

Programación Lógica

Teoría y Práctica

depuración

verificación

certificación

Pascual Julián Iranzo
María Alpuente Frasneda

PEARSON
Prentice Hall

FREE LIBROS

TU BIBLIOTECA VIRTUAL

<http://www.freelibros.com>



Programación Lógica

Teoría y Práctica

Programación Lógica

Teoría y Práctica

Pascual Julián Iranzo

*Departamento de Tecnologías y Sistemas de Información
Universidad de Castilla-La Mancha*

María Alpuente Frasneda

*Departamento de Sistemas Informáticos y Computación
Universidad de Politécnica de Valencia*



Madrid • México • Santa Fe de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • São Paulo • White Plains

JULIÁN IRANZO, P. Y ALPUENTE FRASNEDO, M.
Programación lógica. Teoría y práctica
Pearson Educación S.A., Madrid, 2007

ISBN: 978-84-8322-368-0
Materia: Informática, 004

Formato: 195 x 250 mm. Páginas: 496

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (arts. 270 y sgts. Código Penal). DERECHOS RESERVADOS

2007 por PEARSON EDUCACIÓN S.A.
Ribera del Loira, 28
28042 Madrid

JULIÁN IRANZO, P. Y ALPUENTE FRASNEDO, M.
Programación lógica. Teoría y práctica

ISBN: 978-84-8322-368-0

Depósito Legal: M.
PEARSON PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN S.A.

Equipo editorial
Editor: Miguel Martín-Romo
Técnico editorial: Marta Caicoya
Equipo de producción:
Director: José A. Clares
Técnico: @Libro TEX
Diseño de cubierta: Equipo de diseño de Pearson Educación S.A.
Impreso por:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

a Nieves, X e Y.
P.J.

a Jose, Paloma y Claudia.
M.A.

Índice general

1. Una Panorámica de la Programación Declarativa	1
1.1. Computadores y Lenguajes de Programación	1
1.1.1. Organización de los computadores	1
1.1.2. Características de los lenguajes convencionales	5
1.2. Programación Declarativa	9
1.2.1. Programación Lógica	10
1.2.2. Programación Funcional	12
1.3. Comparación con los Lenguajes Convencionales y Áreas de Aplicación	18
Resumen	24
Cuestiones y Ejercicios	27
 I Fundamentos	 29
2. Sistemas Formales, Lógica y Lenguajes de Programación	31
2.1. Sistemas Formales	31
2.2. Lógica de Predicados: revisión de conceptos	33
2.2.1. El lenguaje formal \mathcal{L}	33
2.2.2. Cálculo deductivo, $\mathcal{K}_\mathcal{L}$	37
2.2.3. Semántica del lenguaje de la lógica de predicados	39
2.2.4. Propiedades de la lógica de predicados	43
2.3. Semánticas y Lenguajes de Programación	45
2.3.1. Necesidad de una definición semántica formal	45
2.3.2. Semánticas formales de los lenguajes de programación	49
2.3.3. Lenguajes de Programación Declarativa	52
Resumen	53
Cuestiones y Ejercicios	55
 3. De la Demostración Automática a la Programación Lógica (I): introducción y métodos semánticos	 61
3.1. Razonamiento Automático	61
3.1.1. Aproximaciones a la demostración automática de teoremas	62
3.1.2. Un demostrador automático genérico	64

3.1.3. Límites de la demostración automática de teoremas	65
3.2. Procedimiento de Prueba por Refutación	66
3.3. Formas Normales	67
3.3.1. Formas normales en la lógica proposicional	69
3.3.2. Formas normales en la lógica de predicados	70
3.4. La Forma Clausal	72
3.4.1. Forma normal de Skolem	72
3.4.2. Cláusulas	75
3.4.3. El papel de la forma clausal en los procedimientos de prueba	76
3.5. Teorema de Herbrand	77
3.5.1. Universo de Herbrand e interpretaciones de Herbrand	77
3.5.2. Modelos de Herbrand	84
3.5.3. El teorema de Herbrand y las dificultades para su implementación	85
Resumen	90
Cuestiones y Ejercicios	92
4. De la Demostración Automática a la Programación Lógica (II): el principio de resolución de Robinson	97
4.1. El Principio de Resolución en la lógica de proposiciones	98
4.2. Sustituciones	102
4.3. Unificación	105
4.4. El Principio de Resolución en la lógica de predicados	108
4.5. Estrategias de Resolución	112
4.5.1. Estrategia de resolución por saturación de niveles	113
4.5.2. Estrategia de borrado	113
4.5.3. Estrategia de resolución semántica	114
4.5.4. Estrategia de resolución lineal	115
4.5.5. Estrategia de resolución lineal de entrada y de preferencia por cláusulas unitarias	117
Resumen	118
Cuestiones y Ejercicios	119
II Programación Lógica	123
5. Programación Lógica	125
5.1. Sintaxis	127
5.1.1. Notación para las cláusulas	127
5.1.2. Cláusulas de Horn y Programas Definidos	128
5.2. Semántica Operacional	131
5.2.1. Procedimientos de prueba y objetivos definidos	131
5.2.2. Resolución SLD y respuesta computada	131

