  
– Luis XIV (1638–1715)

Si la programación declarativa es como un cristal, inmutable y prácticamente eterna, entonces la programación con estado es orgánica: ella crece y evoluciona.  
– Inspirado en *On Growth and Form*, D'Arcy Wentworth Thompson (1860–1948)

A primera vista, el estado explícito es una extensión menor de la programación declarativa: los resultados de un componente, además de depender de sus argumentos, dependen también de un parámetro interno, el cual se llama su “estado.” Este parámetro le provee al componente una memoria de largo plazo, un “sentido de historia” si se quiere.<sup>1</sup> Sin estado, un componente sólo tiene memoria de corto plazo, memoria que sólo existe durante una invocación particular del componente. El estado agrega una rama potencialmente infinita a un programa que se ejecuta por un tiempo finito. Con esto queremos decir lo siguiente. Un componente que se ejecuta por un tiempo finito solamente puede acumular una cantidad finita de información. Si el componente tiene estado, entonces se le puede agregar, a esta información finita, la información almacenada por el estado. Esta “historia” puede ser indefinidamente larga, pues el componente puede tener una memoria que abarca bastante hacia el pasado.

Oliver Sacks describió el caso de gente con un daño cerebral, quienes sólo tenían memoria de corto plazo [149]. Ellos viven en un “presente” continuo sin ninguna memoria más allá de unos pocos segundos hacia el pasado. El mecanismo de su cerebro, para “fijar” las memorias de corto plazo en el almacenamiento de largo plazo, está dañado. Debe ser muy extraño vivir de esta forma. ¿De pronto, esta gente usa el mundo externo como una especie de memoria de largo plazo? Esta analogía permite darse una idea de cuán importante puede ser el estado para la gente. Veremos que el estado es igualmente importante para la programación.

---

1. En el capítulo 5 también se introdujo una forma de memoria de largo plazo, el puerto. Esa memoria fue usada para definir los objetos puerto, entidades activas con una memoria interna. El énfasis se colocó sobre la concurrencia. El énfasis de este capítulo es la expresividad de la noción de estado sin concurrencia.

**Grados de declaratividad**

A la programación sin estado y con estado se les denomina con frecuencia programación declarativa e imperativa, respectivamente. La programación declarativa, literalmente hablando, significa programar con declaraciones, i.e., decir lo que se quiere y dejar que el sistema determine cómo obtenerlo. La programación imperativa, literalmente hablando, significa dar órdenes, i.e., decir cómo hacer algo. En este sentido, el modelo declarativo del capítulo 2 también es imperativo, porque define secuencias de comandos.

El problema real es que “declarativo” no es una propiedad absoluta, sino más bien una cuestión de gradualidad. El lenguaje Fortran, desarrollado en los años 1950, fue el primer lenguaje dominante que permitía escribir expresiones aritméticas en una sintaxis que se parecía a la notación matemática [13]. Comparado con un lenguaje ensamblador, ¡Fortran es definitivamente declarativo! Uno podría decirle al computador que necesita  $I+J$  sin especificar en qué lugar de la memoria almacenar  $I$  y  $J$  ni cuáles instrucciones de máquina ejecutar para recuperar sus valores y sumarlos. En este sentido relativo, los lenguajes se han vuelto más declarativos al paso de los años. Fortran dió paso a Algol-60 y la programación estructurada [45, 46, 121], la cual dió paso a Simula-67 y los lenguajes modernos de programación orientada a objetos [128, 139].<sup>2</sup>

En este libro nos apegamos al uso tradicional de declarativo como sin estado e imperativo como con estado. Denominamos el modelo de computación del capítulo 2 “declarativo” aunque algunos modelos posteriores sean realmente más declarativos, por ser más expresivos. Nos apegamos al uso tradicional porque existe un sentido importante en el cual el modelo declarativo es realmente declarativo de acuerdo al significado literal de la palabra. Este sentido surge cuando miramos el modelo declarativo desde el punto de vista de la programación funcional y lógica:

- Un programa lógico se puede “leer” de dos maneras: como un conjunto de axiomas lógicos (el qué) o como un conjunto de comandos (el cómo). Esto lo resume Robert Kowalski con su famosa ecuación: Algoritmo = Lógica + Control [95, 96]. Los axiomas lógicos, complementados con información del flujo del control (dados implícitamente o explícitamente por el programador), conforman un programa que puede ejecutarse en un computador. En la sección 9.3.3 (en CTM) se explica cómo funciona esto para el modelo declarativo.
- Un programa funcional también se puede “leer” de dos maneras: como una definición de un conjunto de funciones en el sentido matemático (el qué) o como un conjunto de comandos para evaluar esas funciones (el cómo). Visto como un conjunto de comandos, la definición se ejecuta en un orden particular. Los

---

2. Es de resaltar que los tres lenguajes fueron diseñados en un período de diez años, aproximadamente entre 1957 y 1967. Considerando que Lisp y Absys también son del mismo período y que Prolog es de 1972, podemos hablar de una verdadera edad de oro del diseño de lenguajes de programación.

dos órdenes más populares son evaluación ansiosa y perezosa. Cuando se conoce el orden, la definición matemática se puede ejecutar en un computador. En la sección 4.9.2 se explica cómo funciona esto para el modelo declarativo.

Sin embargo, en la práctica, la lectura declarativa de un programa lógico o funcional puede perder mucho de su aspecto “qué” debido a que el programa tiene que especificar muchos detalles del “cómo” (ver el epígrafe de en el encabezado del capítulo 3). Por ejemplo, una definición declarativa de un árbol de búsqueda tiene casi tantas órdenes como una definición imperativa. Sin embargo, la programación declarativa tiene aún tres ventajas cruciales. Primero, es más fácil construir abstracciones en un contexto declarativo, pues las operaciones declarativas son, por naturaleza, compositionales. Segundo, los programas declarativos son más fáciles de comprobar, pues es suficiente con comprobar invocaciones individuales (dar los argumentos y verificar los resultados). La comprobación de los programas con estado es más difícil porque incluye la comprobación de secuencias de invocaciones (debido a la historia interna). Tercero, razonar con la programación declarativa es más sencillo que con la programación imperativa (e.g., es posible un razonamiento algebráico).

### ***Estructura del capítulo***

En este capítulo se presentan las ideas y técnicas básicas para usar el concepto de estado en el diseño de programas. Este capítulo se estructura de la manera siguiente:

- Primero introducimos y definimos el concepto de estado explícito en las tres primeras secciones.
  - En la sección 6.1 se introduce el concepto de estado explícito: se define la noción general de “estado,” la cual es independiente de cualquier modelo de computación, y se muestran las diferentes formas en que los modelos, declarativo y con estado, implementan esta noción.
  - En la sección 6.2 se explican los principios básicos del diseño de sistemas y por qué el concepto de estado es una parte esencial en ese diseño. También se presentan las primeras definiciones de programación basada en componentes y de programación orientada a objetos.
  - En la sección 6.3 se define de forma precisa el modelo de computación con estado.
- Luego, en las dos secciones siguientes, introducimos la abstracción de datos con estado.
  - En la sección 6.4 se explican las diferentes maneras de construir abstracciones de datos, con y sin estado explícito. Se explican los dos estilos principales para construir abstracciones de datos, el estilo TAD y el estilo objeto.
  - En la sección 6.5 se presenta una visión general de algunos TADs útiles, a saber, las colecciones de ítems. Se explica el compromiso entre expresividad y

eficiencia en estos TADs.

- En la sección 6.6 se muestra cómo razonar con estado. Presentamos una técnica, el método de invariantes, que, cuando se puede aplicar, puede hacer que este razonamiento sea casi tan sencillo como razonar sobre programas declarativos.
- En la sección 6.7 se explica la programación en grande, consistente en la programación realizada por un equipo de personas. Esto extiende la presentación de la programación en pequeño realizada en la sección 3.9.
- En la sección 6.8 se presentan algunos casos de estudio de programas que utilizan estado, para aclarar aún más las diferencias con los programas declarativos.
- En la sección 6.9 se introducen unos tópicos más avanzados: las limitaciones de la programación con estado y cómo extender la administración de la memoria para incluir referencias externas.

En el capítulo 7 se continúa la discusión del concepto de estado, desarrollando un estilo de programación particularmente útil, a saber, la programación orientada a objetos. Debido a la amplia aplicabilidad de la programación orientada a objetos, dedicamos todo un capítulo a ella.

---

## 6.1. ¿Qué es estado?

Ya hemos programado con estado en el modelo declarativo del capítulo 3. Por ejemplo, los acumuladores de la sección 3.4.3 representan un estado. ¿Entonces, por qué necesitamos todo un capítulo dedicado al concepto de estado? Para ver por qué, miremos más detenidamente qué es realmente un estado. En su forma más simple, podemos definir estado así:

Un *estado* es una secuencia de valores en el tiempo que contienen los resultados intermedios de una computación específica.

Examinemos las diferentes maneras en que el estado puede estar presente en un programa.

### 6.1.1. Estado implícito (declarativo)

La necesidad de la secuencia sólo existe en la mente del programador. No se necesita ningún tipo de soporte por parte del modelo de computación. Este tipo de estado se denomina estado implícito o estado declarativo. Como ejemplo, miremos la función declarativa `SumList`:

```
fun {SumList xs s}
 case xs
 of nil then s
 [] x|xr then {SumList xr x+s}
 end
end
```

Es una función recursiva. En cada invocación se pasan dos argumentos: `xs`, lo que falta por recorrer de la lista de entrada, y `s`, la suma de la parte recorrida de la lista de entrada. Mientras se calcula la suma de una lista, `SumList` se invoca a sí mismo muchas veces. Observemos la pareja `(xs#s)` en cada invocación, ya que ella nos dará toda la información que necesitamos para caracterizar la invocación. En el caso de la invocación `{SumList [1 2 3 4] 0}`, obtenemos la secuencia siguiente:

```
[1 2 3 4] # 0
[2 3 4] # 1
[3 4] # 3
[4] # 6
nil # 10
```

Esta secuencia es un estado. Mirado de esta manera, `SumList` calcula con estado, aunque ni el programa ni el modelo de computación “conocen” esto. El estado está completamente en la mente del programador.

### 6.1.2. Estado explícito

Para una función puede ser útil contar con un estado que perdure a través de las invocaciones a la función y que esté oculto para los que la invocan. Por ejemplo, podemos extender `SumList` para contar cuántas veces es invocada. No existe ninguna razón para que quien invoca esta función conozca esta extensión. Aún más fuerte: por razones de modularidad quien invoca no debe conocer la extensión. Este comportamiento no puede ser programado en el modelo declarativo. Lo más cerca que podemos llegar consiste en agregar dos argumentos a `SumList` (un contador de entrada y uno de salida) e hilarlos a través de las invocaciones. Para hacerlo sin argumentos adicionales necesitamos un estado explícito:

Un *estado explícito* en un procedimiento es un estado cuyo tiempo de vida se extiende más allá de una invocación al procedimiento sin estar presente en los argumentos del procedimiento.

El estado explícito no se puede expresar en el modelo declarativo. Para tenerlo, extendemos el modelo con una especie de contenedor que denominamos una celda. Una celda tiene un nombre, un tiempo de vida indefinido, y un contenido que puede ser modificado. Si el procedimiento conoce el nombre, puede modificar su contenido. El modelo declarativo extendido con celdas se denomina el modelo con estado. A diferencia del estado declarativo, el estado explícito no se encuentra sólo en la mente del programador, sino que es visible tanto en el programa como en el modelo de computación. Podemos usar una celda para añadir una memoria de largo plazo a `SumList`. Por ejemplo, podemos llevar la cuenta del número de veces que ha sido invocada:

*Estado explícito*

```
local
 C={NewCell 0}
in
 fun {SumList xs s}
 C:=@C+1
 case xs
 of nil then s
 [] x|xr then {SumList xr x+s}
 end
 end
 fun {ContadorSum} @C end
end
```

Esta es la misma definición de antes, salvo que definimos una celda y actualizamos su contenido en el cuerpo de `SumList`. También añadimos la función `ContadorSum` para poder observar el estado. Ahora explicamos las operaciones nuevas que actúan sobre el estado explícito. `NewCell` crea una celda nueva, conteniendo inicialmente 0. `@` permite consultar el contenido de la celda y `:=` permite colocar en la celda un contenido nuevo. Si `ContadorSum` no se utiliza, entonces esta versión de `SumList` no puede ser distinguida de la versión previa: la función es invocada de la misma forma y devuelve los mismos resultados.<sup>3</sup>

La capacidad de tener estado explícito es muy importante. Esto elimina las limitaciones de la programación declarativa (ver sección 4.8). Con estado explícito, las abstracciones de datos mejoran tremadamente en modularidad pues es posible encapsular un estado explícito dentro de ellas. El acceso al estado está limitado por las operaciones de la abstracción de datos. Esta idea está en el corazón de la programación orientada a objetos, un poderoso estilo de programación que se presenta en el capítulo 7. Tanto este capítulo como el capítulo 7, exploran las ramificaciones del estado explícito.

---

## 6.2. Estado y construcción de sistemas

### *El principio de abstracción*

Hasta donde conocemos, el principio más exitoso para la construcción de sistemas por parte de seres inteligentes con unas capacidades finitas de pensamiento, como los seres humanos, es el principio de abstracción. Considere cualquier sistema; éste se puede pensar como constituido de dos partes: una especificación y una implementación. En un sentido matemático, que es más fuerte que el sentido legal, la especificación es un contrato que define cómo se debe comportar el sistema. Decimos que un sistema es correcto si su comportamiento real es acorde con el contrato. Si el sistema se comporta de manera diferente, entonces decimos que el

---

3. Las únicas diferencias son una desaceleración menor y un pequeño incremento en el uso de la memoria. En casi todos los casos, estas diferencias son irrelevantes en la práctica.

sistema falla.

La especificación define cómo interactúa el resto del mundo con el sistema, visto desde el exterior. La implementación es cómo está construido el sistema, visto desde su interior. La propiedad milagrosa de la distinción especificación/implementación es que la especificación es normalmente mucho más sencilla de entender que la implementación. Uno no tiene que saber cómo está hecho un reloj para poder leer la hora en él. Parafraseando al evolucionista Richard Dawkins, no importa si el relojero es ciego o no, mientras que el reloj funcione.

Esto significa que es posible construir un sistema como una serie concéntrica de capas. Se puede proceder etapa por etapa, construyendo capa sobre capa. En cada capa, se construye una implementación que toma la siguiente especificación de más bajo nivel y provee la siguiente especificación de más alto nivel. No es necesario entenderlo todo de una sola vez.

### *Sistemas que crecen*

¿Cómo soporta la programación declarativa este enfoque? Con el modelo declarativo del capítulo 2, todo lo que el sistema “sabe” está en su exterior, salvo un conjunto fijo de conocimientos que nacen con el sistema. Para ser precisos, como los procedimientos no tienen estado, todo su conocimiento, sus “astucias” están en sus argumentos. Entre más inteligente sea el procedimiento, los argumentos se vuelven más numerosos y más “pesados”. La programación declarativa es como un organismo que conserva todo su conocimiento por fuera de sí mismo, en su ambiente. A pesar de pretender lo contrario (ver el epígrafe en el encabezado de este capítulo) esta fue exáctamente la situación de Luis XIV: el estado no estaba en su persona sino en todas aquellas alrededor de él, en el siglo 17 en Francia.<sup>4</sup> Concluimos que el principio de abstracción no está bien soportado por la programación declarativa, porque no podemos agregar conocimiento nuevo dentro de un componente.

En el capítulo 4 se alivia parcialmente este problema agregando concurrencia. Los objetos flujo pueden acumular conocimiento interno en sus argumentos internos. En el capítulo 5 se aumenta dramáticamente el poder expresivo agregando objetos puerto. Un objeto puerto tiene una identidad y puede ser visto desde afuera como una entidad sin estado. Pero esto necesita de la concurrencia. En este capítulo, agregamos estado explícito sin concurrencia. Veremos que esto promueve un estilo de programación muy diferente al estilo de componentes concurrentes del capítulo 5. Existe un orden total entre todas las operaciones en el sistema. Esto consolida una dependencia fuerte entre todas las partes del sistema. Más adelante, en el capítulo 8, agregamos concurrencia para eliminar esta dependencia. Es difícil de programar en el modelo de ese capítulo. Miremos por eso, qué podemos hacer, primero, con estado sin concurrencia.

---

4. Para ser imparcial con Luis, lo que el quiso decir fue que el poder de tomar decisiones del estado estaba investido en su persona.

*Estado explícito***6.2.1. Propiedades de un sistema**

¿Cuáles propiedades debería tener un sistema para soportar, de la mejor manera posible, el principio de abstracción? Tres de ellas son:

- *Encapsulación*. Debería ser posible ocultar lo interno de una parte.
- *Composicionalidad*. Debería ser posible combinar partes para construir nuevas partes.
- *Instanciación/invocación*. Debería ser posible crear muchas instancias de una parte basados en una sola definición. Estas instancias “encajan” por sí mismas en su ambiente (el resto del sistema en el cual viven) en el momento de su creación.

Estas propiedades requieren de un soporte por parte del lenguaje de programación, e.g., el alcance léxico soporta la encapsulación y la programación de alto orden soporta la instanciación. Las propiedades no requieren estado; también se pueden usar con la programación declarativa. Por ejemplo, la encapsulación es ortogonal al estado. Por un lado, se puede usar la encapsulación en programas declarativos sin estado. Ya la hemos usado muchas veces, e.g., en programación de alto orden y en los objetos flujo. Por el otro lado, también se puede usar estado sin encapsulación, definiendo el estado globalmente de manera que todos los componentes tengan acceso libre a él.

***Invariantes***

La encapsulación y el estado explícito son más útiles si se usan juntos. Al agregar estado a la programación declarativa, el razonamiento sobre los programas se vuelve mucho más difícil, pues el comportamiento de los programas depende del estado. Por ejemplo, un procedimiento puede tener efectos de borde, i.e., modificar el estado que es visible al resto del programa. Los efectos de borde dificultan enormemente el razonamiento sobre los programas. La introducción de la encapsulación ayuda bastante para que el razonamiento sea nuevamente posible. Esto sucede gracias a que los sistemas con estado pueden ser diseñados de manera que una propiedad bien definida, denominada un invariante, sea siempre cierta cuando se mira desde el exterior. Esto hace que razonar sobre el sistema sea independiente de razonar sobre su ambiente, y nos devuelve una de las propiedades que hacen tan atractiva la programación declarativa.

Los invariantes son sólo una parte de la historia. Un invariante sólo dice que el componente no se está comportando incorrectamente; pero no garantiza que el componente esté avanzando hacia el objetivo. Por ello, se necesita una segunda propiedad que asegure el progreso. Esto quiere decir que aún con invariantes, la programación con estado no es tan simple como la programación declarativa. En nuestra opinión, en la construcción de sistemas complejos, es una buena práctica conservar declarativos tantos componentes como se pueda. El estado no debe ser “diseminado” en muchos componentes; más bien debe estar concentrado, solamente,

en unos pocos componentes cuidadosamente seleccionados.

### 6.2.2. Programación basada en componentes

Las tres propiedades de encapsulación, compositividad, e instantiación definen la programación basada en componentes (ver sección 6.7). Un componente especifica un fragmento de programa con un interior y un exterior, i.e., con una interfaz bien definida. El interior está oculto del exterior, excepto para lo que la interfaz permita. Los componentes se pueden combinar para formar nuevos componentes. Los componentes se pueden instanciar, creando una instancia nueva ligada dentro de su ambiente. Los componentes son un concepto ubicuo, que hemos visto con varias apariencias:

- *Abstracción procedimental.* Vimos un primer ejemplo de componentes en el modelo de computación declarativo. El componente se denomina definición de procedimiento y su instancia se denomina invocación de procedimiento. La abstracción procedural subyace a los modelos de componentes más avanzados que vienen más adelante.
- *Functors* (unidades de compilación). Una especie particularmente útil de componente es una unidad de compilación, i.e., una unidad que puede ser compilada independientemente de otros componentes. Denominamos a tales componentes functors y a sus instancias módulos.
- *Componentes concurrentes.* Un sistema con entidades independientes, interaccutantes, puede ser visto como un grafo de componentes concurrentes que se envían mensajes entre ellos.

En la programación basada en componentes, la manera natural de extender un componente es utilizando la composición: construir un componente nuevo que contiene el original. El componente nuevo provee una funcionalidad nueva y utiliza el componente antiguo para implementar esa funcionalidad.

Presentamos un ejemplo concreto, a partir de nuestra experiencia, para mostrar la utilidad de los componentes. La programación basada en componentes fue una parte esencial del proyecto Information Cities, en el cual se realizaron simulaciones multi-agente de gran alcance utilizando el sistema Mozart [142, 150]. Las simulaciones tenían la intención de modelar la evolución y el flujo de información en partes de Internet. Se definieron diferentes motores de simulación (en un solo proceso o distribuido, con formas diferentes de sincronización) como componentes reutilizables con interfaces idénticas. De la misma manera, se definieron diferentes comportamientos de agentes. Esto permitió establecer rápidamente muchas simulaciones diferentes y extender el simulador sin tener que recomilar el sistema. Este montaje fue realizado por un programa, utilizando el manejador de módulos que provee el módulo `Module` del sistema. Esto se puede hacer gracias a que los componentes son valores en el lenguaje Oz (ver sección 3.9.3).

### 6.2.3. Programación orientada a objetos

Un conjunto popular de técnicas para la programación con estado se denomina programación orientada a objetos. Dedicamos todo el capítulo 7 a estas técnicas. La programación orientada a objetos se basa en una forma particular de construir abstracciones de datos, denominadas “objetos.” Los objetos y los TADs son fundamentalmente diferentes. Los TADs mantienen los valores y las operaciones por separado. Los objetos los combinan (valores y operaciones) en una sola entidad denominada un “objeto” y a la cual se le puede invocar. La distinción entre TADs y objetos se explica en la sección 6.4. Los objetos son importantes porque facilitan el uso de las técnicas poderosas de polimorfismo y herencia. Como el polimorfismo se puede lograr tanto en la programación basada en componentes como en la programación orientada a objetos, no lo discutimos más aquí. En cambio, introducimos la herencia, el cual es un concepto nuevo que no hace parte de la programación basada en componentes:

- *Herencia.* Es la posibilidad de construir una abstracción de datos de manera incremental, como una pequeña extensión o modificación de otra abstracción de datos.

Los componentes construidos incrementalmente se denominan clases y sus instancias se denominan objetos. La herencia es una manera de estructurar programas de manera que una implementación nueva extienda una ya existente.

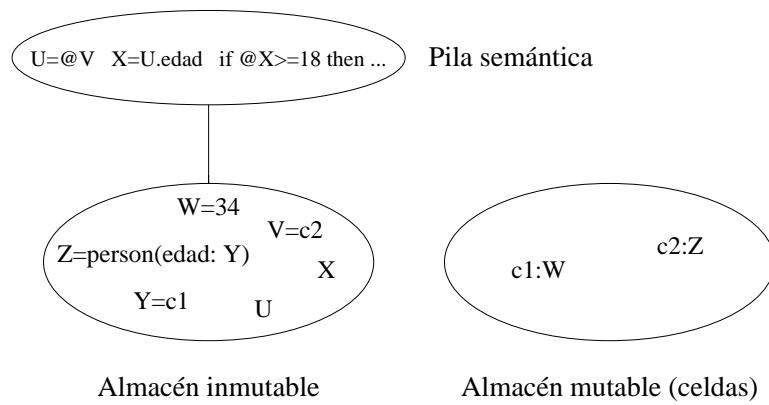
La ventaja de la herencia es que factoriza la implementación para evitar redundancias. Pero la herencia no tiene solo ventajas, pues implica que un componente depende fuertemente de los componentes de los que hereda. Esta dependencia puede ser difícil de manejar. La mayoría de la literatura sobre diseño orientado a objetos, e.g., sobre patrones de diseño [54], se enfoca en el uso correcto de la herencia. Aunque la composición de componentes es menos flexible que la herencia, es mucho más sencilla de usar. Recomendamos usar la composición siempre que sea posible y utilizar la herencia sólo cuando la composición sea insuficiente (ver capítulo 7).

---

## 6.3. El modelo declarativo con estado explícito

Una forma de introducir el estado es teniendo componentes concurrentes que se ejecutan indefinidamente y que se comunican con otros componentes, como los objetos flujo del capítulo 4 o los objetos puerto del capítulo 5. En este capítulo añadimos directamente el estado explícito al modelo declarativo. A diferencia de los capítulos previos, el modelo resultante es aún secuencial. Lo denominamos el modelo con estado.

El estado explícito es una pareja de dos entidades del lenguaje. La primera entidad es la identidad del estado y la segunda es el contenido actual del estado. Existe una operación que dada la identidad del estado devuelve el contenido actual. Esta



**Figura 6.1:** El modelo declarativo con estado explícito.

operación define una asociación amplia en el sistema entre las identidades de los estados y todas las entidades del lenguaje. Lo que lo hace con estado es que la asociación se puede modificar. De manera interesante, ninguna de las dos entidades del lenguaje se modifica; la asociación es la única que cambia.

### 6.3.1. Celdas

Agregamos el estado explícito como un nuevo tipo básico del modelo de computación. A ese tipo lo denominamos celda. Una celda es una pareja formada por una constante, la cual es un valor de tipo nombre, y una referencia al almacén de asignación única. Como los nombres son inexpugnables, las celdas son un ejemplo de un TAD seguro. El conjunto de todas las celdas vive en el almacén mutable. En la figura 6.1 se muestra el modelo de computación resultante. Hay dos almacenes: el almacén inmutable (de asignación única), el cual contiene las variables de flujo de datos que se pueden ligar a un valor, y el almacén mutable, el cual contiene parejas de nombres y referencias. En la tabla 6.1 se muestra el lenguaje núcleo. Comparado con el modelo declarativo, se agregan sólo dos nuevas declaraciones, las operaciones sobre celdas `NewCell` y `Exchange`. Estas operaciones se definen formalmente en la tabla 6.2. Por conveniencia, se agregan dos operaciones más en la tabla, `@` (acceso) y `:=` (asignación). Estas operaciones no proveen ninguna funcionalidad nueva, pues se pueden definir en términos de `Exchange`. Utilizar `C:=Y` como una expresión tiene el efecto de un `Exchange`: devuelve el antiguo valor como resultado.

Sorprendentemente, agregar las celdas con sus operaciones es suficiente para construir todos los conceptos maravillosos que el estado puede proveer. Todos los conceptos sofisticados de objetos, clases, y otras abstracciones de datos se pueden construir con el modelo declarativo, extendido con celdas. En la sección 7.6.2 se explica cómo construir clases y en la sección 7.6.3 se explica cómo construir objetos. En la práctica, sus semánticas se definen de esta manera, pero el lenguaje trae un

Estado explícito

|                                                                                                                                                          |                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| $\langle d \rangle ::=$                                                                                                                                  |                             |
| <b>skip</b>                                                                                                                                              | Declaración vacía           |
| $\langle d \rangle_1 \langle d \rangle_2$                                                                                                                | Declaración de secuencia    |
| <b>local</b> $\langle x \rangle$ <b>in</b> $\langle d \rangle$ <b>end</b>                                                                                | Creación de variable        |
| $\langle x \rangle_1 = \langle x \rangle_2$                                                                                                              | Ligadura variable-variable  |
| $\langle x \rangle = \langle v \rangle$                                                                                                                  | Creación de valor           |
| <b>if</b> $\langle x \rangle$ <b>then</b> $\langle d \rangle_1$ <b>else</b> $\langle d \rangle_2$ <b>end</b>                                             | Condicional                 |
| <b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{patrón} \rangle$ <b>then</b> $\langle d \rangle_1$ <b>else</b> $\langle d \rangle_2$ <b>end</b> | Reconocimiento de patrones  |
| $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$                                                                                  | Invocación de procedimiento |
| $\{ \text{NewName} \langle x \rangle \}$                                                                                                                 | Creación de nombre          |
| $\langle y \rangle = ! ! \langle x \rangle$                                                                                                              | Vista de sólo lectura       |
| <b>try</b> $\langle d \rangle_1$ <b>catch</b> $\langle x \rangle$ <b>then</b> $\langle d \rangle_2$ <b>end</b>                                           | Contexto de la excepción    |
| <b>raise</b> $\langle x \rangle$ <b>end</b>                                                                                                              | Lanzamiento de excepción    |
| $\{ \text{NewCell} \langle x \rangle \langle y \rangle \}$                                                                                               | <b>Creación de celda</b>    |
| $\{ \text{Exchange} \langle x \rangle \langle y \rangle \langle z \rangle \}$                                                                            | <b>Intercambio de celda</b> |

**Tabla 6.1:** El lenguaje núcleo con estado explícito.

| Operación                           | Descripción                                                                                     |
|-------------------------------------|-------------------------------------------------------------------------------------------------|
| $\{ \text{NewCell} \ X \ C \}$      | Crea una celda nueva C con contenido inicial X.                                                 |
| $\{ \text{Exchange} \ C \ X \ Y \}$ | Liga atómicamente X con el contenido antiguo de la celda C y hace que Y sea el contenido nuevo. |
| $X = @C$                            | Liga X con el contenido actual de la celda C.                                                   |
| $C := X$                            | Coloca a X como el contenido nuevo de la celda C.                                               |
| $X = C := Y$                        | Otra sintaxis para $\{ \text{Exchange} \ C \ X \ Y \}$ .                                        |

**Tabla 6.2:** Operaciones sobre celdas.

soporte sintáctico para hacerlos fáciles de usar y la implementación tiene un soporte para hacerlos más eficientes [65].

### 6.3.2. Semántica de las celdas

La semántica de las celdas es bastante similar a la semántica de los puertos presentada en la sección 5.1.2. Es instructivo compararlas. De manera similar a los puertos, primero añadimos un almacén mutable. El mismo almacén mutable puede servir tanto para puertos como para celdas. Luego definimos las operaciones `NewCell` y `Exchange` en términos del almacén mutable.

#### *Extensión del estado de ejecución*

Además del almacén de asignación única  $\sigma$  y el almacén de disparadores  $\tau$ , añadimos un almacén nuevo  $\mu$  denominado el almacén mutable. Este almacén contiene celdas, las cuales son parejas de la forma  $x : y$ , donde  $x$  y  $y$  son variables del almacén de asignación única. Inicialmente, el almacén mutable está vacío. La semántica garantiza que  $x$  siempre esté ligado a un valor de tipo nombre que representa una celda. Por otro lado,  $y$  puede estar ligado a cualquier valor parcial. El estado de ejecución ahora es una tripleta  $(MST, \sigma, \mu)$  (o una cuádrupla  $(MST, \sigma, \mu, \tau)$  si se considera el almacén de disparadores).

#### *La operación NewCell*

La declaración semántica  $(\{\text{NewCell } \langle x \rangle \langle y \rangle\}, E)$  hace lo siguiente:

- Crear un nombre de celda fresco  $n$ .
- Ligar  $E(\langle y \rangle)$  y  $n$  en el almacén.
- Si la ligadura tiene éxito, entonces agregar la pareja  $E(\langle y \rangle) : E(\langle x \rangle)$  al almacén mutable  $\mu$ .
- Si la ligadura falla, entonces lanzar una condición de error.

Un lector observador notará que su semántica es casi idéntica a la de los puertos. La principal diferencia es el tipo. Los puertos se identifican por un nombre de puerto y las celdas por un nombre de celda. Gracias al tipo, podemos controlar que las celdas sólo puedan ser usadas con `Exchange` y los puertos sólo puedan ser usados con `Send`.

#### *La operación Exchange*

La declaración semántica  $(\{\text{Exchange } \langle x \rangle \langle y \rangle \langle z \rangle\}, E)$  hace lo siguiente:

- Si la condición de activación es cierta ( $E(\langle x \rangle)$  está determinada), entonces realizar las acciones siguientes:

*Estado explícito*

- Si  $E(\langle x \rangle)$  no está ligada al nombre de una celda, entonces lanzar una condición de error.
- Si el almacén mutable contiene  $E(\langle x \rangle) : w$ , entonces realizar las acciones siguientes:
  - Actualizar el almacén mutable con  $E(\langle x \rangle) : E(\langle z \rangle)$ .
  - Ligar  $E(\langle y \rangle)$  y  $w$  en el almacén.
- Si la condición de activación es falsa, entonces suspender la ejecución.

**Administración de la memoria**

Se necesitan dos modificaciones a la administración de la memoria debido al almacén mutable:

- Extender la definición de alcanzabilidad: Una variable  $y$  es alcanzable si el almacén mutable contiene la pareja  $x : y$  y  $x$  es alcanzable.
- Recuperación de celdas: Si una variable  $x$  se vuelve inalcanzable, y el almacén mutable contiene la pareja  $x : y$ , entonces eliminar esa pareja.

Se necesitan las mismas modificaciones independientemente de si el almacén mutable contiene celdas o puertos.

**6.3.3. Relación con la programación declarativa**

En general, un programa con estado ya no es declarativo, pues la ejecución del programa varias veces, con las mismas entradas, puede producir diferentes salidas, dependiendo del estado interno. Es posible, sin embargo, escribir programas con estado que se comporten como si fueran declarativos, i.e., escribirlos de manera que satisfagan la definición de una operación declarativa. Es una buena costumbre de diseño el escribir componentes con estado de manera que se puedan comportar declarativamente.

Un ejemplo sencillo de programa con estado que se comporta declarativamente es la función `SumList` que presentamos anteriormente. Miremos un ejemplo más interesante, en el cual el estado se utiliza como una parte esencial en el cálculo de la función. Definimos una función para invertir una lista utilizando una celda:

```
fun {Reverse Xs}
 Rs={NewCell nil}
 in
 for x in Xs do Rs := X|@Rs end
 @Rs
 end
```

Como la celda está encapsulada dentro de `Reverse`, no hay manera de diferenciar esta implementación y una declarativa. Con frecuencia es posible tomar un programa declarativo y convertirlo en un programa con estado con el mismo comportamiento, reemplazando el estado declarativo por un estado explícito. También

es posible hacerlo en la dirección inversa. Dejamos como ejercicio para el lector tomar cualquier implementación con estado y convertirla en una implementación declarativa.

Otro ejemplo interesante es la memorización, en la cual una función recuerda los resultados de invocaciones previas de manera que las invocaciones futuras puedan ser manejadas más rápidamente. En el capítulo 10 (en CTM) se presenta un ejemplo utilizando un calendario gráfico sencillo. Se usa memorización para evitar volver a dibujar la pantalla a menos que ésta haya cambiado.

#### 6.3.4. Referencias compartidas e igualdad

Al introducir las celdas, también hemos extendido el concepto de igualdad. Tenemos que distinguir la igualdad de celdas de la igualdad de sus contenidos. Esto nos lleva al concepto de referencias compartidas e igualdad de lexemas.

##### *Referencias compartidas*

Las referencias compartidas, también conocidas como los alias, aparecen cuando dos identificadores `x` y `y` referencian la misma celda. En ese caso decimos que cada identificador es un alias del otro. De hecho, cambiar el contenido de `x` cambia también el contenido de `y`. Por ejemplo, creamos una celda:

```
x={NewCell 0}
```

Podemos crear una segunda referencia `y` a esta celda:

```
declare y in
y=x
```

Al cambiar el contenido de `y` se cambiará el contenido de `x`:

```
y:=10
{Browse @x}
```

Esto despliega 10. En general, cuando se cambia el contenido de una celda, entonces todos los alias de esa celda ven cambiado su contenido. Cuando se razona sobre un programa, el programador tiene que tener cuidado y tener en cuenta los alias. Esto puede ser difícil, pues los alias pueden estar esparcidos a través de todo el programa. Este problema se puede volver manejable encapsulando el estado, i.e., utilizándolo sólo en una pequeña parte del programa y garantizando que no puede salir de allí. Esta es una de las razones fundamentales de por qué la abstracción de datos es una idea especialmente buena, cuando se utiliza junto con el estado explícito.

##### *Igualdad de lexemas e igualdad estructural*

Dos valores son iguales si tienen la misma estructura. Por ejemplo:

```
x=persona(edad:25 nombre:"Jorge")
y=persona(edad:25 nombre:"Jorge")
{Browse x==y}
```

*Estado explícito*

Esto despliega **true**. Esto lo denominamos igualdad estructural y corresponde a la igualdad que hemos usado hasta ahora. Sin embargo, con las celdas, introducimos una nueva noción de igualdad denominada igualdad de lexemas. ¡Dos celdas no son iguales porque tengan el mismo contenido sino porque son la misma celda! Por ejemplo, creamos dos celdas:

```
X={NewCell 10}
Y={NewCell 10}
```

Estas celdas son diferentes: tienen identidades diferentes. La comparación siguiente:

```
{Browse X==Y}
```

despliega **false**. Es lógico decir que estas celdas no son iguales pues cambiar el contenido de una no cambia el contenido de la otra. Sin embargo, nuestras celdas si tienen el mismo contenido:

```
{Browse @X==@Y}
```

Esto despliega **true**. Esta es una pura coincidencia: no tiene que ser cierto a lo largo de todo el programa. Concluimos comentando que los alias tienen las mismas identidades. El ejemplo siguiente:

```
X={NewCell 10}
Y=X
{Browse X==Y}
```

despliega **true** porque X y Y son alias, i.e., referencian la misma celda.

---

#### 6.4. Abstracción de datos

Una abstracción de datos es una manera de usar datos en forma abstracta, i.e., podemos usar los datos sin tener en cuenta su implementación. Una abstracción de datos consiste de un conjunto de instancias que se pueden usar de acuerdo a ciertas reglas, las cuales conforman su interfaz. Algunas veces usaremos vagamente el término “tipo,” para referirnos a una abstracción de datos. Hay muchas ventajas en utilizar abstracciones de datos en lugar de usar directamente la implementación: usar la abstracción suele ser mucho más sencillo (pues la implementación suele ser complicada), razonar sobre los datos puede ser más sencillo (pues tienen sus propias características), y el sistema puede asegurar que los datos se utilicen de la manera correcta (pues no hay manera de manipular la implementación sino a través de la interfaz).

En la sección 3.7 se mostró una clase particular de abstracción de datos, a saber, un tipo abstracto de datos, o TAD: un conjunto de valores junto con un conjunto de operaciones sobre esos valores. Definimos un TAD pila con valores de tipo pila y operaciones de inserción y lectura de la cabeza de la pila. Los TADs no son la única forma de trabajar de manera abstracta con los datos. Ahora que hemos agregado el estado explícito al modelo, podemos presentar un conjunto más completo de

técnicas para realizar abstracción de datos.

#### 6.4.1. Ocho maneras de organizar una abstracción de datos

Una abstracción de datos con una funcionalidad dada se puede organizar de muchas maneras diferentes. Por ejemplo, en la sección 3.7 vimos que una sencilla abstracción de datos como la pila puede ser abierta o segura. Aquí introducimos dos ejes más, empaquetamiento y estado explícito. El empaquetamiento es una distinción fundamental observada originalmente en 1975 por John Reynolds [38, 61, 146]. Cada eje tiene dos posibilidades, lo cual nos lleva a ocho maneras, en total, de organizar una abstracción de datos. Algunas, raramente se usan; otras son comunes. Para cada una tiene sus ventajas y desventajas. Explicamos brevemente cada eje y presentamos algunos ejemplos. En los ejemplos que aparecen más adelante en el libro, escogemos entre estas maneras, una, la que sea más apropiada para cada caso.

##### *Apertura y seguridad*

Una abstracción de datos es segura si el lenguaje obliga su encapsulación. De no ser así se dice que es abierta. En el caso de una abstracción abierta, la encapsulación la debe forzar la disciplina del programador. Estrictamente hablando, si el lenguaje no obliga la encapsulación, entonces la abstracción de datos no es realmente una abstracción. Nosotros la seguimos llamando abstracción porque aún puede ser usada como tal. La única diferencia está en quién es responsable de forzar la encapsulación: el lenguaje o el programador.

##### *Empaquetamiento*

Una abstracción de datos está desempaquetada si define dos clases de entidades, denominadas valores y operaciones. Los valores se pasan como argumentos a las operaciones y se devuelven como resultados. Una abstracción de datos desempaquetada se denomina, normalmente, un tipo abstracto de datos, o TAD. Una abstracción de datos desempaquetada se puede volver segura utilizando una “llave.” La llave es una autorización para acceder a los datos internos de un valor abstracto (y actualizarlo, si el valor es con estado). Todas las operaciones del TAD conocen la llave. El resto del programa no conoce la llave. La llave puede ser un valor de tipo nombre, la cual es una constante inexpugnable (ver apéndice B.2).

Probablemente, el lenguaje más conocido basado en TADs es CLU, diseñado e implementado en los años 1970 por Barbara Liskov y sus estudiantes [107]. CLU se ha distinguido por ser el primer lenguaje implementado con soporte lingüístico para el estilo TAD.

Una abstracción de datos está empaquetada si define solamente una clase de entidad, denominada objeto, la cual combina las nociones de valor y de operación. Algunas veces, una abstracción de datos empaquetada se denomina abstracción procedural de datos, o APD.

*Estado explícito*

|                                           |                                                                                        |
|-------------------------------------------|----------------------------------------------------------------------------------------|
| Abierto, declarativo,<br>y desempaquetado | <i>El estilo, abierto y declarativo, normal<br/>tal como se usa en Prolog y Scheme</i> |
| Seguro, declarativo,<br>y desempaquetado  | <i>El estilo TAD</i>                                                                   |
| Seguro, declarativo,<br>y empaquetado     | <i>El estilo APD con objetos<br/>declarativos</i>                                      |
| Seguro, con estado,<br>y empaquetado      | <i>El estilo APD con objetos con estado,<br/>tal como se usa en Smalltalk y Java</i>   |
| Seguro, con estado,<br>y desempaquetado   | <i>El estilo TAD con “valores” con estado</i>                                          |

**Figura 6.2:** Cinco maneras de crear una pila.

Una operación se ejecuta invocando al objeto e informándole cuál operación realizar. Esto, algunas veces, se denomina “enviar un mensaje al objeto;” sin embargo esto produce la impresión errada pues no existe envío de mensajes en el sentido del capítulo 5. Una invocación a un objeto es sincrónica, como la invocación a un procedimiento: La siguiente instrucción sólo podrá ser ejecutada una vez ésta termine completamente.

El estilo objeto se ha vuelto inmensamente popular por sus importantes ventajas en cuanto a modularidad y estructura de los programas. Estas ventajas se deben a la facilidad con la cual el estilo objeto soporta el polimorfismo y la herencia. El polimorfismo se explica más adelante en esta sección. La herencia se explica en el capítulo 7, el cual se enfoca completamente en el estilo objeto.

*Estado explícito*

Una abstracción de datos tiene estado si ella usa estado explícito. Si no es así se dice que es una abstracción sin estado o declarativa. En el capítulo 3 se presentan algunos ejemplos sin estado: una pila, una cola, y un diccionario declarativos. En este capítulo se presentan algunos ejemplos con estado: la sección 6.5 trae un diccionario y un arreglo con estado. Todos estos ejemplos están en el estilo TAD.

La decisión de crear una abstracción con estado o sin estado depende de las intenciones en cuanto a modularidad, brevedad, y facilidad de razonamiento. Resaltamos que los “valores” en una abstracción de datos, como se definió arriba, pueden tener estado. Esto es un ligero abuso de terminología, pues los valores básicos que definimos en este libro son, todos, sin estado.

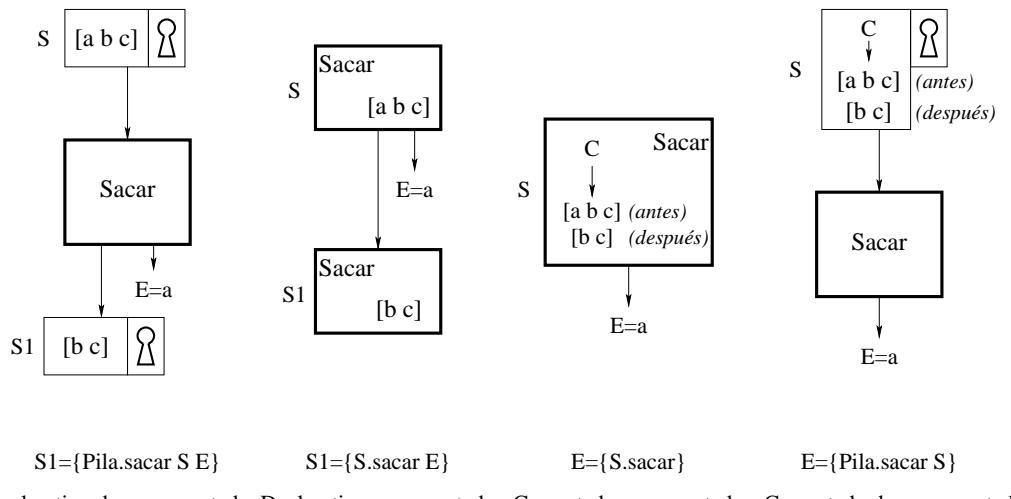


Figura 6.3: Cuatro versiones de una pila segura.

#### 6.4.2. Variaciones sobre una pila

Tomemos el tipo  $\langle \text{Pila } T \rangle$  de la sección 3.7 y miremos cómo adaptarlo a algunas de las ocho posibilidades. Presentamos cinco posibilidades útiles. Empezamos por la más sencilla, la versión abierta y declarativa, y luego la utilizamos para construir cuatro versiones diferentes, seguras. En la figura 6.2 se resumen todas, mientras en la figura 6.3 se presenta una ilustración gráfica de las cuatro versiones seguras y sus diferencias. En esta figura, las cajas de borde grueso etiquetadas “Sacar” son procedimientos que se pueden invocar. Las flechas entrantes representan las entradas al procedimiento, mientras que las flechas salientes representan sus salidas. Las cajas de borde delgado con cerradura son estructuras de datos envueltas (ver sección 3.7.5) usadas como entradas o producidas como salidas del procedimiento `Sacar`. Las estructuras de datos envueltas sólo pueden ser desenvueltas dentro del los procedimientos `Sacar`. Dos de los procedimientos `Sacar` (el segundo y el tercero) encapsulan datos, por sí mismos, utilizando alcance léxico.

##### Pila abierta y declarativa

Creamos el marco para las versiones seguras, presentando primero la funcionalidad básica de la pila en la forma más sencilla:

*Estado explícito*

```
declare
local
 fun {PilaNueva} nil end
 fun {Colocar S E} E|S end
 fun {Sacar S ?E} case S of X|S1 then E=X S1 end end
 fun {EsVacia S} S==nil end
in
 Pila=pila(nueva:PilaNueva colocar:Colocar sacar:Sacar
esVacia:EsVacia)
end
```

Pila es un módulo que agrupa las operaciones sobre pilas.<sup>5</sup> Esta versión es abierta, declarativa y desempaquetada.

*Pila declarativa, desempaquetada, y segura*

La versión anterior la podemos volver segura utilizando una pareja Envolver/Desenvolver, tal como se vió en la sección 3.7:

```
declare
local Envolver Desenvolver
 {NuevoEmpacador Envolver Desenvolver}
 fun {PilaNueva} {Envolver nil} end
 fun {Colocar S E} {Envolver E|{Desenvolver S}} end
 fun {Sacar S ?E}
 case {Desenvolver S} of X|S1 then E=X {Envolver S1} end end
 fun {EsVacia S} {Desenvolver S}==nil end
in
 Pila=pila(nueva:PilaNueva colocar:Colocar sacar:Sacar
esVacia:EsVacia)
end
```

Esta versión es segura, declarativa, y desempaquetada. La pila se desempaquetá cuando entra en una operación del TAD y se empaqueta cuando la operación termina. Por fuera del TAD, la pila siempre está empaquetada.

*Pila declarativa, empaquetada, y segura*

Hagamos ahora una versión empaquetada de la pila declarativa. La idea es ocultar la pila dentro de las operaciones, de manera que no pueda ser separada de ellas. A continuación, se presenta cómo se programa esto:

---

5. Note que Pila es una variable global que requiere un **declare**, aunque se encuentre dentro de una declaración **local**.

```

local
 fun {ObjetoPila S}
 fun {Colocar E} {ObjetoPila E|S} end
 fun {Sacar ?E}
 case S of X|S1 then E=X {ObjetoPila S1} end end
 fun {EsVacia} S==nil end
 in pila(colocar:Colocar sacar:Sacar esVacia:EsVacia) end
 in
 fun {PilaNueva} {ObjetoPila nil} end
 end

```

Esta versión es segura, declarativa, y empaquetada. Esto demuestra un hecho extraordinario: para crear una abstracción de datos segura no se necesita ni estado explícito ni nombres. Se puede lograr solamente con programación de alto orden. La función ObjetoPila toma una lista S y devuelve el registro pila(colocar:Colocar sacar:Sacar esVacia:EsVacia), con valores de tipo procedimiento, en el cual S está oculta gracias al alcance léxico. Este registro representa el objeto y sus campos son los métodos. Es muy diferente del caso declarativo desempaquetado, donde el registro representa un módulo y sus campos son las operaciones del TAD. Este es un ejemplo de uso:

```

declare S1 S2 S3 E in
 S1={PilaNueva}
 {Browse {S1.esVacia}}
 S2={S1.colocar 23}
 S3={S2.sacar E}
 {Browse E}

```

Como esta versión es a la vez empaquetada y segura, podemos considerarla como una forma declarativa de programación orientada a objetos. La pila S1 es un objeto declarativo.

#### *Pila con estado, empaquetada, y segura*

Construyamos ahora una versión con estado de la pila. Invocar PilaNueva crea un objeto pila nuevo:

```

fun {PilaNueva}
 C={NewCell nil}
 proc {Colocar E} C:=E|@C end
 fun {Sacar} case @C of X|S1 then C:=S1 X end end
 fun {EsVacia} @C==nil end
 in
 pila(colocar:Colocar sacar:Sacar esVacia:EsVacia)
 end

```

Esta versión es segura, con estado, y empaquetada. De manera similar a la versión declarativa empaquetada, el objeto se representa por un registro de valores de tipo procedimiento. Esta versión provee la funcionalidad básica de la programación orientada a objetos, a saber, un grupo de operaciones (“métodos”) con un estado interno oculto. El resultado de invocar PilaNueva es una instancia del objeto con los tres métodos: colocar, sacar, y esVacia.

*Estado explícito***Pila con estado, empaquetada, y segura (con procedimiento de despacho)**

Esta es otra manera de implementar una pila con estado, empaquetada, y segura. Se utiliza una declaración **case** dentro de un procedimiento en lugar de un registro:

```
fun {PilaNueva}
 C={NewCell nil}
 proc {Colocar E} C:=E|@C end
 fun {Sacar} case @C of X|S1 then C:=S1 X end end
 fun {EsVacia} @C==nil end
in
 proc {$ Msj}
 case Msj
 of colocar(X) then {Colocar X}
 [] sacar(?E) then E={Sacar}
 [] esVacia(?B) then B={EsVacia}
 end
 end
end
```

Esto se denomina procedimiento de despacho en contraste con la versión previa, la cual utiliza un registro de despacho. Con el procedimiento de despacho, un objeto se invoca como `{S colocar(X)}`. Con el registro de despacho, la misma invocación se escribe `{S.colocar X}`. La técnica de procedimientos de despacho se utilizará a lo largo del capítulo 7.

**Pila con estado, desempaquetada, y segura**

Es posible combinar el envolvimiento con celdas para crear una versión que sea segura, con estado, y desempaquetada. Este estilo es poco usado en la programación orientada a objetos, pero merece ser conocido más ampliamente. Aquí no se utiliza directamente la programación de alto orden. Cada operación tiene un argumento de tipo pila en lugar de los dos de la versión declarativa:

```
declare
local Envolver Desenvolver
{NuevoEmpacador Envolver Desenvolver}
fun {PilaNueva} {Envolver {NewCell nil}} end
proc {Colocar S E} C={Desenvolver S} in C:=E|@C end
fun {Sacar S}
 C={Desenvolver S} in case @C of X|S1 then C:=S1 X end end
 fun {EsVacia S} @{Desenvolver S}==nil end
in
 Pila=pila(nueva:PilaNueva colocar:Colocar sacar:Sacar
esVacia:EsVacia)
end
```

En esta versión, `PilaNueva` sólo necesita `Envolver` y las otras funciones sólo necesitan `Desenvolver`. Al igual que con la versión declarativa, desempaquetada, agrupamos las cuatro operaciones en un módulo.

### 6.4.3. Polimorfismo

En lenguaje cotidiano, el polimorfismo es la capacidad de una entidad de tomar varias formas. En el contexto de la abstracción de datos, decimos que una operación es *polimórfica* si funciona correctamente con argumentos de diferentes tipos.

El polimorfismo es importante para organizar los programas de manera que sean más fáciles de mantener. En particular, el polimorfismo permite que un programa distribuya adecuadamente la responsabilidad entre las partes que lo componen [18]. Una sola responsabilidad no debería dispersarse entre varios componentes, sino que, mejor, debe concentrarse en un solo sitio si es posible.

Consideré la analogía siguiente: un paciente va a ver al doctor especialista en el dominio de la enfermedad del paciente. El paciente solicita al doctor una cura. Dependiendo de la especialidad del doctor, esto puede significar cosas diferentes (tales como prescribir un medicamento o realizar una cirugía). En un programa de computador, el objeto paciente invoca la operación *curar* del objeto doctor. El polimorfismo significa que el objeto doctor puede ser de diferentes tipos, siempre que entienda la operación *curar*. El objeto paciente no sabe cómo curarse a sí mismo; no debería *tener* que saber. El objeto doctor entiende la operación *curar* y hará lo correcto cuando le toque.

Todas las organizaciones de abstracciones de datos, que hemos presentado antes, pueden proveer polimorfismo. La idea básica es sencilla. Suponga que un programa funciona correctamente con una abstracción de datos particular. Entonces, también podría funcionar correctamente con cualquier otra abstracción de datos que tuviera la misma interfaz. La cuestión importante aquí es la corrección: ¿el programa es aún correcto con la otra abstracción de datos? Lo será, si la otra abstracción de datos satisface las mismas propiedades matemáticas que, el programa supone que, la abstracción de datos original satisface. Es decir, el razonamiento que demuestra la corrección con la abstracción original debería poder aplicarse con la otra abstracción.

El estilo objeto tiene una ventaja sobre el estilo TAD en el hecho que el polimorfismo es particularmente más fácil de expresar. Aunque el polimorfismo se puede expresar en el estilo TAD, es más incómodo pues requiere de módulos de primera clase. El estilo TAD tiene una ventaja diferente: ofrece mayor libertad para hacer implementaciones eficientes. Presentamos un ejemplo para mostrar claramente estas diferencias. Primero definimos un tipo Colección y lo implementamos en ambos estilos, estilo TAD y estilo objeto. Luego, agregamos una nueva operación, unión, al tipo Colección y extendemos ambas implementaciones.

#### Ejemplo: un tipo Colección

Digamos que tenemos un tipo Colección con tres operaciones: una operación *agregar* para añadir un elemento, una operación *extraer* para sacar un elemento, y una operación *esVacia* para comprobar si la colección está vacía. Comenzamos el ejemplo implementando el tipo Colección en ambos estilos, el estilo TAD y el estilo

*Estado explícito*

objeto. Implementamos la colección como un TAD utilizando la pila con estado, desempaquetada:

```
local Envolver Desenvolver
 {NuevoEmpacador Envolver Desenvolver}
 fun {NuevaColección} {Envolver {Pila.nueva}} end
 proc {Aregar C X} S={Desenvolver C} in {Pila.colocar S X} end
 fun {Extraer C} S={Desenvolver C} in {Pila.sacar S} end
 fun {EsVacia C} {Pila.esVacia {Desenvolver C}} end
in
 Colección=colección(nueva:NuevaColección agregar:Agregar
extraer:Extraer esVacia:EsVacia)
end
```

Este es un ejemplo de uso:

```
C={Colección.nueva}
{Colección.agregar C 1}
{Colección.agregar C 2}
{Browse {Colección.extraer C}}
{Browse {Colección.extraer C}}
```

Ahora implementemos la colección como un objeto, utilizando la pila con estado empaquetada:

```
fun {NuevaColección}
 S={PilaNueva}
 proc {Aregar X} {S.colocar X} end
 fun {Extraer X} {S.sacar} end
 fun {EsVacia} {S.esVacia} end
in
 colección(agregar:Agregar extraer:Extraer esVacia:EsVacia)
end
```

Este es un ejemplo de uso, haciendo las mismas cosas que hicimos con el TAD:

```
C={NuevaColección}
{C.agregar 1}
{C.agregar 2}
{Browse {C.extraer}}
{Browse {C.extraer}}
```

***Agregando una operación unión en el caso TAD***

Nos gustaría extender el tipo Colección con una operación unión que tome todos los elementos de una colección y los agregue a otra colección. En el estilo TAD esta operación se invocará {Colección.unión C1 C2}, donde todos los elementos de C2 se agregan a C1, dejando C2 vacía. Para implementar unión, introducimos primero una abstracción de control:

```
proc {DoUntil BF S}
 if {BF} then skip else {S} {DoUntil BF S} end
end
```

Ésta ejecuta {S} mientras que {BF} devuelva **false**. Con DoUntil, podemos implementar el tipo Colección como se muestra a continuación (como una extensión

de la implementación original):

```
local Envolver Desenvolver
...
proc {Unión C1 C2}
S1={Desenvolver C1} S2={Desenvolver C2} in
{DoUntil fun {$} {Pila.esVacia S2} end
 proc {$} {Pila.colocar S1 {Pila.sacar S2}} end}
end
in
 Colección=colección(... unión:Unión)
end
```

Note que esta implementación utiliza tanto la representación interna de C1 como la de C2, i.e., ambas pilas. Este es un ejemplo de uso:

```
C1={Colección.nueva} C2={Colección.nueva}
for I in [1 2 3] do {Colección.agregar C1 I} end
for I in [4 5 6] do {Colección.agregar C2 I} end
{Colección.unión C1 C2}
{Browse {Colección.esVacia C2}}
{DoUntil fun {$} {Colección.esVacia C1} end
 proc {$} {Browse {Colección.extraer C1}} end}
```

Podemos hacer una segunda implementación que solamente utilice las interfaces externas de C1 y C2:

```
local Envolver Desenvolver
...
proc {Unión C1 C2}
{DoUntil fun {$} {Colección.esVacia C2} end
 proc {$} {Colección.agregar C1 {Colección.extraer C2}}
end
end
in
 Colección=colección(... unión:Unión)
end
```

En resumen, tenemos la alternativa de utilizar o no la representación interna de cada colección. Esto nos da la posibilidad de lograr una implementación más eficiente, con la advertencia que si utilizamos la representación interna, perdemos polimorfismo.

#### *Agregando una operación unión en el caso objeto*

Agreguemos ahora la operación unión al objeto Colección. En el estilo objeto la invocación será {C1 unión(C2)}. La implementación es así:

*Estado explícito*

```
fun {NuevaColección}
 S1={PilaNueva}
 ...
 proc {Unión C2}
 {DoUntil C2.esVacía
 proc {$} {S1.colocar {C2.extraer}} end}
 end
 in
 colección(... unión:Unión)
 end
```

Esta implementación utiliza la representación interna de c1, pero la interfaz externa de c2. ¡Esta es una diferencia crucial con el estilo TAD! ¿Podemos hacer una implementación en el estilo objeto que utilice ambas representaciones internas, como lo hicimos en el estilo TAD? La respuesta sencilla es no, no sin romper la encapsulación del objeto c2.

Para completar el caso objeto, mostramos cómo hacer una implementación objeto que sólo utilice interfaces externas:

```
fun {NuevaColección}
 ...
 proc {Unión C2}
 {DoUntil C2.esVacía
 proc {$} {This.agregar {C2.extraer}} end}
 end
 This=colección(... unión:Unión)
 in
 This
 end
```

Note que el objeto c1 se referencia a sí mismo a través de la variable This.

*Discusión*

¿Cómo escoger entre los estilos TAD y objeto? Comparémoslos:

- El estilo TAD puede ser más eficiente porque permite acceder a ambas representaciones internas. Utilizar una interfaz externa puede ser menos eficiente pues la interfaz podría no tener implementadas todas las operaciones que necesitamos.
- Algunas veces el estilo TAD es el único que funciona. Suponga que estamos definiendo un tipo entero y que deseamos definir la suma de dos enteros. Si no existen otras operaciones definidas sobre los enteros, entonces realmente necesitamos tener acceso a las representaciones internas. Por ejemplo, la representación puede estar en una forma binaria, de manera que podamos hacer la suma utilizando una instrucción de máquina. Esto justifica el por qué los lenguajes populares orientados a objeto, tal como Java, utilizan el estilo TAD para las operaciones primitivas de tipos básicos como los enteros.
- El estilo objeto provee el polimorfismo “gratis.” Suponga que definimos un segundo tipo colección (con un objeto D) que tiene la misma interfaz del primero.

Entonces los dos tipos colección pueden interoperar aunque sus implementaciones sean independientes. En resumen, {C unión(D)} funciona sin escribir código nuevo.<sup>6</sup>

- El estilo objeto no está limitado a objetos secuenciales. En particular, los objetos flujo (ver capítulo 4), los objetos puerto (ver capítulo 5), y los objetos activos (los cuales veremos en la sección 7.8) son todos objetos tal y como los definimos en esta sección. Todos proveen polimorfismo en la misma forma que los objetos secuenciales.
- El estilo TAD puede proveer polimorfismo si el lenguaje tiene módulos de primera clase. Suponga que definimos un segundo tipo colección como el módulo Colección2. La implementación de unión tiene que asegurarse que C2 siempre utilice una operación de Colección2. Podemos hacer esto agregando Colección2 como argumento de la operación unión , de manera que la invocación será {Unión C1 Colección2 C2}. Esto nos lleva a la definición siguiente:

```
proc {Unión C1 Colección2 C2}
 {DoUntil fun {$} {Colección2.esVacia C2} end
 proc {$} {Colección.agregar C1 {Colección2.extraer C2}} end}
end
Colección=colección(... unión:Unión)
```

Esta técnica se utiliza frecuentemente en lenguajes con módulos de primera clase, tal como Erlang.

- Si utilizamos el estilo TAD sin módulos de primera clase, entonces debemos escribir código nuevo para que los tipos puedan interoperar. Tenemos que escribir una operación unión que conozca *ambas* representaciones internas. Si tenemos tres o más tipos colección, todo es aún más complicado: tenemos que implementar la operación para todas las combinaciones posibles de dos de esos tipos.

Los lenguajes orientados a objetos utilizan el estilo objeto por defecto, lo cual los hace polimórficos en consecuencia. Esta es una de las principales ventajas de la programación orientada a objetos, la cual exploraremos más adelante en el capítulo 7. Los lenguajes orientados a TADs también pueden ser polimórficos, pero sólo si cuentan con módulos de primera clase.

### Otras clases de polimorfismo

La clase de polimorfismo discutida en esta sección se denomina polimorfismo *universal*. Hay un segundo concepto, también considerado una clase de polimorfismo, a saber, el polimorfismo *ad-hoc* [28]. En el polimorfismo ad-hoc, se ejecuta código diferente para argumentos de tipos diferentes. En el polimorfismo universal, se ejecuta el mismo código para todos los tipos de argumentos admisibles. Un ejemplo de polimorfismo ad-hoc es el operador de sobrecarga, donde el mismo operador

---

6. Esto puede parecer milagroso. Funciona porque la implementación de unión de C sólo invoca la interfaz de D. ¡Piense sobre esto!

*Estado explícito*

puede representar muchas funciones diferentes. Por ejemplo, el operador de adición `+` se sobrecarga en muchos lenguajes. El compilador escoge la función de adición apropiada dependiendo de los tipos de los argumentos (e.g., enteros o flotantes).

#### 6.4.4. Paso de parámetros

Las operaciones de abstracción de datos pueden tener argumentos y resultados. Se han ideado bastantes y diversos mecanismos para pasar los argumentos y los resultados entre un programa que invoca y una abstracción. Iremos brevemente sobre los más destacados. Para cada mecanismo, presentaremos un ejemplo en una sintaxis al estilo Pascal, y codificaremos el ejemplo en el modelo con estado de este capítulo. Esta codificación puede ser vista como una definición semántica del mecanismo. Utilizamos Pascal por su simplicidad. Java es un lenguaje más popular, pero explicar su sintaxis más elaborada no es apropiado para esta sección. En la sección 7.7 se presenta un ejemplo de la sintaxis de Java.

##### *Paso de parámetros por referencia*

La identidad de una entidad del lenguaje se pasa al procedimiento. Entonces, el procedimiento puede usar libremente esta entidad del lenguaje. Este es el mecanismo primitivo utilizado por los modelos de computación del libro para todas las entidades del lenguaje incluyendo variables y celdas.

En los lenguajes imperativos el paso de parámetros por referencia, significa algo ligeramente distinto con alguna frecuencia. Ellos suponen que la referencia se almacena en una celda local al procedimiento. En nuestra terminología, esto es un paso de parámetros por valor donde la referencia es considerada como un valor (ver abajo). Cuando se estudia un lenguaje que tiene paso de parámetros por referencia, recomendamos mirar cuidadosamente en la definición del lenguaje lo que eso significa.

##### *Paso de parámetros por variable*

Este es un caso especial de paso de parámetros por referencia. La identidad de una celda se pasa al procedimiento. El siguiente es un ejemplo:

```
procedure cuad(var a:integer);
begin
 a:=a*a
end
var c:integer;
c:=25;
cuad(c);
browse(c);
```

Codificamos este ejemplo así:

```
proc {Cuad A}
 A:=@A*@A
end
local
 C={NewCell 0}
in
 C:=25
 {Cuad C}
 {Browse @C}
end
```

En la invocación {Cuad C}, la A dentro de Cuad es un sinónimo de la C por fuera de él.

#### *Paso de parámetros por valor*

Se pasa un valor al procedimiento y se coloca en una celda local al mismo. La implementación es libre de copiar el valor o pasar una referencia, mientras que el procedimiento no pueda cambiar el valor en el ambiente de invocación. El siguiente es un ejemplo:

```
procedure cuad(a:integer);
begin
 a:=a+1;
 browse(a*a)
end;
cuad(25);
```

Codificamos este ejemplo así:

```
proc {Cuad D}
 A={NewCell D}
in
 A:=@A+1
 {Browse @A*@A}
end
{Cuad 25}
```

La celda A se inicializa con el argumento de Cuad. El lenguaje Java utiliza paso de parámetros por valor tanto para valores como para referencias a objetos. Esto se explica en la sección 7.7.

#### *Paso de parámetros por valor-resultado*

Esta es una modificación del paso de parámetros por variable. Cuando el procedimiento se invoca, el contenido de una celda (i.e., una variable mutable) se coloca dentro de otra variable mutable local al procedimiento. Cuando el procedimiento termina, el contenido de esta última se coloca en la primera. El siguiente es un ejemplo:

*Estado explícito*

```
procedure cuad(inout a:integer);
begin
 a:=a*a
end
var c:integer;
c:=25;
cuad(c);
browse(c);
```

Aquí se usa la palabra reservada “*inout*” para indicar que el paso de parámetros es por valor-resultado. Este tipo de paso de parámetros se utiliza en el lenguaje Ada. Codificamos este ejemplo así:

```
proc {Cuad A}
D={NewCell @A}
in
D:=@D*@D
A:=@D
end
local
C={NewCell 0}
in
C:=25
{Cuad C}
{Browse @C}
end
```

Hay dos variables mutables: una dentro de Cuad (a saber, D) y una por fuera (a saber, C). Apenas empieza la ejecución de una invocación a Cuad, se asigna el contenido de C a D. Justo antes de terminar la ejecución de la invocación, se asigna el contenido de D a C. Durante la ejecución de la invocación a Cuad, las modificaciones a D son invisibles en el exterior.

*Paso de parámetros por nombre*

Este mecanismo es el más complejo. En él se crea una función por cada argumento. El invocar la función devuelve el nombre de una celda, i.e., la dirección de una variable mutable. Cada vez que se necesita el argumento, la función se invoca. Una función utilizada de esta manera se denomina una suspensión<sup>7</sup> o una promesa.<sup>8</sup> Las suspensiones se inventaron originalmente para la implementación de Algol 60. El siguiente es un ejemplo:

---

7. Este es el significado original de este término. También se usa en una forma más general para referirse a una clausura con alcance léxico.

8. Nota del traductor: ésta parece ser la traducción más adecuada para el término original en inglés: *thunk*.

```

procedure cuad(callbyname a:integer);
begin
 a:=a*a
end;
var c:integer;
c:=25;
cuad(c);
browse(c);

```

Se usa la palabra reservada “callbyname” para indicar paso de parámetros por nombre. Codificamos este ejemplo así:

```

proc {Cuad A}
 {A} :=@{A}*@{A}
end
local C={NewCell 0} in
 C:=25
 {Cuad fun {$} C end}
 {Browse @C}
end

```

El argumento **A** es una función que cuando se evalúa devuelve el nombre de una variable mutable. La función se evalúa cada vez que el argumento se necesita. El paso de parámetros por nombre produce resultados contraintuitivos si se utilizan índices de arreglos como argumentos (ver ejercicios, en la sección 6.10).

#### *Paso de parámetros por necesidad*

Esta es una modificación al paso de parámetros por nombre, en la cual la función se invoca a lo sumo una vez. Su resultado es almacenado y utilizado para evaluaciones posteriores. Esta es una forma de codificar el paso de parámetros por necesidad para el ejemplo de paso de parámetros por nombre:

```

proc {Cuad A}
 B={A}
in
 B :=@B*@B
end
local C={NewCell 0} in
 C:=25
 {Cuad fun {$} C end}
 {Browse @C}
end

```

El argumento **A** se evalúa cuando el resultado se necesite. La variable local **B** almacena el resultado. Si el argumento se necesita de nuevo, entonces se utiliza **B**, evitándose así volver a evaluar la función. En el ejemplo de **Cuad** esto es fácil de implementar pues el resultado claramente se necesita tres veces. Si no es claro, a partir de la inspección, si el resultado se necesita, entonces la evaluación perezosa se puede utilizar para implementar directamente el paso de parámetros por necesidad (ver ejercicios, en la sección 6.10).

*Estado explícito*

El paso de parámetros por necesidad es exáctamente el mismo concepto de evaluación perezosa. El término “invocación por necesidad” se utiliza más frecuentemente en un lenguaje con estado, donde el resultado de la evaluación de la función puede ser el nombre de una celda (una variable mutable). El paso de parámetros por nombre es evaluación perezosa sin memorización. El resultado de la evaluación de la función no se almacena, de manera que se evalúa de nuevo cada vez que se necesita.

**Discusión**

¿Cuál de estos mecanismos (si hay alguno) es el “correcto” o el “mejor”? Esto ha sido objeto de bastante discusión (ver, e.g., [109]). El objetivo del enfoque del lenguaje núcleo es factorizar los programas en un conjunto pequeño de conceptos significativos para el programador. Así se justifica, en el caso del paso de parámetros, el utilizar el paso de parámetros por referencia como el mecanismo primitivo que subyace a los otros mecanismos. A diferencia de los otros mecanismos, el paso de parámetros por referencia no depende de conceptos adicionales tales como celdas o valores de tipo procedimiento. Este mecanismo tiene una semántica formal sencilla y se implementa eficientemente. Por otro lado, esto no significa que el paso de parámetros por referencia sea siempre el mecanismo correcto para todo programa. Otros mecanismos de paso de parámetros se pueden codificar combinando paso de parámetros por referencia con celdas y valores de tipo procedimiento. Muchos lenguajes ofrecen estos mecanismos como abstracciones lingüísticas.

**6.4.5. Habilidades revocables**

Algunas veces es necesario controlar la seguridad de una abstracción de datos. Mostraríamos cómo utilizar el estado explícito para construir habilidades revocables. Este es un ejemplo de abstracción de datos que controla la seguridad de otras abstracciones de datos. En el capítulo 3 se introdujo el concepto de habilidad, la cual da a su propietario un derecho irrevocable de hacer algo. Algunas veces nos gustaría otorgar, en su lugar, un derecho revocable, i.e., un derecho que puede ser eliminado. Podemos implementar esto con estado explícito. Sin pérdida de generalidad, suponemos que la habilidad se representa con un procedimiento de un argumento.<sup>9</sup> A continuación presentamos un procedimiento genérico que recibe una habilidad cualquiera y devuelve una versión revocable de esa habilidad:

---

9. Este es un caso importante porque cubre el sistema objeto del capítulo 7.

```

proc {Revocable Obj ?R ?ObjR}
 C={NewCell Obj}
in
 proc {R}
 C:=proc {$ M} raise errorHabilidadRevocada end end
 end
 proc {ObjR M}
 {@C M}
 end
end

```

Dado cualquier procedimiento de un argumento Obj, el procedimiento Revocable devuelve en ObjR una versión revocable de Obj, y un procedimiento sin argumentos R para revocar la habilidad de ObjR cuando se requiera. En principio, ObjR reenvía todos sus mensajes a Obj. Pero después de ejecutar {R}, invocar a ObjR lanza invariablemente la excepción errorHabilidadRevocada. Este es un ejemplo:

```

fun {NuevoRecolector}
 Lst={NewCell nil}
in
 proc {$ M}
 case M
 of agr(X) then T in {Exchange Lst T X|T}
 [] obt(L) then L={Reverse @Lst}
 end
 end
end

declare H R in
 H={Revocable {NuevoRecolector} R}

```

La función NuevoRecolector crea una instancia de una abstracción de datos que denominamos un recolector. Ésta tiene dos operaciones, agr y obt. Con agr, se pueden recolectar ítems en una lista en el orden en que llegan. Con obt, se recupera, en cualquier momento, el valor actual de la lista. Podemos volver revocable el recolector, de manera que cuando haya terminado su trabajo, se pueda hacer inoperante invocando a R.

## 6.5. Colecciones con estado

Un clase importante de TAD es la colección, la cual agrupa un conjunto de valores parciales dentro de una entidad compuesta. Existen diferentes tipos de colección dependiendo de las operaciones que se provean. Por un lado, distinguimos las colecciones indexadas y las no indexadas, dependiendo de si existe o no un acceso rápido a los elementos individuales (a través de un índice). Por otro lado, distinguimos las colecciones que se pueden extender y las que no, dependiendo de si el número de elementos es variable o fijo. Presentamos una visión general, breve, de estas diferentes clases de colecciones, empezando con las colecciones indexadas.

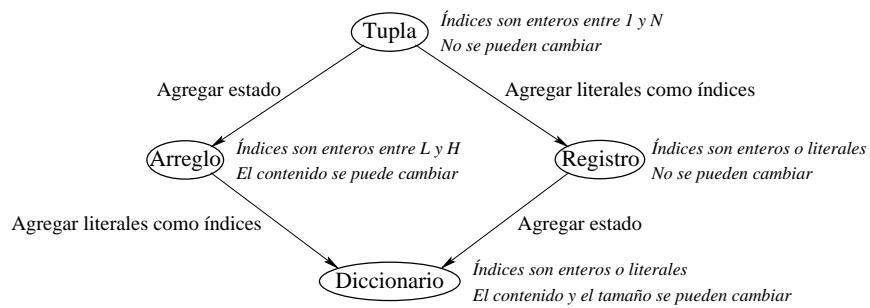


Figura 6.4: Variedades diferentes de colecciones indexadas.

### 6.5.1. Colecciones indexadas

En el contexto de la programación declarativa, ya hemos visto dos clases de colecciones indexadas, a saber, las tuplas y los registros. Podemos añadir estado a estos dos tipos de datos, permitiéndoles que se actualicen de cierta manera. Las versiones con estado de las tuplas y los registros se denominan arreglos y diccionarios, respectivamente.

En total, tenemos cuatro clases diferentes de colecciones indexadas, cada una con un compromiso particular entre expresividad y eficiencia (ver figura 6.4). ¿Con tal proliferación, cómo se escoge cuál usar? En la sección 6.5.2 se comparan las cuatro y se aconseja cómo escoger entre ellas.

#### Arreglos

Un arreglo es una asociación de enteros con valores parciales. El dominio es el conjunto de enteros consecutivos entre un límite inferior y un límite superior. El dominio se fija cuando se declara el arreglo y no se puede cambiar después. Tanto el acceso como la modificación de un elemento del arreglo se hacen en tiempo constante. Si se necesita cambiar el dominio o si el dominio no se conoce al momento de la declaración del arreglo, entonces se debería usar un diccionario en lugar de un arreglo. El sistema Mozart provee los arreglos como un TAD predefinido en el módulo `Estas`. Estas son unas de las operaciones más comunes:

- `A={NewArray L H I}` devuelve un arreglo nuevo, con índices entre `L` y `H`, inclusive, con cada entrada inicializada en `I`.
- `{Array.put A I X}` asocia la posición `I` del arreglo `A` con `X`. Esto también se puede escribir: `A.I:=X`.
- `X={Array.get A I}` devuelve la asociación de `I` en `A`. También se puede escribir: `X=A.I`.
- `L={Array.low A}` devuelve el índice inferior de `A`.

- `H={Array.high A}` devuelve el índice superior de `A`.
- `R={Array.toRecord L A}` devuelve un registro etiquetado con `L` y con los mismos ítems que el arreglo `A`. El registro es una tupla si y sólo si el índice inferior es 1.
- `A={Tuple.toArray T}` devuelve un arreglo con límites entre 1 y `{width T}`, donde los elementos del arreglo son los elementos de `T`.
- `A2={Array.clone A}` devuelve un arreglo nuevo exáctamente con los mismos índices y contenidos que el arreglo `A`.

Existe una relación muy cercana entre arreglos y tuplas. Cada uno de ellos asocia un conjunto de enteros consecutivos con valores parciales. La diferencia esencial es que las tuplas no tienen estado mientras que los arreglos sí lo tienen. Una tupla tiene un contenido fijo para sus campos, mientras que en un arreglo los contenidos se pueden cambiar. Es posible crear una tupla completamente nueva que difiera solamente en un campo con una tupla existente, utilizando las operaciones `Adjoin` y `AdjoinAt`. Estas operaciones consumen tiempo y memoria proporcional al número de campos de la tupla. La operación `put` de un arreglo es una operación de tiempo constante y, por tanto, mucho más eficiente.

### Diccionarios

Un diccionario es una asociación de constantes sencillas (átomos, nombres, o enteros) con valores parciales. Tanto el dominio como el rango de la asociación se pueden cambiar. Un ítem es una pareja formada por una constante sencilla y un valor parcial. Los ítems pueden ser consultados, modificados, agregados o eliminados durante la ejecución. Todas las operaciones son eficientes: consultar y modificar se hacen en tiempo constante, y agregar/eliminar se hace en tiempo constante amortizado. Por tiempo constante amortizado se entiende que una secuencia de  $n$  operaciones de agregar o eliminar ítems se realiza en tiempo proporcional a  $n$ , cuando  $n$  es grande. Esto significa que cada operación individual puede no tomar tiempo constante, pues ocasionalmente el diccionario tiene que ser reorganizado internamente, pero las reorganizaciones ocurren raramente. La memoria activa requerida por un diccionario siempre es proporcional al número de ítems en la asociación. No existen límites al número de campos en la asociación, diferentes al tamaño de la memoria del sistema. En la sección 3.7.3 se presentan algunas medidas aproximadas comparando diccionarios con estado y diccionarios declarativos. El sistema Mozart provee diccionarios como un TAD en el módulo `Dictionary`. Algunas de las operaciones más comunes son:

- `D={NewDictionary}` devuelve un diccionario vacío nuevo.
- `{Dictionary.put D LI X}` coloca en `D` la asociación de `LI` con `X`. También se puede escribir: `D.LI:=X`.
- `X={Dictionary.get D LI}` devuelve la asociación de `LI` en `D`. También se puede escribir: `X=D.LI`, i.e., con la misma notación para los registros.

*Estado explícito*

- `X={Dictionary.condGet D LI Y}` devuelve la asociación de `LI` en `D` si existe. Sino, devuelve `Y`. Esta es una pequeña variación de `get`, extremadamente útil en la práctica.
- `{Dictionary.remove D LI}` elimina la asociación de `LI` en `D`.
- `{Dictionary.member D LI B}` comprueba si `LI` existe en `D`, y liga `B` al resultado booleano.
- `R={Dictionary.toRecord L D}` devuelve un registro etiquetado con `L` y con los mismos ítems que el diccionario `D`. El registro es como una “instantánea” del estado del diccionario en un momento dado.
- `D={Record.toDictionary R}` devuelve un diccionario con los mismos ítems del registro `R`. Esta operación y la anterior son útiles para salvar y recuperar el estado del diccionario.
- `D2={Dictionary.clone D}` devuelve un diccionario nuevo exáctamente con los mismos ítems que el diccionario `D`.

Existe una relación muy cercana entre diccionarios y registros. Cada uno de ellos asocia constantes sencillas con valores parciales. La diferencia esencial es que los registros no tienen estado mientras que los diccionarios si lo tienen. Un registro tiene un conjunto fijo de campos y contenidos, mientras que en un diccionario el conjunto de campos y sus contenidos se pueden cambiar. De la misma manera que para las tuplas, se pueden crear registros nuevos con las operaciones `Adjoin` y `AdjoinAt`, pero éstas toman tiempo proporcional al número de campos del registro. La operación `put` de un diccionario es una operación de tiempo constante, y por tanto mucho más eficiente.

#### 6.5.2. Escogencia de una colección indexada

Las diferentes colecciones indexadas tienen diferentes compromisos en cuanto a operaciones posibles, utilización de la memoria, y tiempo de ejecución. No siempre es fácil decidir cuál tipo de colección es la más indicada para una situación específica. Examinamos las diferencias entre estas colecciones para facilitar este tipo de decisiones.

Hemos visto cuatro tipos de colecciones indexadas: tuplas, registros, arreglos, y diccionarios. Todas proveen tiempos constantes para acceder a sus elementos por medio de índices, los cuales se calculan en tiempo de ejecución. Pero aparte de esto que tienen en común ellas son bastante diferentes. En la figura 6.4 se presenta una jerarquía que muestra cómo están relacionados unos tipos de colección con los otros. Comparémoslos:

- *Tuplas*. Las tuplas son las más restrictivas, pero son las más rápidas y las que requieren menos memoria. Sus índices son enteros positivos consecutivos desde 1 a un máximo `N` el cual se especifica cuando se crea la tupla. Ellas pueden ser usadas como arreglos cuando los contenidos no se tienen que cambiar. Acceder a un campo de una tupla es extremadamente eficiente porque los campos se almacenan

consecutivamente.

■ *Registros*. Los registros son más flexibles que las tuplas porque los índices pueden ser cualquiera de los literales (átomos o nombres) y enteros. Los enteros no tienen que ser consecutivos. El tipo registro, i.e., la etiqueta y la aridad (el conjunto de índices), se especifica cuando se crea el registro. Acceder a los campos de un registro es casi tan eficiente como acceder a los campos de una tupla. Para garantizar esto, los campos de un registro se almacenan consecutivamente, al igual que con las tuplas. Esto implica que crear un nuevo tipo registro (i.e., uno para el que no existe ningún registro todavía) es mucho más costoso que crear un nuevo tipo tupla. Una tabla de hash se crea en el momento en que se crea el tipo registro. Esa tabla hash asocia cada índice con su posición relativa en el registro. Para evitar el uso de la tabla de hash en cada acceso, la posición relativa se oculta en la instrucción de acceso. Crear nuevos registros de un tipo ya existente es tan barato como crear una tupla.

■ *Arreglos*. Los arreglos son más flexibles que las tuplas porque el contenido de cada campo se puede cambiar. Acceder un campo de un arreglo es extremadamente eficiente porque los campos se almacenan consecutivamente. Los índices son enteros consecutivos entre un límite inferior y un límite superior. Los límites se especifican cuando el arreglo se crea, y no se pueden cambiar.

■ *Diccionarios*. Los diccionarios conforman el tipo de colección más general. Ellos combinan la flexibilidad de los arreglos y registros. Los índices pueden ser literales y enteros y el contenido de cada campo se puede cambiar. Los diccionarios se crean vacíos. No se necesita especificar índices. Los índices se pueden agregar y remover eficientemente, en tiempo constante amortizado. Por otro lado, los diccionarios utilizan más memoria que los otros tipos de datos (en un factor constante) y tienen tiempos de acceso inferiores (también en un factor constante). Los diccionarios se implementan como tablas de hash dinámicas.

Cada uno de estos tipos define un compromiso particular que algunas veces es el adecuado. A lo largo de los ejemplos del libro, seleccionamos el tipo de colección indexada adecuado cada vez que necesitamos uno.

### 6.5.3. Otras colecciones

#### *Colecciones no indexadas*

No siempre las colecciones indexadas son la mejor opción. Algunas veces es mejor utilizar una colección no indexada. Hemos visto dos colecciones no indexadas: las listas y los flujos. Ambos son tipos de datos declarativos que recolectan elementos en una secuencia lineal. La secuencia se puede recorrer de adelante hacia atrás. Se puede realizar cualquier número de recorridos simultáneos sobre la misma lista o flujo. Las listas son de longitud finita, fija. Los flujos se denominan también listas incompletas o listas parciales; sus colas son variables no ligadas. Esto significa que siempre pueden ser extendidas, i.e., ellas son potencialmente ilimitadas. El flujo es

*Estado explícito*

```

fun {NewExtensibleArray L H Init}
 A={NewCell {NewArray L H Init}}#Init
 proc {CheckOverflow I}
 Arr=@(A.1)
 Low={Array.low Arr}
 High={Array.high Arr}
 in
 if I>High then
 High2=Low+{Max I 2*(High-Low)}
 Arr2={NewArray Low High2 A.2}
 in
 for K in Low..High do Arr2.K:=Arr.K end
 (A.1):=Arr2
 end
 end
 proc {Put I X}
 {CheckOverflow I}
 @(A.1).I:=X
 end
 fun {Get I}
 {CheckOverflow I}
 @(A.1).I
 end
in extArray(get:Put put:Put)
end

```

**Figura 6.5:** Arreglo extensible (implementación con estado).

una de las colecciones extensibles más eficiente, tanto en uso de memoria como en tiempo de ejecución. Extender un flujo es más eficiente que agregar un nuevo índice a un diccionario y mucho más eficiente que crear un tipo de registro nuevo.

Los flujos son útiles para representar secuencias ordenadas de mensajes. Esta es una representación especialmente apropiada pues el receptor del mensaje se sincronizará automáticamente en la llegada de los mensajes nuevos. Esta es la base de un poderoso estilo de programación declarativa denominado programación por flujos (ver capítulo 4) y de su generalización a paso de mensajes (ver capítulo 5).

### *Arreglos extensibles*

Hasta ahora hemos visto dos colecciones extensibles: los flujos y los diccionarios. Los flujos se extienden eficientemente, pero sus elementos no se pueden acceder eficientemente (se necesita búsqueda lineal). Los diccionarios son más costosos de extender (pero sólo en un factor constante) y se puede acceder a ellos en tiempo constante. Una tercera colección extensible es el arreglo extensible. Este es un arreglo que se puede redimensionar cuando se llena. Tiene las ventajas de acceso en tiempo constante, y de uso de memoria significativamente menor que los diccionarios

(en un factor constante). La operación de redimensionamiento se realiza en tiempo constante amortizado, pues sólo se realiza cuando se encuentra un índice que es más grande que el tamaño actual.

Mozart no provee los arreglos extensibles como tipos predefinidos. Podemos implementarlos utilizando los arreglos estándar y las celdas. En la figura 6.5 se muestra una posible versión, la cual permite a un arreglo aumentar su tamaño pero no le permite disminuirlo. La invocación `{NewExtensibleArray L H x}` devuelve un arreglo seguro extensible con límites iniciales `L` y `H` y contenido inicial `x`. La operación `{A.put i x}` coloca `x` en el índice `i`. La operación `{A.get i}` devuelve el contenido en el índice `i`. Ambas operaciones extienden el arreglo cuando encuentran un índice que está por fuera de los límites. La operación de redimensionamiento siempre dobla, por lo menos, el tamaño del arreglo. Esto garantiza que el costo amortizado de la operación de redimensionamiento sea constante. Para incrementar la eficiencia, se pueden agregar operaciones `put` y `get` “inseguras” que no comprueben si los límites se respetan. En tal caso, es responsabilidad del programador asegurar que los índices permanezcan dentro de los límites.

---

## 6.6. Razonando con estado

Los programas que usan el estado de forma desordenada son muy difíciles de entender. Por ejemplo, si el estado es visible a lo largo de todo el programa, entonces puede ser asignado en cualquier parte. La única forma de razonar sobre un programa así es considerándolo por completo, de una sola vez. En la práctica, esto es imposible con los programas grandes. En esta sección se introduce un método, denominado afirmaciones invariantes, con las cuales el estado se puede controlar. Mostramos cómo usar el método con programas que tienen tanto partes con estado como partes declarativas. La parte declarativa aparece en forma de expresiones lógicas dentro de las afirmaciones. También explicamos el papel de la abstracción (obteniendo nuevas reglas de prueba para las abstracciones lingüísticas) y cómo tomar en cuenta la ejecución por flujo de datos.

La técnica de afirmaciones invariantes se denomina normalmente semántica axiomática, de acuerdo a Floyd, Hoare, y Dijkstra, quienes la desarrollaron inicamente en los años 1960 y 1970. Las reglas de corrección se denominaron “axiomas” y la terminología se adoptó desde entonces. Manna realizó una temprana, pero aún interesante, presentación [110].

### 6.6.1. Afirmaciones invariantes

El método de afirmaciones invariantes permite razonar independientemente sobre las partes de los programas. Esto nos devuelve una de las propiedades más fuertes de la programación declarativa. Sin embargo, esta propiedad se logra pagando el precio de una rigurosa organización de los programas. La idea básica consiste en organizar

*Estado explícito*

un programa como una jerarquía de abstracciones de datos. Cada abstracción puede utilizar otras abstracciones en su implementación. Esto conlleva a un grafo dirigido de abstracciones de datos.

Una organización jerárquica de un programa es buena para muchas más cosas que solamente para razonar sobre el programa. Lo veremos muchas veces en el libro. Encontraremos esto de nuevo en la programación basada en componentes de la sección 6.7 y en la programación orientada a objetos del capítulo 7.

Cada abstracción de datos se especifica con una serie de afirmaciones invariantes, también denominadas, simplemente, invariantes. Un invariante es una oración lógica que define una relación entre los argumentos de la abstracción y su estado interno. Cada operación de la abstracción supone cierto algún invariante y, cuando termina, asegura la verdad de otro invariante. La implementación de la operación garantiza esto. De esta manera, la utilización de invariantes desacopla la implementación de una abstracción, de su uso. Así, podemos razonar sobre cada aspecto separadamente.

Para llevar a cabo esta idea, utilizamos el concepto de afirmación. Una afirmación es una oración lógica unida a un punto dado en el programa, entre dos instrucciones. Una afirmación se puede considerar como una especie de expresión booleana (más adelante veremos exactamente cómo se diferencia de las expresiones booleanas del modelo de computación). Las afirmaciones pueden contener identificadores de variables y de celdas del programa así como variables y cuantificadores que no aparecen en el programa, pero que se utilizan para expresar una relación en particular. Por ahora, considere un cuantificador como un símbolo, tal como  $\forall$  (“para todo”) o  $\exists$  (“existe”), que se usa para expresar afirmaciones que son ciertas para todos los valores de las variables en un dominio, y no solamente para un valor.