5.2.3. Arboles de búsqueda SLD y procedimientos de prueba	136
5.3. Semántica Declarativa y Respuesta Correcta	142
5.4. Corrección y Completitud de la resolución SLD	144
5.5. Significado de los programas	146
5.5.1. Semántica operacional	146
5.5.2. Semántica declarativa	147
5.5.3. Semántica por punto fijo	149
5.5.4. Equivalencia entre semánticas	154
5.6. Semánticas no Estándar	154
5.6.1. Modularidad y Semánticas Composicionales	154
5.6.2. Observable de las Respuestas Computadas	156
Resumen	158
Cuestiones y Ejercicios	161
6. El Lenguaje Prolog: Introducción	167
6.1. Programación Lógica y Prolog	167
6.2. Prolog puro	169
6.2.1. Sintaxis	169
6.2.2. Mecanismo operacional	175
6.2.3. Traza de la ejecución de un objetivo	181
6.3. Programación en Prolog puro	184
6.3.1. Números naturales	184
6.3.2. Listas	188
6.3.3. Conjuntos	195
6.3.4. Autómata finito no determinista	197
6.4. Prolog puro no es pura lógica	200
6.5. Aritmética	203
6.6. Operadores	207
6.7. Arboles	210
6.7.1. Arboles binarios	211
6.7.2. Arboles de búsqueda	214
6.7.3. Arboles n -arios	217
6.8. Distintas clases de igualdad y unificación	219
Resumen	221
Cuestiones y Ejercicios	222
7. El Lenguaje Prolog: Aspectos Avanzados	233
7.1. El Corte, Control de la Vuelta Atrás y Estructuras de Control	234
7.1.1. El Corte	234
7.1.2. Otros predicados de control: <code>fail</code> y <code>true</code>	238
7.1.3. Estructuras de control	240
7.2. Negación	244

7.2.1.	El problema de la información negativa	244
7.2.2.	La negación en la programación lógica	246
7.2.3.	La negación en el Lenguaje Prolog	248
7.3.	Entrada/Salida	250
7.3.1.	Flujos de E/S	250
7.3.2.	Predicados de E/S	251
7.4.	Otros predicados predefinidos	253
7.4.1.	Predicados predefinidos para la catalogación y construcción de términos	253
7.4.2.	Predicados predefinidos para la manipulación de cláusulas y la metaprogramación	255
7.4.3.	El predicado <code>setof</code>	259
7.5.	Generar y Comprobar	262
7.5.1.	El problema del mini-sudoku	263
7.5.2.	El problema de las ocho reinas	265
7.6.	Programación Eficiente en Prolog	268
7.7.	Estructuras Avanzadas: listas diferencia y diccionarios	273
7.7.1.	Relación entre listas y listas diferencia	274
7.7.2.	Manipulación de listas diferencia	276
7.7.3.	Diccionarios	279
Resumen		281
Cuestiones y Ejercicios		283
III	Aplicaciones de la Programación Lógica	293
8.	Representación del conocimiento	295
8.1.	El problema de la representación del conocimiento	295
8.2.	Representación mediante la lógica de predicados	298
8.2.1.	Nombres, functores y relatores	299
8.2.2.	Conjunciones y conectivas	301
8.2.3.	Granularidad y representación canónica	304
8.2.4.	Cuantificadores	304
8.2.5.	Consideraciones y recomendaciones finales	307
8.3.	Representación en forma clausal	309
8.3.1.	Cuantificación existencial, variables extra, negación y respuesta a preguntas	316
8.3.2.	Conjuntos, tipos de datos y las relaciones “instancia” y “es_un”	319
8.3.3.	Representación mediante relaciones binarias	321
8.4.	Elección de una Estructura de Datos: Ventajas e Inconvenientes	324
8.5.	Redes semánticas y representación clausal	328
Resumen		333

Cuestiones y Ejercicios	335
9. Resolución de problemas	345
9.1. Resolución del problema mediante búsqueda en un espacio de estados .	346
9.2. Resolución de problemas y programación lógica	348
9.2.1. Problema del mono y el plátano	348
9.2.2. Problema de los contenedores de agua	350
9.2.3. Generación de planes en el mundo de los bloques	352
9.3. Estrategias de Búsqueda y Control	354
9.3.1. Estrategia de búsqueda en profundidad	356
9.3.2. Estrategia de búsqueda en anchura	358
9.4. Estrategias de búsqueda heurística	363
9.4.1. Búsqueda heurística: necesidad y limitaciones	364
9.4.2. Heurísticas locales	366
9.4.3. Búsqueda <i>primero el mejor</i>	373
Resumen	385
Cuestiones y Ejercicios	388
10. Programación Lógica y Tecnología Software Rigurosa	395
10.1. Motivación	396
10.1.1. Una aproximación <i>lightweight</i>	398
10.2. La Trilogía del Software	399
10.2.1. Datos, propiedades y programas	400
10.2.2. Procesos formales	401
10.2.3. Hiper–arcos y otros procesos formales	406
10.2.4. Herramientas avanzadas para el desarrollo del software: métodos ligeros	410
10.3. Transformación de Programas y Evaluación Parcial	411
10.3.1. Transformaciones de plegado/desplegado	412
10.3.2. Evaluación parcial	415
10.3.3. Corrección de la Evaluación Parcial	419
10.3.4. Generación automática de programas	421
10.3.5. Ventajas de la Evaluación Parcial	423
10.3.6. Evaluación parcial de programas lógicos	425
10.4. Síntesis de programas	427
10.4.1. Programación Lógica Inductiva	428
10.4.2. Aproximaciones y extensiones de ILP (y síntesis inductiva) . . .	428
10.4.3. Corrección de la síntesis	430
10.4.4. Síntesis constructiva	431
10.5. Diagnóstico y depuración automática de programas	432
10.5.1. Diagnóstico Declarativo	433
10.5.2. Depuración Declarativa en Programas Lógicos Puros	434

Resumen	437
Cuestiones y Ejercicios	437
A. Fundamentos y Notaciones Matemáticas 441	
A.1. Conjuntos	441
A.2. Relaciones y Funciones	443
A.2.1. Relaciones	443
A.2.2. Funciones	443
A.2.3. Numerabilidad y secuencias	445
A.2.4. Relaciones binarias	446
A.3. Grafos y Arboles	447
A.3.1. Grafos no dirigidos	447
A.3.2. Grafos dirigidos	448
A.3.3. Arboles	449

Índice de figuras

1.1.	Organización de un computador genérico.	2
1.2.	Un ejemplo de procesador y la MC.	3
3.1.	Diagrama de Hasse para las H-interpretaciones del Ejemplo 3.16.	83
3.2.	Arbol semántico.	86
4.1.	Arbol de deducción para la refutación del Ejemplo 4.5.	101
4.2.	Arbol de deducción para una deducción lineal genérica.	116
4.3.	Arbol de deducción para la refutación lineal del Ejemplo 4.21.	116
4.4.	Arbol de deducción para la refutación lineal del Ejemplo 4.22.	117
5.1.	Arbol de deducción para una derivación SLD genérica.	133
5.2.	Arbol de búsqueda SLD finito del Ejemplo 5.7.	138
5.3.	Arbol de búsqueda SLD infinito del Ejemplo 5.7.	139
5.4.	Efecto de la reordenación de cláusulas en el arbol de búsqueda SLD del Ejemplo 5.7.	140
6.1.	Clasificación de los objetos en Prolog.	170
6.2.	Exploración del árbol de búsqueda de Prolog de la Figura 5.2.	180
6.3.	Diagramas de flujo de ejecución de una llamada a procedimiento.	181
6.4.	Arbol de búsqueda para el programa y objetivo del Ejemplo 6.2.	183
6.5.	Una lista de enteros.	189
6.6.	Fases en la inversión de una lista usando un parámetro de acumulación.	194
6.7.	Un autómata finito no determinista.	197
6.8.	Diferencias entre un árbol binario y el correspondiente árbol 2-ario.	212
6.9.	Un autómata finito no determinista.	228
7.1.	Efecto del corte en el árbol de búsqueda (i).	235
7.2.	Efecto del corte en el árbol de búsqueda(ii).	235
7.3.	Arbol de búsqueda para el programa y objetivo del Ejemplo 7.2: Alternativa sin corte.	237
7.4.	Arbol de búsqueda para el programa y objetivo del Ejemplo 7.2: Alternativa con corte.	238
7.5.	Arbol de búsqueda para el programa y el objetivo del Ejemplo 7.4.	239