Cada operación  $O_i$  de la abstracción de datos se especifica por medio de dos afirmaciones  $A_i$  y  $B_i$ . La especificación estipula que, si  $A_i$  es cierta justo antes de ejecutar  $O_i$ , entonces cuando  $O_i$  termine  $B_i$  será cierta. Esto lo denotamos así:

$$\{ A_i \} O_i \{ B_i \}$$

Esta especificación se denomina, algunas veces, una afirmación de corrección parcial. Es parcial, pues sólo es válida si  $O_i$  termina normalmente.  $A_i$  se denomina la precondición y  $B_i$  se denomina la poscondición. La especificación completa de una abstracción de datos consiste, entonces, de las afirmaciones de corrección parcial de cada una de sus operaciones.

### 6.6.2. Un ejemplo

Ahora que tenemos algún indicio sobre cómo proceder, presentamos un ejemplo sobre cómo especificar una abstracción de datos sencilla y cómo probar su corrección. Usamos la abstracción de pila con estado que introdujimos anteriormente. Para conservar la sencillez de la presentación, introduciremos la notación que vayamos necesitando, gradualmente, a lo largo del ejemplo. La notación no es complicada; es tan sólo una forma de escribir expresiones booleanas de manera que nos permi-

ta expresar lo que necesitamos. En la sección 6.6.3 se define de manera precisa la notación.

### *Especificando la abstracción de datos*

Empezamos por especificar la abstracción de datos en forma independiente de su implementación. La primera operación crea una instancia de una pila, empaquetada y con estado:

```
Pila={PilaNueva}
```

La función `PilaNueva` crea una nueva celda  $c$ , la cual queda oculta dentro de la pila gracias al alcance léxico, y devuelve un registro de tres operaciones, `Colocar`, `Sacar`, y `EsVacía`, con el cual se liga la variable `Pila`. Entonces, podemos decir que una especificación de `PilaNueva` es la siguiente:

```
{ true }
Pila={PilaNueva}
{ @c = nil ∧ Pila = ops(colocar:Colocar sacar:Sacar esVacía:EsVacía) }
```

La precondition es `true`, lo cual significa que no existen condiciones especiales. La notación `@c` denota el contenido de la celda  $c$ .

Esta especificación es incompleta pues no define qué significan las referencias `Colocar`, `Sacar`, y `EsVacía`. Definamos cada una de ellas de manera separada. Empecemos por `Colocar`. Ejecutar `{Pila.colocar x}` es una operación que coloca  $x$  en la pila. La especificamos como sigue:

```
{ @c = s }
{Pila.colocar x}
{ @c = x | s }
```

Ambas especificaciones, la de `PilaNueva` y la de `Pila.colocar`, mencionan la celda interna  $c$ . Esto es razonable cuando se quiere probar la corrección de la pila, pero no es razonable cuando se usa la pila, pues se desea mantener oculta la representación interna. Podemos evitar esto introduciendo un predicado `contenidoPila` con la siguiente definición:

```
contenidoPila(Pila, S) ≡ @c = S
```

donde  $c$  es la celda interna correspondiente a `Pila`. Así se oculta cualquier mención de la celda interna por parte de los programas que utilizan la pila. Entonces, las especificaciones de `PilaNueva` y `Pila.colocar` se vuelven:

```
{ true }
Pila={PilaNueva}
{ contenidoPila(Pila, nil) ∧
 Pila = ops(colocar:Colocar sacar:Sacar esVacía:EsVacía) }
```

```
{ contenidoPila(Pila, S) }
{Pila.colocar x}
{ contenidoPila(Pila, X | S) }
```

### Estado explícito

Continuamos con las especificaciones de `Pila.sacar` y `Pila.esVacía`:

```

{ contenidoPila(Pila, X|S) }
Y={Pila.sacar}
{ contenidoPila(Pila, S) ∧ Y = X }

{ contenidoPila(Pila, S) }
X={Pila.esVacía}
{ contenidoPila(Pila, S) ∧ X = (S==nil) }

```

La especificación completa de la pila consiste de estas cuatro afirmaciones de corrección parcial. Estas operaciones no dicen qué pasa si una operación sobre la pila lanza una excepción. Discutimos esto más adelante.

### Probando que la abstracción de datos es correcta

La especificación presentada arriba trata sobre cómo debería comportarse una pila. ¿Pero, la implementación se comporta realmente de esa manera? Para verificar esto, tenemos que comprobar si cada afirmación de corrección parcial es correcta para nuestra implementación. Esta es la implementación (para facilitar las cosas, hemos desanidado las declaraciones anidadas):

```

fun {PilaNueva}
 C={NewCell nil}
 proc {Colocar X} S in S=@C C:=X|S end
 fun {Sacar} S1 in
 S1=@C
 case S1 of X|S then C:=S X end
 end
 fun {EsVacía} S in S=@C S==nil end
 in
 ops(colocar:Colocar sacar:Sacar esVacía:EsVacía)
 end

```

Con respecto a esta implementación, tenemos que verificar cada una de las cuatro afirmaciones de corrección parcial que componen la especificación de la pila. Enfoquémonos en la especificación de la operación `Colocar`. Dejamos las otras tres verificaciones al lector. La definición de `Colocar` es:

```

proc {Colocar X}
 S in
 S=@C
 C:=X|S
 end

```

La precondición es  $\{ \text{contenidoPila}(\text{Pila}, s) \}$ , la cual expandimos a  $\{ @C = s \}$ , donde `C` referencia la celda interna de la pila. Esto significa que tenemos que probar:

```

{ @C = s }
S=@C
C:=X|S
{ @C = X|s }

```

La abstracción de la pila utiliza la abstracción de la celda en su implementación. Para continuar la prueba, necesitamos conocer las especificaciones de las operaciones  $@ y :=$  sobre celdas. La especificación de  $@$  es

$$\begin{aligned} \{ P \} \\ \langle y \rangle = @\langle x \rangle \\ \{ P \wedge \langle y \rangle = @\langle x \rangle \} \end{aligned}$$

donde  $\langle y \rangle$  es un identificador,  $\langle x \rangle$  es un identificador ligado a una celda, y  $P$  es una afirmación. La especificación de  $:=$  es:

$$\begin{aligned} \{ P(\langle \text{exp} \rangle) \} \\ \langle x \rangle := \langle \text{exp} \rangle \\ \{ P(@\langle x \rangle) \} \end{aligned}$$

donde  $\langle x \rangle$  es un identificador ligado a una celda,  $P(@\langle x \rangle)$  es una afirmación que contiene  $@\langle x \rangle$ , y  $\langle \text{exp} \rangle$  es una expresión permitida en una afirmación. Estas especificaciones también se denominan reglas de prueba, pues se usan como componentes básicos en una prueba de corrección. Cuando aplicamos cada regla, somos libres de escoger que  $\langle x \rangle$ ,  $\langle y \rangle$ ,  $P$ , y  $\langle \text{exp} \rangle$  sean lo que necesitamos.

Aplicaremos las reglas de prueba a la definición de `Colocar`. Empecemos con la declaración de asignación y trabajemos hacia atrás: dada la poscondición, determinar la precondición. (Con la asignación, frecuentemente es más fácil razonar en dirección hacia atrás.) En nuestro caso, la poscondición es  $@C = x | s$ . Emparejando esto con  $P(@\langle x \rangle)$ , vemos que  $\langle x \rangle$  es la celda  $C$  y  $P(@C) \equiv @C = x | s$ . Utilizando la regla para  $:=$ , reemplazamos  $@C$  por  $x | S$ , resultando como precondición  $x | S = x | s$ .

Ahora razonemos hacia adelante desde el acceso a la celda. La precondición es  $@C = s$ . Aplicando la regla de prueba, vemos que la poscondición es  $(@C = s \wedge S = @C)$ . Colocando las dos partes juntas, tenemos:

$$\begin{aligned} \{ @C = s \} \\ S = @C \\ \{ @C = s \wedge S = @C \} \\ \{ x | S = x | s \} \\ C := x | S \\ \{ @C = x | s \} \end{aligned}$$

Esta es una prueba válida por dos razones. Primero, se respetan estrictamente las reglas de prueba para  $@$  y  $:=$ . Segundo,  $(@C = s \wedge S = @C)$  implica  $(x | S = x | s)$ .

### 6.6.3. Afirmaciones

Una afirmación  $\langle af \rangle$  es una expresión booleana asociada a un lugar particular dentro de un programa, el cual denominaremos un punto del programa. La expresión booleana es muy similar a las expresiones booleanas del modelo de computación. Existen algunas diferencias, debido a que las afirmaciones son expresiones matemáticas utilizadas para razonar, y no fragmentos de programas. Una afirmación puede contener identificadores  $\langle x \rangle$ , valores parciales  $x$ , y contenidos de celdas  $@\langle x \rangle$  (con el operador  $@$ ). Por ejemplo, usamos la afirmación  $@C = x | s$  cuando razonamos sobre la abstracción de datos pila. Una afirmación también puede contener cuan-

*Estado explícito*

tificadores y sus variables lógicas (no son variables del lenguaje de programación), y también puede contener funciones matemáticas, las cuales pueden corresponder directamente a funciones escritas en el modelo declarativo.

Para evaluar una afirmación, ésta debe estar asociada a un punto del programa. Los puntos del programa se caracterizan por el ambiente que existe donde están; evaluar una afirmación en un punto del programa significa evaluarla utilizando ese ambiente. Suponemos que todas las variables de flujo de datos están suficientemente ligadas para que la evaluación dé **true** o **false**.

Utilizamos las notaciones  $\wedge$  para la conjunción lógica ( $y$ ),  $\vee$  para la disyunción lógica ( $o$ ),  $\neg$  para la negación lógica ( $no$ ). Usamos los cuantificadores *para todo* ( $\forall$ ) y *existe* ( $\exists$ ):

$\forall x.\langle\text{tipo}\rangle: \langle\text{af}\rangle \quad \langle\text{af}\rangle$  es cierta cuando  $x$  toma cualquier valor del tipo  $\langle\text{tipo}\rangle$   
 $\exists x.\langle\text{tipo}\rangle: \langle\text{af}\rangle \quad \langle\text{af}\rangle$  es cierta para por lo menos un valor  $x$  del tipo  $\langle\text{tipo}\rangle$

En cada una de estas expresiones cuantificadas,  $\langle\text{tipo}\rangle$  es un tipo legal del modelo declarativo, tal como se definió en la sección 2.3.2.

Las técnicas de razonamiento que introducimos aquí se pueden usar en todos los lenguajes con estado. En muchos de esos lenguajes, e.g., C++ y Java, queda claro desde la declaración si un identificador referencia una variable mutable (una celda o atributo) o un valor (i.e., una constante). Como no hay ambigüedad, el símbolo @ puede no usarse cuando se usan estas técnicas en ellos. En nuestro modelo, conservamos el símbolo @ porque podemos distinguir entre el nombre de una celda ( $c$ ) y su contenido ( $@c$ ).

#### 6.6.4. Reglas de prueba

Por cada declaración  $S$  en el lenguaje núcleo, tenemos una regla de prueba que muestra todas las formas correctas posibles de  $\{ A \} S \{ B \}$ . La regla de prueba es sólo una especificación de  $S$ . Podemos probar la corrección de la regla utilizando la semántica operacional del lenguaje núcleo. Miremos cómo son las reglas para el lenguaje núcleo con estado.

##### *Ligadura*

Ya hemos mostrado una regla para la ligadura, en el caso  $\langle y \rangle = @\langle x \rangle$ , donde el lado derecho corresponde al contenido de una celda. La forma general de una ligadura es  $\langle x \rangle = \langle \text{exp} \rangle$ , donde  $\langle \text{exp} \rangle$  es una expresión declarativa cuya evaluación da un valor parcial. La expresión puede contener accesos a celdas (invocaciones a @). Esto nos lleva a la regla de prueba siguiente:

$$\{ P \} \langle x \rangle = \langle \text{exp} \rangle \quad \{ P \wedge \langle x \rangle = \langle \text{exp} \rangle \}$$

donde  $P$  es una afirmación.

### Asignación

La regla de prueba siguiente se cumple para la asignación:

$$\{ P(\langle \text{exp} \rangle) \} \langle x \rangle := \langle \text{exp} \rangle \{ P(@\langle x \rangle) \}$$

donde  $\langle x \rangle$  referencia una celda,  $P(@\langle x \rangle)$  es una afirmación que contiene  $@\langle x \rangle$ , y  $\langle \text{exp} \rangle$  es una expresión declarativa.

### Condicional (declaración **if**)

La declaración **if** tiene la forma:

**if**  $\langle x \rangle$  **then**  $\langle \text{dec} \rangle_1$  **else**  $\langle \text{dec} \rangle_2$  **end**

El comportamiento depende de si  $\langle x \rangle$  se liga a **true** o a **false**. Si sabemos que:

$$\{ P \wedge \langle x \rangle = \text{true} \} \langle \text{dec} \rangle_1 \{ Q \}$$

y también que:

$$\{ P \wedge \langle x \rangle = \text{false} \} \langle \text{dec} \rangle_2 \{ Q \}$$

entonces podemos concluir que:

$$\{ P \} \text{ if } \langle x \rangle \text{ then } \langle \text{dec} \rangle_1 \text{ else } \langle \text{dec} \rangle_2 \text{ end } \{ Q \}.$$

Aquí,  $P$  y  $Q$  son afirmaciones y  $\langle \text{dec} \rangle_1$  y  $\langle \text{dec} \rangle_2$  son declaraciones en el lenguaje núcleo. Resumimos esta regla con la notación siguiente:

$$\begin{array}{c} \{ P \wedge \langle x \rangle = \text{true} \} \langle \text{dec} \rangle_1 \{ Q \} \\ \{ P \wedge \langle x \rangle = \text{false} \} \langle \text{dec} \rangle_2 \{ Q \} \\ \hline \{ P \} \text{ if } \langle x \rangle \text{ then } \langle \text{dec} \rangle_1 \text{ else } \langle \text{dec} \rangle_2 \text{ end } \{ Q \} \end{array}$$

En esta notación, las premisas están arriba de la línea horizontal y la conclusión está debajo de ella. Para utilizar la regla, primero tenemos que probar las premisas.

### Procedimiento sin referencias externas

Suponga que el procedimiento tiene la forma siguiente:

```
proc {⟨p⟩ ⟨x⟩₁ ... ⟨x⟩ₙ}
 ⟨dec⟩
 end
```

donde las únicas referencias externas de  $\langle \text{dec} \rangle$  son  $\{\langle x \rangle_1, \dots, \langle x \rangle_n\}$ . Entonces la regla siguiente se cumple:

$$\begin{array}{c} \{ P(\overline{\langle x \rangle}) \} \langle \text{dec} \rangle \{ Q(\overline{\langle x \rangle}) \} \\ \hline \{ P(\overline{\langle y \rangle}) \} \{ \langle p \rangle \langle y \rangle_1 \dots \langle y \rangle_n \} \{ Q(\overline{\langle y \rangle}) \} \end{array}$$

*Estado explícito*

donde  $P$  y  $Q$  son afirmaciones y la notación  $\overline{\langle x \rangle}$  significa  $\langle x \rangle_1, \dots, \langle x \rangle_n$ .

*Procedimiento con referencias externas*

Suponga que el procedimiento tiene la forma siguiente:

```
proc {⟨p⟩ ⟨x⟩1 … ⟨x⟩n}
 ⟨dec⟩
end
```

donde las referencias externas de  $\langle dec \rangle$  son  $\{\langle x \rangle_1, \dots, \langle x \rangle_n, \langle z \rangle_1, \dots, \langle z \rangle_k\}$ . Entonces la regla siguiente se cumple:

$$\frac{\{ P(\overline{\langle x \rangle}, \overline{\langle z \rangle}) \} \langle dec \rangle \{ Q(\overline{\langle x \rangle}, \overline{\langle z \rangle}) \}}{\{ P(\overline{\langle y \rangle}, \overline{\langle z \rangle}) \} \{ \langle p \rangle \langle y \rangle_1 \dots \langle y \rangle_n \} \{ Q(\overline{\langle y \rangle}, \overline{\langle z \rangle}) \}}$$

donde  $P$  y  $Q$  son afirmaciones.

*Ciclos while*

Las reglas anteriores son suficientes para razonar sobre programas que realizan ciclos usando la recursión. En el modelo con estado, es conveniente añadir otra operación básica para los ciclos: el ciclo **while**. Como el ciclo **while** se puede definir en términos del lenguaje núcleo, no se agrega ninguna expresividad nueva al lenguaje. Por lo tanto, definimos el ciclo **while** como una abstracción lingüística. Introducimos la sintaxis nueva:

```
while ⟨expr⟩ do ⟨dec⟩ end
```

Definimos la semántica del ciclo **while** traduciéndolo en operaciones más sencillas:

```
{While fun {$} ⟨expr⟩ end proc {$} ⟨dec⟩ end}

proc {While Expr Dec}
 if {Expr} then {Dec} {While Expr Dec} end
end
```

Añadimos una regla de prueba específica para el ciclo **while**:

$$\frac{\{ P \wedge \langle expr \rangle \} \langle dec \rangle \{ P \}}{\{ P \} \text{while} \langle expr \rangle \text{do} \langle dec \rangle \text{end} \{ P \wedge \neg \langle expr \rangle \}}$$

Podemos probar que la regla es correcta utilizando la definición del ciclo **while** y el método de afirmaciones invariantes. Normalmente, es más fácil utilizar esta regla que razonar directamente con procedimientos recursivos.

### Ciclos for

En la sección 3.6.3 vimos otro constructor de ciclo, el ciclo **for**. En su forma más simple, este ciclo itera sobre enteros:

```
for <x> in <y>..<z> do <dec> end
```

Esta es, también, una abstracción lingüística, la cual se define como sigue:

```
{For <y> <z> proc {$ <x>} <dec> end}
```

```
proc {For I H Dec}
 if I=<H then {Dec I} {For I+1 H Dec} else skip end
end
```

Añadimos una regla de prueba específica para el ciclo **for**:

$$\forall i. \langle y \rangle \leq i \leq \langle z \rangle : \{ P_{i-1} \wedge \langle x \rangle = i \} \langle dec \rangle \{ P_i \}$$

$$\{ P_{\langle y \rangle - 1} \} \textbf{for } \langle x \rangle \textbf{ in } \langle y \rangle .. \langle z \rangle \textbf{ do } \langle dec \rangle \textbf{ end } \{ P_{\langle z \rangle} \}$$

¡Tenga cuidado con el índice inicial de  $P$ ! Como un ciclo **for** comienza con  $\langle y \rangle$ , el índice inicial de  $P$  tiene que ser  $\langle y \rangle - 1$ , el cual expresa que el ciclo no ha comenzado todavía. Al igual que para el ciclo **while**, podemos probar que esta regla es correcta utilizando la definición del ciclo **for**. Miremos cómo funciona esta regla con un ejemplo sencillo. Considere el fragmento de programa siguiente, para sumar los elementos de un arreglo:

```
local C={NewCell 0} in
 for I in 1..10 do C:=@C+A.I end
end
```

donde  $A$  es un arreglo con índices del 1 al 10. Escojamos un invariante:

$$P_i \equiv @C = \sum_{j=1}^i A_j$$

donde  $A_j$  es el  $j$ -ésimo elemento de  $A$ . Este invariante simplemente define un resultado intermedio en el cálculo que hace el ciclo. Con este invariante podemos probar la premisa para la regla de prueba del ciclo **for**:

$$\begin{aligned} &\{ @C = \sum_{j=1}^{i-1} A_j \wedge I = i \} \\ &C := @C + A.I \\ &\{ @C = \sum_{j=1}^i A_j \} \end{aligned}$$

Esto se deduce inmediatamente a partir de la regla de prueba de la asignación. Como  $P_0$  es claramente cierto antes del ciclo, se concluye, a partir de la regla de prueba del ciclo **for**, que  $P_{10}$  también es cierto. Por lo tanto,  $C$  contiene la suma de todos los elementos del arreglo.

### Razonando en niveles más altos de abstracción

Los ciclos **while** y **for** son ejemplos de razonamiento en niveles más altos de abstracción que el lenguaje núcleo. Para cada ciclo, definimos la sintaxis y su traducción al lenguaje núcleo, y luego definimos una regla de prueba. La idea es verificar la regla de prueba una sola vez y, luego, usarla tan frecuentemente como queramos. Este enfoque, la definición de conceptos nuevos y sus reglas de prueba, es la forma de realizar un razonamiento en la práctica sobre programas con estado. Quedarse siempre en el lenguaje núcleo es bastante más engorroso para todo, salvo para programas de juguete.

### Referencias Compartidas

Las reglas de prueba presentadas atrás son correctas si no existen referencias compartidas. Sin embargo, las reglas deben ser modificadas de la manera obvia, si hay referencias compartidas. Por ejemplo, suponga que  $C$  y  $D$  referencian la misma celda y considere la asignación  $C := @C + 1$ . Suponga que la poscondición es  $@C = 5 \wedge @D = 5 \wedge C = D$ . La regla de prueba estándar nos permite calcular la precondición reemplazando  $@C$  por  $@C + 1$ . Esto nos conduce a un resultado incorrecto porque no se ha tenido en cuenta que  $D$  referencia la misma celda que  $C$ . La regla de prueba se puede corregir reemplazando a  $@D$  también.

#### 6.6.5. Terminación normal

Razonar sobre la corrección parcial no concluye nada sobre si un programa termina normalmente o no. Sólo se puede concluir, que si el programa termina normalmente, entonces tal cosa y tal otra son ciertas. Así el razonamiento es sencillo, pero es sólo una parte de la historia. Tenemos que probar terminación también. Hay tres formas en que un programa puede no terminar normalmente:

- La ejecución cae en un ciclo infinito. Este es un error de programación debido a que la computación no avanza hacia el objetivo.
- La ejecución se bloquea debido a que una variable de flujo de datos no está suficientemente ligada dentro de una operación. Este es un error de programación debido a que se pasó por alto un cálculo o se entró en una situación de muerte súbita<sup>10</sup>.
- La ejecución lanza una excepción debido a un error. Esta excepción es una salida anormal. En esta sección, consideraremos sólo una clase de error, el error de tipo, pero se puede razonar de manera parecida para otras clases de errores.

Miremos cómo manejar cada caso.

---

10. Nota del traductor: *deadlock* en inglés.

### Razonamiento de avance

Cada vez que hay un ciclo, hay un peligro de que no termine. Para mostar que un ciclo termina, es suficiente mostrar que existe una función no negativa que siempre disminuye su valor entre iteraciones sucesivas.

### Razonamiento de suspensión

Es suficiente mostar que todas las variables están suficientemente ligadas antes de ser usadas. Por cada utilización de la variable, se debe determinar si en el origen de todos los posibles caminos de ejecución encontramos una ligadura.

### Comprobación de tipos

Para mostrar que no hay errores de tipo, es suficiente mostrar que todas las variables tienen el tipo correcto. Esto se suele denominar comprobación de tipos. Otras clases de errores necesitan de otro tipo de comprobaciones.

---

## 6.7. Diseño de programas en grande

El buen *software* es bueno en grande y en pequeño, en su arquitectura de alto nivel y en sus detalles de bajo nivel.

– Adaptación libre de *Object-Oriented Software Construction*, 2nd ed., Bertrand Meyer (1997)

Una administración eficiente y exitosa se manifiesta de la misma manera tanto en los pequeños como en los grandes asuntos.

– Memorandum, August 8, 1943, Winston Churchill (1874–1965)

La programación en grande es la programación a cargo de un equipo de gente. La programación en grande incluye todos los aspectos del desarrollo de *software* que requieren comunicación y coordinación entre la gente. Administrar un equipo de manera que funcionen juntos, eficientemente, es difícil en cualquier área—como testimonio está la dificultad de dirigir un equipo de fútbol. En la programación, esto es especialmente difícil pues generalmente los programas están incomprendiblemente llenos de errores pequeños. Los programas exigen una exactitud difícil de satisfacer por parte de los seres humanos. La programación en grande se denomina con frecuencia ingeniería de *software*.

Esta sección se basa en la introducción a la programación en pequeño (ver sección 3.9). Explicamos cómo desarrollar *software* en equipos pequeños. No explicamos qué hacer para equipos más grandes (con cientos o miles de miembros)—lo cual es un tema de otro libro.

### 6.7.1. Metodología de diseño

Muchas cosas, tanto ciertas como falsas, se han publicado sobre la metodología de diseño correcta para programar en grande. La mayor parte de la literatura existente consiste de extrapolación basada en experimentos limitados, pues la validación rigurosa es muy difícil. Para validar una idea nueva, varios equipos, por lo demás idénticos, tendrían que trabajar en circunstancias idénticas. Esto se ha hecho muy raramente y no intentaremos eso tampoco.

En esta sección, se resumen las lecciones que hemos aprendido a partir de nuestra propia experiencia en construcción de sistemas. Hemos diseñado y construido varios sistemas de *software* en grande [79, 120, 177]. Hemos pensado bastante sobre cómo hacer un buen diseño de programa en equipo y hemos mirado cómo trabajan otros equipos de desarrollo exitosos. Hemos tratado de aislar los principios verdaderamente útiles de los otros.

#### *Administración del equipo*

La primera y más importante tarea es asegurarse que el equipo trabaja junto, en forma coordinada. Hay tres ideas para lograrlo:

1. Compartimentar la responsabilidad de cada persona. Por ejemplo, cada componente se puede asignar a una persona, quien es responsable de él. Para nosotros, las responsabilidades deben respetar los límites entre componentes y no se deben solapar. Esto evita discusiones interminables sobre quién debería haber resuelto un problema.
2. En contraste con la responsabilidad, el conocimiento debe ser intercambiado libremente, y no compartimentado. Los miembros de un equipo deben intercambiar con frecuencia información sobre las diferentes partes del sistema. Idealmente, no deben existir miembros indispensables en el equipo. Todos los cambios importantes en el sistema se deben discutir entre los miembros eruditos del equipo antes de implementarse. Esto mejora de manera importante la calidad del sistema. También es importante que el propietario de un componente tenga la última palabra sobre cómo se debe cambiar el componente. Los miembros menos expertos del equipo deben ser asignados como aprendices de los miembros más expertos, para así conocer el sistema. Los miembros menos expertos adquieren experiencia asignándoseles tareas específicas para realizar, las cuales realizan de la manera más independiente posible. Esto es importante para la longevidad del sistema.
3. Documentar cuidadosamente la interfaz de cada componente, pues ésta es también la interfaz entre el propietario del componente y los otros miembros del equipo. La documentación es especialmente importante, mucho más que para la programación en pequeño. La buena documentación, como pilar que es de la estabilidad, puede ser consultada por todos los miembros del equipo.

### Metodología de desarrollo de software

Hay muchas maneras de organizar el desarrollo de un *software*. Las técnicas de desarrollo de arriba hacia abajo, abajo hacia arriba, y aún “del medio hacia afuera” han sido bastante discutidas. Ninguna de esas técnicas es realmente satisfactoria para programas en grande, y combinarlas no ayuda mucho. El problema principal es que esas técnicas se basan en que los requerimientos y la especificación del sistema están bastante completas para empezar con ellas, i.e., que los diseñadores anticiparon la mayoría de los requerimientos funcionales y no funcionales. A menos que el sistema esté muy bien comprendido, esto es casi imposible de lograr. En nuestra experiencia, un enfoque que funciona bien es el enfoque de desarrollo incremental, también denominado, enfoque de desarrollo iterativo o DII (desarrollo iterativo e incremental). Algunas veces también se denomina desarrollo evolucionario, aunque estrictamente hablando esta metáfora no es aplicable pues no existe evolución en el sentido Darwiniano [40].<sup>11</sup> El desarrollo incremental tiene una larga historia tanto al interior como al exterior de la computación. Éste ha sido utilizado exitosamente en el desarrollo de software desde los años 1950 [16, 100]. Sus etapas son las siguientes:

- Empezar con un conjunto pequeño de requerimientos, el cual es un subconjunto del conjunto completo, y construir un sistema completo que los satisfaga. La especificación y la arquitectura del sistema son “caparazones vacíos”: ellas están suficientemente completas para que se pueda construir un programa ejecutable pero no resuelven los problemas del usuario.
- Luego, extender continuamente los requerimientos de acuerdo a la retroalimentación del usuario, extendiendo la especificación y la arquitectura del sistema como se requiera para satisfacer los requerimientos nuevos. Usando una metáfora orgánica, decimos que la aplicación está “creciendo.” En todo momento, existe un sistema ejecutable que satisface su especificación y que puede ser evaluado por usuarios potenciales.
- No optimizar durante el proceso de desarrollo. Es decir, no haga el diseño más complejo sólo por mejorar el desempeño. Utilice algoritmos sencillos con complejidad aceptable y conserve un diseño simple con las abstracciones correctas. No se preocupe por el costo de esas abstracciones. La optimización del desempeño se puede realizar cerca del final del desarrollo, pero sólo si existen problemas de desempeño. En ese momento, se debe perfilar la aplicación para encontrar aquellas partes del sistema (normalmente muy pequeñas) que deben de reescribirse.
- Reorganizar el diseño tanto como sea necesario durante el desarrollo, para conser-

---

11. La evolución darwiniana implica una población donde hay individuos que mueren y unos nuevos que nacen, una fuente de diversidad en los individuos que nacen, y un filtro (selección natural) que elimina los individuos menos aptos. No existe población en el enfoque incremental.

*Estado explícito*

var una buena organización de los componentes. Los componentes deben encapsular las decisiones de diseño o implementar abstracciones comunes. A esta reorganización algunas veces se le llama “refactorización.” Existe un espectro entre los extremos de planificar completamente un diseño y confiar completamente en la refactorización. El mejor enfoque está en alguna parte en el medio.

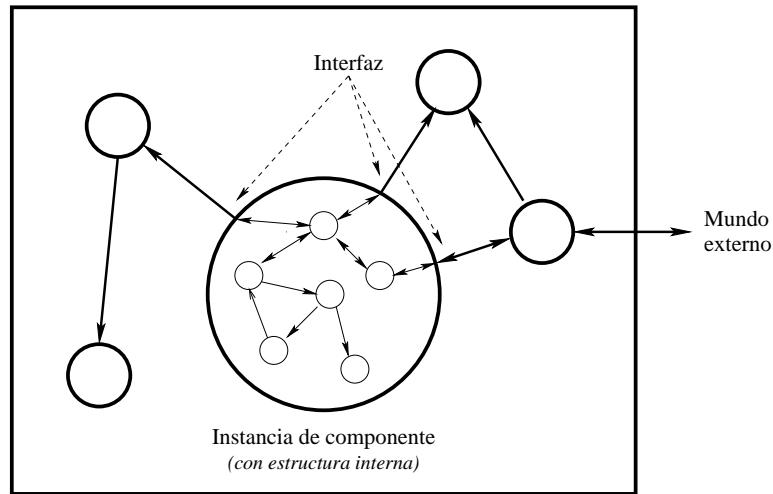
El desarrollo incremental tiene muchas ventajas:

- Los errores de todos los tamaños se detectan tempranamente y se pueden resolver rápidamente.
- La presión de las fechas límites se alivia bastante, pues siempre existe una aplicación que funciona.
- Los desarrolladores están más motivados, pues tienen una retroalimentación rápida a sus esfuerzos.
- Es más probable que los usuarios consigan lo que necesitan, pues tienen la posibilidad de utilizar la aplicación durante el proceso de desarrollo.
- Es más probable que la arquitectura sea buena, pues puede ser corregida desde temprano.
- Es más probable que la interfaz del usuario sea buena, pues se mejora continuamente durante el proceso de desarrollo.

En nuestra opinión, para la mayoría de los sistemas, aún para los más pequeños, es casi imposible determinar de antemano cuáles son sus requerimientos reales, cuál es una buena arquitectura para implementarlos, y cuál debe ser la interfaz del usuario. El desarrollo incremental funciona parcialmente bien porque se hacen muy pocas suposiciones previas. Para una visión complementaria, recomendamos mirar la programación extrema, el cual es otro enfoque que hace énfasis en el compromiso entre la planificación y la refactorización [167]. Para una visión más radical, recomendamos mirar el desarrollo orientado por las pruebas, el cual es incremental, pero de una manera completamente diferente. El desarrollo orientado por las pruebas afirma que es posible escribir un programa sin ninguna fase de diseño, simplemente creando pruebas y haciendo refactorización cada vez que el programa falla en una prueba nueva [17]. ¡Podemos ver que el diseño de pruebas buenas es crucial para el éxito de este enfoque!

### 6.7.2. Estructura jerárquica del sistema

¿Cómo debería estructurarse un sistema para soportar el trabajo en equipo y el desarrollo incremental? Una forma, que funciona en la práctica, consiste en estructurar la aplicación como un grafo jerárquico con interfaces bien definidas en cada nivel (ver figura 6.6). Es decir, la aplicación consiste de un conjunto de nodos, donde cada nodo interactúa con otros nodos. Cada nodo es una instancia de un componente. Cada nodo es en sí mismo una pequeña aplicación, y puede descomponerse a sí mismo en un grafo. La descomposición termina cuando alcanzamos los



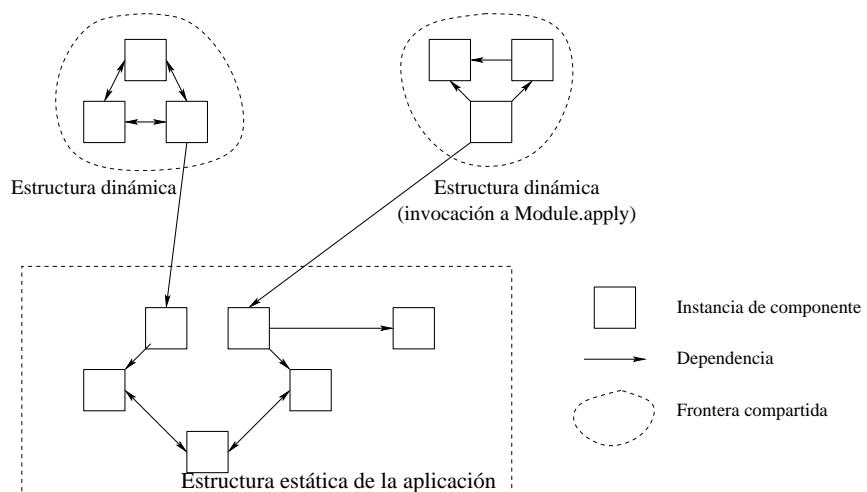
**Figura 6.6:** Un sistema estructurado como un grafo jerárquico.

componentes primitivos proveidos por el sistema subyacente.

## *Conexión de componentes*

La primera tarea, cuando se está construyendo el sistema, consiste en conectar los componentes. Esto tiene tanto un aspecto estático, como uno dinámico:

- *Estructura estática.* Ésta consiste del grafo de componentes conocido en el momento en que se diseña la aplicación. Estos componentes se pueden conectar tan pronto como la aplicación comienza su ejecución. Cada instancia de un componente corresponde, aproximadamente, a un paquete de funcionalidad conocido algunas veces como una biblioteca o un paquete. Por eficiencia, nos gustaría que cada instancia de un componente existiera a lo sumo una vez en el sistema. Si se necesita una biblioteca específica en diferentes partes de la aplicación, entonces deseamos que las instancias compartan esa misma biblioteca. Por ejemplo, un componente puede implementar un paquete gráfico; toda la aplicación puede arreglárselas con una sola instancia.
  - *Estructura dinámica.* Con frecuencia, una aplicación hace cálculos con componentes en tiempo de ejecución. La aplicación puede necesitar conectar componentes nuevos, los cuales se conocen sólo en tiempo de ejecución. O la aplicación también puede calcular un componente nuevo y almacenarlo. Las instancias de los componentes no tienen que ser compartidas; tal vez se necesiten varias instancias de un componente. Por ejemplo, un componente puede implementar una interfaz a una base de datos. Dependiendo de si existe una o más bases de datos externas, el componente debería cargar una o más instancias del componente. Esto se determina en

*Estado explícito***Figura 6.7:** Estructura estática y dinámica de un sistema.

tiempo de ejecución, cada que se añade una base de datos.

En la figura 6.7 se muestra la estructura resultante de la aplicación, con algunos componentes conectados estáticamente y otros conectados dinámicamente.

**Estructura estática** Para soportar la estructura estática, es útil crear los componentes como unidades de compilación almacenadas en archivos. Llamamos a estos componentes functors y a sus instancias módulos. Un functor es un componente que puede ser almacenado en un archivo; es una unidad de compilación porque el archivo puede ser compilado independientemente de otros functors. Las dependencias entre functors se especifican como nombres de archivos. Para poder ser accedido por otros functors, el functor debe ser almacenado en un archivo. Esto permite a otros functors especificar los que ellos necesitan por medio de nombres de archivo.

Un functor tiene dos representaciones: una en forma de fuente, la cual es simplemente un archivo de texto, y una en forma compilada, la cual es un valor en el lenguaje. Si la forma de fuente está en el archivo `foo.oz`, cuyo contenido completo es de la forma `functor ... end`, entonces se puede compilar para producir otro archivo, `foo.ozf`, que contiene la forma compilada. El contenido del archivo `foo.oz` se ve así:

```
functor
import OtroComp1 at Arch1
 OtroComp2 at Arch2
 ...
 OtroCompN at ArchN
export op1:X1 op2:X2 ... opK:Xk
define
 % Definir X1, ..., Xk
 ...
end
```

Este componente depende de los otros componentes `OtroComp1`, ..., `OtroCompN`, almacenados, respectivamente, en los archivos `Arch1`, ..., `ArchN`. Este componente define un módulo con campos `op1`, ..., `opK`, referenciados por medio de `x1`, ..., `xk` y definidos en el cuerpo del functor.

Una aplicación es simplemente un functor compilado. Para ejecutar la aplicación, todos los otros functors compilados que ella necesita, directa o indirectamente, deben ser colocados juntos e instanciados. Cuando la aplicación se ejecuta, ella carga sus componentes y los enlaza (conecta). Enlazar un componente significa evaluarlo con sus módulos importados como argumentos y entregar el resultado a los módulos que lo necesitan. El enlace puede ocurrir en el momento en que la aplicación se lanza (enlace estático), o por necesidad, i.e., uno por uno a medida que se van necesitando (enlace dinámico). Hemos encontrado que el enlace dinámico es normalmente preferible, siempre que todos los functors estén rápidamente accesibles (e.g., existan en el sistema local). Con este comportamiento por defecto, la aplicación comienza rápidamente y ocupa solamente el espacio en memoria que necesita.

**Estructura dinámica** Un functor no es más que otra entidad del lenguaje. Si un programa contiene una declaración de la forma `x=functor ... end`, entonces `x` se ligará a un valor de tipo functor. Al igual que un procedimiento, un functor puede tener referencias externas. La instanciación de un functor produce un módulo, el cual es el conjunto de entidades del lenguaje creadas por la inicialización del functor. Una interfaz es un registro que contiene las entidades del módulo visibles externamente. En un conveniente abuso de notación, a este registro se le llama algunas veces el módulo. Existe una sola operación para instanciar functors:

- `Ms={Module.apply Fs}`. Dada una lista de functors `Fs` (como entidades del lenguaje), esta operación instancia todos los functors y crea una lista de módulos `Ms`. Dentro del alcance de una invocación a `Module.apply`, los functors se comparten. Es decir, si el mismo functor se menciona más de una vez, entonces se enlaza una sola vez.

Cada invocación a `Module.apply` produce un nuevo conjunto fresco de módulos. Esta operación hace parte del módulo `Module`. Si el functor está almacenado en un archivo, éste debe ser cargado antes de invocar a `Module.apply`.

### ***Comunicación de componentes***

Una vez los componentes están conectados, tienen que comunicarse entre ellos. Aquí presentamos seis de los más populares protocolos para la comunicación entre componentes. Los presentamos en orden creciente en términos de la independencia de los componentes:

1. *Procedimiento.* La primera organización es donde la aplicación es secuencial y donde un componente invoca al otro como un procedimiento. El invocador no es necesariamente el único iniciador; pueden existir invocaciones anidadas donde el centro de control pasa hacia adelante y hacia atrás entre los componentes. Pero, existe un único centro de control global, el cual liga fuertemente los dos componentes.
2. *Corutina.* Una segunda organización es donde cada uno de los dos componentes evoluciona independientemente, pero aún en un modo secuencial. Esto introduce el concepto de corutina (ver sección 4.4.2). Un componente invoca a otro, el cual continúa donde había quedado. Existen varios centros de control, uno por cada corutina. Esta organización es más libre que la anterior, pero los componentes siguen siendo dependientes pues se ejecutan alternándose.
3. *Concurrente sincrónico.* Una tercera organización es una en la cual cada componente evoluciona independientemente de los otros y puede iniciar y terminar comunicaciones con otros componentes, de acuerdo a algún protocolo que ambos componentes hayan acordado. Los componentes se ejecutan concurrentemente. Hay varios centros de control, denominados hilos, los cuales evolucionan independiente mente (ver capítulos 4 y 8). Cada componente invoca a los otros sincrónicamente, i.e., cada componente espera una respuesta para poder continuar.
4. *Concurrente asincrónico.* Una cuarta organización consiste en un conjunto de componentes concurrentes que se comunican a través de canales asincrónicos. Cada componente envía mensajes a los otros, pero no tienen que esperar una respuesta para continuar. Los canales pueden tener un orden FIFO (los mensajes se reciben en el orden en que son enviados) o no tener ningún orden. Los canales se denominan flujos en el capítulo 4 y puertos en el capítulo 5. En esta organización, cada componente conoce la identidad del componente con el cual se comunica.
5. *Buzón concurrente.* Una quinta organización es una variación de la cuarta. Los canales asincrónicos se comportan como buzones. Es decir, es posible realizar reconocimiento de patrones para extraer los mensajes de los canales, sin perturbar los mensajes que permanecen sin ser respondidos. Esto es muy útil para muchos programas concurrentes. Ésta es una operación básica en el lenguaje Erlang, el cual cuenta con buzones FIFO (ver capítulo 5). También es posible tener buzones sin orden.
6. *Modelo de coordinación.* Una sexta organización es donde los componentes se pueden comunicar sin que los emisores y receptores conozcan sus identidades. Una abstracción denominada tupla espacio vive en la interfaz. Los componentes son

concurrentes e interactúan únicamente a través de la tupla espacio común. Un componente puede insertar asincrónicamente un mensaje y otro puede recuperar el mensaje. En la sección 8.3.2 se define una forma de abstracción de la tupla espacio y se muestra cómo implementarla.

### *El principio de la independencia del modelo*

Cada componente del sistema se escribe en un modelo de computación en particular. En la sección 4.8.6 se resumieron los modelos de computación más populares utilizados para programar componentes. Durante el desarrollo, una estructura interna del componente puede cambiar drásticamente. No es raro que se cambie su modelo de computación. Un componente sin estado puede convertirse en uno con estado (o concurrente, o distribuido, etc.), o viceversa. Si un cambio como ese sucede dentro de un componente, entonces no se necesita cambiar su interfaz. La interfaz sólo debe cambiarse si la funcionalidad externa, visible, del componente, cambia. Esta es una importante propiedad de modularidad de los modelos de computación. En tanto que la interfaz sea la misma, esta propiedad garantiza que no es necesario cambiar nada más en el resto del sistema. Consideraremos esta propiedad como un principio básico de diseño de los modelos de computación:

#### Principio de la independencia del modelo

La interfaz de un componente debe ser independiente del modelo de computación utilizado para implementar el componente. La interfaz solamente debe depender de la funcionalidad externamente visible del componente.

Un buen ejemplo de este principio es la memorización. Suponga que el componente es una función que calcula su resultado basado en un argumento. Si el cálculo consume bastante tiempo, entonces una forma de reducir de manera importante el tiempo de ejecución consiste en conservar, en memoria oculta, las parejas argumento-resultado. Cuando se invoque la función, se verifica primero si el argumento está en la memoria oculta. De ser así, se devuelve el resultado directamente sin hacer el cálculo. Si no es así, se realiza el cálculo y se agrega la nueva pareja argumento-resultado a la memoria oculta. En la sección 10.4.2 (en CTM) se muestra un ejemplo de memorización. Como la memorización oculta necesita estado, cambiar un componente para que haga memorización significa que el componente puede cambiar de utilizar el modelo declarativo a utilizar el modelo con estado. El principio de la independencia del modelo implica que esto se puede hacer sin cambiar nada más en el programa.

### *Compilación eficiente versus ejecución eficiente*

Un componente es una unidad de compilación. Nos gustaría compilar un componente tan rápida y eficientemente como sea posible. Esto significa que nos gustaría

*Estado explícito*

compilar de manera separada un componente, i.e., sin saber nada sobre los otros componentes. También nos gustaría que el programa final, en el cual están todos los componentes ensamblados, sea tan eficiente y compacto como sea posible. Esto quiere decir que nos gustaría hacer análisis en tiempo de compilación, e.g., comprobación de tipos, inferencia de tipos al estilo Haskell, u optimización global.

Hay una gran tensión entre estos dos deseos. Si la compilación es verdaderamente separada, entonces el análisis no puede cruzar las fronteras del componente. Para hacer un análisis verdaderamente global, el compilador debe, en general, ser capaz de mirar todo el programa de una sola vez. Esto significa que para muchos lenguajes estáticamente tipados, la compilación de programas grandes (digamos, de más de un millón de líneas de código fuente) requiere mucho tiempo y memoria.

Existen muchas técnicas ingeniosas que tratan de conseguir lo mejor de ambos mundos. Idealmente, estas técnicas no deben volver el compilador demasiado complicado. Esto es un problema difícil. Después de cinco décadas de experiencia en diseño de lenguajes y de compiladores, éste es aún un tema de investigación activo.

Los sistemas comerciales de calidad abarcan todo el espectro, desde compilación completamente separada hasta análisis global sofisticado. En la práctica, el desarrollo de aplicaciones se puede hacer en cualquier punto del espectro. El sistema Mozart está en uno de los extremos del espectro. Como Mozart es dinámicamente tipado, los componentes se pueden compilar sin saber nada sobre los otros. Esto significa que la compilación es completamente escalable: compilar un componente toma el mismo tiempo, independientemente de si se utiliza en un programa de un millón de líneas o en un programa de mil líneas. Por el otro lado, existen desventajas: un menor grado de optimización y los errores de tipo sólo se detectan en tiempo de ejecución. Si estos aspectos son críticos o no, depende de la aplicación y de la experiencia del desarrollador.

### 6.7.3. Mantenimiento

Una vez el sistema está construido y funcionando bien, tenemos que asegurarnos que se conserva funcionando bien. El proceso de conservar un sistema funcionando bien después de que es entregado se denomina mantenimiento. ¿Cuál es la mejor forma de estructurar los programas de manera que se puedan mantener con facilidad? A partir de nuestra experiencia, presentamos algunos de los principios más importantes. Miramos esto desde el punto de vista de los componentes como tal y desde el punto de vista del sistema.

#### *Diseño de componentes*

Existen buenas y malas maneras de diseñar componentes. Una mala manera consiste en hacer un diagrama de flujo y dividirlo en pedazos, donde cada pedazo es un componente. Es mucho mejor pensar en un componente como una abstracción. Por ejemplo, suponga que estamos escribiendo un programa que utiliza listas. Entonces, casi siempre es una buena idea colocar todas las operaciones sobre listas

dentro de un componente, el cual define la abstracción lista. Con este diseño, las listas pueden ser implementadas, depuradas, modificadas, y extendidas sin tocar el resto del programa. Por ejemplo, suponga que queremos utilizar el programa con listas que son demasiado grandes para tenerlas en la memoria principal. Entonces, es suficiente modificar el componente lista para almacenarlas en archivos, en lugar de hacerlo en la memoria principal.

**Encapsular decisiones de diseño** De manera más general, podemos decir que un componente debe encapsular una decisión de diseño.<sup>12</sup> De esta manera, cuando la decisión de diseño se modifica, solamente ese componente tiene que ser modificado. Esta es una forma muy poderosa de modularidad. La utilidad de un componente se puede evaluar mirando los cambios que él contiene. Por ejemplo, considere un programa que calcula con caracteres, tal como el ejemplo de las frecuencias de palabras de la sección 3.9.4. Idealmente, la decisión de cuál formato de caracteres utilizar (e.g., ASCII, Latin-1, o Unicode) debe ser encapsulada en un componente. Esto simplifica cambiar de un formato a otro.

**Evitar modificar interfaces de componentes** Un componente se puede modificar ya sea cambiando su implementación o cambiando su interfaz. Modificar la interfaz es problemático pues todos los componentes que dependen de esa interfaz tienen que ser reescritos o recompilados. Por lo tanto, se debe evitar modificar la interfaz. Pero, en la práctica, durante la fase de diseño del componente no se pueden evitar las modificaciones de la interfaz. Todo lo que se puede hacer es minimizar la frecuencia de esas modificaciones. Esto significa que las interfaces de los componentes más frecuentemente necesitados deben ser diseñadas tan cuidadosamente como sea posible desde el comienzo.

Presentamos un ejemplo sencillo para dejar clara esta idea. Considere un componente que ordena listas de cadenas de caracteres. Se puede cambiar su algoritmo de ordenamiento sin cambiar su interfaz. Con frecuencia, esto se puede hacer sin recomilar el resto del programa, simplemente enlazando el componente modificado. Por otro lado, si se modifica el formato de los caracteres, entonces se podría necesitar modificar la interfaz del componente. Por ejemplo, los caracteres pueden cambiar de tamaño de uno a dos bytes (si el formato ASCII se reemplaza con el formato Unicode). Esto requeriría recomilar todos los componentes que utilizan el componente modificado (directa o indirectamente), pues el código compilado puede depender del formato de los caracteres. La recompilación puede ser onerosa; modificar un componente de diez líneas puede requerir la recompilación de la mayor parte del programa, cuando el componente se utiliza frecuentemente.

### *Diseño de sistemas*

---

12. Más románticamente, algunas veces se dice que el componente tiene un “secreto.”

*Estado explícito*

**El menor número posible de dependencias externas** Un componente que depende de otro, i.e., que requiere el otro para su operación, es una fuente de problemas de mantenimiento. Si el otro componente se modifica, entonces hay que modificar el primero también. Esta es una causa muy importante del “deterioro del *software*,” i.e., *software* que funcionaba y que deja de funcionar. Por ejemplo, L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> es un sistema popular de preparación de documentos en la comunidad científica destacado por la altísima calidad de su salida [99]. Un documento L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> puede tener enlaces a otros archivos, para personalizar y extender sus capacidades. Algunos de estos otros archivos, llamados paquetes, están ampliamente estandarizados y estables. Otros, son simples personalizaciones locales, llamados archivos de estilo. De acuerdo a nuestra experiencia, es muy malo que los documentos L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> tengan enlaces a archivos de estilo en otros directorios, tal vez globales. Si estos se modifican, entonces los documentos, con frecuencia, no quedan bien formateados. Para facilitar el mantenimiento, es preferible tener copias de los archivos de estilo en cada directorio local al documento. Esto satisface un invariante sencillo: se asegura que todo documento se puede compilar en todo momento (“el *software* que funciona sigue funcionando”). Este invariante es una enorme ventaja que supera con creces las dos desventajas: (1) la memoria adicional requerida para las copias y (2) la posibilidad de que un documento puesta estar utilizando un archivo de estilo antiguo. Cuando un estilo se actualiza, el programador es libre de utilizar la versión nueva en el documento, pero sólo si es necesario. Entre tanto, el documento permanece consistente. Una segunda ventaja es que es fácil enviar el documento de una persona a otra, pues el documento es autocontenido.

**El menor número posible de niveles de referencias indirectas** Esto está relacionado con la regla anterior. Cuando A apunta a B, entonces actualizar B requiere actualizar A. Cualquier referencia indirecta es una especie de “apuntador.” La idea es evitar que el apuntador quede suelto, i.e., ya no apunte más a la fuente. Una acción en B puede causar que el apuntador de A quede suelto. B no sabe nada del apuntador de A y no puede prevenir tales cosas. Un recurso provisional consiste en no cambiar nunca a B, sino realizar copias modificadas de él. Esto puede funcionar si el sistema realiza una administración automática de la memoria global.

Dos ejemplos típicos de apuntadores problemáticos son los enlaces simbólicos en un sistema de archivos Unix y los URLs. Los enlaces simbólicos son perjudiciales para el mantenimiento de sistemas. Ellos son convenientes porque pueden referenciar otros árboles de directorios existentes, pero son de hecho una gran causa de problemas de sistemas. Los URLs son conocidos por ser extremadamente frágiles. Los URLs se referencian con frecuencia en documentos impresos, pero su tiempo de vida es mucho menor que el del documento. Esto se debe tanto a que el URL puede quedar rápidamente suelto (apuntando a una dirección que ya no existe) como a una baja calidad del servicio de Internet.

**Las dependencias deben ser predecibles** Por ejemplo, considere un comando ‘localizar’ que garantiza recuperar un archivo en una red y hacer una copia local

de él. Ese comando tiene un comportamiento sencillo y predecible, a diferencia del “ocultamiento de páginas”<sup>13</sup> realizado por los browsers de la Web. El ocultamiento de páginas es un término equivocado, pues un ocultamiento verdadero mantiene la coherencia entre el original y la copia. Para cualquier tipo de “ocultamiento”, se debe indicar claramente la política de reemplazo.

**Tomar decisiones en el nivel correcto** Por ejemplo, los tiempos de espera deben de definirse en el nivel correcto. Es un error implementar un tiempo de espera como una decisión irrevocable, tomada en un bajo nivel del sistema (un componente anidado profundamente), que se propaga hasta el más alto nivel sin ninguna forma de intervención por parte de los componentes intermedios. Este comportamiento trunca cualquier esfuerzo que el diseñador de la aplicación pueda hacer para ocultar el problema o resolverlo.

**Violaciones documentadas** Cuando se viola uno de los principios anteriores, seguramente por una buena razón (e.g., restricciones físicas tales como las limitaciones en la memoria o la separación geográfica que obliga la existencia de un apuntador), entonces esto se debe documentar! Es decir, todas las dependencias externas, todos los niveles de referencias indirectas, y todas las decisiones irrevocables, deben ser documentadas.

**Jerarquía de empaquetamiento sencilla** Un sistema no se debe almacenar en forma dispersa en un sistema de archivos, sino que debería estar reunido en un solo lugar tanto como sea pueda. Definimos una jerarquía sencilla para empaquetar componentes de un sistema. Hemos encontrado que esta jerarquía es útil tanto para documentos como para aplicaciones. El diseño más fácil de mantener va de primero. Por ejemplo, si la aplicación está almacenada en un sistema de archivos, entonces podemos definir el orden siguiente:

1. Si es posible, coloque toda la aplicación en un archivo. Este archivo puede estar estructurado en secciones, correspondientes a componentes.
2. Si lo anterior no es posible (e.g., hay archivos de diferentes tipos o gente diferente tiene que trabajar en diferentes partes simultáneamente), entonces coloque toda la aplicación en un directorio.
3. Si lo anterior no es posible (e.g., la aplicación va a ser compilada para múltiples plataformas), coloque la aplicación en una jerarquía de directorios con un directorio raíz.

---

13. Nota del traductor: *page caching*, en inglés.

#### 6.7.4. Desarrollos futuros

##### *Los componentes y el futuro de la programación*

El incremento en la utilización de componentes está cambiando la profesión de la programación. Vemos dos grandes maneras en que este cambio está sucediendo. Primero, los componentes harán accesible la programación a los usuarios de las aplicaciones, y no solamente a los desarrolladores profesionales. Dado un conjunto de componentes con un alto nivel de abstracción y una interfaz gráfica de usuario intuitiva, un usuario puede realizar muchas tareas sencillas de programación por sí mismo. Esta tendencia existe desde hace mucho tiempo en las aplicaciones en nichos específicos tales como los paquetes estadísticos, los paquetes de procesamiento de señales, y los paquetes para el control de experimentos científicos. La tendencia abarca, finalmente, las aplicaciones para el público en general.

Segundo, la programación cambiará para los desarrolladores profesionales. A medida que se desarrollan más y más componentes útiles, la granularidad de la programación se incrementará. Es decir, los elementos básicos utilizados por los programadores serán, cada vez más, los componentes grandes, en lugar de las operaciones específicas de un lenguaje. Esta tendencia es visible en herramientas tal como Visual Basic y en ambientes de componentes tal como Enterprise Java Beans. El principal cuello de botella que limita esta evolución es la especificación del comportamiento de un componente. Los componentes actuales tienden a ser excesivamente complicados y tienen unas especificaciones vagas. Esto limita su utilización. La solución, desde nuestro punto de vista, consiste en tener componentes más sencillos, factorizar mejor sus funcionalidades, y mejorar cómo se pueden conectar entre ellos.

##### *Diseño composicional versus no composicional*

La composición jerárquica puede parecer una manera muy natural de estructurar un sistema. ¡De hecho, ésta no es para nada “natural”! La naturaleza utiliza un enfoque muy diferente, el cual se puede denominar no composicional. Comparémoslos. Primero comparemos sus grafos de componentes. En un grafo de componentes, cada nodo representa un componente y existe una arista entre los nodos si los componentes se conocen entre sí. En un sistema composicional, el grafo es jerárquico. Cada componente está conectado solamente a sus hermanos, sus hijos, y sus padres. Como consecuencia, el sistema se puede descomponer, de muchas maneras, en partes independientes, y tal que la interfaz entre las partes es pequeña.

En un sistema no composicional, el grafo de componentes no tiene esta estructura. El grafo tiende a ser denso y no local. “Denso” significa que cada componente está conectado a muchos otros. “No local” significa que cada componente está conectado a amplias y diferentes partes del grafo. Entonces, descomponer el sistema en partes es más arbitrario. Las interfaces entre las partes tienden a ser más grandes. Esto dificulta la comprensión de los componentes, sin tener en cuenta su relación

con el resto del sistema. Un ejemplo de grafo no composicional es un grafo de un mundo pequeño,<sup>14</sup> el cual tiene la propiedad que el diámetro del grafo es pequeño (cada componente está a unos pocos pasos de cualquier otro componente).

Miremos por qué la composición jerárquica es apropiada para el diseño de sistemas por parte de los seres humanos. La principal restricción para los humanos es el tamaño limitado de su memoria de corto plazo. Un ser humano sólo puede conservar, simultáneamente, un número reducido de conceptos en su memoria [114]. Por lo tanto, un diseño grande debe ser dividido en partes que sean suficientemente pequeñas para ser conservadas en una sola mente de un individuo. Sin ayudas externas, esto lleva a los humanos a construir sistemas composicionales. Por otro lado, el diseño por parte de la naturaleza no tiene tal limitación. Éste funciona de acuerdo al principio de selección natural. Los sistemas nuevos se construyen combinando y modificando los ya existentes. Cada sistema es juzgado como un todo, por la forma como se desempeña en un ambiente natural. Los sistemas más exitosos son aquellos con mayor descendencia. Por lo tanto, los sistemas naturales tienden a ser no composicionales.

Parace que todo enfoque, en su forma pura, es una especie de extremo. El diseño humano es orientado por objetivos y reduccionista. El diseño natural es exploratorio e integral. ¿Tiene sentido tratar de conseguir lo mejor de ambos mundos? Podemos imaginar herramientas de construcción que permitan a los seres humanos utilizar un enfoque más “natural” en el diseño de sistemas. En este libro, no consideraremos más esta dirección, sino que nos enfocamos en el enfoque composicional.

#### 6.7.5. Lecturas adicionales

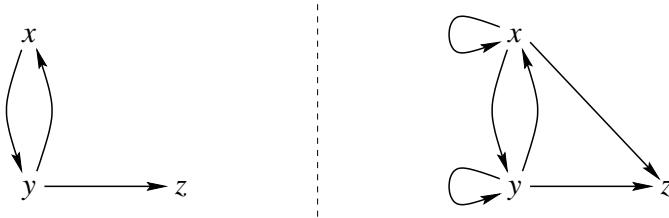
Hay muchos libros sobre diseño de programas e ingeniería de *software*. Sugerimos [138, 141] como textos generales, y [167] para una visión sobre cómo balancear el diseño y la refactorización. El libro *The Mythical Man-Month* escrito por Frederick Brooks data de 1975 pero es todavía una buena lectura [25, 26]. El libro *Software Fundamentals* es una colección de artículos de Dave Parnas, que abarca su carrera, es una buena lectura [133]. *The Cathedral and the Bazaar* de Eric Raymond es un recuento interesante sobre cómo desarrollar *software* de código abierto [143].

#### **Software por componentes: Más allá de la programación orientada a objetos**

Para mayor información, específicamente sobre componentes, recomendamos el libro *Component Software: Beyond Object-Oriented Programming* escrito por Clemens Szyperski [170]. Este libro presenta una visión general del estado del arte de

---

14. Nota del traductor: traducido del inglés *small-world graph*, el cual es un grafo donde la mayoría de los nodos no son adyacentes, pero a la mayoría de los nodos se puede llegar desde cualquier otro en pocos pasos.

*Estado explícito***Figura 6.8:** Un grafo dirigido con su clausura transitiva.

la tecnología de componentes en el momento de esta publicación. El libro combina una discusión de los fundamentos de componentes junto con una visión general de lo que existe comercialmente. Los fundamentos incluyen la definición de componente, los conceptos de interfaz y polimorfismo, la diferencia entre herencia, delegación, y reenvío, los compromisos de utilizar composición de componentes versus herencia, y cómo conectar componentes. Las tres principales plataformas comerciales que se discuten en el libro son la de OMG (Object Management Group) con su estándar de CORBA; la de Microsoft con COM (Component Object Model) y sus derivados DCOM (Distributed COM), OLE (Object Linking and Embedding), y ActiveX; y la de Sun Microsystems con Java y JavaBeans. El libro presenta una visión general, técnicamente bien balanceada, de estas plataformas.

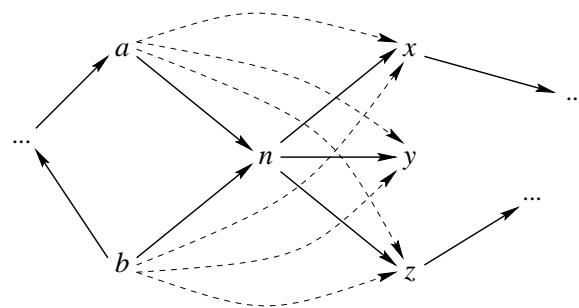
## 6.8. Casos de estudio

### 6.8.1. La clausura transitiva

Calcular la clausura transitiva de un grafo es un problema que se puede resolver razonablemente bien tanto con estado como sin él. Definimos un grafo dirigido  $G = (V, E)$  como un conjunto de nodos (o vértices)  $V$  y un conjunto de aristas  $E$  representadas como parejas  $(x, y)$  con  $x, y \in V$ , tales que  $(x, y) \in E$  si y sólo si existe una arista de  $x$  a  $y$ . El problema se define así:

Considere cualquier grafo dirigido. Calcule un grafo dirigido nuevo, denominado la *clausura transitiva*, que tiene una arista entre dos nodos siempre que el grafo original tenga un camino (una secuencia de una o más aristas) entre esos mismos nodos.

En la figura 6.8 se muestra un ejemplo de un grafo y de su clausura transitiva. Comenzamos con una descripción abstracta de un algoritmo para resolver el problema, independiente de cualquier modelo de computación en particular. Dependiendo de cómo se represente el grafo, la descripción lleva naturalmente a una implementación, declarativa o con estado, del algoritmo.



**Figura 6.9:** Un paso en el algoritmo de la clausura transitiva.

El algoritmo agrega sucesivamente aristas de acuerdo a la estrategia siguiente: para cada nodo en el grafo, agregar las aristas que conecten todos los predecesores del nodo con todos sus sucesores. Miremos cómo funciona con un ejemplo. En la figura 6.9 se muestra parte de un grafo dirigido. Allí el nodo  $n$  tiene predecesores  $a$  y  $b$  y sucesores  $x$ ,  $y$ , y  $z$ . Cuando el algoritmo encuentra el nodo  $n$ , agrega las seis aristas  $a \rightarrow x$ ,  $a \rightarrow y$ ,  $a \rightarrow z$ ,  $b \rightarrow x$ ,  $b \rightarrow y$ , y  $b \rightarrow z$ . Después que el algoritmo ha hecho este tratamiento con todos los nodos, el algoritmo termina. Podemos describir el algoritmo de la manera siguiente:

Para cada nodo  $x$  en el grafo  $G$ :

    Para cada nodo  $y$  en  $\text{pred}(x, G)$ :

        Para cada nodo  $z$  en  $\text{succ}(x, G)$ :

            agregue la arista  $(y, z)$  a  $G$ .

Definimos la función  $\text{pred}(x, G)$  como el conjunto de nodos predecesores de  $x$ , i.e. los nodos con aristas que terminan en  $x$ , y la función  $\text{succ}(x, G)$  como el conjunto de nodos sucesores de  $x$ , i.e., los nodos con aristas que comienzan en  $x$ .

¿Por qué funciona este algoritmo? Considere cualquier dos nodos  $a$  y  $b$  con un camino entre ellos:  $a \rightarrow n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_k \rightarrow b$  (donde  $k \geq 0$ ). Tenemos que mostrar que el grafo final tiene una arista de  $a$  a  $b$ . El algoritmo recorre los nodos del  $n_1$  al  $n_k$ , en algún orden. Cuando el algoritmo llega a un nodo  $n_i$ , realiza un “corto circuito” del nodo, i.e., crea un camino nuevo de  $a$  a  $b$  que evita pasar por ese nodo. Por lo tanto, cuando el algoritmo ha recorrido todos los nodos, ya se ha creado un camino que los evita a todos, i.e., hay una arista directa de  $a$  a  $b$ .

### Representación de un grafo

Para transformar el algoritmo en un programa, tenemos que escoger, primero que todo, una representación para los grafos dirigidos. Consideraremos dos representaciones posibles:

- La representación como *lista de adyacencias*. El grafo es una lista con elementos

*Estado explícito*

de la forma  $i\#Ns$  donde  $i$  identifica un nodo y  $Ns$  es la lista ordenada de sus sucesores inmediatos. Como veremos más adelante, es más eficiente calcular con listas ordendadas de sucesores que con listas no ordenadas.

- La representación como *matriz*. El grafo es un arreglo de dos dimensiones. El elemento con coordenadas ( $i,j$ ) es **true** si existe una arista del nodo  $i$  al nodo  $j$ . Si no es así, entonces ese elemento es **false**.

Encontramos que la escogencia de la representación influye fuertemente en cuál es el mejor modelo de computación para resolver el problema. En adelante, suponemos que todos los grafos tienen al menos un nodo y que los nodos son enteros consecutivos. Primero presentamos un algoritmo declarativo que utiliza la representación de lista de adyacencias [130]. Luego presentamos un algoritmo con estado que utiliza la representación de matriz [39]. Luego presentamos un segundo algoritmo declarativo que también utiliza la representación de matriz. Finalmente, comparamos los tres algoritmos.

*Conversión entre representaciones*

Para que la comparación de los algoritmos sea más fácil, primero definimos rutinas para convertir la representación como lista de adyacencias en una representación como matriz y viceversa. La conversión de lista de adyacencias a matriz se presenta a continuación:

```
fun {LaM GL}
 M={Map GL fun {$ I#_} I end}
 L={FoldL M Min M.1}
 H={FoldL M Max M.1}
 GM={NewArray L H nada}
in
 for I#Ns in GL do
 GM.I:={NewArray L H false}
 for J in Ns do GM.I.J:=true end
 end
 GM
end
```

En esta rutina, así como en las siguientes, usamos  $GL$  para la representación del grafo como lista y  $GM$  para la representación del grafo como matriz. La conversión de matriz a lista de adyacencias se presenta a continuación:

```
fun {MaL GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for I in L..H collect:C do
 {C I#for J in L..H collect:D do
 if GM.I.J then {D J} end
 end}
 end
end
```

```

fun {ClauTransDecl G}
 xs={Map G fun {$ X#_} X end}
in
 {FoldL xs
 fun {$ InG X}
 SX={Suc X InG} in
 {Map InG
 fun {$ Y#SY}
 Y#if {Member X SY} then
 {Unión SY SX} else SY end
 end
 end G}
 end
 }

```

**Figura 6.10:** Clausura transitiva (primera versión declarativa).

Aquí se utiliza la sintaxis de ciclo, incluyendo el procedimiento de acumulación `collect:C`, de gran utilidad.

#### *Algoritmo declarativo*

Presentamos primero un algoritmo declarativo para calcular la clausura transitiva de un grafo. El grafo se representa como una lista de adyacencias. El algoritmo requiere de dos rutinas auxiliares, `Suc`, la cual devuelve la lista de sucesores de un nodo dado, y `Unión`, la cual calcula la unión de dos listas ordenadas. Desarrollamos el algoritmo por transformación sucesiva del algoritmo abstracto.

Este método de diseño se conoce como refinamiento paso a paso.

El ciclo más externo en el algoritmo abstracto transforma el grafo en etapas sucesivas. Nuestro algoritmo declarativo hace lo mismo utilizando la función `FoldL`, la cual define un ciclo con acumulación. Esto significa que podemos definir la función principal `ClauTransDecl` así:

```

fun {ClauTransDecl G}
 xs={Nodos G} in
 {FoldL xs
 fun {$ InG X}
 SX={Suc X InG} in
 Para cada nodo Y en pred(X, InG):
 Para cada nodo Z en SX:
 agregue la arista (Y, Z)
 end G}
 end

```

El siguiente paso es implementar los dos ciclos internos:

```

 Para cada nodo Y en pred(X, InG):
 Para cada nodo Z en SX:
 agregue la arista (Y, Z)

```

Estos ciclos transforman un grafo en otro, agregando aristas. Como nuestro grafo

*Estado explícito*

está representado por una lista, una alternativa natural consiste en utilizar `Map`, la cual transforma una lista en otra. Esto resulta en el código siguiente, donde `Unión` se utiliza para unir la lista de sucesores de `x` con la de sucesores de `y`:

```
{Map InG
 fun {$ Y#SY}
 Y#if "Y en pred(x, InG)" then
 {Unión SY SX} else SY end
 end}
```

Terminamos observando que `y` está en `pred(x, InG)` si y sólo si `x` está en `succ(y, InG)`. Esto significa que podemos escribir la condición de la declaración `if` así:

```
{Map InG
 fun {$ Y#SY}
 Y#if {Member x SY} then
 {Unión SY SX} else SY end
 end}
```

Al colocar todo junto llegamos a la definición final presentada en la figura 6.10. Allí se utiliza `Map` para calcular `{Nodos G}`. Concluimos resaltando que `FoldL`, `Map`, y otras rutinas como `Member`, `Filter`, etc., son procedimientos básicos que deben dominarse cuando se escriben algoritmos declarativos.

Para finalizar nuestra presentación del algoritmo declarativo, definimos las dos rutinas auxiliares. `Suc` devuelve la lista de sucesores de un nodo:

```
fun {Suc x G}
 case G of Y#SY|G2 then
 if X==Y then SY else {Suc x G2} end
 end
end
```

`Suc` supone que `x` se encuentra siempre en la lista de adyacencia, lo cual es cierto en nuestro caso. `Unión` devuelve la unión de dos conjuntos, donde los dos conjuntos están representados por listas ordenadas:

```
fun {Unión A B}
 case A#B
 of nil#B then B
 [] A#nil then A
 [] (X|A2)#(Y|B2) then
 if X==Y then X|{Unión A2 B2}
 elseif X<Y then X|{Unión A2 B}
 elseif X>Y then Y|{Unión A B2}
 end
 end
end
```

El tiempo de ejecución de `Unión` es proporcional a la longitud de la lista más pequeña, debido a que las listas de entrada están ordenadas. Si las listas de entrada no estuvieran ordenadas, su tiempo de ejecución sería proporcional al producto de sus longitudes (¿por qué?), el cual, normalmente, es más grande.

```

proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
 for I in L..H do
 if GM.I.K then
 for J in L..H do
 if GM.K.J then GM.I.J:=true end
 end
 end
 end
 end

```

**Figura 6.11:** Clausura transitiva (versión con estado).

### *Algoritmo con estado*

Presentamos un algoritmo con estado para calcular la clausura transitiva de un grafo. El grafo se representa con una matriz. Este algoritmo supone que la matriz contiene el grafo inicial. Luego calcula la clausura transitiva en el mismo sitio, i.e., actualizando la misma matriz de entrada. En la figura 6.11 se presenta el algoritmo. Para cada nodo  $k$ , se mira cada arista potencial  $(i, j)$  y se agrega al grafo si ya había una arista de  $i$  a  $k$  y otra de  $k$  a  $j$ . Ahora mostramos la transformación paso a paso que lleva al algoritmo. Primero reformulamos el algoritmo abstracto con los nombres de variables apropiados:

Para cada nodo  $k$  en el grafo  $G$ :

Para cada nodo  $i$  en  $\text{pred}(k, G)$ :

Para cada nodo  $j$  en  $\text{succ}(k, G)$ :

agregue la arista  $(i, j)$  a  $G$ .

Esto nos lleva al refinamiento siguiente:

```

proc {ClauTransEstado GM}
 L={Array.low GM}
 H={Array.high GM}
in
 for K in L..H do
 for I in L..H do
 if GM.I.K then
 Para cada J en succ(K, GM) haga GM.I.J:=true
 end
 end
 end

```

*Estado explícito*

```

fun {ClauTransDecl2 GT}
 H={Width GT}
 fun {Ciclo K InG}
 if K=<H then
 G={MakeTuple g H} in
 for I in 1..H do
 G.I={MakeTuple g H}
 for J in 1..H do
 G.I.J = InG.I.J orelse (InG.I.K andthen InG.K.J)
 end
 end
 {Ciclo K+1 G}
 else InG end
 end
in
 {Ciclo 1 GT}
end

```

**Figura 6.12:** Clausura transitiva (segunda versión declarativa).

Note que  $J$  está en  $\text{succ}(K, GM)$  si  $GM.K.J$  es **true**. Esto quiere decir que podemos reemplazar el ciclo interno por:

```

for J in L..H do
 if GM.K.J then GM.I.J:=true end
end

```

*Segundo algoritmo declarativo*

Inspirados en el algoritmo con estado, desarrollamos un segundo algoritmo declarativo. Este nuevo algoritmo utiliza una serie de tuplas para almacenar las aproximaciones sucesivas de la clausura transitiva. Usamos la variable  $GT$  en lugar de  $GM$  para hacer énfasis en este cambio de representación. Una tupla es un registro con campos numerados consecutivamente de 1 a un valor máximo. Por tanto este algoritmo se restringe para nodos cuya numeración comienza en 1. Note que `MakeTuple` crea una tupla de variables no-ligadas. La figura 6.12 presenta el algoritmo.

Éste es algo más complicado que la versión con estado. Cada iteración de el ciclo externo utiliza el resultado de la iteración previa ( $InG$ ) como entrada para la generación siguiente ( $G$ ). La función recursiva `Ciclo` pasa el resultado de una iteración a la siguiente. Aunque esto parece ser un poco complicado, tiene las ventajas del modelo declarativo. Por ejemplo, es sencillo convertirlo en un algoritmo concurrente para calcular la clausura transitiva utilizando el modelo del capítulo 4. El algoritmo concurrente puede ejecutarse bien en un procesador paralelo. Sólo agregamos **thread** ... **end** para parallelizar los dos ciclos externos, como se muestra en la figura 6.13. Esto nos produce una implementación del algoritmo con flujo de datos y paralela. La sincronización se realiza a través de

```

fun {ClauTransDecl2 GT}
 H={Width GT}
 fun {Ciclo K InG}
 if K=<H then
 G={MakeTuple g H} in
 thread
 for I in 1..H do
 thread
 G.I={MakeTuple g H}
 for J in 1..H do
 G.I.J = InG.I.J orelse
 (InG.I.K andthen InG.K.J)
 end
 end
 end
 end
 {Ciclo K+1 G}
 else InG end
 end
in
 {Ciclo 1 GT}
end

```

**Figura 6.13:** Clausura transitiva (versión concurrente/paralela).

las tuplas las cuales contienen, inicialmente, variables no-ligadas. Las tuplas se comportan como estructuras-I en una máquina de flujo de datos (ver sección 4.9.5). Es interesante hacer el ejercicio de hacer un dibujo de un programa en ejecución, con estructuras de datos e hilos.

### Ejemplos de ejecución

Calculemos la clausura transitiva del grafo [1#[2 3] 2#[1] 3#nil]. Este es el mismo grafo que mostramos en la figura 6.8 salvo que usamos enteros para representar los nodos. Así usamos los algoritmos declarativos:

```
{Browse {ClauTransDecl [1#[2 3] 2#[1] 3#nil]}}
```

Así usamos el algoritmo con estado:

```

declare GM in
{ClauTransEstado GM={LaM [1#[2 3] 2#[1] 3#nil]}}
{Browse {MaL GM}}

```

Esta última es un poco más complicada por las invocaciones a LaM y MaL, las cuales usamos para que tanto la entrada como la salida sean listas de adyacencias. Los tres algoritmos dan como resultado [1#[1 2 3] 2#[1 2 3] 3#nil].

### **Discusión**

Tanto los algoritmos declarativos como el algoritmo con estado son variaciones del mismo algoritmo conceptual, el cual se conoce como el algoritmo de Floyd-Warshall. Los tres algoritmos tienen un tiempo de ejecución asintótico de  $O(n^3)$  para un grafo de  $n$  nodos. ¿Entonces, cuál algoritmo es mejor? Exploremos diferentes aspectos de esta pregunta:

■ Un primer aspecto es qué tan fácil de entender es el algoritmo, y qué tan fácil es razonar con él. Sorprendentemente, tal vez, el algoritmo con estado es el de estructura más sencilla. Consiste de tres ciclos anidados sencillos que actualizan una matriz de una manera directa. Ambos algoritmos declarativos tienen una estructura más compleja:

- El primero toma una lista de adyacencias y la pasa a través de una secuencia de etapas conectadas a manera de canal. Cada etapa recibe una lista de entrada y, de manera incremental, crea una lista de salida.
- El segundo tiene una estructura similar a la del algoritmo con estado, pero crea una secuencia de tuplas, donde también las etapas se conectan a manera de canal.

La programación en el modelo declarativo obliga a estructurar el algoritmo a manera de canal, y a escribirlo con componentes independientes pequeños. La programación en el modelo con estado sugiere (pero no obliga) estructurar el algoritmo como un bloque monolítico, el cual es más difícil de descomponer. El modelo con estado ofrece más libertad en la manera de escribir el programa. Dependiendo de su propio punto de vista, esto puede ser una buena o una mala cosa.

■ Un segundo aspecto es el desempeño: tiempo de ejecución y utilización de la memoria. Ambos algoritmos tienen los mismos tiempos de ejecución y el mismo consumo de memoria activa en términos asintóticos. Hemos medido los tiempos de ejecución de ambos algoritmos en varios grafos aleatorios grandes. Considere un grafo aleatorio de 200 nodos en el cual existe una arista entre cualquier par de nodos con probabilidad  $p$ . Para  $p$  mayor que 0.05, el primer algoritmo declarativo tomó alrededor de 10 segundos, el segundo alrededor de 12 segundos, y el algoritmo con estado alrededor de 15 segundos. Para  $p$  tiendiendo a 0, el primer algoritmo declarativo tiende a 0 segundos, y los otros algoritmos incrementan su tiempo a 16 y 20 segundos, respectivamente.<sup>15</sup> Concluimos que el primer algoritmo declarativo siempre tiene mejor desempeño que los otros dos. La representación del grafo como lista de adyacencias es mejor que la representación como matriz cuando el grafo es poco denso.

Por supuesto, las conclusiones de esta comparación no son de ninguna manera definitivas. Hemos escogido versiones sencillas y limpias de cada estilo, pero hay

---

15. Usando Mozart 1.1.0 sobre un procesador Pentium III a 500 MHz.

```
fun {CarDePalabra C} ... end

fun {PalAAtomos PW} ... end

fun {CarsAPalabras PW Cs} ... end

Aregar=Dictionary.agregar
ConsultarCond=Dictionary.consultarCond

proc {IncPal D W}
 {Aregar D W {ConsultarCond D W 0}+1}
end

proc {ContarPalabras D Ws}
 case Ws
 of W|Wr then
 {IncPal D W}
 {ContarPalabras D Wr}
 [] nil then skip
 end
end

fun {FrecPalabras Cs}
 D={DiccionarioNuevo}
in
 {ContarPalabras D {CarsAPalabras nil Cs}}
 D
end
```

Figura 6.14: Frecuencias de palabras (usando diccionario con estado).

muchas posibles variaciones. Por ejemplo, el primer algoritmo declarativo se puede modificar para utilizar una operación Unión con estado. El algoritmo con estado se puede modificar para que salga del ciclo cuando no haya más aristas para agregar. ¿Qué podemos concluir de esta comparación?

- En ambos modelos, declarativo y con estado, es razonable implementar la clausura transitiva.
- La escogencia de la representación (lista de adyacencias o matriz) puede ser más importante que la escogencia del modelo de computación.
- Los programas declarativos tienden a ser menos legibles que los programas con estado, pues los primeros tienen que ser escritos a manera de canal.
- Los programas con estado tienden a ser más monolíticos que los programas declarativos, porque el estado explícito se puede modificar en cualquier orden.
- Puede ser más fácil paralelizar un programa declarativo porque existen pocas dependencias entre sus partes.

### 6.8.2. Frecuencias de palabras (usando diccionario con estado)

En la sección 3.7.3 mostramos cómo usar los diccionarios para contar el número de palabras diferentes en un archivo de texto. Comparamos los tiempos de ejecución de las tres versiones del contador de frecuencias de palabras, cada una con una implementación diferente del diccionario. Las primeras dos versiones usaron diccionarios declarativos (implementados con listas y árboles binarios, respectivamente) y la tercera utilizó la versión de diccionario predefinida en Mozart (implementada con estado). La versión mostrada en la figura 6.14, que utiliza diccionarios con estado, es ligeramente diferente de la mostrada en la figura 3.30, que usa diccionarios declarativos:

- La versión con estado necesita pasar solamente un argumento como entrada a cada procedimiento que utiliza un diccionario.
- La versión declarativa tiene que usar dos argumentos para esos procedimientos: uno para el diccionario de entrada y otro para el diccionario de salida. En la figura 3.30, el argumento correspondiente al diccionario de salida existe, pero está oculto por la notación funcional.

La diferencia se ve en las operaciones `Agregar`, `IncPal`, `ContarPalabras`, y `FrecPalabras`. Por ejemplo, en la figura 6.14 se utiliza `{Agregar D LI X}` con estado, el cual actualiza `D`. Mientras tanto, en la figura 3.30 se utiliza `{Agregar D1 LI X D2}` declarativo, el cual lee `D1` y devuelve un diccionario nuevo `D2`.

### 6.8.3. Generación de números aleatorios

Una operación primitiva muy útil es un generador de números aleatorios. Éste permite al computador “lanzar los dados,” por así decirlo. ¿Cómo generar números aleatorios en un computador? Aquí presentamos unos pocos elementos para comprender; vea Knuth [90] para una discusión profunda de la teoría subyacente a los generadores de números aleatorios y del concepto de aleatoriedad.

#### *Diferentes enfoques*

Uno puede imaginarse las siguientes maneras de generar números aleatorios:

- Una primera técnica consistiría en usar eventos impredecibles en el computador, e.g., relacionados con la concurrencia, como se explicó en el capítulo anterior. Lástimosamente, su impredecibilidad no sigue leyes sencillas. Por ejemplo, utilizar el planificador de hilos como una fuente de aleatoriedad producirá algunas fluctuaciones, pero que no tienen una distribución de probabilidad útil. Adicionalmente, ellas están íntimamente ligadas con la computación misma en una manera no obvia, al punto que aún si su distribución fuera conocida, sería dependiente de la computación. Por tanto, ésta no es una buena fuente de números aleatorios.
- Una segunda técnica consistiría en confiar en una fuente de verdadera aleato-

riedad. Por ejemplo, los circuitos electrónicos generan ruido, el cual es una señal completamente impredecible cuya distribución de probabilidad aproximada es conocida. El ruido proviene de las profundidades del mundo cuántico, así que para efectos prácticos es verdaderamente aleatorio. Pero hay dos problemas. Primero, la distribución de probabilidad no se conoce exactamente: puede variar ligeramente de un circuito a otro o con la temperatura ambiente. El primer problema no es serio; existen maneras de “normalizar” los números aleatorios de manera que su distribución sea una constante, una conocida. Hay un segundo problema, más serio: la aleatoriedad no se puede reproducir, salvo almacenando los números aleatorios y repitiéndolos. Puede parecer extraño buscar que se pueda reproducir una fuente de aleatoriedad, pero esto es perfectamente razonable. Por ejemplo, la aleatoriedad puede ser la entrada a un simulador. Nos gustaría variar algún parámetro en el simulador de forma que cualquier variación en el simulador dependa solamente de ese parámetro, y no de la variación de los números aleatorios. Por esta razón, los computadores, normalmente, no están conectados a verdaderas fuentes de aleatoriedad.

■ Puede parecer que estamos trabajando en una esquina muy estrecha. Nos gustaría tener verdadera aleatoriedad pero también nos gustaría que fuera reproducible. ¿Cómo podemos resolver este dilema? La solución es sencilla: calculemos los números aleatorios. ¿Cómo podría esto generar verdaderos números aleatorios? La respuesta es simple, no se puede. Pero los números pueden parecer aleatorios, para efectos prácticos. Ellos se denominan números seudoaleatorios. ¿Qué significa esto? No es sencillo de definir. Aproximadamente, los números generados deben tener el mismo comportamiento que los verdaderos números aleatorios, para el uso que haremos de ellos.

La tercera solución, calcular los números aleatorios, es la que casi siempre está implementada. La pregunta es, ¿qué algoritmo utilizar? ¡Ciertamente, un algoritmo que no sea escogido al azar! Knuth [90] muestra los obstáculos de este enfoque: casi siempre da malos resultados. Necesitamos un algoritmo del que se conozcan sus buenas propiedades. No podemos garantizar que los números aleatorios sean suficientemente buenos, pero podemos tratar de lograrlo. Por ejemplo, los números aleatorios generados deben satisfacer propiedades estadísticas fuertes, tener la distribución correcta, y su período debe ser suficientemente largo. Vale la pena expandir el último punto: como un generador de números aleatorios realiza un cálculo con información finita, finalmente tendrá que repetirse a sí mismo. Claramente, el período de repetición debe ser muy largo.

### *Números aleatorios uniformemente distribuidos*

Un generador de números aleatorios almacena un estado interno, con el cual calcula el siguiente número aleatorio y el siguiente estado interno. El estado debe ser suficientemente grande para permitir un período largo. El generador de números aleatorios se inicializa con un número denominado su semilla. Inicializarlo de nuevo,

*Estado explícito*

con la misma semilla, debe producir la misma secuencia de números aleatorios. Si no deseamos la misma secuencia, entonces podemos inicializarlo con información que nunca sea la misma, tal como la fecha y la hora actual. Los computadores modernos casi siempre tienen una operación para obtener la información del tiempo. Ahora podemos definir un generador de números aleatorios como una abstracción de datos:

- `{NuevoAlea ?Alea ?Inic ?Max}` devuelve tres referencias: un generador de números aleatorios `Alea`, su procedimiento de inicialización `Inic`, y su máximo valor `Max`. Cada generador tiene su propio estado interno. Para mejores resultados, `Max` debe ser grande. Esto permite al programa reducir los números aleatorios a los dominios más pequeños que el programa necesita para sus propios propósitos.
- `{Inic Semilla}` inicializa el generador con la semilla entera `Semilla`, la cual debe estar en el rango de  $0, 1, \dots, Max-1$ . Para que haya muchas secuencias posibles, `Max` debe ser grande. La inicialización se puede hacer en cualquier momento.
- `X={Alea}` genera un número aleatorio nuevo `x` y actualiza el estado interno. `x` es un entero en el rango  $0, 1, \dots, Max-1$  y tiene una distribución de probabilidad uniforme, i.e., todos los enteros tienen la misma posibilidad de ser generados.

¿Cómo calcular un número aleatorio nuevo? Resulta que un buen método, sencillo, es el generador lineal congruente. Si  $x$  es el estado interno y  $s$  es la semilla, entonces el estado interno se actualiza así:

$$\begin{aligned}x_0 &= s \\x_n &= (ax_{n-1} + b) \bmod m\end{aligned}$$

Las constantes  $a$ ,  $b$ , y  $m$  tienen que ser escogidas cuidadosamente, de manera que la secuencia  $x_0, x_1, x_2, \dots$ , tenga buenas propiedades. El estado interno  $x_i$  es un entero uniformemente distribuido entre  $0$  y  $m - 1$ . Es fácil implementar este generador:

```
local A=333667 B=213453321 M=1000000000 in
proc {NuevoAlea ?Alea ?Inic ?Max}
 X={NewCell 0} in
 fun {Alea} X:=(A*X+B) mod M end
 proc {Inic Semilla} X:=Semilla end
 Max=M
end
end
```

Note que el cuerpo de la función `X:=(A*X+B) mod M` realiza un intercambio, pues ésta es una expresión de posición. Este es uno de los métodos más sencillos que tiene un comportamiento razonablemente bueno. Existen métodos más sofisticados que son aún mejores.

*Utilizando la pereza en lugar del estado*

El algoritmo lineal congruente se puede ampaquetar de una manera completamente diferente, a saber, una función perezosa. Para producir el siguiente número

aleatorio, es suficiente leer el próximo elemento del flujo. Esta es la definición:

```
local A=333667 B=213453321 M=1000000000 in
 fun lazy {ListaAlea S0}
 S1=(A*S0+B) mod M in S1|{ListaAlea S1}
 end
end
```

En lugar de utilizar una celda, el estado se almacena en un argumento recursivo `ListaAlea`. En lugar de invocar `Alea` explícitamente para producir el número siguiente, `ListaAlea` se invoca implícitamente cuando el número siguiente se necesita. La pereza actúa como una especie de freno, asegurándose que la computación sólo avanza a la velocidad que los resultados lo requieren. Observamos que no se necesita programación de alto orden para crear muchos generadores de números aleatorios. Cada invocación a `ListaAlea` genera una nueva secuencia de números aleatorios.

### *Distribuciones no uniformes*

Una buena técnica para generar números aleatorios de cualquier tipo de distribución, es comenzar con un número aleatorio uniformemente distribuido. A partir de él, calculamos un número con otra distribución. Explicamos cómo generar las distribuciones de Gauss y exponencial utilizando esta técnica. Primero definimos un generador nuevo:

```
declare Alea Iinic Max in {NuevoAlea Alea Iinic Max}
```

Ahora, definimos las funciones para generar una distribución uniforme de 0 a 1 y una distribución uniforme entera de `A` a `B`, inclusive:

```
FMax={IntToFloat Max}
fun {Uniforme}
 {IntToFloat {Alea}}/FMax
end

fun {UniformeEnt A B}
 A+{FloatToInt {Floor {Uniforme}*{IntToFloat B-A+1}}}
end
```

Utilizamos `Uniforme` para generar variables aleatorias con otras distribuciones. Primero, generaremos variables aleatorias con una distribución exponencial. Para esta distribución, la probabilidad que  $X \leq x$  es  $D(x) = 1 - e^{-\lambda x}$ , donde  $\lambda$  es un parámetro denominado la intensidad. Como  $X \leq x$  si y sólo si  $D(X) \leq D(x)$ , se sigue que la probabilidad que  $D(X) \leq D(x)$  es  $D(x)$ . Escribiendo  $y = D(x)$ , se sigue que la probabilidad que  $D(X) \leq y$  es  $y$ . Por lo tanto,  $D(X)$  está distribuida uniformemente. Suponga que  $D(X) = U$  donde  $U$  es una variable aleatoria uniformemente distribuida. Entonces tenemos que  $X = -\ln(1 - U)/\lambda$ . Esto nos lleva a la función siguiente:

*Estado explícito*

```
fun {Exponencial Lambda}
 ~{Log 1.0-{Uniforme}}/Lambda
end
```

Ahora generemos una distribución normal con media 0 y varianza 1. Una distribución normal también se denomina una distribución de Gauss. Usamos la técnica siguiente. Sean  $U_1$  y  $U_2$  dos variables distribuidas uniformemente de 0 a 1. Sea  $R = \sqrt{-2 \ln U_1}$  y  $\phi = 2\pi U_2$ . Entonces  $X_1 = R \cos \phi$  y  $X_2 = R \sin \phi$  son variables independientes con distribución de Gauss. La prueba de este hecho está más allá del alcance del libro; ella se puede consultar en [90]. Esto nos lleva a la función siguiente:

```
DosPi=4.0*{Float.acos 0.0}
fun {Gauss}
 {Sqrt ~2.0*{Log {Uniforme}}} * {Cos DosPi*{Uniforme}}
end
```

Como cada invocación nos produce dos variables Gaussianas, podemos usar una celda para recordar un resultado para la siguiente invocación:

```
local CeldaGauss={NewCell nil} in
 fun {Gauss}
 Prev={Exchange CeldaGauss $ nil}
 in
 if Prev\=nil then Prev
 else R Phi in
 R={Sqrt ~2.0*{Log {Uniforme}}}
 Phi=DosPi*{Uniforme}
 CeldaGauss:=R*{Cos Phi}
 R*{Sin Phi}
 end
 end
 end
```

Cada invocación de **Gauss** calcula dos variables Gaussianas independientes; devolvemos una y almacenamos la otra en una celda. La siguiente invocación la devuelve sin realizar ningún cálculo.