8.1.	Razonamiento y funciones de correspondencia.	296
8.2.	Traducción de un enunciado al lenguaje formal de la lógica.	309
8.3.	Situación en el mundo de los bloques.	325
8.4.	Un ejemplo de red semántica.	329
8.5.	Robots en un mundo bidimensional con obstáculos.	336
8.6.	Un circuito lógico simple.	338
8.7.	Un fragmento de red semántica con informaciones personales.	341
8.8.	Red semántica y a atributo genérico “ <i>tiene_parte</i> ”.	343
9.1.	Un árbol representando el espacio de estados de un problema.	347
9.2.	Reordenación en el mundo de los bloques.	352
9.3.	Conversión de un árbol de búsqueda en un grafo.	355
9.4.	Búsqueda de un camino en un grafo dirigido.	359
9.5.	Costes del proceso de búsqueda.	366
9.6.	Grafo para un problema del viajante de comercio.	368
9.7.	Pasos en un proceso de búsqueda <i>primero el mejor</i>	374
9.8.	Modificación del árbol de exploración.	378
9.9.	Mapa de Carreteras.	380
9.10.	Un ejemplo de árbol de búsqueda.	392
9.11.	Estados inicial y final para el problema del puzzle-8.	393
10.1.	La Trilogía del Software	399
10.2.	Perspectiva de los Métodos Formales de la Ingeniería del Software Automática	402
10.3.	Perspectiva de los Métodos Formales de la Ingeniería del Software Automática – Ciclos	403
10.4.	Perspectiva de los Métodos Formales de la Ingeniería del Software Automática - Hiper–arcos	408
10.5.	Perspectiva completa de la Ingeniería del Software Automática	411
10.6.	Reglas de transformación de Burstall y Darlington.	414
A.1.	Diferentes clases de funciones.	445
A.2.	Distintas clases de grafos.	447
A.3.	Representación gráfica de un árbol 3-ario.	449

Índice de tablas

1.1.	Estilos de Programación declarativa y características heredadas de la lógica que la fundamenta.	10
1.2.	Características de la programación lógica y de la programación funcional: Diferencias esenciales.	12
2.1.	Axiomas para un fragmento de un lenguaje imperativo.	50
3.1.	Fórmulas equivalentes de la lógica de proposiciones.	68
3.2.	Fórmulas equivalentes de la lógica de predicados.	71
4.1.	Algunos hechos notables sobre el principio de resolución.	99
5.1.	Clasificación de las cláusulas y objetivos.	128
6.1.	Conectivas lógicas y sintaxis de Prolog.	169
6.2.	Operaciones aritméticas	204
6.3.	Precedencia de las operaciones aritméticas	206
6.4.	Operaciones de comparación de expresiones aritméticas.	206
6.5.	Asociatividad de los operadores	209
8.1.	Enunciados categóricos y su formalización.	305
8.2.	Esquemas para facilitar la representación en forma clausal de una fórmula de la lógica de predicados.	315
A.1.	Tipos de grafos.	449

Prólogo

La *programación declarativa* se basa en la idea de utilizar un cierto tipo de lógica como lenguaje de programación. Esto incluye tanto la programación lógica (o relacional, que usa un lenguaje clausal para programar y el principio de resolución como mecanismo de ejecución) como la funcional (que usa el lenguaje de las funciones matemáticas y la reducción de expresiones como mecanismo computacional).

Este libro aspira a desarrollar una presentación básica completa del paradigma de la programación lógica. Así, aunque se dedica una gran atención a los conceptos de base y se proporciona para ellos una formación en *amplitud* no exenta de *profundidad*, también se concede gran importancia a las aplicaciones prácticas.

Concretamos el objetivo anterior en una serie de objetivos más específicos, que describimos a continuación:

- Entender los fundamentos teóricos que sustentan el paradigma de programación lógica (y por extensión el de la programación declarativa).
- Proporcionar una perspectiva histórica de la programación lógica, sus motivaciones, sus ventajas y sus defectos (en particular, en comparación con el estilo de programación imperativo).
- Mostrar que es posible caracterizar diferentes subconjuntos de la lógica para los que existen métodos eficientes de deducción automática y comprender que dichos subconjuntos pueden armarse como un lenguaje de programación.
- Introducir conceptos sobre la teoría de los lenguajes de programación que son difíciles de abordar cuando se emplean otros paradigmas. Por ejemplo: el concepto de lenguaje de programación como sistema formal; la necesidad e importante utilidad práctica de una definición semántica formal para los lenguajes de programación, etc.
- Entender los mecanismos computacionales asociados a un lenguaje programación lógica y controlar los mecanismos de deducción correspondientes, aprendiendo a combinar corrección con eficiencia.
- Presentar los distintos métodos, técnicas y herramientas para el desarrollo de aplicaciones mediante un lenguaje de programación lógica.

- Mostrar la relación de esta materia con otras disciplinas de la Informática, ya que se pretende capacitar al lector para que pueda aplicar las ideas y técnicas propias de la programación declarativa a otros campos de la informática.
- Transmitir el enorme potencial de la programación lógica para dar satisfacción a las crecientes demandas comerciales de software de calidad.

Para poder alcanzar dichos objetivos, nos centramos en la enseñanza de los elementos principales de la programación lógica: sus fundamentos sintácticos y semánticos, las técnicas de programación asociadas y los aspectos prácticos de utilización de las mismas.