#### 6.8.4. Simulación “de boca en boca”

Simulemos la manera en que los usuarios Web “navegan” en Internet. Navegar entre sitios Web significa cargar sucesivamente diferentes sitios Web. Para conservar la sencillez de nuestro simulador, sólo miraremos un aspecto del sitio Web, a saber su desempeño. Esto es razonable cuando se navega entre portales Web, cada uno de los cuales provee un conjunto de servicios similares. Suponga que existen  $n$  sitios Web de igual contenido y un total de  $m$  usuarios. Cada sitio Web tiene un desempeño constante. A cada usuario le gustaría ir al sitio Web de mejor desempeño. Pero no existe una medida global del desempeño; la única manera en que un usuario puede inferir algo sobre el desempeño, es preguntándole a otros usuarios. Decimos que la información pasa “de boca en boca.” Esto define las siguientes reglas de simulación:

- Cada sitio tiene un desempeño constante. Suponemos que las constantes están

uniformemente distribuidas entre los sitios.

- Cada usuario sabe en qué sitio está.
- Cada sitio sabe cuántos usuarios hay en él.
- Cada usuario trata de ir a un sitio con el mejor desempeño. Sin embargo, la información del desempeño no es exacta: está perturbada por ruido Gaussiano.
- Una ronda de la simulación consiste en que todos los usuarios van a un sitio.

Con estas reglas podríamos esperar que los usuarios abarroten, finalmente, los sitios de mejor desempeño. ¿Pero, ocurre realmente así? Una simulación puede darnos la respuesta.

Escribamos un programa de simulación pequeño. Primero, establezcamos las constantes globales. Usamos las funciones `Inic`, `UniformeEnt`, y `Gauss` definidas en la sección anterior. Hay  $n$  sitios,  $m$  usuarios, y  $t$  rondas de simulación. Inicializamos el generador de números aleatorios y escribimos, durante la simulación, la información en el archivo `'debocaenboca.txt'`. Usamos las operaciones de escritura incremental en archivos, definidas en el módulo `File` en el sitio Web del libro. Con 10000 sitios, 500000 usuarios, y 200 rondas, este programa comienza así:

```
declare
N=10000 M=500000 T=200
{Inic 0}
{File.writeOpen 'debocaenboca.txt'}
proc {Escribir S}
 {File.write {Value.toVirtualString S 10 10}#"\\n"}
end
```

Luego, decidimos cómo almacenar la información de la simulación. Nos gustaría almacenarla en registros o tuplas, pues éstas estructuras son fáciles de manipular. Pero, no pueden ser modificadas. Entonces, almacenamos la información en diccionarios. Los diccionarios son muy parecidos a los registros salvo que se pueden modificar dinámicamente (ver sección 6.5.1). Cada sitio elige su desempeño aleatoriamente. Cada sitio tiene un diccionario contenido su desempeño y el número de usuarios que hay en él. El código siguiente crea la información inicial del sitio:

```
Sitios={MakeTuple sitios N}
for I in 1..N do
 Sitios.I={Record.toDictionary
 o(usuarios:0 desempeño:{IntToFloat {UniformeEnt 1
80000}})}
end
```

Cada usuario elige su sitio al azar. Cada usuario tiene un diccionario contenido su sitio actual, y actualiza la información de `Sitios`. El código siguiente crea la información inicial de los usuarios:

```
Usuarios={MakeTuple usuarios M}
for I in 1..M do S={UniformeEnt 1 N} in
 Usuarios.I={Record.toDictionary o(sitioActual:S)}
 Sitios.S.usuarios := Sitios.S.usuarios + 1
end
```

*Estado explícito*

```
proc {RondaUsuario I}
 U=Usuarios.I
 % Pregunta a otros tres usuarios su información de desempeño
 L={List.map [{UniformeEnt 1 M} {UniformeEnt 1 M} {UniformeEnt
 1 M}]}
 fun {$ X}
 (Usuarios.X.sitioActual) #
 Sitios.(Usuarios.X.sitioActual).desempeño
 + {Gauss}*{IntToFloat N}
 end}
 % Calcula el mejor sitio
 MS#MP = {List.foldL L
 fun {$ X1 X2} if X2.2>X1.2 then X2 else X1 end end
 U.sitioActual #
 Sitios.(U.sitioActual).desempeño
 + {Abs {Gauss}*{IntToFloat N}}}
 in
 if MS\=U.sitioActual then
 Sitios.(U.sitioActual).usuarios :=
 Sitios.(U.sitioActual).usuarios - 1
 U.sitioActual := MS
 Sitios.MS.usuarios := Sitios.MS.usuarios + 1
 end
 end
```

**Figura 6.15:** Un turno en la simulación “de boca en boca.”

Ahora que tenemos todas las estructuras de datos, realicemos una ronda de la simulación. En la figura 6.15 se define la función `{RondaUsuario I}`, la cual simula el turno del usuario `I`, en la ronda. En un turno, el usuario le pregunta a otros tres usuarios por el desempeño de sus sitios; con esa información calcula su sitio nuevo, y actualiza el sitio y la información del usuario.

En la figura 6.16 se presenta la simulación completa. Para lograr que el simulador sea autocontenido, podemos colocar todo el código del simulador en un procedimiento con parámetros `N` (número de sitios), `M` (número de usuarios), `T` (número de rondas), y el nombre del archivo de salida. Esto nos permite realizar muchas simulaciones con parámetros diferentes.

¿Cuál es el resultado de la simulación? ¿Se agruparán los usuarios alrededor de los sitios con mejor desempeño, aunque cada uno de ellos sólo tenga una vista inexacta y limitada de lo que está sucediendo? Al ejecutar la simulación propuesta, se encuentra que el número de sitios no vacíos (con un usuario al menos) decrece suavemente en forma exponencial inversa de un número inicial de 10000 a menos de 100 después de ochenta y tres rondas. El desempeño promedio de los sitios de los usuarios se incrementa de aproximadamente 40000 (la mitad del máximo) a más de 75000 (cerca del 6 % del máximo) después de tan solo diez rondas. Luego podemos concluir, preliminarmente, que los mejores sitios se encuentran rápidamente y que

```

for J in 1..N do
 {Escribir {Record.adjoinAt
 {Dictionary.toRecord sitio Sitios.J} nombre J}}
end
{Escribir finDeRonda(tiempo:0 sitiosNoVacíos:N)}
for I in 1..T do X={NewCell 0} in
 for U in 1..M do {RondaUsuario U} end
 for J in 1..N do H=Sitios.J.usuarios in
 if H\=0 then
 {Escribir {Record.adjoinAt
 {Dictionary.toRecord sitio Sitios.J} name J}}
 X := @X+1
 end
 end
 {Escribir finDeRonda(tiempo:I sitiosNoVacíos:@X)}
end
{File.writeClose}

```

**Figura 6.16:** La simulación “de boca en boca” completa.

los peores sitios se abandonan rápidamente, aún cuando se propaga “de boca en boca” una información muy aproximada. Por supuesto, nuestra simulación tiene algunas suposiciones que simplifican el contexto. Siéntase en libertad de cambiar las suposiciones y explore los resultados. Por ejemplo, suponer que un usuario puede escoger cualesquiera otros tres usuarios no es realista—supone que cada usuario conoce a todos los otros. Esto hace que la convergencia sea demasiado rápida. Vea la sección 6.10, de ejercicios, para una suposición más realista del conocimiento del usuario.

## 6.9. Temas avanzados

### 6.9.1. Limitaciones de la programación con estado

La programación con estado tiene algunas limitaciones fuertes debidas al uso del estado explícito. La programación orientada a objetos es un caso especial de programación con estado, luego ella sufre de las mismas limitaciones.

#### *El mundo real es paralelo*

La principal limitación del modelo con estado es que los programas son secuenciales. En el mundo real, las entidades tienen estado y actúan en paralelo. La programación con estado secuencial no modela la ejecución paralela.

Algunas veces esta limitación es apropiada, e.g., cuando se escriben simuladores donde todos los eventos deben estar coordinados (pasando, en forma controlada, de

*Estado explícito*

un estado global a otro ). En otros casos, e.g., cuando hay interacción con el mundo real, la limitación es un obstáculo. Para eliminar la limitación, el modelo necesita tener tanto estado como concurrencia. Hemos visto una manera sencilla de lograr esto en el capítulo 5. Otra forma se presenta en el capítulo 8. Como se explica en la sección 4.8.6, la concurrencia en el modelo puede modelar el paralelismo del mundo real.

***El mundo real es distribuido***

El estado explícito es difícil de usar bien en un sistema distribuido. En el capítulo 11 (en CTM)se explica en profundidad esta limitación. Aquí sólo presentamos los puntos principales. En un sistema distribuido, el almacén está dividido en partes separadas. Dentro de una parte, el almacén se comporta eficientemente, como hemos visto. Entre las partes, la comunicación es muchos órdenes de magnitud más costosa. Unas partes se coordinan con las otras para mantener un nivel deseado de consistencia global. Para las celdas, esto puede ser muy costoso porque el contenido de las celdas puede cambiar en cualquier momento y en cualquier parte. El programador tiene que decidir tanto el nivel de consistencia como el algoritmo de coordinación utilizado. Esto hace que se vuelva delicado hacer programación distribuida con estado.

El modelo declarativo y su extensión al modelo concurrente por paso de mensajes del capítulo 5 son mucho más fáciles de usar. Como se explica en el capítulo 5, un sistema se puede descomponer en componentes independientes que se comunican por medio de mensajes. Esto encaja muy bien con el almacén dividido de un sistema distribuido. Cuando se programa un sistema distribuido, recomendamos utilizar, siempre que sea posible, el modelo de paso de mensajes para la coordinación de las partes. En el capítulo 11 (en CTM)se explica cómo programar un sistema distribuido y cuándo usar los diferentes modelos de computación en un contexto distribuido.

**6.9.2. Administración de la memoria y referencias externas**

Como se explicó en la sección 2.5, la recolección de basura es una técnica para la administración automática de la memoria que recupera memoria de todas las entidades del modelo de computación que ya no hacen parte de la computación. Esto no es suficientemente bueno para las entidades externas al modelo de computación. Tales entidades existen porque hay un mundo externo al modelo de computación, interactuando con él. ¿Cómo podemos hacer administración automática de la memoria para ellas? Hay dos casos:

- Desde el interior del modelo de computación, hay una referencia a una entidad por fuera de él. Llamamos tal referencia un recurso apuntador. Presentamos algunos ejemplos:

- Un descriptor de archivo, el cual apunta a una estructura de datos admi-

nistrada por el sistema operativo. Cuando el descriptor de archivo deja de ser referenciado, nos gustaría cerrar el archivo.

- Una manija para acceder a una base de datos externa. Cuando la manija deja de ser referenciada, nos gustaría cerrar la conexión a la base de datos.
  - Un apuntador a un bloque de memoria reservada a través de la interfaz Mozart C++. Cuando la memoria deja de ser referenciada, nos gustaría que fuera liberada.
- Desde el mundo exterior, existe una referencia al interior del modelo de computación. LLamamos tal referencia un tiquete. Los tiquetes se utilizan en la programación distribuida como un mecanismo para conectar procesos (ver capítulo 11 (en CTM)).

En el segundo caso, en general, no existe una manera segura de recuperar memoria. Segura quiere decir, no liberar memoria mientras existan referencias externas. El mundo externo es tan grande que el modelo de computación no puede saber si aún existe una referencia o no. Una solución pragmática consiste en añadir la entidad del lenguaje al conjunto raíz por un período limitado de tiempo. Esto se conoce como un mecanismo de contrato de tiempo. El período de tiempo puede renovarse cuando se accede a la entidad del lenguaje. Si el período de tiempo expira sin una renovación, suponemos que no hay más referencias externas. La aplicación tiene que ser diseñada para manejar el caso raro en que esta suposición es incorrecta.

En el primer caso, existe una solución sencilla basada en parametrizar el recolector de basura. Esta solución, denominada finalización, ofrece la posibilidad de realizar una acción definida por el usuario cuando una entidad del lenguaje se vuelve inalcanzable. Esto está implementado en el módulo del sistema `Finalize`. Primero explicamos cómo funciona el módulo y luego presentamos algunos ejemplos de cómo se usa.

### ***Finalización***

La finalización esta soportada por el módulo `Finalize`. El diseño de este módulo está inspirado en el concepto de guardián de [49]. `Finalize` cuenta con las dos operaciones siguientes:

- `{Finalize.register X P}` registra una referencia `X` y un procedimiento `P`. Cuando `X` no se puede alcanzar de otra manera (diferente que a través de la finalización), finalmente se ejecuta `{P X}` en su propio hilo. Durante esta ejecución, `X` se hace alcanzable de nuevo hasta que su referencia no se puede acceder más.
- `{Finalize.everyGC P}` registra un procedimiento `P` para ser invocado, al final, después de cada recolección de basura.

En ambas operaciones, no se puede confiar en cuán rápido será invocado el procedimiento `P` después de la recolección de basura. En principio, es posible que la invocación sólo pueda ser planificada después de varias recolecciones de basura, si

*Estado explícito*

el sistema tiene muchos hilos vivos y genera basura a una gran velocidad.

No existen límites en lo que el procedimiento `P` puede hacer. Esto es debido a que `P` no se ejecuta durante la recolección de basura, cuando las estructuras de datos del sistema pueden ser, temporalmente, inconsistentes, sino que es planificado para ejecución después de la recolección de basura. `P` aún puede referenciar `x` y él mismo invocar `Finalize`.

Un ejemplo interesante es la misma operación `everyGC`, la cual se define en términos de `register`:

```
proc {EveryGC P}
 proc {HACER _} {P} {Finalize.register HACER HACER} end
 in
 {Finalize.register HACER HACER}
 end
```

Aquí se crea un procedimiento `HACER` y lo registra utilizándose a sí mismo como su propio manejador. Cuando `EveryGC` termina, la referencia a `HACER` se pierde. Esto significa que `HACER` se invocará en la siguiente recolección de basura. Cuando esto pase, él invocará a `P` y se registrará a sí mismo de nuevo.

*Pereza y recursos externos*

Para lograr que la evaluación perezosa sea práctica para los recursos externos como los archivos, necesitamos utilizar la finalización para liberar los recursos externos cuando estos no se necesiten más. Por ejemplo, en la sección 4.5.5 definimos una función `LeerListaPerezosa` que lee perezosamente un archivo. Esta función cierra el archivo después de que ha sido leído por completo. Pero esto no es suficientemente bueno; aún si sólo se necesita una parte del archivo, éste también debe cerrarse. Podemos implementar esto con la finalización. Extendemos la definición de `LeerListaPerezosa` para cerrar el archivo cuando no se pueda acceder más a él:

```
fun {LeerListaPerezosa FN}
 {File.readOpen FN}
 fun lazy {LeerSiguiente}
 L T I in
 {File.readBlock I L T}
 if I==0 then T=nil {File.readClose} else T={LeerSiguiente}
 end
 L
 end
 in
 {Finalize.register F proc {$ F} {File.readClose} end}
 {LeerSiguiente}
 end
```

Esto requiere de una sola invocación a `Finalize`.

---

## 6.10. Ejercicios

1. *La importancia de las secuencias.* En la sección 6.1 se presenta una definición de estado. En este ejercicio, compare y contraste esa definición con la definición siguiente de dibujos animados, hecha por Scott McCloud [112] y adaptada para esta traducción:

**dibujos animados n. 1.** Ilustraciones y otras imágenes yuxtapuestas en una secuencia deliberada, con el objetivo de comunicar información y/o producir una respuesta estética en quien la mira.

...

*Sugerencias:* ¿Nos interesa toda la secuencia o solamente el resultado final? ¿La secuencia existe en el espacio o en el tiempo? ¿Es importante la transición entre elementos de la secuencia?

2. *Estado con celdas.* En la sección 6.1 se define la función `SumList`, la cual tiene un estado codificado como los valores sucesivos de los dos argumentos en las invocaciones recursivas. En este ejercicio, reescriba `SumList`, de manera que el estado ya no esté codificado en los argumentos, sino en celdas.

3. *Emulando el estado con la concurrencia.* Este ejercicio explora si la concurrencia se puede utilizar para obtener el estado explícito.

a) Primero, utilice la concurrencia para crear un contenedor actualizable. Creamos un hilo que utiliza un procedimiento recursivo para leer un flujo. El flujo tiene dos instrucciones posibles: `acceder(X)`, la cual liga `X` al contenido del contenedor actual, y `asignar(X)`, la cual asigna `X` como el contenido nuevo. Esto se hace así:

```
fun {CrearEstado Inic}
 proc {Ciclo S V}
 case S of acceder(X)|S2 then
 X=V {Ciclo S2 V}
 [] asignar(X)|S2 then
 {Ciclo S2 X}
 else skip end
 end S
in
 thread {Ciclo S Inic} end S
end
```

La invocación `S={CrearEstado 0}` crea un contenedor nuevo con contenido inicial 0. Utilizamos el contenedor colocando instrucciones en el flujo. Por ejemplo, considere la siguiente secuencia de tres instrucciones para el contenedor `S`:

```
declare S1 X Y in
S=acceder(X)|asignar(3)|acceder(Y)|S1
```

Esto liga `X` a 0 (el contenido inicial), coloca 3 en el contenedor, y luego liga `Y` a 3.

b) Ahora, reescriba `SumList` usando este contenedor para contar el número de invocaciones. ¿Este contenedor puede ser encapsulado, i.e., puede ser añadido sin cambiar los argumentos de `SumList`? ¿Por qué o por qué no? ¿Qué pasa cuando tratamos de añadir la función `ContadorSum` como en la sección 6.1.2?

*Estado explícito*

4. *Implementando puertos.* En el capítulo 5 se introdujo el concepto de puerto, el cual es un canal de comunicación sencillo. Los puertos tienen las operaciones `{NewPort S P}`, la cual devuelve un puerto `P` con flujo `S`, y `{Send P x}`, la cual envía un mensaje `x` por el puerto `P`. De acuerdo a estas operaciones, es claro que los puertos son un TAD con estado. En este ejercicio, implemente los puertos en términos de celdas, utilizando las técnicas presentadas en la sección 6.4.

5. *Estado explícito y seguridad.* En la sección 6.4 se presentaron cuatro formas de construir abstracciones de datos seguras. De acuerdo a esas abstracciones, parece que la habilidad de construir abstracciones seguras es una consecuencia de utilizar uno o más de los tres conceptos siguientes: valores de tipo procedimiento (los cuales proveen ocultamiento a través del alcance léxico), valores de tipo nombre (los cuales son inexpugnables e imposibles de adivinar), y los pedazos (los cuales proveen acceso selectivo). En particular, el estado explícito parece tener un rol con respecto a la seguridad. En este ejercicio, piense cuidadosamente sobre esta afirmación. ¿Es cierta? ¿Por qué sí o por qué no?

6. *Objetos declarativos e identidad.* En la sección 6.4.2 se muestra cómo construir un objeto declarativo, el cual combina valores y operaciones en una forma segura. Sin embargo, en la implementación presentada hace falta un aspecto de los objetos, a saber, la identidad. Es decir, un objeto debe conservar la misma identidad después de los cambios de estado. En este ejercicio, extienda los objetos declarativos de la sección 6.4.2 para que tengan una identidad.

7. *Habilidades revocables.* En la sección 6.4.5 se define el procedimiento `Revocable`, el cual toma una habilidad y utiliza estado explícito para crear dos cosas: una versión revocable de la habilidad y un revocador. Para el procedimiento `Revocable`, la habilidad está representada por un procedimiento de un argumento y el revocador como un procedimiento sin argumentos. En este ejercicio, escriba una versión de `Revocable` que sea un procedimiento de un argumento y donde el revocador también sea un procedimiento de un argumento. Esto permite que `Revocable` pueda ser usado recursivamente sobre todas las habilidades incluyendo los revocadores y él mismo. Por ejemplo, la capacidad de revocar una habilidad puede ser revocable.

8. *Abstracciones y administración de la memoria.* Considere el TAD con estado siguiente, el cual permite recolectar información en una lista. El TAD tiene tres operaciones. La invocación `C={NuevoRecolector}` crea un recolector nuevo `C`. La invocación `{Recolectar C x}` agrega `x` a la colección `C`. La invocación `L={FinRecolección C}` devuelve la lista final que contiene todos los ítems que se recolectaron en el orden en que fueron recolectados. Hay dos formas de implementar recolectores que vamos a comparar:

- `C` es una celda que contiene una pareja `H|T`, donde `H` es la cabeza de la lista recolectada y `T` es la cola sin ligar. `Recolectar` se implementa así:

```
proc {Recolectar C x}
 H T in
 {Exchange C H|(x|T) H|T}
 end
```

Implemente las operaciones `NuevoRecolector` y `FinRecolección` con esta representación.

- `C` es una pareja `H|T`, donde `H` es la cabeza de la lista recolectada y `T` es una celda que contiene su cola sin ligar. `Recolectar` se implementa así:

```
proc {Recolectar C X}
 T in
 {Exchange C.2 X|T T}
end
```

Implemente las operaciones `NuevoRecolector` y `FinRecolección` con esta representación.

- Comparemos las dos implementaciones con respecto a la administración de la memoria. Utilice la tabla de la sección 3.5.2 para calcular cuántas palabras de memoria se reservan para cada versión de `Recolectar`. ¿En cada versión, cuántas de esas palabras se vuelven inactivas inmediatamente? ¿Qué implicaciones tiene esto sobre la recolección de basura? ¿Cuál versión es mejor?

Este ejemplo se tomó del sistema Mozart. La recolección dentro del ciclo `for` se implementó originalmente con una versión, y fue finalmente reemplazada por la otra. (Note que ambas versiones funcionan bien en un contexto concurrente, i.e., si `Recolectar` se invoca desde múltiples hilos.)

9. *Invocación por nombre*. En la sección 6.4.4 se muestra cómo codificar la invocación por nombre en el modelo de computación con estado. En este ejercicio, considere el siguiente ejemplo tomado de [52]:

```
procedure swap(callbyname x,y:integer);
var t:integer;
begin
 t:=x; x:=y; y:=t
end;
var a:array [1..10] of integer;
var i:integer;
i:=1; a[1]:=2; a[2]:=1;
swap(i, a[i]);
writeln(a[1], a[2]);
```

Este ejemplo muestra un comportamiento curioso de la invocación por nombre. Ejecutar el ejemplo no intercambia `i` y `a[i]` como uno esperaría. Esto muestra una interacción indeseable entre el estado explícito y la evaluación retardada de un argumento.

- Explique este ejemplo utilizando su comprensión de la invocación por nombre.
- Codifique el ejemplo en el modelo de computación con estado. Utilice la codificación siguiente de `array[1..10]`:

```
A={MakeTuple array 10}
for J in 1..10 do A.J={NewCell 0} end
```

*Estado explícito*

Es decir, codifique el arreglo como una tupla de celdas.

- De nuevo, explique el comportamiento en términos de su codificación.

10. *Invocación por necesidad*. Con la invocación por nombre, el argumento se evalúa de nuevo cada vez que se necesita. Con la invocación por necesidad, el argumento se evalúa una vez a lo sumo.

- En este ejercicio, repita el ejemplo de `swap` del ejercicio anterior con invocación por necesidad en lugar de invocación por nombre. ¿Sigue sucediendo el comportamiento contraintuitivo? Si no, ¿pueden suceder problemas similares con la invocación por necesidad, modificando la definición de `swap`?

- En el ejemplo de invocación por necesidad de la sección 6.4.4, el cuerpo de `Cuad` siempre invoca a la función `A`. Esto está bien para `Cuad`, pues podemos ver por inspección que el resultado se necesita tres veces. ¿Pero, qué pasa si la necesidad no se puede determinar por inspección? No deseamos invocar `A` sin necesidad. Una posibilidad consiste en utilizar funciones perezosas. Modifique el código de la sección 6.4.4 de manera que utilice la pereza para invocar a `A` sólo cuando sea necesario, aún si la necesidad no puede ser determinada por inspección. `A` debe ser invocado a lo sumo una vez.

11. *Evaluando colecciones indexadas*. En la sección 6.5.1 se presentan cuatro tipos de colecciones indexadas, a saber, tuplas, registros, arreglos, y diccionarios, con compromisos de desempeño/expresividad diferentes. En este ejercicio, compare estos cuatro tipos en varios escenarios de utilización. Evalúe sus desempeños y utilidad relativos.

12. *Arreglos extensibles*. El arreglo extensible de la sección 6.5 sólo extiende el arreglo en una dirección. En este ejercicio, modifique el arreglo extensible de manera que extienda el arreglo en ambas direcciones.

13. *Diccionarios generalizados*. El diccionario predefinido de Mozart sólo funciona para llaves de tipo literal, i.e., números, átomos, o nombres. En este ejercicio, implemente un diccionario que pueda utilizar cualquier valor como llave. Una solución posible usa el hecho de que la operación `==` puede comparar cualquier par de valores. Utilizando esta operación, el diccionario podría almacenar las entradas como una lista de asociación, la cual es una lista de parejas `Llave#Valor`, y realizar búsqueda lineal simple.

14. *Ciclos y afirmaciones invariantes*. Utilice el método de afirmaciones invariantes para mostrar que las reglas de prueba para los ciclos `while` y `for`, presentadas en la sección 6.6.4, son correctas.

15. *La declaración break*. Un bloque es un conjunto de declaraciones con un punto de entrada y uno de salida bien definidos. Muchos lenguajes de programación imperativos modernos, tales como Java y C++, se basan en el concepto de bloque. Estos lenguajes permiten definir bloques anidados y proveen una operación para saltar inmediatamente de cualquier sitio dentro del bloque al punto de salida. Esta operación se denomina `break`. En este ejercicio, defina una construcción de bloque con una operación `break`, que se pueda invocar de la manera siguiente:

```
{Block proc {$ Break} <dec> end}
```

El comportamiento de esta construcción debe tener exactamente el mismo comportamiento que ejecutar `<dec>`, salvo que la ejecución de `{Break}` dentro de `<dec>` debe inmediatamente salir del bloque. Su solución debe funcionar correctamente con bloques anidados y con excepciones que se lanzan dentro de los bloques. Si `<dec>` crea hilos, entonces estos no deben verse afectados por la operación `break`. *Sugerencia:* utilice el mecanismo de manejo de excepciones.

16. *Aplicación de frecuencias de palabras.* En la sección 6.8.2 se presenta una versión del algoritmo de frecuencias de palabras que utiliza diccionarios con estado. Reescriba la aplicación de frecuencias de palabras de la sección 3.9.4 para utilizar esta versión con estado.

17. *Simulación de un “Mundo pequeño”.* La simulación “de boca en boca” de la sección 6.8.4 hace algunas suposiciones fuertes que simplifican el problema. Por ejemplo, la simulación supone que cada usuario puede escoger cualesquier tres usuarios al azar para consultarles sobre su desempeño. Esta es una suposición muy fuerte. El problema es que la elección comprende todos los usuarios. Esto le provee a cada usuario un cantidad de conocimiento potencialmente ilimitado. En la realidad, cada usuario tiene un conocimiento limitado: una red pequeña de conocidos que cambia, pero lentamente. Cada usuario consulta solamente miembros de su red de conocidos. Reescriba el programa de simulación para tener en cuenta esta suposición. Esto puede hacer que la convergencia sea mucho más lenta. Con esta suposición, la simulación se denomina una simulación de un “mundo pequeño” [181].

18. *Efectos de desempeño en la simulación de boca en boca.* La simulación de boca en boca de la sección 6.8.4 supone que el desempeño de cada sitio es constante. Una mejor manera de modelar el desempeño consiste en suponer que es constante hasta un número dado de usuarios, el cual es fijo para cada sitio. Más allá de ese umbral, el desempeño decae en proporción inversa al número de usuarios. Esto está basado en la premisa que para un número pequeño de usuarios, el desempeño de Internet es el cuello de botella, y para un gran número de usuarios, el cuello de botella es el desempeño del sitio.

19. (ejercicio avanzado) *Desarrollo orientado por las pruebas.* En la sección 6.7.1 se explica por qué el desarrollo incremental es una buena idea. La sección termina con una breve mención del desarrollo orientado por las pruebas, un enfoque más radical que también es incremental a su manera. En este ejercicio, explore el desarrollo orientado por las pruebas y compárela con el desarrollo incremental. Desarrolle una o más aplicaciones utilizando el desarrollo orientado por las pruebas y trate de lograr un enfoque balanceado que combine lo mejor de ambas técnicas de desarrollo.

Draft  
532

## 7

## Programación orientada a objetos

La fruta es tan conocida que no necesita ninguna descripción de sus características externas.

– Adaptación libre de la entrada “Apple,” en la Encyclopaedia Britannica, 11th edition

La programación orientada a objetos (POO) es una de las áreas más exitosas y dominantes en informática. Desde sus inicios en los años 1960, la POO ha invadido cada área de la informática, tanto en la investigación científica como en el desarrollo tecnológico. El primer lenguaje orientado a objetos fue Simula 67, desarrollado en 1967 como un descendiente de Algol 60 [121, 128, 139]. Sin embargo, la POO no se volvió popular industrialmente, hasta la aparición de C++ a principios de los años 1980 [166]. Otro paso importante fue Smalltalk-80, liberado en 1980 como el resultado de una investigación realizada en los años 1970 [56]. Tanto C++ como Smalltalk fueron influenciados directamente por Simula [87, 165]. Los lenguajes de programación más populares en la actualidad, Java y C++, son orientados a objetos [166, 168], ambos. Las ayudas más populares para el diseño, “independientes del lenguaje,” el lenguaje de modelamiento unificado (UML) y los patrones de diseño, suponen implícitamente que el lenguaje de programación subyacente es orientado a objetos [54, 147].

Con todo este impulso, uno podría pensar que la POO está bien comprendida (ver el epígrafe arriba). Sin embargo, esto está lejos de ser así. El propósito de este capítulo no es cubrir la POO en 100 páginas o menos. Esto es imposible. En su lugar, presentamos una introducción que enfatiza las áreas donde las otras presentaciones son débiles: la relación con otros modelos de computación, la semántica precisa, y las posibilidades de tipamiento dinámico. También presentamos la motivación para las decisiones de diseño hechas por la POO y los compromisos envueltos en esas decisiones.

### *Principios de la programación orientada a objetos*

El modelo de computación de la POO es el modelo con estado del capítulo 6. El primer principio de la POO es que los programas son colecciones de abstracciones de datos que interactúan entre ellas. En la sección 6.4 vimos una gran y desconcertante variedad de formas de construir abstracciones de datos. La programación orientada a objetos impone un orden dentro de esta variedad. La POO propone dos principios

para construir abstracciones de datos:

1. Las abstracciones de datos deben tener estado por defecto. El estado explícito es importante para la modularidad de los programas (ver sección 6.2). El estado explícito posibilita escribir programas como partes independientes que se pueden extender sin cambiar sus interfaces. El principio opuesto (i.e., declarativo por defecto) también es razonable, pues hace que el razonamiento sea más sencillo (ver sección 4.8.5) y es más natural para la programación distribuida (ver capítulo 11 (en CTM)).

Desde nuestro punto de vista, tanto las abstracciones declarativas como con estado deberían ser igualmente fáciles de usar.

2. El estilo objeto, por defecto, debería ser el estilo APD. El estilo objeto es importante porque fomenta el polimorfismo y la herencia. El polimorfismo se explicó en la sección 6.4. Con él, un programa puede repartir apropiadamente la responsabilidad entre sus partes. La herencia es un concepto nuevo que introducimos en este capítulo. Con él, se pueden construir abstracciones de forma incremental. Agregamos una abstracción lingüística, denominada clase, para soportar la herencia en el lenguaje.

Resumiendo, podemos caracterizar, aproximadamente, la programación orientada a objetos como la programación con abstracción de datos en forma de objetos, estado explícito, polimorfismo, y herencia.

### *Estructura del capítulo*

El capítulo consiste de las partes siguientes:

- *Herencia* (sección 7.1). Primero introducimos y motivamos el concepto de herencia de manera general y lo situamos con respecto a otros conceptos de estructuración de programas.
- *Un modelo de computación orientado a objetos* (secciones 7.2 y 7.3). Definimos un sistema de objetos sencillo que provee herencia simple y múltiple con ligadura estática y dinámica. El sistema objeto saca ventaja del tipamiento dinámico para combinar simplicidad y flexibilidad. Los mensajes y los atributos son de primera clase, las clases son valores, y se pueden programar alcances arbitrarios. Esto nos permite explorar mejor los límites de la POO y situar los lenguajes existentes dentro de ella. Presentamos la sintaxis del sistema objeto y la implementación de soporte para hacerlo más fácil de usar y más eficiente.
- *Programación con herencia* (sección 7.4). Explicamos las técnicas y principios básicos para utilizar la herencia en la construcción de programas orientados a objetos. El principio más importante se denomina la propiedad de substitución. Ilustramos los principios con ejemplos de programas realistas. Presentamos las referencias a la literatura sobre diseño orientado a objetos.
- *Relación con otros modelos de computación* (sección 7.5). Desde el punto de vista de múltiples modelos de computación, mostramos cómo y cuándo utilizar,

y no utilizar, la programación orientada a objetos. Relacionamos la POO con la programación basada en componentes, la programación basada en objetos, y la programación de alto orden. Presentamos otras técnicas de diseño que se posibilitan cuando la POO se usa con otros modelos. Explicamos los pros y los contras del muy mencionado principio que afirma que cada entidad del lenguaje debe ser un objeto. Este principio ha guiado el diseño de varios lenguajes orientados a objetos importantes, luego es importante entender lo que significa.

- *Implementación del sistema de objetos* (sección 7.6). Presentamos una semántica precisa y sencilla de nuestro sistema de objetos, implementándolo en términos del modelo de computación con estado. Como la implementación utiliza un modelo de computación con una semántica precisa, podemos considerarla como una definición semántica.
- *El lenguaje Java* (sección 7.7). Presentamos una visión general de la parte secuencial de Java, un lenguaje de programación orientado a objetos muy popular. Mostramos cómo los conceptos de Java encajan en el sistema de objetos del capítulo.
- *Objetos activos* (sección 7.8). Los objetos activos extienden los objetos puerto del capítulo 5 utilizando una clase para definir su comportamiento. Allí se combinan las capacidades de la programación orientada a objetos y la concurrencia por paso de mensajes. Mostramos cómo programar con objetos activos y los comparamos con la concurrencia declarativa.

### *Construcción de software orientado a objetos*

Para mayor información sobre las técnicas y principios de programación orientada a objetos, recomendamos el libro *Object-Oriented Software Construction* escrito por Bertrand Meyer [113]. Este libro es especialmente interesante por su discusión detallada de la herencia, incluyendo la herencia múltiple.

---

## 7.1. Herencia

La herencia está basada en la observación de que, frecuentemente, muchas abstracciones de datos tienen mucho en común. Miremos el ejemplo de los conjuntos. Existen muchas abstracciones diferentes que son “como conjuntos,” en el sentido que son colecciones a las cuales podemos agregarles o eliminarles elementos. Algunas veces deseamos que esas abstracciones se comporten como pilas, agregando y eliminando elementos en orden LIFO (last-in, first-out). Otras veces deseamos que esas abstracciones se comporten como colas, agregando y eliminando elementos en orden FIFO. Algunas veces el orden en que se agreguen o eliminen elementos no tiene importancia. Y así sucesivamente, con muchas otras posibilidades. Cada una de estas abstracciones comparte la propiedad fundamental de ser una colección de elementos. ¿Podemos implementarlas sin duplicar las partes comunes? Tener partes duplicadas no sólo hace que el programa sea más largo, sino que es una pesadilla

tanto para el programador como para quien mantiene el programa, pues cada vez que se modifica una copia, las otras también deben ser modificadas. Aún peor, a menudo las diferentes copias son ligeramente diferentes, lo cual hace que la relación entre todos los cambios no sea obvia.

Introducimos el concepto de herencia para reducir el problema de la duplicación de código y para clarificar las relaciones entre las abstracciones de datos. Se puede definir una abstracción de datos “heredando” de una o varias otras abstracciones de datos, i.e., teniendo esencialmente la misma funcionalidad que las otras, posiblemente con algunas extensiones y modificaciones. Sólo se tienen que especificar las diferencias entre la abstracción de datos y sus ancestros. A tal definición incremental de una abstracción de datos se le denomina una *clase*.

Una clase nueva se define por medio de una especie de transformación: se combinan una o más clases existentes con una descripción de las extensiones y modificaciones para producir la clase nueva. Los lenguajes orientados a objetos soportan esta transformación añadiendo las clases como una abstracción lingüística. La transformación se puede ver como una manipulación sintáctica, donde la sintaxis de la clase nueva se deriva a partir de las clases originales (ver sección 7.3). En el sistema de objetos de este capítulo, la transformación también tiene sentido semánticamente (ver sección 7.6.4). Como las clases son valores, la transformación se puede ver como una función que recibe como entrada valores de tipo clase y devuelve como salida la clase nueva.

Mientras la herencia parece muy prometedora, la experiencia muestra que ella debe ser usada como mucho cuidado. Primero que todo, la transformación se debe definir con un conocimiento profundo de las clases ancestro, pues ellas pueden romper fácilmente un invariante de clase. Otro problema es que el uso de la herencia añade una interfaz adicional a un componente. Es decir, la capacidad de extender una clase se puede ver como una manera adicional de interactuar con esa clase. Esta interfaz tiene que manetenerse a lo largo de la vida del componente. Por esta razón, toda clase se define, por defecto, como final, i.e., se prohíbe que otras clases puedan heredar de ella. Crear una clase (o parte de una clase) de la que se pueda heredar requiere la acción explícita del programador.

La herencia incrementa las posibilidades de factorizar una aplicación, i.e., de asegurarse que las partes comunes existan una sola vez, pero el precio es dispersar la implementación de una abstracción en muchas partes del programa. La implementación no existe en un sitio; todas las abstracciones de las que se hereda tienen que considerarse juntas. Esto dificulta entender la abstracción, y, paradójicamente, puede dificultar su mantenimiento. Aún peor, una abstracción puede heredar de una clase que existe sólo como código objeto, sin ningún acceso a su código fuente. La lección es que la herencia debe utilizarse en “pequeñas cantidades.”

En lugar de usar la herencia, una alternativa consiste en utilizar la programación basada en componentes, i.e., utilizar los componentes directamente y componerlos. La idea es definir un componente que encapsula a otro componente y provee una funcionalidad modificada. Existe un compromiso entre la herencia y la composición de componentes: la herencia es más flexible pero puede romper un invariante de

clase, mientras que la composición de componentes es menos flexible pero no puede romper un invariante de componente. Este compromiso debe considerarse cuidadosamente cuando se deba extender una abstracción.

En un principio, se creyó que la herencia resolvería el problema de la reutilización del *software*. Por ejemplo, la herencia debería facilitar la construcción de bibliotecas que se puedan distribuir a terceras partes, para ser usadas en otras aplicaciones. Esto ha tenido algún éxito a través de los marcos de *software*. Un marco de *software* es un sistema de *software* que se ha hecho de manera genérica. Instanciar el marco significa proporcionar valores reales para los parámetros genéricos. Como veremos en este capítulo, esto se puede lograr con la herencia utilizando clases genéricas o clases abstractas.

---

## 7.2. Clases como abstracciones de datos completas

El corazón del concepto de objeto es el acceso controlado a datos encapsulados. El comportamiento de un objeto se especifica por medio de una clase. En el caso más general, una clase es una definición incremental de una abstracción de datos, que define la abstracción como una modificación de otras. Hay un conjunto rico de conceptos para definir clases. Clasificamos estos conceptos en dos conjuntos de acuerdo a cuáles permiten que la clase defina una abstracción de datos completamente y cuáles permiten que lo haga incrementalmente:

- *Abstracción de datos completa*. Estos son todos los conceptos que permiten que una clase en sí misma defina una abstracción de datos. Hay dos conjuntos de conceptos:
  - La definición de los diversos elementos que conforman una clase (ver secciones 7.2.3 and 7.2.4), a saber, los métodos, los atributos, y las propiedades. Los atributos se pueden inicializar de varias formas, por objeto o por clase (ver sección 7.2.5).
  - Los que aprovechan el tipamiento dinámico. Esto nos lleva a mensajes de primera clase (ver sección 7.2.6) y atributos de primera clase (ver sección 7.2.7). Con esto se logran formas poderosas de polimorfismo que son difíciles o imposibles de conseguir en lenguajes estáticamente tipados. Este incremento de libertad viene con un incremento de la responsabilidad del programador para usarla correctamente.
- *Abstracción de datos incremental*. Estos son todos los conceptos relacionados con la herencia, i.e., los que definen cómo se relaciona una clase con las clases existentes. Estos se presentan en la sección 7.3.

Para explicar qué son las clases empezamos presentando un ejemplo que muestra cómo definir una clase y un objeto. En la sección 7.2.1 se presenta el ejemplo utilizando la sintaxis de clase y luego, en la sección 7.2.2, se presenta su semántica precisa definiéndola en el modelo con estado.

```
class Contador
 attr val
 meth inic(Valor)
 val:=Valor
 end
 meth browse
 {Browse @val}
 end
 meth inc(Valor)
 val:=@val+Valor
 end
end
```

Figura 7.1: Ejemplo de una clase Contador (con la sintaxis **class**).

### 7.2.1. Un ejemplo

Para mostrar cómo funcionan las clases y los objetos en el sistema de objetos, definimos un ejemplo de clase y la usamos para crear un objeto. Suponemos que el lenguaje tiene un constructor sintáctico nuevo, la declaración **class**. Suponemos también que las clases son valores de primera clase en el lenguaje. Esto nos permite usar una declaración **class** como una declaración en sí o como una expresión, de forma similar a la declaración **proc**. Más adelante veremos cómo definir clases directamente en el modelo con estado. Eso quiere decir que podemos considerar la declaración **class** como una abstracción lingüística.

En la figura 7.1 se define una clase referenciada por la variable **Contador**. Esta clase tiene un atributo, **val**, que contiene el valor actual de un contador, y tres métodos, **inic**, **browse**, e **inc**, para inicializar, desplegar, e incrementar el contador. El atributo se asigna con el operador **:=** y se accede a su contenido con el operador **@**. Esto es bastante similar a la forma como lo hacen otros lenguajes, módulo una sintaxis diferente. ¡Pero las apariencias engañan!

La declaración de la figura 7.1 se ejecuta realmente en tiempo de ejecución, i.e., es una declaración que crea un valor de tipo clase y lo liga a la variable **Contador**. Si se reemplaza “**Contador**” por “**\$**,” entonces la declaración se puede usar como una expresión. Si se coloca esta declaración al principio del programa, se declarará la clase antes de ejecutar el resto, lo cual es un comportamiento familiar. Pero esta no es la única posibilidad. La declaración puede colocarse en cualquier sitio donde pueda ir una declaración. Por ejemplo, si se coloca la declaración dentro de un procedimiento, se creará una clase nueva, distinta, cada vez que se invoca el procedimiento. Más adelante utilizaremos esta posibilidad para crear clases parametrizadas.

Creemos un objeto de la clase **Contador** y realicemos algunas operaciones con él:

```

local
 proc {Inic M S}
 inic(Valor)=M in (S.val):=Valor
 end
 proc {Browse2 M S}
 {Browse @(S.val)}
 end
 proc {Inc M S}
 inc(Valor)=M in (S.val):=@(S.val)+Valor
 end
in
 Contador=c(atrbs:[val]
 métodos:m(inic:Inic browse:Browse2 inc:Inc))
end

```

**Figura 7.2:** Definición de la clase Contador (sin soporte sintáctico).

```

C={New Contador inic(0)}
{C inc(6)} {C inc(6)}
{C browse}

```

Aquí se crea el objeto contador C con 0 como valor inicial, luego se incrementa dos veces en 6, y por último se despliega el valor del contador. La declaración {C inc(6)} se denomina una aplicación de un objeto. El mensaje inc(6) es utilizado por el objeto para invocar el método correspondiente. Ahora ensaye lo siguiente:

```

local x in {C inc(X)} x=5 end
{C browse}

```

¡Esto no despliega absolutamente nada! La razón es que la aplicación del objeto

```
{C inc(X)}
```

se bloquea dentro del método inc. ¿Puede ver exactamente dónde? Ahora ensaye la siguiente variante:

```

declare S in
local x in thread {C inc(X)} S=listo end x=5 end
{Wait S} {C browse}

```

Ahora las cosas funcionan como se esperaba. Vemos que la ejecución de flujo de datos conserva su comportamiento familiar cuando se utiliza con objetos.

### 7.2.2. Semántica del ejemplo

Antes de entrar a describir las capacidades adicionales de las clases, presentamos la semántica del ejemplo Contador. Esta es una aplicación directa de programación de alto orden con estado explícito. La semántica que presentamos aquí es ligeramente simplificada; deja por fuera las capacidades de **class** que no se utilizan en el ejemplo (tales como herencia y **self**). En la sección 7.6 se presenta la semántica completa.

```
fun {New Clase Inic}
 Fs={Map Clase.atrbs fun {$ x} x#{NewCell _} end}
 S={List.toRecord estado Fs}
 proc {Obj M}
 {Clase.métodos.{Label M} M S}
 end
in
 {Obj Inic}
 Obj
end
```

**Figura 7.3:** Creación de un objeto Contador.

En la figura 7.2 se muestra lo que hace la figura 7.1 presentando la definición de la clase Contador en el modelo con estado sin la sintaxis **class**. De acuerdo a esta definición podemos ver que una clase es sencillamente un registro que contiene un conjunto de nombres de atributos y un conjunto de métodos. Un nombre de atributo es un literal. Un método es un procedimiento que tiene dos argumentos, el mensaje y el estado del objeto. En cada método, la asignación de un atributo (“**val:=**”) se realiza con una asignación de celdas y el acceso al contenido de un atributo (“**@val**”) se realiza con un acceso de celdas.

En la figura 7.3 se define la función **New** la cual se utiliza para crear objetos a partir de clases. Esta función crea el estado del objeto, define un procedimiento de un argumento **Obj** que es de hecho el objeto, e inicializa el objeto antes de devolverlo. El estado del objeto es un registro **S** que contiene una celda por cada atributo. El estado del objeto queda oculto dentro de **Obj** gracias al alcance léxico.

#### 7.2.3. Definición de clases y objetos

Una clase es una estructura de datos que define un estado interno de un objeto (atributos), su comportamiento (métodos), las clases de las cuales hereda, y otras propiedades y operaciones que veremos más adelante. De manera más general, una clase es una estructura de datos que define una abstracción de datos y entrega su implementación parcial o total. En la tabla 7.1 se presenta la sintaxis de las clases.

Puede haber cualquier número de objetos de una clase dada. A cada uno de ellos se le llama una instancia de la clase. Estos objetos tienen identidades diferentes y pueden tener valores diferentes en su estado interno. Por lo demás, todos los objetos de una clase dada se comportan de acuerdo a la definición de la clase. Un objeto se crea con la operación **New**:

```
MiObj={New MiClase Inic}
```

Esto crea un objeto nuevo **Miobj** de la clase **MiClase** e invoca el objeto con el mensaje **Inic**. Este mensaje se utiliza para inicializar el objeto. El objeto **MiObj** se invoca con la sintaxis **{MiObj M}**. El objeto se comporta como un procedimiento

|                                                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{declaración} \rangle ::= \text{class } \langle \text{variable} \rangle \{ \langle \text{descriptorClase} \rangle \}$                                                                    |
| $\quad \{ \text{meth } \langle \text{cabezaMétodo} \rangle [ \text{`:=`} \langle \text{variable} \rangle ]$                                                                                            |
| $\quad \quad ( \langle \text{expresiónEn} \rangle   \langle \text{declaraciónEn} \rangle ) \text{ end }$                                                                                               |
| $\quad \text{end}$                                                                                                                                                                                     |
| $\quad   \text{lock } [ \langle \text{expresión} \rangle \text{ then } ] \langle \text{declaraciónEn} \rangle \text{ end }$                                                                            |
| $\quad   \langle \text{expresión} \rangle \text{ `:=`} \langle \text{expresión} \rangle$                                                                                                               |
| $\quad   \langle \text{expresión} \rangle \text{ `,' } \langle \text{expresión} \rangle$                                                                                                               |
| $\quad   \dots$                                                                                                                                                                                        |
| $\langle \text{expresión} \rangle ::= \text{class } \text{'$'} \{ \langle \text{descriptorClase} \rangle \}$                                                                                           |
| $\quad \{ \text{meth } \langle \text{cabezaMétodo} \rangle [ \text{`:=`} \langle \text{variable} \rangle ]$                                                                                            |
| $\quad \quad ( \langle \text{expresiónEn} \rangle   \langle \text{declaraciónEn} \rangle ) \text{ end }$                                                                                               |
| $\quad \text{end}$                                                                                                                                                                                     |
| $\quad   \text{lock } [ \langle \text{expresión} \rangle \text{ then } ] \langle \text{expresiónEn} \rangle \text{ end }$                                                                              |
| $\quad   \langle \text{expresión} \rangle \text{ `:=`} \langle \text{expresión} \rangle$                                                                                                               |
| $\quad   \langle \text{expresión} \rangle \text{ `,' } \langle \text{expresión} \rangle$                                                                                                               |
| $\quad   \text{'@'} \langle \text{expresión} \rangle$                                                                                                                                                  |
| $\quad   \text{self}$                                                                                                                                                                                  |
| $\quad   \dots$                                                                                                                                                                                        |
| $\langle \text{descriptorClase} \rangle ::= \text{from } \{ \langle \text{expresión} \rangle \} +   \text{prop } \{ \langle \text{expresión} \rangle \} +$                                             |
| $\quad   \text{attr } \{ \langle \text{inicAtrb} \rangle \} +$                                                                                                                                         |
| $\langle \text{inicAtrb} \rangle ::= ( [ \text{`!`} \text{`} ] \langle \text{variable} \rangle   \langle \text{átomo} \rangle   \text{unit}   \text{true}   \text{false} )$                            |
| $\quad [ \text{`::`} \text{`} \langle \text{expresión} \rangle ]$                                                                                                                                      |
| $\langle \text{cabezaMétodo} \rangle ::= ( [ \text{`!`} \text{`} ] \langle \text{variable} \rangle   \langle \text{átomo} \rangle   \text{unit}   \text{true}   \text{false} )$                        |
| $\quad [ \text{`(`} \text{`} \{ \langle \text{argMétodo} \rangle \} [ \text{`...`} \text{`} ] \text{`)`} ]$                                                                                            |
| $\quad [ \text{`=:`} \langle \text{variable} \rangle ]$                                                                                                                                                |
| $\langle \text{argMétodo} \rangle ::= [ \langle \text{feature} \rangle \text{ `:=`} ] ( \langle \text{variable} \rangle   \text{`_`}   \text{'$' } ) [ \text{`<=`} \langle \text{expresión} \rangle ]$ |

Tabla 7.1: Sintaxis de clase.

de un argumento, donde el argumento es el mensaje. Los mensajes `Inic` y `M` se representan con registros. Una invocación a un objeto es similar a una invocación a un procedimiento; la invocación culmina cuando el método se haya ejecutado completamente.

#### 7.2.4. Miembros de una clase

Una clase define las partes que constituirán cada uno de sus objetos. En terminología orientada objeto, estas partes se denominan frecuentemente miembros. Existen tres tipos de miembros:

- *Atributos* (declarados con la palabra reservada “`attr`”). Un atributo es una celda que contiene parte del estado de la instancia. En terminología orientada objeto, un

atributo se suele denominar una variable de instancia. El atributo puede contener cualquier entidad del lenguaje y sólo es visible en la definición de la clase y en todas las clases que heredan de ella. Cada instancia tiene un conjunto separado de atributos. La instancia puede actualizar un atributo con las operaciones siguientes:

- Una declaración de asignación:  $\langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$ . Esta declaración asigna el resultado de evaluar  $\langle \text{expr} \rangle_2$  al atributo cuyo nombre se obtiene evaluando  $\langle \text{expr} \rangle_1$ .
- Una operación de acceso:  $@\langle \text{expr} \rangle$ . Esta operación da acceso al atributo cuyo nombre se obtiene al evaluar  $\langle \text{expr} \rangle$ . La operación de acceso se puede usar en cualquier expresión que esté, léxicamente hablando, dentro de la definición de la clase. En particular, puede ser utilizada dentro de los procedimientos que están definidos dentro de la clase.
- Una operación de intercambio. Si la asignación  $\langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$  se utiliza como una expresión, entonces tiene el efecto de un intercambio. Por ejemplo, considere la declaración  $\langle \text{expr} \rangle_3 = \langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$ . Primero se evalúan las tres expresiones. Luego, se unifica  $\langle \text{expr} \rangle_3$  con el contenido del atributo  $\langle \text{expr} \rangle_1$  y, atómicamente, se actualiza su contenido con el resultado de  $\langle \text{expr} \rangle_2$ .
- *Métodos* (declarados con la palabra reservada “**meth**”). Un método es una especie de procedimiento que se invoca en el contexto de un objeto particular y que tiene acceso a los atributos del objeto. El método consta de una cabeza y un cuerpo. La cabeza consta de una etiqueta, la cual debe ser un átomo o un nombre, y de un conjunto de argumentos. Los argumentos deben ser variables diferentes; de otra manera sería un error de sintaxis. Para una mayor expresividad, las cabezas de los métodos son similares a los patrones y los mensajes son similares a los registros. En la sección 7.2.6 se explican estas posibilidades.
- *Propiedades* (declarados con la palabra reservada “**prop**”). Una propiedad modifica cómo se comporta un objeto. Por ejemplo:
  - La propiedad **locking** crea un candado nuevo con cada instancia de objeto. Se puede acceder al candado dentro de la clase con la construcción sintáctica **lock** . . . **end**. Esta propiedad se explica en el capítulo 8.
  - La propiedad **final** hace que la clase sea una clase final, i.e., una clase que no se puede extender con herencia. Una buena práctica consiste en hacer que cada clase sea **final** salvo si se diseña, específicamente, para heredar.

Las etiquetas de los atributos y de los métodos son literales. Si se definen utilizando la sintaxis de átomo, entonces son átomos. Si se definen con la sintaxis de identificadores (e.g., en mayúscula), entonces el sistema creará nombres nuevos para ellos, cuyo alcance es la definición de la clase. Utilizar nombres permite un control de granularidad fina sobre la seguridad de los objetos, como lo veremos. En la sección 7.2.5 se muestra cómo inicializar los atributos.

Además de estos tres tipos de miembros, en la sección 7.3 se muestra cómo una clase puede heredar miembros de otras clases.

### 7.2.5. Inicialización de atributos

Los atributos se pueden inicializar de dos maneras: por instancia o por clase.

- *Por instancia.* Un atributo puede tener un valor inicial diferente para cada instancia. Esto se logra no inicializándolos en la definición de clase. Por ejemplo:

```
class UnApart
 attr nombreCalle
 meth inic(x) @nombreCalle=x end
end
Apt1={New UnApart inic(pasoancho)}
Apt2={New UnApart inic(calleQuinta)}
```

Cada instancia, incluyendo Apt1 y Apt2, referencia inicialmente una variable no-ligada diferente. Cada variable se liga a un valor diferente.

- *Por clase.* Un atributo puede tener un valor que sea el mismo para todas las instancias de la clase. Esto se hace, inicializándolo con ":" en la definición de clase. Por ejemplo:

```
class ApartQuinta
 attr
 nombreCalle:calleQuinta
 numeroCalle:100
 colorPared:_
 superficiePiso:madera
 meth inic skip end
end
Apt3={New ApartQuinta inic}
Apt4={New ApartQuinta inic}
```

Todas las instancias, incluyendo Apt3 y Apt4, tienen los mismos valores iniciales para los cuatro atributos. Esto incluye colorPared, aunque el valor inicial sea una variable no-ligada. Este valor se puede establecer por cualquiera de las instancias, ligando el atributo, i.e., @colorPared=blanco. Entonces, todas las instancias verán este valor. Tenga cuidado de no confundir las dos operaciones @colorPared=blanco y @colorPared:=blanco.

- *Por marca.* Esta es otra manera de utilizar la inicialización por clase. Una marca es un conjunto de clases relacionadas de alguna manera, pero no por herencia. Un atributo puede tener un valor que sea el mismo para todos los miembros de una marca, inicializándolo con la misma variable para todos los miembros. Por ejemplo<sup>1</sup>:

```
L=linux
class RedHat attr tiposo:L end
class SuSE attr tiposo:L end
class Debian attr tiposo:L end
```

Cada instancia de cada clase será inicializada con el mismo valor.

---

1. Nuestras disculpas a todas las distribuciones de Linux omitidas.

### 7.2.6. Mensajes de primera clase

El principio es sencillo: los mensajes son registros y las cabezas de los métodos son patrones que reconocen un registro. Como consecuencia de ello, las siguientes son las posibilidades que existen en cuanto a invocación de objetos y definición de métodos. En la invocación de objeto {Obj M}, los siguientes casos son posibles:

1. *Registro estático como mensaje.* En el caso más sencillo, M es un registro que se conoce en tiempo de compilación, e.g., como en la invocación {Contador inc(X)}.
2. *Registro dinámico como mensaje.* Es posible invocar {Obj M} donde M es una variable que referencia un registro que se calcula en tiempo de ejecución. Como el tipamiento es dinámico, es posible crear nuevos tipos de registros en tiempo de ejecución (e.g., con Adjoin o List.toRecord).

En la definición de un método, los siguientes casos son posibles:

1. *Lista fija de argumentos.* La cabeza del método es un patrón que consiste de una etiqueta seguida de una serie de argumentos en paréntesis. Por ejemplo:

```
meth foo(a:A b:B c:C)
 % Cuerpo del método
end
```

La cabeza del método foo(a:A b:B c:C) es un patrón que debe reconocer exactamente el mensaje, i.e., la etiqueta foo y la aridad [a b c] deben coincidir. Los campos (a, b, y c) pueden estar en cualquier orden. Una clase solamente puede tener una definición de método con una misma etiqueta; en caso contrario hay un error de sintaxis.

2. *Lista flexible de argumentos.* La cabeza del método es la misma que en el caso de la lista fija de argumentos salvo que termina en “...”. Por ejemplo:

```
meth foo(a:A b:B c:C ...)
 % Cuerpo del método
end
```

Los puntos suspensivos “...” en la cabeza del método significa que se acepta cualquier mensaje que tenga como mínimo los argumentos listados. Esto significa lo mismo que “...” en los patrones, e.g., en una declaración **case**. La etiqueta dada debe coincidir con la del mensaje y la aridad dada debe ser un subconjunto de la aridad del mensaje.

3. *Referencia variable a la cabeza del método.* Toda la cabeza del método se referencia por una variable. Esto es particularmente útil con las listas flexibles de argumentos, pero también se puede utilizar con una lista fija de argumentos. Por ejemplo:

```
meth foo(a:A b:B c:C ...)=M
 % Cuerpo del método
end
```

La variable M referencia el mensaje completo como un registro. El alcance de M es el cuerpo del método.

4. *Argumento opcional.* Se define un valor por defecto para un argumento. Este valor se usa sólo si el argumento no viene en el mensaje. Por ejemplo:

```
meth foo(a:A b:B<=v)
 % Cuerpo del método
end
```

El texto “`<=v`” en la cabeza del método significa que el campo `b` es opcional en la invocación al objeto. Es decir, el método se puede invocar con o sin ese campo. Con el campo, un ejemplo de invocación es `foo(a:1 b:2)`, lo cual ignora la expresión `v`. Sin el campo, un ejemplo de invocación es `foo(a:1)`, en cuyo caso el mensaje real que se recibe es `foo(a:1 b:v)`.

5. *Etiqueta privada de método.* Dijimos que las etiquetas de los métodos pueden ser nombres. Esto se denota utilizando un identificador de variable:

```
meth A(bar:X)
 % Cuerpo del método
end
```

El método `A` se liga a un nombre fresco cuando se define la clase. Inicialmente, `A` sólo es visible en el alcance de la definición de la clase. Si tiene que ser usada en otra parte del programa, debe ser pasada explícitamente.

6. *Etiqueta dinámica de método.* Es posible calcular una etiqueta de método en tiempo de ejecución, utilizando un identificador de variable marcado con el símbolo de escape “`!`”. Esto es posible gracias a que las clases se crean en tiempo de ejecución. La etiqueta del método debe ser conocida en el momento en que la definición de clase sea ejecutada. Por ejemplo:

```
meth !A(bar:X)
 % Cuerpo del método
end
```

hace que la etiqueta del método sea cualquier cosa a lo que la variable `A` haya sido ligada. La variable debe ser ligada a un átomo o a un nombre. Utilizando nombres, esta técnica sirve para crear métodos seguros (ver sección 7.3.3).

7. *El método otherwise.* El método cuya cabeza está etiquetada con `otherwise`, es un método que acepta cualquier mensaje para el cual no exista ningún otro método. Por ejemplo:

```
meth otherwise(M)
 % Cuerpo del método
end
```

Una clase sólo puede tener un método cuya cabeza sea `otherwise`; sino, existe un error de sintaxis. Este método sólo debe tener un argumento; de otra manera, se produce el error “`arity mismatch`” en tiempo de ejecución.

Si este método existe, entonces el objeto acepta cualquier mensaje. Si no hay ningún otro método definido para el mensaje, entonces el método `otherwise(M)` es invocado con el mensaje completo `M` como un registro. Este mecanismo permite la implementación de la delegación, una alternativa para la herencia que se explica en la sección 7.3.4. Este mecanismo también permite crear empaquetadores alrededor de las invocaciones de métodos.

Todas estas posibilidades están cubiertas por la sintaxis de la tabla 7.1. En general, para la invocación {Obj M}, el compilador trata de determinar estáticamente cuales son el objeto Obj y el método M. Si puede hacerlo, entonces compila a una instrucción muy rápida y especializada. Si no puede hacerlo, entonces compila a un instrucción general de invocación de un objeto. La instrucción general utiliza el ocultamiento. La primera invocación es lenta, porque busca el método y oculta el resultado. Las invocaciones subsiguientes buscan el método en la memoria oculta y son casi tan rápidas como las invocaciones especializadas.

#### 7.2.7. Atributos de primera clase

Los nombres de los atributos se pueden calcular en tiempo de ejecución. Por ejemplo, se pueden escribir métodos para acceder y asignar cualquiera de los atributos:

```
class Inspector
 meth acceder(A ?X)
 X=@A
 end
 meth asignar(A X)
 A:=X
 end
end
```

El método `acceder` puede acceder al contenido de cualquier atributo y el método `asignar` puede asignar cualquier atributo. Cualquier clase que tenga estos métodos abrirá sus atributos para uso público. Esta capacidad es peligrosa en la programación, pero puede ser muy útil para la depuración.

#### 7.2.8. Técnicas de programación

El concepto de clase que hemos introducido hasta el momento presenta una sintaxis conveniente para definir abstracciones de datos con estado encapsulado y múltiples operaciones. La declaración `class` define un valor de tipo clase, el cual puede ser instanciado para producir objetos. Además de tener una sintaxis conveniente, los valores de tipo clase como se definieron aquí conservan todas las ventajas de los valores de tipo procedimiento. Todas las técnicas de programación con procedimientos se pueden aplicar con las clases. Las clases pueden tener referencias externas de la misma manera que los valores de tipo procedimiento. Las clases son compositivas, pues se pueden anidar dentro de otras clases. Las clases son compatibles con los valores de tipo procedimiento pues se pueden anidar dentro de procedimientos y viceversa. Las clases no son tan flexibles en todos los lenguajes de programación orientados a objetos; normalmente se imponen algunas limitaciones, tal como se explica en la sección 7.5.

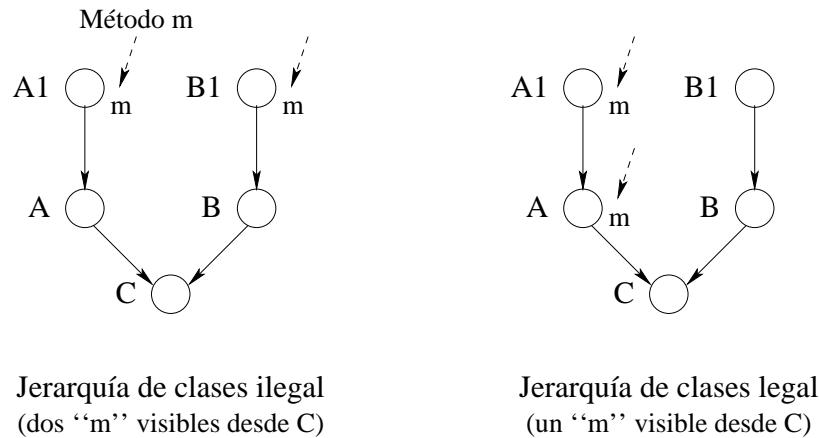


Figura 7.4: Jeraquías de clases ilegal y legal.

### 7.3. Clases como abstracciones de datos incrementales

Tal como se explicó antes, la característica principal que la POO añade a la programación basada en componentes es la herencia. La POO permite definir una clase de manera incremental, por medio de la extensión de las clases existentes. Nuestro modelo incluye tres conjuntos de conceptos para la herencia:

- El primero es el grafo de herencia (ver sección 7.3.1), el cual define cuáles clases preexistentes se extenderán. Nuestro modelo permite tanto herencia sencilla como herencia múltiple.
- El segundo es el control de acceso a los métodos (ver sección 7.3.2), el cual define cómo se accede a los métodos en particular, tanto en la clase nueva como en las clases preexistentes. Esto se logra con ligaduras estática y dinámica y con el concepto de `self`.
- El tercero es el control de la encapsulación (ver sección 7.3.3), el cual define qué parte de un programa puede ver los atributos y métodos de una clase. Definimos los alcances más populares para los miembros de los objetos y mostramos cómo se pueden programar otros alcances.

Concluimos utilizando el modelo para expresar otros conceptos relacionados, tales como reenvío, delegación, y reflexión (ver secciones 7.3.4 y 7.3.5).

#### 7.3.1. Grafo de herencia

La herencia es una forma de construir clases nuevas a partir de clases existentes. La herencia define qué atributos y métodos están disponibles en la clase nueva. Restringimos nuestra discusión sobre la herencia a los métodos. Las mismas reglas

se aplican a los atributos. Los métodos disponibles en una clase C se definen a través de una relación de precedencia sobre los métodos que aparecen en la jerarquía de clases. Denominamos a esta relación, relación de anulación<sup>2</sup>:

- Un método en la clase C anula cualquier método con la misma etiqueta en todas las superclases de C.

Las clases pueden heredar de una o más clases, las cuales aparecen después de la palabra reservada **from** en la declaración de la clase. Se dice que una clase que hereda exactamente de una clase utiliza herencia sencilla (también llamada algunas veces herencia simple). Heredar de más de una clase se denomina herencia múltiple. Una clase B es una superclase de la clase A si

- B aparece en la parte **from** de la declaración de A, o
- B es una superclase de una clase que aparece en la parte **from** de la declaración de A.

Una jerarquía de clases con la relación de superclases se puede ver como un grafo dirigido cuya raíz es la clase actual. Las aristas son dirigidas hacia las subclases. Existen dos requerimientos para que la herencia sea legal. Primero, la relación de herencia es dirigida y acíclica. Por lo tanto, lo siguiente no es permitido:

```
class A from B ... end
class B from A ... end
```

Segundo, sin tener en cuenta todos los métodos anulados, cada método restante debe tener una etiqueta única y debe estar definido en una sola clase en la jerarquía. Por lo tanto, la clase C en el ejemplo siguiente es ilegal pues los dos métodos etiquetados m permanecen allí:

```
class A1 meth m(...) ... end end
class B1 meth m(...) ... end end
class A from A1 end
class B from B1 end
class C from A B end
```

En la figura 7.4 se muestra esta jerarquía y una ligeramente diferente que si es legal. La clase C definida a continuación también es ilegal, pues los dos métodos etiquetados m están disponibles en C:

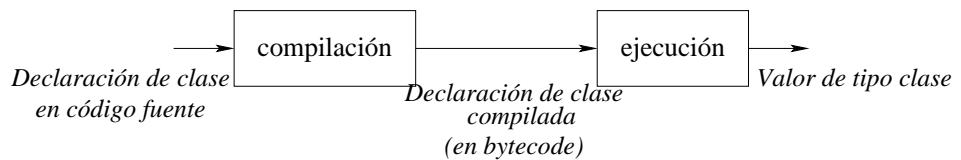
```
class A meth m(...) ... end end
class B meth m(...) ... end end
class C from A B end
```

### *Todo se hace en tiempo de ejecución*

Si se compila en Mozart un programa que contiene la declaración de la clase C, entonces el sistema no se quejará. Solamente, cuando el programa execute

---

2. Nota del traductor: traducido del inglés *overriding relation*.



**Figura 7.5:** Una declaración de clase es ejecutable.

la declaración, el sistema lanzará una excepción. Si el programa no ejecuta la declaración, entonces no se lanza ninguna excepción. Por ejemplo, un programa que contiene el código fuente siguiente:

```

fun {ClaseExtraña}
 class A meth foo(X) X=a end end
 class B meth foo(X) X=b end end
 class C from A B end
in C end

```

se puede compilar y ejecutar exitosamente. Su ejecución tiene el efecto de definir la función `ClaseExtraña`. Ya será solamente, durante la invocación `{ClaseExtraña}` que se lanzará una excepción. Esta “detección tardía de errores” no es solamente una propiedad de las declaraciones de clases, sino una propiedad general del sistema Mozart, consecuencia de la naturaleza dinámica del lenguaje. A saber, no hay distinción entre tiempo de compilación y tiempo de ejecución. El sistema de objetos comparte esta naturaleza dinámica. Por ejemplo, se pueden definir clases cuyas etiquetas de métodos se calculen en tiempo de ejecución (ver sección 7.2.6).

El sistema Mozart vuelve borrosa la distinción entre tiempo de ejecución y tiempo de compilación, al punto en que todo es en tiempo de ejecución. El compilador hace parte del sistema en tiempo de ejecución. Una declaración de clase es ejecutable. Compilarla y ejecutarla, crea una clase, la cual es un valor en el lenguaje (ver figura 7.5). El valor de tipo clase se puede pasar a `New` para crear un objeto.

Un sistema de programación no necesita distinguir estrictamente entre tiempo de compilación y tiempo de ejecución. La distinción no es más que una manera de ayudar al compilador a realizar cierto tipo de optimizaciones. La mayoría de los lenguajes dominantes, incluyendo C++ y Java, hacen esta distinción. Típicamente, unas pocas operaciones (como las declaraciones) se pueden ejecutar solamente en tiempo de compilación, y todas las otras operaciones se pueden ejecutar solamente en tiempo de ejecución. Entonces, el compilador puede ejecutar todas las declaraciones al mismo tiempo, sin ninguna interferencia de parte de la ejecución del programa. Esto le permite realizar más optimizaciones poderosas al momento de generar código, pero reduce de manera importante la flexibilidad del lenguaje. Por ejemplo, la genericidad y la instanciación dejan de estar disponibles, como herramientas generales para el programador.

Debido a la naturaleza dinámica de Mozart, el papel del compilador es mínimo. Como el compilador no ejecuta realmente ninguna declaración (sólo las convierte

*Programación orientada a objetos*

```
class Cuenta
 attr saldo:0
 meth transferir(Cant)
 saldo:=@saldo+Cant
 end
 meth pedirSaldo(Sal)
 Sal=@saldo
 end
 meth transferEnLote(CantList)
 for A in CantList do {self transferir(A)} end
 end
end
```

Figura 7.6: Cuenta: un ejemplo de clase.

en declaraciones ejecutables), entonces necesita muy poco conocimiento de la semántica del lenguaje. De hecho, el compilador sí tiene cierto conocimiento de la semántica del lenguaje, pero esto es una optimización que permite la detección temprana de ciertos errores y un código compilado más eficiente. Aún se puede añadir un poco más de conocimiento al compilador, e.g., para detectar errores en la jerarquía de clases en los casos en que el compilador pueda deducir cuáles son las etiquetas de los métodos.

### 7.3.2. Control de acceso a los métodos (ligadura estática y dinámica)

Cuando se ejecuta un método dentro de un objeto, con frecuencia se desea invocar otro método en el mismo objeto, i.e., hacer una especie de invocación recursiva. Esto parece bastante sencillo, pero se vuelve ligeramente más complicado cuando está involucrada la herencia. La herencia se utiliza para definir una clase nueva que extiende una clase existente. Dos clases están involucradas en esta definición: la clase nueva y la clase existente. Ambas pueden tener métodos con el mismo nombre, y la clase nueva puede querer invocar cualquiera de ellos. Esto significa que necesitamos dos maneras de realizar una invocación recursiva, las cuales se denominan ligadura estática y ligadura dinámica. Las introducimos por medio de un ejemplo.

#### *Un ejemplo*

Considere la clase `Cuenta` definida en la figura 7.6. Esta clase modela una cuenta de banco sencilla, con un saldo. Podemos trasnferir dinero a ella con el método `transferir`, consultar el saldo con el método `pedirSaldo`, y realizar una serie de transferencias con el método `transferEnLote`. Note que `transferEnLote`. invoca a `transferir` para cada transferencia.

Extendamos `Cuenta` para conservar un registro de todas las transacciones que se hacen. Una forma, consiste en utilizar la herencia, anulando el antiguo método

transferir y creando uno nuevo, propio de esta clase:

```
class CuentaVigilada from Cuenta
 meth transferir(Cant)
 {LogObj agregaEntrada(transferir(Cant))}

 ...
end
```

donde LogObj es un objeto que conserva los registros. Creemos una cuenta vigilada con un saldo inicial de 100:

```
CtaVig={New CuentaVigilada transferir(100)}
```

Ahora la pregunta es, ¿qué pasa cuando invocamos transferEnLote? ¿Se invoca el método transferir de Cuenta o el nuevo método transferir de CuentaVigilada? Podemos deducir cuál debería ser la respuesta, si suponemos que una clase define una abstracción de datos en el estilo objeto. La abstracción de datos tiene un conjunto de métodos. Para CuentaVigilada, este conjunto consiste de los métodos pedirSaldo y transferEnLote definidos en Cuenta así como del método transferir definido en la misma CuentaVigilada. Por lo tanto, la respuesta es que transferEnLote debe invocar el nuevo transferir de CuentaVigilada. Esto se denomina ligadura dinámica, y se escribe como una invocación a **self**, i.e., como **{self transferir(A)}**.

Cuando se definió Cuenta, no existía CuentaVigilada todavía. Al utilizar ligadura dinámica se conserva abierta la posibilidad de que se pueda extender Cuenta por medio de la herencia, así como se asegura que la clase nueva es una abstracción de datos que extiende correctamente la anterior. Es decir, se conserva la funcionalidad de la anterior abstracción al tiempo que se añade una funcionalidad nueva.

Sin embargo, la ligadura dinámica no es normalmente suficiente para implementar la abstracción extendida. Para ver por qué, investiguemos más de cerca cómo se define el nuevo transferir. La definición completa es:

```
class CuentaVigilada from Cuenta
 meth transferir(Cant)
 {LogObj agregaEntrada(transferir(Cant))}

 Cuenta,transferir(Cant)
 end
end
```

Dentro del nuevo método transferir, tenemos que invocar el antiguo método transferir. No podemos utilizar ligadura dinámica, pues esto siempre invocaría al nuevo método transferir. En su lugar, utilizamos otra técnica, denominada ligadura estática. En ésta, invocamos un método señalando la clase del método. La notación Cuenta,transferir(Cant) señala el método transferir en la clase Cuenta.

## Discusión

Tanto la ligadura estática como la dinámica se necesitan para anular métodos cuando se utiliza la herencia. La ligadura dinámica permite a la clase nueva extender correctamente la clase antigua dejando que los métodos antiguos invoquen los métodos nuevos, aunque los métodos nuevos no existan en el momento en que los métodos antiguos se definen. La ligadura estática, por su parte, permite que los nuevos métodos invoquen los métodos antiguos cuando tengan que hacerlo. Resumimos las dos técnicas:

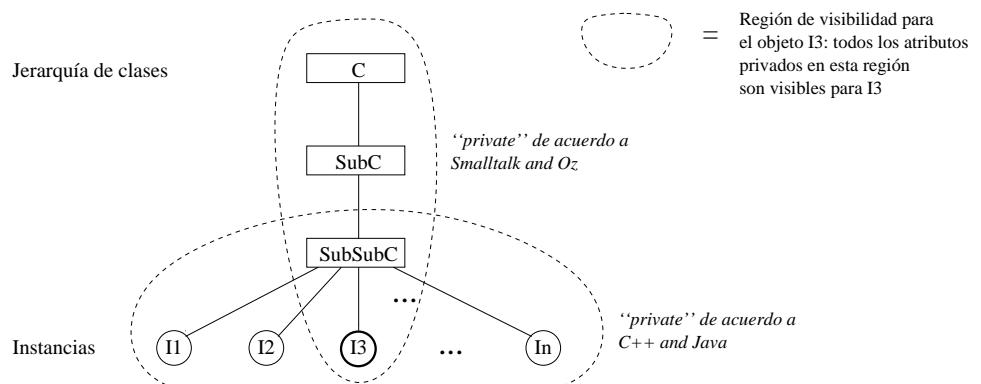
- *Ligadura dinámica.* Se escribe `{self M}`. Este tipo de ligadura escoge el correspondiente método `M` visible en el objeto actual, tomando en cuenta la anulación que haya sido realizada.
- *Ligadura estática.* Se escribe `C, M` (con una coma), donde `C` es una clase donde se define el correspondiente método `M`. Este tipo de ligadura escoge el método `M` visible en la clase `C`, tomando en cuenta las anulaciones de métodos desde la clase raíz hasta la clase `C`, pero no más. Si el objeto es de una subclase de `C` que ha anulado el método `M` de nuevo, entonces ésta anulación no es tenida en cuenta.

La ligadura dinámica es el único comportamiento posible para los atributos. La ligadura estática no es posible para ellos pues los atributos anulados simplemente no existen, ni en un sentido lógico (el único objeto que existe es la instancia resultante de la clase después de realizada la herencia) ni en un sentido práctico (la implementación no reserva memoria para ellos).

### 7.3.3. Control de encapsulación

El principio para controlar la encapsulación en un lenguaje orientado a objetos consiste en limitar el acceso a los miembros de la clase, a saber, atributos y métodos, de acuerdo a los requerimientos de la arquitectura de la aplicación. Cada miembro se define con un alcance. El alcance es la parte del texto del programa en el cual el miembro es visible, i.e., donde, al mencionar su nombre, se accede a él. Normalmente, el alcance se define estáticamente, por la estructura del programa, pero también se puede definir dinámicamente, a saber, durante la ejecución, si se utilizan nombres (ver más abajo).

Normalmente, los lenguajes de programación fijan un alcance por defecto a cada miembro en el momento en que se declara. Este alcance por defecto puede ser alterado con palabras reservadas especiales. Típicamente, estas palabras reservadas son `public`, `private`, y `protected`. Desafortunadamente, los distintos lenguajes usan estos términos para definir alcances ligeramente diferentes. La visibilidad en los lenguajes de programación es un concepto difícil. En el espíritu de [50], trataremos de colocar orden en este caos.



**Figura 7.7:** El significado de “private.”

### *Alcances público y privado*

Los dos más importantes tipos de alcances son el alcance público y el privado, con los significados siguientes:

- Un miembro privado es aquel que sólo es visible en la instancia correspondiente al objeto. Esta instancia puede ver todos los miembros definidos en su clase y en sus superclases. El alcance privado define, entonces, una especie de visibilidad vertical.
- Un miembro público es aquel que es visible en cualquier parte del programa.

De acuerdo a esta definición, tanto en Smalltalk como en Oz, los atributos son privados y los métodos son públicos.

Estas definiciones de alcance privado y público son naturales si las clases se utilizan para construir abstracciones de datos. Miremos por qué:

- Primero que todo, ¡una clase no es lo mismo que la abstracción de datos que ella define! La clase es un incremento; ella define una abstracción de datos como una modificación incremental de sus superclases. La clase solamente se necesita durante la construcción de la abstracción. Por su parte, la abstracción de datos no es un incremento; ella vale por sí misma, con todos sus atributos y métodos propios. Muchos de estos pueden venir de las superclases y no de la clase.
- Segundo, los atributos son internos a la abstracción de datos y deben ser invisibles desde el exterior. Esta es exactamente la definición de alcance privado.
- Finalmente, los métodos conforman la interfaz externa de la abstracción de datos, por lo tanto deberían ser visibles para todas las entidades que referencian la abstracción. Esta es exactamente la definición de alcance público.

### Construyendo otros alcances

Las técnicas para escribir programas para controlar la encapsulación están basadas esencialmente en dos conceptos: alcance léxico y valores de tipo nombre. Los alcances privado y público definidos atrás se pueden implementar con estos dos conceptos. Sin embargo, también se pueden expresar otros tipos de alcance con ellos. Por ejemplo, es posible expresar los alcances privado y protegido de C++ y Java, así como escribir programas que tengan políticas de seguridad mucho más elaboradas. La técnica fundamental es permitir que las cabezas de los métodos sean valores de tipo nombre en lugar de átomos. Un nombre es una constante inexpugnable; la única manera de conocer un nombre es que alguien entregue la referencia a él (ver sección 3.7.5 y apéndice B.2). De esta manera, un programa puede pasar la referencia de forma controlada solamente a aquellas áreas del programa en las cuales ésta debe ser visible.

En los ejemplos de las secciones previas, usamos átomos como etiquetas de los métodos. Pero los átomos no son seguros: si un tercero consigue la representación impresa del átomo (ya sea adivinándola o por otro medio), entonces él o ella también puede invocar el método. Los nombres son una solución sencilla para tapar este tipo de agujeros de seguridad. Esto es importante para los proyectos de desarrollo de *software* con interfaces bien definidas entre los diferentes componentes. Y es aún más importante para programas distribuidos abiertos, en los cuales suele coexistir código escrito en momentos diferentes por grupos diferentes (ver capítulo 11 (en CTM)).

#### Métodos privados (en el sentido de C++ y Java)

Cuando una cabeza de método es un valor de tipo nombre, entonces su alcance se limita a todas las instancias de la clase, pero no a las subclases o sus instancias. Esto es lo que significa privado en el sentido que lo hacen C++ y Java. Debido a su utilidad, el sistema de objetos de este capítulo ofrece soporte sintáctico para esta técnica. Hay dos maneras de escribirla, dependiendo de si el nombre se define implícitamente dentro de la clase o si proviene de afuera de ella:

- Utilizando un identificador de variable como la cabeza del método. Cuando la clase se define, se crea implícitamente un nombre que se liga a esa variable. Por ejemplo:

```
class C
 meth A(X)
 % Cuerpo del método
 end
end
```

La cabeza del método `A` se liga a un nombre. La variable `A` solamente es visible dentro de la definición de la clase. Una instancia de `C` puede invocar el método `A` en cualquier otra instancia de `C`. El método `A` es invisible para las definiciones de subclases. Esta es una especie de visibilidad horizontal, que corresponde al concepto

de método privado tal como se maneja en C++ y Java (pero no en Smalltalk). Como se muestra en la figura 7.7, el significado de privado en C++ y Java es muy diferente del significado de privado en Smalltalk y Oz. En estos últimos, lo privado es relativo a un objeto y sus clases, e.g., I3 en la figura. En C++ y Java, lo privado es relativo a una clase y sus instancias, e.g., SubSubC en la figura.

■ Utilizando un identificador de variable marcado, con el símbolo de escape “!”, como la cabeza del método. El signo de admiración ! indica que declararemos y ligaremos el identificador de variable por fuera de la clase. Cuando la clase se defina, la cabeza del método será cualquier cosa a la que esté ligada la variable. Este es un mecanismo muy general que se puede usar para proteger los métodos de muchas maneras. También se puede usar para otros propósitos diferentes a seguridad (ver sección 7.2.6). El ejemplo a continuación hace exactamente lo mismo que el ejemplo previo:

```
local
 A={NewName}
in
 class C
 meth !A(X)
 % Cuerpo del método
 end
end
end
```

Esto crea un nombre en el momento de la definición de la clase, tal como en el caso previo, y liga la cabeza del método con ese nombre. De hecho, la definición previa es precisamente una abreviación de este ejemplo.

Dejar que el programador determine la etiqueta del método permite definir una política de seguridad con una granularidad muy fina. El programa puede pasar la etiqueta del método solamente a aquellas entidades que necesiten conocerla.

#### *Métodos protegidos (en el sentido de C++)*

Por defecto, los métodos, en el sistema de objetos de este capítulo, son públicos. Utilizando nombres, podemos construir el concepto de un método protegido, incluyendo tanto la versión de C++ como la versión de Java. En C++, un método protegido solamente es accesible desde la clase donde fue definido o en las clases descendientes (y en todas las instancias de objetos de esas clases). El concepto de método protegido es una combinación de la noción de método privado de Smalltalk con la noción de método privado de C++/Java: tiene tanto un componente horizontal como uno vertical. Miremos cómo expresar la noción de método protegido de C++. La noción en Java es algo diferente, y la dejamos como un ejercicio. En la clase siguiente, el método A está protegido:

```
class C
 attr pa:A
 meth A(X) skip end
 meth foo(...) {self A(5)} end
end
```

El método A está protegido porque el atributo pa almacena una referencia a A. Ahora creamos una subclase C1 de C. Podemos acceder al método A de la subclase, de la manera siguiente:

```
class C1 from C
 meth b(...) A=@pa in {self A(5)} end
end
```

El método b accede al método con etiqueta A a través del atributo pa, el cual existe en la subclase. La etiqueta del método se puede almacenar en el atributo pues es simplemente un valor.

### *Alcance de los atributos*

Los atributos siempre son privados. La única forma de volverlos públicos es por medio de métodos. Como el tipamiento es dinámico, es posible definir métodos genéricos que dan derecho de acceso para lectura y escritura de todos los atributos. La clase Inspector en la sección 7.2.7 muestra una forma de hacerlo. Cualquier clase que hereda de Inspector tendrá todos sus atributos potencialmente públicos. Los atributos átomos no son seguros porque pueden ser adivinados. Los atributos nombres son seguros aún utilizando Inspector, porque no pueden ser adivinados.

### *¿Átomos o nombres como cabezas de métodos?*

¿Cuando se debería usar un átomo o un nombre como cabeza de método? Por defecto, los átomos son visibles a través de todo el programa y los nombres sólo son visibles en el alcance léxico de su creación. Podemos dar una regla sencilla para la implementación de clases: use nombres para métodos internos, y use átomos para métodos externos.

Los lenguajes más populares de programación orientada a objetos (e.g., Smalltalk, C++, y Java) sólo soportan átomos como cabezas de métodos; no tienen soporte para los nombres. Estos lenguajes tienen operaciones especiales sobre las cabezas de métodos para restringir su visibilidad (e.g., las declaraciones `private` y `protected`). Por otro lado, los nombres son bastante prácticos. Su visibilidad se puede extender pasando sus referencias al alrededor. Pero el enfoque basado en habilidades ejemplificado por los nombres no es popular todavía. Miremos un poco más de cerca los compromisos entre usar nombres versus usar átomos.

Los átomos se identifican únicamente por sus representaciones impresas. Esto significa que se pueden almacenar en archivos fuente, correos electrónicos, páginas Web, etc. En particular, ¡los átomos se pueden almacenar en la cabeza del programador! Cuando se escribe un programa grande, un método se puede invocar desde cualquier parte con tan sólo dar su representación impresa. Por el otro lado,

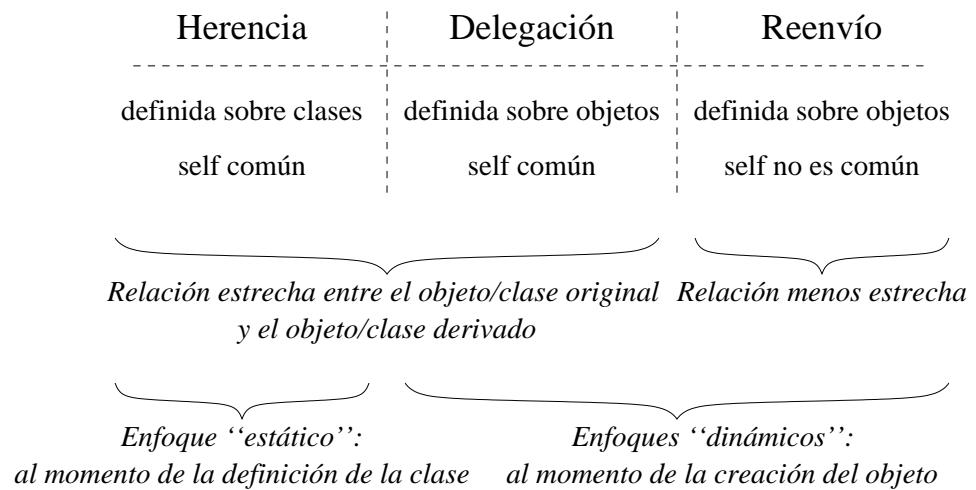


Figura 7.8: Maneras diferentes de extender la funcionalidad.

con los nombres esto es más difícil: el programa mismo tiene que pasar de alguna manera el nombre a quien desea invocar el método. Esto añade cierta complejidad al programa así como cierto trabajo adicional al programador. Los átomos ganan, entonces, tanto por la sencillez de los programas, como por el factor sicológico de comodidad durante el desarrollo.

Los nombres tienen otras ventajas. Primero, es imposible tener conflictos con la herencia (sencilla o múltiple). Segundo, la encapsulación se puede manejar mejor, pues una referencia a un objeto no da necesariamente derecho a invocar los métodos del objeto. Por lo tanto, el programa como un todo puede ser menos propenso a errores y mejor estructurado. Un punto final es que los nombres pueden tener soporte sintáctico para simplificar su uso. Por ejemplo, en el sistema objeto de este capítulo, es suficiente con colocar en mayúscula la cabeza del método.

#### 7.3.4. Reenvío y delegación

La herencia es una manera de reutilizar funcionalidades ya definidas al momento de definir funcionalidades nuevas. La herencia puede ser difícil de usar bien, porque implica una relación estrecha entre la clase original y su extensión. A veces es mejor utilizar enfoques más permisivos, como el reenvío<sup>3</sup> y la delegación. Ambas se definen al nivel de los objetos: si el objeto Obj1 no entiende el mensaje M, entonces se pasa transparentemente M al objeto Obj2. En la figura se comparan estos dos enfoques con la herencia.