A lo largo del libro se destacan dos aspectos de la programación lógica: por un lado, la precisión y sencillez de sus fundamentos lógicos; por otro lado, sus ventajas prácticas para el desarrollo de programas de un nivel de abstracción muy elevado, lo que facilita la concisión, claridad y mantenimiento de dichos programas.

ASPECTOS DIFERENCIALES

Nuestro enfoque se ajusta a un esquema de presentación (estándar) de un curso introductorio de programación lógica en las Escuelas de Ingeniería Informática. Sin embargo, queremos resaltar algunas características que distinguen nuestra obra y justifican el libro. Nuestra propuesta es la siguiente:

- **Desarrollo soportado por sólidas bases formales.**

Esta obra aspira a desarrollar sólidas bases formales que permitan al alumno entender con facilidad las relaciones existentes entre los lenguajes de programación declarativa y las bases teóricas que la sustentan (los modelos y cálculos deductivos que pueden constituir un soporte para el cómputo). Esta visión tiene indudables beneficios prácticos. Por ejemplo, el lenguaje lógico más popular, Prolog, presenta características impuras y, si se desconoce el ideal de la programación lógica, es del todo imposible distinguir entre buenas y malas técnicas de diseño de programas declarativos. Cuando se presentan las técnicas de programación lógica sin introducir las bases formales que sustentan las mismas, se pueden reproducir algunos hábitos incorrectos, propios de la programación imperativa, al no entender la verdadera naturaleza del lenguaje.

- **Equilibrio entre los contenidos formales y los prácticos.**

Sin embargo, los contenidos propuestos guardan un equilibrio entre el rigor formal en la presentación de los conceptos y la utilización práctica de esos conceptos. Conscientemente se ha buscado un equilibrio entre ambos extremos, lo que distingue a este libro de otras propuestas existentes en la literatura, en las que o bien se hace hincapié en los contenidos teóricos y formales, con un fuerte tratamiento matemático (difícil de entender para el tipo de alumno de grado al que preferentemente va dirigida nuestra obra), o bien se aborda el estudio de la programación

lógica de forma meramente descriptiva y centrándose en la enseñanza del lenguaje de programación lógica Prolog, sin apenas referencias a las bases teóricas.

Para lograr nuestro objetivo, todos los conceptos son definidos formalmente, y su importancia y utilidad se ilustran a través de numerosos ejemplos, que ayudan a entender dichos conceptos. También, al término de cada capítulo, se proponen muchos ejercicios para poner en práctica los diferentes conceptos.

Los contenidos progresan desde la teoría a las aplicaciones prácticas. Una vez desarrollada la teoría de la programación lógica, se introduce el lenguaje de programación Prolog, dejando claras las características que lo alejan del ideal de la programación lógica. Finalmente, después de haber introducido las técnicas básicas de programación con Prolog, se presentan diversas aplicaciones provenientes de distintos campos de la informática.

De entre ellas, destacamos aquí, por su novedad, las que entroncan con el área de los métodos formales y el desarrollo de software de calidad.

- **Soporte para métodos formales ligeros.**

Por su naturaleza, el análisis y predicción del comportamiento fiable del software cada vez más complejo que demanda la Sociedad de la Información debe descansar en fundamentos sólidos que permitan al usuario de los mismos (ingeniero de requisitos, diseñador o gestor) confiar en la aplicación bajo cualquier circunstancia. El estudio de los lenguajes de programación, como el estudio de la lógica subyacente, se relaciona con la potencia expresiva de las notaciones formales, con correspondencia directa entre la sintaxis (programas) y la semántica (lo que significan), y con los medios mediante los cuales se puede analizar (manual o automáticamente) el texto en un lenguaje formal para extraer conclusiones. Sin embargo, el alto coste que suponen los métodos formales tradicionales han dificultado en el pasado su adopción por la industria del software. Los lenguajes declarativos constituyen un vehículo natural para los llamados *métodos ligeros*, que pueden traer grandes beneficios a un coste reducido. Al final de la obra se incidirá también en la posibilidad real de su aplicación al desarrollo de software fiable y de calidad, proporcionando una introducción a la Ingeniería del Software Automática siguiendo la aproximación lógica.

- **Obra autocontenido.**

El libro no requiere de conocimientos previos en lógica y matemáticas más allá de los impartidos normalmente en los primeros años de la universidad. En cualquier caso, en la primera parte del libro se suministra un resumen de los principales conceptos y resultados de la lógica de predicados, así como en el Apéndice A las notaciones y nociones matemáticas imprescindibles, que son necesarios para seguir sin dificultades el resto de los capítulos.

AUDIENCIA DE LA OBRA

El libro es principalmente un manual universitario, aunque la casi total ausencia de requisitos matemáticos lo hacen también adecuado para el estudio personal de quien desee iniciarse en las relaciones entre lógica y programación.

Especificamente, la obra va dirigida a alumnos de segundo o tercer curso de Ingeniería en Informática que sigan un curso de Programación Declarativa (posiblemente dentro de un curso sobre Lenguajes o Paradigmas de Programación), o más específicamente uno de Programación Lógica. Sirve de base para asignaturas de Lenguaje Natural, Modelos de Razonamiento, Bases de Datos, Ingeniería de Requisitos o Compiladores. También puede ser útil a alumnos de informática que sigan un curso de Inteligencia Artificial y de Métodos Formales en Ingeniería del Software, debido a la posición preeminente que los lenguajes declarativos tienen en estas áreas.

La elección de los temas y el estilo de presentación de la obra es el resultado de veinte años de experiencia de los autores en la impartición de diversos cursos sobre Programación Funcional y Lógica en la U. de Castilla La Mancha y la U. Politécnica de Valencia, respectivamente, en los que se han formado varias generaciones de estudiantes.

ORGANIZACIÓN Y CONTENIDOS

Tras un primer capítulo introductorio, en el que se realiza una aproximación a la programación declarativa, mostrando sus principales aplicaciones y qué ventajas aporta con relación a otras familias de lenguajes, el contenido de este libro se ha organizado como sigue:

- **Parte I. Fundamentos.**

Esta primera parte del libro presenta las bases conceptuales sobre las que se asienta la programación lógica y hace un recorrido selectivo de sus orígenes, en el campo de la demostración automática de teoremas, centrándose en aquellos puntos útiles para su fundamentación.

- **Capítulo 2. Sistemas Formales, Lógica y Lenguajes de Programación.**

En este capítulo se revisan una serie de nociones básicas de la lógica de predicados que son utilizadas de forma recurrente durante todo el libro (sistema formal, sintaxis, cálculo deductivo y semántica) y se ponen en relación con los lenguajes de programación. Se hace ver cómo la caracterización semántica formal es una parte imprescindible de la definición de un lenguaje de programación. Se introducen diversas aproximaciones a la semántica formal de los lenguajes de programación: operacional, axiomática, declarativa (y, dentro de ésta, las semánticas por teoría de modelos, algebraica, de punto

fijo y denotacional). Se trata de hacer notar que la semántica de los programas, en general, y la semántica declarativa en particular, es una manera de abstraer los detalles del proceso de cómputo. El capítulo concluye transmitiendo la idea de que los lenguajes de programación pueden verse como sistemas formales y que, de entre todos ellos, los lenguajes de programación declarativa son los que mejor se adaptan a esta visión.

- **Capítulo 3. De la Demostración Automática a la Programación Lógica (I): introducción y métodos semánticos.**

En éste y en el próximo capítulo se realiza un recorrido por algunos de los conceptos y resultados básicos de la demostración automática de teoremas que sirven de fundamento teórico a la programación lógica. Después de introducir brevemente los objetivos y limitaciones de la demostración automática de teoremas, se discute el papel de los procedimientos de prueba por refutación y de la forma clausal dentro de los mismos. Dado el interés en obtener una representación simplificada para las fórmulas que facilite el proceso de automatización eficiente, se estudian las formas estándares de representar fórmulas, y en particular la forma clausal, y se proporcionan algoritmos para su generación. Por otra parte, se muestra que un conjunto de cláusulas y la fórmula original a la que representa son equivalentes respecto a la insatisfacibilidad. Debido a que en los demostradores automáticos las pruebas se hacen por refutación, este resultado es la principal razón de que la forma clausal tenga un lugar de privilegio en el campo de la demostración automática de teoremas. Finalmente se estudia el teorema de Herbrand y los conceptos asociados: universo de Herbrand; base de Herbrand; interpretaciones de Herbrand; etc. El teorema de Herbrand permite diseñar un algoritmo de prueba por refutación basado en métodos semánticos. Este algoritmo establece la insatisfacibilidad de un conjunto de cláusulas (si realmente lo es). También se estudian las dificultades para mecanizar la lógica cuando se utilizan sistemas de prueba basados en la semántica y se discuten las razones que condujeron a la necesidad de diseñar un método de prueba sintáctico basado en el principio de resolución de Robinson.