El reenvío y la delegación difieren en la forma como cada uno trata el **self**.

---

3. Nota del traductor: *forwarding* en inglés.

```
local
 class MezclarReenvío
 attr Reenvío:ninguno
 meth asgnReenvío(F) Reenvío:=F end
 meth otherwise(M)
 if @Reenvío==ninguno then raise métodoIndefinido end
 else {@Reenvío M} end
 end
 end
in
 fun {NewF Clase Inic}
 {New class $ from Clase MezclarReenvío end Inic}
 end
end
```

Figura 7.9: Implementación del reenvío.

En el reenvío, los objetos Obj1 y Obj2 conservan sus identidades separadas. Una invocación a **self** en Obj2 permanecerá en Obj2. En la delegación, sólo existe una identidad, a saber, la del objeto Obj1. Una invocación a **self** en Obj2 invocará a Obj1. Decimos que la delegación, al igual que la implementación de la herencia, implica un **self** común. Por su parte, el reenvío no implica un **self** común.

Mostremos cómo expresar el reenvío y la delegación. Definimos las funciones NewF y NewD, para la creación especial de objetos, para el reenvío y la delegación, respectivamente. Nos ayudamos de la flexibilidad de nuestro sistema de objetos: usamos el método otherwise, mensajes como valores, y la creación dinámica de clases. Empezamos con el reenvío pues es el más sencillo.

### Reenvío

Un objeto puede reenviar a cualquier otro objeto. En el sistema de objetos de este capítulo, esto se puede implementar con el método otherwise(*M*) (ver sección 7.2.6). El argumento *M* es un mensaje de primera clase que se puede enviar a otro objeto. En la figura 7.9 se presenta la implementación de NewF, la cual reemplaza a New para la creación de objetos. Los objetos creados con NewF tienen un método asgnReenvío(*F*) que les permite definir dinámicamente el objeto al cual van a reenviar los mensajes que no entiendan. Creemos dos objetos Obj1 y Obj2 tales que Obj2 reenvíe a Obj1:

```

local
 AsgnSelf={NewName}
 class MezclarDelegación
 attr this Delegado:ninguno
 meth !AsgnSelf(S) this:=S end
 meth asign(A X) A:=X end
 meth obt(A ?X) X=@A end
 meth asignDelegado(D) Delegado:=D end
 meth Del(M S) SS in
 SS=@this this:=S
 try {self M} finally this:=SS end
 end
 meth invcr(M) SS in
 SS=@this this:=self
 try {self M} finally this:=SS end
 end
 meth otherwise(M)
 if @Delegado==ninguno then
 raise métodoIndefinido end
 else
 {@Delegado Del(M @this)}
 end
 end
 end
 in
 fun {NewD Clase Inic}
 Obj={New class $ from Clase MezclarDelegación end Inic}
 in
 {Obj AsgnSelf(Obj)}
 Obj
 end
 end
 end

```

Figura 7.10: Implementación de la delegación (parte 1: creación de objetos).

```

class C1
 meth inic skip end
 meth cubo(A B) B=A*A*A end
end

class C2
 meth inic skip end
 meth cuadrado(A B) B=A*A end
end

Obj1={NewF C1 inic}
Obj2={NewF C2 inic}
{Obj2 asignReenvío(Obj1)}

```

Invocar {Obj2 cubo(10 X)} llevará a Obj2 a reenviar el mensaje a Obj1.

| Operation               | Sintaxis original | Sintaxis de delegación        |
|-------------------------|-------------------|-------------------------------|
| Invocación de objeto    | {<obj> M}         | {<obj> invcr(M)}              |
| Invocación de Self      | {self M}          |                               |
| Consulta atributo       | @{attr}           | {@this obt({attr} \$)}        |
| Modificación atributo   | {attr} := X       | {@this asgn({attr} X)}        |
| Definición del delegado |                   | {<obj>1 asgnDelegado(<obj>2)} |

Tabla 7.2: Implementación de la delegación (parte 2: utilizando objetos y atributos).

### Delegación

La delegación es una herramienta poderosa para estructurar dinámicamente un sistema [106]. Ella nos permite construir una jerarquía entre objetos en lugar de hacerlo entre clases. En lugar de que un objeto herede de una clase (en el momento de la definición de la clase), permitimos que un objeto le delegue a otro objeto (en el momento de la creación del objeto). La delegación puede lograr los mismos efectos que la herencia, con dos diferencias importantes: la jerarquía es entre objetos, no entre clases, y se puede cambiar en cualquier momento.

Dados cualesquier dos objetos Obj1 y Obj2, suponemos que existe un método `asgnDelegado` tal que {Obj2 `asgnDelegado`(Obj1)} configura a Obj2 para delegar en Obj1. En otras palabras, Obj1 se comporta como la “superclase” de Obj2. Cuando se invoca un método que no está definido en Obj2, la invocación se ensaya de nuevo en Obj1. La cadena de delegación puede crecer a cualquier tamaño. Si existe un objeto Obj3 que delega a Obj2, entonces invocar a Obj3 puede escalar la cadena hasta Obj1.

Una propiedad importante de la semántica de la delegación es que el `self` se preserva siempre: es el `self` del objeto original que inició la cadena de delegación. Se sigue, entonces, que el estado del objeto (los atributos) también es el estado del objeto original. En ese sentido, los otros objetos juegan el papel de clases: en una primera instancia, son sus métodos los que son importantes en la delegación, no los valores de sus atributos.

Implementemos la delegación utilizando nuestro sistema de objetos. En la figura 7.10 se presenta la implementación de `NewD`, la cual reemplaza a `New` para la creación de objetos. Podemos usar la delegación con estos objetos y sus atributos si usamos la sintaxis mostrada en la tabla 7.2. Esta sintaxis especial podría eliminarse con una abstracción lingüística apropiada. Ahora presentamos un ejemplo sencillo para ver cómo funciona la delegación. Definimos dos objetos Obj1 y Obj2 y hacemos que Obj2 delegue en Obj1. Cada objeto tiene un atributo `i` y una manera de incrementarlo. Con herencia, esto se vería así:

```

class C1
 attr i:0
 meth inic skip end
 meth inc(I)
 {@this asgn(i {@this obt(i $)}+I)}
 end
 meth browse
 {@this inc(10)}
 {Browse c1#{@this obt(i $)}}
 end
 meth c {@this browse} end
end
Obj1={NewD C1 inic}

class C2
 attr i:0
 meth inic skip end
 meth browse
 {@this inc(100)}
 {Browse c2#{@this obt(i $)}}
 end
end
Obj2={NewD C2 inic}
{Obj2 asgnDelegado(Obj1)}

```

Figura 7.11: Un ejemplo de delegación.

```

class C1NoDeleg
 attr i:0
 meth inic skip end
 meth inc(I) i:=@i+I end
 meth browse {self inc(10)} {Browse c1#@i} end
 meth c {self browse} end
end

class C2NoDeleg from C1NoDeleg
 attr i:0
 meth inic skip end
 meth browse {self inc(100)} {Browse c2#@i} end
end

```

Con nuestra implementación de la delegación podemos conseguir el mismo efecto utilizando el código de la figura 7.11. Parece más verboso, pero esto es solamente porque el sistema no tiene soporte sintáctico para la delegación, y no por el uso del concepto en sí. Note que esto sólo roza la superficie de lo que podríamos hacer con la delegación. Por ejemplo, invocar `asgnDelegado` de nuevo, podría cambiar la jerarquía del programa en tiempo de ejecución. Ahora invoquemos `Obj1` y `Obj2`:

```

{Obj2 invcr(c)}
{Obj1 invcr(c)}

```

Al realizar estas invocaciones varias veces se puede ver que cada objeto guarda su propio estado local, que `Obj2` “hereda” los métodos `inc` y `c` del objeto `Obj1`, y que `Obj2` “anula” el método `browse`. Hagamos la cadena de delegación más larga:

```
class C2b
 attr i:0
 meth inic skip end
end
ObjX={NewD C2b inic}
{ObjX asgnDelegado(Obj2)}
```

`ObjX` hereda todo su comportamiento de `Obj2`. Es idéntico a `Obj2` salvo que tiene un estado local diferente. La jerarquía de delegación tiene ahora tres niveles: `ObjX`, `Obj2`, y `Obj1`. Cambiemos la jerarquía haciendo que `ObjX` delegue en `Obj1`:

```
{ObjX asgnDelegado(Obj1)}
{ObjX invcr(c)}
```

En la jerarquía nueva, `ObjX` hereda su comportamiento de `Obj1`. `ObjX` utiliza el método `browse` de `Obj1`, por lo tanto se incrementará en 10 en lugar de hacerlo en 100.

### 7.3.5. Reflexión

Un sistema es reflectivo si puede consultar parte de su estado de ejecución mientras se está ejecutando. La reflexión puede ser puramente introspectiva (solamente leer el estado interno, sin modificarlo) o intrusiva (lectura y modificación del estado interno). La reflexión se puede realizar a un alto o un bajo nivel de abstracción. Un ejemplo de reflexión de alto nivel sería la capacidad de ver las entradas de la pila semántica como clasururas. Esto se explicaría de manera sencilla en términos de la máquina abstracta. Por otro lado, la capacidad de leer la memoria como un arreglo de enteros sería reflexión de bajo nivel. No existe una manera sencilla de explicar eso en términos de la máquina abstracta.

#### *Protocolos meta-objeto*

Debido a su riqueza, la programación orientada a objetos es un área particularmente fértil para la reflexión. Por ejemplo, el sistema podría permitir examinar, o incluso cambiar, la jerarquía de herencia, mientras el programa está ejecutándose. Esto es posible en Smalltalk. El sistema podría permitir cambiar la forma como los objetos se ejecutan en un nivel básico, e.g., cómo funciona la herencia (cómo se realiza la búsqueda de un método en la jerarquía de clases) y cómo se invocan los métodos. La descripción de cómo funciona un sistema de objetos en un nivel básico se denomina un protocolo meta-objeto. La capacidad de cambiar el protocolo meta-objeto es una poderosa manera de modificar un sistema de objetos. Los protocolos meta-objeto se utilizan con muchos propósitos: depuración, personalización, y separación de asuntos (e.g., adición transparente de la codificación o de cambios de formato en la invocación de los métodos). Los protocolos meta-objeto se inventaron

originalmente en el contexto de Common Lisp Object System (CLOS) [89, 131], y son aún un área activa de investigación en POO.

### *Envolvimiento de métodos*

Los protocolos meta-objeto se usan comúnmente para “envolver” los métodos, i.e., interceptar cada invocación de método, posiblemente realizar una operación definida por el usuario antes y después de la invocación, y posiblemente modificar los argumentos de la invocación misma. En nuestro sistema de objetos, podemos implementar esto de manera sencilla, aprovechando el hecho de que los objetos son procedimientos de un argumento. Por ejemplo, escribamos un programa Vigilante que vigile el comportamiento de un programa orientado a objetos. El programa Vigilante debe desplegar en el browser la etiqueta del método antes y después de terminar su invocación. Esta es una versión de `New` que implementa esto:

```
fun {NuevoVigilante Clase Inic}
 Obj={New Clase Inic}
 proc {ObjVigilado M}
 {Browse entrandoA({Label M})}
 {Obj M}
 {Browse saliendoDe({Label M})}
 end
in ObjVigilado end
```

Un objeto creado con `NuevoVigilante` se comporta de manera idéntica a uno creado con `New`, salvo que las invocaciones a sus métodos (excepto las invocaciones a `self`) se vigilan. La definición de `NuevoVigilante` utiliza programación de alto orden: el procedimiento `ObjVigilado` tiene una referencia externa a `Obj`. Esta definición se puede extender fácilmente para envolver los métodos de forma más sofisticada. Por ejemplo, el mensaje `M` podría transformarse de alguna manera antes de pasárselo a `Obj`.

Una segunda manera de implementar `NuevoVigilante` es por medio de una clase y no de un procedimiento. Esto nos lleva a la definición siguiente:

```
fun {NuevoVigilante2 Clase Inic}
 Obj={New Clase Inic}
 VInic={NewName}
 class Vigilante
 meth !VInic skip end
 meth otherwise(M)
 {Browse entrandoA({Label M})}
 {Obj M}
 {Browse saliendoDe({Label M})}
 end
 end
in {New Vigilante VInic} end
```

Esta implementación tiene la misma limitación anterior con `self`. Aquí se utiliza la creación dinámica de clases, el método `otherwise`, y un nombre fresco `VInic` para el método de inicialización para evitar conflictos con otras etiquetas de métodos. Note

que la clase `Vigilante` tiene una referencia externa a `obj`. Si su lenguaje no permite esto, entonces se puede modificar la técnica para almacenar `obj` en un atributo de la clase `Vigilante`. Como bien se puede ver, la solución con programación de alto orden es más concisa.

### **Reflexión del estado del objeto**

Mostremos un ejemplo, sencillo pero útil, de reflexión en POO. Nos gustaría ser capaces de leer y escribir el estado completo de un objeto, independientemente de la clase del objeto. El sistema de objetos de Mozart provee esta capacidad a través de la clase `ObjectSupport.reflect`. Al heredar de esta clase se tienen los tres métodos adicionales siguientes:

- `clone(x)` crea una copia de `self` y la liga a `x`. La copia es un objeto nuevo de la misma clase y con los mismos valores de los atributos.
- `toChunk(x)` liga `x` con un valor protegido (un “pedazo”) que contiene el valor actual de los atributos.
- `fromChunk(x)` modifica el estado del objeto a `x`, donde `x` se obtuvo previamente de una invocación a `toChunk`.

Un pedazo es como un registro, pero con un conjunto de operaciones restringidas. Un pedazo está protegido en el sentido que solamente los programas autorizados pueden mirar dentro de él (ver apéndice B.4). Los pedazos se pueden utilizar para construir abstracciones de datos seguras, tal como se mostró en la sección 3.7.5. Extendamos la clase `Contador` que vimos anteriormente para realizar reflexión del estado:

```
class Contador from ObjectSupport.reflect
 attr val
 meth inic(Valor)
 val:=Valor
 end
 meth browse
 {Browse @val}
 end
 meth inc(Valor)
 val:=@val+Valor
 end
end
```

Podemos definir dos objetos:

```
C1={New Contador inic(0)}
C2={New Contador inic(0)}
```

y luego transferir el estado del uno al otro:

```
{C1 inc(10)}
local x in {C1 toChunk(x)} {C2 fromChunk(x)} end
```

En este momento `C2` también tiene el valor 10. Este es un ejemplo simplista, pero la reflexión del estado es realmente una herramienta poderosa, que se puede

utilizar para construir abstracciones genéricas sobre objetos, i.e., abstracciones que funcionan sobre objetos de cualquier clase.

---

## 7.4. Programación con herencia

Todas las técnicas de programación asociadas a la programación con estado y a la programación declarativa siguen siendo válidas en el sistema de objetos de este capítulo. Las técnicas basadas en encapsulación y estado, para lograr programas modulares, son particularmente útiles. Vea el capítulo previo, y especialmente la discusión de programación basada en componentes, la cual se fundamenta en la encapsulación.

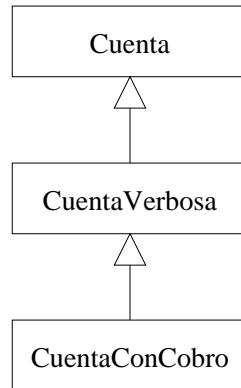
Esta sección se enfoca en las técnicas nuevas que aparecen gracias a la POO. Todas estas técnicas se centran en el uso de la herencia: primero, para usarla correctamente, y luego, para aprovecharse de su poder.

### 7.4.1. Utilización correcta de la herencia

Hay dos maneras de ver la herencia:

- *La visión de tipo.* En esta visión, las clases son tipos y las subclases son subtipos. Por ejemplo, suponga una clase `VentanaEtiquetada` que hereda de la clase `Ventana`. Todas las ventanas etiquetadas también son ventanas. La visión de tipo es consistente con el principio de que las clases deben modelar entidades del mundo real o algunas versiones abstractas de ellas. En la visión de tipo, las clases satisfacen la *propiedad de sustitución*: cada operación que funciona para un objeto de una clase `C` también funciona para objetos de una suclase de `C`. La mayoría de los lenguajes orientados a objetos, como Java y Smalltalk, se diseñaron con la visión de tipo [56, 58]. En la sección 7.4.1 se explora qué pasa si no se respeta la visión de tipo.
- *La visión de estructura.* En esta visión, la herencia es tan sólo otra herramienta de programación, utilizada para estructurar programas. Esta visión se *desaconseja fuertemente* porque las clases dejan de satisfacer la propiedad de sustitución. La visión de estructura es, casi, una fuente interminable de errores y de malos diseños. Los proyectos comerciales más importantes, los cuales permanecerán anónimos aquí, han fallado por esta razón. Unos pocos lenguajes orientados a objetos, notablemente Eiffel, se diseñaron desde el comienzo para permitir tanto la visión de tipo como la visión de estructura [113].

En la visión de tipo, cada clase existe por sí misma, por así decirlo, como una abstracción de datos de buena fé. En la visión de estructura, las clases son algunas veces un simple andamiaje, el cual sólo existe por su papel en la estructuración del programa.



**Figura 7.12:** Una jerarquía sencilla con tres clases.

### *Un ejemplo*

En la gran mayoría de los casos, la herencia debe respetar la visión de tipo. No hacerlo así, puede llevar a errores sutiles y perjudiciales que pueden envenenar todo el sistema. Presentamos un ejemplo. Tomamos como clase básica la clase `Cuenta` que vimos anteriormente, la cual se definió en la figura 7.6. Un objeto `A` de la clase `Cuenta` satisface la regla algebráica siguiente:

$$\{A \text{ pedirSaldo}(b)\} \quad \{A \text{ transferir}(s)\} \quad \{A \text{ pedirSaldo}(b')\}$$

con  $b + s = b'$ . Es decir, si el saldo inicial es  $b$  y se realiza una transferencia de  $s$ , entonces el saldo final es  $b + s$ . Esta regla algebráica se puede mirar como una especificación de `Cuenta`, o como lo veremos más adelante, un contrato entre `Cuenta` y el programa que utiliza sus objetos. (En una definición práctica de `Cuenta`, también habría otras reglas. Las dejamos por fuera por el propósito del ejemplo.) De acuerdo a la visión de tipo, las subclases de `Cuenta` también deberían implementar este contrato. Ahora, extendamos `Cuenta`, de dos formas. La primera extensión es conservativa, i.e., respeta la visión de tipo:

```

class CuentaVerbosa from Cuenta
 meth transferVerbosa(Cant)
 {self transferir(Cant)}
 {Browse `Saldo:#@saldo`}
 end
end

```

Sencillamente agregamos un método nuevo, `transferVerbosa`. Como los métodos existentes no cambiaron, se concluye que el contrato vale aún. Un objeto de la clase `CuentaVerbosa` funcionará correctamente en todos los casos en que un objeto de la clase `Cuenta` funcione. Ahora, realicemos una segunda extensión, más peligrosa:

```
class CuentaConCobro from CuentaVerbosa
 attr cobro:5
 meth transferir(Cant)
 CuentaVerbosa,transferir(Cant-@cobro)
 end
end
```

En la figura 7.12 se muestra la jerarquía resultante. La flecha utilizada en esta figura es la notación normal para representar un enlace de herencia. `CuentaConCobro` anula el método `transferir`. La anulación no es un problema en sí misma. El problema es que un objeto de la clase `CuentaConCobro` no funciona correctamente cuando se mira como un objeto de la clase `Cuenta`. Considere la siguiente secuencia de tres invocaciones:

```
{A pedirSaldo(B)} {A transferir(S)} {A pedirSaldo(B2)}
```

Si `A` es un objeto de la clase `CuentaConCobro`, entonces  $B+S-@cobro=B2$ . Si  $@cobro \neq 0$  entonces el contrato ya no vale. Esto daña cualquier programa que confíe en el comportamiento de los objetos de la clase `Cuenta`. Típicamente, el origen del daño no será obvio, pues está cuidadosamente escondido dentro de un método en alguna parte en una gran aplicación. Este daño aparecerá un buen tiempo después de que se haya hecho el cambio, en la forma de un ligero descuadre en los libros. La depuración de esos problemas “sencillos” es sorprendentemente difícil y consumidora de tiempo.

El resto de la sección 7.4 considera ante todo la visión de tipo. Casi todos los usos de la herencia deben respetar la visión de tipo. Sin embargo, la visión de estructura es, ocasionalmente, útil. Su principal uso está en cambiar el comportamiento del propio sistema de objetos. Para este propósito, la herencia debería ser utilizada sólo por expertos implementadores del lenguaje quienes entiendan claramente las ramificaciones de lo que están haciendo. Un ejemplo sencillo es el envolvimiento de métodos (ver sección 7.3.5), el cual requiere utilizar la visión de estructura. Recomendamos [113] para una discusión más profunda de la visión de tipo versus la visión de estructura.

### Diseño por contrato

La clase `Cuenta` ilustra un ejemplo de una técnica general para diseñar programas correctos y verificar su corrección. Decimos que un programa es *correcto* si se desempeña de acuerdo a su especificación. Una manera de probar la corrección de un programa consiste en razonar con una semántica formal. Por ejemplo, en un programa con estado podemos razonar utilizando la semántica axiomática de la sección 6.6. También podemos razonar con reglas algebraicas como en el ejemplo de `Cuenta`. Basado en estas técnicas para especificación formal, Bertrand Meyer desarrolló un método para diseñar programas correctos denominado *diseño por contrato* y lo implementó en el lenguaje Eiffel [113].

La idea principal del diseño por contrato consiste en que una abstracción de datos implica un contrato entre el diseñador de la abstracción y sus usuarios. Los

usuarios deben garantizar que la abstracción se invoca en la forma correcta, y en contraprestación, la abstracción se comporta en la forma correcta. Hay una analogía deliberada con los contratos en la sociedad. Se espera que todas las partes cumplan el contrato. El contrato puede ser formulado en términos de reglas algebraicas, como lo hicimos con el ejemplo de Cuenta, o en términos de precondiciones y poscondiciones, como lo vimos en la sección 6.6. El usuario asegura que las precondiciones son ciertas antes de invocar una operación. Entonces, la implementación de la abstracción de datos asegura que las poscondiciones son ciertas una vez la operación termina.

Existe una división de responsabilidades: el usuario es responsable de las precondiciones y la abstracción de datos es responsable de las poscondiciones.

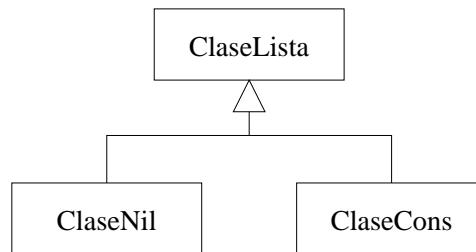
La abstracción de datos comprueba si el usuario cumple el contrato. Esto es especialmente sencillo si se utilizan precondiciones y poscondiciones. Las precondiciones se comprueban en la frontera entre el usuario y la abstracción de datos. Una vez dentro de la abstracción de datos, suponemos que las precondiciones fueron satisfechas. Ninguna comprobación se realiza dentro de la abstracción, de tal manera que no se complica su implementación. Esto es análogo con el funcionamiento de la sociedad. Por ejemplo, para entrar a un país, uno se presenta en la frontera para las comprobaciones de rigor (e.g., control de pasaporte, comprobaciones de seguridad, y aduana). Una vez uno ha entrado, no hay más comprobaciones.

Las precondiciones se pueden comprobar en tiempo de compilación o en tiempo de ejecución. Muchos sistemas industriales, e.g., en telecomunicaciones, realizan las comprobaciones en tiempo de ejecución. Estos sistemas ejecutan largos períodos de pruebas durante los cuales se registran las violaciones a los contratos. La verificación en tiempo de compilación es particularmente deseable pues las violaciones a los contratos se determinan antes de ejecutar el programa. Un tema interesante de investigación, consiste en determinar cuán expresivas pueden ser las precondiciones de manera que aún puedan comprobarse en tiempo de compilación.

### *Una historia que sirve de escarmiento*

Terminamos la discusión sobre el uso correcto de la herencia con una historia que sirve de escarmiento. Hace algunos años, una reconocida empresa emprendió un proyecto ambicioso basado en la POO. A pesar de un presupuesto de varios billones de dólares, el proyecto fracasó. Una de las muchas razones para el fracaso fue el uso incorrecto de la POO, en particular en lo que concierne a la herencia. Se cometieron dos grandes errores:

- La propiedad de substitución fue violada frecuentemente. Algunas rutinas que funcionaban correctamente con objetos de una clase dada no funcionaban con objetos de una subclase. Esto complicó el uso de los objetos: en lugar de que una rutina fuera suficiente para muchas clases, se necesitaron muchas rutinas.
- Se crearon subclases para resolver pequeños problemas de las clases. En lugar de resolver los problemas en la misma clase, se definieron subclases para “remendar”



**Figura 7.13:** Construcción de una jerarquía de acuerdo al tipo.

o “parchar” la clase. Esto se hizo tan frecuentemente que produjo capas sobre capas de parches. Las invocaciones a los objetos se volvieron lentas en un orden de magnitud. La jerarquía de clases se volvió innecesariamente profunda, lo cual incrementó la complejidad del sistema.

La lección a aprender es que se debe tener el cuidado de usar la herencia de manera correcta. Hay que respetar la propiedad de sustitución en la medida de lo posible. Hay que utilizar la herencia para agregar funcionalidades nuevas y no para parchar problemas en una clase. Hay que estudiar los patrones comunes de diseño para aprender el uso correcto de la herencia.

**Reingeniería** En este punto, deberíamos mencionar la disciplina de la reingeniería, la cual se puede usar para resolver problemas arquitecturales como esos dos usos incorrectos de la herencia [43, 15]. El objetivo general de la reingeniería es tomar un sistema existente y tratar de mejorar algunas de sus propiedades por medio de la modificación del código fuente. Muchas propiedades se pueden mejorar de esta manera: la arquitectura del sistema, la modularidad, el desempeño, la portabilidad, la calidad de la documentación, y la utilización de tecnologías nuevas. Sin embargo, la reingeniería no puede resucitar un proyecto fracasado. Eso sería como curar una enfermedad. Si el diseñador ha realizado una elección, el mejor enfoque sigue siendo prevenir la enfermedad, i.e., diseñar un sistema de manera que se pueda adaptar a requerimientos cambiantes. En la sección 6.7 y a lo largo del libro, presentamos principios de diseño que apuntan hacia este objetivo.

#### 7.4.2. Construcción de una jerarquía de acuerdo al tipo

En la sección 3.4.2 vimos que, para escribir programas recursivos, era una buena idea definir en primera instancia la estructura de datos, y luego construir el programa recursivo de acuerdo al tipo definido. Podemos usar una idea similar para construir jerarquías de herencia. Por ejemplo, considere el tipo lista  $\langle \text{Lista } T \rangle$ , el cual se define así:

*Programación orientada a objetos*

```

class ClaseLista
 meth esNil() raise metodoIndefinido end end
 meth concat(_) raise metodoIndefinido end end
 meth display raise metodoIndefinido end end
end

class ClaseNil from ClaseLista
 meth inic skip end
 meth esNil(B) B=true end
 meth concat(T U) U=T end
 meth desplegar {Browse nil} end
end

class ClaseCons from ClaseLista
 attr cab cola
 meth inic(H T) cab:=H cola:=T end
 meth esNil(B) B=false end
 meth concat(T U)
 U2={@cola concat(T $)}
 in
 U={New ClaseCons inic(@cab U2)}
 end
 meth desplegar {Browse @cab} {@cola desplegar} end
end

```

Figura 7.14: Listas en el estilo orientado a objetos.

$$\langle \text{Lista } T \rangle ::= \text{nil} \\ | \quad T \cdot | \cdot \langle \text{Lista } T \rangle$$

De acuerdo a esta definición una lista es el átomo `nil` o es de la forma `x|xr` donde `x` es del tipo  $\langle T \rangle$  y `xr` es del tipo  $\langle \text{Lista } T \rangle$ . Implementemos la abstracción lista en la clase `ClaseLista`. De acuerdo a la definición del tipo, definimos otras dos clases que heredan de `ClaseLista`, las cuales denominamos `ClaseNil` y `ClaseCons`. En la figura 7.13 se muestra la jerarquía. Esta jerarquía responde a un diseño natural para respetar la propiedad de substitución. Una instancia de `ClaseNil` es una lista, por lo tanto es fácil de utilizar en donde se requiera una lista. Lo mismo vale para `ClaseCons`.

En la figura 7.14 se define una abstracción de lista de acuerdo a esta jerarquía. En esta figura, `ClaseLista` es una clase abstracta: una clase en la cual algunos de sus métodos se dejan indefinidos. Tratar de invocar los métodos `esNil`, `concat`, y `desplegar` lanzarán una excepción. Las clases abstractas no se hacen con la intención de ser instanciadas, pues ellas no definen algunos métodos. La idea es definir otras clases que hereden de la clase abstracta y añadan los métodos faltantes. Esto nos lleva a una clase concreta, la cual puede ser instanciada pues todos sus métodos están definidos. `ClaseNil` y `ClaseCons` son clases concretas, pues en ellas se definen los métodos `esNil`, `concat`, y `desplegar`. La invocación `{L1 concat(L2)}`

```

class OrdGenérico
 meth inic skip end
 meth qsort(Xs Ys)
 case Xs
 of nil then Ys = nil
 [] P|Xr then S L in
 {self partición(Xr P S L)}
 {Append {self qsort(S $)}
P|{self qsort(L $)} Ys}
 end
 end
 meth partición(Xs P Ss Ls)
 case Xs
 of nil then Ss=nil Ls=nil
 [] X|Xr then Sr Lr in
 if {self comp(X P $)} then
 Ss=X|Sr Ls=Lr
 else
 Ss=Sr Ls=X|Lr
 end
 {self partición(Xr P Sr Lr)}
 end
 end
end

```

**Figura 7.15:** Una clase genérica de ordenamiento (con herencia).

L3) } liga L3 con la concatenación de L1 y L2, sin cambiar L1 o L2. La invocación {L desplegar} despliega la lista en el browser. Ahora, hagamos algunos cálculos con listas:

```

L1={New ClaseCons
 inic(1 {New ClaseCons
 inic(2 {New ClaseNil inic}))})
L2={New ClaseCons inic(3 {New ClaseNil inic}))}
L3={L1 concat(L2 $)}
{L3 desplegar}

```

Aquí se crean dos listas L1 y L2 y se concatenan para formar L3. Luego se despliega el contenido de L3 en el browser, como 1, 2, 3, nil.

#### 7.4.3. Clases Genéricas

Una clase genérica es una clase que solamente define parte de la funcionalidad de una abstracción de datos. Miremos dos formas de definir clases genéricas. La primera forma, utilizada frecuentemente en POO, usa la herencia. La segunda forma utiliza programación de alto orden. Veremos que la primera forma es simplemente una variación sintáctica de la segunda. En otras palabras, la herencia se puede ver como un estilo de programación basado en la programación de alto orden.

```
class OrdEntero from OrdGenérico
 meth comp(X Y B)
 B=(X<Y)
 end
end

class OrdRacional from OrdGenérico
 meth comp(X Y B)
 1/1(P Q)=X
 1/1(R S)=Y
 in B=(P*S<Q*R) end
end
```

Figura 7.16: Ordenamientos concretos (con herencia).

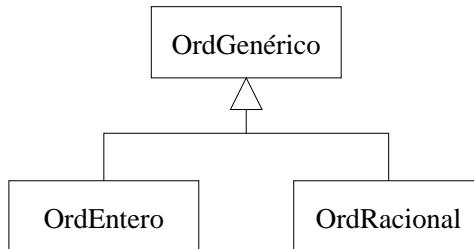


Figura 7.17: Una jerarquía de clases para genericidad.

### Utilizando la herencia

En la POO se utilizan las clases abstractas para hacer clases más genéricas. Por ejemplo, en la figura 7.15 se define una clase abstracta `OrdGenérico` para ordenar una lista. Esta clase utiliza el algoritmo quicksort, el cual necesita un operador de comparación booleano. La definición del operador de comparación booleano depende del tipo de datos que se esté ordenando. Otras clases podrán heredar de `OrdGenérico` y agregar sus definiciones de `comp`, e.g., para enteros, racionales, o cadenas de caracteres. En este caso, especializamos la clase abstracta para formar una clase concreta, i.e., una clase en la cual todos los métodos están definidos. En la figura 7.16 se definen las clases concretas `OrdEntero` y `OrdRacional`, las cuales heredan, ambas, de `OrdGenérico`. En la figura 7.17 se muestra la jerarquía resultante.

```

fun {HagaOrdenamiento Comp}
 class $
 meth inic skip end
 meth qsort(Xs Ys)
 case Xs
 of nil then Ys = nil
 [] P|Xr then S L in
 {self partición(Xr P S L)}
 {Append {self qsort(S $)} P|{self qsort(L $)} Ys}
 end
 end
 meth partición(Xs P Ss Ls)
 case Xs
 of nil then Ss=nil Ls=nil
 [] X|Xr then Sr Lr in
 if {Comp X P} then
 Ss=X|Sr Ls=Lr
 else
 Ss=Sr Ls=X|Lr
 end
 {self partición(Xr P Sr Lr)}
 end
 end
 end

```

**Figura 7.18:** Una clase genérica de ordenamiento (con programación de alto orden).

#### *Utilizando programación de alto orden*

Hay una segunda manera, natural, de crear clases genéricas, a saber, utilizando directamente programación de alto orden. Como las clases son valores de primera clase, podemos definir una función que recibe algunos argumentos y devuelve una clase especializada con esos argumentos. En la figura 7.18 se define la función HagaOrdenamiento que recibe una comparación booleana como su argumento y devuelve una clase de ordenamiento especializada con esa comparación. En la figura 7.19 se definen dos clases, OrdEntero y OrdRacional, que pueden ordenar listas de enteros y listas de números racionales (los últimos representados por parejas de enteros con etiqueta `'/'`). Ahora podemos ejecutar las siguientes declaraciones:

```

OrdEnt={New OrdEntero inic}
OrdRac={New OrdRacional inic}

{Browse {OrdEnt qsort([1 2 5 3 4] $)}}
{Browse {OrdRac qsort(['/(23 3) '/(34 11) '/(47 17)] $)}}

```

```
OrdEntero = {HagaOrdenamiento fun {$ X Y} X<Y end}

OrdRacional = {HagaOrdenamiento fun {$ X Y}
 '/'(P Q) = X
 '/'(R S) = Y
 in P*S<Q*R end}
```

**Figura 7.19:** Ordenamientos concretos (con programación de alto orden).

### Discusión

Es claro que estamos usando la herencia para “conectar” una operación dentro de otra. Esto es simplemente una forma de programación de alto orden, en la cual la primera operación se le pasa a la segunda. ¿Cuál es la diferencia entre las dos técnicas? En la mayoría de los lenguajes de programación, la jerarquía de herencia se debe definir en tiempo de compilación, lo cual lleva a una genericidad estática. Como es estática, el compilador es capaz de generar un código mejor o de hacer comprobación temprana de errores. La programación de alto orden, cuando ella es posible, nos permite definir clases nuevas en tiempo de ejecución. Esto nos lleva a una genericidad dinámica, la cual es más flexible.

#### 7.4.4. Herencia múltiple

La herencia múltiple es útil cuando un objeto tiene que ser, a la vez, dos cosas diferentes en el mismo programa. Por ejemplo, considere un paquete gráfico que puede mostrar una variedad de figuras geométricas, incluyendo círculos, líneas, y figuras más complejas. Nos gustaría definir una operación de “agrupamiento” que pueda combinar cualquier número de figuras en una sola figura compuesta. ¿Cómo podemos modelar esto con POO? Diseñaremos un programa sencillo, completamente funcional. Usaremos herencia múltiple para agregar la capacidad de agrupamiento a las figuras. La idea de este diseño viene de Bertrand Meyer [113]. Este sencillo programa puede ser extendido fácilmente a un paquete gráfico completo.

#### Figuras geométricas

Primero definimos la clase `Figura` para modelar figuras geométricas, con los métodos `inic` (inicializar la figura), `mover(X Y)` (mover la figura), y `desplegar` (visualizar la figura):

```
class Figura
 meth otherwise(M)
 raise metodoIndefinido end
 end
end
```

Esta es una clase abstracta; cualquier intento de invocar uno de sus métodos lanzará una excepción. Las figuras reales son instancias de subclases de `Figura`. Por ejemplo, esta es la clase `Línea`:

```
class Línea from Figura
 attr lienzo x1 y1 x2 y2
 meth inic(Lien X1 Y1 X2 Y2)
 lienzo:=Lien
 x1:=X1 y1:=Y1
 x2:=X2 y2:=Y2
 end
 meth mover(X Y)
 x1:=@x1+X y1:=@y1+Y
 x2:=@x2+X y2:=@y2+Y
 end
 meth desplegar
 {@lienzo create(line @x1 @y1 @x2 @y2)}
 end
end
```

Y esta es la clase `Círculo`:

```
class Círculo from Figura
 attr lienzo x y r
 meth inic(Lien X Y R)
 lienzo:=Lien
 x:=X y:=Y r:=R
 end
 meth mover(X Y)
 x:=@x+X y:=@y+Y
 end
 meth desplegar
 {@lienzo create(oval @x-@r @y-@r @x+@r @y+@r)}
 end
end
```

En la figura 7.20 se muestra cómo `Línea` y `Círculo` heredan de `Figura`. Este tipo de diagrama se denomina diagrama de clases. Este diagrama hace parte de UML, el Lenguaje de Modelamiento Unificado, un amplio conjunto de técnicas para modelar programas orientados a objetos [50]. Los diagramas de clase son una forma útil de visualizar la estructura de clases de un programa orientado a objetos. Cada clase se representa con un rectángulo con tres partes, una que contiene el nombre de la clase, otra que contiene los atributos, y una última que contiene los métodos que la clase define. Estos rectángulos se pueden conectar con líneas que representan enlaces de herencia.

### *Listas enlazadas*

Para agrupar varias figuras definimos la clase `ListaEnlazada`, con los métodos `inic` (inicializar la lista enlazada), `agr(F)` (agregar una figura a la lista), y `paratodos(M)` (ejecutar `{F M}` para todas las figuras):

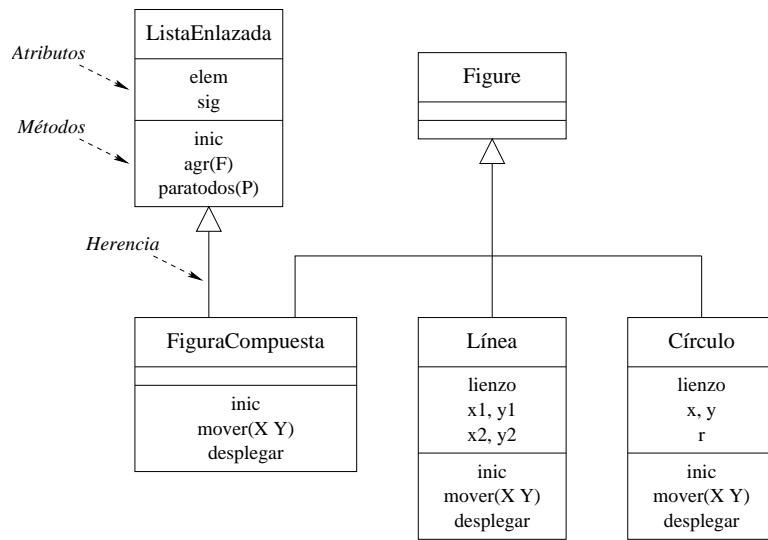


Figura 7.20: Diagrama de clases del paquete gráfico.

```

class ListaEnlazada
 attr elem sig
 meth inic(elem:E<=>null sig:N<=>null)
 elem:=E sig:=N
 end
 meth agr(E)
 sig:={New ListaEnlazada inic(elem:E sig:@sig)}
 end
 meth paratodos(M)
 if @elem\=null then {@elem M} end
 if @sig\=null then {@sig paratodos(M)} end
 end
end

```

El método `paratodos(M)` es particularmente interesante porque usa mensajes de primera clase. Una lista enlazada se representa como una secuencia de instancias de esta misma clase. El atributo `sig` de cada instancia referencia al siguiente en la lista. El último elemento tiene el atributo `sig` igual a `null`. Siempre existe un elemento en la lista, denominado el encabezado. El encabezado no puede ser visto por los usuarios de la lista enlazada; sólo se necesita para la implementación. El atributo `elem` del encabezado siempre tiene el valor `null`. Por lo tanto, una lista encadenada vacía corresponde a un encabezado con ambos atributos, `elem` y `sig` iguales a `null`.

### Figuras compuestas

¿Qué es una figura compuesta? Una figura compuesta es a la vez una figura y una lista enlazada de figuras. En consecuencia, definimos una clase `FiguraCompuesta` que hereda tanto de `Figura` como de `ListaEnlazada`:

```
class FiguraCompuesta from Figura ListaEnlazada
 meth inic
 ListaEnlazada,inic
 end
 meth mover(X Y)
 {self paratodos(mover(X Y)) }
 end
 meth desplegar
 {self paratodos(desplegar) }
 end
end
```

En la figura 7.20 se muestra la herencia múltiple. Esta herencia es correcta porque las dos funcionalidades son completamente diferentes y no tienen interacciones indeseables. El método `inic` tiene el cuidado de inicializar la lista enlazada, y no necesita inicializar la figura. Como en todas las figuras, existe un método `mover` y un método `desplegar`. El método `mover(X Y)` mueve todas las figuras de la lista enlazada. El método `desplegar` permite visualizar todas las figuras de la lista enlazada.

¿Se logra ver la belleza de este diseño? Con él, una figura consiste de otras figuras, algunas de las cuales consisten de otras figuras, y así sucesivamente, en cualquier número de niveles. La estructura de la herencia garantiza que `mover` y `desplegar` funcionarán correctamente siempre. Este es un buen ejemplo de polimorfismo: todas las clases `FiguraCompuesta`, `Línea`, y `Círculo`, entienden los mensajes `mover(X Y)` y `desplegar`.

### Ejemplo de ejecución

Ejecutemos este ejemplo. Primero, configuramos una ventana con un campo de visualización gráfico:

```
declare
W=250 H=150 Lien
Wind={QTk.build td(title:"Paquete gráfico sencillo"
 canvas(width:W height:H bg:white
handle:Lien))}
{Wind show}
```

Utilizamos la herramienta gráfica `QTk`, la cual se explica en el capítulo 10 (en CTM). Por ahora simplemente supongamos que esto nos configura un lienzo, el cual es el espacio para dibujar nuestras figuras geométricas. Ahora, definimos una figura compuesta `F1` que contiene un triángulo y un círculo:



**Figura 7.21:** Dibujando con el paquete gráfico sencillo.

```
declare
F1={New FiguraCompuesta inic}
{F1 agr({New Línea inic(Lien 50 50 150 50)})})
{F1 agr({New Línea inic(Lien 150 50 100 125)})})
{F1 agr({New Línea inic(Lien 100 125 50 50)})})
{F1 agr({New Círculo inic(Lien 100 75 20)})})
```

Podemos visualizar la figura así:

```
{F1 desplegar}
```

Se despliega la figura una sola vez. Movamos la figura alrededor y visualicémosla cada vez:

```
for I in 1..10 do {F1 desplegar} {F1 mover(3 ~ 2)} end
```

En la figura 7.21 se muestra el resultado.

#### *Figuras compuestas con herencia sencilla*

En lugar de definir `FiguraCompuesta` con herencia múltiple, podemos definirla usando herencia sencilla colocando la lista de figuras en un atributo. El resultado es:

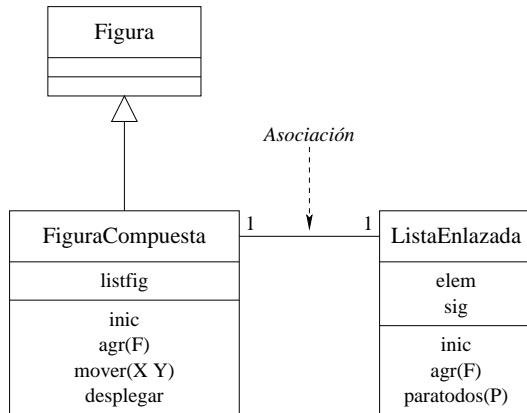


Figura 7.22: Diagrama de clases con una asociación.

```

class FiguraCompuesta from Figura
 attr listfig
 meth inic
 listfig:= {New ListaEnlazada inic}
 end
 meth agr(F)
 {@listfig agr(F)}
 end
 meth mover(X Y)
 {@listfig paratodos(mover(X Y)) }
 end
 meth desplegar
 {@listfig paratodos(desplegar) }
 end
end

```

En la figura 7.22 se muestra el diagrama de clases para este caso. El enlace entre **FiguraCompuesta** y **ListaEnlazada** se denomina una asociación, y representa una relación entre las dos clases. Los números que aparecen en los dos bordes del enlace son las cardinalidades; cada número dice cuántos elementos hay por una instancia en particular. El número 1 en el lado de la lista enlazada significa que existe exactamente una lista enlazada por cada figura compuesta, y de manera similar para el otro lado. El enlace de asociación es una especificación; no dice cómo se implementa. En nuestro caso, cada figura compuesta tiene un atributo `listfig` que referencia una lista enlazada.

El ejemplo de ejecución anterior también funcionará en el caso de herencia sencilla. ¿Cuáles son los compromisos que se hacen por utilizar herencia sencilla o herencia múltiple en este ejemplo? En ambos casos, las figuras que componen la figura compuesta están encapsuladas. La principal diferencia es que la herencia múltiple lleva las operaciones sobre listas enlazadas al mismo nivel de las de las figuras:

- Con herencia múltiple, una figura compuesta también es una lista enlazada. Todas las operaciones de la clase `ListaEnlazada` se pueden usar directamente sobre las figuras compuestas. Esto es importante si deseamos realizar cálculos de listas enlazadas con las figuras compuestas.
- Con herencia sencilla, una figura compuesta esconde completamente su estructura. Esto es importante si deseamos proteger las figuras compuestas de cualquier cálculo con operaciones diferentes a las definidas en su clase.

### ***Escalando el ejemplo***

Es sencillo extender este ejemplo a un paquete gráfico completo. Algunos de los cambios que se deberían hacer son:

- Se pueden definir muchas más figuras que hereden de `Figura`.
- En la implementación actual, las figuras están pegadas a su lienzo. Esto tiene la ventaja que permite que las figuras se puedan propagar sobre múltiples lienzos. Pero normalmente no necesitamos esta capacidad. Mejor aún, nos gustaría ser capaces de dibujar la misma figura sobre diferentes lienzos. Esto significa que el lienzo no debería ser un atributo de los objetos figura, sino que debería ser pasado como argumento al método `desplegar`.
- Se puede agregar una facilidad de registro. Es decir, debería ser posible registrar secuencias de instrucciones de dibujo, i.e., secuencias de invocaciones a las figuras, y manipular los registros como ciudadanos de primera clase. Estos registros representan dibujos en un alto nivel de abstracción, y pueden ser manipulados por la aplicación, almacenados en archivos, pasados a otras aplicaciones, etc.
- El método `desplegar` debería poder pasar parámetros arbitrarios desde el programa de aplicación, a través del paquete gráfico, al sistema gráfico subyacente. Cambiamos este método en la clase `Línea` así (análogamente en la clase `Círculo`):

```
meth desplegar(...)=M
 {@lienzo {Adjoin M create(line @x1 @y1 @x2 @y2)}}
end
```

La operación `Adjoin` combina dos registros, donde el segundo anula al primero en caso de conflictos. Esto permite pasar parámetros arbitrarios a las instrucciones de dibujo sobre el lienzo, a través de `desplegar`. Por ejemplo, la invocación `{F desplegar(fill:red width:3)}` dibuja una figura roja con un ancho de línea igual a 3.

#### **7.4.5. Reglas prácticas para la herencia múltiple**

La herencia múltiple es una técnica poderosa que tiene que ser usada con cuidado. Recomendamos utilizar la herencia múltiple de la manera siguiente:

- La herencia múltiple funciona bien cuando combinamos dos abstracciones completamente independientes. Por ejemplo, las figuras y las listas enlazadas no tienen

nada en común, luego se pueden combinar fructíferamente.

■ La herencia múltiple es mucho más difícil de usar correctamente cuando las abstracciones tienen algo en común. Por ejemplo crear una clase `TrabajoEstudio` a partir de las clases `Estudiante` y `Empleado` es dudoso, porque tanto los estudiantes como los empleados son seres humanos. Ellos pueden, de hecho, heredar de una clase `Persona` común. Aún si no tuvieran un ancestro compartido, puede haber problemas si tienen conceptos en común.

■ ¿Qué pasa cuando superclases hermanas comparten (directa o indirectamente) una clase ancestro común que especifica un objeto con estado (i.e., una clase con atributos)? Esto se conoce como el problema de la implementación compartida. Esto puede llevar a tener operaciones duplicadas sobre ancestros comunes. Esto sucede típicamente cuando se inicializa un objeto. El método de inicialización, normalmente, tiene que inicializar sus superclases, por tanto el ancestro común se inicializa dos veces. La única solución es entender cuidadosamente la jerarquía de herencia para evitar tal duplicación. Alternativamente, se podría solamente heredar de múltiples clases si éstas no comparten un ancestro en común, con estado.

■ Cuando ocurren conflictos de nombre, i.e., la misma etiqueta de método se utiliza para superclases adyacentes, entonces el programa debe definir un método local que anula los métodos que causan el conflicto. De otra forma, el sistema de objetos señala un mensaje de error. Una manera sencilla de evitar conflictos de nombre consiste en usar valores de tipo nombre como cabezas de los métodos. Esta es una técnica útil para unas clases, tales como las clases de mezclas anteriores (`MezclarReenvío` y `MezclarDelegación`), de las cuales se hereda frecuentemente por medio de la herencia múltiple.

#### 7.4.6. El propósito de los diagramas de clases

Los diagramas de clases son una herramienta excelente para visualizar la estructura de clases de una aplicación. Ellos hacen parte del corazón del enfoque UML para modelar aplicaciones orientadas a objetos, y como tal se usan ampliamente. Esta popularidad ha ocultado con frecuencia sus limitaciones. Los diagramas de clases tienen tres limitaciones claras:

- Ellos no especifican la funcionalidad de una clase. Por ejemplo, si los métodos de una clase hacen cumplir un invariante, entonces ese invariante no se muestra en el diagrama de clases.
- Ellos no modelan el comportamiento dinámico de la aplicación, i.e., su evolución en el tiempo. El comportamiento dinámico se da tanto a gran escala como a pequeña escala. Las aplicaciones pasan frecuentemente a través de diferentes fases, para las cuales son válidos diferentes diagramas de clases. Las aplicaciones son, con frecuencia, concurrentes, con partes independientes que interactúan en forma coordinada.
- Ellos sólo modelan un nivel en la jerarquía de componentes de la aplicación. Como

se explica en la sección 6.7, las aplicaciones bien estructuradas se descomponen jerárquicamente. Las clases y los objetos están cerca de la base de esta jerarquía. Un diagrama de clases explica la descomposición a ese nivel.

El enfoque UML reconoce estas limitaciones y provee herramientas que las alivian parcialmente, e.g., el diagrama de interacción y el diagrama de paquetes. Los diagramas de interacción modelan parte del comportamiento dinámico. Los diagramas de paquetes modelan componentes a un nivel más alto en la jerarquía que el nivel de las clases.

#### 7.4.7. Patrones de diseño

Cuando se diseña un sistema de *software*, es común encontrarse con los mismos problemas una y otra vez. El enfoque de patrones de diseño reconoce explícitamente esto y propone soluciones a estos problemas. Un patrón de diseño es una técnica que resuelve uno de estos problemas comunes. Este libro está lleno de patrones de diseño en ese sentido. Por ejemplo, estos dos:

- En programación declarativa, en la sección 3.4.2 se introduce la regla de construir una función siguiendo la estructura de un tipo. Un programa que utiliza una estructura de datos recursiva complicada puede, con frecuencia, escribirse fácilmente mirando el tipo de la estructura de datos. La estructura del programa es un espejo de la definición del tipo.
- En la sección 6.4.2 se introducen una serie de técnicas para construir una abstracción de datos segura empaquetando la funcionalidad en una capa segura. Estas técnicas son independientes de lo que hace la abstracción; ellas funcionan para cualquier abstracción.

Los patrones de diseño se popularizaron en primera instancia gracias a un libro muy influyente escrito por Gamma, Helm, Johnson, y Vlissides [54], en el cual se presenta un catálogo de patrones de diseño en POO y se explica cómo usarlos. El catálogo hace énfasis en patrones basados en herencia, utilizando el punto de vista de tipo. Miremos un patrón de diseño típico de este catálogo desde el punto de vista del programador, quien piensa en términos de modelos de computación.

#### *El patrón Composición*

El patrón Composición es un ejemplo típico de patrón de diseño. El propósito de Composición es construir jerarquías de objetos. Dada una clase que define una hoja, el patrón muestra cómo utilizar la herencia para definir árboles. En la figura 7.23, tomada de Gamma et al, [54] se muestra el diagrama de herencia del patrón Composición. La manera natural de usar este patrón consiste en conectarle una clase hoja inicial, `Hoja`. Luego el patrón define tanto la clase `Composición` como la clase `Componente`. `Componente` es una clase abstracta. En la jerarquía, un árbol es o una instancia de `Hoja` o de `Composición`.

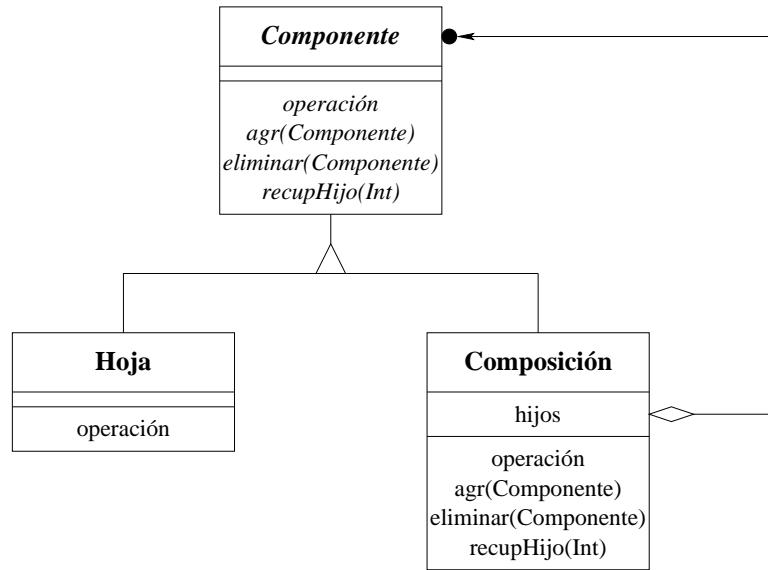


Figura 7.23: El patrón Composición.

Podemos usar el patrón Composición para definir figuras gráficas compuestas. En la sección 7.4.4 se resuelve el problema combinando una figura y una lista enlazada (con herencia sencilla o con herencia múltiple). El patrón Composición es una solución más abstracta, en el sentido que no supone que el agrupamiento se realice por medio de una lista enlazada. La clase **Composición** tiene las operaciones **agr** y **eliminar** pero no dice cómo se deben implementar. Entonces, ellas pueden implementarse con listas enlazadas, pero también pueden implementarse de manera diferente, e.g., como un diccionario o como una lista declarativa.

Dada una clase que define una hoja, el patrón Composición devuelve una clase que define el árbol. Puesto de esta forma, se parece mucho a la programación de alto orden: nos gustaría definir una función que acepta una clase y devuelve otra. Sin embargo, la mayoría de los lenguajes de programación, tales como C++ y Java, no permiten definir este tipo de funciones. Hay dos razones para ello. Primero, la mayoría de los lenguajes no consideran las clases como valores de primera clase. Segundo, la función define una nueva superclase de la clase de entrada. La mayoría de los lenguajes permiten definir subclases nuevas, pero no superclases nuevas. A pesar de estas limitaciones, aún nos gustaría utilizar el patrón Composición en nuestros programas.

La solución natural a este dilema consiste en considerar, ante todo, los patrones de diseño como una manera de organizar los pensamientos propios, sin que tengan, necesariamente, un soporte del lenguaje de programación. Un patrón puede existir únicamente en la mente del programador. Entonces, los patrones de diseño se pueden usar en lenguajes como C++ o Java, aún si no se pueden implementar

como abstracciones en esos lenguajes. Esto se puede hacer más fácil utilizando un preprocesador de código fuente. El programador puede entonces programar directamente con los patrones de diseño y el preprocesador genera el código fuente para el lenguaje objetivo.

### **Sopportando el patrón Composición**

El sistema de objetos de este capítulo nos permite soportar un patrón de agrupamiento, como el patrón Composición, desde el interior del modelo de computación. Implementemos una estructura de árbol cuyas hojas y nodos internos sean objetos. Las hojas son instancias de la clase `Hoja`, la cual se provee en tiempo de ejecución. Los nodos internos reenvían todas las invocaciones de métodos a las hojas en su subárbol. La forma más sencilla de implementar esto es definiendo una clase `Composición` para los nodos internos. Esta clase contiene una lista de sus hijos, los cuales pueden ser instancias de `Composición` o de `Hoja`. Suponemos que todas las instancias tienen el método de inicialización `inic` y que las instancias de `Composición` tienen el método `agr` para agregar un subárbol nuevo.

```
class Composición
 attr hijos
 meth inic
 hijos:=nil
 end
 meth agr(E)
 hijos:=E|@hijos
 end
 meth otherwise(M)
 for N in @hijos do {N M} end
 end
end
```

Si los nodos tienen muchos subnodos, entonces eliminar nodos sería inefficiente con esta implementación. En esta situación, una buena alternativa puede ser utilizar diccionarios en lugar de listas. A continuación se presenta un ejemplo de cómo construir un árbol:

```
N0={New Composición inic}
L1={New Hoja inic} {N0 agr(L1)}
L2={New Hoja inic} {N0 agr(L2)}
N3={New Composición inic} {N0 agr(N3)}
L4={New Hoja inic} {N0 agr(L4)}

L5={New Hoja inic} {N3 agr(L5)}
L6={New Hoja inic} {N3 agr(L6)}
L7={New Hoja inic} {N3 agr(L7)}
```

Si `Hoja` es la clase `Figura` de la sección 7.4.4, entonces `Composición` define las figuras compuestas.

**Creando únicamente árboles válidos** Esta implementación funciona para cualquier clase `Hoja` gracias al tipamiento dinámico. La desventaja de esta solución es que el sistema no obliga a que todas las hojas sean instancias de una misma clase. Agreguemos tal obligación a `Composición`:

```
class Composición
 attr hijos válido
 meth inic(Válido)
 hijos:=nil
 @válido=Válido
 end
 meth agr(E)
 if {Not {@válido E}} then raise nodoInválido end end
 hijos:=E|@hijos
 end
 meth otherwise(M)
 for N in @hijos do {N M} end
 end
end
```

Cuando se inicializa una instancia de `Composición`, se le pasa una función `válido`, la cual se liga al atributo `válido`. La función `válido` se utiliza para comprobar la validez de cada nodo que se agrega.

---

## 7.5. Relación con otros modelos de computación

El lenguaje no le impide anidar clases profundamente, pero el buen gusto debería hacerlo. . . . Un anidamiento a más de dos niveles lleva a un desastre de legibilidad y, probablemente, nunca se debería intentar.

— Adaptación libre de *The Java Programming Language*, 2nd edition, Ken Arnold y James Gosling (1998)

La programación orientada a objetos es una manera de estructurar programas, utilizada, con frecuencia, con estado explícito. Comparada con otros modelos de computación, se caracteriza ante todo por la utilización del polimorfismo y la herencia. El polimorfismo es una técnica general que no se limita a la POO, y por eso no la discutiremos más aquí. Más bien, nos enfocamos en la herencia y en cómo se relaciona ésta con otros conceptos de programación. Desde el punto de vista de múltiples modelos de computación, la herencia no es un concepto nuevo en el lenguaje núcleo, sino que surge de la forma en que se define la abstracción lingüística `class`. En esta sección se examina cómo la herencia se relaciona con otras técnicas de alto orden. En esta sección también se examina la afirmación común que predica como objetivo de diseño que “todo debe ser un objeto,” para descubrir lo que significa y hasta qué punto tiene sentido.

### 7.5.1. Programación basada en objetos y programación basada en componentes

La programación basada en objetos es la POO sin herencia. Esto es como la programación basada en componentes con sintaxis para las clases, lo cual nos ofrece una notación conveniente para encapsular el estado y definir múltiples operaciones sobre él. Sin herencia, la abstracción objeto se vuelve más sencilla. No existen problemas de anulación ni conflictos por la herencia múltiple. Las ligaduras estática y dinámica son idénticas.

### 7.5.2. Programación de alto orden

La programación orientada a objetos y la programación de alto orden están estrechamente relacionadas. Por ejemplo, examinemos el caso de una rutina de ordenamiento parametrizada por una función de orden. Se puede crear una rutina de ordenamiento nueva pasando como argumento una función de orden específica. En programación de alto orden, esto se hace de la manera siguiente:

```
proc {NuevaRutinaOrdenamiento Orden ?Ordenar}
 proc {Ordenar LEn LSal}
 % ... {Orden X Y} calcula el orden
 end
end
```

En POO, esto se puede escribir así:

```
class ClaseRutinaOrdenamiento
 attr ord
 meth inic(Orden)
 ord:=Orden
 end
 meth ordenar(LEn LSal)
 % ... {@ord orden(X Y $)} calcula el orden
 end
end
```

La relación de orden como tal se escribe así:

```
proc {Orden X Y ?B}
 B=(X<Y)
end
```

o así:

```
class ClaseOrden
 meth inic skip end
 meth orden(X Y B)
 B=(X<Y)
 end
end
```

Una instancia de la rutina de ordenamiento se crea así:

```
Ordenar={NuevaRutinaOrdenamiento Orden}
```

o así:

```
Ordenar={New ClaseRutinaOrdenamiento inic({New ClaseOrden inic})}
```

### ***Embellimientos añadidos por la programación orientada a objetos***

Es claro que los valores de tipo procedimiento y los objetos están estrechamente relacionados. Comparemos la programación de alto orden y la POO más cuidadosamente. La principal diferencia es que la POO “embellece” la programación de alto orden. La POO es una abstracción más rica que provee una colección de modismos adicionales que van más allá de la abstracción procedimental:

- El estado explícito se puede definir y usar fácilmente.
- Se pueden definir fácilmente múltiples métodos que comparten el mismo estado explícito. Al invocar un objeto, se invoca uno de los métodos.
- Se proveen las clases, las cuales definen un conjunto de métodos y se pueden instanciar. Cada instancia cuenta con un estado explícito fresco. Si los objetos son como procedimientos, entonces las clases son como procedimientos que devuelven procedimientos.
- Se provee la herencia, para definir conjuntos de métodos nuevos a partir de los ya existentes, extendiendo, modificando, y combinando los existentes. Las ligaduras dinámica y estática hacen particularmente rica esta capacidad.
- Se pueden definir diferentes grados de encapsulación entre las clases y los objetos. Los atributos y métodos pueden ser privados, públicos, protegidos, o tener algún otro grado de encapsulación, definido por el usuario.

Es importante notar que estos mecanismos no proveen ninguna capacidad fundamentalmente nueva. Todos ellos se pueden definir en términos de programación de alto orden, estado explícito, y valores de tipo nombre. Por el otro lado, los mecanismos son modismos útiles que llevan a un estilo de programación frecuentemente conveniente.

La programación orientada a objetos es una abstracción que provee una notación rica para usar cualesquiera o todos estos mecanismos juntos, cuando se necesite. Esta riqueza es una espada de doble filo. Por un lado, la abstracción es particularmente útil para muchas tareas de programación. Por otro lado, la abstracción tiene una semántica compleja y es difícil razonar sobre ella. Por esta razón, recomendamos usar la orientación a objetos sólo en aquellos casos donde simplifica de manera significativa la estructura del programa, e.g., donde exista una clara necesidad de la herencia: el programa contiene un conjunto de abstracciones de datos estrechamente relacionadas. En otros casos, recomendamos usar técnicas de programación más sencillas.

### ***Limitaciones comunes***

El sistema de objetos definido en este capítulo es particularmente próximo a la programación de alto orden. No todos los sistemas de objetos son tan cercanos. En particular, las características siguientes están ausentes con frecuencia o son incómodas de usar:

- Las clases como valores. Ellas pueden ser creadas en tiempo de ejecución, pasadas como argumentos, y almacenadas en estructuras de datos.
- Alcance léxico completo. Eso significa que el lenguaje soporta valores de tipo procedimiento con referencias externas. Esto permite que una clase se pueda definir dentro del alcance de un procedimiento o de otra clase. Tanto Smalltalk-80 como Java soportan valores de tipo procedimiento (con algunas restricciones). En Java, los valores de tipo procedimiento son instancias de clases internas (i.e., clases anidadas). Además, son bastante pródigos en texto debido a la sintaxis de clase (ver el epígrafe al principio de esta sección).
- Mensajes de primera clase. Normalmente, las etiquetas de los mensajes y los métodos tienen que conocerse en tiempo de compilación. La manera más general de eliminar esta restricción es permitir que los mensajes sean valores del lenguaje, de manera que se puedan calcular en tiempo de ejecución. Tanto Smalltalk-80 como Java proveen esta capacidad, aunque es más pródiga en texto que las invocaciones normales de métodos (estática). Por ejemplo, esta es una manera genérica de agregar “procesamiento de mensajes en lotes” a una clase C:

```
class ProcPorLotes
 meth nil skip end
 meth `|`(M Ms) {self M} {self Ms} end
end
```

Mezclar cualquier otra clase con la clase `ProcPorLotes` le agrega a la clase original la capacidad de procesar mensajes por lotes:

```
C={New class $ from Contador ProcPorLotes end inic(0)}
{C [inc(2) browse inc(3) inc(4)]}
```

En la sección 7.8.5 se presenta otra forma de agregar el procesamiento de mensajes por lotes.

Algunos lenguajes orientados a objetos, e.g., C++, no soportan la programación de alto orden completamente porque no pueden definir valores de tipo procedimiento con alcance léxico en tiempo de ejecución (tal como se explicó en la sección 3.6.1). En estos lenguajes, muchas de las capacidades de la programación de alto orden se obtienen por medio de la encapsulación y la herencia, más un pequeño esfuerzo por parte del programador:

- Un valor de tipo procedimiento se puede codificar como un objeto. Los atributos del objeto representan las referencias externas del procedimiento y los argumentos del método son los argumentos del procedimiento. Cuando se crea el objeto, sus atributos se inicializan con las referencias externas. El objeto se puede pasar e

invocar igual que un valor de tipo procedimiento. Con un poco de disciplina por parte del programador, esto permite programar con valores de tipo procedimiento, y por tanto, se logra una verdadera programación de alto orden.

- Un procedimiento genérico se puede codificar como una clase abstracta. Un procedimiento genérico es aquél que recibe procedimientos como argumentos y devuelve un procedimiento específico. Por ejemplo, una rutina de ordenamiento genérica puede tomar una operación de comparación para un tipo dado y devolver una rutina que ordena arreglos de elementos de ese tipo. Una clase abstracta es una clase con métodos indefinidos. Los métodos se definen en las subclases.

### *Codificando valores de tipo procedimiento como objetos*

Presentamos un ejemplo de cómo codificar un valor de tipo procedimiento en un lenguaje orientado a objetos típico. Suponga que tenemos una declaración cualquiera  $\langle \text{decl} \rangle$ . Con abstracción procedural, podemos definir un procedimiento **proc**  $\{P\}$   $\langle \text{decl} \rangle$  **end** y ejecutarlo más tarde como  $\{P\}$ . Para codificar esto en nuestro sistemas de objetos tenemos que conocer las referencias externas de  $\langle \text{decl} \rangle$ . Suponga que son  $x$  y  $y$ . Definimos la clase siguiente:

```
class Proc
 attr x y
 meth inic(X Y) @x=X @y=Y end
 meth aplicar X=@x Y=@y in <decl> end
end
```

Las referencias externas se representan por medio de los atributos sin estado  $x$  y  $y$ . Definimos  $P$  haciendo  $P=\{\text{New Proc inic}(X Y)\}$  y lo invocamos con  $\{P\} \text{ aplicar}$ . Esta codificación se puede usar en cualquier lenguaje orientado a objetos. Con ella, podemos usar casi todas las técnicas de programación de alto orden del libro. Sin embargo, tiene dos desventajas con respecto a los procedimientos: es más engorrosa de escribir, y las referencias externas tienen que escribirse explícitamente.

#### 7.5.3. Descomposición funcional versus descomposición por tipos

¿Cómo organizamos una abstracción de datos, basada en un tipo  $\langle T \rangle$  con subtipos  $\langle T \rangle_1$ ,  $\langle T \rangle_2$ ,  $\langle T \rangle_3$ , que incluye un conjunto de operaciones  $\langle F \rangle_1, \dots, \langle F \rangle_n$ ? En programación declarativa, en la sección 3.4.2 se recomienda construir funciones de acuerdo a la definición del tipo. En POO, en la sección 7.4.2 se recomienda construir jerarquías de herencia de manera similar, de acuerdo, también, a la definición del tipo. En ambas secciones se presentan ejemplos basados en listas. En la figura 7.24 se presenta un esquema general, aproximado, comparando los dos enfoques. Como resultado se obtienen programas con estructuras totalmente diferentes, las cuales denominamos descomposición funcional y descomposición por tipos. Estas estructuras corresponden a los dos estilos de abstracción de datos, TAD y objeto, introducidos en la sección 6.4. En la descomposición funcional, cada función es un todo auto contenido, pero los tipos se dispersan en todas las funciones.

### Definición de tipo

$\langle T \rangle ::= \langle T \rangle_1 \mid \langle T \rangle_2 \mid \langle T \rangle_3$

### Operaciones

$\langle F \rangle_1, \langle F \rangle_2, \dots, \langle F \rangle_n$

### Descomposición funcional

```
fun {<F>1 <T> ...}
 case <T>
 of <T>1 then
 ...
 [] <T>2 then
 ...
 [] <T>3 then
 ...
 end
end

fun {<F>2 <T> ...}
 ...
end

...
fun {<F>n <T> ...}
 ...
end
```

### Descomposición por tipos

```
class <T> ... end

class <T>1 from <T>
 ...
 meth <F>1(...)
 ...
end
meth <F>2(...)
 ...
end
...
meth <F>n(...)
 ...
end

class <T>2 from <T> ... end
class <T>3 from <T> ... end
```

Figura 7.24: Descomposición funcional versus descomposición por tipos.

En la descomposición por tipos, cada tipo es un todo auto contenido, pero las definiciones de función se dispersan por todos los tipos. ¿Cuál enfoque es mejor? Resulta que cada uno tiene sus usos:

- En la descomposición funcional, uno puede modificar una función o agregar una nueva sin cambiar las otras definiciones de función. Sin embargo, cambiar o agregar un tipo requiere de la modificación de todas las definiciones de función.
- En la descomposición por tipos, uno puede modificar un tipo (i.e., una clase) o agregar un nuevo tipo (incluso por medio de la herencia) sin cambiar las otras definiciones de tipos. Sin embargo, cambiar o agregar una función requiere la modificación de todas las definiciones de clase.

Al momento de diseñar un programa, es bueno preguntarse a sí mismo qué tipo de modificación es más importante. Si el tipo es relativamente sencillo y existe un gran número de operaciones, entonces el enfoque funcional, normalmente, es más

claro. Si el tipo es complejo, con un número relativamente pequeño de operaciones, entonces el enfoque de tipos puede ser más claro. Existen técnicas que combinan algunas de las ventajas de ambos enfoques. Vea, e.g., [189], el cual explica algunas de estas técnicas y cómo usarlas para construir compiladores extensibles.

#### 7.5.4. ¿Todo debería ser un objeto?

En discusiones sobre la POO, se invoca con frecuencia el principio que “todo debería ser un objeto.” Analicemos este principio para descubrir qué es realmente lo que se está tratando de decir.

##### *Objetos fuertes*

Una forma sensible de enunciar el principio es “todas las entidades del lenguaje deben ser instancias de abstracciones de datos con tantas propiedades genéricas como sea posible.” En su forma extrema, esto implica seis propiedades: todas las entidades del lenguaje se deberían definir con el estilo objeto (ver sección 6.4), en términos de clases que puedan ser instanciadas, extensibles con herencia, con una identidad única, con un estado encapsulado, y tal que se pueda acceder a ellas por medio de una sintaxis uniforme. La palabra “objeto” se utiliza algunas veces para entidades con todas esta propiedades. Para evitar confusiones, nosotros las denominaremos objetos fuertes. Un lenguaje orientado a objetos se denomina puro si todas sus entidades son objetos fuertes.

El deseo de pureza puede llevar a buenas cosas. Por ejemplo, muchos lenguajes tienen el concepto de “excepción” para manejar eventos anormales que se presentan durante la ejecución. Puede ser bastante conveniente que las excepciones sean objetos dentro de una jerarquía de herencia. Esto permite clasificarlas en categorías diferentes, capturarlas solamente si son de una clase dada (o de sus subclases) y, posiblemente, modificarlas (añadir información) si son excepciones con estado.

Smalltalk-80 es un buen ejemplo de lenguaje para el cual la pureza fue un objetivo explícito de diseño [56, 78]. Todos los tipos de datos en Smalltalk, incluyendo los más sencillos como los enteros, son objetos. Sin embargo, en Smalltalk no todo es un objeto; el estilo TAD se utiliza para operaciones primitivas de tipos básicos como los enteros y existe un concepto denominado bloque que es un valor de tipo procedimiento para construir abstracciones de control.

En la mayoría de los lenguajes, algunas entidades no son objetos fuertes. Presentamos algunos ejemplos en Java. Un entero en Java es un valor puro en el estilo TAD; no está definido por una clase y no encapsula un estado. Un objeto en Java sólo puede tener atributos con alcance `final`, lo cual significa que son atributos sin estado. Un arreglo en Java no se puede extender con herencia. Los arreglos se comportan como si fueran definidos en una clase final. En resumen, Java es orientado a objetos, pero no es puro.

¿Debería un lenguaje tener objetos fuertes solamente? Es claro que la respuesta es no, por muchas razones. Primero, el estilo TAD es esencial en algunas oportunidades.

dades, tal como lo explicamos en la sección 6.4.3. Segundo, las entidades sin estado pueden jugar un papel importante. Con ellas, las poderosas técnicas de razonamiento de la programación declarativa se vuelven posibles. Por esta razón, muchos de los diseños de lenguajes las permiten. Citamos Objective Caml [31], el cual tiene un núcleo funcional, y Java [10], el cual tiene objetos inmutables. Además, las entidades sin estado son esenciales para lograr una programación distribuida transparente en la práctica (ver capítulo 11 (en CTM)). Tercero, todas las entidades no necesitan una identidad única. Por ejemplo, las entidades estructuradas como las tuplas en una base de datos se identifican por sus contenidos, no por sus nombres. Cuarto, la sencillez de una sintaxis uniforme es ilusoria, como lo explicamos más abajo.

Al parecer, eliminamos cada propiedad una por una. Nos quedamos con dos principios: todas las entidades del lenguaje deben ser instancias de abstracciones de datos y se debe explotar la uniformidad entre abstracciones de datos cuando sea razonable. Algunas abstracciones de datos tendrán todas las propiedades de los objetos fuertes; otras tendrán solamente algunas de esas propiedades, pero también tendrán algunas otras propiedades completamente diferentes. Estos principios son consistentes con la utilización de múltiples modelos de computación, práctica por la que abogamos en este libro. La construcción de un sistema consiste ante todo en el diseño de abstracciones y su implementación como datos o abstracciones de control.

### *Objetos y complejidad de un programa*

Dado un objeto específico, ¿Cómo se puede predecir su comportamiento en el futuro? Eso depende de dos factores:

1. Su estado interno, el cual depende, potencialmente, de todas las invocaciones anteriores. Estas invocaciones se pueden realizar desde muchas partes del programa.
2. Su definición textual, la cual depende de todas las clases de las cuales hereda. Estas clases pueden estar definidas en muchos lugares en el texto del programa.

Vemos entonces que la semántica de un objeto está esparcida tanto en el tiempo como en el espacio. Esto es lo que hace que un objeto sea más difícil de entender que una función. La semántica de una función se concentra en un solo lugar, a saber, la definición textual de la función. La función no tiene una historia; sólo depende de su definición y de sus argumentos.

Presentamos un ejemplo que muestra por qué es más difícil programar con objetos cuando tienen estado. Suponga que estamos haciendo cálculos aritméticos con el estándar de punto fijo de la IEEE y que hemos implementando el estándar por completo. Esto quiere decir, e.g., que podemos cambiar el modo de redondeo de las operaciones aritméticas durante la ejecución del programa (redondear al par más cercano, redondear hacia arriba, redondear hacia abajo, etc.). Si no utilizamos esta capacidad con cuidado, entonces no tendremos ni idea cuál será el resultado de una suma  $x+y$  a menos que hayamos hecho un seguimiento a toda la ejecución. Cualquier parte del programa puede cambiar el modo de redondeo. Esto puede causar estragos

en los métodos numéricos, los cuales dependen de redondeos predecibles para obtener buenos resultados.

Para evitar este problema, tanto como sea posible, el lenguaje no debe favorecer el estado explícito ni la herencia. Es decir, no usar esos mecanismos debe ser sencillo. En el caso de la herencia, esto casi nunca es un problema, pues siempre es más difícil usarla que evitarla. En el caso del estado explícito, eso depende del lenguaje. En el modelo orientado a objetos de este capítulo, es ligeramente más fácil definir funciones (sin estado) que definir objetos (con estado). Los objetos tienen que ser definidos como instancias de clases, las cuales tienen que ser definidas con una declaración **class ... end** que empaqueta una o más declaraciones **meth ... end**. Las funciones sólo requieren la declaración **fun ... end**. En los lenguajes orientados a objetos populares, el estado explícito es, desafortunadamente, casi siempre el mecanismo por defecto, y las funciones son, normalmente, sintácticamente engorrosas. Por ejemplo, en Java no existe un soporte sintáctico para las funciones y los atributos de los objetos son con estado a menos que se declaren con alcance **final**. En Smalltalk, todos los atributos son con estado, pero los valores de tipo función se pueden definir fácilmente.

### *Sintaxis uniforme de objetos*

La sintaxis de un lenguaje debe ayudar y no entorpecer las actividades de diseño, escritura, y razonamiento sobre los programas, por parte de los programadores. Un principio importante en diseño de sintaxis es que la forma debe reflejar el contenido. Las diferencias en la semántica deben ser visibles como diferencias en sintaxis y viceversa. Por ejemplo, el ciclo “while,” como se usa en muchos lenguajes, tiene una sintaxis similar a **while <expr> do <decl>**. El hecho de escribir **<expr>** antes que **<decl>**, es un reflejo sintáctico del hecho de que la condición **<expr>** se evalúa antes de ejecutar **<decl>**. Si **<expr>** es falsa, entonces **<decl>** no se ejecuta de ninguna manera. El lenguaje Cobol hace las cosas de manera diferente. Cobol cuenta con un ciclo “perform”, el cual se escribe **perform <decl> until <expr>**. La sintaxis es engañosa pues **<expr>** se comprueba antes de **<decl>**, aunque se escriba después de **<decl>**. La semántica de este ciclo es **while not <expr> do <decl>**.

¿Todas las operaciones sobre las entidades de un lenguaje deberían tener la misma sintaxis? Aunque con esto se logre facilitar la transformación de los programas (e.g., con los macros de Lisp [59]), no se mejora necesariamente la legibilidad. Por ejemplo, Scheme tiene una sintaxis uniforme lo cual no necesariamente lo hace más legible. Para nosotros, una sintaxis uniforme solamente desplaza la riqueza del lenguaje hacia los nombres de los objetos y las clases. Esto agrega una segunda capa de sintaxis, haciendo que el lenguaje sea más pródigo en texto. Miremos un ejemplo tomado de los lenguajes simbólicos de programación. Los valores sin estado se pueden crear de manera natural, con una sintaxis compacta. Un valor de tipo lista se puede crear con sólo mencionarlo, e.g.:

```
LV=[1 2 3]
```

Esta es aproximadamente la sintaxis usada por los lenguajes que soportan la programación simbólica, tales como Lisp, Prolog, Haskell, Erlang, y sus relacionados. Esto contrasta con la utilización de una sintaxis uniforme de objetos:

```
ClaseLista *lv= new ClaseCons(1, new ClaseCons(2,
new ClaseCons(3, new ClaseNil())));
```

Esta es sintaxis de C++, la cual es similar a la sintaxis de Java. Para descomponer un valor de tipo lista, existe otra notación natural que utiliza reconocimiento de patrones:

```
case LV of X|LV2 then ... end
```

El reconocimiento de patrones se usa comúnmente en los lenguajes simbólicos. Esto también es engorroso de hacer con una sintaxis uniforme de objetos. Existe un incremento adicional de texto cuando se escriben programas concurrentes en la sintaxis de objetos. Esto se debe a que la sintaxis uniforme requiere de sincronización explícita. Esto no sucede así para la sintaxis **case** de arriba, la cual es suficiente para la programación concurrente si el modelo de computación realiza implícitamente la sincronización por flujo de datos.

Otro ejemplo es la herramienta QTk, del capítulo 10 (en CTM), para programar interfaces gráficas de usuario, la cual depende fuertemente de los valores de tipo registro y su sintaxis concisa. Inspirado en esta herramienta, Christophe Ponsard escribió un prototipo en Java de una herramienta similar. La versión de Java es más engorrosa de usar que la versión de Oz, ante todo porque no tiene soporte sintáctico para los valores de tipo registro. Desafortunadamente, este incremento de texto es una propiedad inherente a Java. No existe una forma sencilla de evitarlo.

---

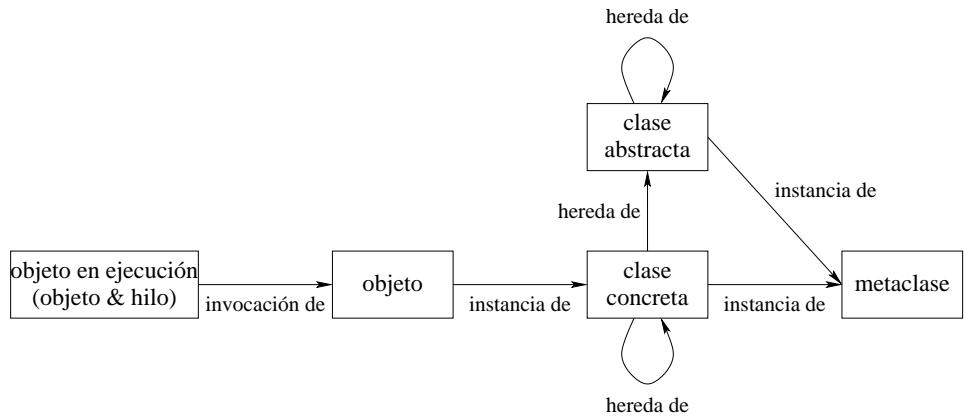
## 7.6. Implementando el sistema de objetos

Todo el sistema de objetos se puede implementar de manera directa a partir del modelo de computación declarativo con estado. En particular, las características principales se obtienen de la combinación de programación de alto orden con estado explícito. Con esta construcción, se entenderán los objetos y las clases por completo.

Aunque la construcción presentada en esta sección funciona bien y es razonablemente eficiente, un implementación real añadirá optimizaciones para hacerlo aún mejor. Por ejemplo, en una implementación real se puede lograr que la invocación a un objeto sea tan rápida como la invocación de un procedimiento. En esta sección no se presentan esas optimizaciones.

### 7.6.1. Diagrama de abstracción

El primer paso para comprender cómo construir un sistema de objetos consiste en entender cómo se relacionan las diferentes partes que lo componen. La programación orientada a objetos define una jerarquía de abstracciones relacionadas entre



**Figura 7.25:** Abstracciones en programación orientada a objetos.

sí por una especie de relación “especificación-implementación.” Existen muchas variaciones sobre esta jerarquía. Presentamos una sencilla que contiene la mayoría de las ideas principales. Estas son las abstracciones, en orden, de la más concreta a la más abstracta:

- *Objeto en ejecución.* Un objeto en ejecución es una invocación activa de un objeto, en la cual se asocia un hilo a un objeto. El hilo contiene un conjunto de marcos de ambiente (la parte de la pila del hilo que se crea mientras se ejecuta el objeto) así como un objeto.
- *Objeto.* Un objeto es un procedimiento que encapsula un estado explícito (una celda) y un conjunto de procedimientos que referencian el estado.
- *Clase.* Una clase es un registro empaquetado que encapsula un conjunto de procedimientos cuyos nombres son literales y un conjunto de atributos, los cuales son simplemente literales. Los procedimientos se denominan métodos. Los métodos toman un estado, representado por los atributos, como argumento y modifican ese estado. Los métodos pueden invocar a los otros sólo indirectamente, a través de los literales, es decir, de los nombres de los métodos. Con frecuencia, la siguiente distinción es útil:
  - *Clase abstracta.* Una clase abstracta es una clase en la cual algunos métodos tienen nombre pero no tienen ninguna definición en la clase.
  - *Clase concreta.* Una clase concreta es una clase en la cual todos los métodos están definidos.

Si el lenguaje soporta mensajes de primera clase, entonces se pueden hacer invocaciones de la forma `{obj M}`, donde `M` se calcula en tiempo de ejecución. Si tal invocación existe en el programa, entonces la distinción entre clase concreta y abstracta desaparece en el programa (aunque exista aún conceptualmente). Si `M` no existe en `obj`, entonces la ejecución de la invocación `{obj M}` lanza una excepción.

```
class Contador
 attr val
 meth inic(Valor)
 val:=Valor
 end
 meth inc(Valor)
 val:=@val+Valor
 end
 meth browse
 {Browse @val}
 end
end
```

Figura 7.26: Un ejemplo de clase: Contador (de nuevo).

- *Metaclase.* Una metaclase es una clase con un conjunto particular de métodos que corresponden a las operaciones básicas de una clase, e.g.: creación de un objeto, política de herencia (cuáles métodos heredar), invocación de un método, terminación de un método, escogencia del método a invocar, asignación de atributos, acceso a los atributos, invocación del **self**. Escribir estos métodos permite personalizar la semántica de los objetos.

En la figura 7.25 se muestra cómo se relacionan estos conceptos. Existen tres relaciones, “invocación de,” “instancia de,” y “hereda de.” Estas relaciones significan, intuitivamente, lo siguiente:

- Un objeto en ejecución se crea cuando un hilo invoca a un objeto. El objeto en ejecución existe hasta que la ejecución del hilo termine. Pueden existir simultáneamente múltiples invocaciones del mismo objeto.
- Un objeto se puede crear como una instancia de una clase. Si el sistema de objetos distingue entre clases concretas y abstractas, entonces lo normal es que sólo se pueden crear instancias a partir de las clases concretas. El objeto existe para siempre.<sup>4</sup> El objeto encapsula una celda que fue creada especialmente para él. Pueden existir simultáneamente múltiples instancias de la misma clase.
- Una clase se puede crear heredando de una lista de otras clases. La clase nueva existe para siempre. La herencia toma un conjunto de métodos y una lista de clases y devuelve una clase nueva con un conjunto nuevo de métodos. Pueden existir simultáneamente múltiples clases que hereden de la misma clase. Si una clase puede heredar de varias clases, entonces tenemos herencia múltiple. Sino, si una clase solamente puede heredar de una clase, entonces tenemos herencia sencilla.

---

4. En la práctica, hasta que el programa en ejecución pierda todas las referencias a él. En este momento, el recolector de basura puede recuperar esa memoria, y se puede realizar una última acción de finalización si fuera necesario.

```

declare Contador
local
 Atrbs = [val]
 TablaDeMétodos = m(browse:MiBrowse inic:Inic inc:Inc)
 proc {Inic M S Self}
 inic(Valor)=M
 in
 (S.val):=Valor
 end
 proc {Inc M S Self}
 X
 inc(Valor)=M
 in
 X=@(S.val) (S.val):=X+Valor
 end
 proc {MiBrowse M S Self}
 browse=M
 {Browse @(S.val)}
 end
in
 Contador = {Envolver c(métodos:TablaDeMétodos atrbs:Atrbs)}
end

```

**Figura 7.27:** Un ejemplo de construcción de una clase.

- Se puede crear una clase como una instancia de una metaclasa. La clase nueva existe para siempre. Las operaciones básicas de la clase se definen por los métodos particulares de la metaclasa. Pueden existir simultáneamente múltiples instancias de la misma metaclasa.

### 7.6.2. Implementando clases

Primero explicamos la abstracción lingüística **class**. La clase Contador de la figura 7.26 se traduce internamente en la definición de la figura 7.27. Esta figura muestra que una clase es, sencillamente, un valor de tipo registro, protegido contra espías gracias al empaquetamiento logrado con **Envolver** (ver sección 3.7.5). (Más adelante, cuando la clase se utilice para crear objetos, ésta será desempacada con la correspondiente función **Desenvolver**.) El registro de la clase contiene:

- Un conjunto de métodos en una tabla de métodos. Cada método es un procedimiento de tres argumentos que toma un mensaje **M**, el cual siempre es un registro, un parámetro adicional **S** que representa el estado del objeto actual, y **Self**, que es una referencia al objeto mismo.
- Un conjunto de nombres de atributos, conteniendo los atributos que cada instancia de la clase (objeto) poseerá. Cada atributo es una celda con estado que se puede acceder por el nombre del atributo, el cual es un átomo o un nombre de Oz.

```
fun {New ClaseEmp MétodoInicial}
 Estado Obj Clase={Desenvolver ClaseEmp}
in
 Estado={MakeRecord s Clase.atrbs}
 {Record.forAll Estado proc {$ A} {NewCell _ A} end}
 proc {Obj M}
 {Clase.métodos.{Label M} M Estado Obj}
 end
 {Obj MétodoInicial}
 Obj
end
```

**Figura 7.28:** Un ejemplo de construcción de un objeto.

Este ejemplo está ligeramente simplificado pues no se muestra cómo soportar ligadura estática (ver ejercicios, sección 7.9). La clase Contador tiene un solo atributo al que se accede por medio del átomo val. Esta clase tiene una tabla de métodos a la que se accede a través de los campos browse, inic, e inc. Como se puede ver, el método inic asigna el valor valor al atributo val, el método inc incrementa el atributo val, y el método browse despliega el valor actual de val.

### 7.6.3. Implementando objetos

Podemos usar la clase Contador para crear objetos. En la figura 7.28 se muestra una función genérica New que crea un objeto a partir de cualquier clase. Esta función comienza por desempaquetar la clase. Luego crea un estado del objeto, el cual es un registro, a partir de los (nombres de los) atributos de la clase. Como los atributos solamente se conocen, en general, en tiempo de ejecución, se realiza una creación dinámica del registro con MakeRecord, y se inicializa cada campo de este registro con una celda nueva (cuyo valor es no-ligado, inicialmente). Se utiliza el iterador Record.forAll para iterar sobre todos los campos del registro.

El objeto Obj devuelto por New es un procedimiento de un argumento. Una vez se invoca como {Obj M}, él busca e invoca el procedimiento correspondiente a M en la tabla de métodos. Debido al alcance léxico, el estado del objeto sólo es visible dentro de Obj. Se puede decir que Obj es un procedimiento que encapsula el estado.