- **Capítulo 4. De la Demostración Automática a la Programación Lógica (II): el principio de resolución de Robinson.**

El principio de resolución de Robinson es una regla de inferencia que se aplica a fórmulas en forma clausal y que, junto con el procedimiento de unificación, constituye un cálculo deductivo completo. Para comprender de forma intuitiva el mecanismo operacional de la resolución, se concreta para el caso más simple de la lógica de proposiciones. Esto nos permite obtener una idea precisa de su modo de operación, formalizar el concepto de prueba dentro de este contexto, y ver, de forma sencilla, que es una regla de inferencia muy potente que incorpora otras reglas de inferencia de la lógica. Después se de-

fine el principio de resolución en el marco de la lógica de predicados, para lo que es imprescindible introducir el concepto de sustitución y las nociones asociadas de instancia, composición de sustituciones, unificador y unificador más general. Presentamos el algoritmo de unificación de Martelli y Montanari y estudiamos algunas de sus propiedades formales. Una vez realizado este trabajo preparatorio, se formaliza la regla de resolución y se presentan sus propiedades de corrección y completitud. Se construye un algoritmo de resolución genérico, que no da preferencia a una derivación particular. Se muestra cómo una aplicación sin restricciones del principio de resolución puede generar cláusulas que son redundantes o irrelevantes para los objetivos de la prueba, dando lugar a un espacio de búsqueda muy amplio. Para evitar este problema se introduce el concepto de estrategia, como una limitación en el número de pasos de resolución admisibles con el fin de disminuir el espacio de búsqueda. Mostramos cómo se han diseñado diferentes estrategias de resolución y se estudian las características de algunas de ellas. Las más notables, desde nuestra perspectiva, son las estrategias de resolución lineal y de resolución lineal de entrada. La última está en la base de la estrategia de resolución SLD, que en esencia, constituye el mecanismo operacional del lenguaje Prolog.

- **Parte II. Programación Lógica.**

Esta segunda parte del libro desarrolla los fundamentos teóricos de la programación lógica y describe las características básicas del lenguaje Prolog, así como algunas de carácter avanzado. Se trata de mostrar que la programación lógica puede usarse como un lenguaje de programación efectivo.

- **Capítulo 5. Programación Lógica.**

Siguiendo el discurso del Capítulo 2, que define un lenguaje de programación como un sistema formal compuesto por una sintaxis, una semántica operacional y una semántica declarativa, en este capítulo estudiamos los distintos componentes sintáticos y semánticos que caracterizan un lenguaje de programación lógica. Introducimos la *notación clausal* para programas lógicos y explicamos el mecanismo computacional de la programación lógica, que la convierten en una herramienta viable para la programación: la estrategia de resolución SLD. Definimos los conceptos de derivación SLD, respuesta computada y árbol de búsqueda SLD y describimos algunas de sus propiedades fundamentales. Se estudia la interpretación declarativa de la programación lógica, que se fundamenta en la semántica (teoría de modelos) de la lógica de predicados. Se establece la corrección y completitud del sistema formal basado en la estrategia de resolución SLD y se describen diversas reformulaciones de este resultado. La semántica sirve tanto para asignar significado a las estructuras del lenguaje como a los propios programas, de manera que pueda establecerse un criterio formal de equivalencia

entre programas. Se estudian las diferentes caracterizaciones del significado de los programas lógicos, que explican “qué” computan los programas lógicos abstrayéndose de los detalles operacionales. Terminamos con un inciso sobre semánticas no estándar: las semánticas composicionales y la semántica de respuestas computadas.

- **Capítulo 6. El Lenguaje Prolog: Introducción.**

El objetivo de este capítulo es describir las principales características del lenguaje Prolog que deben tenerse en cuenta para su correcto aprendizaje. Primero, presentamos un subconjunto del lenguaje denominado Prolog puro, describimos su sintaxis y su mecanismo operacional (una adaptación del principio de resolución SLD). A través de sencillos ejemplos de programación sobre diferentes dominios (números naturales, listas, conjuntos) introducimos las técnicas de programación más comunes del paradigma de programación lógica: la recursión como alternativa a las técnicas iterativas; la definición de propiedades por inducción estructural y el uso de la unificación, como único y potente medio de enlazar valores a las variables. Mostramos que incluso este pequeño subconjunto puede apartarse en ciertos aspectos del ideal expresado por la teoría de la programación lógica. Estudiamos también estructuras más complejas (como los árboles o los grafos —estos últimos para especificar un autómata—) y un subconjunto mayor del lenguaje Prolog que nos permite tratar con la aritmética y el uso de operadores definidos por el usuario.