La definición de la figura 7.28 funciona correctamente, pero puede no ser la manera más eficiente de implementar un objeto. El sistema actual puede usar una implementación diferente, más eficiente, siempre que se comporte de la misma manera. Por ejemplo, el sistema Mozart utiliza una implementación en la cual las invocaciones de los objetos son casi tan eficientes como las invocaciones a los procedimientos [65, 66].

El postre se prueba comiéndolo. Verifiquemos entonces que la clase funciona como se ha afirmado. Creemos ahora la clase Contador y ensayemos New así:

```

fun {HeredarDe C1 C2 C3}
 c(métodos:M1 atrbs:A1)={Desenvolver C1}
 c(métodos:M2 atrbs:A2)={Desenvolver C2}
 c(métodos:M3 atrbs:A3)={Desenvolver C3}
 MA1={Arity M1}
 MA2={Arity M2}
 MA3={Arity M3}
 MetEnConf1={Minus {Inter MA2 MA3} MA1}
 AtrEnConf1={Minus {Inter A2 A3} A1}
in
 if MetEnConf1\=nil then
 raise herenciaillegal(metEnConf1:MetEnConf1) end
 end
 if AtrEnConf1\=nil then
 raise herenciaillegal(atrEnConf1:AtrEnConf1) end
 end
 {Envolver c(métodos:{Adjoin {Adjoin M2 M3} M1}
 atrbs:{Union {Union A2 A3} A1})}
end

```

**Figura 7.29:** Implementando la herencia.

```

C={New Contador inic(0)}
{C inc(6)} {C inc(6)}
{C browse}

```

Esto se comporta exactamente de la misma manera que el ejemplo de la sección 7.2.1.

#### 7.6.4. Implementando la herencia

La herencia calcula un nuevo registro de clase a partir de los registros de clases existentes, combinados de acuerdo a las reglas de herencia presentadas en la sección 7.3.1. La herencia se puede definir por medio de la función `HeredarDe`, donde la invocación `C={HeredarDe C1 C2 C3}` devuelve un nuevo registro de clase cuya definición de base es `C1` y hereda de `C2` y `C3`. Esta función corresponde a la siguiente sintaxis de clase:

```

class C from C2 C3
 ...
 ... % La clase base es C1
end

```

En la figura 7.29 se muestra la definición de `HeredarDe`, la cual utiliza las operaciones de conjunto del módulo `Set`, el cual se puede encontrar en el sitio Web del libro. Lo primero que hace `HeredarDe` es comprobar si existen conflictos entre las tablas de métodos y las listas de atributos. Si se encuentra un método o un atributo duplicado entre `C2` y `C3` que no sea anulado por `C1`, entonces se lanza una excepción. Luego, `HeredarDe` construye la tabla de métodos y la lista de atributos nuevas. La anulación se maneja apropiadamente por medio de la función `Adjoin` sobre las

*Programación orientada a objetos*

tablas de métodos (ver apéndice B.3.2). La definición es ligeramente simplificada pues no maneja ligadura estática y supone que hay exáctamente dos superclases.

---

## 7.7. El lenguaje Java (parte secuencial)

Java es un lenguaje orientado a objetos, concurrente, con una sintaxis parecida a la de C++. En esta sección se presenta una breve introducción a la parte secuencial de Java. Explicamos cómo escribir un programa sencillo, cómo definir clases, y cómo usar la herencia. Dejamos el tema de la concurrencia en Java para el capítulo 8. No decimos nada sobre el paquete `reflection`, el cual permite hacer mucho más de lo que el sistema de objetos de este capítulo deja hacer (aunque de manera más pródiga en texto).

Java es un lenguaje orientado a objetos casi puro, i.e., casi todo es un objeto. Solamente un pequeño conjunto de tipos primitivos, a saber, enteros, flotantes, booleanos, y caracteres, utilizan el estilo TAD y, por tanto, no son objetos. Java es un lenguaje relativamente limpio, con una sintaxis relativamente sencilla. A pesar de su similaridad sintáctica, existe una diferencia importante en la filosofía del lenguaje entre Java y C++ [10, 166]. C++ provee acceso a la representación de máquina de los datos y una traducción directa a instrucciones de máquina; también permite una administración manual de la memoria. Gracias a estas propiedades, C++ es con frecuencia conveniente como reemplazo del lenguaje ensamblador. En contraste, Java oculta la representación de los datos y realiza una administración automática de la memoria. Java soporta programación distribuida sobre múltiples plataformas. Además tiene un sistema de objetos más sofisticado. Estas propiedades sitúan a Java mejor para el desarrollo de aplicaciones de propósito general.

### 7.7.1. Modelo de computación

Java consiste de programación orientada a objetos estáticamente tipada, con clases, objetos pasivos, e hilos. El modelo de computación de Java es cercano al modelo concurrente con estado compartido (el cual cuenta tanto con hilos como con celdas), sin las variables de flujo de datos, los disparadores, y los nombres. En la sección 8.6 se presenta una introducción a la parte concurrente de Java. El paso de parámetros se realiza por valor, tanto para tipos primitivos como para referencias a objetos. A las variables recién declaradas se les coloca un valor inicial por defecto dependiendo de su tipo. Existe soporte para la asignación única: las variables y los atributos de los objetos se pueden declarar como `final`, lo cual significa que la variable sólo puede ser asignada una sola vez. Las variables finales deben ser asignadas antes de que sean usadas.

Java introduce su propia terminología para algunos conceptos. Las clases contienen campos (atributos, en nuestra terminología), métodos, otras clases, o interfaces, los cuales se conocen colectivamente como los miembros de la clase. Las variables son campos, o variables locales (declaradas en bloques locales a los métodos) o parámetros de métodos. Las variables se declaran dando su tipo, identificador, y un conjunto opcional de modificadores (e.g., `final`). El concepto `self` se denomina `this`.

## *Interfaces*

Java tiene una solución elegante para los problemas de la herencia múltiple (ver secciones 7.4.4 y 7.4.5). Java introduce el concepto de interfaz, el cual se parece, sintácticamente, a una clase que sólo tiene declaraciones de métodos. Una interfaz no tiene implementación. Una clase puede implementar una interfaz lo cual sencillamente significa que la clase define todos los métodos que hay en la interfaz. Java soporta herencia sencilla para las clases, evitando así los problemas de la herencia múltiple. Pero, para preservar las ventajas de la herencia múltiple, Java soporta herencia múltiple para las interfaces.

Java soporta la programación de alto orden de forma trivial por medio de la codificación presentada en la sección 7.5.2. Además de esto, Java cuenta con un soporte más directo para la programación de alto orden a través de las clases internas. Una clase interna es una definición de clase anidada dentro de otra clase o dentro de un bloque de código (tal como un cuerpo de método). Una instancia de una clase interna se puede pasar afuera del cuerpo del método o del bloque de código. Una clase interna puede tener referencias externas, pero existe una restricción si llega a estar anidada en un bloque de código: en ese caso no puede tener referencias a variables que no sean finales. Podríamos decir que una instancia de una clase interna es casi que un valor de tipo procedimiento. Probablemente la restricción existe porque los diseñadores del lenguaje deseaban que las variables no finales dentro de los bloques de código se pudieran implementar en una pila, la cual sería desocupada cuando se terminara el método. Sin la restricción, se pueden crear referencias sueltas.

### **7.7.2. Introducción a la programación en Java**

Presentamos una breve introducción a la programación en Java. Explicamos cómo escribir un programa sencillo, cómo definir clases, y cómo usar la herencia. En esta sección sólo se araña la superficie de lo que es posible hacer en Java. Para mayor información, referimos al lector a uno de los muy buenos libros de programación en Java [10].

#### *Un programa sencillo*

Nos gustaría calcular la función factorial. En Java, las funciones se definen como métodos que devuelven un resultado:

```
class Factorial {
 public long fact(long n) {
 long f=1;
 for (int i=1; i<=n; i++) f=f*i;
 return f;
 }
}
```

Las declaraciones se terminan con un punto y coma “;” a menos que sean declaraciones compuestas, las cuales se delimitan con corchetes {...}. Los identificadores de variables se declaran precedidos por su tipo, como en `long f`. La asignación se denota con el símbolo de igualdad `=`. Escrito en el sistema de objetos del capítulo 7 esto se vería así:

```
class Factorial
 meth fact(N ?X)
 F={NewCell 1} in
 for I in 1..N do F:=@F*I end
 X=@F
 end
 end
```

Note que `i` es una variable assignable (una celda) que se actualiza en cada iteración, mientras que `I` es un valor que se declara de nuevo en cada iteración. `Factorial` tambien se puede definir recursivamente:

```
class Factorial {
 public long fact(long n) {
 if (n==0) return 1;
 else return n*this.fact(n-1);
 }
}
```

En nuestro sistema de objetos esto se escribiría así:

```
class Factorial
 meth fact(N ?F)
 if N==0 then F=1
 else F=N*{self fact(N-1 $)} end
 end
end
```

Existen unas cuantas diferencias con el sistema de objetos del capítulo 7. La palabra reservada `this` en Java es lo mismo que `self` en nuestro sistema de objetos. Java es estáticamente tipado. El tipo de todas las variables se declara en tiempo de compilación. Nuestro modelo es dinámicamente tipado. Una variable puede ser ligada a una entidad de cualquier tipo. En Java, la visibilidad de `fact` se declara como pública. En nuestro modelo, `fact` es público por defecto; para que tuviera otra visibilidad, tendríamos que declararlo como un nombre.

### *Entrada/salida*

Cualquier programa real en Java tiene que tener entrada/salida (E/S). Java tiene un subsistema de E/S bastante elaborado, basado en la noción de flujo, el cual es una secuencia ordenada de datos que tiene una fuente (para el caso de un flujo de entrada) o un destino (para el caso de un flujo de salida). Esto no debe confundirse con el concepto de flujo tal como se ha usado en el resto del libro: una lista con la cola no-ligada. El concepto de flujo de Java generaliza el concepto Unix de E/S estándar, i.e., los archivos standard input (`stdin`) y standard output (`stdout`).

Los flujos pueden codificar muchos tipos, incluyendo los tipos primitivos, objetos, y grafos de objetos. (Un grafo de objeto es un objeto junto con los otros objetos que él referencia, directa o indirectamente.) Los flujos pueden ser flujos de bytes o flujos de caracteres. Los caracteres no son lo mismo que bytes pues Java soporta Unicode. Un byte en Java es un entero sin signo de 8 bits. Un carácter en Java es un carácter en Unicode 2.0, el cual tiene un código de 16 bits. No trataremos más el tema de E/S en esta sección.

### ***Definiendo clases***

La clase `Factorial` es bastante atípica. Ella tiene sólo un método y ningún atributo. Definamos una clase más realista. Esta es una clase para definir puntos en un espacio de dos dimensiones:

```
class Punto {
 public double x, y;
}
```

Los atributos `x` y `y` son públicos, lo que significa que son visibles desde afuera de la clase. Los atributos públicos no son normalmente una buena idea; casi siempre es mejor definirlos privados y usar métodos para acceder a ellos:

```
class Punto {
 double x, y;
 Punto(double x1, y1) { x=x1; y=y1; }
 public double obtX() { return x; }
 public double obtY() { return y; }
}
```

El método `Punto` se denomina un constructor; se utiliza para inicializar los objetos nuevos creados con `new`, como en:

```
Punto p=new Punto(10.0, 20.0);
```

el cual crea un nuevo objeto `p` de la clase `Punto`. Agreguemos algunos métodos para calcular con puntos:

```
class Punto {
 double x, y;
 Punto(double x1, y1) { x=x1; y=y1; }
 public double obtX() { return x; }
 public double obtY() { return y; }
 public void origen() { x=0.0; y=0.0; }
 public void sumar(Punto p) { x+=p.obtX(); y+=p.obtY(); }
 public void escalar(double s) { x*=s; y*=s; }
}
```

El argumento `p` de `sumar` es una variable local cuyo valor inicial es una referencia al argumento. En nuestro sistema de objetos podemos definir `Punto` de la manera siguiente:

```

class MiEntero {
 public int val;
 MiEntero(int x) { val=x; }
}

class EjemploInvoc {
 public static void sqr(MiEntero a) {
 a.val=a.val*a.val;
 a=null;
 }

 public static void main(String[] args) {
 MiEntero c=new MiEntero(25);
 EjemploInvoc.sqr(c);
 System.out.println(c.val);
 }
}

```

**Figura 7.30:** Paso de parámetros en Java.

```

class Punto
 attr x y
 meth inic(X Y) x:=X y:=Y end
 meth obtX(X) X=@x end
 meth obtY(Y) Y=@y end
 meth origen x:=0.0 y:=0.0 end
 meth sumar(P) x:=@x+{P obtX($)} y:=@y+{P obtY($)} end
 meth escalar(S) x:=@x*S y:=@y*S end
end

```

Esta definición es muy similar a la definición en Java. Existen otras diferencias sintácticas menores, tales como los operadores `+=` y `**=`. Ambas definiciones tienen atributos privados. Existe una diferencia sutil en la visibilidad de los atributos. En Java, los atributos privados son visibles para todos los objetos de la misma clase. Esto significa que el método `sumar` podría escribirse de manera diferente:

```
public void sumar(Punto p) { x+=p.x; y+=p.y; }
```

Esto se explica más adelante en la sección 7.3.3.

#### *Paso de parámetros y programa principal*

El paso de parámetros a los métodos se hace por valor. Se pasa una copia al método, la cual puede ser modificada dentro del mismo, pero no tiene efectos sobre el valor original. Para valores primitivos como los enteros y los flotantes, esto es sencillo. Java también pasa referencias a objetos (no los objetos mismos) por valor. Por tanto, se puede casi considerar que los objetos usan paso de parámetros por referencia. La diferencia es que, dentro del método, el campo se puede modificar

para referenciar a otro objeto.

En la figura 7.30 se presenta un ejemplo. Este ejemplo es un programa completo autónomo; puede ser compilado y ejecutado tal cual. Cada programa Java tiene un método `main`, que se invoca cuando el programa empieza. La referencia al objeto `c` se pasa por valor al método `sqr`. Dentro del método `sqr`, la asignación `a=null` no tiene ningún efecto sobre `c`.

El argumento de `main` es un arreglo de cadenas de caracteres que contiene los argumentos de la línea de comandos del programa cuando se invoca desde el sistema operativo. La invocación al método `System.out.println` imprime su argumento en la salida estándar.

### ***Herencia***

Podemos usar la herencia para extender la clase `Punto`. Por ejemplo, se puede extender para representar un pixel, el cual es el área más pequeña que se puede desplegar sobre un aparato gráfico de salida, de dos dimensiones, tal como una pantalla de computador. Los pixels tienen coordenadas, tal como los puntos, pero también tienen un color.

```
class Pixel extends Punto {
 Color color;
 public void origen() {
 super.origen();
 color=null;
 }
 public Color obtC() { return color; }
 public void asgnC(Color c) { color=c; }
}
```

La palabra reservada `extends` se utiliza para denotar la herencia; ella corresponde al `from` de nuestro sistema de objetos. Suponemos que la clase `Color` está definida en alguna parte. La clase `Pixel` anula el método `origen`. El nuevo `origen` inicializa tanto el punto como el color. Este método utiliza `super` para acceder al método anulado en la clase ancestro inmediata. Con respecto a la clase actual, esta clase se denomina con frecuencia la superclase. En nuestro sistema de objetos, `Pixel` se puede definir así:

```
class Pixel from Punto
 attr color
 meth origen
 Punto,origen
 color:=null
 end
 meth obtC(C) C=@color end
 meth asgnC(C) color:=C end
end
```

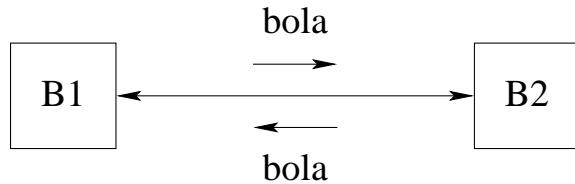
```

class JuegoDeBola
 attr otro cont:0
 meth inic(Otro)
 otro:=Otro
 end
 meth bola
 cont:=@cont+1
 {@otro bola}
 end
 meth obt(X)
 X=@cont
 end
end

B1={CrearObjActivo JuegoDeBola inic(B2)}
B2={CrearObjActivo JuegoDeBola inic(B1)}
{B1 bola}

```

**Figura 7.31:** Dos objetos activos jugando con la bola (definición).



**Figura 7.32:** Dos objetos activos jugando con la bola (ilustración).

---

## 7.8. Objetos activos

Un objeto activo es un objeto puerto cuyo comportamiento se define por medio de una clase. Un objeto activo consiste entonces de un puerto, un hilo que lee mensajes del flujo del puerto, y un objeto que es una instancia de una clase. Cada mensaje que se recibe llevará a que se invoque uno de los métodos del objeto. Los objetos activos combinan las capacidades de la POO (incluyendo polimorfismo y herencia) y las capacidades de la concurrencia por paso de mensajes (incluyendo concurrencia e independencia de los objetos). Con respecto a los objetos activos, los otros objetos de este capítulo los denominamos objetos pasivos, pues ellos no cuentan con un hilo interno.

### 7.8.1. Un ejemplo

Empecemos con un ejemplo. Considere dos objetos activos, donde cada objeto tiene una referencia al otro. Cuando un objeto recibe el mensaje `bola`, le envía el mensaje `bola` al otro. La bola será pasada de un objeto al otro indefinidamente. Definimos el comportamiento de los objetos activos por medio de una clase. En la figura 7.31 se definen los objetos y en la figura 7.32 se ilustra cómo pasan los mensajes entre ellos. Cada objeto guarda una referencia al otro en el atributo `otro`. También agregamos un atributo `cont` para contar el número de veces que se recibe el mensaje `bola`. La invocación inicial `{B1 bola}` comienza el juego. Con el método `obt(x)` podemos hacer seguimiento al progreso del juego:

```
declare x in
{B1 obt(x)}
{Browse x}
```

Al hacer esto varias veces se verá una secuencia de números que se incrementa rápidamente.

### 7.8.2. La abstracción `CrearObjActivo`

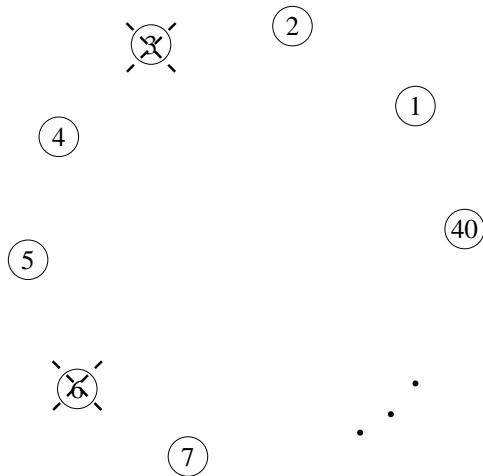
El comportamiento de los objetos activos se define por medio de una clase. Cada método de la clase corresponde a un mensaje que el objeto activo acepta. En la figura 7.31 se presenta un ejemplo. Enviar un mensaje `M` a un objeto activo `A` se escribe `{A M}`, con la misma sintaxis con que se invoca a un objeto pasivo estándar. A diferencia de lo que sucede con los objetos pasivos, la invocación de un objeto activo es asíncrona: la invocación termina inmediatamente, sin esperar a que el mensaje haya sido manejado. Podemos definir una función `CrearObjActivo` que funciona exactamente igual que `New` salvo que crea un objeto activo:

```
fun {CrearObjActivo Clase Inic}
Obj={New Clase Inic}
P
in
thread S in
{NewPort S P}
for M in S do {Obj M} end
end
proc {$ M} {Send P M} end
end
```

Esto hace que la definición de objetos activos sea muy intuitiva.

### 7.8.3. El problema de Flavio Josefo

Abordamos ahora un problema más grande. Lo introducimos con una anécdota histórica bien conocida. Flavio Josefo fue un historiador romano de origen judío. Durante las guerras judío-romanas del primer siglo D.C., estaba en una cueva con otros soldados, cuarenta en total, rodeado por las tropas enemigas romanas. Ellos



**Figura 7.33:** El problema de Flavio Josefo.

acordaron suicidarse, colocándose inicialmente en una fila en forma de círculo, y contando de tres en tres a partir del primero. Cada soldado que quedaba de tercero, se suicidaba, y el círculo se hacía más pequeño. En la figura 7.33 se ilustra el problema. Josefo, quien no quería morir, manipuló las cosas para colocarse, en el inicio, en la posición del último en suicidarse, que, al no hacerlo, resultó ser en realidad la posición del sobreviviente.

En la versión general de este problema, existen  $n$  soldados numerados del 1 al  $n$  y cada  $k$ -ésimo soldado debe ser eliminado. El conteo comienza desde el primer soldado. ¿Cuál es la posición inicial del sobreviviente? Modelemos este problema representando los soldados como objetos activos. Existe un círculo de objetos activos donde cada objeto conoce a sus dos vecinos. El siguiente es un posible protocolo de paso de mensajes para resolver el problema. Un mensaje `matar(x s)` circula alrededor del círculo, donde  $x$  cuenta el número de objetos vivos que ha atravesado el mensaje y  $s$  contiene el número total de objetos vivos restantes. Inicialmente, el mensaje `matar(1 n)` se envía al primer objeto. Cuando el objeto  $i$  recibe el mensaje `matar(x s)` hace lo siguiente:

- Si está marcado como vivo y  $s = 1$ , entonces él es el sobreviviente. El objeto indica esto ligando una variable global. No se reenvían más mensajes.
- Si está marcado como vivo y  $x \bmod k = 0$ , entonces el objeto se marca como muerto y envía el mensaje `matar(x+1 s-1)` al objeto siguiente en el círculo.
- Si está marcado como vivo y  $x \bmod k \neq 0$ , entonces envía el mensaje `matar(x+1 s)` al objeto siguiente en el círculo.
- Si está marcado como muerto, entonces reenvía el mensaje `matar(x s)` al objeto

```

class Víctima
 attr ident paso sobrev suc pred vivo:true
 meth inic(I K L) ident:=I paso:=K sobrev:=L end
 meth asignSuc(S) suc:=S end
 meth asignPred(P) pred:=P end
 meth matar(X S)
 if @vivo then
 if S==1 then @sobrev=@ident
 elseif X mod @paso==0 then
 vivo:=false
 {@pred nuevosuc(@suc)}
 {@suc nuevopred(@pred)}
 {@suc matar(X+1 S-1)}
 else
 {@suc matar(X+1 S)}
 end
 else {@suc matar(X S)} end
 end
 meth nuevosuc(S)
 if @vivo then suc:=S
 else {@pred nuevosuc(S)} end
 end
 meth nuevopred(P)
 if @vivo then pred:=P
 else {@suc nuevopred(P)} end
 end
end

fun {Josefo N K}
 A={NewArray 1 N null}
 Sobrev
 in
 for I in 1..N do
 A.I:={CrearObjActivo Víctima inic(I K Sobrev)}
 end
 for I in 2..N do {A.I asignPred(A.(I-1))} end
 {A.1 asignPred(A.N)}
 for I in 1..(N-1) do {A.I asignSuc(A.(I+1))} end
 {A.N asignSuc(A.1)} {A.1 matar(1 N)}
 Sobrev
 end

```

**Figura 7.34:** El problema de Flavio Josefo (versión con objetos activos).

```

fun {Canal Xs L H F}
 if L=<H then {Canal {F Xs L} L+1 H F} else Xs end
end

fun {Josefo2 N K}
 fun {Víctima Xs I}
 case Xs of matar(X S)|Xr then
 if S==1 then Sobrev=I nil
 elseif X mod K==0 then
 matar(X+1 S-1)|Xr
 else
 matar(X+1 S)|{Víctima Xr I}
 end
 [] nil then nil end
 end
 Sobrev Zs
in
 Zs={Canal matar(1 N)|Zs 1 N
 fun {$ Is I} thread {Víctima Is I} end end}
 Sobrev
end

```

**Figura 7.35:** El problema de Flavio Josefo (versión concurrente dirigida por los datos).

siguiente en el círculo.<sup>5</sup>

En la figura 7.34 se presenta un programa que implementa este protocolo. La función *Josefo* devuelve inmediatamente una variable no ligada, la cual será ligada con el número del sobreviviente una vez se conozca.

### *Protocolo corto-circuito*

La solución de la figura 7.34 elimina los objetos muertos del círculo con un protocolo corto-circuito. Si esto no se hace, finalmente el mensaje gastaría la mayor parte del tiempo siendo reenviado por objetos muertos. El protocolo corto-circuito utiliza los métodos *nuevosuc* y *nuevopred*. Cuando un objeto muere, él le señala tanto a su sucesor como a su predecesor que debe ser evitado. El protocolo corto-circuito no es más que una optimización para reducir el tiempo de ejecución. Éste se puede eliminar y el programa aún funcionaría correctamente.

Sin el protocolo corto-circuito, el programa es realmente secuencial pues solamente hay un mensaje circulando. Entonces, se habría podido escribir como un programa secuencial. Con el protocolo corto-circuito deja de ser secuencial, pues más de un mensaje puede estar atravesando la red en cualquier momento.

---

5. Un estudiante hizo la observación que un objeto marcado como muerto es una especie de zombi.

### Una solución declarativa

Como programadores perspicaces, nos damos cuenta que la solución de la figura 7.34 tiene un no-determinismo no observable. Por lo tanto, podemos escribirlo por completo en el modelo concurrente declarativo del capítulo 4. Hagámoslo y comparemos los dos programas. En la figura 7.35 se muestra una solución declarativa que implementa el mismo protocolo que la versión con objetos activos. Al igual que esa versión, ésta implementa el corto-circuito y termina, finalmente, con la identidad del sobreviviente. Vale la pena comparar las dos versiones cuidadosamente. El tamaño de la versión declarativa es la mitad del tamaño de la versión con objetos activos. Una razón es que los flujos son entidades de primera clase. Esto hace que el corto-circuito sea muy fácil: simplemente devolver el flujo de entrada como salida.

El programa declarativo utiliza una abstracción concurrente, `canal`, definida especialmente para este programa. Si  $l \leq h$ , entonces la invocación `{Canal xs L H F}` crea un canal con  $h - l + 1$  objetos flujo, numerados del  $l$  al  $h$  inclusive. Cada objeto flujo se crea a partir de la invocación `{F Is I}`, la cual recibe un flujo de entrada `Is` y un entero `I`, y devuelve el flujo de salida. Creamos un círculo cerrado reconectando el flujo de salida `zs` con el de entrada, con el mensaje adicional `matar(1 N)` para comenzar la ejecución.

#### 7.8.4. Otras abstracciones de objetos activos

En la sección 5.3 se muestran algunos de los protocolos útiles que se pueden construir sobre el modelo de paso de mensajes. Tomemos dos de estos protocolos y convirtámoslos en abstracciones para objetos activos.

#### Objetos activos sincrónicos

Es fácil extender los objetos activos para dotarlos de un comportamiento sincrónico, como un objeto estándar o un objeto RMI. Una invocación sincrónica no termina hasta que el método `M` correspondiente se ejecuta por completo. Dentro de la abstracción, usamos una variable de flujo de datos para realizar la sincronización. Esta es la definición de `CrearObjActSinc`, la cual crea un objeto activo sincrónico:

```
fun {CrearObjActSinc Clase Inic}
 P Obj={New Clase Inic} in
 thread S in
 {NewPort S P}
 for M#X in S do {Obj M} X=listo end
 end
 proc {$ M} X in {Send P M#X} {Wait X} end
 end
```

Cada mensaje que se envía al objeto contiene un testigo de sincronización `X`, el cual sólo se liga cuando el mensaje ha sido manejado por completo.

### *Objetos activos con manejo de excepciones*

Hacer que los objetos activos realicen manejo de excepciones explícitamente puede ser engoroso, pues eso significa añadir un **try** en cada método del servidor y un **case** después de cada invocación. Ocultemos esas declaraciones dentro de la abstracción. La abstracción cuenta con un argumento adicional que puede ser usado para comprobar si ocurrió una excepción o no. En lugar de añadir el argumento adicional en el método, lo añadimos en la invocación misma del objeto. De esta manera, esto funciona automáticamente para todos los métodos. El argumento adicional se ligará a **normal** si la invocación se completa normalmente, y a **excepción(E)** si el objeto lanza la excepción **E**. Esta es la definición de **CrearObjActExc**:

```
fun {CrearObjActExc Clase Inic}
P Obj={New Clase Inic} in
 thread S in
 {NewPort S P}
 for M#X in S do
 try {Obj M} X=normal
 catch E then X=excepción(E) end
 end
 end
 proc {$ M X} {Send P M#X} end
end
```

El objeto **Obj** se invoca como **{Obj M X}**, donde **X** es el argumento adicional. Por tanto el envío es aún asíncrono y el cliente puede examinar en cualquier momento si la invocación se completó exitosamente o no. Para el caso sincrónico, podemos colocar la declaración **case** dentro de la abstracción:

```
proc {$ M}
 X in
 {Send P M#X}
 case X of normal then skip
 [] excepción(E) then raise E end end
end
```

Esto nos permite invocar el objeto exactamente como un objeto pasivo.

#### 7.8.5. Manejador de eventos con objetos activos

Podemos utilizar los objetos activos para implementar un manejador concurrente de eventos sencillo. El manejador de eventos contiene un conjunto de manejadores de eventos. Cada manejador es una tripleta **Id#F#S**, donde **Id** identifica de manera única el manejador, **F** define la función de actualización del estado, y **S** es el estado del manejador. Cuando sucede un evento **E**, cada tripleta **Id#F#S** se reemplaza por **Id#F#{F E S}**. Es decir, cada manejador es una máquina de estados finitos, la cual en el momento en que sucede el evento **E** realiza una transición del estado **S** al estado **{F E S}**.

El manejador de eventos fue escrito originalmente en Erlang [6]. El modelo de

```

class ManejadorEventos
 attr
 manejadores
 meth inic manejadores:=nil end
 meth evento(E)
 manejadores:=
 {Map @manejadores fun {$ Id#F#S} Id#F#{F E S} end}
 end
 meth agr(F S ?Id)
 Id={NewName}
 manejadores:=Id#F#S|@manejadores
 end
 meth elim(DId ?DS)
 manejadores:={List.partition
 @manejadores fun {$ Id#F#S} DId==Id end [_#_#DS]}
 end
end

```

**Figura 7.36:** Manejador de eventos con objetos activos.

computación de Erlang se basa en comunicación de objetos activos (ver capítulo 5). La traducción del código original al modelo concurrente con estado fue directa.

Definimos el manejador de eventos ME como un objeto activo con cuatro métodos:

1. {ME inic} inicializa el manejador de eventos.
2. {ME evento(E)} coloca el mensaje E en el manejador de eventos.
3. {ME agr(F S Id)} agrega un nuevo manejador con función de actualización F y estado inicial S. Devuelve un identificador único Id.
4. {ME elim(Id S)} elimina el manejador identificado con Id, si existe. Devuelve el estado del manejador en S.

En la figura 7.36 se muestra cómo definir el manejador de eventos como una clase. Ahora, mostramos cómo usar el manejador de eventos para realizar el registro de errores. Primero, definimos un manejador de eventos nuevo:

```
ME={CrearObjActivo ManejadorEventos inic}
```

Luego instalamos un manejador basado en memoria, el cual registra cada evento en una lista interna:

```
ManMem=fun {$ E Mem} E|Mem end
Id={ME agr(ManMem nil $)}
```

Durante la ejecución, podemos reemplazar el manejador basado en memoria, por un manejador basado en disco, sin perder ninguno de los eventos registrados hasta ese momento. En el código siguiente, eliminamos el manejador basado en memoria, abrimos un archivo de registros, escribimos en el archivo los eventos que ya se han registrado, y luego definimos e instalamos el manejador basado en disco:

```

class ReemplazoManejadorEventos from ManejadorEventos
 meth reemplazar(NuevaF NuevoS ViejoId NuevoId
 insertar:P<=proc {$ _} skip end)
 Mem=ManejadorEventos, elim(ViejoId $)
 in
 {P Mem}
 NuevoId=ManejadorEventos, agr(NuevaF NuevoS $)
 end
end

```

**Figura 7.37:** Agregando funcionalidad con herencia.

```

ManDisco=fun {$ E A} {A write(vs:E)} F end
Arch={New Open.file init(name:'evento.log' flags:[write create])}
Mem={ME elim(Id $)}
for E in {Reverse Mem} do {Arch write(vs:E)} end
Id2={ME agr(ManDisco Arch $)}

```

Aquí se utiliza el módulo `Open` del sistema para escribir el registro. Podríamos usar el módulo `File`, pero entonces el resto del programa no lo podría usar, pues este módulo solamente soporta tener un archivo abierto para escritura a la vez.

### *Agregando funcionalidad con herencia*

El manejador de eventos de la figura 7.36 tiene el defecto que los eventos que sucedan durante un reemplazo, i.e., entre las operaciones de eliminación de un manejador y adición del otro, no quedarán registrados. ¿Cómo resolver este defecto? Una solución sencilla consiste en agregar un método nuevo, `reemplazar`, al `ManejadorEventos` que haga tanto la eliminación como la adición. Como en un objeto activo todos los métodos se ejecutan secuencialmente, ningún evento sucederá entre la eliminación y la adición. Podemos agregar este método nuevo directamente a la clase `ManejadorEventos`, o indirectamente, a una subclase por medio de la herencia. Decidir cuál posibilidad es la solución correcta depende de varios factores. Primero, si tenemos o no acceso al código fuente de `ManejadorEventos`. Si no tenemos acceso, entonces la única posibilidad es la herencia. Si tenemos acceso al código fuente, la herencia puede ser aún la respuesta correcta. Depende de cuán frecuentemente necesitemos la funcionalidad de `reemplazar`. Si la necesitamos casi siempre en los manejadores de eventos, entonces deberíamos modificar directamente `ManejadorEventos` y no crear una segunda clase. Pero, si raramente la necesitamos, entonces su definición no debería volver más compleja a la clase `ManejadorEventos`, y podemos separarla utilizando la herencia.

Utilicemos la herencia para este ejemplo. La figura 7.37 define una clase nueva `ReemplazoManejadorEventos` que hereda de `ManejadorEventos` y agrega un método nuevo `reemplazar`. Las instancias de `ReemplazoManejadorEventos` tienen todos los métodos de `ManejadorEventos` así como el método `reemplazar`. El campo `insertar` es opcional; puede usarse para insertar una operación que debe

```
class ProcPorLotes
meth lote(L)
 for X in L do
 if {IsProcedure X} then {X} else {self X} end
 end
end
```

**Figura 7.38:** Procesando por lotes una lista de mensajes y de procedimientos.

ser realizada entre las operaciones de eliminación y adición. Ahora definimos un manejador de eventos nuevo:

```
ME={CrearObjActivo ReemplazoManejadorEventos inic}
```

Podemos hacer el reemplazo así:

```
ManDisco=fun {$ E A} {A write(vs:E)} A end
Arch={New Open.file init(name:'evento.log' flags:[write create])}
Id2
{ME reemplazar(ManDisco Arch Id Id2
 insert:
 proc {$ S}
 for E in {Reverse S} do
 {Arch write(vs:E)} end
 end)}
```

Como `reemplazar` se ejecuta dentro del objeto activo, este mensaje es serializado con todos los otros mensajes que llegan al objeto. Esto asegura que ningún evento puede llegar entre las operaciones de eliminación y adición.

### *Procesamiento de operaciones por lotes utilizando una clase de mezcla*

Una segunda manera de remediar el defecto consiste en agregar un método nuevo que haga procesamiento por lotes, i.e., ejecute una lista de operaciones. En la figura 7.38 se define una clase nueva, `ProcPorLotes`, que tiene un solo método, `lote(L)`. La lista `L` puede contener mensajes o procedimientos sin argumentos. Cuando se invoca el método `lote(L)`, los mensajes se pasan a `self` y los procedimientos se ejecutan, en el orden en que suceden en `L`. Este es un ejemplo de utilización de mensajes de primera clase. Como los mensajes también son entidades del lenguaje (son registros), entonces se pueden colocar en una lista y se pueden pasar a `ProcPorLotes`. Definamos una clase nueva que herede de `ManejadorEventos` y tenga la funcionalidad de `ProcPorLotes`:

```
class ManejadorEventosPorLotes from ManejadorEventos ProcPorLotes
end
```

Utilizamos la herencia múltiple porque `ProcPorLotes` puede ser útil para cualquier clase que necesite procesamiento por lotes, no solamente para los manejadores de eventos. Ahora podemos definir un manejador de eventos nuevo:

```
ME={CrearObjActivo ManejadorEventosPorLotes inic}
```

Todas las instancias de `ManejadorEventosPorLotes` tienen todos los métodos de `ManejadorEventos` así como el método `lote`. La clase `ProcPorLotes` es un ejemplo de una clase de mezcla: le agrega funcionalidad a una clase existente sin necesidad de conocer nada sobre esa clase. Ahora, podemos reemplazar el manejador basado en memoria por uno basado en disco:

```
ManDisco=fun {$ E A} {A write(vs:E)} A end
Arch={New Open.file init(name:'evento.log' flags:[write create])}
Mem Id2
{ME lote([elim(Id Mem)
proc {$}
 for E in {Reverse Mem} do {Arch write(vs:E)} end
end
agr(ManDisco Arch Id2)])}
```

El método `lote` garantiza la atomicidad de la misma forma que el método `reemplazar`, i.e., por ejecutarse dentro del objeto activo.

¿Cuáles son las diferencias entre la solución de reemplazar explícitamente y la solución usando procesamiento por lotes? Son dos:

- La solución de reemplazar explícitamente es más eficiente porque el método `reemplazar` está directamente codificado. En cambio, el método `lote` agrega una capa de interpretación.
- La solución usando procesamiento por lotes es más flexible. El procesamiento por lotes se puede añadir a cualquier clase utilizando herencia múltiple. No se necesita definir nuevos métodos. Además, se puede procesar cualquier lista de mensajes y procedimientos, aún una lista calculada en tiempo de ejecución. Sin embargo, la solución usando procesamiento por lotes requiere que el lenguaje soporte mensajes de primera clase.

### *Combinando modelos de computación*

El manejador de eventos es una combinación interesante de los modelos de computación declarativo, orientado a objetos, y concurrente con estado:

- Cada manejador de eventos se define declarativamente por su función de actualización de estado. Más aún, cada método del manejador de eventos se puede ver como una definición declarativa. Cada método toma los estados internos de los manejadores de eventos del atributo `manejadores`, hace una operación declarativa sobre ellos, y almacena el resultado en `manejadores`.
- Todos los métodos se ejecutan de forma secuencial, como si estuviéramos en un modelo con estado sin concurrencia. Toda la concurrencia se maneja por medio de la abstracción de objeto activo, tal como está implementada por `CrearObjActivo`. Esta abstracción garantiza que todas las invocaciones al objeto son serializadas. Especialmente, no se necesitan candados ni ningún otro control de concurrencia.

- La funcionalidad nueva, e.g., reemplazo explícito o procesamiento por lotes, se agregó usando la herencia orientada a objetos. Como los métodos nuevos se ejecutan dentro del objeto activo, se garantiza su atomicidad.

El resultado es que los manejadores de eventos se definen de manera secuencial y declarativa, y pueden ser usados en un ambiente concurrente con estado. Este es un ejemplo de adaptación de impedancias, como se definió en la sección 4.8.7. La adaptación de impedancias es un caso general del principio de separación de asuntos. Los asuntos que tienen que ver con el estado y la concurrencia se separan de la definición de los manejadores de eventos. Es una buena práctica de programación separar los asuntos tanto como sea posible. La utilización de diferentes modelos de computación a la vez, ayuda, con frecuencia, a lograr la separación de asuntos.

---

## 7.9. Ejercicios

1. *Objetos no inicializados.* La función `New` crea un objeto nuevo a partir de una clase dada y un mensaje inicial. Escriba otra función `New2` que no requiera un mensaje inicial. Es decir, la invocación `Obj={New2 Clase}` crea un objeto nuevo sin inicializarlo. *Sugerencia:* escriba `New2` en términos de `New`.
2. *Métodos protegidos en el sentido de Java.* Un método protegido en Java tiene dos partes: a él se puede acceder a través del paquete que define la clase y también por los descendientes de la clase. En este ejercicio, defina una abstracción lingüística que permita anotar un método o atributo como `protected` en el sentido de Java. Muestre cómo codificar esto en el modelo de la sección 7.3.3 utilizando valores de tipo nombre. Utilice funtores para representar los paquetes de Java. Por ejemplo, un enfoque podría consistir en definir globalmente el valor de tipo nombre en el functor y almacenarlo en un atributo denominado `conjuntoDeAtributosProtegidos`. Como el atributo se hereda, el nombre del método es visible en todas las subclases. Elabore los detalles de este enfoque.
3. *Empaquetamiento de métodos.* En la sección 7.3.5 se muestra cómo hacer el empaquetamiento de métodos. La definición de `NuevoVigilante2` presentada allí utiliza una clase `vigilante` que tiene una referencia externa. Esto no se permite en algunos sistemas de objetos. En este ejercicio, reescriba `NuevoVigilante2` de manera que se utilice una clase sin referencias externas.
4. *Implementación de herencia y ligadura estática.* En este ejercicio, generalice la implementación del sistema de objetos presentada en la sección 7.6 para manejar ligadura estática y manejar herencia con cualquier número de superclases (no solamente dos).
5. *Protocolos de mensajes con objetos activos.* En este ejercicio, reescriba los protocolos de mensajes de la sección 5.3 con objetos activos en lugar de objetos puerto.
6. *El problema de Flavio Josefo.* En la sección 7.8.3 se resuelve este problema de dos maneras, usando objetos activos, y usando concurrencia dirigida por los datos.

En este ejercicio, haga lo siguiente:

- a) Use un tercer modelo, el modelo secuencial con estado, para resolver el problema. Escriba dos programas: el primero sin corto-circuito, y el segundo con él. Trate de hacerlos ambos tan concisos y naturales como le sea posible en el modelo. Por ejemplo, sin corto-circuito, un arreglo de booleanos es una estructura natural para representar el círculo. Compare la estructura de ambos programas con los dos programas de la sección 7.8.3.
  - b) Compare los tiempos de ejecución de las diferentes versiones. Hay dos análisis ortogonales para hacer. Primero, medir las ventajas (si las hay) de usar el corto-circuito, para varios valores de  $n$  y  $k$ . Esto se puede hacer en cada uno de los tres modelos de computación. Para cada modelo, divida el plano  $(n, k)$  en dos regiones, dependiendo de si el corto-circuito es más rápido o no. ¿Las regiones son las mismas para cada modelo? Segundo, compare las tres versiones con corto-circuito. ¿Estas versiones tienen la misma complejidad asintótica en tiempo como una función de  $n$  y  $k$ ?
7. (ejercicio avanzado) *Herencia sin estado explícito*. La herencia no requiere del estado explícito; los dos conceptos son ortogonales. En este ejercicio, diseñe e implemente un sistema de objetos con clases y herencia pero sin estado explícito. Un posible punto de partida es la implementación de objetos declarativos de la sección 6.4.2.
8. ( proyecto de investigación) *Programación de patrones de diseño*. En este ejercicio, diseñe un lenguaje orientado a objetos que permita herencia “hacia arriba” (definición de una superclase nueva de una clase dada) así como programación de alto orden. La herencia “hacia arriba” normalmente se denomina generalización. Implemente y evalúe la utilidad de su lenguaje. Muestre cómo programar los patrones de diseño de Gamma et al. [54] como abstracciones en su lenguaje. ¿Necesita usted de otras operaciones nuevas, además de la generalización?



## Concurrencia con estado compartido

Por supuesto, los candados y las llaves fueron un pequeño comienzo, y el lenguaje ya estaba muy desarrollado, quizás demasiado, antes de que se diera este pequeño paso en el camino hacia un equivalente mecánico del lenguaje. Hay un largo camino desde los primeros candados, pasando por telares y reproductores mecánicos de música hasta los primeros computadores mecánicos, y luego otro largo camino hasta el computador electrónico.

– Adaptación libre de *Invention and Evolution: Design in Nature and Engineering*, Michael French (1994)

El modelo concurrente con estado compartido es una extensión sencilla del modelo concurrente declarativo en la que se agrega estado explícito en forma de celdas, las cuales son una especie de variables mutables. Este modelo es equivalente en términos de expresividad al modelo concurrente por paso de mensajes del capítulo 5, puesto que las celdas se pueden implementar eficientemente con puertos y viceversa. Cuando deseamos cubrir ambos modelos, hablamos del modelo concurrente con estado. Los modelos no son equivalentes en la práctica puesto que cada uno fomenta un estilo de programación muy diferente. El modelo con estado compartido fomenta los programas donde los hilos acceden concurrentemente a un repositorio de datos compartidos. El modelo por paso de mensajes fomenta los programas multiagentes. Es más difícil programar en el modelo con estado compartido que en el modelo por paso de mensajes. Miremos cuál es el problema y cómo resolverlo.

### *La dificultad inherente al modelo*

Miremos primero exactamente por qué el modelo con estado compartido es difícil. La ejecución consiste de múltiples hilos, ejecutándose todos de manera independiente, y accediendo todos a celdas compartidas. En cierto nivel, una ejecución de un hilo se puede ver como una secuencia de instrucciones atómicas. En el caso de las celdas, estas instrucciones son @ (acceso), := (asignación), y Exchange. Debido a la semántica de intercalación, toda la ejecución sucede como si hubiera un orden global de las operaciones. Por lo tanto, todas las operaciones sobre todos los hilos están “intercaladas” según ese orden. Hay muchas intercalaciones posibles; su número sólo está limitado por las dependencias de los datos (cálculos que necesitan los resultados de otras operaciones). Cualquier ejecución en particular lleva a cabo una intercalación. Como el planificador de hilos es no-determinístico, no hay manera

*Concurrencia con estado compartido*

de saber cuál intercalación será la escogida.

¿Pero, cuántas intercalaciones posibles hay? Consideremos un caso sencillo: dos hilos, cada uno haciendo  $k$  operaciones sobre celdas. El hilo  $T_1$  realiza las operaciones  $a_1, a_2, \dots, a_k$  y el hilo  $T_2$  realiza las operaciones  $b_1, b_2, \dots, b_k$ . ¿Cuántas ejecuciones posibles existen, intercalando todas estas operaciones? Es fácil ver que el número es  $\binom{2k}{k}$ . Cualquier ejecución intercalada consta de  $2k$  operaciones, de las cuales cada hilo hace  $k$ . Considere estas operaciones como enteros de 1 a  $2k$ , y colóquelos en un conjunto. Entonces  $T_1$  toma  $k$  enteros de este conjunto y  $T_2$  toma los otros. Este número es exponencial en  $k$ .<sup>1</sup> Para tres o más hilos, el número de intercalaciones es aún más grande (ver ejercicios, en la sección 8.7).

Se pueden escribir algoritmos en este modelo y probar su corrección razonando sobre todas las intercalaciones posibles. Por ejemplo, si suponemos que las únicas operaciones atómicas sobre las celdas son `@` y `:=`, entonces el algoritmo de Dekker implementa la exclusión mutua. Aunque el algoritmo de Dekker es corto (e.g., 48 líneas de código en [42], utilizando un lenguaje del estilo de Pascal), el razonamiento ya es bastante difícil. Para programas más grandes, esta técnica se vuelve, muy rápidamente poco práctica. La técnica es poco manejable y es fácil dejar pasar por alto algunas intercalaciones.

*¿Por qué no utilizar la concurrencia declarativa?*

Dada la dificultad inherente de programar en el modelo concurrente con estado compartido, una pregunta obvia es ¿por qué no quedarse con el modelo concurrente declarativo del capítulo 4? Es bastante más sencillo programar en él, que en el modelo con estado compartido. Y es casi tan fácil razonar en él como en el modelo declarativo, el cual es secuencial.

Examinemos brevemente por qué el modelo concurrente declarativo es tan fácil. Eso se debe a que las variables de flujo de datos son monótonas: pueden ser ligadas a un solo valor. Una vez ligadas, el valor no cambia. Los hilos que comparten una variable de flujo de datos, e.g., un flujo, pueden, por lo tanto, calcular con el flujo como si el fuera un valor sencillo. Esto contrasta con las celdas, las cuales son no monótonas: ellas pueden ser asignadas cualquier número de veces a valores que no tienen ninguna relación entre ellos. Los hilos que comparten una celda no pueden hacer ninguna suposición sobre su contenido: en cualquier momento, el contenido puede ser completamente diferente al contenido en un instante anterior.

El problema con el modelo concurrente declarativo es que los hilos deben comunicarse en una especie de “marcha al mismo paso” o de forma “sistólica”. Dos hilos que se comunican con un tercero no pueden ejecutarse independientemente; ellos deben coordinarse con cada uno de los otros. Esto es una consecuencia de que el modelo aún es declarativo, y por tanto determinístico.

Nos gustaría permitir que dos hilos sean completamente independientes y pue-

---

1. Usando la fórmula de Stirling lo aproximamos como  $2^{2k}/\sqrt{\pi k}$ .

dan comunicarse también con el mismo tercer hilo. Por ejemplo, nos gustaría que los clientes hagan consultas independientes a un servidor común o incrementen independientemente un estado compartido. Para expresar esto, debemos dejar el dominio de los modelos declarativos. Esto se debe a que dos entidades independientes comunicándose con una tercera introducen un no-determinismo observable. Una manera sencilla de resolver el problema consiste en agregar estado explícito al modelo. Los puertos y las celdas son dos maneras importantes de agregar estado explícito. Esto nos vuelve a llevar al modelo con concurrencia y estado a la vez. Pero el razonamiento directo en este modelo es poco práctico. Miremos cómo sortear este problema.

### *Sorteando la dificultad*

Programar en el modelo concurrente con estado es, en gran parte, cuestión de administrar las intercalaciones. Hay dos enfoques exitosos:

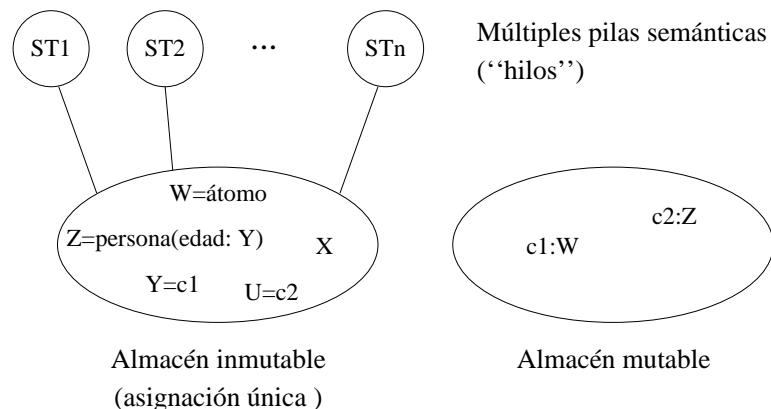
- Paso de mensajes entre objetos puerto. Este es el tema del capítulo 5. En este enfoque, los programas consisten de objetos puerto que envían mensajes asincrónicos entre ellos. Internamente, un objeto puerto se ejecuta en un solo hilo.
- Acciones atómicas sobre celdas compartidas. Este es el tema de este capítulo. En este enfoque, los programas consisten de objetos pasivos que son invocados por hilos. Las abstracciones se usan para construir acciones atómicas extensas (e.g., utilizando candados, monitores, o transacciones) de manera que el número de intercalaciones posibles sea pequeño.

Cada enfoque tiene sus ventajas y desventajas. La técnica de invariantes, tal como se explicó en el capítulo 6, se puede usar en ambos enfoques para razonar sobre los programas. Los dos enfoques son equivalentes en un sentido teórico, pero no lo son en un sentido práctico: un programa escrito usando un enfoque puede reescribirse para usar el otro enfoque, pero puede no ser tan fácil de entender [101].

### *Estructura del capítulo*

El capítulo consta de seis secciones principales:

- En la sección 8.1 se define el modelo concurrente con estado compartido.
- En la sección 8.2 se reunen y se comparan brevemente todos los diferentes modelos concurrentes que hemos introducido en el libro. Esto nos ofrece una perspectiva balanceada sobre cómo hacer programación concurrente en la práctica.
- En la sección 8.3 se introduce el concepto de candado, el cual es el concepto básico para crear acciones atómicas de granularidad gruesa. Un candado define un área del programa dentro de la cual sólo se ejecuta un hilo a la vez.
- En la sección 8.4 se extiende el concepto de candado para llegar al concepto de monitor, el cual ofrece un mejor control sobre cuáles hilos pueden entrar y salir



**Figura 8.1:** El modelo concurrente con estado compartido.

de un área con candado. Los monitores posibilitan la construcción de programas concurrentes más sofisticados.

- En la sección 8.5 se extiende el concepto de candado para llegar al concepto de transacción, el cual permite que un candado se realice o se cancele (aborte). En el último caso, es como si el candado nunca se hubiera ejecutado. Las transacciones simplifican de manera importante la programación de programas concurrentes que puedan manejar eventos poco frecuentes y salidas no locales.
- En la sección 8.6 se resume cómo se hace la concurrencia en Java, un lenguaje de programación popular, concurrente y orientado a objetos.

### 8.1. El modelo concurrente con estado compartido

En el capítulo 6 se agregó el estado explícito al modelo declarativo. Esto permite la programación orientada a objetos. En el capítulo 4 se agregó concurrencia al modelo declarativo. Esto permite tener múltiples entidades activas que evolucionan independientemente. La etapa siguiente consiste en agregar tanto estado explícito como concurrencia al modelo declarativo. Una manera de hacerlo fue presentada en el capítulo 5: agregando puertos. En este capítulo se presenta una manera alternativa de hacerlo: agregando celdas.

El modelo resultante, denominado el modelo concurrente con estado compartido, se muestra en la figura 8.1. Su lenguaje núcleo se define en la tabla 8.1. Si consideramos el subconjunto de operaciones hasta `ByNeed` entonces tenemos el modelo concurrente declarativo. Agregamos nombres, variables de sólo lectura, excepciones, y estado explícito a ese modelo.

|                                                                                                                                                          |                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| $\langle d \rangle ::=$                                                                                                                                  |                             |
| <b>skip</b>                                                                                                                                              | Declaración vacía           |
| $\langle d \rangle_1 \langle d \rangle_2$                                                                                                                | Declaración de secuencia    |
| <b>local</b> $\langle x \rangle$ <b>in</b> $\langle d \rangle$ <b>end</b>                                                                                | Creación de variable        |
| $\langle x \rangle_1 = \langle x \rangle_2$                                                                                                              | Ligadura variable-variable  |
| $\langle x \rangle = \langle v \rangle$                                                                                                                  | Creación de valor           |
| <b>if</b> $\langle x \rangle$ <b>then</b> $\langle d \rangle_1$ <b>else</b> $\langle d \rangle_2$ <b>end</b>                                             | Condicional                 |
| <b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{patrón} \rangle$ <b>then</b> $\langle d \rangle_1$ <b>else</b> $\langle d \rangle_2$ <b>end</b> | Reconocimiento de patrones  |
| $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$                                                                                  | Invocación de procedimiento |
| <b>thread</b> $\langle d \rangle$ <b>end</b>                                                                                                             | Creación de hilo            |
| $\{ \text{NewName} \langle x \rangle \}$                                                                                                                 | Creación de nombre          |
| $\langle y \rangle = ! ! \langle x \rangle$                                                                                                              | Vista de sólo lectura       |
| <b>try</b> $\langle d \rangle_1$ <b>catch</b> $\langle x \rangle$ <b>then</b> $\langle d \rangle_2$ <b>end</b>                                           | Contexto de la excepción    |
| <b>raise</b> $\langle x \rangle$ <b>end</b>                                                                                                              | Lanzamiento de excepción    |
| $\{ \text{NewCell} \langle x \rangle \langle y \rangle \}$                                                                                               | Creación de celda           |
| $\{ \text{Exchange} \langle x \rangle \langle y \rangle \langle z \rangle \}$                                                                            | Intercambio de celda        |

Tabla 8.1: El lenguaje núcleo con concurrencia con estado compartido.

## 8.2. Programación con concurrencia

Hasta ahora, hemos visto muchas formas diferentes de escribir programas concurrentes. Antes de sumergirnos en la programación concurrente con estado compartido, hagamos un ligero desvío y coloquemos todas estas formas en perspectiva. Primero presentamos una visión general, breve, de los principales enfoques. Luego, examinamos más detenidamente los enfoques nuevos posibilitados por la concurrencia con estado compartido.

### 8.2.1. Visión general de diferentes enfoques

Para el programador, existen principalmente cuatro enfoques prácticos para escribir programas concurrentes:

- *Programación secuencial* (ver capítulos 3, 6, y 7). Este es el enfoque básico que no tiene concurrencia. Puede ser ansioso o perezoso.
- *Concurrencia declarativa* (ver capítulo 4). Esto es concurrencia en el modelo declarativo, la cual ofrece los mismos resultados que un programa secuencial, pero puede producirlos incrementalmente. Este modelo se puede usar cuando existe un no-determinismo no observable. Puede ser ansioso (concurrencia dirigida por los datos) o perezoso (concurrencia dirigida por la demanda).

| Modelo                                           | Enfoques                                                                                              |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| <i>Secuencial<br/>(declarativo o con estado)</i> | Programación secuencial<br>Concurrencia que determina orden<br>Corutinas<br>Evaluación perezosa       |
| <i>Concurrente declarativo</i>                   | Concurrencia dirigida por los datos<br>Concurrencia dirigida por la demanda                           |
| <i>Concurrente con estado</i>                    | Usa el modelo directamente<br>Concurrencia por paso de mensajes<br>Concurrencia con estado compartido |
| <i>Concurrente no determinístico</i>             | Objetos flujo con mezcla                                                                              |

Figura 8.2: Diferentes enfoques de programación concurrente.

- *Concurrencia por paso de mensajes* (ver capítulo 5 y sección 7.8). Esto es pssio de mensajes entre objetos puerto, los cuales internamente son secuenciales. Esto limita el número de intercalaciones. Los objetos activos (ver sección 7.8) son una variación de los objetos puerto donde el comportamiento de los objetos se define por medio de una clase.
- *Concurrencia con estado compartido* (este capítulo). Esto es hilos actualizando objetos pasivos compartidos utilizando acciones atómicas de granularidad gruesa. Este es otro enfoque para limitar el número de intercalaciones.

En la figura 8.2 se presenta una lista completa de estos enfoques y algunos otros. En los capítulos previos ya se ha explicado la programación secuencial y la programación concurrente declarativa. En este capítulo miraremos los otros. Primero demos una mirada general de los cuatro enfoques principales.

### *Programación secuencial*

En un modelo secuencial, existe un orden total entre todas las operaciones. Este es el invariante más fuerte, en cuanto al orden, que un programa puede tener. Hemos visto dos formas en que este orden se puede relajar un poco, permaneciendo en el modelo secuencial:

- *Concurrencia “para determinación del orden”* (ver sección 4.4.1). En este modelo, todas las operaciones se ejecutan en un orden total, como con la ejecución secuencial, pero el orden es desconocido por el programador. La ejecución concurrente con flujo de datos encuentra el orden dinámicamente.
- *Corutinas* (ver sección 4.4.2). En este modelo, la prevención es explícita, i.e., el programa decide cuando pasar el control a otro hilo. La evaluación perezosa, en la

cual se agrega la pereza a un programa secuencial, se hace con corutinas. Aún, ambas variaciones del modelo son determinísticas.

### ***Concurrencia declarativa***

Todos los modelos concurrentes declarativos del capítulo 4 agregan hilos al modelo declarativo. Esto no altera el resultado de un cálculo, pero si puede cambiar el orden en que se obtiene el resultado. Por ejemplo, el resultado se puede obtener incrementalmente. Esto permite la construcción de una red dinámica de objetos flujo concurrentes conectados por medio de flujos. Gracias a la concurrencia, al agregar un elemento al flujo de entrada de un objeto flujo, éste puede producir una salida inmediatamente.

Estos modelos son no-determinísticos en su implementación, pues el sistema es quien escoge cómo avanza la ejecución de los hilos. Pero para continuar siendo declarativo, el no-determinismo no debe ser observable por el programa. Los modelos concurrentes declarativos garantizan esto siempre que no se lancen excepciones (pues las excepciones son los testigos de un no-determinismo observable). En la práctica, esto significa que cada objeto flujo debe conocer en todo momento de cuál flujo viene la próxima entrada.

El modelo concurrente dirigido por la demanda, también conocido como ejecución perezosa (ver sección 4.5), es una forma de concurrencia declarativa. Ésta no altera el resultado de un cálculo, pero si afecta la manera en que se realizan los cálculos para obtenerlo. Incluso, algunas veces, puede dar un resultado en casos en donde el modelo dirigido por los datos entraría en un ciclo infinito. Esto es importante para la administración de los recursos, i.e., controlar cuántos recursos computacionales se necesitan. Los cálculos sólo se inician cuando sus resultados son necesitados por otros cálculos. La ejecución perezosa se implementa con los disparadores by-need.

### ***Concurrencia por paso de mensajes***

El paso de mensajes es un estilo de programación básico del modelo concurrente con estado. Éste se explica en el capítulo 5 y en la sección 7.8. Allí se extiende el modelo concurrente declarativo con un canal de comunicación sencillo, un puerto. También se definen los objetos puerto, los cuales extienden los objetos flujo para leer desde los puertos. Un programa es, en consecuencia, una red de objetos puerto comunicándose entre sí por medio de paso de mensajes asincrónicos. Cada objeto puerto decide en qué momento manejar cada mensaje. El objeto puerto procesa los mensajes secuencialmente, lo cual limita las intercalaciones posibles y nos permite razonar usando invariantes. El envío y la recepción de los mensajes entre objetos puerto introduce una causalidad entre los eventos (envío, recepción, e internos). El razonamiento sobre tales sistemas requiere razonar sobre las cadenas de causalidad.

## Concurrencia con estado compartido

### Concurrencia con estado compartido

La concurrencia con estado compartido es otro estilo básico de programación del modelo concurrente con estado. Éste se explica en este capítulo. El modelo consiste en un conjunto de hilos que acceden a un conjunto de objetos pasivos compartidos. Los hilos se coordinan entre ellos el acceso a los objetos compartidos. Ellos lo hacen por medio de acciones atómicas de granularidad gruesa, e.g., candados, monitores, o transacciones. De nuevo, esto limita las intercalaciones posibles y nos permite razonar usando invariantes.

### Relación entre puertos y celdas

Los modelos por paso de mensajes y con estado compartido son equivalentes en términos de expresividad. Esto es consecuencia de que los puertos se pueden implementar con celdas y viceversa. (Es un ejercicio divertido implementar la operación `Send` utilizando `Exchange` y viceversa.) Parecería entonces que tenemos que escoger entre agregar puertos o celdas al modelo concurrente declarativo. Sin embargo, en la práctica no es así. Los dos modelos de computación enfatizan en estilos de programación bastante diferentes, apropiados para diferentes tipos de aplicaciones. El estilo por paso de mensajes es de programas como entidades activas que se coordinan entre ellas. El estilo con estado compartido es de programas como repositorios pasivos de datos que se modifican de manera coherente.

### Otros enfoques

Además de estos cuatro enfoques existen otros dos que vale la pena mencionar:

- *Utilización del modelo concurrente con estado directamente.* Esto consiste en programar directamente en el modelo concurrente con estado, ya sea en el estilo por paso de mensajes (utilizando hilos, puertos, y variables de flujo de datos; ver sección 5.6), o en el estilo con estado compartido (utilizando hilos, celdas, y variables de flujo de datos; ver sección 8.2.2), o en un estilo mixto (utilizando tanto celdas como puertos).
- *Modelo concurrente no-determinístico* (ver sección 5.8.1). Este modelo agrega un operador de escogencia no-determinística al modelo concurrente declarativo. Es un paso más hacia el modelo concurrente con estado.

Ellos son menos comunes, pero pueden ser útiles en algunas circunstancias.

### ¿Cuál modelo concurrente utilizar?

¿Cómo decidir cuál enfoque utilizar al momento de escribir un programa concurrente? Estas son algunas reglas generales:

- Escoja el modelo concurrente más sencillo que sea suficiente para su progra-

ma. Por ejemplo, si el uso de la concurrencia no simplifica la arquitectura del programa, entonces utilice un modelo secuencial. Si su programa no tiene ningún no-determinismo observable, tal como clientes independientes interactuando con un servidor, entonces utilice el modelo concurrente declarativo.

- Si es absolutamente necesario usar tanto estado como concurrencia, entonces utilice el enfoque por paso de mensajes o el enfoque con estado compartido. El enfoque por paso de mensajes es, con frecuencia, el mejor, para programas multi agentes, i.e., programas que consisten de entidades autónomas (“agentes”) que se comunican entre sí. Por su lado, el enfoque con estado compartido es, con frecuencia, el mejor para programas centrados en datos, i.e., programas que consisten de un gran repositorio de datos (“base de datos”) al que se accede y se le actualiza concurrentemente. Ambos enfoques se pueden usar al tiempo para diferentes partes de una misma aplicación.
- Haga su programa lo más modular posible y concentre los aspectos concurrentes en el menor número posible de lugares. La mayoría de las veces, gran parte del programa puede ser secuencial o utilizar concurrencia declarativa. Una manera de implementar esto es con la técnica de adaptación de impedancias explicada en la sección 4.8.7. Por ejemplo, los objetos activos se pueden usar como interfaz de objetos pasivos. Si los objetos pasivos se invocan todos desde el mismo objeto activo, entonces los objetos pasivos pueden usar un modelo secuencial.

### *Demasiada concurrencia es malo*

Existe un modelo, el modelo maximalmente concurrente, que tiene aún más concurrencia que el modelo concurrente con estado. En el modelo maximalmente concurrente, cada operación se ejecuta en su propio hilo. El orden de ejecución sólo está restringido por la dependencia de los datos. Este modelo tiene la mayor concurrencia posible.

El modelo maximalmente concurrente ha sido utilizado como la base para lenguajes experimentales de programación paralela. Pero, es difícil de programar con ellos y difícil de implementarlos eficientemente (ver ejercicios, sección 8.7). Esto se debe a que las operaciones tienden a ser de granularidad fina comparadas con el costo de la planificación y la sincronización. El modelo con estado compartido de este capítulo no tiene ese problema puesto que la creación de hilos es explícita. Esto permite al programador controlar la granularidad. No presentamos el modelo maximalmente concurrente con más detalle en este capítulo. Una variación de este modelo se utiliza para la programación por restricciones (ver capítulo 12 (en CTM)).

#### **8.2.2. Utilizando directamente el modelo con estado compartido**

Como lo vimos al comienzo de este capítulo, programar directamente en el modelo con estado compartido puede ser difícil. Esto se debe a que existe un

## Concurrencia con estado compartido

```
fun {PilaNueva}
 Pila={NewCell nil}
proc {Colocar x}
 S in
 {Exchange Pila S X|S}
 end
fun {Sacar}
 X S in
 {Exchange Pila X|S S}
 X
 end
in
pila(colocar:Colocar sacar:Sacar)
end
```

**Figura 8.3:** Pila concurrente.

número potencialmente enorme de intercalaciones, y el programa debe funcionar correctamente para todas ellas. Esta es la razón principal por la que se desarrollaron enfoques de más alto nivel, como los objetos activos y las acciones atómicas. Aún así, algunas veces es útil usar el modelo directamente. Antes de pasar al uso de acciones atómicas, miremos qué se puede hacer directamente en el modelo con estado compartido. Prácticamente, esto se reduce a programar con hilos, procedimientos, celdas, y variables de flujo de datos. En esta sección se presentan algunos ejemplos.

### Pila concurrente

Una *abstracción de datos concurrente* es una abstracción de datos en la cual múltiples hilos pueden ejecutar operaciones de forma simultánea. La primera y más sencilla abstracción de datos concurrente que mostramos es una pila en el estilo objeto. La pila provee operaciones, para colocar y sacar elementos de la pila, no bloqueantes, i.e., ellas nunca esperan, sino que tienen éxito o fallan inmediatamente. Utilizando la operación de intercambio, su implementación es muy compacta, como se muestra en la figura 8.3. La operación de intercambio hace dos cosas: accede al viejo contenido de la celda y le asigna uno nuevo. Como la operación de intercambio es atómica, se puede usar en un contexto concurrente. Como las operaciones para colocar y sacar elementos de la pila tienen una sola operación de intercambio, entonces pueden ser intercaladas en cualquier forma y trabajarán aún correctamente. Cualquier número de hilos puede acceder a la pila concurrentemente, y ella funcionará correctamente. La única restricción es que no se debe intentar una operación *sacar* sobre una pila vacía. En tal caso, se levanta una excepción, e.g., así:

```
fun {Sacar}
 X S in
 try {Exchange Pila X|S S}
 catch failure(...) then raise pilaVacía end end
 X
 end
```

La pila concurrente es sencilla pues cada operación realiza sólo un intercambio. Las cosas se vuelven más complejas cuando una operación de la abstracción realiza más de una operación sobre celdas. En general, para que la operación de la abstracción sea correcta, las operaciones sobre celdas tendrían que realizarse atómicamente. Para garantizar esto de forma sencilla, recomendamos utilizar el enfoque de objetos activos o de acciones atómicas.

### *Simulando una red lenta*

La invocación {Obj M} invoca inmediatamente al objeto Obj y termina cuando la invocación termina. Nos gustaría modificar esto para simular una red asincrónica lenta, donde el objeto se invoca asincrónicamente después de un retraso que representa el retraso de la red. Una solución sencilla que funciona para cualquier objeto, es la siguiente:

```
fun {RedLenta1 Obj D}
 proc {$ M}
 thread
 {Delay D} {Obj M}
 end
 end
end
```

La invocación {RedLenta1 Obj D} devuelve una versión “lenta” del objeto Obj. Cuando se invoca el objeto lento, él espera al menos D ms antes de invocar al objeto original.

**Preservando el orden de los mensajes con paso de testigos** La solución anterior no preserva el orden de los mensajes. Es decir, si el objeto lento se invoca varias veces dentro del mismo hilo, entonces no existe garantía de que los mensajes lleguen en el mismo orden en que fueron enviados. Aún más, si el objeto se invoca desde diversos hilos, las diferentes ejecuciones del objeto pueden solaparse en el tiempo, lo cual podría resultar en un estado inconsistente del objeto. Presentamos una solución que preserva el orden de los mensajes y que garantiza que sólo un hilo a la vez puede ejecutarse dentro del objeto:

*Concurrencia con estado compartido*

```
fun {RedLenta2 Obj D}
C={NewCell listo} in
proc {$ M}
Viejo Nuevo in
{Exchange C Viejo Nuevo}
thread
{Delay D} {Wait Viejo} {Obj M} Nuevo=listo
end
end
end
```

Esta solución utiliza una técnica general denominada paso de testigos<sup>2</sup>, que extrae un orden de ejecución de una parte del programa y lo impone en otra parte. La técnica de paso de testigos se implementa creando una secuencia de variables de flujo de datos  $x_0, x_1, x_2, \dots$ , y pasando pares consecutivos de variables a las operaciones que se deben realizar en ese mismo orden. Una operación que recibe la pareja  $(x_i, x_{i+1})$  realiza los siguientes pasos en orden:

1. Espera hasta que el testigo llegue, i.e., hasta que  $x_i$  sea ligado ( $\{\text{wait } x_i\}$ ).
2. Realiza el cálculo.
3. Envía el testigo a la pareja siguiente, i.e., liga  $x_{i+1}$  ( $x_{i+1}=\text{listo}$ ).

En la definición de `RedLenta2`, cada vez que el objeto lento se invoca, se crea una pareja de variables (`Viejo`, `Nuevo`). Ésta se inserta en la secuencia por medio de la invocación `{Exchange C Viejo Nuevo}`. Como `Exchange` es atómica, esto también funciona en un contexto concurrente donde muchos hilos invocan el objeto lento. Cada pareja comparte una variable con la pareja anterior (`Viejo`) y una con la pareja siguiente (`Nuevo`). Esto coloca efectivamente las invocaciones al objeto en una cola ordenada. Cada invocación se realiza en un hilo nuevo. La invocación, primero espera que la invocación previa haya terminado ( $\{\text{Wait } \text{Viejo}\}$ ), luego invoca al objeto ( $\{\text{Obj } M\}$ ), y finalmente señala que la próxima invocación puede continuar ( $\text{Nuevo}=\text{listo}$ ). La invocación `{Delay D}` se debe realizar antes de `{Wait Viejo}`; de otra manera cada invocación al objeto tomaría por lo menos  $D$  ms, lo cual es incorrecto.

### 8.2.3. Programando con acciones atómicas

Comenzando en la próxima sección, presentamos las técnicas de programación para concurrencia con estado compartido utilizando acciones atómicas. Introducimos los conceptos gradualmente, iniciando con los candados. Refinamos los candados en monitores y transacciones. En la figura 8.4 se muestra la relación jerárquica entre estos tres conceptos.

- Los candados permiten el agrupamiento de unas pocas operaciones atómicas en

---

2. Nota del traductor: del inglés, *token passing*.

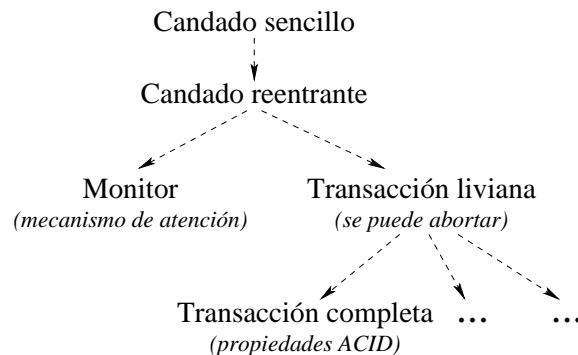


Figura 8.4: La jerarquía de las acciones atómicas.

una gran operación atómica. Con un candado reentrantre<sup>3</sup>, el mismo candado puede proteger partes no contiguas del programa. Un hilo que está dentro de una parte puede volver a entrar al candado, en cualquier parte, sin suspenderse.

- Los monitores son un refinamiento de los candados con puntos de espera. Un punto de espera es una pareja compuesta por una salida y una entrada correspondiente sin código entre ellas. (Los puntos de espera algunas veces se denominan puntos de retraso [5].) Los hilos se pueden “estacionar” por sí mismos en un punto de espera, justo afuera del candado. Los hilos salientes pueden despertar a los hilos estacionados.
- Las transacciones son un refinamiento de los candados con dos salidas posibles: una normal (denominada efectuar<sup>4</sup>) y una excepcional (denominada abortar). La salida excepcional puede ser tomada en cualquier momento durante la transacción. Cuando es tomada, la transacción deja el estado de la ejecución sin cambios, i.e., tal cual estaba a la entrada.

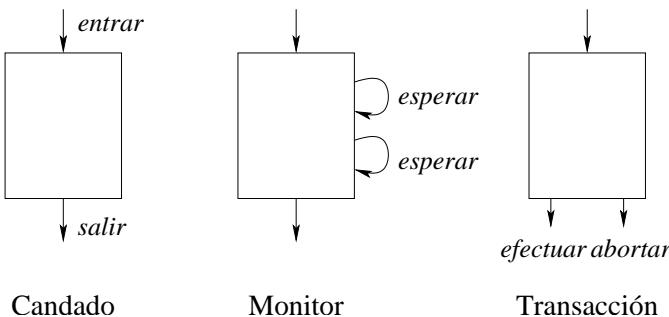
En la figura 8.5 se resumen las principales diferencias entre los tres conceptos. Existen muchas variantes de estos conceptos diseñadas para resolver problemas específicos. Esta sección sólo presenta una breve introducción a las ideas básicas.

### Razonando con acciones atómicas

Considere un programa que utiliza acciones atómicas por todas partes. Probar que el programa es correcto consta de dos partes: probar que cada acción atómica es correcta (cuando es considerada en sí misma) y probar que el programa las usa correctamente. La primera etapa consiste en mostrar que cada acción atómica, e.g., candado, monitor, o transacción, es correcta. Cada acción atomica define una

3. Nota del traductor: *reentrant lock* en inglés.

4. Nota del traductor: *commit* en inglés.

*Concurrencia con estado compartido*

**Figura 8.5:** Diferencias entre acciones atómicas.

abstracción de datos. La abstracción de datos debe tener una afirmación invariante, i.e., una afirmación que es cierta cuando no hay hilos dentro de la abstracción. Este razonamiento es parecido al utilizado con los programas con estado y los objetos activos, salvo que la abstracción de datos puede ser accedida concurrentemente. Como la acción atómica sólo puede tener por dentro un hilo a la vez, entonces podemos usar inducción matemática para mostrar que la afirmación es invariante. Tenemos que probar dos cosas:

- La afirmación se satisface, al momento de definir la abstracción de datos.
- Cuando un hilo sale de la abstracción de datos, la afirmación se satisface.

La existencia del invariante muestra que la acción atómica es correcta. El siguiente paso consiste en mostrar que el programa que usa las acciones atómicas es correcto.

#### 8.2.4. Lecturas adicionales

Existen libros muy buenos sobre programación concurrente. Los cuatro siguientes son particularmente apropiados como compañeros de este libro. Ellos ofrecen más técnicas prácticas y fundamento teórico para los dos paradigmas concurrentes de concurrencia por paso de mensajes y con estado compartido. Al momento de escribir el libro, no conocemos ningún libro que trate con el tercer paradigma concurrente de concurrencia declarativa.

##### *Concurrent Programming in Java*

El primer libro trata la concurrencia con estado compartido: *Concurrent Programming in Java* escrito por Doug Lea [103]. Este libro presenta un amplio conjunto de técnicas de programación prácticas, particularmente apropiadas para Java, un lenguaje popular de programación orientada a objetos (ver capítulos 7 y 8). Sin embargo, esas técnicas se pueden usar con muchos otros lenguajes incluyendo el modelo con estado compartido de este libro. El libro está orientado hacia el enfoque

con estado compartido; el paso de mensajes sólo se menciona lateralmente.

La principal diferencia entre el libro de Java y este capítulo es que el libro de Java supone que los hilos son costosos. Esto es cierto para las implementaciones actuales de Java. Debido a eso, el libro de Java agrega un nivel conceptual entre los hilos y los procedimientos, las tareas, y aconseja al programador planificar múltiples tareas en un hilo. Si los hilos son livianos, este nivel conceptual no se necesita. El rango de técnicas de programación prácticas se amplía y con frecuencia se encuentran soluciones más sencillas. Por ejemplo, contar con hilos livianos facilita la utilización de objetos activos, lo cual, con frecuencia, simplifica la estructura del programa.<sup>5</sup>

### *Concurrent Programming in Erlang*

El segundo libro trata la concurrencia por paso de mensajes; *Concurrent Programming in Erlang* escrito por Joe Armstrong, Mike Williams, Claes Wikström, y Robert Virding [9]. Este libro se complementa con el libro de Doug Lea. Este libro presenta un amplio conjunto de técnicas de programación práctica, basadas en el lenguaje Erlang. El libro está completamente basado en el enfoque por paso de mensajes.

### *Concurrent Programming: Principles and Practice*

El tercer libro es *Concurrent Programming: Principles and Practice* escrito por Gregory Andrews [5]. Este libro es más riguroso que los dos anteriores. En él se explican tanto el enfoque por paso de mensajes como el enfoque con estado compartido. Hay una buena introducción al razonamiento formal con estos conceptos, utilizando afirmaciones invariantes. El formalismo se presenta a un justo nivel de detalle, de manera que es a la vez preciso y utilizable por los programadores. El libro también contempla la evolución histórica de estos conceptos e incluye algunos pasos intermedios, interesantes, que ya no se usan.

### *Transaction Processing: Concepts and Techniques*

El último libro es *Transaction Processing: Concepts and Techniques* escrito por Jim Gray y Andreas Reuter [60]. Este libro es una mezcla exitosa de conocimiento teórico e información práctica empírica. Este libro presenta una mirada a diversos tipos de procesamiento de transacciones, cómo se usan, y cómo se implementan en la práctica. También presenta una pizca de teoría, cuidadosamente seleccionada para ser relevante con la información práctica.

---

5. Algunos casos especiales de objetos activos son posibles si los hilos son costosos; ver e.g., la sección 5.6.1.

### 8.3. Candados

Frecuentemente ocurre que varios hilos desean acceder a un recurso compartido, pero ese recurso sólo puede ser usado por un hilo a la vez. Para ayudar a manejar esta situación, introducimos un concepto del lenguaje denominado candado, que sirve para controlar el acceso al recurso. Un candado controla dinámicamente el acceso a una parte de un programa, denominada una región crítica. La función básica del candado es asegurar un acceso exclusivo a la región crítica, i.e., que solamente un hilo a la vez pueda ejecutarse dentro de ella. Si al recurso compartido sólo se accede desde adentro de la región crítica, entonces el candado se puede utilizar para controlar el acceso al recurso.

El recurso compartido puede estar, bien sea dentro del programa (e.g., un objeto), o fuera de él (e.g., un recurso del sistema operativo). Los candados pueden ayudar en ambos casos. Si el recurso está dentro del programa, entonces el programador puede garantizar que no pueda ser referenciado desde afuera de la región crítica, utilizando alcance léxico. Este tipo de garantías no se pueden dar, en general, para recursos por fuera del programa. Para aquellos recursos, los candados son una ayuda para el programador, pero éste debe ser disciplinado y solamente referenciar el recurso dentro de la región crítica.

Existen diversos tipos de candados que proveen diferentes tipos de control de acceso. La mayoría de ellos se pueden implementar en Oz utilizando las entidades del lenguaje que ya hemos visto (i.e., celdas, hilos, y variables de flujo de datos). Sin embargo, una clase particularmente útil de candado, el candado de hilo reentrant, lo soporta directamente el lenguaje. Las operaciones siguientes se proveen:

- `{NewLock L}` devuelve un candado nuevo.
- `{isLock x}` devuelve `true` si y sólo si `x` referencia un candado.
- `lock x then <decl> end` protege `<decl>` con el candado `x`. Si ningún hilo está ejecutando actualmente alguna declaración protegida por el candado `x`, entonces cualquier hilo puede entrar. Si un hilo está ejecutando actualmente una declaración protegida por un candado, entonces el mismo hilo puede entrar de nuevo, si él encuentra el mismo candado en una ejecución anidada. Un hilo se suspende si trata de entrar a ejecutar una declaración protegida mientras hay otro hilo en una declaración protegida por el mismo candado.

Note que `lock x then ... end` se puede invocar muchas veces con el mismo candado `x`. Es decir, la sección crítica no tiene que ser contigua. El candado asegurará que a lo sumo haya un hilo dentro de cualquiera de las partes que él protege.

#### 8.3.1. Construyendo abstracciones de datos concurrentes con estado

Ahora que hemos introducido los candados, estamos listos para programar abstracciones de datos concurrentes con estado. Hacemos esto por etapas. Presentamos

```

fun {ColaNueva}
 X in
 q(0 X X)
 end

fun {InsCola q(N S E) X}
 E1 in
 E=X|E1 q(N+1 S E1)
 end

fun {ElimDeCola q(N S E) X}
 S1 in
 S=X|S1 q(N-1 S1 E)
 end

```

**Figura 8.6:** Cola (versión declarativa).

```

fun {ColaNueva}
 X C={NewCell q(0 X X)}
 proc {InsCola X}
 N S E1 in
 q(N S X|E1)=@C
 C:=q(N+1 S E1)
 end
 fun {ElimDeCola}
 N S1 E X in
 q(N X|S1 E)=@C
 C:=q(N-1 S1 E)
 X
 end
 in
 cola(insCola:InsCola elimDeCola:ElimDeCola)
 end

```

**Figura 8.7:** Cola (versión secuencial con estado).

una manera sistemática de transformar una abstracción de datos declarativa en una abstracción de datos concurrente con estado. También mostraremos cómo modificar una abstracción de datos secuencial con estado para volverla concurrente.

Ilustramos las diferentes técnicas por medio de un ejemplo sencillo, una cola. No existe ninguna limitación, pues estas técnicas funcionan para cualquier abstracción de datos. Empezamos por una implementación declarativa y mostramos cómo convertirla en una implementación con estado que puede ser usada en un contexto concurrente:

- La figura 8.6 muestra esencialmente la cola declarativa de la sección 3.4.5. (Por

*Concurrencia con estado compartido*

```

fun {ColaNueva}
 X C={NewCell q(0 X X)}
 L={NewLock}
 proc {InsCola X}
 N S E1 in
 lock L then
 q(N S X|E1)=@C
 C:=q(N+1 S E1)
 end
 end
 fun {ElimDeCola}
 N S1 E X in
 lock L then
 q(N X|S1 E)=@C
 C:=q(N-1 S1 E)
 end
 X
 end
in
 cola(insCola:InsCola elimDeCola:ElimDeCola)
end

```

**Figura 8.8:** Cola (versión concurrente con estado y con candado).

brevedad hemos dejado por fuera la función `EsvacíaCola`.) Las operaciones de borrado nunca se bloquean: si la cola está vacía al momento en que se borra un elemento, entonces se devuelve una variable de flujo de datos que se ligará con el siguiente elemento insertado. El tamaño `N` es positivo si existen más inserciones que eliminaciones y es negativo en caso contrario. Todas las funciones tienen la forma `Qsal={ColaOp Qen ...}`, recibiendo una cola de entrada `Qen` y devolviendo una cola de salida `Qsal`. Esta cola funcionará correctamente en un contexto concurrente, en la medida en que pueda ser usada allí. El problema es que el orden de las operaciones de la cola está determinado explícitamente por el programa. Realizar estas operaciones sobre la cola en diferentes hilos causará, de inmediato, la sincronización de los hilos. Esto es casi con seguridad un comportamiento indeseado.

- La figura 8.7 muestra la misma cola, pero en una versión con estado que encapsula los datos de la cola. Esta versión no se puede usar en un contexto concurrente sin algunas modificaciones. El problema es que la encapsulación del estado requiere leer el estado (`@`), realizar la operación, y luego escribir el estado nuevo(`:=`). Si se tienen dos hilos, y cada uno realiza una inserción, entonces ambas lecturas pueden ser realizadas antes de ambas escrituras, lo cual es incorrecto. Una versión concurrente correcta requiere que la secuencia lectura-operación-escritura sea atómica.
- En la figura 8.8 se muestra una versión concurrente con estado de la cola, utilizando un candado para asegurar la atomicidad de la secuencia de operaciones lectura-operación-escritura. La ejecución de operaciones de la cola desde diferentes

```

class Cola
 attr cola
 prop locking

 meth init
 cola:=q(0 X X)
 end

 meth insCola(X)
 lock N S E1 in
 q(N S X|E1)=@cola
 cola:=q(N+1 S E1)
 end
 end

 meth elimDeCola(X)
 lock N S1 E in
 q(N X|S1 E)=@cola
 cola:=q(N-1 S1 E)
 end
 end
end

```

**Figura 8.9:** Cola (versión concurrente orientada a objetos con candado).

hilos no impone ninguna sincronización entre ellos. Esta propiedad es una consecuencia de usar estado.

- En la figura 8.9 se muestra la misma versión, escrita en sintaxis orientada a objetos. La celda es reemplazada por el atributo `cola` y el candado está implícitamente definido por la propiedad `locking`.
- En la figura 8.10 se muestra otra versión concurrente, utilizando un `Exchange` para asegurar la atomicidad. Como hay sólo una única operación de estado (el intercambio), no se necesitan candados. Esta versión funciona gracias a la propiedad de asignación única de las variables de flujo de datos. Un detalle importante: las operaciones aritméticas `N-1` y `N+1` se deben realizar después del intercambio. ¿Por qué?

Ahora, hablemos sobre las ventajas y desventajas de estas soluciones:

- La versión declarativa de la figura 8.6 es la más sencilla, pero no puede ser usada como un recurso compartido entre hilos independientes.
- Las dos versiones concurrentes de las figuras 8.8 y 8.10 son bastante buenas. El uso de un candado en la figura 8.8 es más general, pues un candado se puede usar para volver atómico cualquier conjunto de operaciones. Esta versión se puede escribir con una sintaxis orientada a objetos, tal como se muestra en la figura 8.9. La versión con intercambio mostrada en la figura 8.10 es compacta pero menos

*Concurrencia con estado compartido*

```
fun {ColaNueva}
 X C={NewCell q(0 X X)}
proc {InsCola X}
 N S E1 N1 in
 {Exchange C q(N S X|E1) q(N1 S E1)}
 N1=N+1
 end
fun {ElimDeCola}
 N S1 E N1 X in
 {Exchange C q(N X|S1 E) q(N1 S1 E)}
 N1=N-1
 X
 end
in
 cola(insCola:InsCola elimDeCola:ElimDeCola)
end
```

Figura 8.10: Cola (versión concurrente con estado con intercambio).

general; solamente es posible para operaciones que manipulan una sola secuencia de datos.

### 8.3.2. Espacio de tuplas (“Linda”)

Los espacios de tuplas son una abstracción popular para la programación concurrente. La primera abstracción de espacio de tuplas denominada Linda, fue introducida por David Gelernter en 1985 [55, 29, 30]. Esta abstracción juega dos papeles diferentes. Desde un punto de vista teórico, este es uno de los primeros modelos de programación concurrente. Desde un punto de vista práctico, es una abstracción útil para programas concurrentes. Como tal, puede ser agregada a cualquier lenguaje, produciendo una versión concurrente de ese lenguaje (e.g., C con Linda se denomina C-Linda). Una abstracción de espacio de tuplas se denomina algunas veces un modelo de coordinación y a un lenguaje de programación que contenga esa abstracción se le denomina un lenguaje de coordinación. En su forma básica, la abstracción es sencilla de definir. Ella consta de un multiconjunto ET de tuplas con tres operaciones básicas:

- {ET escribir(T)} agrega la tupla T al espacio de tuplas.
- {ET leer(L T)} espera hasta que el espacio de tuplas contenga al menos una tupla con etiqueta L. Luego, elimina tal tupla y la liga a T.
- {ET leernobloq(L T B)} no espera, sino que termina inmediatamente. Devuelve B=false si el espacio de tuplas no contiene ninguna tupla con etiqueta L. Sino, devuelve B=true, elimina una tupla con etiqueta L y la liga a T.

```

fun {ColaNueva}
 X ET={New EspacioTuplas init}
 proc {InsCola X}
 N S E1 in
 {ET leer(q q(N S X|E1))}
 {ET escribir(q(N+1 S E1))}
 end
 fun {ElimDeCola}
 N S1 E X in
 {ET leer(q q(N X|S1 E))}
 {ET escribir(q(N-1 S1 E))}
 X
 end
 in
 {ET escribir(q(0 X X))}
 cola(insCola:InsCola elimDeCola:ElimDeCola)
 end

```

**Figura 8.11:** Cola (versión concurrente con espacio de tuplas).

Esto simplifica ligeramente la formulación habitual de Linda, en la cual la operación de lectura puede hacer reconocimiento de patrones. Esta abstracción tiene dos propiedades importantes. La primera propiedad es que la abstracción provee una memoria direccionable por el contenido: las tuplas sólo se identifican por sus etiquetas. La segunda propiedad es que los lectores están desacoplados de los escritores. La abstracción no realiza ninguna comunicación diferente de la mencionada arriba entre lectores y escritores.

### *Ejemplo de ejecución*

Creemos primero un espacio de tuplas nuevo:

```
ET={New EspacioTuplas inic}
```

En `ET` podemos leer y escribir todo tipo de tuplas en cualquier orden. El resultado final siempre es el mismo: las lecturas ven las escrituras en el orden en que se escriben. Hacer `{ET escribir(foo(1 2 3))}` agrega una tupla con etiqueta `foo` y tres argumentos. El código siguiente espera hasta que exista una tupla con etiqueta `foo`, y cuando aparece, la elimina y la despliega en el browser:

```
thread {Browse {ET leer(foo $)}} end
```

El código siguiente comprueba inmediatamente si existe una tupla con etiqueta `foo`:

```
local T B in {ET leernobloq(foo T B)} {Browse T#B} end
```

Esta operación no se bloquea, por eso no necesita colocarse en su propio hilo.

*Concurrencia con estado compartido*

```
class EspacioTuplas
 prop locking
 attr dictuplas

 meth inic dictuplas:={NewDictionary} end

 meth AsegurarPresencia(L)
 if {Not {Dictionary.member @dictuplas L}}
 then @dictuplas.L:={ColaNueva} end
 end

 meth Limpiar(L)
 if {@dictuplas.L.size}==0
 then {Dictionary.remove @dictuplas L} end
 end

 meth escribir(Tuple)
 lock L={Label Tuple} in
 {self AsegurarPresencia(L)}
 {@dictuplas.L.insert Tuple}
 end
 end

 meth leer(L ?Tuple)
 lock
 {self AsegurarPresencia(L)}
 {@dictuplas.L.delete Tuple}
 {self Limpiar(L)}
 end
 {Wait Tuple}
 end

 meth leernobloq(L ?Tuple ?B)
 lock
 {self AsegurarPresencia(L)}
 if {@dictuplas.L.size}>0 then
 {@dictuplas.L.delete Tuple} B=true
 else B=false end
 {self Limpiar(L)}
 end
 end
end
```

Figura 8.12: Espacio de tuplas (versión orientada a objetos).

### *Implementando una cola concurrente*

Ahora podemos mostrar otra implementación de una cola concurrente, utilizando espacios de tuplas en lugar de celdas. En la figura 8.11 se muestra cómo se hace. El espacio de tuplas `ET` contiene una única tupla `q(N S E)` que representa el estado de la cola. El espacio de tuplas se inicializa con la tupla `q(0 X X)` que representa una cola vacía. No se necesita ningún candado debido a que la operación `leer` elimina atómicamente la tupla del espacio de tuplas. Esto quiere decir que se puede considerar a la tupla como un único testigo, el cual se pasa entre el espacio de tuplas y las operaciones sobre la cola. Si existieran dos operaciones de `InsCola` concurrentes, solamente una conseguiría la tupla y la otra tendría que esperar. Este es otro ejemplo de la técnica de paso de testigos introducida en la sección 8.2.2.

### *Implementando espacios de tuplas*

Un espacio de tuplas se puede implementar con un candado, un diccionario, y una cola concurrente. En la figura 8.12 se muestra una implementación sencilla en un estilo orientado a objetos. Esta implementación es completamente dinámica; en cualquier momento se pueden leer y escribir tuplas con cualquier etiqueta. Las tuplas se almacenan en un diccionario. La llave es la etiqueta de la tupla y la entrada es una cola de tuplas con esa etiqueta. Los métodos en mayúscula `AsegurarPresencia` y `Limpiar` son privados a la clase `EspacioTuplas` e invisibles para los usuarios de los objetos espacio de tuplas (ver sección 7.3.3). La implementación realiza una correcta administración de la memoria: una entrada nueva se agrega con la primera ocurrencia de una etiqueta en particular; y cuando la cola está vacía, la entrada se elimina.

La implementación del espacio de tuplas utiliza una cola concurrente con estado la cual corresponde a una versión ligeramente extendida de la figura 8.8. Sólo agregamos una operación, una función que devuelve el tamaño de la cola, i.e., el número de elementos que contiene. Nuestra cola extiende la de la figura 8.8 así:

```
fun {ColaNueva}
 ...
 fun {Tamaño}
 lock L then @C.1 end
 end
 in
 cola(insCola:InsCola elimDeCola:ElimDeCola tamaño:Tamaño)
 end
```

Extenderemos esta cola de nuevo para implementar los monitores.

#### **8.3.3. Implementando candados**

Los candados se pueden definir en el modelo concurrente con estado utilizando celdas y variables de flujo de datos. Primero mostramos la definición de un candado sencillo, luego un candado que maneja excepciones correctamente, y finalmente

*Concurrencia con estado compartido*

```
fun {CandadoSencillo}
 Testigo={NewCell listo}
 proc {Candado P}
 Viejo Nuevo in
 {Exchange Testigo Viejo Nuevo}
 {Wait Viejo}
 {P}
 Nuevo=listo
 end
 in
 `candado`(`candado`:Candado)
 end
```

**Figura 8.13:** Candado (versión no reentrante sin manejo de excepciones).

```
fun {CandadoSencilloCorrecto}
 Testigo={NewCell listo}
 proc {Candado P}
 Viejo Nuevo in
 {Exchange Testigo Viejo Nuevo}
 {Wait Viejo}
 try {P} finally Nuevo=listo end
 end
 in
 `candado`(`candado`:Candado)
 end
```

**Figura 8.14:** Candado (versión no reentrante con manejo de excepciones).

un candado de hilo reentrantte. Los candados predefinidos que provee el sistema son candados de hilo reentrantte con la semántica que damos acá, pero con una implementación de bajo nivel más eficiente.

Un candado sencillo es un procedimiento  $\{L\ P\}$  que recibe como argumento un procedimiento sin argumentos  $P$  y ejecuta  $P$  en una sección crítica. Cualquier hilo que trate de entrar al candado mientras exista un hilo adentro se suspenderá. Se dice que este candado es sencillo pues un hilo que está dentro de una sección crítica no puede entrar a ninguna otra sección crítica protegida por el mismo candado. Primero tendría que salir de la sección crítica inicial. Los candados sencillos se pueden crear con la función `CandadoSencillo` definida en la figura 8.13. Si múltiples hilos tratan de acceder el cuerpo del candado, entonces sólo a uno se le permite el acceso y los otros hacen cola. Cuando un hilo deja la sección crítica, se le otorga el acceso al siguiente hilo en la cola. Para esto se utiliza la técnica de paso de testigos vista en la sección 8.2.2.

¿Pero qué sucede si el cuerpo  $\{P\}$  del candado lanza una excepción? El candado

```

fun {NuevoCandado}
 Testigo={NewCell listo}
 HiloAct={NewCell listo}
 proc {Candado P}
 if {Thread.this}==@HiloAct then
 {P}
 else Viejo Nuevo in
 {Exchange Testigo Viejo Nuevo}
 {Wait Viejo}
 HiloAct:={Thread.this}
 try {P} finally
 HiloAct:=listo
 Nuevo=listo
 end
 end
 in
 `candado`(`candado`:Candado)
 end

```

**Figura 8.15:** Candado (versión reentrantre con manejo de excepciones).

de la figura 8.13 no funciona pues Nuevo nunca será ligado. Podemos resolver este problema con una declaración **try**. En la figura 8.14 se presenta una versión de candado sencillo con manejo de excepciones. La declaración **try**  $\langle\text{decl}\rangle_1$  **finally**  $\langle\text{decl}\rangle_2$  **end** es un azúcar sintáctico que asegura que  $\langle\text{decl}\rangle_2$  sea ejecutada, tanto en el caso normal como en el caso excepcional, i.e., una excepción no impide que el candado sea liberado.

Un candado de hilo reentrantre extiende el candado sencillo para permitir que el mismo hilo entre a otras regiones críticas protegidas por el mismo candado. Incluso es posible anidar secciones críticas protegidas por el mismo candado. Otros hilos que quieran adquirir el candado harán cola hasta que P termine. Cuando el candado se libere, el turno se otorga al hilo que está de primero en la cola. En la figura 8.15 se muestra cómo definir candados de hilo reentrantre. Aquí se supone que cada hilo tienen un identificador único T el cual es diferente del literal listo y se obtiene invocando al procedimiento {Thread.this T}. Las asignaciones de HiloAct tienen que hacerse exáctamente en los lugares donde están. ¿Qué funcionaría mal si intercambiamos {Wait Viejo} y HiloAct:={Thread.this} o intercambiamos HiloAct:=listo y Nuevo=listo?

#### 8.4. Monitores

Los candados son una herramienta importante para construir abstracciones concurrentes en un modelo con estado, pero no son suficientes. Por ejemplo,

*Concurrencia con estado compartido*

considere el caso sencillo de una memoria intermedia limitada. Un hilo puede querer colocar un elemento en la memoria intermedia. No es suficiente proteger la memoria intermedia con un candado. ¿Qué pasa si la memoria intermedia está llena? ¡El hilo entra y no puede hacer nada! Lo que realmente deseamos es que exista una manera para que el hilo espere hasta que haya espacio en la memoria intermedia, y luego si continúe. Esto no se puede hacer solamente con candados. Se necesita que de alguna manera unos hilos se coordinen con otros. Por ejemplo, un hilo que elimina un elemento de la memoria intermedia puede notificar a otro, que quiere colocar uno, que la memoria no está llena.

La manera estándar de coordinar hilos en un modelo con estado es utilizando monitores. Los monitores fueron introducidos por Brinch Hansen [22, 23] y desarrollados posteriormente por Hoare [72]. Los monitores siguen siendo ampliamente usados; e.g., ellos son un concepto básico en el lenguaje Java [103]. Un monitor es un candado extendido con control para programar cómo los hilos en espera entran y salen del candado. Este control posibilita el uso del monitor como un recurso que se comparte entre diferentes actividades concurrentes. Existen varias formas de ofrecer este control. Típicamente, un monitor tiene, bien sea un conjunto de hilos en espera, o varias colas de hilos en espera. El caso más sencillo es cuando hay un conjunto; lo consideraremos primero.

El monitor agrega una operación `wait` y una operación `notify` a las operaciones de entrada y salida del candado. (Algunas veces `notify` es denominada `signal`.) Las operaciones `wait` y `notify` solamente se pueden usar desde adentro del monitor. Una vez dentro del monitor, un hilo puede realizar explícitamente una operación `wait`; inmediatamente después el hilo se suspende, entra en el conjunto de espera del monitor, y libera el candado del monitor. Cuando un hilo realiza la operación `notify`, permite que un hilo en el conjunto de espera continúe. Este hilo intenta conseguir el candado del monitor de nuevo. Si tiene éxito, continúa ejecutándose desde donde había quedado.

Primero presentamos una definición informal de los monitores. Luego programamos algunos ejemplos, tanto con monitores como en el modelo concurrente declarativo. Esto nos permitirá comparar los dos enfoques. Concluimos la sección presentando una implementación de monitores en el modelo concurrente con estado compartido.

#### 8.4.1. Definición

Existen diversas variaciones de monitores, con ligeras diferencias semánticas. Primero explicamos la versión de Java porque es la más sencilla y la más popular. (En la sección 8.4.5 se presenta una versión alternativa.) La siguiente definición es tomada de [102]. En Java, un monitor siempre hace parte de un objeto. Un monitor es un objeto con un candado interno y un conjunto de espera. Los métodos de los objetos se pueden proteger con el candado comentándolos como `synchronized`. Existen tres operaciones para administrar el candado: `wait`, `notify`, y `notifyAll`. Estas operaciones sólo pueden ser invocadas por los hilos que tengan el candado, y

tienen el significado siguiente:

- La operación `wait` hace lo siguiente:
  - El hilo actual se suspende.
  - El hilo se coloca en el conjunto de espera interno del objeto.
  - El candado del objeto se libera.
- La operación `notify` hace lo siguiente:
  - Si existe algún hilo en el conjunto de espera interno del objeto, entonces se elimina uno cualquiera,  $T$ , arbitrariamente.
  - $T$  intenta conseguir el candado, justo como cualquier otro hilo. Esto significa que  $T$  se suspenderá siempre por un tiempo corto, hasta que el hilo que notifica libere el candado.
  - $T$  reanuda la ejecución en el punto donde la suspendió.
- La operación `notifyAll` es similar a `notify` salvo que realiza las etapas mencionadas para todos los hilos en el conjunto de espera interno. El conjunto de espera queda, por lo tanto, vacío.

En los ejemplos que siguen, suponemos que existe una función `NuevoMonitor` con la especificación siguiente:

- $M = \{NuevoMonitor\}$  crea un monitor con las operaciones  $\{M.\text{'lock'}$ } (procedimiento candado del monitor),  $\{M.wait\}$  (operación `wait`),  $\{M.notify\}$  (operación `notify`), y  $\{M.notifyAll\}$  (operación `notifyAll`).

De la misma manera que con los candados, suponemos que el candado del monitor es de hilo reentrant y maneja correctamente las excepciones. En la sección 8.4.4 se explica cómo se implementa el monitor.

Los monitores se diseñaron para construir abstracciones de datos concurrentes basadas en estado compartido. Para facilitar el uso de monitores, algunos lenguajes los proveen como abstracciones lingüísticas. Esto permite que el compilador garantice que las operaciones `wait` y `notify` se ejecuten solamente dentro del candado del monitor. Esto también facilita al compilador el asegurar propiedades de seguridad, e.g., que a las variables compartidas sólo se tenga acceso a través del monitor [24].

#### 8.4.2. Memoria intermedia limitada

En el capítulo 4, mostramos cómo implementar declarativamente una memoria intermedia limitada, con comunicación por flujos, tanto ansiosa como perezosa. En esta sección la implementamos con un monitor. Luego, comparamos esta solución con las dos implementaciones declarativas. La memoria intermedia limitada es un objeto con tres operaciones:

- $B = \{\text{New MemIntermedia inic(N)}\}$ : crea una memoria intermedia nueva  $B$  de tamaño  $N$ .

*Concurrencia con estado compartido*

```
class MemIntermedia
 attr
 mem primero último n i

 meth inic(N)
 mem:={NewArray 0 N-1 null}
 primero:=0 último:=0 n:=N i:=0
 end

 meth colocar(X)
 ... % espera hasta que i<n
 % ahora agrega un elemento:
 @mem.@último:=X
 último:=(@último+1) mod @n
 i:=@i+1
 end

 meth obtener(X)
 ... % espera hasta que i>0
 % ahora elimina un elemento:
 X=@mem.@primero
 primero:=(@primero+1) mod @n
 i:=@i-1
 end
end
```

**Figura 8.16:** Memoria intermedia limitada (definición parcial de la versión con monitor).

- {B colocar(x)}: coloca el elemento x en la memoria intermedia limitada. Si la memoria está llena, esta operación se bloqueará hasta que haya campo para el elemento.
- {B obtener(x)}: elimina el elemento x de la memoria. Si la memoria está vacía, esta operación se bloqueará hasta que exista al menos un elemento.

La idea de la implementación es sencilla: las operaciones colocar y obtener esperarán, cada una, hasta que la memoria no esté llena y no esté vacía, respectivamente. Esto nos lleva a la definición parcial de la figura 8.16. La memoria intermedia utiliza un arreglo de  $n$  elementos, indexado por primero y último. El arreglo es cíclico: después del elemento  $n - 1$  viene el elemento 0. El máximo tamaño del arreglo es  $n$ ;  $i$  representa el número de entradas usadas del arreglo. Ahora codifiquémosla con un monitor. La solución ingenua es la siguiente (donde M es un registro monitor):

```

meth colocar(X)
 {M.`lock` proc {$}
 if @i>=@n then {M.wait} end
 @mem.@último:=X
 último:=(@último+1) mod @n
 i:=@i+1
 {M.notifyAll}
 end}
 end

```

Es decir, si la memoria está llena, entonces {M.wait} sencillamente espera hasta que deje de estarlo. Cuando obtener(X) elimina un elemento, él hace un {M.notifyAll}, el cual despierta al hilo dormido. Esta solución ingenua no es suficientemente buena, pues no hay garantía de que la memoria no estará llena después de la espera. Cuando el hilo libera el candado con la invocación {M.wait}, otros hilos pueden entrar desapercibidamente a colocar y eliminar elementos. Una solución correcta consiste en esperar tanto como sea necesario, comprobando la comparación  $@i >= @n$  en todo momento. Esto nos lleva al código siguiente:

```

meth colocar(X)
 {M.`lock` proc {$}
 if @i>=@n then {M.wait} {self colocar(X)}
 else
 @mem.@último:=X
 último:=(@último+1) mod @n
 i:=@i+1
 {M.notifyAll}
 end
 end}
 end

```

Después de esperar, se invoca al método colocar de nuevo, para volver a realizar la comprobación. Como el candado es reentrant, él dejará al hilo entrar de nuevo. La comprobación se hace dentro de la sección crítica, lo cual elimina cualquier interferencia con otros hilos. Ahora podemos colocar todas las piezas juntas. En la figura 8.17 se presenta la solución final. El método `inic` crea el monitor y almacena los procedimientos del monitor entre los atributos del objeto. Los métodos `colocar` y `obtener` utilizan la técnica vista arriba para esperar en un ciclo.

Comparemos esta versión con las versiones concurrentes declarativas del capítulo 4. En la figura 4.14 se presenta la versión ansiosa y en la figura 4.27 la versión perezosa. La versión perezosa es la más sencilla. Cualquiera de las versiones concurrentes declarativas puede ser usada en cualquier parte donde no exista un no-determinismo observable, e.g., en conexiones punto a punto que conectan un lector y un escritor. Otro caso es cuando hay múltiples lectores que leen los mismos ítems. La versión con monitor se puede utilizar cuando el número de escritores independientes es mayor que uno o cuando el número de lectores independientes es mayor que uno.

*Concurrencia con estado compartido*

```

class MemIntermedia
 attr m mem primero último n i

 meth inic(N)
 m:={NuevoMonitor}
 mem:={NewArray 0 N-1 null}
 n:=N i:=0 primero:=0 último:=0
 end

 meth colocar(X)
 {@m.`lock` proc {$}
 if @i>=@n then {@m.wait} {self colocar(X)}
 else
 @mem.@último:=X
 último:=(@último+1) mod @n
 i:=@i+1
 {@m.notifyAll}
 end
 end}
 end

 meth obtener(X)
 {@m.`lock` proc {$}
 if @i==0 then {@m.wait} {self obtener(X)}
 else
 X=@mem.@primero
 primero:=(@primero+1) mod @n
 i:=@i-1
 {@m.notifyAll}
 end
 end}
 end
end

```

**Figura 8.17:** Memoria intermedia limitada (versión con monitor).

#### 8.4.3. Programando con monitores

La técnica que usamos en la implementación de la memoria intermedia limitada, es una técnica general para programar con monitores. La explicamos ahora en un contexto general. Por simplicidad, suponga que estamos definiendo una abstracción de datos completamente concurrente en una sola clase. La idea es que cada método se encuentra en una sección crítica vigilada, i.e., existe una condición booleana que debe ser cierta para que un hilo pueda entrar al cuerpo del método. Si la condición es falsa, entonces el hilo espera hasta que se vuelva cierta. Un método vigilado se denomina también una sección crítica condicional.

Los métodos vigilados se implementan utilizando las operaciones `wait` y `notifyAll`. Este es un ejemplo en pseudocódigo:

```

meth cabezaDeMétodo
 lock
 while not <expr> do wait;
 <decl>
 notifyAll;
 end
end

```

En este ejemplo,  $\langle \text{expr} \rangle$  es la guarda y  $\langle \text{decl} \rangle$  el cuerpo siendo vigilado. Cuando invocan el método, el hilo entra al candado y espera por que se cumpla la condición en un ciclo **while**. Si la condición es cierta, entonces el cuerpo se ejecuta inmediatamente. Si la condición es falsa, entonces el hilo queda en espera. Cuando el hilo termina la espera, entonces el ciclo se repite, i.e., la condición se comprueba de nuevo. Esto garantiza que la condición será cierta cuando el cuerpo se ejecute. Justo antes de terminar, el método notifica a todos los otros hilos en espera que pueden continuar. Todos los hilos se despertarán e intentarán entrar al candado del monitor para comprobar su condición. El primero para el que su condición sea cierta continuará su ejecución. Los otros esperarán de nuevo.

#### 8.4.4. Implementando monitores

Mostremos cómo implementar monitores en el modelo concurrente con estado compartido. Esto nos precisa su semántica. En la figura 8.20 se muestra la implementación, la cual es de hilo reentrant y maneja correctamente las excepciones. También se implementa la exclusión mutua utilizando el candado obtener-liberar de la figura 8.19, y el conjunto de espera utilizando la cola extendida mostrada en la figura 8.18. Implementar el conjunto de espera con una cola evita la inanición pues ofrece al hilo que más haya esperado, la primera oportunidad de entrar al monitor.

La implementación sólo funciona si `M.wait` siempre se ejecuta dentro de un candado activo. Para ser prácticos, la implementación debería extenderse para comprobar esto en tiempo de ejecución. Dejamos esta sencilla extensión al lector. Otro enfoque consiste en embeber la implementación dentro de una abstracción lingüística que haga cumplir esto estáticamente.

Cuando se escriben programas concurrentes en el modelo con estado compartido, normalmente es más sencillo utilizar el enfoque de flujo de datos que el de monitores. Por eso, la implementación de Mozart no realiza optimizaciones especiales para mejorar el desempeño del monitor. Sin embargo, la implementación de la figura 8.20 se puede optimizar de muchas maneras, lo cual es importante si las operaciones de monitor se hacen frecuentes.

#### *Cola concurrente extendida*

Para la implementación del monitor, extendemos la cola concurrente de la figura 8.8 con las tres operaciones: `Tamaño`, `ElimTodo`, y `ElimNoBloq`. Esto nos lleva a la definición de la figura 8.18.

Esta cola es un buen ejemplo de utilidad de los candados reentrantes. Sólo mire

*Concurrencia con estado compartido*

```
fun {ColaNueva}
 ...
 fun {Tamaño}
 lock L then @C.1 end
 end
 fun {ElimTodo}
 lock L then
 X q(_ S E)=@C in
 C:=q(0 X X)
 E=nil S
 end
 end
 fun {ElimNoBloq}
 lock L then
 if {Tamaño}>0 then [{ElimDeCola}] else nil end
 end
 end
in
 cola(insCola:InsCola elimDeCola:ElimDeCola tamaño:Tamaño
 elimTodo:ElimTodo elimNoBloq:ElimNoBloq)
end
```

**Figura 8.18:** Cola (versión concurrente con estado, extendida).

la definición de `ElimNoBloq`: se invoca a `Tamaño` y `ElimDeCola`. Esto sólo funciona si el candado es reentrant.

#### *Candado obtener-liberar reentrant*

Para la implementación del monitor extendemos el candado reentrant de la figura 8.15 a un candado obtener-liberar. En éste se exportan las acciones de obtener y liberar un candado como operaciones separadas, `Obtener` y `Liberar`. Esto nos lleva a la definición de la figura 8.19. Las operaciones tienen que estar separadas porque se usan tanto en `LockM` como en `WaitM`.

#### **8.4.5. Otra semántica para monitores**

En el concepto de monitor que introdujimos arriba, la operación `notify` tiene un solo efecto: hace que un hilo en espera deje el conjunto de espera. Este hilo entonces trata de obtener el candado del monitor. El hilo notificado no libera inmediatamente el candado del monitor. Cuando lo hace, el hilo notificado compite con los otros hilos por el candado. Esto significa que una afirmación que se satisface en el momento de la notificación puede dejar de satisfacerse en el momento en que el hilo notificado consigue el candado. Por eso es que un hilo entrante debe comprobar de nuevo la condición.

Existe una variante que es, a la vez, más eficiente y más fácil para razonar con

```

fun {NuevoCandadoOL}
 Testigo1={NewCell listo}
 Testigo2={NewCell listo}
 HiloAct={NewCell listo}

 proc {ObtenerCandado}
 if {Thread.this}\=@HiloAct then Viejo Nuevo in
 {Exchange Testigo1 Viejo Nuevo}
 {Wait Viejo}
 Testigo2:=Nuevo
 HiloAct:={Thread.this}
 end
 end

 proc {LiberarCandado}
 HiloAct:=listo
 listo=@Testigo2
 end
in
 `lock`(obtener:ObtenerCandado liberar:LiberarCandado)
end

```

**Figura 8.19:** Candado obtener-liberar (versión reentrant).

ella. La idea es que **notify** haga dos operaciones atómicamente: primero, hacer que un hilo en espera deje el conjunto de espera (como antes), y segundo, que inmediatamente después pase el candado del monitor a ese hilo. De ese modo el hilo notificante sale del monitor. Esto tiene la ventaja de que una afirmación que se satisface al momento de la notificación seguirá siendo cierta cuando el hilo notificado continúe su ejecución. La operación **notifyAll** no tiene sentido en esta variante, por lo tanto desaparece.

Normalmente, los lenguajes que implementan los monitores de esta manera permiten declarar varios conjuntos de espera. Un conjunto de espera es visto por el programador como una instancia de una abstracción de datos denominada una condición. El programador puede crear nuevas instancias de condiciones, las cuales se denominan variables de condición. Cada variable de condición *c* tiene dos operaciones, *c.wait* y *c.notify*.

Podemos reimplementar la memoria intermedia limitada utilizando esta variante. La nueva memoria intermedia limitada tendrá dos condiciones que llamaremos no-vacía y no-llena. El método *colocar* espera que se satisfaga la condición no-llena y luego señala la condición no-vacía. El método *obtener* espera que se satisfaga la condición no-vacía y luego señala la condición no-llena. Esto es más eficiente que la implementación previa pues es más selectiva. En lugar de despertar todos los hilos en espera del monitor con la operación *notifyAll*, sólo se despierta un hilo del conjunto de espera acertado. Dejamos la codificación real para un ejercicio.

*Concurrencia con estado compartido*

```
fun {NuevoMonitor}
Q={ColaNueva}
L={NuevoCandadoOL}

proc {LockM P}
 {L.obtener} try {P} finally {L.liberar} end
end

proc {WaitM}
X in
 {Q.insert X} {L.liberar} {Wait X} {L.obtener}
end

proc {NotifyM}
U={Q.elimNoBloq} in
 case U of [X] then X=listo else skip end
end

proc {NotifyAllM}
L={Q.elimTodo} in
 for X in L do X=listo end
end
in
monitor(`lock':LockM wait:WaitM notify:NotifyM
 notifyAll:NotifyAllM)
end
```

**Figura 8.20:** Implementación del monitor.

---

## 8.5. Transacciones

Las transacciones fueron introducidas como un concepto básico para la administración de grandes bases de datos compartidas. Idealmente, las bases de datos deben soportar una velocidad alta de actualizaciones concurrentes al mismo tiempo que mantienen la coherencia de los datos y sobreviven a las caídas de los sistemas. Este no es un problema fácil de resolver. Para ver por qué, considere una base de datos representada como una arreglo grande de celdas. Muchos clientes desean actualizar la base de datos de manera concurrente. Una implementación ingenua consiste en utilizar un solo candado para proteger el arreglo entero. Esta solución es poco práctica por muchas razones. Un problema es que un cliente que gasta un minuto realizando una operación no dejará que ninguna otra operación se realice durante ese tiempo. Este problema se puede resolver con transacciones.

El término “transacción” ha adquirido un significado preciso: es cualquier operación que satisface las cuatro propiedades ACID [19, 60]. ACID es un acrónimo:

- A significa atómica: ninguno de los estados intermedios de la ejecución de una transacción es observable. Es como si la transacción sucediera instantáneamente

o no sucediera en absoluto. La transacción se puede completar normalmente (se efectuó<sup>6</sup>) o puede ser cancelada (se aborta).

■ C significa consistente: los cambios observables del estado respetan los invariantes del sistema. La consistencia está estrechamente relacionada con la atomicidad. La diferencia es que la consistencia es responsabilidad del programador, mientras que la atomicidad es responsabilidad de la implementación del sistema de transacciones.

■ I significa aislamiento<sup>7</sup>: se pueden ejecutar varias transacciones de forma concurrente sin que interfieran entre ellas. Ellas se ejecutan como si fueran secuenciales. A esta propiedad también se le denomina serializabilidad<sup>8</sup>; significa que las transacciones tienen una semántica de intercalación, igual a la del modelo de computación subyacente. Hemos abstraído la semántica de intercalación del modelo al nivel de las transacciones.

■ D significa duración: los cambios observables del estado se conservan (sobreviven) a lo largo de las caídas del sistema. La duración es denominada, con frecuencia, persistencia. La implementación de la duración requiere de un almacén estable (tal como un disco) que almacene los cambios observables del estado.

Este capítulo solamente presenta una introducción breve a los sistemas de transacciones. La referencia clásica sobre transacciones es el libro [19] de Bernstein, Hadzilacos, y Goodman. Este libro es claro y preciso e introduce la teoría de transacciones con el formalismo justo para ayudar a la intuición. Desafortunadamente, el libro ya no se edita más. Con frecuencia, las bibliotecas buenas tienen una copia. Otro buen libro sobre transacciones es [60] de Gray y Reuter. En el libro [182], de Weikum y Vossen, se presenta un tratamiento extenso y matemáticamente riguroso de las transacciones.

### *Transacciones livianas (ACI)*

En aplicaciones diferentes a las bases de datos, no siempre se necesitan las cuatro propiedades. Esta sección utiliza el término “transacción” en un sentido más fino el cual es más cercano a las necesidades de la programación concurrente de propósito general. Siempre que exista un riesgo de confusión, la llamaremos una transacción liviana. Sencillamente, una transacción liviana no es más que una acción atómica abortable, con las propiedades ACID salvo la D (duración). Una transacción liviana puede efectuarse o abortar. La falla (aborto) puede ser debido a una causa interna al programa (e.g., debido a conflictos de acceso a datos compartidos) o externa al programa (e.g., debido al fallo de una parte del sistema, como un disco o la red).

---

6. Nota del traductor: el término usado en la versión en inglés es *commit*.

7. Nota del traductor: del inglés *isolation*.

8. Nota del traductor: del inglés *serializability*. Hasta el momento no he encontrado una buena manera de traducirla diferente a la invención de esta palabra.

### ***Motivaciones***

Ya vimos que una motivación para introducir las transacciones era incrementar el rendimiento de los accesos concurrentes a una base de datos. Miremos otras motivaciones adicionales. Una segunda motivación es la programación concurrente con restricciones. La mayoría de las rutinas tienen dos maneras posibles de terminar: o terminan normalmente o lanzan una excepción. Normalmente las rutinas se comportan atómicamente si terminan normalmente, i.e., quien la invoca ve el estado inicial y el resultado pero no ve nada en el intermedio. Ese no es el caso cuando se lanza una excepción. La rutina pudo haber colocado parte del sistema en un estado inconsistente. ¿Cómo podemos evitar esta situación indeseable? Existen dos soluciones:

- El inovador puede limpiar el desorden de la rutina invocada. Esto significa que la rutina debe escribirse cuidadosamente de manera que su desorden sea limitado en extensión.
- La rutina puede estar dentro de una transacción. Esta solución es más difícil de implementar, pero puede hacer que el programa sea mucho más sencillo. Lanzar una excepción corresponde a abortar la transacción.

Una tercera motivación es la tolerancia a las fallas. Las transacciones livianas son importantes para escribir aplicaciones tolerantes a las fallas. Con respecto a un componente, e.g., una aplicación realizando una transacción, definimos una falla como un comportamiento incorrecto en uno de sus subcomponentes. Idealmente, la aplicación debería continuar correctamente cuando hay fallas, i.e., debería ser tolerante a las fallas. Cuando una falla sucede, una aplicación, tolerante a las fallas, debe realizar tres pasos: (1) detectar la falla, (2) mantener la falla en una parte limitada de la aplicación, y (3) reparar cualquier problema causado por la falla. Las transacciones livianas son un buen mecanismo para el aislamiento de las fallas.

Una cuarta motivación es la administración de los recursos. Las transacciones livianas permiten la adquisición de múltiples recursos sin provocar que una aplicación concurrente se detenga a causa de una situación indeseable denominada muerte súbita. Esta situación se explica abajo.

**Punto de vista desde el lenguaje núcleo** Desviémonos brevemente y examinemos las transacciones desde el punto de vista de los modelos de computación. La solución transaccional satisface uno de nuestros criterios para agregar un concepto al modelo de computación, a saber que los programas en el modelo extendido sean más sencillos. ¿Pero, cuál es exactamente el modelo que se debe agregar? Este es aún un tema abierto de investigación; en nuestra opinión un tema muy importante. Algun día, la solución a esta pregunta formará parte importante de todos los lenguajes de programación de propósito general. En esta sección no resolvemos este problema. Implementaremos las transacciones como una abstracción en el modelo concurrente con estado, sin modificar el modelo.

### 8.5.1. Control de la concurrencia

Considere una base de datos grande a la que acceden muchos clientes al mismo tiempo. ¿Qué implicaciones tiene sobre las transacciones? Significa que las transacciones deben ser concurrentes, pero conservando la propiedad de serialibilidad. La implementación debe permitir transacciones concurrentes y asegurarse que sigan siendo serializables. Existe una fuerte tensión entre estos dos requerimientos, pues no son fáciles de satisfacer simultáneamente. El diseño de sistemas de transacciones que satisfagan ambos requerimientos, ha llevado a la construcción de una teoría rica y de muchos algoritmos ingeniosos [19, 60].

El control de la concurrencia consiste en el conjunto de técnicas utilizadas para construir y programar sistemas concurrentes con propiedades transaccionales. Introducimos estas técnicas y los conceptos en que se basan y mostramos un algoritmo práctico. Técnicamente hablando, nuestro algoritmo hace un control optimista de la concurrencia con dos fases estrictas de adquisición y liberación de candados, y evitando la muerte súbita. Explicamos el significado de estos términos y por qué son importantes. Nuestro algoritmo es interesante porque es a la vez práctico y sencillo. Una implementación completa funcional sólo consta de dos páginas de código.

#### *Candados y marcas de tiempo*

Los dos enfoques más ampliamente utilizados para el control de la concurrencia son los candados y las marcas de tiempo:

- Control de la concurrencia basado en candados. Cada entidad con estado tiene un candado que controla el acceso a la entidad. Por ejemplo, una celda puede tener un candado que sólo permita que una transacción la use al tiempo. Para utilizar una celda, la transacción debe tener un candado sobre ella. Los candados son importantes para hacer cumplir la serialibilidad. Esta es una propiedad de seguridad, i.e., una afirmación que siempre es cierta durante la ejecución. Una propiedad de seguridad no es más que un invariante del sistema. En general, los candados permiten restringir el comportamiento del sistema de manera que sea seguro.
- Control de la concurrencia basado en marcas de tiempo. Cada transacción es marcada con una marca de tiempo lo cual le otorga una prioridad. Las marcas de tiempo se toman de un conjunto ordenado, algo así como los tiquetes de turnos numerados en las tiendas para asegurar que los clientes sean atendidos en orden. Las marcas de tiempo son importantes para asegurar que la ejecución progrese, como por ejemplo, que cada transacción finalmente terminará o abortará. Esta es una propiedad de vitalidad,<sup>9</sup> i.e., una afirmación que finalmente siempre será cierta.

---

9. Nota del traductor: del inglés *liveness*. Vitalidad entendida como la cualidad de tener vida.

*Concurrencia con estado compartido*

Las propiedades de seguridad y vitalidad describen cómo se comporta un sistema en función del tiempo. Para razonar con estas propiedades, es importante tener cuidado sobre el significado exacto de los términos “siempre es cierta” y “finalmente será cierta.” Estos términos son relativos a la etapa actual de ejecución. Una propiedad siempre es cierta, en el sentido técnico, si es cierta en todas las etapas de la ejecución comenzando por la etapa actual. Una propiedad finalmente será cierta, en el sentido técnico, si existe al menos una etapa de la ejecución, en el futuro, donde la propiedad sea cierta. Podemos combinar siempre y finalmente para hacer propiedades más complicadas. Por ejemplo, una propiedad que siempre finalmente será cierta significa que en cada etapa de la ejecución, a partir de la actual, la propiedad finalmente será cierta. La propiedad “una transacción activa finalmente abortará o se efectuará” es de esta clase. Este estilo de razonamiento se puede presentar con una sintaxis y semántica formal, lo cual lleva a una variación de la lógica, denominada lógica temporal.

*Planificación optimista y pesimista*

Existen muchos algoritmos para el control de la concurrencia, los cuales varían en diferentes ejes. Uno de los ejes es el grado de optimismo o pesimismo del algoritmo. Introducimos esto con dos ejemplos tomados de la vida real. Ambos ejemplos tienen que ver con viajar, en avión o en tren.

Con frecuencia, las aerolíneas aceptan un exceso de reserva en sus vuelos, i.e., venden más pasajes que el cupo del vuelo. En el momento del vuelo, normalmente han habido suficientes cancelaciones para que esto no sea un problema (todos los pasajeros consiguen abordar el vuelo). Pero, ocasionalmente, algunos pasajeros no encuentran cupo, y deben ser acomodados de otra forma (e.g., registrándolos en un vuelo posterior y reembolsándolos por la molestia). Este es un ejemplo de planificación optimista: a un pasajero que requiere un pasaje, se le vende aún si el cupo ya está completamente vendido, siempre que el sobrecupo sea inferior a un límite. Los problemas ocasionales se toleran pues vender el vuelo con sobrecupo permite incrementar el número de sillas realmente ocupadas en un vuelo y, en consecuencia, los problemas se reparan con facilidad.

Los ferrocarriles son muy cuidadosos para asegurar que nunca existan dos trenes viajando en direcciones contrarias sobre el mismo segmento de la vía. Un tren sólo puede entrar en un segmento específico de la vía si en ese momento no existe ningún otro tren en el mismo segmento. Los protocolos y los mecanismos de señales se han concebido para asegurar esto. Este es un ejemplo de planificación pesimista: un tren que requiera entrar en un segmento puede tener que esperar hasta que el segmento esté desocupado. A diferencia del caso del sobrecupo en la reserva de vuelos, en este caso no se toleran los accidentes porque son extremadamente costosos y, normalmente, irreparables en términos de pérdida de vidas.

Miremos cómo se aplican estos enfoques en las transacciones. Una transacción requiere un candado sobre una celda y hace este requerimiento al planificador. El planificador decide si el requerimiento se debe satisfacer y cuándo. Existen

tres respuestas posibles: satisfacer inmediatamente el requerimiento, rechazar el requerimiento (en consecuencia la transacción se aborta), o posponer la decisión. Un planificador optimista tiende a entregar el candado inmediatamente, aún si esto puede causar problemas más adelante (muertes súbitas y bloqueos vivientes<sup>10</sup>; ver abajo). Un planificador pesimista tiende a retardar la entrega del candado hasta que esté seguro que no habrá problemas. Dependiendo de cuán frecuentemente las transacciones trabajan sobre datos compartidos, puede ser más apropiado usar un planificador optimista o uno pesimista. Por ejemplo, si las transacciones se realizan mayormente sobre datos independientes, entonces un planificador optimista puede tener un desempeño mejor. Si las transacciones se realizan con frecuencia sobre datos compartidos, entonces un planificador pesimista puede tener un desempeño mejor. El algoritmo que presentamos abajo es optimista; algunas veces tiene que reparar los errores debidos a escogencias anteriores.

### *Adquisición y liberación de candados en dos fases*

La adquisición y liberación de candados en dos fases<sup>11</sup> es la técnica más popular para asegurar las transacciones. Esta técnica es utilizada por casi todos los sistemas comerciales de procesamiento de transacciones. Se puede probar que al realizar este aseguramiento en dos fases las transacciones son serializables. En este tipo de aseguramiento una transacción tiene dos fases: una fase de expansión en la cual la transacción adquiere los candados pero no los libera, y una fase de contracción, en la cual la transacción libera los candados pero no los adquiere. Una transacción no puede liberar un candado y adquirir otro enseguida. Esta restricción significa que una transacción podría mantener un candado más tiempo del que lo necesita. La experiencia muestra que este problema no es importante.

Un refinamiento popular del aseguramiento de las transacciones en dos fases, utilizado por muchos sistemas, se denomina aseguramiento estricto en dos fases. En este refinamiento, todos los candados se liberan simultáneamente al final de la transacción, después que ella se efectúa o aborta. Esto evita un problema llamado aborto en cascada. Considere el escenario siguiente. Suponga que se está utilizando el aseguramiento en dos fases con dos transacciones T1 y T2 que comparten la celda C. Primero, T1 asegura C (toma su candado) y cambia su contenido. Luego T1 libera el candado en su fase de contracción pero continúa activa. Finalmente, T2 asegura C, realiza un cálculo con C, y termina. ¿Qué pasa si T1, la cual está activa aún, aborta ahora? Si T1 aborta, entonces T2 tiene que abortar también, pues tiene un valor de C modificado por T1. T2 podría estar enlazado de manera similar a una transacción T3 y así sucesivamente. Si T1 aborta, entonces todas las otras transacciones tienen que abortar también, en cascada, aunque todas ellas hayan terminado. Si los candados sólo se liberan después de que las transacciones han

---

10. Nota del traductor: *livelock* en inglés.

11. Nota del traductor: del inglés *two-phase locking*

*Concurrencia con estado compartido*

terminado o abortado, entonces no sucede este problema.

### 8.5.2. Un administrador de transacciones sencillo

Diseñemos un administrador de transacciones sencillo, con control optimista de la concurrencia y con aseguramiento estricto en dos fases. Primero diseñamos el algoritmo utilizando un refinamiento paso a paso. Luego mostramos cómo implementar un administrador de transacciones basado en este algoritmo.

#### *Un algoritmo ingenuo*

Empezamos el diseño con la siguiente idea sencilla: Cuando una transacción requiera el candado de una celda no asegurada, ésta lo adquiere inmediatamente sin condiciones adicionales. Si la celda ya está asegurada, entonces se deja la transacción esperando hasta que la celda quede sin candado. Cuando una transacción se efectúa o aborta, entonces libera todos sus candados. Este algoritmo es optimista porque supone que conseguir el candado no traerá problemas más adelante. Si aparecen los problemas (¡ver el próximo párrafo!), entonces el algoritmo tiene que resolverlos.

#### *Muerte súbita*

Nuestro algoritmo ingenuo tiene un problema importante: sufre de posibles muertes súbitas. Considere dos transacciones T1 y T2, cada una usando las celdas C1 y C2. Digamos que la transacción T1 utiliza C1 y C2 en ese orden, y la transacción T2 utiliza C2 y C1, en orden inverso. Debido a la concurrencia, puede suceder que T1 tenga el candado de C1 y T2 el de C2. Cuando cada transacción trata de adquirir el otro candado que necesita, se quedan esperando. Ambas transacciones, por lo tanto, esperarán indefinidamente. Esta clase de situación, en la cual las entidades activas (transacciones) esperan un recurso (celdas) en un ciclo, y tal que ninguna entidad puede continuar, se denomina una muerte súbita.

¿Cómo podríamos asegurar que nuestro sistema no sufrirá nunca las consecuencias de una muerte súbita? Como ocurre en general con las enfermedades, existen dos enfoques básicos: prevención y cura. El objetivo de la prevención de la muerte súbita (también denominado evitar la muerte súbita) es prevenir que ésta suceda. Una transacción no puede colocar un candado a un objeto si puede llevar a una muerte súbita. El objetivo de la cura de la muerte súbita (también denominada detección y resolución de la muerte súbita) es detectar cuándo sucede una muerte súbita y realizar las acciones para revertir sus efectos.

Ambos conceptos están basados en un concepto denominado el grafo de espera.<sup>12</sup> Este es un grafo dirigido con nodos para las entidades activas (e.g., transacciones) y los recursos (e.g., celdas). Hay una arista desde cada entidad activa hacia el recurso

---

12. Nota del traductor: *wait-for graph*, en inglés.

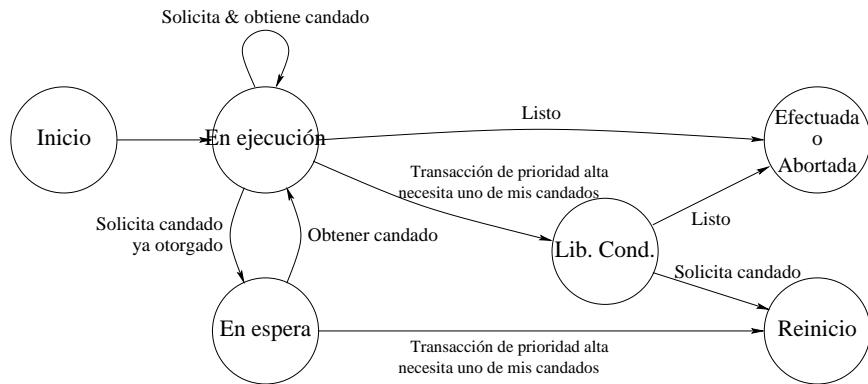
que está esperando tener, pero que todavía no tiene. Existe una arista desde cada recurso hacia la entidad activa que lo tiene (si hay alguna). Una muerte súbita corresponde a un ciclo en el grafo de espera. La prevención de la muerte súbita consiste en prohibir que se añada una arista que formaría un ciclo. La detección de la muerte súbita consiste en detectar la existencia de ciclos y, en consecuencia, eliminar una de sus aristas. El algoritmo que presentamos abajo realiza la prevención de la muerte súbita, evitando que una transacción consiga un candado que pudiera llevar a una muerte súbita.

### *El algoritmo correcto*

En el algoritmo ingenuo podemos evitar las muertes súbitas asignando más alta prioridad a las primeras transacciones que a las últimas. La idea básica es sencilla. Cuando una transacción trata de conseguir un candado, se compara su prioridad con la prioridad de la transacción que tiene el candado. Si la última tiene más baja prioridad, i.e., es una transacción más reciente, entonces ésta es reiniciada y la primera recibe el candado. Definimos un algoritmo basado en esta idea. Suponemos que las transacciones realizan operaciones sobre celdas y que cada celda viene con una cola de prioridad de transacciones en espera, i.e., las transacciones esperan en el orden de sus prioridades. Usamos marcas de tiempo para implementar las prioridades. Este es el algoritmo completo:

- A toda transacción nueva se le asigna una prioridad más baja que la de todas las transacciones activas.
- Cuando una transacción intenta obtener un candado de una celda, entonces se hace una de las acciones siguientes:
  - Si la celda está actualmente sin candado, entonces la transacción toma inmediatamente el candado y continúa.
  - Si la transacción ya tiene el candado de esa celda, entonces la transacción simplemente continúa.
  - Si la celda está asegurada por una transacción de más alta prioridad, entonces la transacción actual espera, i.e., se mete ella misma en la cola de la celda.
  - Si la celda está asegurada por una transacción de más baja prioridad, entonces se reinicia esta última y se entrega el candado a la transacción con más alta prioridad. Reiniciar una transacción consiste de dos acciones: primero abortar la transacción y segundo empezarla de nuevo con la misma prioridad.
- Cuando una transacción se efectúa, entonces libera todos sus candados y saca de la cola una transacción en espera por cada candado liberado (si existe alguna transacción en espera).
- Cuando una transacción aborta (porque lanza una excepción o realiza explícitamente una operación de aborto), entonces libera todas sus celdas aseguradas, restablece sus estados, y saca de la cola una transacción en espera por celda liberada (si existe alguna transacción en espera).

*Concurrencia con estado compartido*



**Figura 8.21:** Diagrama de estados de una encarnación de una transacción.

*Reiniciando en un punto bien definido*

Existe un pequeño problema con el algoritmo anterior: él detiene transacciones en ejecución en un punto arbitrario durante la misma. Esto puede causar problemas, pues puede llevar a inconsistencias en las estructuras de datos, en tiempo de ejecución, de la transacción. Esto puede llevar a complicaciones en la implementación del mismo administrador de transacciones.

Una solución sencilla consiste en terminar la transacción en un punto bien definido en su ejecución. Un punto bien definido es, e.g., el instante en que una transacción solicita al administrador de transacciones un candado. Refinamos el algoritmo anterior para reiniciar solamente en esos puntos. De nuevo, empezamos con una idea básica sencilla: en lugar de reiniciar una transacción de baja prioridad, la marcamos. Más adelante, cuando trate de obtener un candado, el administrador de transacciones nota que está marcada y la reinicia. Para implementar esta idea, extendemos el algoritmo así:

- Las transacciones pueden estar en uno de tres estados (las marcas):
  - enEjecución: este es el estado sin marca. La transacción se está ejecutando libremente y puede obtener los seguros como antes.
  - libCondicional: este es el estado marcado. La transacción aún se ejecuta libremente, pero la próxima vez que trate de obtener un candado, será reiniciada. Si no solicita más candados, finalmente se efectuará.
  - esperandoSobre(C): esto significa que la transacción está esperando por el candado sobre la celda C. Ella obtendrá el candado cuando esté disponible. Sin embargo, si una transacción de prioridad alta desea un candado que ésta tiene mientras está en espera, la transacción será reiniciada.

En la figura 8.21 se presenta el diagrama de estado de una encarnación de una transacción de acuerdo a este esquema. Por encarnación se entiende una parte de la

vida de una transacción, desde su primer inicio o un reinicio, hasta que se efectúa, aborta, o reinicia de nuevo.

- Cuando una transacción trata de obtener un candado, entonces se comprueba su estado antes de obtener los candados. Si está en el estado `libCondicional`, entonces se reinicia inmediatamente. Esto está bien, pues la transacción se encuentra en un punto bien definido.
- Cuando una transacción trata de obtener un candado y la celda está asegurada por una transacción de prioridad más baja, entonces se hace lo siguiente. Meter en la cola la transacción de prioridad alta y realizar una acción dependiendo del estado de la transacción de baja prioridad:
  - `enEjecución`: cambiar el estado a `libCondicional` y continuar.
  - `libCondicional`: no hacer nada.
  - `esperandoSobre(C)`: eliminar la transacción de la cola donde está esperando y reiniciarla inmediatamente. Esto está bien, pues la transacción se encuentra en un punto bien definido.
- Cuando se mete en la cola una transacción sobre una celda `C`, cambia su estado a `esperandoSobre(C)`. Cuando se saca de la cola una transacción, cambia su estado a `enEjecución`.

### 8.5.3. Transacciones sobre celdas

Definimos ahora una abstracción de datos para realizar transacciones sobre celdas utilizando el algoritmo de la sección anterior.<sup>13</sup> Definimos la abstracción así:

- `{NuevaTrans ?Trans ?NuevaCeldaT}` crea un nuevo contexto para transacciones y devuelve dos operaciones: `Trans` para crear transacciones y `NuevaCeldaT` para crear celdas nuevas.
- Una celda nueva se crea invocando `NuevaCeldaT` de la misma forma que con el procedimiento estándar `NewCell`:

```
{NuevaCeldaT x C}
```

Esto crea una celda nueva en el contexto para transacciones y la liga a `C`. La celda solamente se puede utilizar dentro de las transacciones de este contexto. El valor inicial de la celda es `x`.

- Una transacción nueva se crea invocando la función `Trans` así:

```
{Trans fun {$ T} <expr> end B}
```

La expresión secuencial `<expr>` puede interactuar con su ambiente solamente de

---

13. Se puede definir una abstracción de datos similar para transacciones sobre objetos, pero la implementación es un poco más complicada pues hay que tener en cuenta las clases y los métodos. Por lo tanto, por simplicidad, nos limitamos nosotros mismos a las celdas.

*Concurrencia con estado compartido*

las maneras siguientes: leyendo valores (incluyendo procedimientos y funciones) y realizando operaciones sobre celdas creadas con NuevaCeldaT. La invocación a Trans ejecuta  $\langle\text{expr}\rangle$  de forma transaccional y termina cuando  $\langle\text{expr}\rangle$  termina. Si  $\langle\text{expr}\rangle$  lanza una excepción, entonces la transacción abortará y lanzará la misma excepción. Si la transacción se efectúa, entonces el efecto es el mismo que el de una ejecución atómica de  $\langle\text{expr}\rangle$  y devuelve el mismo resultado. Si la transacción aborta, entonces es como si  $\langle\text{expr}\rangle$  no se hubiera ejecutado en absoluto (todos los cambios del estado se deshacen). B se liga a efectuada o aborto, respectivamente, dependiendo de si la transacción se efectuó o abortó.

- Hay cuatro operaciones que se pueden realizar dentro de  $\langle\text{expr}\rangle$ :
  - T.access, T.assign, y T.exchange tienen la misma semántica que las tres operaciones estándar de las celdas. Estas operaciones deben usar solamente celdas creadas con NuevaCeldaT.
  - T.aborto es un procedimiento sin argumentos que cuando se invoca hace que la transacción aborte inmediatamente.
- Solamente hay dos maneras en que una transacción puede abortar: o lanza una excepción o invoca a T.aborto. En todos los otros casos, la transacción finalmente se efectuará.

*Un ejemplo*

Primero creamos un nuevo ambiente de transacciones:

```
declare Trans NuevaCeldaT in
{NuevaTrans Trans NuevaCeldaT}
```

Ahora definimos dos celdas en este ambiente:

```
C1={NuevaCeldaT 0}
C2={NuevaCeldaT 0}
```

Ahora aumentamos C1 y disminuimos C2 en la misma transacción:

```
{Trans proc {$ T _}
{T.assign C1 {T.access C1}+1}
{T.assign C2 {T.access C2}-1}
end _ _}
```

(Utilizamos sintaxis de procedimiento porque no estamos interesados en la salida.) Podemos repetir esta transacción muchas veces en diferentes hilos. Como las transacciones son atómicas, estamos seguros que  $@C1 + @C2 = 0$  siempre será cierta. Ese es un invariante de nuestro sistema. Esto no pasaría si el aumento y la disminución se ejecutaran por fuera de una transacción. Para leer los contenidos de C1 y C2, tenemos que utilizar otra transacción:

```
{Browse {Trans fun {$ T} {T.access C1}#{T.access C2} end _}}
```

*Otro ejemplo*

El ejemplo anterior no muestra las verdaderas ventajas de las transacciones. Lo mismo se hubiera podido lograr con candados. Nuestra abstracción de transacción tiene dos ventajas con respecto a los candados: abortar hace que se restablezcan los estados originales de las celdas y los candados pueden ser solicitados en cualquier orden sin riesgo de muerte súbita. Presentamos ahora un ejemplo más sofisticado que explota estas dos ventajas. Creamos una tupla con 100 celdas y realizamos cálculos transaccionales con ella. Empezamos por crear e inicializar la tupla:

```
BD={MakeTuple bd 100}
for I in 1..100 do BD.I={NuevaCeldaT I} end
```

(Utilizamos una tupla de celdas en lugar de un arreglo porque nuestro sistema de transacciones solamente maneja celdas.) Ahora, definimos dos transacciones, *Mezclar* y *Sumar*. *Sumar* calcula la suma de todos los contenidos de las celdas. *Mezclar* “revuelve” los contenidos de las celdas en forma aleatoria pero la suma total sigue siendo igual. Esta es la definición de *Mezclar*:

```
fun {Rand} {OS.rand} mod 100 + 1 end
proc {Mezclar} {Trans
 proc {$ T _}
 I={Rand} J={Rand} K={Rand}
 A={T.access BD.I} B={T.access BD.J} C={T.access BD.K}
 in
 {T.assign BD.I A+B-C}
 {T.assign BD.J A-B+C}
 if I==J orelse I==K orelse J==K then {T.aborto} end
 {T.assign BD.K ~A+B+C}
 end _ _
 end
}
```

El generador de números aleatorios *Rand* se implementa con el módulo *os*. La función que revuelve los contenidos de las celdas, reemplaza los contenidos *a*, *b*, *c* de tres celdas tomadas al azar, por los nuevos contenidos *a* + *b* - *c*, *a* - *b* + *c*, y -*a* + *b* + *c*. Para garantizar que se toman tres celdas diferentes, *Mezclar* aborta si dos de ellas son la misma. El aborto se puede realizar en cualquier punto dentro de la transacción. Esta es la definición de *Sumar*:

```
S={NuevaCeldaT 0}
fun {Sumar}
 {Trans
 fun {$ T} {T.assign S 0}
 for I in 1..100 do
 {T.assign S {T.access S}+{T.access BD.I}} end
 {T.access S}
 end __
 end
}
```

*Sumar* utiliza la celda *S* para guardar la suma. Note que *Sumar* es una transacción grande, pues simultáneamente asegura todas las celdas de la tupla. Ahora podemos hacer algunos cálculos:

Concurrencia con estado compartido

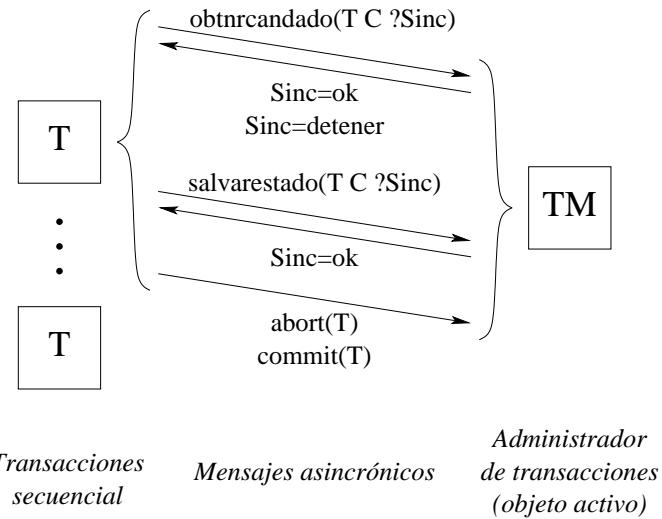


Figura 8.22: Arquitectura del sistema de transacciones.

```
{Browse {Sumar}} % Muestra 5050
for I in 1..1000 do thread {Mezclar} end end
{Browse {Sumar}} % Aún muestra 5050
```

5050 es la suma de los enteros entre 1 y 100. Se puede comprobar que los valores de cada celda están verdaderamente bien mezclados:

```
{Browse {Trans fun {$ T} {T.access BD.1}#{T.access BD.2} end _}}
```

Inicialmente, esto despliega 1#2, pero en adelante muestra valores muy diferentes.

#### 8.5.4. Implementando transacciones sobre celdas

Ahora, mostramos cómo construir un sistema de transacciones que implemente nuestro algoritmo optimista de aseguramiento en dos fases. La implementación consta de un administrador de transacciones y de un conjunto de transacciones en ejecución. (Los administradores de transacciones vienen en muchas variedades y algunas veces se denominan monitores de procesamiento de transacciones [60].) El administrador de transacciones y las transacciones en ejecución, se ejecutan cada uno en su propio hilo. Esto permite terminar una transacción en ejecución sin afectar al administrador de transacciones. Un hilo en ejecución envía cuatro clases de mensajes al administrador de transacciones: obtener un candado (`obtnrcandado`), salvar el estado de una celda (`salvarestado`), terminar (`efectuada`), y abortar (`aborts`). En la figura 8.22 se muestra la arquitectura.

El administrador de transacciones siempre está activo y acepta órdenes de los hilos que están ejecutando transacciones. Cuando se reinicia una transacción, ella empieza en un hilo nuevo, aunque guarda la misma marca de tiempo. Implemen-

```

class ClaseAT
 attr marca_de_tiempo at
 meth inic(AT) marca_de_tiempo:=0 at:=AT end

 meth LibereTodo(T Restablecer)
 for est_celda(celda:C estado:S) in {Dictionary.items
T.estadoCeldas} do
 (C.prprio):=listo
 if Restablecer then (C.estado):=S end
 if {Not {C.cola.esVaciaCola}} then
 Sinc2#T2={C.cola.atender} in
 (T2.estado):=enEjecución
 (C.prprio):=T2 Sinc2=ok
 end
 end
 end

 meth Trans(P ?R TS) /* Ver figura siguiente */ end
 meth obtncandado(T C ?Sinc) /* Ver figura siguiente */ end

 meth nuevatrans(P ?R)
 marca_de_tiempo:=@marca_de_tiempo+1
 {self Trans(P R @marca_de_tiempo)}
 end
 meth salvarestado(T C ?Sinc)
 if {Not {Dictionary.member T.estadoCeldas C.nombre}} then
 (T.estadoCeldas).(C.nombre):=est_celda(celda:C
estado:@(C.estado))
 end Sinc=ok
 end
 meth efectuada(T) {self LibereTodo(T false)} end
 meth aborto(T) {self LibereTodo(T true)} end
end

proc {NuevaTrans ?Trans ?NuevaCeldaT}
AT={CrearObjActivo ClaseAT inic(AT)} in
 fun {Trans P ?B} R in
 {AT nuevatrans(P R)}
 case R of aborto then B=aborts listo
 [] aborto(Exc) then B=aborts raise Exc end
 [] efectuada(Res) then B=efectuada Res end
 end
 fun {NuevaCeldaT X}
 celda(nombre:{NewName} prprio:{NewCell listo}
 cola:{NuevaColaDePrioridad} estado:{NewCell X})
 end
end

```

**Figura 8.23:** Implementación del sistema de transacciones (parte 1).

Concurrencia con estado compartido

```

meth Trans(P ?R MT)
 Detener={NewName}
 T=trans(marca:MT estadoCeldas:{NewDictionary} cuerpo:P
 estado:{NewCell enEjecución} resultado:R)
 proc {ExcT C X Y} S1 S2 in
 {@at obtncandado(T C S1)}
 if S1==detener then raise Detener end end
 {@at salvarestado(T C S2)} {Wait S2}
 {Exchange C.estado X Y}
 end
 proc {AccT C ?X} {ExcT C X X} end
 proc {AssT C X} {ExcT C _ X} end
 proc {AboT} {@at aborto(T)} R=aborto raise Detener end end
in
 thread try Res={T.cuerpo t(access:AccT assign:AssT
 exchange:ExcT aborto:AboT)}
 in {@at efectuada(T)} R=efectuada(Res)
 catch E then
 if E\=Detener then {@at aborto(T)} R=aborto(E) end
 end end
end

meth obtncandado(T C ?Sinc)
 if @(T.estado)==libCondicional then
 {@self LibereTodo(T true)}
 {@self Trans(T.cuerpo T.resultado T.marca)} Sinc=detener
 elseif @(C.prprio)==listo then
 (C.prprio):=T Sinc=ok
 elseif T.marca==@(C.prprio).marca then
 Sinc=ok
 else /* T.marca\=@(C.prprio).marca */ T2=@(C.prprio) in
 {C.cola.insCola Sinc#T T.marca}
 (T.estado):=esperandoSobre(C)
 if T.marca<T2.marca then
 case @(T2.estado) of esperandoSobre(C2) then
 Sinc2#=C2.cola.elimDeCola T2.marca} in
 {@self LibereTodo(T2 true)}
 {@self Trans(T2.cuerpo T2.resultado T2.marca)}
 Sinc2=detener
 [] enEjecución then
 (T2.estado):=libCondicional
 [] libCondicional then skip end
 end
 end
end

```

Figura 8.24: Implementación del sistema de transacciones (parte 2).

tamos el administrador de transacciones como un objeto activo, utilizando la función `CrearObjActivo` de la sección 7.8. El objeto activo tiene dos métodos internos, `LibereTodo` y `Trans`, y cinco métodos externos, `nuevatrans`, `obtnrcandado`, `salvarestado`, `efectuada`, y `aborts`. En las figuras 8.23 y 8.24 se muestra la implementación del sistema de transacciones. En conjunto con `CrearObjActivo` y la cola de prioridad, esta es una implementación completa y que funciona. Cada transacción activa está representada por un registro con cinco campos:

- **marca**: Esta es la marca de tiempo de la transacción, un entero único que identifica la transacción y su prioridad. Este número se incrementa para transacciones consecutivas. Por lo tanto, una prioridad alta está asociada a una marca de tiempo pequeña.
- **estadoCeldas**: Este es un diccionario indexado por el nombre de las celdas (ver abajo), y que contiene entradas de la forma `est_celda(celda:C estado:S)`, donde `C` es un registro para la celda (representado como se describe abajo) y `S` es el estado original de la celda.
- **cuerpo**: Esta es la función `fun { $ T } <expr> end` que representa el cuerpo de la transacción.
- **estado**: Esta es una celda que contiene uno de los valores siguientes: `enEjecución`, `libCondicional`, y `esperandoSobre(C)`. Si contiene `libCondicional`, significa que la transacción será reiniciada la próxima vez que intente obtener un candado. Si contiene `esperandoSobre(C)`, significa que la transacción será reiniciada inmediatamente si una transacción de prioridad más alta necesita a `C`.
- **resultado**: Esta es una variable de flujo de datos que será ligada a `efectuada(Res)`, `aborts(Exc)`, o `aborts` cuando la transacción se complete.

Cada celda está representada por un registro con cuatro campos:

- **nombre**: Este es un valor de tipo nombre que es el identificador único de la celda.
- **prptrio**: Este es el valor `listo`, si ninguna transacción ha obtenido el candado de la celda, o el registro de la transacción que tiene actualmente el candado de la celda.
- **cola**: Esta es una cola de prioridad que contiene parejas de la forma `Sinc#T`, donde `T` es un registro de transacción y `Sinc` es la variable de sincronización sobre la cual la transacción está bloqueada actualmente. La prioridad es la marca de tiempo de la transacción. `Sinc` siempre será finalmente ligada a `ok` o `detener` por el administrador de transacciones.
- **estado**: Esta es una celda que almacena el contenido transaccional de la celda.

Cuando una transacción `T` realiza una operación de intercambio sobre la celda `C`, se ejecuta el procedimiento `ExcT` definido en `Trans`. Este procedimiento envía primero el mensaje `obtnrcandado(T C Sinc1)` al administrador de transacciones para solicitar un candado sobre la celda. El administrador de transacciones responde con `Sinc1=ok` si la transacción obtiene exitosamente el candado y `Sinc1=detener`

*Concurrencia con estado compartido*

```
fun {NuevaColaDePrioridad}
 Q={NewCell nil}
 proc {InsCola X Prio}
 fun {InsertLoop L}
 case L of pair(Y P)|L2 then
 if Prio<P then pair(X Prio)|L
 else pair(Y P)|{InsertLoop L2} end
 [] nil then [pair(X Prio)] end
 end
 in Q:={InsertLoop @Q} end

 fun {Atender}
 pair(Y _)|L2=@Q
 in
 Q:=L2 Y
 end

 fun {ElimDeCola Prio}
 fun {CicloEliminación L}
 case L of pair(Y P)|L2 then
 if P==Prio then X=Y L2
 else pair(Y P)|{CicloEliminación L2} end
 [] nil then nil end
 end X
 in Q:={CicloEliminación @Q} X end

 fun {EsVacíaCola} @Q==nil end
 in
 cola(insCola:InsCola atender:Atender
 elimDeCola:ElimDeCola esVacíaCola:EsVacíaCola)
 end
```

**Figura 8.25:** Cola de prioridad.

si el hilo actual debe ser terminado. En el último caso, obtncrcandado asegura que la transacción se reinicie. Si la transacción obtiene el candado, entonces Exct invoca a salvarestado(T C Sinc2) para guardar el estado original de la celda.

### *Cola de prioridad*

El administrador de transacciones utiliza colas de prioridad para asegurar que las transacciones de más alta prioridad tengan la primera oportunidad de obtener el candado de las celdas. Una cola de prioridad es una cola cuyas entradas siempre están ordenadas de acuerdo a alguna prioridad. En nuestra cola, las prioridades son enteros y el valor más bajo tiene la prioridad más alta. Definimos la abstracción de datos cola de prioridad así:

- Q={NuevaColaDePrioridad} crea una cola de prioridad vacía.

- $\{Q.\text{insCola } x \ p\}$  inserta  $x$  con prioridad  $p$ , donde  $p$  es un entero.
- $x=\{Q.\text{atender}\}$  devuelve la entrada con el menor valor entero y la elimina de la cola.
- $x=\{Q.\text{elimDeCola } p\}$  devuelve una entrada con prioridad  $p$  y la elimina de la cola.
- $B=\{Q.\text{esVaciaCola}\}$  devuelve **true** o **false** dependiendo de si la cola está vacía o no.

En la figura 8.25 se muestra una implementación sencilla de la cola de prioridad. La cola de prioridad se representa internamente como una celda que contiene una lista de parejas  $\text{pair}(x \ p)$ , las cuales están ordenadas crecientemente de acuerdo a  $p$ . La operación `atender` se ejecuta con una complejidad en tiempo de  $O(1)$ . Las operaciones `insCola` y `elimDeCola` se ejecutan con una complejidad en tiempo de  $O(s)$  donde  $s$  es el tamaño de la cola. Se pueden lograr implementaciones más sofisticadas con mejores órdenes de complejidad en tiempo.

#### 8.5.5. Más sobre transacciones

Apenas hemos arañado la superficie del procesamiento de transacciones. Finalizamos esta sección mencionando algunas de las extensiones más útiles [60]:

- Duración. No hemos mostrado cómo hacer persistente un cambio de estado. Esto se hace colocando los cambios de estado sobre almacenamientos estables, tal como un disco. Las técnicas para hacer esto se diseñaron cuidadosamente para mantener la atomicidad, no importa en qué instante del tiempo suceda una caída del sistema.
- Transacciones anidadas. Frecuentemente sucede que tenemos una transacción de larga vida que contiene una serie de transacciones más pequeñas. Por ejemplo, una transacción bancaria compleja podría consistir de una larga serie de actualizaciones de muchas cuentas. Cada una de las actualizaciones es una transacción. La serie misma debería ser una transacción: si algo va mal en medio de ella, se cancela. Existe una fuerte relación entre transacciones anidadas, encapsulación, y modularidad.
- Transacciones distribuidas. Con frecuencia sucede que una base de datos está repartida entre diversos sitios físicos, ya sea por razones de desempeño o por razones organizacionales. Nos gustaría en este caso poder realizar transacciones sobre la base de datos.

---

## 8.6. El lenguaje Java (parte concurrente)

En la introducción de la sección 7.7 solamente hablamos sobre la parte secuencial de Java. Ahora hablaremos de la parte concurrente. La programación concurrente en Java se basa en dos conceptos: hilos y monitores. Java se diseñó para hacer concurrencia con estado compartido. Los hilos son muy pesados para soportar

eficientemente un enfoque de objetos activos. Los monitores tienen la semántica presentada en la sección 8.4. Los monitores son entidades livianas asociadas a objetos individuales.

Cada programa comienza con un hilo, el hilo que ejecuta `main`. Los hilos nuevos se pueden crear de dos maneras: instanciando una subclase de la clase `Thread` o implementando la interfaz `Runnable`. Por defecto, el programa termina cuando todos sus hilos terminan. Como los hilos tienden a ser pesados en las actuales implementaciones de Java, al programador se le sugiere no crear muchos. Utilizar la clase `Thread` ofrece más control, pero puede ser exagerado para algunas aplicaciones. Utilizar la interfaz `Runnable` es más liviano. Ambas técnicas suponen la existencia de un método `run`:

```
public void run();
```

que define el cuerpo del hilo. La interfaz `Runnable` consiste únicamente de este sólo método.

Los hilos interactúan por medio de objetos compartidos. Para controlar la interacción, cualquier objeto Java puede ser un monitor, tal como se definió en la sección 8.4. Los métodos se pueden ejecutar dentro del candado del monitor con la palabra reservada `synchronized`. Los métodos que no tienen esta palabra reservada se denominan no sincronizados. Ellos se ejecutan por fuera del candado del monitor, pero pueden ver aún los atributos del objeto. Esta capacidad ha sido fuertemente criticada porque el compilador no puede entonces garantizar que los atributos del objeto sean accedidos secuencialmente [24]. Los métodos no sincronizados pueden ser más eficientes, pero deberían ser usados muy raramente.

Presentamos dos ejemplos. El primer ejemplo utiliza métodos sincronizados sólo para colocar candados. El segundo ejemplo utiliza todas las operaciones del monitor. Para lecturas adicionales, recomendamos [103].

### 8.6.1. Candados

La manera más sencilla de programar concurrentemente en Java es con múltiples hilos que acceden objetos compartidos. Como ejemplo extendemos la clase `Punto`:

```
class Punto {
 double x, y;
 Punto(double x1, y1) { x=x1; y=y1; }
 public double obtX() { return x; }
 public double obtY() { return y; }
 public synchronized void origen() { x=0.0; y=0.0; }
 public synchronized void sumar(Punto p)
 { x+=p.obtX(); y+=p.obtY(); }
 public synchronized void escalar(double s) { x*=s; y*=s; }
 public void dibujar(Graphics g) {
 double lx, ly;
 synchronized (this) { lx=x; ly=y; }
 g.dibujarPunto(lx, ly);
 }
}
```

Cada instancia de `Punto` tiene su propio candado. Gracias a la palabra reservada `synchronized`, los métodos `origen`, `sumar`, y `escalar` se ejecutan todos dentro del candado. Solamente el método `dibujar` está parcialmente sincronizado. Esto se debe a que invoca un método externo (no definido en el ejemplo). Si se colocara el método dentro del candado del objeto, entonces se incrementaría la probabilidad de que se presente una muerte súbita en el programa. En su lugar, `g` debería tener su propio candado.

### 8.6.2. Monitores

Los monitores son una extensión de los candados que ofrecen mayor control sobre la forma como los hilos entran y salen. Los monitores se pueden utilizar para realizar tipos más sofisticados de cooperación entre hilos que acceden un objeto compartido. En la sección 8.4.2 se muestra cómo escribir una memoria intermedia limitada utilizando monitores. La solución presentada allí se puede trasladar fácilmente a Java; el resultado se muestra en la figura 8.26. Allí se define una memoria intermedia limitada de enteros. Se utiliza un arreglo de enteros, `mem`, cuyo espacio se reserva en el momento en que la memoria intermedia se inicializa. El signo de porcentaje `%` denota la operación módulo, i.e., el residuo después de la división entera.

---

## 8.7. Ejercicios

1. *Número de intercalaciones.* Generalice el argumento utilizado en la introducción del capítulo para calcular el número de intercalaciones posibles de  $n$  hilos, cada uno realizando  $k$  operaciones. Utilizando la fórmula de Stirling para la función factorial,  $n! \approx \sqrt{2\pi}n^{n+1/2}e^{-n}$ , calcule una aproximación, en forma cerrada, de esta función.

*Concurrencia con estado compartido*

```
class MemIntermedia
 int[] mem;
 int primero, último, n, i;

 public void inic(int size) {
 mem=new int[size];
 n=size; i=0; primero=0; último=0;
 }

 public synchronized void colocar(int x) {
 while (i<n) wait();
 mem[último]=x;
 último=(último+1)%n;
 i=i+1;
 notifyAll();
 }

 public synchronized int obtener() {
 int x;
 while (i==0) wait();
 x=mem[primero];
 primero=(primero+1)%n;
 i=i-1;
 notifyAll();
 return x;
 }
}
```

**Figura 8.26:** Memoria intermedia limitada (versión Java).

2. *Contador concurrente.* Implementemos un contador concurrente en la forma más sencilla posible. El contador tiene una operación para incrementarlo. Nos gustaría que esta operación se pueda invocar desde cualquier número de hilos. Considere la implementación siguiente que utiliza una celda y un Exchange:

```
local X in {Exchange C X X+1} end
```

Esta solución no funciona.

- Explique por qué el programa anterior no funciona y proponga una solución sencilla.
- ¿Su solución funcionaría aún en un lenguaje que no tenga variables de flujo de datos? Explique por qué sí o por qué no.
- Dé una solución (tal vez sea la misma del punto anterior) que funcione en un lenguaje sin variables de flujo de datos.

3. *Concurrencia maximal y eficiencia.* Entre los modelos concurrente con estado compartido y maximalmente concurrente, existe un modelo interesante denominado

el modelo concurrente basado en trabajo. Este modelo es idéntico al modelo concurrente con estado compartido, salvo que cuando una operación se bloquearía, se crea un nuevo hilo con esa única operación (que se denomina un trabajo) y el hilo original continúa su ejecución.<sup>14</sup> En términos prácticos, el modelo basado en trabajo tiene toda la concurrencia del modelo maximalmente concurrente, y además puede ser implementado fácilmente. En este ejercicio, investigue el modelo basado en trabajo. ¿Le parece una buena elección para un lenguaje de programación concurrente? Explique por qué sí o por qué no.

4. *Simulando redes lentas.* En la sección 8.2.2 se define una función `RedLenta2` que crea una versión “lenta” de un objeto. Pero esta definición impone un restricción fuerte de orden. Cada objeto lento define un orden global de sus invocaciones y garantiza que los objetos originales se invocan en ese orden. Con frecuencia esta restricción es demasiado fuerte. Una versión más refinada sólo impondría ese orden entre las invocaciones al objeto dentro del mismo hilo. Entre hilos diferentes, no existe razón para imponer ese orden. Defina una función `RedLenta3` que cree objetos lentos con esta propiedad.

5. *La abstracción MVar.* Una `MVar` es una caja que puede estar llena o vacía. La caja viene con dos procedimientos, `Colocar` y `Obtener`. `{Colocar x}` coloca `x` en la caja si está vacía, llenándola. Si la caja está llena, `colocar` espera hasta que esté vacía. `{Obtener x}` liga `x` con el contenido de la caja y la deja vacía. Si la caja está vacía, `obtener` espera hasta que esté llena. En este ejercicio, implemente la abstracción `MVar`. Utilice el enfoque de concurrencia que sea más natural.

6. *Comunicando procesos secuenciales (CSP).*<sup>15</sup> El lenguaje CSP consiste de hilos independientes (denominados “procesos” en la terminología de CSP) que se comunican a través de canales sincrónicos [73, 152]. Los canales tienen dos operaciones, enviar y recibir, con semántica rendezvous. Es decir, un envío se bloquea hasta que se presente una recepción y viceversa. Cuando el envío y la recepción se presentan simultáneamente, entonces ambas operaciones se terminan atómicamente, transfiriéndose la información del envío a la recepción. El lenguaje Ada también utiliza semántica rendezvous. Además, existe una operación de recepción no-determinística, la cual puede escuchar varios canales simultáneamente. Tan pronto llegue un mensaje en alguno de los canales, la recepción no-determinística se termina. En este ejercicio, implemente estas operaciones de CSP a través de la abstracción de control siguiente:

- `C={Canal.nuevo}` crea un canal nuevo `C`.
- `{Canal.enviar C M}` envía el mensaje `M` por el canal `C`.
- `{Canal.mrecibir [C1#S1 C2#S2 ... Cn#Sn]}` escucha de forma no determinística por los canales `C1, C2, ..., y Cn`. Si es un procedimiento de un argumento `proc { $ M } <decl> end` que se ejecuta cuando el mensaje `M` se reci-

---

14. Una versión anterior del lenguaje OZ utilizó el modelo basado en trabajo [161].

15. De sus siglas en inglés: Communicating Sequential Processes.

be en el canal `ci`.

Ahora extienda la operación `Canal.mrecibir` con guardas:

- `{Canal.mrecibir [C1#B1#S1 C2#B2#S2 ... Cn#Bn#Sn]}`, donde `Bi` es una función booleana de un argumento `fun {$ M} <expr> end` que debe devolver `true` para que se pueda recibir un mensaje por el canal `ci`.

7. *Comparando Linda con Erlang.* Linda cuenta con una operación de lectura que puede recuperar tuplas, de manera selectiva, de acuerdo a un patrón (ver sección 8.3.2). Erlang cuenta con una operación `receive` que puede recibir mensajes, de manera selectiva, de acuerdo a un patrón (ver sección 5.7.3). En este ejercicio, compare y contraste estas dos operaciones y las abstracciones de las que hacen parte. ¿Qué tienen en común y en qué difieren? ¿Para qué tipos de aplicaciones es más apropiada cada una?

8. *Detección de terminación con monitores.* Este ejercicio tiene que ver con detectar en qué momento han terminado todos los hilos de un grupo de hilos. En la sección 4.4.3 se presenta un algoritmo que funciona para un espacio delgado de hilos, donde los hilos no pueden crear hilos nuevos. En la sección 5.6.3 se presenta un algoritmo que funciona para un espacio jerárquico de hilos, donde los hilos pueden crear hilos nuevo sin límites en el nivel de anidamiento. El segundo algoritmo utiliza un puerto para recolectar la información de terminación. En este ejercicio, escriba un algoritmo que funcione para un espacio jerárquico de hilos, como el segundo algoritmo, pero que utilice un monitor en lugar de un puerto.

9. *Monitores y condiciones.* En la sección 8.4.5 se presenta una semántica alternativa para los monitores en la cual hay varios conjuntos de espera, los cuales se denominan condiciones. El propósito de este ejercicio es estudiar esta alternativa y compararla con el enfoque principal presentado en el texto.

- Reimplemente el ejemplo de la memoria intermedia limitada de la figura 8.17 utilizando monitores con condiciones.
- Modifique la implementación de monitor de la figura 8.20 para implementar monitores con condiciones. Permita que se pueda crear más de una condición para un monitor.

10. *Disolviendo transacciones grandes.* El segundo ejemplo de la sección 8.5.3 define la transacción `Sumar` que obtiene el candado de todas las celdas de la tupla mientras calcula su suma. Mientras `Sumar` esté activa, ninguna otra transacción puede continuar. En este ejercicio, reescriba `Sumar` como una serie de transacciones pequeñas. Cada transacción pequeña sólo debería obtener el candado de unas pocas celdas. Defina una representación para una suma parcial, de manera que una transacción pequeña pueda ver lo que se ha hecho hasta ahora y determinar cómo continuar. Compruebe su trabajo mostrando que usted puede realizar transacciones mientras se realiza un cálculo de la suma.

11. *Ocultando candados.* Por simplicidad, el administrador de transacciones de la sección 8.5.4 tiene algunas ineficiencias sin importancia. Por ejemplo, los mensajes `obtnrcandado` y `salvarestado` se envían cada que una transacción utiliza una

celda. Es claro que esos mensajes sólo se necesitan la primera vez. En este ejercicio, optimice los protocolos `obtnrcandado` y `salvarestado` de manera que usen el menor número posible de mensajes.

12. *Candados de lectura y escritura.* El administrador de transacciones de la sección 8.5 asegura una celda desde su primera utilización. Si las transacciones T1 y T2 desean leer, las dos, el contenido de una misma celda, entonces no pueden asegurar la celda simultáneamente. Podemos relajar este comportamiento introduciendo dos clases de candados, candados de lectura y candados de escritura. Una transacción que tiene un candado de lectura sólo tiene permitido leer el contenido de la celda, no modificarlo. Una transacción que tiene un candado de escritura puede hacer todas las operaciones sobre celdas. Una celda puede estar asegurada por exactamente un candado de escritura o por cualquier número de candados de lectura. En este ejercicio, extienda el administrador de transacciones para utilizar candados de lectura y escritura.

13. *Transacciones concurrentes.* El administrador de transacciones de la sección 8.5 maneja correctamente cualquier número de transacciones que se ejecuten concurrentemente, pero cada transacción individual debe ser secuencial. En este ejercicio, extienda el administrador de transacciones de manera que las transacciones individuales puedan ejecutarse, ellas mismas, concurrentemente. *Sugerencia:* agregue el algoritmo de detección de terminación de la sección 5.6.3.

14. *Combinando monitores y transacciones.* Diseñe e implemente una abstracción de concurrencia que combine las capacidades de los monitores y las transacciones. Es decir, una abstracción que tenga la capacidad de esperar y notificar, y también la capacidad de abortar sin modificar ningún estado. ¿Es útil esta abstracción?

15. ( proyecto de investigación) *Modelo de computación transaccional.* Extienda el modelo concurrente con estado compartido de este capítulo para permitir transacciones, tal como fue sugerido en la sección 8.5. Su extensión debe satisfacer las propiedades siguientes:

- Debe tener una semántica formal sencilla.
- Debe ser eficiente, i.e., sólo debe causar un sobrecosto al sistema en el momento en que las transacciones se usen.
- Debe conservar las propiedades buenas del modelo, e.g., composicionalidad.

Esto permitirá a los programas utilizar las transacciones sin codificaciones costosas y pesadas. Implemente un lenguaje de programación que utilice su extensión y evalúela con programas concurrentes realistas.



---

## II APÉNDICES



---

## A Ambiente de Desarrollo del Sistema Mozart

¡Guárdate de los idus de marzo!

– Adivino a Julio César, William Shakespeare (1564–1616)

El sistema Mozart cuenta con un IDE<sup>1</sup> (ambiente de desarrollo interactivo). Presentamos una breve visión general a modo de iniciación. Para información adicional recomendamos leer la documentación del sistema.

---

### A.1. Interfaz interactiva

El sistema Mozart cuenta con una interfaz interactiva basada en el editor de texto Emacs. La interfaz interactiva también se denomina la OPI, que significa interfaz de programación de Oz<sup>2</sup>. La OPI está dividida en diversos espacios de memoria denominados *buffers*: scratch pad, Oz emulator, Oz compiler, y un *buffer* por cada archivo abierto. Esta interfaz permite acceder a varias herramientas: el compilador incremental (el cual puede compilar cualquier fragmento de programa legal), el *Browser* (para visualizar el almacén de asignación única), el panel (utilización de recursos), el panel del compilador (ajuste del compilador y del ambiente), panel de distribución (subsistema de distribución incluyendo tráfico de mensajes), y el explorador (interfaz gráfica para la resolución de problemas de restricciones). Esta herramientas también se pueden manipular desde el interior de los programas, e.g., el módulo *Compiler* puede compilar cadenas de caracteres desde el interior de los programas.

#### A.1.1. Órdenes de la interfaz

Se puede acceder a todas las órdenes importantes de la OPI a través de los menús en la parte superior de la ventana. La mayoría de estos comandos tienen una combinación de teclas equivalente. Las órdenes más importantes se listan en la tabla A.1. La notación “CTRL-x” significa presionar la tecla x una vez, mientras mantiene presionada la tecla CTRL. La orden CTRL-g es especialmente útil si usted

---

1. Nota del traductor: del inglés *interactive development environment*.

2. Nota del traductor: del inglés *Oz programming interface*.

| Orden         | Efecto                                                              |
|---------------|---------------------------------------------------------------------|
| CTRL-x CTRL-f | Lee un archivo en un nuevo <i>buffer</i> de edición                 |
| CTRL-x CTRL-s | Salva el <i>buffer</i> actual en su archivo                         |
| CTRL-x i      | Inserta un archivo dentro del <i>buffer</i> actual                  |
| CTRL-. CTRL-l | Alimenta Mozart con la línea actual                                 |
| CTRL-. CTRL-r | Alimenta Mozart con la región seleccionada                          |
| CTRL-. CTRL-p | Alimenta Mozart con el párrafo actual                               |
| CTRL-. CTRL-b | Alimenta Mozart con el ém buffer actual                             |
| CTRL-. h      | Detiene el sistema en tiempo de ejecución (pero conserva el editor) |
| CTRL-x CTRL-c | Detiene el sistema por completo                                     |
| CTRL-. e      | Muestra la ventana del emulador                                     |
| CTRL-. c      | Muestra la ventana del compilador                                   |
| CTRL-x 1      | LLena toda la ventana con el <i>buffer</i> actual                   |
| CTRL-g        | Cancela el comando actual                                           |

Tabla A.1: Algunas órdenes de la interfaz de programación de Oz.

está perdido. Alimentar<sup>3</sup> Mozart con un texto significa compilarlo y ejecutarlo. Una región es una parte contigua de un *buffer*, que se puede seleccionar arrastrando el cursor sobre la región mientras se mantiene presionado el botón izquierdo del ratón. Un párrafo es un conjunto de líneas no vacías de texto delimitadas por líneas vacías, o por el principio o el final de un *buffer*.

La ventana del emulador muestra los mensajes enviados por el emulador. Allí se presentan la salida de Show y los mensajes en tiempo de ejecución, e.g., excepciones no capturadas. La ventana del compilador muestra los mensajes enviados por el compilador. Allí se dice si el código fuente con que se alimentó al sistema se acepta, y presenta los mensajes de error en tiempo de compilación en caso de que no se acepte.

### A.1.2. Utilizando los functors interactivamente

Los functors son especificaciones de componentes de *software* que ayudan a construir programas bien estructurados. Un functor puede ser instanciado, creando así un módulo. Un módulo es una entidad en tiempo de ejecución, que agrupa a otras entidades en tiempo de ejecución. Los módulos pueden contener registros, procedimientos, objetos, clases, hilos en ejecución, y cualquier otra entidad que pueda existir en tiempo de ejecución.

Los functors son unidades de compilación, i.e., su código fuente puede ser colocado en un archivo y compilado en una unidad. Los functors también se pueden utilizar

---

3. Nota del traductor: *Feed* en inglés.

en la interfaz interactiva. Esto sigue el principio de Mozart que dice que todo se puede hacer interactivamente.

- Un functor compilado se puede cargar y enlazar interactivamente. Por ejemplo, suponga que el módulo `Set`, el cual se puede conseguir en el sitio Web del libro, se compila en el archivo `Set.ozf`. Este módulo puede ser cargado y enlazado interactivamente con el código siguiente:

```
declare
[Set]={Module.link ["Set.ozf"]}
```

Esto crea y enlaza el módulo `Set`. La función `Module.link` toma una lista de nombres de archivos o de URLs y devuelve una lista de módulos.

- Un functor es sencillamente un valor, como una clase. Se puede definir interactivamente utilizando una sintaxis similar a la de las clases:

```
F=functor $ define skip end
```

Esto define un functor y liga `F` con él. Podemos crear un módulo a partir de `F` así:

```
declare
[M]={Module.apply [F]}
```

Esto crea y enlaza el módulo `M`. La función `Module.apply` toma una lista de valores de tipo functor y devuelve una lista de módulos.

Para otras manipulaciones de los functors, vea la documentación del módulo `Module`.

---

## A.2. Interfaz de la línea de comandos

El sistema Mozart se puede utilizar desde una línea de comandos. Los archivos fuente en Oz se pueden compilar y enlazar. Los archivos fuente para compilar deben contener functors, i.e., empezar con la palabra reservada `functor`. Por ejemplo, suponga que tenemos el archivo fuente `Set.oz`. Podemos crear el functor compilado `Set.ozf` digitando, en la interfaz de la línea de comandos, el comando siguiente:

```
ozc -c Set.oz
```

Podemos crear un archivo `Set`, ejecutable de manera autónoma, digitando lo siguiente:

```
ozc -x Set.oz
```

(En el caso de `Set.oz`, el ejecutable autónomo no hace mayor cosa: sólo define las operaciones de conjuntos.) Por defecto, Mozart utiliza enlaces dinámicos, i.e., los módulos se cargan y se enlazan en el momento en que se necesitan en la aplicación. Esto conserva pequeño, el tamaño de los archivos compilados. Pero, también es posible enlazar todos los módulos importados durante la compilación (enlace estático) de manera que no se necesite el enlace dinámico.

Draft  
684

## B Tipos de Datos Básicos

Wie het kleine niet eert is het grote niet weert.

*Aquel que no le hace honor a las pequeñas cosas no es digno de grandes cosas.*

– Proverbio holandés.

Este apéndice explica los tipos de datos más comunes en Oz junto con algunas operaciones comunes. Los tipos incluidos son los números (incluyendo números enteros y de punto flotante), los caracteres (los cuales se representan como enteros pequeños), los literales (constantes de dos tipos, átomos o nombres), los registros, las tuplas, los pedazos (registros con un conjunto limitado de operaciones), las listas, las cadenas de caracteres (los cuales se representan como listas de caracteres), y las cadenas de caracteres virtuales (cadenas de caracteres representadas como tuplas).

Para cada tipo discutido en este apéndice, existe un módulo Base correspondiente en el sistema Mozart que define todas las operaciones sobre el tipo de datos. En este apéndice se presentan algunas de estas operaciones, pero no todas. Vea la documentación del sistema Mozart para una información completa al respecto [48].

---

### B.1. Números (enteros, flotantes, y caracteres)

El siguiente fragmento de código introduce cuatro variables, I, H, F, y C. En él se ligan I a un entero, H a un entero en notación hexadecimal, F a un flotante, y C al carácter t, en este orden; luego se muestran en el browser I, H, F, y C:

```
declare I H F C in
I = ~5
H = 0xDadBeddedABadBadBabe
F = 5.5
C = &t
{Browse I} {Browse H} {Browse F} {Browse C}
```

Note que ~ (virgulilla) es el símbolo usado para el “menos” unario. En el browser se ve lo siguiente:

```
~5
1033532870595452951444158
5.5
116
```

|                                |                                                                     |
|--------------------------------|---------------------------------------------------------------------|
| <code>&lt;carácter&gt;</code>  | <code>::= (cualquier entero en el rango 0...255)</code>             |
|                                | <code>  `&amp;` &lt;carCar&gt;</code>                               |
|                                | <code>  `&amp;` &lt;seudoCar&gt;</code>                             |
| <code>&lt;carCar&gt;</code>    | <code>::= (cualquier carácter en línea salvo \ y NUL)</code>        |
| <code>&lt;pseudoCar&gt;</code> | <code>::= (`\` seguido de tres dígitos octales)</code>              |
|                                | <code>  (`\x` or `\X` seguido por dos dígitos hexadecimales)</code> |
|                                | <code>  `\a`   `\b`   `\f`   `\n`   `\r`   `\t`</code>              |
|                                | <code>  `\v`   `\\`   `\`   `\"   `^`   `&amp;`</code>              |

Tabla B.1: Sintaxis léxica de los caracteres.

Oz tiene soporte para enteros en notación binaria, octal, decimal, y hexadecimal, los cuales pueden tener cualquier número de dígitos. Un entero octal comienza con un 0 (cero), seguido por cualquier cantidad de dígitos octales, i.e., con valores de 0 a 7. Un entero binario comienza con 0b o 0B (cero seguido de la letra b o B), seguido por cualquier cantidad de dígitos binarios i.e., 0 o 1. Un entero hexadecimal comienza con 0x o 0X (cero seguido de la letra x o X). Los dígitos hexadecimales del 10 al 15 se denotan con las letras a hasta f y A hasta F.

Los flotantes difieren de los enteros en que ellos son aproximaciones de los números reales. Estos son algunos ejemplos de flotantes:

`~3.14159265359 3.5E3 ~12.0e~2 163.`

Note que Mozart utiliza ~ (virgulilla) como el símbolo unario menos, de la misma forma que para los enteros. Los flotantes se representan internamente con doble precisión (64 bits) utilizando el estándar de punto flotante de IEEE. Un flotante debe escribirse con un punto decimal y al menos un dígito antes del punto decimal. Puede haber cero o más dígitos después del punto decimal. Los flotantes pueden ser escalados por potencias de diez agregándoles la letra e o E seguida por un entero decimal (el cual puede ser negativo con el símbolo ~~).

Los caracteres son un subtipo de los enteros en el rango de 0 a 255. Se utiliza la codificación del estándar ISO 8859-1. Este código extiende el código ASCII incluyendo letras y caracteres acentuados de la mayoría de los lenguajes cuyos alfabetos se basan en el Alfabeto Romano. Unicode es un código de 16 bits que extiende el código ASCII incluyendo los caracteres y escrituras específicas (como la dirección de escritura) de la mayoría de los alfabetos utilizados en el mundo. Actualmente no se utiliza en Mozart, pero podría utilizarse en un futuro. Hay cinco maneras de escribir caracteres:

- Un carácter se puede escribir como un entero en el rango 0, 1, ..., 255, de acuerdo a la sintaxis de los enteros presentada antes.
- Un carácter se puede escribir como el símbolo & seguido de una representación específica de un carácter. Existen cuatro representaciones específicas:

|                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{expresión} \rangle ::= \langle \text{expresión} \rangle \langle \text{opBinario} \rangle \langle \text{expresión} \rangle$ |
|                                                                                                                                           |
| { } { }                                                                                                                                   |
| ...                                                                                                                                       |
| $\langle \text{opBinario} \rangle ::= +   -   *   /   \text{div}   \text{mod}   \dots$                                                    |

**Tabla B.2:** Sintaxis de expresiones (en parte).

- Cualquier carácter en línea excepto \ (barra diagonal invertida) y el carácter NUL. Algunos ejemplos son &t, & (note el espacio), y &+. Los caracteres de control en línea son aceptables.
- Una barra diagonal invertida \ seguida de tres dígitos octales, e.g., &\215 es un carácter. El primer dígito no debe ser mayor que 3.
- Una barra diagonal invertida \ seguida por la letra x o X, seguida por dos dígitos hexadecimales, e.g., &\x3f es un carácter.
- Una barra diagonal invertida \ seguida por uno de los caracteres siguientes: a (= \007, campana), b (= \010, tecla de retroceso), f (= \014, *form feed*), n (= \012, línea nueva), r (= \015, *carriage return*), t (= \011, tabulador horizontal), v (= \013, tabulador vertical), \ (= \134, barra diagonal invertida), ' (= \047, comilla sencilla), " (= \042, comilla doble), ` (= \140, comilla invertida), y & (= \046, *ampersand*). Por ejemplo, &\\ es el carácter barra diagonal invertida, i.e., el entero 92 (el código ASCII de \).

En la tabla B.1 se resumen estas posibilidades.

No existe conversión automática en Oz, por tanto 5.0 = 5 lanzará una excepción. En la próxima sección se explican las operaciones básicas sobre los números, incluyendo los procedimientos primitivos para conversión explícita de tipos. El conjunto completo de operaciones sobre los caracteres, enteros, y flotantes se presentan en los módulos `Base Char`, `Float`, e `Int`. Otras operaciones genéricas adicionales sobre todos los números se presentan en el módulo `Base Number`. Vea la documentación para mayor información.

### B.1.1. Operaciones sobre números

Para expresar un cálculo con números, usamos dos tipos de operaciones: operaciones binarias, tales como la adición o la substracción, y aplicaciones de función, tales como conversiones de tipo. En la tabla B.2 se presenta la sintaxis de estas expresiones. En la tabla B.3 se presentan algunas de las operaciones sobre números; otras aparecen en la tabla 2.3. Todos los números, i.e., tanto enteros como flotantes, tienen soporte para la adición, la substracción, y la multiplicación:

## Tipos de Datos Básicos

| Operación         | Descripción                                                  |
|-------------------|--------------------------------------------------------------|
| {IsInt I}         | Devuelve un booleano que dice si I es un entero              |
| {IsFloat F}       | Devuelve un booleano que dice si F es un flotante            |
| {IntToFloat I}    | Devuelve el flotante más cercano al entero I                 |
| {IntToString I}   | Devuelve una cadena de caracteres que describe el entero I   |
| {FloatToInt F}    | Devuelve el entero más cercano al flotante F                 |
| {FloatToString F} | Devuelve una cadena de caracteres que describe el flotante F |
| {Round F}         | Devuelve el flotante entero más cercano al flotante F        |
| {Floor F}         | Devuelve el flotante entero más grande $\leq$ al flotante F  |
| {Ceil F}          | Devuelve el flotante entero más pequeño $\geq$ al flotante F |
| {Sin F}           | Devuelve el seno del flotante F                              |
| {Cos F}           | Devuelve el coseno del flotante F                            |
| {Tan F}           | Devuelve la tangente del flotante F                          |
| {Sqrt F}          | Devuelve la raíz cuadrada del flotante F                     |
| {Exp F}           | Devuelve $e^F$ para el flotante F es un                      |
| {Log F}           | Devuelve $\log_e F$ para el flotante F                       |

Tabla B.3: Algunas operaciones sobre números.

```

declare I Pi Radio Circunferencia in
I = 7 * 11 * 13 + 27 * 37
Pi = 3.1415926536
Radio = 10.
Circunferencia = 2.0 * Pi * Radio

```

Ambos argumentos de +, -, y \* deben ser enteros o flotantes; no se hace ninguna conversión implícita. La aritmética entera es de precisión arbitraria. La aritmética flotante tiene una precisión fija. Los enteros tienen soporte para la división entera (símbolo **div**) y para el módulo (símbolo **mod**). Los flotantes tienen soporte para la división flotante (símbolo /). La división entera trunca la parte fraccional. La división entera y el módulo satisfacen la identidad siguiente:

$$A = B * (A \text{ div } B) + (A \text{ mod } B)$$

Existen varias operaciones de conversión entre flotantes y enteros.

- Hay una operación para convertir un entero en flotante, a saber, **IntToFloat**. Esta operación encuentra el mejor flotante que se aproxime al entero dado. Como los enteros se calculan con precisión arbitraria, puede pasar que un entero sea más grande que un flotante representable. En este caso se devuelve el flotante **inf** (infinito).
- Hay una operación para convertir un flotante en un entero, a saber, **FloatToInt**. Esta operación sigue el modo de redondeo por defecto del estándar de punto flotante de IEEE, i.e., si hay dos posibilidades, entonces escoge el entero par. Por ejemplo, {FloatToInt 2.5} y {FloatToInt 1.5} ambas devuelven el entero 2.

| Operación            | Descripción                                       |
|----------------------|---------------------------------------------------|
| {IsChar C}           | Devuelve un booleano que dice si C es un carácter |
| {Char.toAtom C}      | Devuelve el átomo correspondiente a C             |
| {Char.toLowerCase C} | Devuelve la letra minúscula correspondiente a C   |
| {Char.toUpperCase C} | Devuelve la letra mayúscula correspondiente a C   |

**Tabla B.4:** Algunas operaciones sobre caracteres.

|                                                                          |
|--------------------------------------------------------------------------|
| ⟨expresión⟩ ::= <b>unit</b>   <b>true</b>   <b>false</b>   ⟨átomo⟩   ... |
|--------------------------------------------------------------------------|

**Tabla B.5:** Sintaxis de un literal (en parte).

Esto elimina la tendencia que resultaría de redondear siempre hacia arriba todo flotante terminado en .5.

- Hay tres operaciones para convertir un flotante en otro pero sin parte fraccional: `Floor`, `Ceil`, y `Round`.
  - `Floor` redondea hacia menos infinito, e.g., `{Floor ~3.5}` da `~4.0` y `{Floor 4.6}` da `4.0`.
  - `Ceil` redondea hacia más infinito, e.g., `{Ceil ~3.5}` da `~3.0` y `{Ceil 4.6}` da `5.0`.
  - `Round` redondea hacia el par más cercano, e.g., `{Round 4.5}`=4 y `{Round 5.5}`=6. `Round` es idéntico a `FloatToInt` salvo que devuelve un flotante, i.e., `{Round X}` = `{IntToFloat {FloatToInt X}}`.

### B.1.2. Operaciones sobre caracteres

Todas las operaciones sobre los enteros también funcionan sobre los caracteres. Hay unas pocas operaciones adicionales que funcionan sólo sobre caracteres. En la tabla B.4 se listan algunas de ellas. En el módulo Base Char se encuentran todas.

```

⟨átomo⟩ ::= (car minúscula) { (car alfanumérico) } (salvo palabras reservadas)
| ' ' { ⟨carAtom⟩ | ⟨seudoCar⟩ } ' '
⟨carAtom⟩ ::= (cualquier carácter en línea salvo ' ', '\n', and NUL)
⟨seudoCar⟩ ::= ('\' seguido de tres dígitos octales)
| ('\'x' o '\'X' seguido de dos dígitos hexadecimales)
| '\a' | '\b' | '\f' | '\n' | '\r' | '\t'
| '\v' | '\\\\' | '\\'' | '\"' | '\\^' | '\\&'
```

**Tabla B.6:** Sintaxis léxica de un átomo.

## B.2. Literales (átomos y nombres)

Los tipos atómicos son tipos cuyos miembros no tienen estructura interna.<sup>1</sup> En la sección anterior se presentó una clase de tipo atómico, a saber, los números. Además de los números, los literales son una segunda clase de tipo atómico (ver tablas B.5 y B.6). Los literales pueden ser átomos o nombres. Un átomo es un valor cuya identidad está determinada por una secuencia de caracteres imprimibles. Un átomo se puede escribir de dos maneras. Primero, como una secuencia de caracteres alfanuméricos comenzando por una letra minúscula. Esta secuencia no puede ser una palabra reservada del lenguaje. Segundo, por una secuencia arbitraria de caracteres imprimibles encerrados entre comillas sencillas. Estos son algunos átomos válidos:

```
a foo '=' ':=' 'Oz 3.0' 'Hola Mundo' 'if' '\n,\n' una_persona
```

No hay confusión entre la palabra reservada **if** y el átomo `'if'` debido a las comillas. El átomo `'\n,\n'` consiste de cuatro caracteres. Los átomos se ordenan lexicográficamente, basados en la codificación subyacente para caracteres del estándar ISO 8859-1.

Los nombres son una segunda clase de literal. Un nombre es un valor atómico único que no se puede falsificar ni imprimir. A diferencia de los números o los átomos, los nombres son verdaderamente atómicos, en el sentido original de la palabra: no se pueden descomponer de ninguna manera. Los nombres sólo tienen dos operaciones definidas sobre ellos: la creación y la comparación de igualdad. La única manera de crear un nombre es invocando la función `{NewName}`, la cual devuelve un nombre nuevo con garantía de ser único. Note que en la tabla B.5 no existe representación para los nombres. La única manera de referenciar un nombre es a través de una variable que esté ligada a ese nombre. Tal como se explica en

1. Pero al igual que los átomos físicos, si se utilizan las herramientas adecuadas, los valores atómicos se pueden descomponer algunas veces, e.g., los números tienen una representación binaria de ceros y unos y los átomos tienen una representación imprimible como una secuencia de caracteres.

| Operación        | Descripción                                                   |
|------------------|---------------------------------------------------------------|
| {IsAtom A}       | Devuelve un booleano que dice si A es un átomo                |
| {AtomToString A} | Devuelve la cadena de caracteres correspondiente al átomo A   |
| {StringToAtom S} | Devuelve el átomo correspondiente a la cadena de caracteres S |

Tabla B.7: Algunas operaciones sobre átomos.

|                                    |                                                                                                                                                                      |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{expresión} \rangle$ | $::= \langle \text{etiqueta} \rangle \wedge (\wedge \{ [ \langle \text{campo} \rangle \wedge : \wedge ] \langle \text{expresión} \rangle \} \wedge ) \wedge   \dots$ |
| $\langle \text{etiqueta} \rangle$  | $::= \text{unit}   \text{true}   \text{false}   \langle \text{variable} \rangle   \langle \text{átomo} \rangle$                                                      |
| $\langle \text{campo} \rangle$     | $::= \text{unit}   \text{true}   \text{false}   \langle \text{variable} \rangle   \langle \text{átomo} \rangle   \langle \text{ent} \rangle$                         |
| $\langle \text{opBinario} \rangle$ | $::= \wedge \wedge   \langle \text{consOpBin} \rangle   \dots$                                                                                                       |
| $\langle \text{consOpBin} \rangle$ | $::= \wedge \# \wedge   \dots$                                                                                                                                       |

Tabla B.8: Sintaxis de registros y tuplas (en parte).

la sección 3.7, los nombres juegan un papel muy importante para encapsular de manera segura los TADs.

Existen tres nombres especiales con palabras reservadas para ellos. Las palabras reservadas son **unit**, **true**, y **false**. Los nombres **true** y **false** se utilizan para denotar los valores booleanos verdadero y falso. El nombre **unit** se utiliza frecuentemente como testigo de sincronización en programas concurrentes. Estos son algunos ejemplos:

```
local X Y B in
 X = foo
 {NewName Y}
 B = true
 {Browse [X Y B]}
end
```

### B.2.1. Operaciones sobre átomos

En la tabla B.7 se presentan las operaciones del módulo Base `Atom` y algunas de las operaciones del módulo Base `String`, relacionadas con átomos.

---

## B.3. Registros y tuplas

Los registros son estructuras de datos que permiten agrupar referencias del lenguaje en una sola entidad. El siguiente es un registro que agrupa cuatro variables:

*Tipos de Datos Básicos*

```
árbol(llave:I valor:Y izq:AI der:AD)
```

El registro tiene cuatro componentes y la etiqueta `árbol`. Para evitar ambigüedad, no debe existir espacio entre la etiqueta y el paréntesis izquierdo. Cada componente consiste de un identificador, denominado campo, y una referencia al almacén. Un campo puede ser un literal o un entero. En la tabla B.8 se presenta la sintaxis de los registros y las tuplas. El registro anterior tiene cuatro campos, `llave`, `valor`, `izq`, y `der`, que identifican cuatro referencias del lenguaje, `I`, `Y`, `AI`, y `AD`.

Se permite omitir campos en la sintaxis de registros. En ese caso, el campo será un entero, empezando por 1 para el primer caso, e incrementando de a 1 para cada componente siguiente que no tenga un campo. Por ejemplo, el registro `árbol(llave:I valor:Y AI AD)` es idéntico al registro `árbol(llave:I valor:Y 1:AI 2:AD)`.

El orden de los componentes etiquetados no tiene importancia; puede ser cambiado sin cambiar el registro. Decimos que estos componentes no tienen orden. El orden de los componentes no etiquetados sí importa, pues determina cómo se numeran los campos. Es como si fueran dos “mundos”: el mundo ordenado y el mundo sin orden. Ninguno tiene efectos sobre el otro y pueden ser intercalados de cualquier manera. Todas las notaciones siguientes denotan el mismo registro:

|                                             |                                           |
|---------------------------------------------|-------------------------------------------|
| <code>árbol(llave:I valor:Y AI AD)</code>   | <code>árbol(valor:Y llave:I AI AD)</code> |
| <code>árbol(llave:I AI valor:Y AD)</code>   | <code>árbol(valor:Y AI llave:I AD)</code> |
| <code>árbol(llave:I I AI AD valor:Y)</code> | <code>árbol(valor:Y AI AD llave:I)</code> |
| <code>árbol(AI llave:I valor:Y AD)</code>   | <code>árbol(AI valor:Y llave:I AD)</code> |
| <code>árbol(AI llave:I AD valor:Y)</code>   | <code>árbol(AI valor:Y AD llave:I)</code> |
| <code>árbol(AI AD llave:I valor:Y)</code>   | <code>árbol(AI AD valor:Y llave:I)</code> |

Dos registros son idénticos si tienen los mismos conjuntos de componentes y los componentes ordenados están en el mismo orden.

Es un error si un campo aparece más de una vez. Por ejemplo, las notaciones `árbol(llave:I llave:J)` y `árbol(1:I valor:Y AI AD)` están erradas. El error se descubre al momento de la construcción del registro. Esto puede ser en tiempo de compilación o en tiempo de ejecución. Sin embargo, `árbol(3:I valor:Y AI AD)` y `árbol(4:I valor:Y AI AD)` están correctas, pues ningún campo aparece más de una vez. Las características enteras no tienen que ser consecutivas.

### B.3.1. Tuplas

Si el registro solamente tiene campos enteros consecutivos empezando de 1, entonces se llama una tupla. Todos los campos de una tupla se pueden omitir. Considere la tupla siguiente:

```
árbol(I Y AI AD)
```

Es exactamente la misma que la tupla siguiente:

```
árbol(1:I 2:Y 3:AI 4:AD)
```

| Operación         | Descripción                                           |
|-------------------|-------------------------------------------------------|
| R.F               | Devuelve el campo F de R                              |
| {HasFeature R F}  | Devuelve un booleano que dice si el campo F está en R |
| {IsRecord R}      | Devuelve un booleano que dice si R es un registro     |
| {MakeRecord L Fs} | Devuelve un registro con etiqueta L y campos Fs       |
| {Label R}         | Devuelve la etiqueta de R                             |
| {Arity R}         | Devuelve la lista de campos (aridad) de R             |
| {Record.toList R} | Devuelve la lista de campos R, en orden               |
| {Width R}         | Devuelve el número de campos de R                     |
| {AdjoinAt R F X}  | Devuelve R aumentado con el campo F y el valor X      |
| {Adjoin R1 R2}    | Devuelve R1 aumentado con todos los campos de R2      |

**Tabla B.9:** Algunas operaciones sobre registros.

Las tuplas cuya etiqueta es `#` tienen otra notación usando el símbolo # como un operador “mixfix” (ver apéndice C.4). Esto significa que a#b#c es una tupla con tres argumentos, a saber, `(a b c)`. Tenga cuidado de no confundirla con la pareja a#(b#c), cuyo segundo argumento es, a su vez, la pareja b#c. La notación *mixfix* sólo se puede usar con tuplas de por lo menos dos argumentos. Esta notación se usa para cadenas de caracteres virtuales (ver sección B.7).

### B.3.2. Operaciones sobre registros

En la tabla B.9 se presentan unas pocas operaciones básicas sobre los registros. Existen muchas otras operaciones en el módulo Base Record. En este apéndice se muestran tan solo unas pocas, a saber, aquellas que tienen que ver con extraer información de los registros y con construir registros nuevos. Para seleccionar un campo de un componente de un registro, utilizamos el operador infijo punto, e.g., árbol(llave:I valor:Y AI AD).valor devuelve Y. Para comparar dos registros, utilizamos la operación de comprobación de igualdad. Dos registros son el mismo si tienen el mismo conjunto de campos y las referencias en cada campo son las mismas.

La aridad de un registro es una lista de campos del registro ordenada lexicográficamente. La invocación {Arity R} se ejecuta tan pronto R esté ligada a un registro y devuelve la aridad del registro. No la confunda con la función width, la cual devuelve el número de campos del registro. Si se alimenta Oz con la declaración

```
declare T W L R in
T=árbol(llave:a izq:L der:R valor:1)
W=árbol(a L R 1)
{Browse {Arity T}}
{Browse {Arity W}}
```

se desplegará en el browser

*Tipos de Datos Básicos*

| Operación       | Descripción                                         |
|-----------------|-----------------------------------------------------|
| {MakeTuple L N} | Devuelve la tupla con etiqueta L y campos 1, ..., N |
| {IsTuple T}     | Devuelve un booleano que dice si T es una tupla     |

**Tabla B.10:** Algunas operaciones sobre tuplas.

|                                                                                                   |
|---------------------------------------------------------------------------------------------------|
| $\langle \text{expresión} \rangle ::= `[\,` \{ \langle \text{expresión} \rangle \} + `]`   \dots$ |
| $\langle \text{consOpBin} \rangle ::= ` `   \dots$                                                |

**Tabla B.11:** Sintaxis de lista (en parte).

```
[llave izq der valor]
[1 2 3 4]
```

La función {AdjoinAt R1 F x} devuelve el registro resultante de adjuntar (i.e., añadir) a R1 el componente de campo F y referencia x. El registro R1 no se modifica. Si R1 ya tenía el campo F, entonces el resultado es idéntico a R1 salvo para el campo R1.F, cuyo valor se vuelve x. En cualquier otro caso, el campo F se añade a R1. Por ejemplo:

```
declare T W L R in
T=árbol(llave:a izq:L der:R valor:1)
W=árbol(a L R 1)
{Browse {AdjoinAt T 1 b}}
{Browse {AdjoinAt W llave b}}
```

mostrará en el browser

```
árbol(b llave:a izq:L der:R valor:1)
árbol(a L R 1 llave:b)
```

La operación {Adjoin R1 R2} devuelve el mismo resultado como si AdjoinAt fuera invocada sucesivamente, empezando con R1 e iterando a lo largo de todos los campos de R2.

### B.3.3. Operaciones sobre tuplas

Todas las operaciones sobre registros también funcionan sobre tuplas. Hay unas pocas operaciones adicionales que funcionan sólo sobre las tuplas. En la tabla B.10 se listan algunas de ellas. El módulo Base Tuple se encuentran todas.

## B.4. Pedazos (registros limitados)

Un pedazo<sup>2</sup> es la terminología que usa Mozart para un registro con un conjunto limitado de operaciones. Solamente hay dos operaciones básicas sobre los pedazos: crear un pedazo a partir de un registro y extraer información con el operador de selección de campo “.”:

```
declare
C={Chunk.new cualquierregistro(a b c)} % Creación de un pedazo
F=C.2 % Selección de campo de un pedazo
```

Las operaciones `Label` y `Arity` no están definidas y la unificación no es posible. Los pedazos son una forma de “envolver” la información de manera que el acceso a ella sea restringido, i.e., sólo aquellas computaciones que conozcan los campos pueden acceder a la información. Si el campo es un valor de tipo nombre, entonces este sólo lo puede conocer una computación si se le pasa explícitamente; no puede ser adivinado. De esta manera los pedazos y los nombres son útiles para construir abstracciones de datos seguras. Por ejemplo, los utilizamos en la sección 3.7.5 para definir un TAD pila seguro. Los pedazos se utilizan en algunos módulos de la biblioteca para proveer encapsulación segura de datos, tal como se hace en el módulo `ObjectSupport`.

---

## B.5. Listas

Una lista es el átomo `nil` representando la lista vacía o una tupla con `|` como operador infijo y dos argumentos que corresponden, respectivamente, a la cabeza y la cola de la lista. Los campos de los dos argumentos están numerados 1 y 2. La cabeza puede ser cualquier tipo de dato, mientras que la cola es una lista. Denominamos esta tupla un par lista. Con frecuencia, también se denomina una celda “cons” porque en Lisp se crea con una operación denominada `cons`. Lisp es el más antiguo de los lenguajes de procesamiento de listas y es pionero en muchos conceptos de listas y en su terminología. Cuando el segundo argumento no es una lista, entonces la tupla se denomina un par punto, debido a que Lisp lo escribe con un operador punto infijo. En nuestra notación, una lista con las letras `a`, `b`, y `c` se escribe

`a|b|c|nil`

Proveemos una notación más concisa para las listas completas (i.e., cuando el argumento más a la derecha es `nil`):

`[a b c]`

En la tabla B.11 se muestra la sintaxis de estas dos formas de escribir una lista. La lista parcial que contiene los elementos `a` y `b` y cuya cola es la variable `x` se ve así:

---

2. Nota del traductor: *chunk* en la versión en inglés.

a | b | x

También se puede usar la notación estándar de registros para las listas:

' | ' ( a ' | ' ( b X ) )

o, aún más (hacer explícitos los nombres de los campos):

```
' | `'(1:a 2:` | `'(1:b 2:X))
```

Las listas circulares están permitidas. Por ejemplo, lo siguiente es legal:

```
declare X in
X=a|b|X
{Browse X}
```

Por defecto, el browser despliega la lista sin tomar en cuenta lo que está compartido, i.e., sin tomar en cuenta múltiples referencias a la misma parte de la lista. En la lista x, después de los primeros dos elementos a y b, encontramos de nuevo x. Por defecto, el browser ignora esta referencia repetida. El browser despliega x así:

Para evitar ciclos infinitos, el browser tiene una límite de profundidad ajustable. Las tres comas , , , representan la parte de la lista que no se despliega. Seleccione **Graph** en la entrada **Representation** del menú **Options** del browser y alímate la interfaz de Oz, de nuevo, con el mismo fragmento de programa. Ahora se desplegará la lista como un grafo (ver figura B.1):

C1=a | b | C1

El browser introduce la variable nueva `c1` para referenciar otra parte de la lista. Vea el manual del browser para mayor información sobre lo que el browser puede desplegar.

### B.5.1. Operaciones sobre listas

En la tabla B.12 se presentan unas pocas operaciones básicas sobre listas. Existen muchas otras operaciones en el módulo Base List. Este es un cálculo simbólico sencillo con listas:

```
declare A B in
A=[a b c]
B=[1 2 3 4]
{Browse {Append A B}}
```

En el browser se despliega la lista [a b c 1 2 3 4]. Como todas las operaciones, éstas exhiben el comportamiento correcto de las variables de flujo de datos. Por ejemplo, {Length a|b|x} se bloquea hasta que x sea ligada. Las operaciones `Sort`, `Map`, `ForAll`, `Filter`, y `FoldL` son ejemplos de operaciones de alto orden, i.e., operaciones que toman funciones o procedimientos como argumentos. Hablamos sobre la programación de alto orden en el capítulo 3. Por ahora, este es un ejemplo

| Operación           | Descripción                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------------------|
| {Append L1 L2}      | Devuelve la concatenación de L1 y L2                                                                      |
| {Member X L}        | Devuelve un booleano que dice si X está en L                                                              |
| {Length L}          | Devuelve la longitud de L                                                                                 |
| {List.drop L N}     | Devuelve L sin los primeros N elementos, o nil si es más corta                                            |
| {List.last L}       | Devuelve el último elemento de la lista no vacía L                                                        |
| {Sort L F}          | Devuelve L ordenada de acuerdo a la función de comparación F                                              |
| {Map L F}           | Devuelve la lista resultante de aplicar F a cada elemento de L                                            |
| {ForAll L P}        | Aplica el procedimiento unario P a cada elemento de L                                                     |
| {Filter L F}        | Devuelve la lista de los elementos de L para los cuales F da true                                         |
| {FoldL L F N}       | Devuelve el valor obtenido al insertar F entre todos los elementos de L                                   |
| {Flatten L}         | Devuelve la lista de todos los elementos de L que no son lista, no importa la profundidad del anidamiento |
| {List.toTuple A L}  | Devuelve una tupla con etiqueta A y campos ordenados de L                                                 |
| {List.toRecord A L} | Devuelve un registro con etiqueta A y campos F#X en L                                                     |

Tabla B.12: Algunas operaciones sobre listas.

que da el sabor de lo que se puede hacer:

```
declare L in
L=[juan pablo jorge renato]
{Browse {Sort L Value.'<'}}
```

ordena L de acuerdo a la función de comaparación '`<`' y despliega el resultado:

```
[jorge juan pablo renato]
```

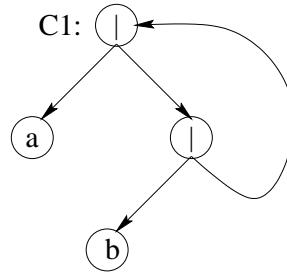
Visto como un operador infijo, la comparación se escribe `x<y`, pero la operación de comparación como tal está en el módulo Base `Value`. Su nombre completo es `Value.'<'`. Los módulos se explican en la sección 3.9.

## B.6. Cadenas de caracteres

Las listas cuyos elementos son códigos de caracteres se denominan cadenas de caracteres. Por ejemplo:

```
"Mozart 1.2.3"
```

corresponde a la lista:



**Figura B.1:** Representación en forma de grafo de la lista infinita  $C1=a|b|C1$ .

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{expresión} \rangle ::= \langle \text{cadenaCars} \rangle \mid \dots$<br>$\langle \text{cadenaCars} \rangle ::= " " \{ \langle \text{carCadena} \rangle \mid \langle \text{seudoCar} \rangle \} " "$<br>$\langle \text{carCadena} \rangle ::= (\text{cualquier carácter en línea salvo } ", \backslash, \text{ y NUL})$<br>$\langle \text{seudoCar} \rangle ::= (" \backslash " \text{ seguido de tres dígitos octales})$<br>$\quad \mid (" \backslash x " \text{ o } " \backslash X " \text{ seguido de dos dígitos hexadecimales})$<br>$\quad \mid " \backslash a " \mid " \backslash b " \mid " \backslash f " \mid " \backslash n " \mid " \backslash r " \mid " \backslash t "$<br>$\quad \mid " \backslash v " \mid " \backslash \backslash " \mid " \backslash \^{} " \mid " \backslash " " \mid " \backslash \~{} " \mid " \backslash \& "$ |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Tabla B.13:** Sintaxis léxica de las cadenas de caracteres.

[ 77 111 122 97 114 116 32 49 46 50 46 51 ]

o de forma equivalente a:

[ &M &O &z &a &r &t & &1 &. &2 &. &3 ]

Utilizar las listas para representar las cadenas de caracteres es conveniente porque todas las operaciones sobre listas están disponibles para realizar cálculos simbólicos con cadenas de caracteres. Las operaciones sobre caracteres junto con las operaciones sobre listas se pueden usar para realizar cálculos con el contenido de las cadenas de caracteres. La sintaxis de las cadenas de caracteres se muestra en la tabla B.13. El código del carácter NUL que se menciona en la tabla es 0 (cero). Vea la sección B.1 para una explicación del significado de `"\a"`, `"\b"`, etc.

Existe otra representación para las secuencias de caracteres, más eficiente en memoria, denominada *bytestring*. Esta representación sólo debe ser usada en caso de ser necesaria debido a limitaciones de memoria.

## B.7. Cadenas virtuales de caracteres

| Operación                     | Descripción                                                                                                                                      |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| {VirtualString.toString VS}   | Devuelve una cadena de caracteres con los mismos caracteres que VS                                                                               |
| {VirtualString.toAtom VS}     | Devuelve un átomo con los mismos caracteres que VS                                                                                               |
| {VirtualString.length VS}     | Devuelve el número de caracteres de VS                                                                                                           |
| {Value.toVirtualString X D W} | Devuelve una cadena virtual de caracteres que representa el valor X, donde los registros están limitados en profundidad por D y en anchura por W |

**Tabla B.14:** Algunas operaciones sobre cadenas virtuales de caracteres.

Una cadena virtual de caracteres es una tupla con etiqueta `#` que representa una cadena de caracteres. La cadena virtual de caracteres reúne diferentes subcadenas concatenadas con concatenación virtual. Es decir, la concatenación nunca se realiza en realidad, lo cual ahorra tiempo y memoria. Por ejemplo, la cadena virtual de caracteres

```
123#" - "#23#" es "#(123-23)
```

representa la cadena de caracteres

```
"123-23 es 100"
```

Salvo en casos especiales, toda operación de la biblioteca que espere una cadena de caracteres puede recibir, en su lugar, una cadena virtual de caracteres. Por ejemplo, las cadenas virtuales de caracteres se pueden utilizar para todas las operaciones de E/S. Una cadena virtual de caracteres puede estar compuesta por números, cadenas de caracteres, cadenas virtuales de caracteres (i.e., tuplas etiquetadas con `#`), y todos los átomos excepto nil y `#`. En la tabla B.14 se presentan algunas operaciones sobre cadenas virtuales de caracteres.

# Draft<sup>700</sup>

*Tipos de Datos Básicos*

---

## C Sintaxis del Lenguaje

El diablo está en los detalles.

– Proverbio tradicional.

Dios está en los detalles.

– Proverbio tradicional.

¡Yo no sé que hay en estos detalles  
pero debe ser algo importante!

– Proverbio irreverente.

En este apéndice se define la sintaxis completa del lenguaje utilizado en el libro, incluyendo las conveniencias sintácticas. El lenguaje es un subconjunto del lenguaje Oz tal como fue implementado en el sistema Mozart. El apéndice está dividido en seis secciones:

- En la sección C.1 se define la sintaxis de las declaraciones interactivas, i.e., declaraciones con las que se puede alimentar la interfaz interactiva.
- En la sección C.2 se define la sintaxis de las declaraciones y las expresiones.
- En la sección C.3 se define la sintaxis de los símbolos no terminales que se necesitan para definir las declaraciones y las expresiones.
- En la sección C.4 se presenta la lista de los operadores del lenguaje con su precedencia y su asociatividad.
- En la sección C.5 se presenta la lista de las palabras reservadas del lenguaje.
- En la sección C.6 se define la sintaxis léxica del lenguaje, i.e., cómo se transforma una secuencia de caracteres en una secuencia de lexemas.

Para ser precisos, en este apéndice se define una sintaxis independiente del contexto para un superconjunto del lenguaje. Esto conserva la sintaxis sencilla y fácil de leer. La desventaja de una sintaxis independiente del contexto es que con ella no se capturan todas las condiciones sintácticas de los programas legales. Por ejemplo, considere la declaración **local** *x* **in** ⟨declaración⟩ **end**. La declaración que la contenga debería declarar todos los identificadores de variables libres que haya en ⟨declaración⟩, salvo, posiblemente, *x*. Esta condición no es independiente del contexto.

En este apéndice se define la sintaxis de un subconjunto de todo el lenguaje Oz, tal como se definió en [47, 67]. Este apéndice difiere de [67] de varias maneras:

|                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{declaraciónInter} \rangle ::= \langle \text{declaración} \rangle$ $  \textbf{declare} \{ \langle \text{parteDeclaración} \rangle \} + [ \langle \text{declaraciónInter} \rangle ]$ $  \textbf{declare} \{ \langle \text{parteDeclaración} \rangle \} + \textbf{in} \langle \text{declaraciónInter} \rangle$ |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Tabla C.1:** Declaraciones interactivas.

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{declaración} \rangle ::= \langle \text{constAnid(declaración)} \rangle   \langle \text{decAnid(variable)} \rangle$ $  \textbf{skip}   \langle \text{declaración} \rangle \langle \text{declaración} \rangle$ $\langle \text{expresión} \rangle ::= \langle \text{constAnid(expresión)} \rangle   \langle \text{decAnid(`$') } \rangle$ $  \langle \text{opUnario} \rangle \langle \text{expresión} \rangle$ $  \langle \text{expresión} \rangle \langle \text{opBinEval} \rangle \langle \text{expresión} \rangle$ $  `\$`   \langle \text{term} \rangle   \textbf{self}$ $\langle \text{declaraciónEn} \rangle ::= [ \{ \langle \text{parteDeclaración} \rangle \} + \textbf{in} ] \langle \text{declaración} \rangle$ $\langle \text{expresiónEn} \rangle ::= [ \{ \langle \text{parteDeclaración} \rangle \} + \textbf{in} ] [ \langle \text{declaración} \rangle ] \langle \text{expresión} \rangle$ $\langle \text{en(declaración)} \rangle ::= \langle \text{declaraciónEn} \rangle$ $\langle \text{en(expresión)} \rangle ::= \langle \text{expresiónEn} \rangle$ |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

**Tabla C.2:** Declaraciones y expresiones.

introduce construcciones que se pueden anidar, declaraciones que se pueden anidar, y los términos para factorizar las partes comunes de la sintaxis de las declaraciones y de las expresiones; define las declaraciones interactivas y los ciclos **for**; deja por fuera la traducción al lenguaje núcleo (la cual se presenta para cada abstracción lingüística en el texto principal del libro); y realiza otras simplificaciones sencillas por claridad (pero sin sacrificar la precisión).

## C.1. Declaraciones interactivas

En la tabla C.1 se presenta la sintaxis de las declaraciones interactivas. Una declaración interactiva es un superconjunto de una declaración; además de todas las declaraciones regulares, puede contener una declaración **declare**. La interfaz interactiva siempre debe ser alimentada con declaraciones interactivas. Todos los identificadores de variables libres en la declaración interactiva deben existir en el ambiente global; de otra forma el sistema señala el error “variable not introduced”.

## C.2. Declaraciones y expresiones

```

⟨constAnid(α)⟩ ::= ⟨expresión⟩ (`= ` | `:= ` | `, `) ⟨expresión⟩
| ` { ` ⟨expresión⟩ { ⟨expresión⟩ } ` } `
| local { ⟨parteDeclaración⟩ }+ in [⟨declaración⟩] ⟨α⟩ end
| ` (` ⟨en(α)⟩ `) `
| if ⟨expresión⟩ then ⟨en(α)⟩
{ elseif ⟨expresión⟩ then ⟨en(α)⟩ }
[else ⟨en(α)⟩] end
| case ⟨expresión⟩ of ⟨patrón⟩ [andthen ⟨expresión⟩] then ⟨en(α)⟩
{ ` [] ` ⟨patrón⟩ [andthen ⟨expresión⟩] then ⟨en(α)⟩ }
[else ⟨en(α)⟩] end
| for { ⟨decCiclo⟩ }+ do ⟨en(α)⟩ end
| try ⟨en(α)⟩
[catch ⟨patrón⟩ then ⟨en(α)⟩
{ ` [] ` ⟨patrón⟩ then ⟨en(α)⟩ }]
[finally ⟨en(α)⟩] end
| raise ⟨expresiónEn⟩ end
| thread ⟨en(α)⟩ end
| lock [⟨expresión⟩ then] ⟨en(α)⟩ end

```

Tabla C.3: Construcciones que se pueden anidar (no declaraciones).

```

⟨decAnid(α)⟩ ::= proc ` { ` α { ⟨patrón⟩ } ` } ` ⟨declaraciónEn⟩ end
| fun [lazy] ` { ` α { ⟨patrón⟩ } ` } ` ⟨expresiónEn⟩ end
| functor α
[import { ⟨variable⟩ [at ⟨átomo⟩]
| ⟨variable⟩ ` (` { ⟨átomo⟩ | ⟨ent⟩) [` : ` ⟨variable⟩] }+ `) `+
] +
[export { [⟨átomo⟩ | ⟨ent⟩) ` : `] ⟨variable⟩ }+]
define { ⟨parteDeclaración⟩ }+ [in ⟨declaración⟩] end
| class α { ⟨descriptorClase⟩ }
{ meth ⟨cabezaMétodo⟩ [` = ` ⟨variable⟩]
(⟨expresiónEn⟩ | ⟨declaraciónEn⟩) end }
end

```

Tabla C.4: Declaraciones que se pueden anidar.

|                                                                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{term} \rangle ::= [ \text{'!'} ] \langle \text{variable} \rangle   \langle \text{ent} \rangle   \langle \text{flot} \rangle   \langle \text{caracter} \rangle$   |
| $  \langle \text{átomo} \rangle   \langle \text{cadenaCars} \rangle   \text{unit}   \text{true}   \text{false}$                                                                 |
| $  \langle \text{etiqueta} \rangle [ \{ [ \langle \text{campo} \rangle [ \text{':'} ] \langle \text{expresión} \rangle \} ] ]$                                                  |
| $  \langle \text{expresión} \rangle \langle \text{consOpBin} \rangle \langle \text{expresión} \rangle$                                                                          |
| $  [ \{ \langle \text{expresión} \rangle \} + ] ]$                                                                                                                              |
| $\langle \text{patrón} \rangle ::= [ \text{'!'} ] \langle \text{variable} \rangle   \langle \text{ent} \rangle   \langle \text{flot} \rangle   \langle \text{caracter} \rangle$ |
| $  \langle \text{átomo} \rangle   \langle \text{cadenaCars} \rangle   \text{unit}   \text{true}   \text{false}$                                                                 |
| $  \langle \text{etiqueta} \rangle [ \{ [ \langle \text{campo} \rangle [ \text{':'} ] \langle \text{patrón} \rangle \} [ \text{'} \dots \text{'}] ] ]$                          |
| $  \langle \text{patrón} \rangle \langle \text{consOpBin} \rangle \langle \text{patrón} \rangle$                                                                                |
| $  [ \{ \langle \text{patrón} \rangle \} + ] ]$                                                                                                                                 |

Tabla C.5: Términos y patrones.

En la tabla C.2 se presenta la sintaxis de las declaraciones y las expresiones. Muchas de las construcciones del lenguaje se pueden usar ya sea en la posición de una declaración o en la posición de una expresión. Decimos que tales construcciones se pueden anidar. Escribimos las reglas gramaticales para presentar su sintaxis sólo una vez, de forma que funcione tanto para las posiciones de las declaraciones como para las posiciones de las expresiones. En la tabla C.3 se presenta la sintaxis de las construcciones que se pueden anidar, sin incluir las declaraciones. En la tabla C.4 se presenta la sintaxis de las declaraciones que se pueden anidar. Las reglas gramaticales para las construcciones y declaraciones que se pueden anidar consisten en plantillas de un argumento. La plantilla se instancia cada vez que se utiliza. Por ejemplo,  $\langle \text{constAnid}(\alpha) \rangle$  define la plantilla de las construcciones que se pueden anidar sin incluir las declaraciones. La plantilla se utiliza dos veces, una como  $\langle \text{constAnid}(\text{declaración}) \rangle$  y otra como  $\langle \text{constAnid}(\text{expresión}) \rangle$ , y cada una corresponde a una regla gramatical.

### C.3. No terminales para las declaraciones y las expresiones

En las tablas C.5 y C.6 se definen los símbolos no terminales que se necesitan para completar la sintaxis de las declaraciones y las expresiones de la sección anterior. En la tabla C.5 se define la sintaxis de los términos y de los patrones. Note la relación estrecha entre términos y patrones. Ambos se usan para definir valores parciales. Sólo hay dos diferencias: (1) los patrones sólo pueden contener identificadores de variables, mientras que los términos pueden contener expresiones, y (2) los patrones pueden ser parciales (utilizando  $\text{'} \dots \text{'}$ ), mientras que los términos no.

En la tabla C.6 se definen los símbolos no terminales para las partes de definición de las declaraciones y los ciclos, para los operadores unarios, para los operadores binarios (operadores de “construcción”  $\langle \text{consOpBin} \rangle$  y operadores de “evaluación”

|                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{parteDeclaración} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{patrón} \rangle \stackrel{\sim}{=} \langle \text{expresión} \rangle \mid \langle \text{declaración} \rangle$                                                                                                                                                                                                          |
| $\langle \text{decCiclo} \rangle ::= \langle \text{variable} \rangle \text{ in } \langle \text{expresión} \rangle [ \stackrel{\sim}{\dots} \stackrel{\sim}{\text{expresión}} ] [ \stackrel{\sim}{;} \stackrel{\sim}{\text{expresión}} ]$                                                                                                                                                                              |
| $\quad \mid \langle \text{variable} \rangle \text{ in } \langle \text{expresión} \rangle \stackrel{\sim}{;} \stackrel{\sim}{\text{expresión}}$                                                                                                                                                                                                                                                                        |
| $\quad \mid \text{break } \stackrel{\sim}{:} \stackrel{\sim}{\text{variable}}$                                                                                                                                                                                                                                                                                                                                        |
| $\quad \mid \text{continue } \stackrel{\sim}{:} \stackrel{\sim}{\text{variable}}$                                                                                                                                                                                                                                                                                                                                     |
| $\quad \mid \text{return } \stackrel{\sim}{:} \stackrel{\sim}{\text{variable}}$                                                                                                                                                                                                                                                                                                                                       |
| $\quad \mid \text{default } \stackrel{\sim}{:} \stackrel{\sim}{\text{expresión}}$                                                                                                                                                                                                                                                                                                                                     |
| $\quad \mid \text{collect } \stackrel{\sim}{:} \stackrel{\sim}{\text{variable}}$                                                                                                                                                                                                                                                                                                                                      |
| $\langle \text{opUnario} \rangle ::= \stackrel{\sim}{\sim} \mid \stackrel{\sim}{@} \mid \stackrel{\sim}{!!}$                                                                                                                                                                                                                                                                                                          |
| $\langle \text{opBinario} \rangle ::= \langle \text{consOpBin} \rangle \mid \langle \text{opBinEval} \rangle$                                                                                                                                                                                                                                                                                                         |
| $\langle \text{consOpBin} \rangle ::= \stackrel{\sim}{\#} \mid \stackrel{\sim}{\cdot} \mid \stackrel{\sim}{\wedge}$                                                                                                                                                                                                                                                                                                   |
| $\langle \text{opBinEval} \rangle ::= \stackrel{\sim}{+} \mid \stackrel{\sim}{-} \mid \stackrel{\sim}{\ast} \mid \stackrel{\sim}{/} \mid \text{div} \mid \text{mod} \mid \stackrel{\sim}{\cdot} \mid \text{andthen} \mid \text{orelse}$                                                                                                                                                                               |
| $\quad \mid \stackrel{\sim}{=:} \mid \stackrel{\sim}{\cdot}, \stackrel{\sim}{\cdot} \mid \stackrel{\sim}{\cdot} \stackrel{\sim}{=} \mid \stackrel{\sim}{\cdot} \backslash \stackrel{\sim}{=} \mid \stackrel{\sim}{\cdot} < \stackrel{\sim}{\cdot} \mid \stackrel{\sim}{\cdot} = < \stackrel{\sim}{\cdot} \mid \stackrel{\sim}{\cdot} > \stackrel{\sim}{\cdot} \mid \stackrel{\sim}{\cdot} > = \stackrel{\sim}{\cdot}$ |
| $\quad \mid \stackrel{\sim}{\cdot} \stackrel{\sim}{::} \mid \stackrel{\sim}{\cdot} \stackrel{\sim}{=} \mid \stackrel{\sim}{\cdot} \backslash \stackrel{\sim}{=} \mid \stackrel{\sim}{\cdot} = < \stackrel{\sim}{\cdot}$                                                                                                                                                                                               |
| $\langle \text{etiqueta} \rangle ::= \text{unit} \mid \text{true} \mid \text{false} \mid \langle \text{variable} \rangle \mid \langle \text{átomo} \rangle$                                                                                                                                                                                                                                                           |
| $\langle \text{campo} \rangle ::= \text{unit} \mid \text{true} \mid \text{false} \mid \langle \text{variable} \rangle \mid \langle \text{átomo} \rangle \mid \langle \text{ent} \rangle$                                                                                                                                                                                                                              |
| $\langle \text{descriptorClase} \rangle ::= \text{from} \{ \langle \text{expresión} \rangle \}^+ \mid \text{prop} \{ \langle \text{expresión} \rangle \}^+$                                                                                                                                                                                                                                                           |
| $\quad \mid \text{attr} \{ \langle \text{inicAtrb} \rangle \}^+$                                                                                                                                                                                                                                                                                                                                                      |
| $\langle \text{inicAtrb} \rangle ::= ( [ \stackrel{\sim}{!} \stackrel{\sim}{\cdot} ] \langle \text{variable} \rangle \mid \langle \text{átomo} \rangle \mid \text{unit} \mid \text{true} \mid \text{false} )$                                                                                                                                                                                                         |
| $\quad \mid [ \stackrel{\sim}{:} \stackrel{\sim}{\text{expresión}} ]$                                                                                                                                                                                                                                                                                                                                                 |
| $\langle \text{cabezaMétodo} \rangle ::= ( [ \stackrel{\sim}{!} \stackrel{\sim}{\cdot} ] \langle \text{variable} \rangle \mid \langle \text{átomo} \rangle \mid \text{unit} \mid \text{true} \mid \text{false} )$                                                                                                                                                                                                     |
| $\quad \mid [ \stackrel{\sim}{(} \stackrel{\sim}{\cdot} \{ \langle \text{argMétodo} \rangle \} [ \stackrel{\sim}{\dots} \stackrel{\sim}{\cdot} ] \stackrel{\sim}{)} ]$                                                                                                                                                                                                                                                |
| $\quad \mid [ \stackrel{\sim}{=} \stackrel{\sim}{\cdot} \langle \text{variable} \rangle ]$                                                                                                                                                                                                                                                                                                                            |
| $\langle \text{argMétodo} \rangle ::= [ \langle \text{campo} \rangle \stackrel{\sim}{:} \stackrel{\sim}{\cdot} ] ( \langle \text{variable} \rangle \mid \stackrel{\sim}{\_} \mid \stackrel{\sim}{\$} ) [ \stackrel{\sim}{<} \stackrel{\sim}{=} \langle \text{expresión} \rangle ]$                                                                                                                                    |

Tabla C.6: Otros no terminales necesitados para las declaraciones y las expresiones.

( $\langle \text{opBinEval} \rangle$ ), para los registros (etiquetas t campos), y para las clases (descriptores, atributos, métodos, etc.).

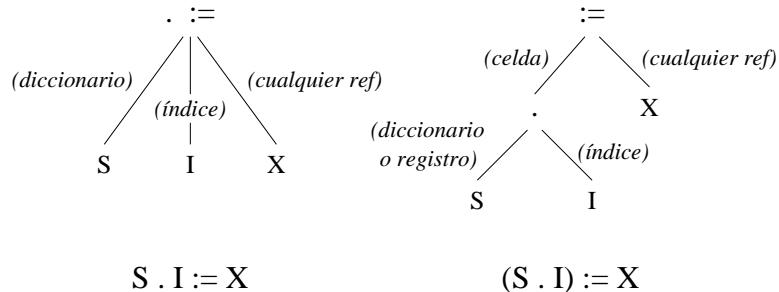
#### C.4. Operadores

En la tabla C.7 se presentan la precedencia y asociatividad de todos los operadores usados en el libro. Todos los operadores son infijos binarios, salvo para tres casos. El signo menos  $\stackrel{\sim}{\sim}$  es un operador prefijo unario. El símbolo numeral  $\stackrel{\sim}{\#}$  es un operador mixfix n-ario. La combinación “ $\stackrel{\sim}{\cdot} \stackrel{\sim}{:} =$ ” es un operador infijo ternario que se explica en la sección siguiente. No hay operadores posfijos. Los operadores aparecen en orden creciente de precedencia, i.e., prioridad de la ligadura. Los operadores más abajo en la tabla son los que se ligan primero. Definimos la asociatividad de cada operador así:

- Izquierda. Para operadores binarios, significa que los operadores repetidos se agrupan hacia la izquierda. Por ejemplo,  $1+2+3$  significa lo mismo que  $((1+2)+3)$

| Operador                                               | Asociatividad |
|--------------------------------------------------------|---------------|
| =                                                      | derecha       |
| <code>:= “. :=”</code>                                 | derecha       |
| <code>orelse</code>                                    | derecha       |
| <code>andthen</code>                                   | derecha       |
| <code>== \= &lt; =&lt; &gt; &gt;= =: \=: =&lt;:</code> | ninguna       |
| <code>::</code>                                        | ninguna       |
| <code> </code>                                         | derecha       |
| <code>#</code>                                         | <i>mixfix</i> |
| <code>+ -</code>                                       | izquierda     |
| <code>* / div mod</code>                               | izquierda     |
| <code>,</code>                                         | derecha       |
| <code>~</code>                                         | izquierda     |
| <code>.</code>                                         | izquierda     |
| <code>@ !!</code>                                      | izquierda     |

Tabla C.7: Operadores con su respectiva precedencia y asociatividad.



$S . I := X$

$(S . I) := X$

Figura C.1: El operador ternario “. :=”.

- Derecha. Para operadores binarios, significa que los operadores repetidos se agrupan hacia la derecha. Por ejemplo,  $a|b|x$  significa lo mismo que  $(a|(b|x))$ .
- *Mixfix*. El operador que se repite es realmente un sólo operador, y todas las expresiones son sus argumentos. Por ejemplo,  $a\#b\#c$  significa lo mismo que  $\#(a b c)$ .
- Ninguna. Para operadores binarios, significa que el operador no se puede repetir. Por ejemplo,  $1<2<3$  es un error.

Los paréntesis se pueden utilizar para anular la precedencia por defecto.

|                 |                     |                 |                    |                    |
|-----------------|---------------------|-----------------|--------------------|--------------------|
| <b>andthen</b>  | <b>default</b>      | <b>false</b>    | <b>lock</b>        | <b>require (*)</b> |
| <b>at</b>       | <b>define</b>       | <b>feat (*)</b> | <b>meth</b>        | <b>return</b>      |
| <b>attr</b>     | <b>dis (*)</b>      | <b>finally</b>  | <b>mod</b>         | <b>self</b>        |
| <b>break</b>    | <b>div</b>          | <b>for</b>      | <b>not (*)</b>     | <b>skip</b>        |
| <b>case</b>     | <b>do</b>           | <b>from</b>     | <b>of</b>          | <b>then</b>        |
| <b>catch</b>    | <b>else</b>         | <b>fun</b>      | <b>or (*)</b>      | <b>thread</b>      |
| <b>choice</b>   | <b>elsecase (*)</b> | <b>functor</b>  | <b>orelse</b>      | <b>true</b>        |
| <b>class</b>    | <b>elseif</b>       | <b>if</b>       | <b>otherwise</b>   | <b>try</b>         |
| <b>collect</b>  | <b>elseof (*)</b>   | <b>import</b>   | <b>prepare (*)</b> | <b>unit</b>        |
| <b>cond (*)</b> | <b>end</b>          | <b>in</b>       | <b>proc</b>        |                    |
| <b>continue</b> | <b>export</b>       | <b>lazy</b>     | <b>prop</b>        |                    |
| <b>declare</b>  | <b>fail</b>         | <b>local</b>    | <b>raise</b>       |                    |

Tabla C.8: Palabras reservadas.

#### C.4.1. Operador ternario

Hay un operador ternario (tres argumentos), “`. :=`”, el cual está diseñado para actualizar arreglos y diccionarios. Este operador tiene la misma precedencia y asociatividad que `:=`, y puede ser usado en una posición de expresión igual que `:=`, donde tiene el efecto de un intercambio. La declaración `S.I:=X` consiste de un operador ternario con argumentos `S`, `I`, y `X`. Esta declaración se utiliza para actualizar diccionarios y arreglos. No se debe confundir con `(S.I):=X`, la cual consiste de dos operadores binarios anidados `.` y `:=`. Esta última declaración se usa para actualizar una celda que se encuentra dentro de un diccionario. ¡Los paréntesis son bastante significativos! En la figura C.1 se muestra la diferencia en términos de sintaxis abstracta, entre `S.I:=X` y `(S.I):=X`. En la figura, *(celda)* significa cualquier celda o atributo de un objeto, y *(diccionario)* significa cualquier diccionario o arreglo.

La distinción es importante porque los diccionarios pueden contener celdas. Para actualizar un diccionario `D`, escribimos `D.I:=X`. Para actualizar una celda dentro de un diccionario que contiene celdas, escribimos `(D.I):=X`. Esto tiene el mismo efecto que `local C=D.I in C:=X end` pero es mucho más conciso. El primer argumento del operador binario `:=` debe ser una celda o un atributo de un objeto.

---

## C.5. Palabras reservadas

En la tabla C.8 se presenta la lista de palabras reservadas del lenguaje en orden alfabetico. Las palabras reservadas maracadas con (\*) existen en OZ pero no se utilizan en el libro. Las palabras reservadas en negrilla pueden ser usadas como

|                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{variable} \rangle ::= (\text{car en mayúscula}) \{ (\text{car alfanumérico}) \}$                                       |
| $  `` `` \{ \langle \text{carVariable} \rangle   \langle \text{seudoCar} \rangle \} `` ``$                                            |
| $\langle \text{átomo} \rangle ::= (\text{car en minúscula}) \{ (\text{car alfanumérico}) \} (\text{salvo ninguna palabra reservada})$ |
| $  `` `` \{ \langle \text{carAtom} \rangle   \langle \text{pseudoCar} \rangle \} `` ``$                                               |
| $\langle \text{cadenaCars} \rangle ::= `` `` \{ \langle \text{carCadena} \rangle   \langle \text{pseudoCar} \rangle \} `` ``$         |
| $\langle \text{caracter} \rangle ::= (\text{cualquier entero en el rango } 0 \dots 255)$                                              |
| $  `` `` \langle \text{carCar} \rangle   `` `` \langle \text{pseudoCar} \rangle$                                                      |

**Tabla C.9:** Sintaxis léxica de variables, átomos, cadenas de caracteres, y caracteres.

|                                                                                                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{carVariable} \rangle ::= (\text{cualquier carácter en línea salvo `` , \, and NUL})$                                                                                                                        |
| $\langle \text{carAtom} \rangle ::= (\text{cualquier carácter en línea salvo `` , \, and NUL})$                                                                                                                            |
| $\langle \text{carCadena} \rangle ::= (\text{cualquier carácter en línea salvo `` , \, and NUL})$                                                                                                                          |
| $\langle \text{carCar} \rangle ::= (\text{cualquier carácter en línea salvo \ and NUL})$                                                                                                                                   |
| $\langle \text{pseudoCar} \rangle ::= `` `` \langle \text{dígitoOctal} \rangle \langle \text{dígitoOctal} \rangle \langle \text{dígitoOctal} \rangle$                                                                      |
| $  `` `` \langle \text{dígitoHex} \rangle \langle \text{dígitoHex} \rangle$                                                                                                                                                |
| $  `` `` \langle \text{a} \rangle   `` `` \langle \text{b} \rangle   `` `` \langle \text{f} \rangle   `` `` \langle \text{n} \rangle   `` `` \langle \text{r} \rangle   `` `` \langle \text{t} \rangle$                    |
| $  `` `` \langle \text{v} \rangle   `` `` \langle \text{\} \rangle   `` `` \langle \text{\` \} \rangle   `` `` \langle \text{\` \` \} \rangle   `` `` \langle \text{\` \` \` \} \rangle   `` `` \langle \text{\&} \rangle$ |

**Cuadro C.10:** Símbolos no terminales necesitados para la sintaxis léxica.

átomos encerrándolas entre comillas sencillas. Por ejemplo, `then` es un átomo, mientras que **then** es una palabra reservada. Las palabras reservadas que no están en negrita se pueden usar directamente como átomos, sin comillas.

## C.6. Sintaxis léxica

Esta sección define la sintaxis léxica de Oz, i.e., cómo se transforma una secuencia de caracteres en una secuencia de lexemas.

### C.6.1. Lexemas

#### *Variables, átomos, cadenas de caracteres, y caracteres*

En la tabla C.9 se define la sintaxis léxica para los identificadores de variables, lo átomos, las cadenas de caracteres, y los caracteres en las cadenas. A diferencia de las anteriores secciones en donde se definían secuencias de lexemas, en ésta sección definimos secuencias de caracteres. Un carácter alfanumérico es una letra

|                                       |                                                                                                                                                                                                                  |
|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\langle \text{ent} \rangle$          | $::= [ \sim ] \langle \text{dígitoNoCero} \rangle \{ \langle \text{dígito} \rangle \}$                                                                                                                           |
|                                       | $  [ \sim ] 0 \{ \langle \text{dígitoOctal} \rangle \}^+$                                                                                                                                                        |
|                                       | $  [ \sim ] ( \sim 0x \sim   \sim 0X \sim ) \{ \langle \text{dígitoHex} \rangle \}^+$                                                                                                                            |
|                                       | $  [ \sim ] ( \sim 0b \sim   \sim 0B \sim ) \{ \langle \text{dígitoBin} \rangle \}^+$                                                                                                                            |
| $\langle \text{flot} \rangle$         | $::= [ \sim ] \{ \langle \text{dígito} \rangle \}^+ \cdot \{ \langle \text{dígito} \rangle \} [ ( \sim e \sim   \sim E \sim ) [ \sim ] \{ \langle \text{dígito} \rangle \}^+ ]$                                  |
| $\langle \text{dígito} \rangle$       | $::= 0   1   2   3   4   5   6   7   8   9$                                                                                                                                                                      |
| $\langle \text{dígitoNoCero} \rangle$ | $::= 1   2   3   4   5   6   7   8   9$                                                                                                                                                                          |
| $\langle \text{dígitoOctal} \rangle$  | $::= 0   1   2   3   4   5   6   7$                                                                                                                                                                              |
| $\langle \text{dígitoHex} \rangle$    | $::= \langle \text{dígito} \rangle   \sim a \sim   \sim b \sim   \sim c \sim   \sim d \sim   \sim e \sim   \sim f \sim$<br>$  \sim A \sim   \sim B \sim   \sim C \sim   \sim D \sim   \sim E \sim   \sim F \sim$ |
| $\langle \text{dígitoBin} \rangle$    | $::= 0   1$                                                                                                                                                                                                      |

**Tabla C.11:** Sintaxis léxica de números enteros y de punto flotante.

(mayúscula o minúscula), un dígito, o un guión de subíndice. Las comillas sencillas se usan para delimitar representaciones de átomos que puedan contener caracteres no alfanuméricos y las comillas invertidas se utilizan, de la misma forma, para identificadores de variables. Note que un átomo no puede tener la misma secuencia de caracteres que una palabra reservada a menos que el átomo esté entre comillas. En la tabla C.10 se definen los símbolos no terminales necesarios para la tabla C.9. “Cualquier carácter en línea” incluye los caracteres de control y los caracteres acentuados. El código del carácter NUL es 0 (cero).

### Números enteros y de punto flotante

En la tabla C.11 se define la sintaxis léxica de los números enteros y de punto flotante. Note el uso de la  $\sim$  (tilde) para el símbolo del operador unario menos.

#### C.6.2. Espacio en blanco y comentarios

Los lexemas se pueden separar por cualquier cantidad de espacios en blanco y por comentarios. El espacio en blanco es cualquiera de los siguientes caracteres: tab (código de carácter 9), línea nueva (código 10), tab vertical (código 11), *form feed* (código 12), *carriage return* (código 13), y espacio (código 32). Un comentario es alguna de las tres posibilidades siguientes:

- Una secuencia de caracteres que comienza con el carácter % (porcentaje) hasta el final de la línea o del archivo (lo que termine primero).
- Una secuencia de caracteres que comienza con /\* y termina con \*/, inclusive. Esta clase de comentarios pueden estar anidados.
- El carácter ? (signo de interrogación). Este carácter está destinado para marcar los argumentos de salida de los procedimientos, como en

```
proc {Max A B ?C} ... end
```

donde C es una salida. Un argumento de salida es un argumento que se liga al interior del procedimiento.

---

## D      Modelo General de Computación

La eliminación de la mayor parte de la complejidad accidental de la programación significa que, lo que queda, es la complejidad intrínseca de la aplicación.

– Adaptación libre de *Security Engineering*, Ross J. Anderson (2001)

Si usted quiere que la gente haga algo de la forma correcta, usted debe lograr que la forma correcta sea fácil.

– Adaptación libre de un dicho tradicional

En este apéndice se reunen todos los conceptos generales introducidos en el libro.<sup>1</sup> Denominamos al modelo de computación resultante, el modelo general de computación. Su semántica se presenta en el capítulo 13 (en CTM). Aunque este modelo es bastante general, tampoco es la palabra final en cuanto a modelos de computación. Es tan sólo una instantánea que captura nuestra comprensión actual de la programación. Futuras investigaciones seguramente lo cambiarán o lo extenderán. En el libro se mencionan el alcance dinámico y el soporte a las transacciones como dos áreas donde se requiere un mayor soporte de parte del modelo.

El modelo general de computación fue diseñado en forma de capas, empezando por un modelo básico sencillo al cual se le agregaron, sucesivamente, conceptos nuevos. Cada vez que notamos una limitación en la expresividad de un modelo de computación, tuvimos la oportunidad de agregar un concepto nuevo. Siempre había una alternativa: dejar el modelo tal cual y hacer programas más complicados, o agregar un concepto nuevo y conservar los programas sencillos. La decisión de agregar o no un concepto, estuvo basada en nuestro juicio de cuán complicado serían el modelo y sus programas, cuando se consideraran al tiempo. La “complejidad” en este sentido cubre tanto la expresividad como la facilidad de razonamiento con la combinación.

Existe un factor muy fuerte de creatividad en este enfoque. Cada concepto trae consigo algo nuevo que no estaba allí antes. A esto lo llamamos el principio de extensión creativa. Algunos conceptos útiles no quedaron en el modelo general. Otros conceptos se añadieron sólo hasta el momento en que fueron reemplazados por conceptos posteriores. Por ejemplo, ese es el caso de la escogencia no-determinística (ver sección 5.8.1), la cual se reemplaza por el concepto de estado explícito. El

---

1. Salvo por los espacios de computación los cuales subyacen al modelo de computación relacional y al modelo de computación basado en restricciones.

modelo general es solo uno entre muchos posibles modelos de expresividad similar. Su juicio en este proceso, puede ser diferente del nuestro. Nos interesaría oír a cualquier lector que haya alcanzado conclusiones significativamente diferentes.

Debido a que los primeros modelos de computación son subconjuntos de los últimos, los últimos se pueden considerar como marcos dentro de los cuales pueden coexistir muchos modelos de computación. En ese sentido, el modelo general de computación es el marco más completo del libro.

---

### D.1. Principio de extensión creativa

Presentamos un ejemplo para explicar y motivar el principio de extensión creativa. Empecemos con el sencillo lenguaje declarativo del capítulo 2. En ese capítulo, agregamos dos conceptos al lenguaje declarativo: funciones y excepciones. Pero, en la manera como se añadió cada concepto, hubo algo que fue fundamentalmente diferente. Las funciones se añadieron como una abstracción lingüística, por medio de la definición de una sintaxis nueva y mostrando cómo traducirla al lenguaje núcleo (ver sección 2.6.2). Las excepciones, por su parte, se añadieron al mismo lenguaje núcleo por medio de operaciones primitivas nuevas y definiendo su semántica (ver sección 2.7). ¿Por qué escogimos hacerlo de esa manera? Podríamos haber agregado funciones al lenguaje núcleo y haber definido las excepciones por medio de una traducción, pero no lo hicimos. Hay una sencilla pero profunda razón para ello: las funciones se pueden definir por medio de una traducción local, no así las excepciones. Una traducción de un concepto es local si ella requiere cambios solamente en las partes de programa que usan el concepto.

A partir del lenguaje núcleo declarativo del capítulo 2, el libro agrega conceptos uno por uno. Por cada concepto, debemos decidir si lo agregamos como abstracción lingüística (sin cambiar el lenguaje núcleo) o lo agregamos al lenguaje núcleo. Una abstracción lingüística es una buena idea si la traducción es local. Extender el lenguaje núcleo es una buena idea si no existe traducción local.

Esta escogencia siempre trae un compromiso. Un criterio es que el esquema global, incluyendo tanto el lenguaje núcleo como el esquema de traducción en el lenguaje núcleo, debería ser lo más sencillo posible. Esto es lo que nosotros denominamos el principio de extensión creativa. En algún grado, la simplicidad es un juicio subjetivo. En este libro hacemos una elección particular sobre qué debería estar en el lenguaje núcleo y qué debería estar por fuera. Seguramente, otras alternativas razonables también son posibles.

Una restricción adicional sobre los lenguajes núcleo del libro es que todos ellos han sido cuidadosamente escogidos como subconjuntos del lenguaje Oz en su totalidad. Esto quiere decir que todos ellos están implementados en el sistema Mozart. Los usuarios pueden verificar que la traducción de un programa al lenguaje núcleo se comporta exactamente en la misma forma que el programa. La única diferencia entre los dos es la eficiencia. Esto es útil, tanto para aprender los lenguajes núcleo, como para depurar los programas. El sistema Mozart implementa ciertas

|                                                                                                                                                          |                             |
|----------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| $\langle d \rangle ::=$                                                                                                                                  |                             |
| <b>skip</b>                                                                                                                                              | Declaración vacía           |
| $\langle d \rangle_1 \langle d \rangle_2$                                                                                                                | Declaración de secuencia    |
| <b>local</b> $\langle x \rangle$ <b>in</b> $\langle d \rangle$ <b>end</b>                                                                                | Creación de variable        |
| $\langle x \rangle_1 = \langle x \rangle_2$                                                                                                              | Ligadura variable-variable  |
| $\langle x \rangle = \langle v \rangle$                                                                                                                  | Creación de valor           |
| { $\langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n$ }                                                                                    | Invocación de procedimiento |
| <b>if</b> $\langle x \rangle$ <b>then</b> $\langle d \rangle_1$ <b>else</b> $\langle d \rangle_2$ <b>end</b>                                             | Condicional                 |
| <b>case</b> $\langle x \rangle$ <b>of</b> $\langle \text{patrón} \rangle$ <b>then</b> $\langle d \rangle_1$ <b>else</b> $\langle d \rangle_2$ <b>end</b> | Reconocimiento de patrones  |
| <b>thread</b> $\langle d \rangle$ <b>end</b>                                                                                                             | Creación de hilo            |
| {WaitNeeded $\langle x \rangle$ }                                                                                                                        | Sincronización by-need      |
| {NewName $\langle x \rangle$ }                                                                                                                           | Creación de nombre          |
| $\langle y \rangle = ! ! \langle x \rangle$                                                                                                              | Vista de sólo lectura       |
| <b>try</b> $\langle d \rangle_1$ <b>catch</b> $\langle x \rangle$ <b>then</b> $\langle d \rangle_2$ <b>end</b>                                           | Contexto de la excepción    |
| <b>raise</b> $\langle x \rangle$ <b>end</b>                                                                                                              | Lanzamiento de excepción    |
| {FailedValue $\langle x \rangle \langle y \rangle$ }                                                                                                     | Valor fallido               |
| {NewCell $\langle x \rangle \langle y \rangle$ }                                                                                                         | Creación de celda           |
| {Exchange $\langle x \rangle \langle y \rangle \langle z \rangle$ }                                                                                      | Intercambio de celda        |
| {IsDet $\langle x \rangle \langle y \rangle$ }                                                                                                           | Prueba de ligadura          |

Tabla D.1: El lenguaje núcleo general.

construcciones más eficientemente que su representación en el lenguaje núcleo. Por ejemplo, las clases y los objetos en Oz están implementados más eficientemente que sus definiciones en el lenguaje núcleo.

---

## D.2. Lenguaje núcleo

En la tabla D.1 se presenta el lenguaje núcleo del modelo general de computación. Por claridad, dividimos la tabla en cinco partes:

- La primera parte es el modelo declarativo descriptivo. Este modelo permite la construcción de estructuras de datos complejas (grafos con raíz, cuyos nodos son registros y valores de tipo procedimiento) pero no permite calcular con ellas.
- Las primera y segunda partes, juntas, forman el modelo declarativo concurrente. Este es el modelo puramente declarativo, más general del libro. Todos los programas escritos en este modelo son declarativos.
- La tercera parte añade seguridad: la capacidad de construir abstracciones de datos seguras y programas con habilidades.
- La cuarta parte agrega excepciones: la capacidad de manejar situaciones excepcionales realizando salidas no locales.
- La quinta parte añade estado explícito, el cual es importante para escribir programas modulares y programas que cambian en el tiempo.

Al tomar todas las partes tenemos el modelo general de computación. Este es el más general del libro. En el capítulo 13 (en CTM) se presenta la semántica de este modelo y de todos sus subconjuntos.

---

## D.3. Conceptos

Ahora recapitulemos la metodología de diseño del modelo general, comenzando con un modelo sencillo de base y explicando brevemente qué expresividad nueva trae consigo cada concepto. Txodos los modelos son Turing-completos, i.e., todos son equivalentes en poder de cálculo a una máquina de Turing. Sin embargo, la Turing-completitud sólo es una pequeña parte del cuento. La facilidad con que se puede escribir un programa o razonar sobre él difiere de manera importante en estos modelos. Un incremento en la expresividad viene, típicamente, con un incremento en la dificultad para razonar sobre los programas.

### D.3.1. Modelos declarativos

#### *Modelo funcional estricto*

El modelo práctico más sencillo es la programación funcional estricta con valores. El modelo se define en la sección 2.8.1. En este modelo no hay variables no-ligadas; cada variable nueva se liga inmediatamente a un valor. Este modelo es cercano al cálculo  $\lambda$ , el cual sólo contiene definición e invocación de procedimientos, y deja por fuera el condicional y el reconocimiento de patrones. El cálculo  $\lambda$  es Turing-

completo, pero es demasiado engorroso para hacer programación práctica.

### ***Modelo declarativo secuencial***

El modelo declarativo secuencial está definido en el capítulo 2. El modelo contiene todos los conceptos de la tabla D.1 hasta la invocación de procedimientos, los condicionales, y el reconocimiento de patrones. El modelo extiende el modelo funcional estricto introduciendo las variables de flujo de datos. Hacer esto es una etapa crítica porque prepara el camino para la concurrencia declarativa. Para ligar las variable de flujos de datos, se utiliza una operación general denominada unificación. Esto significa que el modelo declarativo secuencial hace tanto programación lógica determinística, como programación funcional.

### ***Hilos***

El concepto de hilo se define en la sección 4.1. Los hilos permiten expresar en el modelo actividades que se ejecutan independientemente. Este modelo es aún declarativo pues el resultado de los cálculos no se modifica. Solamente el orden en que se hacen los cálculos es más flexible. Los programas se vuelven más incrementales: la construcción incremental de una entrada lleva a la construcción incremental de la salida. Esta es la primera forma de concurrencia declarativa.

### ***Sincronización by-need***

El concepto de sincronización by-need (la operación `WaitNeeded`) se define en la sección 13.1.13 (en CTM). Agregarlo permite al modelo expresar la ejecución dirigida por la demanda. Las funciones perezosas y la operación `ByNeed` se definen en términos de `WaitNeeded`. Un disparador, tal como se definió en la sección 4.5.1, no es más que el hilo suspendido creado por la ejecución de `ByNeed`. Este modelo es aún declarativo pues el resultado de los cálculos no se modifica. Solamente cambia la cantidad de cálculos realizados para llegar al resultado (puede ser menor). Algunas veces, el modelo dirigido por la demanda puede dar resultados en casos donde el modelo dirigido por los datos se quedaría en un ciclo infinito. Esta es una segunda forma de concurrencia declarativa

### **D.3.2. Seguridad**

#### ***Nombres***

El concepto de nombre se define en la sección 3.7.5. Un nombre es una constante inexpugnable que no existe por fuera de un programa. Un nombre no tiene datos ni operaciones asociadas a él; es una “llave” de primera clase o un “derecho.” Los nombres son la base de técnicas de programación tales como la abstracción de datos desempaquetada (ver sección 6.4) y el control de encapsulación (ver sección 7.3.3).

En el modelo declarativo, las abstracciones de datos seguras se pueden construir sin nombres utilizando valores de tipo procedimiento para ocultar valores. El valor oculto es una referencia externa al procedimiento. Pero los nombres agregan una expresividad adicional, crucial: posibilitan programar con derechos, e.g., separar los datos de las operaciones en una abstracción de datos segura o pasar llaves a los programas para permitir operaciones seguras.

Estrictamente hablando, los nombres no son declarativos, pues invocaciones consecutivas a `NewName` dan resultados diferentes. Es decir, el mismo programa puede devolver dos resultados diferentes si los nombres se utilizan únicamente para identificar el resultado. Pero, si los nombres sólo se usan para hacer cumplir las propiedades de seguridad, entonces el modelo es aún declarativo.

### *Vistas de sólo lectura*

El concepto de vista de sólo lectura se define en la sección 3.7.5. Una vista de sólo lectura es una variable de flujo de datos que se puede leer pero no se puede ligar. La vista de sólo lectura siempre está emparejada con otra variable de flujo de datos que es igual a ella pero que se puede ligar. Las vistas de solo lectura se necesitan para construir abstracciones que exportan variables no-ligadas. La abstracción exporta la vista de sólo lectura, la cual, como no puede ser ligada por fuera de la abstracción, permite a ésta mantener su propiedad invariante.

### **D.3.3. Excepciones**

#### *Manejo de la excepción*

El concepto de excepción se define en la sección 2.7.2. Agregar excepciones permite salir, en un paso, de un número arbitrariamente grande de invocaciones anidadas de procedimientos. Eso permite escribir programas que traten los casos raros de manera correcta, sin complicar el programa en el caso común.

#### *Valores fallidos*

El concepto de valor fallido se define en la sección 4.9.1. Un valor fallido es un tipo especial de valor que encapsula una excepción. Cualquier intento de utilizar el valor o de ligarlo a un valor determinado lanzará una excepción. Mientras que el manejado de la excepción sucede en un solo hilo, los valores fallidos permiten que las excepciones detectadas en un hilo sean pasadas a otros hilos.

Los valores fallidos son útiles en modelos que tienen tanto excepciones como computación by-need. Suponga que un programa realiza una computación by-need para calcular un valor, pero la computación, en lugar de hacer eso, lanza una excepción. ¿Cuál debería ser el valor de la computación? Puede ser un valor fallido. Esto hará que cualquier hilo que necesite ese valor lance una excepción.

#### D.3.4. Estado explícito

##### *Celdas (estado explícito)*

El estado explícito se define en la sección 6.3. Agregar estado le provee a un programa memoria: un procedimiento puede cambiar su comportamiento sobre invocaciones consecutivas. En el modelo declarativo esto no es posible pues todo el conocimiento se encuentra en los argumentos del procedimiento.

El modelo con estado mejora de manera importante la modularidad de los programas comparado con los modelos sin estado. Las posibilidades de cambiar la implementación de un módulo sin cambiar su interfaz se ve incrementada (ver sección 4.8).

##### *Puertos (estado explícito)*

Otra forma de agregar estado explícito es por medio de puertos, los cuales son una especie de canal de comunicación asincrónica. Como se explicó en la sección 7.8, los puertos y las celdas son equivalentes: cada uno puede implementar el otro de manera sencilla. Los puertos son útiles para la programación por paso de mensajes con objetos activos. Las celdas son útiles para la programación de acciones atómicas con estado compartido.

##### *Prueba de ligadura (estado débil)*

La prueba de ligadura `IsDet` nos permite usar variables de flujo de datos como una forma débil de estado explícito. La prueba comprueba si una variable ya está ligada o aún está no-ligada, sin esperar en el caso en que la variable no esté ligada. Para muchas técnicas de programación, el conocimiento del estado de la ligadura de una variable no es importante. Sin embargo, puede ser importante cuando se está programando una ejecución dependiente del tiempo, i.e., cuando se necesita conocer cuál es el estado instantáneo de una ejecución (ver sección 4.8.3).

##### *Programación orientada a objetos*

La programación orientada a objetos se introduce en el capítulo 7. Tiene el mismo lenguaje núcleo que los modelos con estado, y está basada en tres principios: los programas son colecciones de abstracciones de datos que interactúan entre ellas, las abstracciones de datos deberían tener estado por defecto (lo cual es bueno para la modularidad), y las abstracciones de datos deberían utilizar el estilo objeto (APD) por defecto (lo cual estimula el polimorfismo y la herencia).

---

#### D.4. Formas diferentes de estado

Agregar estado explícito es un cambio tan fuerte al modelo que es importante tener formas débiles de estado. En los anteriores modelos introdujimos cuatro formas de estado. Resumamos estas formas en términos de cuántas veces podemos asignar una variable, i.e., cambiar su estado. En orden creciente de fuerza, ellas son:

- No hay asignación, i.e., se hace programación con valores únicamente (ejecución monótona). Esta es la programación funcional como se entiende normalmente. Los programas son completamente determinísticos, i.e., el mismo programa siempre da el mismo resultado.
- Una única asignación, i.e., se hace programación con variables de flujo de datos (ejecución monótona). Esto también es programación funcional, pero más flexible, pues permite la concurrencia declarativa (con ambos tipos de ejecución, perezosa y ansiosa). Los programas son completamente determinísticos, pero el resultado se puede obtener incrementalmente.
- Una única asignación con prueba de ligadura, i.e., se hace programación con variables de flujo de datos y con `IsDet` (ejecución no monótona). Los programas dejan de ser determinísticos.
- Asignación múltiple, i.e., se hace programación con celdas o puertos (ejecución no monótona). Los programas dejan de ser determinísticos. Este es el modelo más expresivo.

Podemos entender estas formas diferentes de estado en términos de una propiedad importante denominada monotonicidad. En cualquier momento, a una variable se le puede asignar un elemento de un conjunto  $S$  de valores. La asignación es monótona si a medida que progresá la ejecución, se pueden eliminar valores de  $S$  pero no se pueden agregar. Por ejemplo, ligar una variable de flujo de datos  $x$  a un valor, reduce  $S$  de todos los valores posibles a un solo valor. Una función  $f$  es monótona si  $S_1 \subset S_2 \implies f(S_1) \subset f(S_2)$ . Por ejemplo, `IsDet` no es monótona pues `{IsDet x}` devuelve `true` cuando  $x$  está ligada y `false` cuando no lo está, pero `{true}` no es un subconjunto de `{false}`. La ejecución de un programa es monótona si todas sus operaciones son monótonas. La monotonicidad es la que hace posible la concurrencia declarativa.

---

## D.5. Otros conceptos

### D.5.1. ¿Qué sigue?

El modelo general de computación del libro es tan solo una instantánea de un proceso en curso. En el futuro, se continuará el descubrimiento de conceptos nuevos utilizando el principio de extensión creativa. ¿Cuáles serán estos conceptos nuevos? No podemos asegurarlos, pues anticipar un descubrimiento es equivalente a hacer el descubrimiento! Pero existen pistas sobre unos cuantos conceptos. Tres conceptos, sobre los que estamos bastante seguros, pero que no conocemos su forma final

son: alcance dinámico, membranas, y soporte a las transacciones. Con el alcance dinámico, el comportamiento de un componente depende de su contexto. Con las membranas, se puede definir un cerramiento alrededor de una parte de un programa, de manera que “algo sucede” cuando cualquier referencia lo cruza. Esto sucederá aún a las referencias ocultas dentro de una estructura de datos o de un valor de tipo procedimiento. Con el soporte a las transacciones, la ejecución de un componente se puede cancelar si no se puede completar exitosamente. De acuerdo al principio de extensión creativa, todos estos conceptos deberían agregarse al modelo de computación.

#### D.5.2. Conceptos de dominio específico

Este libro presenta muchos conceptos generales que son útiles para todo tipo de programas. Además de esto, cada dominio de aplicación tiene su propio conjunto de conceptos que son útiles solamente en ese dominio. Estos conceptos adicionales complementan los conceptos generales. Por ejemplo, podemos citar la inteligencia artificial [127, 148], el diseño de algoritmos [39], los patrones de diseño orientado a objetos [54], la programación multiagentes [183], las bases de datos [41], y el análisis numérico [140].

---

### D.6. Diseño de lenguajes por capas

El modelo general de computación tiene un diseño por capas. Cada capa ofrece su propio y especial compromiso de expresividad y facilidad de razonamiento. El programador puede escoger la capa que mejor se adapte a cada parte de su programa. A partir de la evidencia presentada en el libro, es claro que esta estructura por capas es benéfica para un lenguaje de programación de propósito general. Esto facilita que el programador diga directamente lo que desea, sin codificaciones engorrosas.

El diseño por capas del modelo general de computación se puede encontrar en algún grado en muchos lenguajes. Los lenguajes de programación orientada a objetos tales como Smalltalk, Eiffel, y Java tienen dos capas: un núcleo orientado a objetos y una segunda capa proveyendo concurrencia con estado compartido [56, 113, 10]. El lenguaje funcional Erlang tiene dos capas: un núcleo funcional ansioso y una segunda capa proveyendo concurrencia por paso de mensajes entre objetos activos [9] (ver también la sección 5.7). Los objetos activos se definen dentro del núcleo funcional. El lenguaje lógico Prolog tiene tres capas: un núcleo lógico que es un probador de teoremas sencillo, una segunda capa que modifica la operación del probador de teoremas, y una tercera capa proveyendo estado explícito [163] (ver también la sección 9.7 (en CTM)). El lenguaje funcional Concurrent ML tiene tres capas: un núcleo funcional ansioso, una segunda capa proveyendo estado explícito, y una tercera capa proveyendo concurrencia [145]. El lenguaje multiparadigma Oz tiene muchas capas, lo que nos llevó a utilizarlo como el lenguaje de base del

*Modelo General de Computación*

libro [161].

---

## Bibliografía

- [1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.
- [2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*, 2nd edition. MIT Press, Cambridge, MA, 1996.
- [3] Edward G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, Englewood Cliffs, NJ, 1994.
- [4] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.
- [5] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, Menlo Park, CA, 1991.
- [6] Joe Armstrong. Higher-order processes in Erlang, January 1997. Unpublished talk.
- [7] Joe Armstrong. Concurrency oriented programming in Erlang, November 2002. Invited talk, Lightweight Languages Workshop 2002.
- [8] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology (KTH), Kista, Sweden, November 2003.
- [9] Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [10] Ken Arnold and James Gosling. *The Java Programming Language*, 2nd edition. Addison-Wesley, 1998.
- [11] Arvind and R. E. Thomas. I-Structures: An efficient data type for functional languages. Technical Report 210, MIT, Laboratory for Computer Science, Cambridge, MA, 1980.
- [12] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
- [13] John Backus. The history of FORTRAN I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, August 1978.
- [14] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.
- [15] Holger Bär, Markus Bauer, Oliver Ciupke, Serge Demeyer, Stéphane Ducasse, Michele Lanza, Radu Marinescu, Robb Nebbe, Oscar Nierstrasz, Michael Przybilski, Tamar Richner, Matthias Rieger, Claudio Riva, Anne-Marie Sassen, Benedikt Schulz, Patrick Steyaert, Sander Tichelaar, and Joachim Weisbrod. *The FAMOOS Object-Oriented Reengineering Handbook*. October 1999. Deliverable, ESPRIT project FAMOOS.
- [16] Victor R. Basili and Albert J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, 1(4):390–396, December 1975.
- [17] Kent Beck. *Test-driven development: by example*. Addison-Wesley, 2003.
- [18] Joseph Bergin and Russel Winder. Understanding object-oriented programming, 2000. Available at <http://csis.pace.edu/~bergin/>.
- [19] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.
- [20] Richard Bird. *Introduction to Functional Programming using Haskell*, 2nd edition. Prentice Hall, Englewood Cliffs, NJ, 1998.

## Bibliografía

- [21] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [22] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.
- [23] Per Brinch Hansen. *Operating System Principles*. Prentice Hall, Englewood Cliffs, NJ, 1973.
- [24] Per Brinch Hansen. Java’s insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.
- [25] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.
- [26] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, 1995.
- [27] Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.
- [28] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- [29] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
- [30] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.
- [31] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d’applications avec Objective Caml*. O’Reilly & Associates, Paris, 2000.
- [32] Keith L. Clark. PARLOG: The language and its applications. In A. J. Nijman, J. W. de Bakker, and P. C. Treleaven, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE), volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 30–53, Eindhoven, the Netherlands, June 1987. Springer-Verlag.
- [33] Keith L. Clark and Frank McCabe. The control facilities of IC-Prolog. In D. Michie, editor, *Expert Systems in the Micro-Electronic Age*, pages 122–149. Edinburgh University Press, Edinburgh, 1979.
- [34] Keith L. Clark, Frank G. McCabe, and Steve Gregory. IC-PROLOG — language features. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.
- [35] Arthur C. Clarke. *Profiles of the Future*, revised edition. Pan Books, 1973.
- [36] William Clinger and Jonathan Rees. The revised<sup>4</sup> report on the algorithmic language Scheme. *LISP Pointers*, 4(3):1–55, July-September 1991.
- [37] Alain Colmerauer. The birth of Prolog. *ACM SIGPLAN Notices*, 28(3):37–52, March 1993. Originally appeared in History of Programming Languages Conference (HOPL-II), 1993.
- [38] William R. Cook. Object-oriented programming versus abstract data types. In *REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 173 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, 1990.
- [39] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, Cambridge, MA, 1990.
- [40] Charles Darwin. *On the Origin of Species by means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. Harvard University Press (originally John Murray, London, 1859).
- [41] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1994.
- [42] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, 1984.
- [43] Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz, and Ralph E. Johnson. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [44] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3), March 1966.
- [45] Edsger W. Dijkstra. *A Primer of Algol 60 Programming*. Academic Press, 1962.
- [46] Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

- [47] Denys Duchier. Loop support. Technical report, Mozart Consortium, 2003. Available at <http://www.mozart-oz.org/>.
- [48] Denys Duchier, Leif Kornstaedt, and Christian Schulte. The Oz base environment. Technical report, Mozart Consortium, 2003. Available at <http://www.mozart-oz.org/>.
- [49] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 207–216, Albuquerque, NM, June 1993.
- [50] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman, 2000.
- [51] Michael J. French. *Invention and Evolution: Design in Nature and Engineering*. Cambridge University Press, Cambridge, UK, 1988.
- [52] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 1992.
- [53] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. KLIC: A portable implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94)*, pages 66–79, Tokyo, December 1994. Institute for New Generation Computer Technology (ICOT).
- [54] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [55] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
- [56] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1983.
- [57] James Edward Gordon. *The Science of Structures and Materials*. Scientific American Library, 1988.
- [58] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at <http://www.javasoft.com>.
- [59] Paul Graham. *On Lisp*. Prentice Hall, Englewood Cliffs, NJ, 1993. Available for download from the author.
- [60] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.
- [61] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, Cambridge, MA, 1994.
- [62] Robert H. Halstead, Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
- [63] Richard Hamming. *The Art of Doing SCIENCE and Engineering: Learning to Learn*. Gordon and Breach Science Publishers, Amsterdam, the Netherlands, 1997.
- [64] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, May 1998.
- [65] Martin Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, November 1997.
- [66] Martin Henz. Objects in Oz. Doctoral dissertation, Saarland University, Saarbrücken, Germany, May 1997.
- [67] Martin Henz and Leif Kornstaedt. The Oz notation. Technical report, Mozart Consortium, 2003. Available at <http://www.mozart-oz.org/>.
- [68] Martin Henz, Gert Smolka, and Jörg Würtz. Oz—a programming language for multi-agent systems. In Ruzena Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence*, pages 404–409, Chambéry, France, August 1993. Morgan Kaufmann.
- [69] Martin Henz, Gert Smolka, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In Pascal Van Hentenryck and Vijay Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 29–48, Cambridge, MA, 1995. MIT Press.
- [70] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.
- [71] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism

## Bibliografía

- for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, August 1973.
- [72] Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
  - [73] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
  - [74] Paul Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3):359–411, September 1989.
  - [75] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell, version 98, June 2000. Available at <http://www.haskell.org/tutorial/>.
  - [76] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.
  - [77] Robert A. Iannucci. *Parallel Machines: Parallel Machine Languages. The Emergence of Hybrid Dataflow Computer Architectures*. Kluwer, Dordrecht, the Netherlands, 1990.
  - [78] Daniel H. H. Ingalls. Design principles behind Smalltalk. *Byte*, 6(8):286–298, 1981.
  - [79] Intelligent Systems Laboratory, Swedish Institute of Computer Science. SICStus Prolog user's manual, April 2003. Available at <http://www.sics.se/sicstus/>.
  - [80] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, New York, 1991.
  - [81] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *International Symposium on Logic Programming*, pages 167–183, October 1991.
  - [82] K. Jensen and N. Wirth. *Pascal: User Manual and Report*, 2nd edition. Springer-Verlag, 1978.
  - [83] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, New York, 1996.
  - [84] Andreas Kågedal, Peter Van Roy, and Bruno Dumant. Logical State Threads 0.1, January 1997. Available at <http://www.info.ucl.ac.be/people/PVR/implementation.html>.
  - [85] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
  - [86] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.
  - [87] Alan C. Kay. The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3):69–95, March 1993. Originally appeared in History of Programming Languages Conference (HOPL-II), 1993.
  - [88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (ANSI C)*, 2nd edition. Prentice Hall, Englewood Cliffs, NJ, 1988.
  - [89] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
  - [90] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, MA.
  - [91] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA, 1973.
  - [92] Donald E. Knuth. Structured programming with **go to** statements. *Computing Surveys*, 6(4):261–301, December 1974.
  - [93] Leif Kornstaedt. Gump—a front-end generator for Oz. Technical report, Mozart Consortium, 2003. Available at <http://www.mozart-oz.org/>.
  - [94] S. Rao Kosaraju. Analysis of structured programs. *Journal of Computer and System Sciences*, 9(3):232–255, December 1974.
  - [95] Robert A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.
  - [96] Robert A. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.
  - [97] James F. Kurose and Keith W. Ross. *Computer Networking: a Top-down Approach Featuring the Internet*. Addison-Wesley, 2001.

- [98] Raymond Kurzweil. *The Singularity is Near*. Viking/Penguin Books, 2003. Expected publication date.
- [99] Leslie Lamport. *LATEX: A Document Preparation System*, 2nd edition. Addison-Wesley, 1994.
- [100] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *IEEE Computer*, 36(6):47–56, June 2003.
- [101] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems, IRIA*, October 1978. Reprinted in *Operating Systems Review*, 13(2), April 1979, pp. 3–19.
- [102] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.
- [103] Doug Lea. *Concurrent Programming in Java*, 2nd edition. Addison-Wesley, 2000.
- [104] Nancy Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.
- [105] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, MA, 1984. Available for download from the author.
- [106] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *1st Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA 86)*, pages 214–223, September 1986. Also in *Object-Oriented Computing*, Gerald Peterson, editor, IEEE Computer Society Press, 1987.
- [107] Barbara Liskov. A history of CLU, April 1992. Technical Report, Laboratory for Computer Science, MIT.
- [108] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.
- [109] Bruce J. MacLennan. *Principles of Programming Languages*, 2nd edition. WB Saunders, Philadelphia, 1987.
- [110] Zohar Manna. *The Mathematical Theory of Computation*. McGraw-Hill, New York, 1974.
- [111] John McCarthy. *LISP 1.5 Programmer’s Manual*. MIT Press, Cambridge, MA, 1962.
- [112] Scott McCloud. *Understanding Comics: The Invisible Art*. Kitchen Sink Press, 1993.
- [113] Bertrand Meyer. *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, Englewood Cliffs, NJ, 2000.
- [114] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.
- [115] Mark Miller, Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, and Norm Hardy. E: Open source distributed capabilities, 2001. Available at <http://www.erights.org>.
- [116] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Draft available at <http://zesty.ca/capmyths>, 2003.
- [117] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Financial Cryptography 2000*, Anguilla, British West Indies, February 2000.
- [118] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
- [119] J. Paul Morrison. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, New York, 1994.
- [120] Mozart Consortium. The Mozart Programming System, version 1.3.0, 2003. Available at <http://www.mozart-oz.org/>.
- [121] Peter Naur, John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John L. McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.
- [122] Rishiyur S. Nikhil. ID language reference manual version 90.1. Technical Report Memo 284-2, MIT, Computation Structures Group, Cambridge, MA, July 1994.
- [123] Rishiyur S. Nikhil. An overview of the parallel language Id—a foundation for pH, a parallel dialect of Haskell. Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1994.
- [124] Rishiyur S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann,

*Bibliografía*

- 2001.
- [125] Donald A. Norman. *The Design of Everyday Things*. Basic Books, New York, 1988.
  - [126] Theodore Norvell. Monads for the working Haskell programmer—a short tutorial. Available at <http://www.haskell.org/>.
  - [127] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.
  - [128] K. Nygaard and O. J. Dahl. *The Development of the SIMULA Languages*, pages 439–493. Academic Press, 1981.
  - [129] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.
  - [130] Richard A. O’Keefe. *The Craft of Prolog*. MIT Press, 1990.
  - [131] Andreas Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, Cambridge, MA, 1993.
  - [132] David Lorge Parnas. Teaching programming as engineering. In *9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. Reprinted in *Software Fundamentals*, Addison-Wesley, 2001.
  - [133] David Lorge Parnas. *Software Fundamentals*. Addison-Wesley, 2001.
  - [134] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*, 2nd edition. Morgan Kaufmann, 1996.
  - [135] Simon L. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001. Presented at the 2000 Marktoberdorf Summer School, Marktoberdorf, Germany.
  - [136] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003. Also published as the January 2003 special issue of *Journal of Functional Programming*.
  - [137] Simon L. Peyton Jones, Andrew Gordon, and Sigrbjorn Finne. Concurrent Haskell. In *Principles of Programming Languages (POPL)*, pages 295–308, St. Petersburg Beach, FL, January 1996. ACM Press.
  - [138] Shari Lawrence Pfleeger. *Software Engineering: The Production of Quality Software*, 2nd edition. Macmillan, 1991.
  - [139] R. J. Pooley. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publishers, 1987.
  - [140] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK, 1986.
  - [141] Roger S. Pressman. *Software Engineering*, 6th edition. Addison-Wesley, 2000.
  - [142] Mahmoud Rafea, Fredrik Holmgren, Konstantin Popov, Seif Haridi, Stelios Lelis, Petros Kavassalis, and Jakka Sairamesh. Application architecture of the Internet simulation model: Web Word of Mouth (WoM). In *IASTED International Conference on Modelling and Simulation MS2002*, May 2002.
  - [143] Eric Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O’Reilly & Associates, January 2001.
  - [144] Juris Reinfelds. Teaching of programming with a programmer’s theory of programming. In *Informatics Curricula, Teaching Methods, and Best Practice (ICTEM 2002, IFIP Working Group 3.2 Working Conference)*, Boston, 2002. Kluwer Academic Publishers.
  - [145] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, UK, 1999.
  - [146] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In David Gries, editor, *Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3*, pages 309–317. Springer-Verlag, 1978. Originally published in *New Directions in Algorithmic Languages*, INRIA Rocquencourt, 1975.
  - [147] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language*

- Reference Manual.* Addison-Wesley, 1999.
- [148] Stuart Russell and Peter Norvig. *Artificial Intelligence: A modern approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.
  - [149] Oliver Sacks. *The Man Who Mistook His Wife for a Hat And Other Clinical Tales*. Harper & Row, Publishers, 1987.
  - [150] Jakka Sairamesh, Petros Kavassalis, Manolis Marazakis, Christos Nikolaos, and Seif Haridi. Information cities over the Internet: Taxonomy, principles and architecture. In *Digital Communities 2002*, November 2001.
  - [151] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Principles of Programming Languages (POPL)*, pages 333–352, Orlando, FL, January 1991.
  - [152] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley & Sons, New York, 2000.
  - [153] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, New York, 1996.
  - [154] Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
  - [155] Christian Schulte and Gert Smolka. Encapsulated search for higher-order concurrent constraint programming. In *1994 International Symposium on Logic Programming*, pages 505–520. MIT Press, November 1994.
  - [156] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology (ICOT), Cambridge, MA, January 1983.
  - [157] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1-2. MIT Press, Cambridge, MA, 1987.
  - [158] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.
  - [159] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples*. McGraw-Hill, New York, 1982.
  - [160] Gert Smolka. The definition of Kernel Oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, Berlin, 1995.
  - [161] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
  - [162] Guy L. Steele, Jr. *Common Lisp: The Language*, 2nd edition. Digital Press, Bedford, MA, 1990.
  - [163] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. Series in Logic Programming. MIT Press, Cambridge, MA, 1986.
  - [164] Marc Stiegler. *The E Language in a Walnut*. 2000. Draft available at <http://www.erights.org>.
  - [165] Bjarne Stroustrup. A history of C++. *ACM SIGPLAN Notices*, 28(3):271–297, March 1993. Originally appeared in History of Programming Languages Conference (HOPL-II), 1993.
  - [166] Bjarne Stroustrup. *The C++ Programming Language*, 3rd edition. Addison-Wesley, 1997.
  - [167] Giancarlo Succi and Michele Marchesi. *Extreme Programming Examined*. Addison-Wesley, 2001.
  - [168] Sun Microsystems. *The Java Series*. Sun Microsystems, Mountain View, CA, 1996. Available at <http://www.javasoft.com>.
  - [169] Sun Microsystems. *The Remote Method Invocation Specification*, 1997. Available at <http://www.javasoft.com>.
  - [170] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, 1999.
  - [171] Gerard Tel. *An Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, UK, 1994.
  - [172] Evan Tick. The deevolution of concurrent logic programming. *Journal of Logic Programming*

*Bibliografía*

- ming, 23(2):89–123, May 1995.
- [173] Kasunori Ueda. Guarded Horn Clauses. In Eiti Wada, editor, *Proceedings of the 4th Conference on Logic Programming*, volume 221 of *Lecture Notes in Computer Science*, pages 168–179, Tokyo, July 1985. Springer-Verlag.
  - [174] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, Englewood Cliffs, NJ, 1998.
  - [175] Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Computer Science Division, University of California at Berkeley, December 1990. Technical Report UCB/CSD 90/600.
  - [176] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: The Oz experience. *Theory and Practice of Logic Programming*, 3(6):715–763, November 2003.
  - [177] Peter Van Roy and Alvin Despain. High-performance logic programming with the Aquarius Prolog compiler. *IEEE Computer*, pages 54–68, January 1992.
  - [178] Peter Van Roy and Seif Haridi. Teaching programming broadly and deeply: The kernel language approach. In *Informatics Curricula, Teaching Methods, and Best Practice (ICTEM 2002, IFIP Working Group 3.2 Working Conference)*, Boston, 2002. Kluwer Academic Publishers.
  - [179] Peter Van Roy and Seif Haridi. Teaching programming with the kernel language approach. In *Workshop on Functional and Declarative Programming in Education (FD-PE02), at Principles, Logics, and Implementations of High-Level Programming Languages (PLI2002)*. University of Kiel, Germany, October 2002.
  - [180] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.
  - [181] Duncan J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, Princeton, NJ, 1999.
  - [182] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.
  - [183] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, 1999.
  - [184] Claes Wikström. Distributed programming in Erlang. In the *1st International Symposium on Parallel Symbolic Computation (PASCO 94)*, pages 412–421, Singapore, September 1994. World Scientific.
  - [185] Herbert S. Wilf. *generatingfunctionology*. Academic Press, 1994.
  - [186] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, Cambridge, MA, 1993.
  - [187] Noel Winstanley. What the hell are Monads?, 1999. Available at <http://www.haskell.org>.
  - [188] David Wood. Use of objects and agents at Symbian, September 2000. Talk given at the Newcastle Seminar on the Teaching of Computing Science, Newcastle, UK.
  - [189] Matthias Zenger and Martin Odersky. Implementing extensible compilers. In *1st International Workshop on Multiparadigm Programming with Object-Oriented Languages*, pages 61–80, Budapest, Hungary, June 2001. John von Neumann Institute for Computing (NIC). Workshop held as part of ECOOP 2001.

---

## Índice alfabético

- ! (símbolo de escape para variables), 108
- != comparación (igualdad), 60
- =< comparación (menor o igual), 60
- ? (argumento de salida), 709
- ? (argumento de salida), 62
- @ (acceso al estado), 542
- @ operación (acceso al estado), 538
- # constructor (tuplas), 155
- constructor # (tuplas), 692, 699
- % (comentario para terminar una línea), 709
- % operación (módulo) (en Java), 673
- & operador (caracter en línea), 686
- álgebra, xxiii, 125
- árbol, 165
  - balanceado, 166
  - binario
    - ordenado, 166
  - de sintaxis, 34
  - dibujo, 174
  - expresión Haskell, 340
  - finito, 165
  - mezclador de flujos, 436
  - profundidad, 167
  - profundidad de un nodo, 171
  - recorrido, 170
  - sintaxis, 178
  - ternario, 166
- árbol de sintaxis abstracta, 180
- átomo, 56, 690
  - definición de alcance, 554
- < comparación (estrictamente menor), 60
- comparación < (estrictamente menor), 697
- operador <= (argumento opcional del

*Índice alfabético*

- método), 545
- > comparación (estrictamente mayor), 60
- >= comparación (mayor o igual), 60
- \ (barra diagonal invertida), 687
- notación \ (en Haskell), 342
- \= comparación (desigualdad), 60
- \~ (virgulilla) signo menos, 686
- \` (backquote), 54, 687
- \` (comilla invertida), 709
- operación (comilla sencilla inversa en Lisp)', 42
- aparato* (en GUI), 234
- | constructor (par lista), 57
- abajo hacia arriba
  - desarrollo de *software*, 493
- Abelson, Harold, 45
- aborto en cascada, 659
- abrazo mortal
  - invocación de vuelta, 392
  - simulación de la lógica digital, 297
- abstracción, *xvii*, véase abstracción de control, véase abstracción de datos
- buzón, 428
- canal de objetos flujo, 612
- candado, 636, 652
- ciclo, 199, 284
- ciclo de vida, 43
- clase, 537
- colección, 475
- componente de *software*, 242
- conector, 357
- detección de terminación, 419
- espacio de tuplas, 640
- iteración, 135
- jerarquía en programación orientada a objetos, 594
- jerarquía especializada, *xvii*
- Linda (espacio de tuplas), 640
- lingüística, véase abstracción lingüística
- lista por comprehensión, 332
- monitor, 645
- objeto activo, 607
- objeto flujo, 292
- objeto puerto, 384
- procedimental, 195
- protector, 357
- recolector, 208, 357, 475, 528
- replicador, 357
- serialización (en transacciones), 655
- transacción, 654

- abstracción lingüística  
  **compuerta** (compuerta lógica), 298
- abstracción de datos  
  programación orientada a objetos (POO), 533
- acceso  
  operación de celda, 453  
  operación sobre celda, 18
- acción atómica, 632–671  
  cuando usarla, 629  
  enfoques de concurrencia, 626  
  razonamiento, 633
- ActiveX, 506
- acumulador, 153–155  
  abstracción de ciclo, 204, 284  
  analizador sintáctico, 177  
  estado declarativo, 446  
  ciclo **for**, 208  
  Prolog, 154  
  recorrido de árbol, 211  
  recorrido en amplitud, 172  
  recorrido en profundidad, 171  
  relación con la operación de plegado, 204  
  relación con listas de diferencias, 156
- adaptación de impedancias  
  concurrencia, 629
- adaptación de impedancias, 356  
  ejemplo manejador de eventos, 618
- operación **Adjoin**, 176, 477, 478, 580, 599, 693
- operación **AdjoinAt**, 415, 477, 478, 693
- administración de la memoria, 78–85, 524–526  
  C, 197  
  Pascal, 197  
  recolección de basura, 82
- administrador de almacenamiento, 357
- administrador de transacciones, 666
- adquisición y liberación de candados en dos fases, 659
- adversario, 228
- afirmación, 485
- agente, 384  
  componente concurrente, 397  
  enfoque por paso de mensajes, 629
- aislamiento  
  transacción, 656
- operación **Alarm**, 334, 431
- alcance, 61, 552  
  atributo, 556  
  definido por el usuario, 554  
  dinámico, véase alcance, dinámico, 64  
  estático, véase alcance, léxico, véase léxico  
  léxico, 62, 63, 69, 554  
  encapsulación, 450  
  ocultamiento, 483, 528, 540, 598  
  ocultar, 463  
  oculto, 243  
  lexico, 588
- alcance léxico, véase alcance, léxico
- alcance público, 553
- alcance privado, 553, 554  
  en el sentido de C++ y Java, 554  
  en el sentido de Smalltalk y Oz, 553
- alcance protegido, 554  
  en el sentido de C++, 555  
  en el sentido de Java, 618
- algoritmo  
  árbol, 166  
  **mergesort**, 152
- administrador de transacciones, 661
- algoritmo de Dekker para exclusión mutua, 622
- clausura transitiva, 507
- clausura transitiva paralela, 512
- cola, 160

*Índice alfabético*

- persistente, 327  
prioridad, 670  
composición concurrente, 304  
compresión, 194  
detección de terminación, 419  
dibujar árboles, 174  
el problema de Flavio Josefo, 609  
elevador, 413  
Floyd-Warshall, 514  
frecuencia de palabras, 218  
generación de números aleatorios, 516  
generador lineal congruente, 518  
método de Newton para raíces cuadradas, 131  
mergesort, 183  
mergesort (genérico), 199  
planificación de hilos, 265  
problema de Hamming, 322  
quicksort, 255, 572  
recolección de basura, 82  
recolección de basura “copying dual-space”, 85  
recorrido en amplitud, 171  
recorrido en profundidad, 170  
simulación, 520  
triángulo de Pascal, 11  
unificación, 109  
Alimentar Mozart con un texto (en OPI), 682  
almacén  
    asignación única, 45–52, 65  
    de valores, 46  
    disparadores, 310  
    mutable (para celdas), 455  
    mutable (para puertos), 383  
    sólo lectura, 227  
almacén de asignación única, 45–52, 65  
    importancia, 46  
almacén de valores, 46  
alto orden  
    programación, 194–214  
ambiente, 48, 66  
calculando con, 67  
contextual, 71  
extensión, 67  
interfaz interactiva, 94  
restricción, 67  
análisis de flexibilidad, 341  
análisis de rigidez, 318  
analizador léxico, 34, 178  
analizador sintáctico, 34, 177–182  
    herramienta **gump**, 42  
Anderson, Ross J., 711  
Andrews, Gregory, 635  
operador **andthen**, 90  
anotación  
    estado explícito, 27  
APD (abstracción procedimental de datos), 459  
operación **Aplanar**, 157, 254, 325  
aplicación  
    autónoma, 244  
    Java, 606  
    despliegue de video, 352  
    ejemplo ping-pong, 336  
aplicación autónoma, 244  
    **declare** no permitido, 94  
    Java, 606  
aplicativo  
    orden de reducción, 363  
operación **Append**, 696  
Apple Corporation, xxvii  
apuntador  
    bloque de memoria, 525  
    dependencia, 502  
    recurso, 524  
archivo, 231  
archivo de texto, 231  
arco, xviii  
aridad, 693  
    cabeza del método, 544  
    semántica de **case**, 73  
    consumo de memoria de un registro, 191  
    desempeño del registro, 479  
aritmética, 59

- aritmética de precisión infinita, 4  
aritmética entera de precisión arbitraria, 688  
operación **Arity**, 693  
Armstrong, Joe, 635  
Arnold, Ken, 585  
arreglo, 476  
    compromisos de uso, 479  
    extensible, 480  
arriba hacia abajo  
    desarrollo de *software*, 493  
ASCII (American Standard Code for Information Interchange), 241, 501  
aseguramiento estricto en dos fases, 659  
asignación  
    única, 45, 718  
    a una variable iterativa, 207  
    de memoria, 81  
    en el sitio, 345  
    Java, 603  
    múltiple, 718  
    monotona, 718  
    operación de celda, 453  
    operación sobre celda, 18  
    semántica axiomática, 487  
asignación destructiva, véase estado  
asociación, 579  
ATM (Asynchronous Transfer Mode), 425  
atomicidad, 23, 654  
operación **AtomToString**, 691  
atributo  
    final (en Java), 593  
    initialización, 543  
    objeto, 541  
attribute  
    final (en Java), 601  
AXD 301 ATM switch, 425  
axioma  
    programación lógica, 444  
    semántica axiomática, 481  
azúcar sintáctico, 43, 86–91  
creación dinámica de registros, 182  
diagrama de transición de estados, 405  
declaración **local**, 43  
  
búsqueda  
    binaria, 166  
    lineal, 217  
    programación por restricciones, 302  
    uso intenso, xxi  
Backus, John, 35  
Bal, Henri E., 367  
banda elástica, 276  
base de datos  
    conurrencia con estado compartido, 629  
    en memoria, 189  
    Mnesia (en Erlang), 425  
    transacción, 654  
Baum, L. Frank, 1  
Bernstein, Philip A., 655  
Biblioteca, 252  
    Biblioteca Estándar de Mozart, 236, 248  
    módulos Base y System en Mozart, 252  
    MOGUL (Mozart Global User Library), 244  
bloque  
    archivo, 321  
    memoria, 80  
    Smalltalk, 591  
booleano, 56  
Brand, Per, xxvi  
**break** (en ciclo **for**), 209  
declaración **break**, 530  
Brinch Hansen, Per, 646  
Brooks, Jr., Frederick P., 505  
Browser, véase Mozart Programming System  
buzón, 498  
Erlang, 424

*Índice alfabético*

- implementación, 428
- operación `ByNeed`, 310
- byte (en Java), 604
- bytestring, 698
- cálculo, xvi
  - análisis, 187
  - $\pi$ , xvi, 44, 58
  - $\lambda$ , xvi, 44, 105, 363, 377, 714
- cálculo  $\pi$ , xvi, 44
- código fuente, 244
  - alcance textual, 69
  - conjunto de functors, 313
  - entrada al preprocesador, 349
  - inexistente, 536
  - interactivo, 682
  - millones de líneas, xvi, 39, 425
  - nombre de variable, 48
  - reingeniería, 569
  - un millón de líneas, 500
- código objeto, 244
- cable, 398
  - de muchos tiros, 398
  - de un tiro, 398, 404
- cadena, 57
  - cadena de caracteres, 697
  - cadena virtual, 232
  - cadena virtual de caracteres, 699
- calculadora, 1
- campo (en registro), 692
- canal, 285
  - asincrónico, 382, 422
  - de muchos tiros, 398
  - de un tiro, 398, 404
  - interfaz del componente, 498
  - puerto, 382
  - sincrónico, 675
  - variable de flujo de datos, 364
- candado, 632, 636
  - declaración `lock`, 23
  - escritura, 677
  - hilo reentrant, 645
  - implementación, 643
  - introducción, 23
- Java, 672
- lectura, 677
- obtener-liberar, 652
- sencillo, 644
- transacción, 657
- capasa
  - diseño de lenguajes, 719
- caracter, 686
  - alfanumérico, 53, 708
  - Java, 604
- declaración `case`, 72
- cláusula `catch`(en cláuse `try`), 102
- causalidad
  - conurrencia, 262
  - eventos de paso de mensajes, 388
- operación `Ceil`, 689
- celda (estado explícito), 447, 453–456, 717
- celda “cons” (par lista), 695
- celda cons (par lista), 57
- chunk, 224
- Church-Rosser
  - teorema, 363
- Churchill, Winston, 491
- ciclo de vida
  - abstracción, 43
  - bloque de memoria, 80
- ciclo `for`, 489
- ciclo `while`, 488
- ciencia, xiv, xviii
- ciencias de la computación, xxi
- circuitos biestables (lógica digital), 297
- cláusula
  - declaración `case`, 89
  - Erlang, 425
- cláusula de Horn, xxi
- cláusula `define`, 244
- Clarke, Arthur C.
  - segunda ley, 114
  - tercera ley, 345
- clase, 452, 536, 540, 595
  - abstracta, 570, 595
  - concreta, 570, 572, 595

- control de encapsulación, 552
- de mezcla, 617
- definición completa, 537
- definición incremental, 547
- delegación, 557
- diagrama, 575, 581
- ejemplo manejador de eventos, 614
- final, 536, 542, 591
- genérica, 571
- herencia, 547
- implementación, 597
- interna (en Java), 588, 602
- introducción, 19
- limitaciones comunes, 588
- metaclase, 595
- miembro, 541
- objeto activo, 608
- parametrizada, 572
- patche, 569
- programación de alto orden, 573
- propiedad de sustitución, 565, 568, 570
- reenvío, 557
- reflexión del estado del objeto, 564
- técnicas de programación, 565
- visión de estructura, 565
- visión de tipo, 566
- clase `Thread` (en Java), 672
- clausura, véase valor de tipo procedimiento
- clausura de alcance léxico, véase valor de tipo procedimiento
- clausura transitiva, 506–515
  - algoritmo con estado, 511
  - algoritmo de Floyd-Warshall, 514
  - algoritmo declarativo, 509, 512
- clonar
  - arreglo, 477
  - diccionario, 478
- coherencia
  - ocultamiento, 503
- cola, 160
- concurrente, 417, 636
- efímera amortizada, 161
- efímera peor caso, 162
- eliminación no bloqueante, 651
- persistente amortizada, 327
- persistente en tiempo constante
  - en el peor caso, 329
- recorrido en amplitud, 171
- cola de prioridad, 661, 670
- colocación de alias, 457
- COM (Component Object Model), 506
- combinaciones, 4
- combinador, 305
- comentario (en Oz), 709
- compactación, 82
- comparación, 60
- compartiendo
  - hilo, 414
- compilación
  - autónoma, 251
  - ejemplo ping-pong, 336
  - separada, 115
  - unidad, 244, 451, 682
  - unidad de, 496
- compilador, 178
  - extensible, 591
  - operación de unificación, 108
  - smart-aleck, 345
- Compiler Panel tool, véase Mozart Programming System
- complejidad
  - amortizada, 161, 191, 477
  - asintótica, 183
  - espacio, 189
  - introducción, 11–12
  - método del banquero, 192
  - método del físico, 192
  - notación  $\mathcal{O}$ , 183
  - peor caso, 183, 189
  - tiempo, 12, 183
- completitud
  - desarrollo de software, 241
  - Turing, 714

*Índice alfabético*

- componente, 124, 201, 243  
abstracción, 500  
diagrama, 386  
encapsula una decisión de diseño, 501  
evitar dependencias, 501  
functor, 451  
grafo, 504  
implementación, 243  
interfaz, 243  
lectura adicional, 505  
módulo, 451  
papel en el futuro, 504  
software, 243  
composición concurrente, 304, 422  
composicionalidad, 450, 677  
clase, 546  
manejo de excepciones, 98  
comprobación  
programación en pequeño, 241  
programas con estado, 445  
programas declarativos, 123, 445  
tipamiento dinámico, 115  
comprobación de no unificación, 112–114  
comprobación de unificación, 108, 112–114  
compromiso  
alcance dinámico vs. estático, 63  
colecciones indexadas, 476  
colocación de procedimientos auxiliares, 134  
comprobación de programas declarativo vs. con estado, 445  
descomposición funcional vs. descomposición por tipos, 589  
diseño composicional vs. no composicional, 504  
diseño del lenguaje núcleo, 712  
ejecución ansiosa vs. perezosa, 361  
estado explícito vs. estado implícito, 346, 448  
expresividad vs. eficiencia en ejecución, 129  
flexibilidad vs. estado explícito, 364, 377  
herencia sencilla vs. herencia múltiple, 579  
herencia vs. composición de componentes, 536  
lenguaje de especificación vs. lenguaje de programación, 129  
nombres vs. átomos, 556  
objetos vs. TADs, 468  
planificación optimista y pesimista, 658  
tiempo de compilación vs. eficiencia en la ejecución, 499  
tipamiento dinámico vs. estático, 114  
utilización de memoria vs. velocidad de ejecución, 194  
visión de tipo vs. visión de estructura, 567  
compuerta de retraso (lógica digital), 297  
compuerta lógica, 294  
compuesta  
declaración (en Java), 603  
entidad, 475  
estructura de datos, 20, 56  
figura gráfica, 583  
computación, véase informática, 66  
bloqueada, 265  
iterativa, 130  
recursiva, 137  
segura, 228  
terminada, 265  
computación en tiempo real  
dura, 191, 278  
estricto, 334  
no estricto, 334  
recolección de basura, 83  
computación tiempo real  
dura, 280  
computador Cray-1, 192  
computador Macintosh, xxvii

- computador personal, 4, 277, 280, 318, 334  
bajo costo, 80  
de bajo costo, 192  
comunicación asincrónica, 364  
Erlang, 423  
interacción de componentes, 498  
objeto activo, 608  
paso de mensajes, 379  
puerto, 382  
recepción, 365  
red lenta, 631  
remisión, 365  
reporte de errores, 395  
RMI, 391  
semántica distribuida, 420  
comunicación muchos-a-uno, 385  
comunicación sincrónica, 364  
CSP, 675  
dependencia, 424  
detección de fallos, 438  
interacción de componentes, 498  
recepción, 365  
remisión, 365  
reporte de errores, 395  
variante para objeto activo, 612  
declaración **conc**, 305  
conciencia de la red, 424  
concurrencia, 353  
cola, 636  
cooperativa, 280  
declarativa, 257, 267  
determinación del orden, 301  
diferencia con paralelismo, 353  
enfoques prácticos, 625  
enseñanza, xxii  
Erlang, 423  
flujo de datos, 16  
importancia para la programación, xv  
intercalación, 22  
interfaz interactiva, 97  
introducción, 16  
Java, 671  
lecturas adicionales, 634  
no-determinismo, 21  
reglas generales, 628  
seguridad  
monitor, 647  
transacción, 657  
concurrencia competitiva, 280  
concurrencia con estado compartido, véase acción atómica, véase candado, véase monitor, véase transacción  
concurrencia declarativa, véase concurrencia, declarativa  
concurrencia por paso de mensajes, véase objeto, activo, véase objeto, puerto  
condición  
Haskell, 340  
lista por comprehensión, 332  
condición de carrera, 21  
condición de carrera, 258  
condición global  
árbol binario ordenado, 170  
alcanzabilidad, 81  
conector, 357  
confinamiento  
falla en el modelo declarativo, 271  
confluencia, 363  
conjunto por comprehensión, 332  
cons  
celda, 5  
consistencia (en transacción), 655  
construcción de puentes, xv, xvi  
constructor, 604  
consumidor, 283  
contenedor  
futuro (en Multilisp), 369  
contexto, 98  
continuación, 414  
procedimiento, 394  
registro, 393  
continue (en ciclo **for**), 209  
contrato, 567

*Índice alfabético*

- control de concurrencia, 617  
control de la concurrencia<sup>1</sup>, 657  
control del flujo, 107, 288  
    ejecución perezosa, 288  
convolución (simbólica), 255  
copiar  
    objeto, 564  
CORBA (Common Object Request Broker Architecture), 391, 506  
corrección, 448, 567  
    introducción, 9–10  
cortafuegos, 357  
corto circuito  
    composición concurrente, 304  
corto-circuito  
    el problema de Flavio Josefo, 611  
corutina, 302, 498  
    relación con la pereza, 313, 316, 626  
operación `CrearObjActExc`, 613  
operación `CrearObjActivo`, 608  
CSP (Communicating Sequential Processes), 675  
cuantificador, 482, 486  
currículo (informática), xxii  
currificación, xxi, 213–214, 255  
    Haskell, 342
- Danvy, Olivier, 255  
Darwin, Charles, 493  
Dawkins, Richard, 449  
DCOM (Distributed Component Object Model), 506  
declaración  
    **case**, 72  
    **catch** (cláusula en **try**), 102  
    **conc**, 305  
    **declare**, 2, 94  
    **finally** (cláusula en **try**), 103  
    **for**, 206  
    **fun**, 92  
    **functor**, 246  
    **gate**, 299
- thread**, 265  
**if**, 72  
**local**, 61, 68  
**lock**, 636  
**proc**, 70  
**raise**, 101  
**skip**, 68  
**try**, 101  
break, 530  
**lock**, 23  
composición secuencial, 68  
compuesta, 130  
compuesta (en Java), 603  
creación de valores, 69  
interactiva, 94  
invocación de procedimiento, 72  
lenguaje núcleo declarativo, 53  
ligadura variable-variable, 68  
    que se puede suspender, 71  
declarativa, 123, 444  
declaración **declare**, 2, 94  
    convención sintáctica del libro, xxix, 94  
del medio hacia afuera  
    desarrollo de *software*, 493  
operación **Delay**, 334  
operación **delay** (en Multilisp), 369  
delegación, 560–562  
    método **otherwise**, 545  
Dell Corporation, xxvii, 220, 514  
dependencia  
    componente declarativo, 344  
    componente, 247  
    componente declarativo, 355  
    conurrencia para determinación  
        del orden, 301  
    contexto gramatical, 36  
    eliminación de lo secuencial, 421  
    flujo de datos, 368  
    herencia, 452  
depuración, 562  
    componente declarativo, 344  
    inspección de los atributos de los  
        objetos, 546

- referencia suelta, 82, 198  
suspensión errónea, 52, 97  
derecho, véase nombre  
desarrollo de *software*, 239, 491  
    componentes concurrentes, 397  
    composicional, 494  
    en grande, 491  
    en pequeño, 239  
    evolucionario, 493  
    importancia de los nombres, 554  
    incremental, 493  
    interfaz interactiva, 94  
    iterativo, 493  
    iterativo e incremental, 493  
    marco, 537  
    orientado por las pruebas, 494  
    programación extrema, 494  
    refinamiento paso a paso, 509,  
        660  
desarrollo de *software* de abajo hacia  
    arriba, 493  
desarrollo de *software* de arriba a aba-  
    jo, 9  
desarrollo de *software* de arriba hacia  
    abajo, 493  
desarrollo de *software* del medio hacia  
    afuera, 493  
desarrollo orientado por las pruebas,  
    494  
descomposición del tipo, 569  
descomposición funcional, 149, 589  
descomposición por tipos, 589  
desempeño  
    anotación, 27  
    clausura transitiva, 514  
    computador personal, 192  
    conurrencia competitiva, 281  
    simulación de boca en boca, 531  
    diccionario, 220  
    lenguaje perezoso, 318  
    medición, 183  
    memorización, 346  
    monitor, 651  
    Mozart Programming System,  
        416  
Mozart Programming System,  
    220  
papel de la optimización, 194,  
    292  
papel del paralelismo, 261, 353  
precio de la concurrencia, 372  
programación declarativa, 344  
rol de la optimización, 333  
supercomputador Cray-1, 192  
desreferenciación, 49  
detección tardía del error (en tiempo  
    de ejecución), 548  
deterioro del software, 502  
determinismo (programación declara-  
    tiva), 123  
devolución  
    de memoria, 81  
DHCP (Dynamic Host Connection  
    Protocol), 227  
diagrama de componentes, 386  
diagrama de transición de estados,  
    388, 403–404  
componente ascensor, 407  
componente controlador del as-  
    ensor, 405  
componente piso, 407  
diseño de componentes, 400  
    transacción, 662  
dibujos animados, 527  
diccionario, 215, 477  
    autónomo, 247  
    compromisos de uso, 479  
    con estado, 218  
    declarativo, 215  
    declarativo seguro, 228  
    eficiencia, 220  
    estructura interna, 220  
    implementación basada en árbo-  
        les, 218  
    implementación basada en listas,  
        217  
    implementación de espacio de tu-  
        plas, 643

*Índice alfabético*

- relación con flujo, 480  
relación con registro, 478  
simulación de boca en boca, 521  
dictionario  
    implementación frecuencias de palabras, 516  
Dijkstra, Edsger Wybe, 481  
dinámica  
    creación de registro, 598  
    creación de registros, 182  
    ligadura, 552  
dinámico  
    enlace, 313  
    tipamiento, 114–116  
discriminante, 196  
diseño de lenguaje  
    propiedades del objeto, 591  
diseño de lenguajes  
    ciclo de vida de una abstracción, 43  
    declarativo, 361  
    edad de oro, 444  
    ejecución perezosa, 361  
    por capas, 719  
diseño de programas, *véase* desarrollo de *software*  
diseño de *software*, *véase* diseño de lenguajes, *véase* metodología de diseño  
diseño natural, 504  
diseño no composicional, 504  
diseño por contrato, 567  
disparador, 309–313, 715  
    activación, 309  
    by-need, 627  
    implícito, 310  
    programado, 212, 288, 310  
distribución de Gauss, 520, 521  
distribución exponencial, 519  
distribución normal, 520  
distribución uniforme,, 518  
Distribution Panel tool, *véase* Mozart Programming System  
**div** operación (división entera) , 59  
operación **div** (división entera), 688  
dividir y conquistar, 152  
Duchier, Denys, xxvi, 419  
duración, 655  
EBNF (Extended Backus-Naur Form), 34  
Eco, Umberto, 97  
ecuación cuadrática, 196  
ecuaciones de recurrencia, 184  
efecto de borde, 450  
efecto lateral  
    declarativo, 316  
Einstein, Albert, 306  
ejecución ansiosa  
    flujo productor/consumidor, 283  
    modelo declarativo, 107  
    relación con sincronización, 367  
    rigidez, 363  
ejecución by-need, 309–313  
    Multilisp, 369  
ejecución dirigida por la demanda,  
    *véase* ejecución perezosa  
ejecución dirigida por la oferta, *véase*  
    ejecución ansiosa  
ejecución dirigida por los datos, *véase*  
    ejecución ansiosa  
ejecución perezosa, 306  
    control del flujo, 288  
    incremental, 324  
    memoria intermedia limitada, 289  
    monolítica, 326, 375  
    problema de Hamming, 322  
    programación de alto orden, 212  
    relación con sincronización, 367  
    requiere finalización, 526  
ejercicio avanzado, xxiv  
elefante, xi  
elemento neutro, 205  
elevación  
    booleanos a flujos, 299  
elevador, *véase* ascensor  
Emacs (editor de texto), 681

- embebimiento, 201  
empaquetamiento, 459  
encapsulación, 19, 450  
    abstracción de datos, 459  
Encyclopaedia Britannica (11th edition), 533  
Ende, Michael, 379  
enfoque de desarrollo incremental, 493  
enfoque de desarrollo iterativo, 493  
enfoque de desarrollo iterativo e incremental (DII), 493  
enfoque del lenguaje núcleo, xvi, 39–45  
    escogencia de formalismo, xix  
enlace, 232, 244, 246, 251  
    componente, 246  
    detección de fallos Erlang, 424  
    dinámico, 244, 313  
    estático, 244  
enlace simbólico, 502  
enlazar, 497, 683  
    componente, 501  
entero  
    aritmética, 688  
    binario, 686  
    hexadecimal, 686  
    número, 685  
    octal, 686  
    precisión arbitraria, 4, 688  
Enterprise Java Beans, 504  
envío  
    latencia, 290  
envolver y desenvolver, 224  
equivalencia lógica, 268  
Ericsson (Telefonaktiebolaget LM Ericsson), 358, 423  
Erlang, 423–432  
error, 98  
    suspensión errónea, 52  
    absoluto, 132  
    “arity mismatch”, 545  
    ciclo infinito, 490  
    comunicación asincrónica, 395  
confinamiento, 98, 221  
declaración de variable, xxix  
declaración `if`, 72  
detección en tiempo de compilación, 115  
detección en tiempo de ejecución, 114, 548  
dominio, 104  
“excepción no capturada”, 101  
falla en la unificación, 104  
inanición, 303  
invocación de procedimiento, 72  
ligadura incompatible, 47  
memoria insuficiente, 82  
programación concurrente, 22  
recuperación de memoria, 81  
referencia suelta, 81  
relativo, 132  
sistema Mozart en tiempo de ejecución, 682  
suspensión errónea, 97  
suspensipón errónea, 490  
tipo errado, 54  
tipo incorrecto, 104, 490, 491  
“variable not introduced”, 702  
“variable usada antes de ligarla”, 51  
escándalo Therac-25, 22  
escalabilidad  
    compilación, 500  
    desarrollo de programas, 115  
espacio blanco, véase espacio en blanco  
espacio de tuplas, 640–643  
espacio en blanco (en Oz), 709  
espacio insuficiente, véase memoria insuficiente  
especificación, 448  
    componente, 504  
estándar de punto flotante de IEEE, 686  
estática  
    ligadura, 552  
estático

*Índice alfabético*

- tipamiento, 55, 114–116
- estado
- administración de la memoria, 84
  - celda (variable mutable), 453
  - declarativo, 446
  - ejecución perezosa, 526
  - explícito, 17
  - filtración, 153
  - habilidad revocable, 474
  - implícito, 446
  - interacción con la invocación por nombre, 529
  - lenguaje flexible, 364
  - lenguaje perezoso, 364
  - propiedad de modularidad, 346
  - puerto (canal de comunicación), 382
  - razonando con, 481
  - razonar con, 41
  - transformación, 147
- estado de la ejecución, 66
- estado explícito, 447
- estado explícito, véase estado estricto ... , véase ansioso ...
- estructura
- compilador, 178
  - composicional, 504
  - diferencias, 156
  - distribución, 281
  - efecto de la concurrencia, 277
  - gramática, 33
  - jerárquica, 494
  - no composicional, 504
  - programa, 240, 242
- estructura de datos
- árbol, 165
  - activa, 83
  - almacén, 81
  - alto orden, 201
  - cíclica, 109
  - clase, 540
  - cola, 160
  - compuesta, 20, 56
  - diccionario, 215
- efímera, 161
- estructura de diferencias, 156
- externa, 84
- grafo, 507
- infinitas, 12
- larga vida, 85
- lista, 57, 143
- lista de diferencias, 156
- parcial, 49
- persistente, 327
- pila, 214
- protegida, 223, 461
- recursiva, 143
- registro, 56
- tamaño, 189
- tupla, 56
- estructura de datos efímera, 161, 327
- estructura de diferencias, 156
- estructura-I, 369, 513
- estructural
- igualdad, 458
- estructuras cíclicas, 112
- etiqueta (identificación de registro), 20, 56
- Euclides, 1
- evaluación ansiosa
- introducción, 12
- evaluación flexible, 363
- Haskell, 341
- evaluación perezosa, 107
- análisis de flexibilidad, 341
  - análisis de rigidez, 318
  - corutina, 626
  - explícita, 201
  - Haskell, 341
  - introducción, 12
  - planificación, 377
  - relación con evaluación flexible, 363
  - relación con la invocación por necesidad, 530
  - relación con paso de parámetros por necesidad, 473
- evento

- manejador, 613
- evolución
  - Darwiniana, 493, 505
  - desarrollo de *software*, 493
  - Internet, 451
- excepción, 97–105, 591, 716
  - error, 104
  - failure, 104
  - no capturada, 101
  - system, 104
- excepción del sistema, 104
- operación Exchange, 455
- Explorer tool, véase Mozart Programming System
- cláusula **export** (en functor), 243
- expresión, 88
  - operador **andthen**, 90
  - notación  $\lambda$ , 105, 342
  - mal formada, 100
  - marca de anidamiento (\$), 91
  - operación básica, 59
  - orden de reducción normal, 362
  - operador **orelse**, 90
  - receive** (Erlang), 428
  - valor básico, 53
  - valor de tipo procedimiento, 70
- extensión (ambiente), 67
- fórmula de Stirling para factorial, 673
- operación FailedValue, 360
- falla, 449, 656
  - aislamiento, 656
  - concurrencia declarativa, 270
  - excepción, 104
  - transacción, 655
  - unificación, 111
  - valor fallido, 360
- falla parcial, 357
- false**, 691
- FCFS (first-come, first-served), 402
- FGCS (Fifth Generation Computer System), Japanese project, 434
- FIFO (first-in, first-out), 255, 370, 402, 417, 498
- Erlang, 424
- puerto, 382
- operación Filter, 210, 696
- finalización, 84, 525
  - ejecución perezosa, 526
- cláusula **finally**(en **try**), 103
- finalmente será cierta (en lógica temporal), 658
- firma (de un procedimiento), 142
- Flavio Josefo
  - problema, 608–612
- flip-flop, 297
- operación FloatToInt, 688
- operación Floor, 689
- Floyd, Robert W., 481
- flujo
  - compromisos de uso, 479
  - determinístico, 283
  - Java, 603
  - mezcla, 432
  - productor/consumidor, 283
- flujo de datos, 65
  - banda elástica, 276
  - canal, 364
  - clausura transitiva paralela, 512
  - ejecución perezosa, 314
  - ejemplos, 273
  - error, 97
  - estructura-I, 369
  - introducción, 16
  - modelo declarativo, 17
  - variable, 45, 51, 52
- fold operation
  - operación FoldL, 696
- declaración **for**, 206
- operación ForAll, 696
- fracaso
  - proyecto de *software*, 568
- Francia, 449
- French, Michael, 621
- fresco
  - nombre, 224

*Índice alfabético*

- frontera de protección, 222  
fuente (productor), 285  
declaración **fun**, 92  
función, 92  
    generadora, 187  
    incremental, 326  
    introducción, 2  
    monótona, 718  
    monolítica, 326  
    parcialmente aplicada, 213  
    transición de estados, 385  
función de Fibonacci, 275  
función factorial, 2–25, 137–140, 253  
función generadora, 187  
función incremental, 324  
función potencial, 192  
functor, 243, 496  
    principal, 244  
    utilización interactiva, 682  
declaración **functor**, 246  
operación **future** (en Multilisp), 369  
futuro, 369
- Gamma, Erich, 582  
declaración **gate**, 299  
Gelernter, David, 640  
generación de números aleatorios, 516  
generador de código, 178  
generalización, 619  
genericidad, 198–200  
    estática y dinámica, 574  
    programación orientada a objetos, 571  
Glynn, Kevin, 339  
Goldberg, Mayer, 255  
Goodman, Nathan, 655  
Gosling, James, 585  
grafo  
    camino, 506  
    componente, 504  
    denso, 504  
    expresión Haskell, 340  
    herencia, 547  
    implementación, 507  
    jerárquico, 504  
    no local, 504  
grafo de invocación, 349  
grafo de objeto, 604  
grafo dirigido, 506  
gramática, 33–39  
    ambigua, 37, 182  
    cláusula definida (DCG), 154  
    EBNF (Extended Backus-Naur Form), 34  
    eliminación de ambigüedad, 37  
    independiente del contexto, 35  
    símbolo no terminal, 35, 179  
    símbolo terminal, 35, 179  
    sensitiva al contexto, 36  
Gramática de Cláusulas Definidas (DCG), 154  
Gray, Jim, 635, 655  
guarda  
    CSP, 676  
    Erlang, 425  
    programación lógica concurrente, 435  
    expresión **receive** (en Erlang), 428, 440  
guardián, 525  
GUI (Interfaz Gráfica de Usuario)  
    programación basada en componentes, 504  
GUI (Interfaz gráfica de usuario)  
    componentes Swing, xxi  
    entrada/salida de texto, 234  
    QTk, 234  
    uso en aplicación, 248  
    uso interactivo **use**, 236
- habilidad, 229  
    declarativa, 231  
    revocable, 474, 528  
Hadzilacos, Vassos, 655  
Hamming  
    problema de, 322, 376  
Hamming, Richard, 322  
operación **HasFeature**, 693

- Haskell, 339–344  
Helm, Richard, 582  
herencia, 20, 452, 460, 535  
    argumentos, 177  
    dirigida y acíclica, 548  
    evitando conflictos de métodos, 557  
    factorización, 536  
    generalización, 619  
    grafo, 547  
    hacia arriba, 619  
    historia que sirve de escarmiento, 568  
    implementación, 599  
    Java, 602  
    ligadura estática y dinámica, 550  
    múltiple, 548, 574  
        reglas prácticas, 580  
    manejador de eventos, 615  
    patrones de diseño, 582  
    prioridad de hilos, 279  
    problema de la implementación  
        compartida, 581  
    sencilla, 548, 578  
    simple, 548  
    visión de estructura, 565  
    visión de tipo, 565  
Hewitt, Carl, 379  
hilo, 715  
    ejecutable, 263  
    interfaz interactiva, 97  
    introducción, 16  
    Java, 672  
    listo, 263  
    modelo declarativo, 257  
    pendiente, 437  
    prioridad, 278  
    propiedad de monotonicidad, 263  
    sincronización, 366  
    suspendido, 264  
declaración **thread**, 265  
historia  
    conurrencia declarativa, 370  
    desarrollo incremental, 493  
lenguaje TAD, 459  
lenguajes de programación, 444  
Oz, xxv  
peligros de la concurrencia, 22  
programación orientada a objetos, 533  
tecnología de computador, 193  
Hoare, Charles Antony Richard, 255, 481, 646  
Holanda, 685  
Horn  
    cláusula, xxi  
HTML (Hypertext Markup Language), 127  
Hudak, Paul, 105, 345  
IBM Corporation, 44  
IDE (Ambiente de desarrollo interactivo), xxix  
IDE (ambiente de desarrollo interactivo), 681  
identificador, 2, 48  
    ocurrencia libre, 62, 69  
    ocurrencia ligada, 69  
    símbolo de escape, 555  
IEEE  
    estándar de punto fijo, 592  
declaración **if**, 72  
igualdad  
    estructura, 113  
igualdad de lexemas, 458  
igualdad estructural, 113, 458  
imparcialidad  
    mezclador de flujos, 435  
    planificación de hilos, 264, 265, 277, 366  
imperativa, 444  
implementación, 448  
cláusula **import** (en functor), 243  
inanición, 303  
    implementación del conjunto de espera, 651  
indecidable  
    problema, 230

*Índice alfabético*

- independencia  
  componentes, 399, 498  
  conceptos, xvii  
  concurrencia, 16, 21, 97, 257  
  diseño composicional, 504  
  herencia múltiple, 580  
  ins y atn (en cola), 418  
  modularidad, 350  
  polimorfismo, 469  
  principio, 499  
  procesos Erlang, 424  
  programación abierta, 115  
  programación declarativa, 123  
  razonamiento con afirmaciones  
    invariantes, 481  
  recolección de basura, 82  
independencia de terceras partes, 368  
independencia del orden, 52  
  unificación, 121, 255  
India, xi  
inducción matemática, 10  
infinito (punto flotante), 688  
informática, *xxi*  
  éxito de la programación orientada a objetos, 533  
  currículo, *xxii*  
  utilidad de los modelos de computación, *xii*  
ingeniería de la computación, *xxi*  
ingeniería de software  
  concurrencia, 257  
ingeniería de *software*, 491  
  currículo en informática, *xxii*  
  lectura adicional, 505  
instanciación, 200–201, 450  
instantánea (del estado), 478  
intérprete, 45  
  metacircular, 45  
integer  
  virgulilla ~ como signo menos, 686  
interfaz, 458  
  concepto general, 243  
  herencia, 536  
Java, 602  
  **Runnabl** (en Java), 672  
interfaz interactiva, 94  
interfaz **Runnable** (en Java), 672  
Internet, 227, 379, 388  
  simulación del uso del Web, 520  
interoperabilidad, 116  
interpretación  
  enfoque para definir la semántica, 45  
interpretador  
  Erlang original, 425  
operación **IntToFloat**, 688  
invariante, 148, 419, 450, 481  
invocación por ..., véase paso de parámetros  
IP (Internet Protocol), 227  
operación **IsAtom**, 691  
operación **IsChar**, 689  
operación **IsDet**, 351, 352, 365, 432, 717  
operación **IsLock**, 636  
ISO 8859-1  
  codificación de caracteres, 686, 690  
función **IsProcedure**, 60  
operación **IsRecord**, 693  
operación **IsTuple**, 694  
Janson, Sverker, xxvi  
Japón, 434  
Java, 601–606, 671–673  
  semántica, 646  
JavaBeans, 506  
Jefferson, Thomas, 10  
Johnson, Ralph, 582  
Kahn, Gilles, 370  
Knuth, Donald Ervin, 187, 516  
Kowalski, Robert A., 444  
Kurzweil, Raymond, 193  
lápiz, xvii  
léxica  
  sintaxis (de Oz), 708

- lógica  
    programación, 444  
    temporal, 658
- lógica digital, 293  
    problema de satisfactibilidad, 193
- lógica secuencial, 296
- lógica temporal, 658
- línea de control, 302
- operación Label, 693
- lambda  
    cálculo  $\lambda$ , xvi, 44, 105, 363, 377, 714
- LAN (local area network), 388
- language  
    Haskell, 361  
    Prolog, 361, 425
- Lao-tzu, 306
- lapso de tiempo, 278–280  
    duración, 279
- latencia, 290  
    tolerancia, 368
- LATEX 2 $\varepsilon$ : sistema de composición tipográfica, 502
- Latin-1, 501
- Lea, Doug, 634
- operación Length, 696
- lenguaje  
    Prolog, 5  
    Scheme, xx  
    AKL, xxvi  
    Absys, 444  
    Ada, 472, 675  
    Algol, 444, 472, 533  
        concurrente declarativo, 370  
    Alice, 116  
    C++, 46, 51, 81, 198, 367, 486, 530, 533, 549, 554–556, 583, 588, 594, 601  
    C-Linda, 640  
    CLOS (Common Lisp Object System), 563  
    CLU, 459  
    CSP (Communicating Sequential Processes), 675
- Clean, 344
- Cobol, 593
- Common Lisp, 64, 209
- Concurrent Haskell, xx
- Concurrent ML, xx, 719
- Concurrent Prolog, 434
- C, 81, 197
- Eiffel, 565, 567, 719
- Erlang, xii, 82, 107, 358, 379, 423–432, 469, 498, 594, 613, 635, 719
- E, 229
- FCP (Flat Concurrent Prolog), 434
- FP, 361
- Flat GHC, 434
- Fortran, 444
- GHC (Guarded Horn Clauses), 434
- Haskell, xii, xv, 46, 82, 107, 128, 151, 213, 300, 307, 315, 339–344, 363, 367, 370, 376, 500, 594
- IC-Prolog, 434
- Id, 369
- Java, xii, xv, 44, 46, 51, 82, 198, 367, 391, 468, 470, 486, 506, 530, 533, 549, 554–556, 583, 588, 591–594, 601–606, 618, 634, 646, 671–673, 719  
    monitor, 646
- Leda, xvii
- Linda extensión, 640
- Lisp, xii, 5, 42, 64, 82, 143, 444, 593, 695
- ML, véase Standard ML, Concurrent ML, Objective Caml
- Mercury, xii, 128, 344
- Miranda, 307, 376
- Multilisp, 369
- Objective Caml, 592
- Oz, xix, xxvi, 344, 553, 594, 712, 719

## Índice alfabético

- Parlog*, 434  
*Pascal*, 177, 197, 470  
*Prolog*, xii, xv, xxi, 32, 51, 82, 128, 154, 157, 300, 315, 367, 435, 444, 594, 719  
    SICStus, 209  
*Scheme*, xii, xv, xvii, 31, 46, 64, 106, 107, 315, 593  
*Simula*, 444, 533  
*Smalltalk*, xii, 46, 82, 367, 533, 553, 555, 556, 562, 588, 591, 593, 719  
*Standard ML*, xii, xx, 31, 46, 106, 107, 128, 151, 213, 315, 344, 362  
    *Visual Basic*, 504  
*pH (parallel Haskell)*, 369  
alto orden, 195  
coordinación, 640  
ensamblador, 229, 344, 601  
especificación, 129  
extensión de *Linda*, 676  
flexible, 363  
formal, 35  
multiparadigma, xvii  
natural, 33, 41  
práctico, 33  
primer orden, 194  
seguro, 229  
simbólico, 58, 593  
lenguaje de especificación, 129  
lenguaje de modelamiento unificado (UML), 533  
lenguaje ensamblador, 444  
lenguaje formal, 35  
lenguaje núcleo, véase modelo de computación  
lenguaje natural, 41  
lenguaje orientado a objetos puro, 591  
lexema  
    igualdad, 458  
ley  
    de Moore, 192  
segunda de Clarke, 114  
TAD pila, 214  
lgica  
    combinatoria, 294  
libro  
    *Concurrent Programming in Erlang*, 635  
    *Concurrent Programming in Java*, 634  
    *Concurrent Programming: Principles and Practice*, 635  
    *Object-Oriented Software Construction*, 535  
    *Software Fundamentals*, 505  
    *Software por componentes: Más allá de la programación orientada a objetos*, 505  
    *Structure and Interpretation of Computer Programs*, xviii  
    *The Mythical Man-Month*, 505  
    *Transaction Processing: Concepts and Techniques*, 635  
LIFO (last-in, first-out), 535  
ligadura  
    dinámica, 552  
    estática, 552  
    variable-variable, 50  
*Linda ()*, 640  
Linux, xxvii, 514, 543  
Liskov, Barbara, 459  
lista, 57, 141, 695  
    anidada, 149  
    aplanar, 157  
    circular, 696  
    completa, 57, 695  
    compromisos de uso, 479  
    de diferencias  
        ventaja, 160  
    diferencias, 156  
    incompleta, 479  
    introducción, 4  
    parcial, 479, 695  
lista completa, 57  
lista de adyacencias

- representación de un grafo, 507  
lista de diferencias, 156  
lista por comprehensión, 332  
literal, 690  
declaración **local**, 61, 68  
declaración **lock**, 636  
Luis XIV, 443, 449  
Lynch, Nancy, 388
- máquina abstracta, 44, 60–85, 100–102, 264–266, 309–311, 383–384, 455–456  
basada en substituciones, 139–140  
máquina de estados finitos  
manejador de eventos, 613  
máquina de Turing, 44, 128, 714  
máquina virtual, xvi, 44  
método  
    envolver, 563  
    objeto, 20, 542  
método científico, xvi  
método de Newton para raíces cuadradas, 131  
método vigilado, 650  
módulo, 201, 243, 496  
    **Array**, 476  
    **Atom**, 691  
    **Browser**, 247  
    **Char**, 687, 689  
    **Compiler**, 681  
    **Dictionary**, 477  
    **File** (suplemento), 232, 322, 615  
    **Finalize**, 525  
    **Float**, 687  
    **Int**, 687  
    **List**, 284, 421  
    **Module**, 246, 451, 497, 683  
    **MyList** (ejemplo), 244  
    **Number**, 15, 59, 200  
    **OS**, 407, 665  
    **ObjectSupport**, 564  
    **Open**, 615  
    **Pickle**, 238, 246
- Property**, 102, 279, 281  
**QTk**, 234  
    uso en aplicación, 248  
    uso interactivo, 236  
**Record**, 693  
**Remote**, 281  
**String**, 691  
**Thread**, 281, 304  
**Time**, 334  
**Tuple**, 694  
**Value**, 360, 697  
**Base**, 245, 252  
biblioteca, 252  
enlace dinámico, 313  
    falla, 361  
**Erlang**, 426  
especificación, 243  
importación, 247  
interfaz, 497  
**System**, 245, 252  
tipamiento dinámico, 114  
    unidad de compilación, 496  
mónada, 339, 364  
mónadas, xxi  
Mac OS X, xxiv, xxvii, 280  
MacQueen, David, 370  
macro  
    Lisp, 42  
        ciclo (en Common Lisp), 209  
operación **MakeRecord**, 182, 693  
operación **MakeTuple**, 411, 694  
Manchester Mark I, 39  
manejador  
    excepción, 98, 99  
    finalización, 526  
    GUI design, 235  
manejador de eventos, 613  
    agregando funcionalidad con herencia, 615  
    objetos activos, 613  
Manna, Zohar, 481  
mantenimiento, 500  
    herencia, 536  
polimorfismo, 465

*Índice alfabético*

- manzana, 533  
operación `Map`, 209, 510, 696  
marca, 543  
marca de anidamiento, 57, 91, 390, 401  
marca de tiempo, 657  
marco  
    modelo de computación, 712  
    reutilización de *software*, 537  
matriz  
    implementación con lista de listas, 255  
    representación de un grafo, 508  
operación `Max`, 213  
McCloud, Scott, 527  
mecanismo de contrato de tiempo, 525  
medición de tiempo  
    objeto activo, 416  
medición de tiempos  
    clausura transitiva, 514  
medida  
    consumo de memoria, 190  
medida de tiempo  
    frecuencia de palabras, 220  
operación `Member`, 696  
memoria  
    ciclo de vida, 80  
    consumo, 189  
    dirección en la máquina abstracta, 60  
    direccional por el contenido, 641  
    insuficiente, 82  
memoria activa, 80  
    tamaño, 189, 345  
memoria alcanzable, 80  
memoria inactiva, 81  
memoria intermedia limitada, 290  
    versión con monitor, 647  
    versión concurrente dirigida por los datos, 290  
    versión perezosa, 319  
memoria libre, 81  
memorización, 457, 499  
    invocación por necesidad, 474  
    programación declarativa, 345  
    unificación, 112  
mensaje, 544  
mente del programador  
    conurrencia para determinación del orden, 301  
    estado (implícito vs. explícito), 446  
    forzando la encapsulación, 459  
    habilidades (átomos vs. nombres, 556  
    uso de restricciones, 302  
metodología, véase desarrollo de *software*  
metodología de diseño  
    composicional, 504  
    lenguaje, 43, 361, 591, 719  
    no composicional, 504  
    programa grande, 492  
    programa pequeño, 240  
    programas concurrentes, 400  
Meyer, Bertrand, 491, 535, 567, 574  
Microsoft Corporation, 506  
mirada funcional, 215  
Mnesia (base de datos de Erlang), 425  
**mod** (módulo entero) operation, 688  
**mod** operación (módulo entero), 59  
modelo  
    computación, 31, véase modelo de computación  
    programación, 31  
modelo actor, 379  
modelo concurrente basado en trabajo, 675  
modelo de computación, xi, 31  
    basado en objetos, 586  
    con estado, 452  
    conurrencia para determinación del orden, 301  
    conurrencia por paso de mensajes, 381  
    concurrente con estado comparti-

- do*, 624  
*concurrente con estado*, 621, 627, 628  
*concurrente declarativo*, 259  
*concurrente dirigido por la demanda*, 260, 309  
*concurrente dirigido por los datos*, 260  
*concurrente dirigido por la demanda con excepciones*, 360  
*concurrente no-determinística*, 432  
*concurrente perezoso*, 260, 309  
*declarativo con excepciones*, 100  
*declarativo concurrente con excepciones*, 358  
*declarativo descriptivo*, 127  
*declarativo seguro*, 223  
*declarativo*, 52  
*funcional estricto*, 106  
*general*, 711  
*maximalmente concurrente*, 629, 674  
*objeto activo*, 607  
*abierto*, 357  
*acción atómica*, 632  
*cerrado*, 357  
Erlang, 423  
falla parcial, 357  
Haskell, 340  
inseguro, 357  
Java, 601, 671  
principios de diseño, xix, 711  
programación concurrente por restricciones, 370  
programación lógica concurrente, 315, 432  
programación paralela determinada, 370  
reunión de diferentes modelos, xx  
seguro, 357  
usando diferentes modelos a la vez, 356  
modelo de coordinación, 498, 640  
modelo de programación, xi, 31  
modularidad, xxi, 346, 447, 501  
descomposición del sistema, 499  
inadecuación del modelo declarativo, 346  
reingeniería, 569  
relación con concurrencia, 264, 350  
relación con estado explícito, 346  
relación con la concurrencia, 277  
module  
    List, 696  
    Number, 687  
    ObjectSupport, 695  
MOGUL (Mozart Global User Library), 244  
monitor, 633, 646–653  
    condition variable, 652  
    implementation, 651  
    lenguaje Java, 672  
    método vigilado, 650  
    semántica de Java, 646  
monitor de procesamiento de transacciones, 666  
monolítica  
    función, 326  
monolítico  
    programación con estado, 515  
monotonicidad, 718  
    propiedad de necesidad, 312  
    reducción de hilo, 263  
    variable de flujo de datos, 622  
Moore, Gordon, 192  
Morrison, J. Paul, 282  
Mozart Consortium, xxiv, xxvii, 116  
Mozart Programming System, 280  
    Biblioteca Estándar, 236, 248, 252  
    bibliotecas como módulos, 252  
Browser, 1  
compilación separada, 500  
Compiler Panel tool, 681  
concurrencia económica, 277  
consumo de memoria, 190

*Índice alfabético*

- descripción general, xxiv  
desempeño, 220, 416  
Distribution Panel tool, 681  
excepción no capturada, 101  
Explorer tool, 681  
herramienta Brower, 96  
mostrando estructuras cílicas, 112  
interfaz de la línea de comandos, 683  
interfaz interactiva, 1, 94, 681  
kernel languages, 712  
licencia Open Source, xxiv  
módulos Base, 252  
módulos System, 252  
Panel tool, 681  
papel limitado del compilador, 549  
recolección de basura, 84  
thread scheduler, 279  
muerte súbita, 660  
control de la concurrencia, 660  
detección, 660  
evitar, 660  
prevención, 660  
resolución, 660  
multiconjunto, 264  
multimedia, 193  
mundo pequeño  
simulación, 531  
Myriorama, 301  
número, 56, 685  
números seudoaleatorios, 517  
natural design, 621  
Naur, Peter, 35  
necesidad de una variable, 311  
New operation, 598  
operación New, 540  
operación NewArray, 476  
operación NewCell, 455  
operación NewDictionary, 477  
operación NewLock, 23, 636  
operación NewName, 224, 690  
operación NewPort, 383  
Newton, Isaac, 306  
nil, 695  
no-determinismo  
introducción, 21  
limitación del modelo declarativo, 350  
modelo concurrente declarativo, 627  
no observable, 625  
observable, 21, 22, 258, 350, 623, 629  
operación Filter, 423  
memoria intermedia limitada, 649  
relación con excepciones, 358  
planificador de hilos, 278  
relación con corutinas, 303  
no-determinismo no sé, 356  
nodo (en Erlang), 424  
nombre, 223, 690, 715  
definición de alcance, 554  
fresco, 224  
generación, 227  
nondeterminism  
observable, 627  
normal  
orden de reducción, 362  
NP-completo  
problema, 193  
caracter NUL, 709  
O'Keefe, Richard, 123, 445  
objeto puerto  
razonamiento, 387  
objeto, 452, 459, 591, 595  
declarativo, 463, 528, 619  
fuerte, 591  
introducción, 18  
Java, 601  
pasivo, 607, 628, 629  
objeto, activo, 384, 607–618  
comparación con objeto pasivo, 607

- definiendo el comportamiento con una clase, 608  
polimorfismo, 469  
objeto, flujo, *emph>282, 292–293, 450, 452, 627*  
comparación con objeto puerto, 385  
el problema de Flavio Josefo, 612  
iteración de alto orden, 284  
polimorfismo, 469  
productor/consumidor, 283  
transductor, 285  
objeto, puerto, 380, 384, 452  
compartiendo un hilo, 414  
comunicación muchos-a-uno, 385  
cuándo usarlo, 629  
enfoques de concurrencia, 625  
Erlang, 613  
lecturas adicionales, 635  
polimorfismo, 469  
reactivo, 386  
Ockham, William of, 31  
ocurrencia de identificador libre, 62  
ocurrencia libre de un identificador, 69  
ocurrencia ligada de un identificador, 69  
Okasaki, Chris, 192  
Okasaki, Chris, 362  
OLE (Object Linking and Embedding), 506  
OMG (Object Management Group), 506  
Open Source software, xxiv  
operación  
en abstracción de datos, 459  
operación . (selector de campo) , 60  
operación Arity , 60  
operación Label , 60  
operación bloqueante, 264  
operación de intercambio  
sobre atributos de objetos, 542  
operación de plegado, 204  
FoldL, 205, 509  
FoldR, 205  
objetos flujo, 284  
operación exchange  
sobre celdas, 455  
operación no bloqueante, 365  
receive (en Erlang), 432  
eliminación (en cola), 651  
leer (en espacio de tuplas), 640  
recepción, 365  
remisión, 365  
operación notify (en monitor), 646  
operación notifyAll (en monitor), 646  
operación o-exclusivo, 15  
operación signal (en monitor), 646  
operación wait (en monitor), 646  
operador  
*mixfix*, 705  
asociatividad, 37, 705  
biario, 704  
binario, 37  
infijo, 57, 90, 204, 693, 695, 697, 705  
posfijo, 705  
precedencia, 37, 705  
prefijo, 705  
ternario, 707  
unario, 704  
operator  
*mixfix*, 693  
OPI (interfaz de programación de Oz)  
, 681  
optimización, 194  
compilador, 178  
computador estándar, 345  
desempeño del monitor, 651  
detección temprana de errores, 550  
ejecución ansiosa, 333  
prioridades de los hilos, 292  
protocolo corto-circuito, 611  
sistema de objetos, 594  
optimización de última invocación, 78

*Índice alfabético*

- optimización de invocación de cola, 78  
oración lógica  
afirmación, 482  
invariante, 482  
orden de reducción, 362–364  
orden de reducción aplicativo, 363  
orden de reducción normal, 362  
orden lexicográfico (de átomos), 690  
orden lexicográfico( de átomos), 60  
orden normal  
Haskell, 340  
orden parcial, 262  
operador **orelse**, 90  
método **otherwise**, 545  
OTP (Ericsson Open Telecom Platform), 423  
Oz, Mago de, 1  
comando **ozc**, 251, 683  
  
párrafo (en OPI), 682  
público  
alcance, 553  
palabra  
especificación, 241  
frecuencia, 218  
memoria, 80  
palabras reservadas (tabla de), 707  
Panangaden, Prakash, 370  
Panel tool, véase Mozart Programming System  
Papert, Seymour, xi, 257  
par lista, 57, 695  
par punto, 5, 695  
parada  
problema, 230  
paradigma, xi, xvi, véase modelo de computación  
de programación, xvi  
de programación, xi  
declarativo, 31  
escuela de pensamiento, xv  
parallelismo, 261, 353  
diferencia con concurrencia, 353  
  
importancia de la complejidad en el peor caso, 191  
importancia de la flexibilidad, 363  
parcial  
falla, 357  
terminación, 371  
paridad, 15  
Parnas, David Lorge, xxii, 505  
Pascal, Blaise, 5  
paso de parámetros, 470–474  
invocación por necesidad, 473  
evaluación perezosa, 473  
exercise, 530  
invocación por nombre, 472  
exercise, 529  
invocación por reference, 470  
invocación por valor, 471  
invocación por valor-resultado, 471  
invocación por variable, 470  
por referencia, 62  
Java, 605  
por valor  
Java, 605  
paso de testigos, 632, 644  
paso por ..., véase paso de parámetros  
patrón de diseño, 582  
patrones de diseño, xx, 452, 533, 582–585  
concurrencia declarativa, 400  
patrón Composición, 582  
pedazo, 695  
utilización en reflexión de objetos, 564  
perfilamiento, 194  
permutaciones, 2  
persistencia  
Erlang, 425  
estructura de datos, 164, 327  
transacciones, 655  
pila  
abierta y declarativa, 461

- administración de la memoria, 80  
con estado, desempaquetada, y segura, 464  
con estado, empaquetada, y segura, 463  
concurrente con estado, 630  
declarativa abierta, 214  
declarativa segura desempaquetada, 226  
declarativa, desempaquetada, y segura, 462  
declarativa, empaquetada, y segura, 462  
objeto declarativo, 463  
prueba de corrección, 482  
recorrido en profundidad, 172  
semántica, 66  
pila semántica, 67  
ejecutable, 67  
suspendido, 67  
terminado, 67  
pixel, 606  
planificación optimista, 658  
planificación pesimista, 658  
planificador  
aleatoriedad, 516  
operación `Delay`, 335  
determinístico, 278  
hilo, 263, 277  
no-determinístico, 278  
round-robin, 278, 282  
sistema de control de ascensores, 402  
transacción, 658  
POLA (Principio de Menor Autoridad), 230  
polimorfismo, 19, 116, 465, 506, 537  
ad-hoc, 469  
distribución de responsabilidad, 465  
ejemplo, 577  
Haskell, 342  
objetos activos, 469  
objetos flujo, 469  
objetos puerto, 469  
programación orientada a objetos, 534  
universal, 469  
Ponsard, Christophe, 594  
portal, 520  
poscondición, 482, 568  
precisión aritmética arbitraria, 4  
precondición, 482, 568  
preprocesador, 349  
DCG extendida (en Prolog), 154  
falacia del, 349  
patrones de diseño, 584  
presupuesto de billones de dólares, 568  
prevención, 278  
principio  
*software* que funciona se conserva funcionando, 64  
abstracción, 448  
abstracción funcional, 4  
ansioso por defecto, perezoso declarado, 362  
balance entre planificación y re-factorización, 494  
compartimentar la responsabilidad, 492  
componente con estado con comportamiento declarativo, 456  
concentración del estado explícito, 450  
concurrencia declarativa, 267, 309  
confinamiento del error, 98  
de descomposición del sistema, 231  
de extensión creativa, xii  
decisiones en el nivel correcto, 503  
definición de tipo primero, 151  
dependencias predecibles, 502  
diseño de lenguajes por capas, 719  
documentar interfaces de los com-

*Índice alfabético*

- ponentes, 492
- el *software* que funciona sigue funcionando, 502
- encapsular decisiones de diseño, 501
- evitar la optimización prematura, 493
- evitar modificar interfaces, 501
- evite la optimización prematura, 194
- explotar la uniformidad de las abstracciones de datos, 592
- extensión creativa, 712
- independencia del modelo, 499
- intercambio libre de conocimiento, 492
- la estructura de la función sigue la estructura del tipo, 149
- la forma debe reflejar el contenido, 593
- más no es mejor (o peor), solo es diferente, *xix*
- mínima expresividad, 354
- menor autoridad (POLA), 230
- menor privilegio, 230
- minimizar dependencias, 424, 501
- minimizar referencias indirectas, 502
- no compartimentar la responsabilidad, 465
- objetos sobre TADs, 534
- optimización de última invocación, 78
- pague sólo cuando lo use, 677
- propiedad de substitución, 565, 568, 570
- reglas de diseño de Mozart, xxvi
- selección natural, 493, 505
- separación de asuntos, 356, 618
- toda clase es final por defecto, 536
- “todo debería ser un objeto”, 591
- todo se hace en tiempo de ejecución, 549
- usar abstracción de datos en todas partes, 592
- usar abstracciónde datos en todas partes, 533
- violaciones documentadas, 503
- privado
  - alcance, 553, 554
- probabilidad
  - distribución de Gauss, 520
  - distribución exponencial, 519
  - distribución normal, 520
  - distribución uniforme,, 518
- probador de teoremas, 129
- problema
  - de Hamming, 322
  - intratable, 193
  - lógica digital
    - satisfactibilidad, 193
  - NP-completo, 193
- problema de Flavio Josefo, 608–612
- problema de Hamming, 376
- problema de la parada, 230
- problema indecidible, 230
- declaración **proc**, 70
- procedimiento
  - como componente, 451
  - importancia, 58
  - operaciones básicas, 60
  - orden, 194
  - recursivo por la cola, 78
  - referencia externa, 71
- procesador, 261
  - máquina de flujo de datos, 513
  - programación funcional paralelo, 363
- procesador Pentium III, 220, 514
- procesamiento por lotes
  - invocaciones de objetos, 588, 616
- proceso
  - CSP, 675
  - diseño de programas concurren-  
tes, 400

- diseño de programas en grande, 492  
diseño de programas pequeños, 240  
Erlang, 384, 423, 426  
error en tiempo de ejecución, 104  
sistema operativo, 281  
procesos  
    cálculo concurrente, 58  
productor, 283  
programa abierto, 221  
programa declarativo, 269  
programa legal, 33  
Programación  
    orientada a objetos (OOP), 533  
programación, *xiv*, 1  
    abierta, 115, 221  
    alto orden, 126, 136  
    introducción, 14  
    relación con orientación a objetos, 586  
    basada en componentes, 451  
    basada en flujos, 283  
    basada en objetos, 20, 586  
    buen estilo, xx  
    centrada en datos, 629  
    con estado, 31  
    concurrente, 625  
    declarativa, 31, 444  
        descriptiva, 127  
        programable, 128  
    desarrollos futuros, 504  
    enfoque del lenguaje núcleo, *xvi*  
Erlang, 425  
funcional, 444  
Haskell, 339  
imperativa, 31, 444  
Java, 602, 671  
lógica, 47, 109, 157  
multi agente, 629  
multi-agente, 451  
multiparadigma, *xii*, *xxvi*  
    manejador de eventos, 617  
orientada a objetos(POO), 20,  
    452  
paradigma, *xv*, 31, *véase* modelo  
    de computación  
por acumuladores, 153  
por restricciones, 47  
restricción, 629  
sin estado, 31  
sincrónica, 293  
tiempo real, 334  
programación sincrónica, 293  
programación de alto orden, 126,  
    194–214  
    abstracción de iteración, 136  
    introducción, 14  
    operaciones en biblioteca, 696  
    relación con POO, 586  
programación extrema, 494  
programación lógica, 47, 444  
    lista de diferencias, 157  
    modelo de procesos, 432  
    unificación, 109  
programación orientada a la concu-  
    rrencia (POC), *xv*, *xxii*  
programación orientada por concu-  
    rrencia (POC), 625  
programación por restricciones, 47,  
    302  
    hilos, 629  
programacion por restricciones  
    hilos, 280  
programas de ejemplo (cómo correr-  
    los), *xxix*  
Prolog  
    Aquarius, 154  
    paquete de hilos de estado, 209  
    SICStus, 209  
propiedad  
    objeto, 542  
    seguridad, 657  
    vitalidad, 657  
propiedad de substitución, 565, 568,  
    570  
propiedades ACID, 654  
protector, 357

*Índice alfabético*

- protegido  
    alcance, 554
- protocolo, 388  
    by-need, 311  
    corto-circuito, 611  
    DHCP (Dynamic Host Connection Protocol), 227  
    IP (Internet Protocol), 227  
    meta-objeto, 562  
    negociación, 413  
    temporizador, 404  
    protocolo meta-objeto, *véase* protocolo, meta-objeto
- proyecto ACCLAIM, xxvi  
 proyecto de investigación, xxiv  
 proyecto Information Cities, 451  
 prueba de ingeniería, 129  
 Psion Series 3 palmtop computer, 414  
 puerto (estado explícito), 383–384, 717  
 semántica distribuida, 420
- pulsos  
    producción de, 336
- puntero, 82  
    recolección de basura, 82  
    tipamiento dinámico, 115
- punto  
    espacio bidimensional, 604  
    sincronización, 366
- punto caliente, 194
- punto de entrada (en bloque), 530
- punto de espera, 633
- punto de programa, 662
- punto de reposo, 371
- punto de retraso, 633
- punto de salida (en bloque), 530
- punto del programa, 485
- punto flotante  
    aritmética, 688  
    número, 686
- QTk, 234  
    uso en aplicación, 248  
    uso interactivo, 236
- quantum (en planificación de hilos), 278
- declaración **raise**, 101
- Raymond, Eric, 505
- razonamiento  
    acción atómica, 633  
    algebraico, 123, 128, 355  
    causal, 388, 627  
    lógica, xviii  
    modelo con estado, 355, 481  
    modelo concurrente con estado compartido, 356  
    modelo concurrente por paso de mensajes, 387  
    sistema de control de ascensores, 411  
    razonamiento lógico, 123  
    expresión **receive** (en Erlang), 428
- recepción  
    asincrónica, 365  
    o bloqueante, 365  
    sincrónica, 365
- receptor  
    variable de flujo de datos, 94
- receptor(consumidor), 285
- recolección de basura, 80, 82  
    “copying dual-space”, 85  
    conjunto raíz, 82  
    finalización, 525  
    generacional, 85  
    pausa en ejecución, 82  
    pausa en el programa, 338  
    referencias externas, 524  
    tiempo real, 83
- recolector, 208, 357, 475, 528
- reconocimiento de patrones  
    declaración **case**, 6, 72  
    función (en Erlang), 425  
    Haskell, 339  
    expresión **receive** (en Erlang), 428
- recorrido en amplitud, 171
- recorrido en profundidad, 170

- recursión, 3, 126, 137  
    directa, 126  
    indirecta, 126  
    mutua, 120  
    optimización de recursión de cola, 78  
    polimórfica, 339  
    programando con, 140
- recurso  
    canal productor/consumidor, 288  
    descriptor de archivo, 322  
    externo, 84, 525  
    uso de la pereza, 318
- Red Hat Corporation, xxvii, 220, 514
- reenvío, 558–559
- refactorización, 494
- referencia suelta, 70, 81, 198, 502, 602
- referencias compartidas, 457, 490
- reflexión, 562
- región (en OPI), 682
- región crítica, 636
- registro, 20, 56, 691  
    adjuntar, 694  
    administración de la memoria, 80  
    compromisos de uso, 479  
    creación dinámica, 182, 598  
    finalización, 525  
    importancia, 57  
    máquina abstracta, 60  
    operaciones básicas, 59, 693  
    tipo, 479
- registro de errores, 614
- reglas de prueba, 485
- reingeniería, 569
- relación de anulación, 548
- reloj  
    circuito digital, 298  
    lógica digital, 294  
    sincronización, 338
- reloj de alarma, xiv
- remisión  
    asincrónica, 365  
    o bloqueante, 365  
    sincrónica, 365
- rendezvous, 675
- rendimiento, 290
- replicador, 357
- reposo  
    punto, 371
- resolución  
    muerte súbita, 660
- responsabilidad  
    administración de la memoria, 83  
    atomicidad y consistencia (en transacciones), 655  
    compartimentar (en un equipo), 492  
    confinamiento de la falla, 271  
    corutina (evitar inanición), 303  
    diseño por contrato, 568  
    inferencia de tipos, 151  
    papel del polimorfismo, 465
- restricción, 269  
    dibujar árboles, 174
- restricción (ambiente), 67
- restrictiones  
    programación por, 302
- return (en ciclo `for`), 209
- Reuter, Andreas, 635, 655
- Reynolds, John C., 459
- Rinard, Martin C., 370
- RMI (invocación remota de métodos), 388
- operación Round, 689
- RPC (invocación remota de procedimiento), 389
- rueda, xviii
- ruido (electrónico), 517
- síntesis  
    argumentos, 177
- Sacks, Oliver, 443
- Saint-Exupéry, Antoine de, 123
- Saraswat, Vijay A., 370
- Schulte, Christian, xxvi
- sección crítica condicional, 650
- secuencial  
    lógica, 296

*Índice alfabético*

- seguridad, 657  
    átomo vs. nombre, 554  
abstracción de datos, 458–475  
abstracción lingüística, 42  
conceptos del lenguaje núcleo,  
    715  
derecho, 715  
habilidad, 229  
lenguaje, 229  
mecanismo, 229  
política, 228  
sociedad humana, 228  
tipamiento estático, 115  
tipo abstracto de datos, 221–231  
selección natural, 493, 505  
**self**  
    clone, 564  
    delegación, 557  
    Java, 603  
    ligadura dinámica, 551  
    reenvío, 557  
        notación `this`, 601  
**self** (en Erlang), 427  
semántica, 33  
    axiomática, 40, 481–491  
    celda, 455  
    declaración semántica, 66  
    denotacional, 41  
    disparador by-need, 309  
    enfoque del lenguaje núcleo, 40  
    excepciones, 100  
    hilo, 264  
    lógica, 41  
    lenguaje núcleo, véase máquina  
        abstracta  
    máquina abstracta, 60–85, 100–  
        102, 264–266, 309–311, 383–  
        384, 455–456  
    monitor (en Java), 646  
    operacional, 40, 65  
    puerto, 383, 420  
    tipos seguros, 223  
semántica axiomática, 40, 481–491  
semántica de intercalación, 261  
semántica denotacional, 41  
semántica lógica, 41  
semántica operacional, 40, 65  
operación `Send`, 383  
    semántica de reserva de espacio,  
        421  
sensitiva al contexto  
    gramática, 36  
separación de asuntos, 356, 618  
serializabilidad, 655  
serializador, 356  
servicios Web, 379  
servidor de nombres, 441  
dirección de 64 bits, 85  
palabra de 64 bits, 80, 192, 686  
Shakespeare, William, 681  
operación `Show`, 373  
siempre es cierta (en lógica temporal),  
    658  
signo menos (uso de virgulilla ~), 686  
simulación  
    componentes, 451  
    de boca en boca, 520  
    Internet, 451  
    lógica digital, 293–300  
    multi-agente, 451  
    mundo pequeño, 531  
    red lenta, 631  
simulación de boca en boca, 520  
sin estado (programación declarativa), 123  
sincronización  
    reloj, 338  
sincronización, 365–370  
singularidad, 193  
sintaxis, 33  
    construcciones que se pueden  
        anidar (en Oz), 702  
    convención para los ejemplos,  
        xxix  
    declaraciones que se pueden anidar  
        (en Oz), 702  
    lenguaje, 33  
    lenguaje Oz, 701

- término (en Oz), 702  
    sintaxis de los lexemas (de Oz), 701  
    sintaxis léxica de Oz, 708  
    sistema de control de ascensores, 402  
    sistema distribuido, 379  
        abierto, 554  
        falla parcial, 357  
        semántica de puerto, 421  
    sistema interactivo  
        diseño GUI, 234  
        garantía en tiempo de reacción, 191  
        IDE de Mozart, 681  
        uso pedagógico, xviii  
    sistema multiagentes (MAS), 379  
    sistema operativo, 229  
        Linux, xxvii, 220, 514, 543  
        Mac OS X, xxiv, xxvii, 280  
        Solaris, xxvii, xxix  
        Unix, xxiv, 280, 502, 603  
            pipe, xv  
            tubería, xv  
        VM(Virtual Machine), 44  
        Windows, xxiv, 280  
    sistema operativo Unix, 336  
        tubería, 282  
    sistemas de información, xxi  
    sistemas multi agentes (MAS), 629  
    sistemas multiagentes, xxiii  
    sistemas multiagentes (MAS), 398  
    declaración **skip**, 68  
    Smolka, Gert, xxvi  
    sobrecarga, 469  
    sobrecargados, 344  
    Solaris, xxvii, xxix  
    operación **Sort**, 696  
    (entrada estándar) **stdin**, 251  
    **stdin** (standard input), 603  
    **stdout** (standard output), 603  
    Steiner, Jennifer G., 367  
    operación **StringToAtom**, 691  
    substitución, 139  
    subtipo  
        jerarquía de clase, 565  
    tipos básicos, 55  
    sumador  
        n-bit, 374  
    sumador completo, 295  
    Sun Microsystems, xxvii, 506  
    superclase, 548, 560, 606  
    supercomputador, 192  
    suspensión, 472  
        operación **Delay**, 335  
        hilo, 264, 304  
        por un error del programa, 52, 97  
    Sussman, Gerald Jay, 45  
    Sussman, Julie, 45  
    Symbian Ltd., 414  
    palabra clave **synchronized**, 646  
    palabra reservada **synchronized**, 672  
    Szyperski, Clemens, 505  
  
térmico  
    Oz, 702  
tabla de hash, 479  
Tanenbaum, Andrew S., 367  
tecnología, xiv  
    Software por componentes, 506  
    computación molecular, 193  
    digital sincrónica, 294  
    hostoria de la computación, 193  
    magia, 345  
    peligros de la concurrencia, 22  
    reingeniería, 569  
    singularidad, 193  
        transición a 64 bits, 85  
Tel, Gerard, 388  
teléfono celular, xiv, xviii  
temporizador  
    protocolo, 404  
teorema  
    del binomio, 5  
teorema de Church-Rosser, 363  
teorema del binomio, 5  
tercera ley de Clarke, 345  
terminación  
    detección, 304, 419  
        ejemplo ping-pong, 336

*Índice alfabético*

- falla en un programa declarativo, 270  
parcial, 268  
prueba, 490  
terminación parcial, 268, 371  
Thalys high-speed train, 419  
**this**, véase **self**  
Thompson, DÁrcy Wentworth, 443  
tiempo de espera  
    diseño de sistemas, 503  
    Erlang, 428–432  
tipo, 54, 214  
    abierto, 459  
    abstracción de datos concurrente, 630  
    abstracto, 214  
    APD (objeto), 459  
    atómico, 690  
    básico, 56  
    clase, 343  
    con estado, 460  
    débil, 114  
    declarativa, 460  
    descriptiva, 143  
    dinámico, 55, 114–116  
    empaquetado, 459  
    estático, 55, 114–116  
    firma, 142, 339  
    fuerte, 114, 339  
    inferencia, 107, 151, 339  
    jerarquía, 55  
    polimórfico, 342  
    seguro, 221–231, 458–475  
    sin estado, 460  
    TAD, 214  
tipo abstracto de datos (TAD), 214–231, 458, 459  
tipo de datos, véase tipo  
tiquete, 525  
tolerancia a fallos  
    Erlang, 424  
    sistema de control de ascensores, 413  
tolerancia a los fallos, 656  
traductor, véase compilador  
transacción, 633, 654–671  
    adquisición y liberación de canales en dos fases, 659  
    anidada, 671  
    aseguramiento en dos fases, 659  
    distribuida, 671  
    ejemplo de registro, 550  
    ejemplo del banco, 222  
    encarnación, 662  
    reinicio, 661  
    transacción liviana, 655  
transductor, 285  
transición, 404  
transparencia a la red, 281  
transparencia de red, 424  
transparencia referencial, 125  
traslado  
    sincronización, 393  
dirección de 32 bits, 85  
palabra de 32 bits, 80, 190  
triángulo de Pascal, 5  
**true**, 691  
declaración **try**, 101  
tupla, 56, 692  
    compromisos de uso, 478  
tupla espacio, 498  
Turing, Alan, 128  
UML (Lenguaje de Modelamiento Unificado), 575  
Unicode, 501, 686  
    Java, 604  
unificación, 108–114, 715  
    algoritmo, 109  
    conjunto de variables equivalentes, 110  
    ejemplos, 108  
**unit**, 691  
Unix, xv, xxiv, 280, 502, 603  
    sistema operativo, 336  
URL (Uniform Resource Locator), 233  
vida limitada, 502

- valor, 46  
    compatible, 50  
    completo, 50  
    en abstracción de datos, 459  
    fallido, 360, 716  
    parcial, 49  
valor completo, 50  
valor de tipo procedimiento (clausura), 70–71, 195  
anónimo, 57  
codificación como un objeto, 589  
limitación común, 197, 602  
programación de alto orden, 194  
relación con clase interna, 602  
valor parcial, 49  
variable, 2  
    conjunto de equivalencia, 110  
    de flujo de datos, 715  
    declarativa, 45  
    determinada, 71, 110, 227, 365  
    especial (en Common Lisp), 64  
    final (en Java), 601  
    flujo de datos, 45, 52  
    global, 95  
    identificador, 2  
    identificador de, 48  
    instancia, 542  
    interactiva, 95  
    lógica, 109  
    ligadura, 47, 49  
    local, 197  
    mutable, véase celda  
    necesitada, 311  
    sólo lectura, 226, 382, 417, 716  
    símbolo de escape, 545, 555  
    unificación, 108  
variable de asignación única, 369  
variable de condición, 653  
ventana del compilador (en OPI), 682  
ventana del emulador (en OPI), 682  
vigilancia  
    monitor, 650  
Virding, Robert, 635  
virtual  
cadena, 232  
cadena de caracteres, 699  
visibilidad  
    horizontal, 554  
    vertical, 553  
Visual Basic, 504  
vitalidad, 657  
Vlissides, John, 582  
sistema operativo VM (Virtual Machine), 44  
Vossen, Gottfried, 655  
  
operación `WaitNeeded`, 715  
operación `WaitTwo`, 351, 352, 431, 434, 437  
Weikum, Gerhard, 655  
operación `Width`, 693  
Wikström, Claes, 635  
Wilf, Herbert S., 187  
Williams, Mike, 635  
Windows, xxiv, 280  
Wood, David, 414  
WWW (World Wide Web), 233  
  
XML (Extensible Markup Language), 127  
  
zombi, 611