- **Capítulo 7. El Lenguaje Prolog: Aspectos Avanzados.**

En este capítulo continuamos nuestro estudio del lenguaje Prolog discutiendo algunas de las características extralógicas, que hacen posible utilizarlo en la práctica, y ciertos aspectos avanzados del lenguaje. Contrariamente a los predicados de Prolog puro, que se definen mediante reglas, los predicados extralógicos realizan tareas que no pueden especificarse fácilmente mediante fórmulas lógicas. Esencialmente, existen tres clases de estos predicados: i) Predicados que facilitan el control del modo de ejecución de un programa Prolog (e.g., el operador de corte); ii) Predicados para la gestión de la base de datos interna del sistema Prolog (e.g., `assert`, `retract`); iii) Predicados del sistema, que interaccionan con el sistema operativo (e.g., los operadores de entrada/salida). Se ilustra el empleo de estos predicados mediante una serie de programas y mostramos, a través de diferentes ejemplos, cómo los predicados extralógicos pueden producir efectos laterales y destruir la declaratividad de los programas. Entre otras aplicaciones se emplea el operador de corte y el predicado `fail` para simular ciertas estructuras de control de los lenguajes convencionales y se discute cuándo pueden ser útiles. Hacemos especial énfasis en el tratamiento de la negación y nos centramos en la regla de *negación como fallo* y su implementación en el lenguaje Prolog,

discutiendo sus peculiaridades. Entre los aspectos avanzados se estudian: la metaprogramación y el orden superior; la programación no determinista y el uso de estructuras avanzadas (como listas diferencia y diccionarios) para incrementar la eficiencia de los programas. También se proporcionan algunos consejos/criterios que pueden ahorrar tiempo de ejecución y memoria empleada en los programas Prolog.

■ **Parte III. Aplicaciones de la Programación Lógica.**

En esta tercera y última parte del libro se desarrollan algunas de las aplicaciones de la programación lógica. En la selección de las aplicaciones concretas nos ha guiado, entre otros criterios, el valor intrínseco que éstas puedan tener como fuente para la introducción de técnicas estándares de programación. Las dos primeras aplicaciones de la programación lógica que presentamos corresponden al área de la inteligencia artificial. La última proviene del área de la ingeniería del software y los lenguajes de programación.

• **Capítulo 8. Representación del conocimiento.**

En este capítulo se aborda de manera sistemática el problema de la representación del conocimiento y se pone de relieve el papel de los lenguajes de programación lógica como herramientas de representación del conocimiento. Primero se discuten algunos de los problemas teóricos y, seguidamente, se estudia cómo se puede emplear la lógica de predicados como vehículo para la representación del conocimiento. Se introduce un método sistemático que permite pasar desde una formulación de un problema, expresada en lenguaje natural, a un programa lógico. Dado el papel tan importante que juega la forma clausal en la programación lógica, se estudian algunas peculiaridades de la representación del conocimiento mediante cláusulas: se pone en relación la cuantificación existencial con la aparición de variables extra en las cláusulas, la negación y la respuesta a preguntas; se estudia como representar los conjuntos (y de una forma simple los tipos de datos) como individuos y el papel de las relaciones “instancia” y “es_un” en dicha representación; también se trata el tema de la representación mediante relaciones binarias. Desde un punto de vista más práctico y circunscritos al lenguaje Prolog, también se dan una serie de consejos que pueden ayudar a establecer la elección de una buena estructura de datos, ya que ésta puede ser determinante a la hora de representar el conocimiento de forma eficiente. Se discuten las ventajas e inconvenientes de la elección de una determinada estructura de datos. Finalmente, se relaciona la representación en forma clausal con otras técnicas de representación del conocimiento, como son las redes semánticas y los marcos.

• **Capítulo 9. Resolución de problemas.**

En este capítulo se presenta el método de resolución de problemas mediante la técnica de búsqueda en un espacio de estados. La búsqueda es un pro-

ceso de gran importancia en la resolución de problemas difíciles, para los que no se dispone de técnicas más directas. En este contexto, un problema puede modelarse mediante un grafo dirigido cuyos nodos son estados (que abstraen las características del problema) y los arcos representan operaciones (de cambio de estado). La solución a un problema se obtiene mediante la búsqueda de un camino que conduce desde un estado inicial a un estado objetivo, que se considera una solución aceptable del problema. Por consiguiente, el estudio de las estrategias de control para la búsqueda de caminos óptimos constituyen un apartado importante dentro de este capítulo. Después de describir las características esenciales de esta técnica, se utiliza para proporcionar solución a una serie de problemas clásicos (el mono y el plátano, contenedores de agua y generación de planes en el mundo de los bloques) que se abordan desde una perspectiva puramente declarativa, dejando el control al mecanismo de búsqueda automático del lenguaje. Finalmente, se muestra que, si se desea implementar soluciones prácticas de problemas más complejos, se hace imprescindible especificar también la estrategia de búsqueda. Se estudian las principales estrategias de búsqueda sin información (búsqueda en anchura y profundidad), así como algunas estrategias de búsqueda heurística, que utilizan información específica del dominio del problema. Se proporcionan programas Prolog que implementan dichas estrategias y que pueden servir de base para incrustar otros mecanismos de resolución de problemas.

• **Capítulo 10 Programación Lógica y Tecnología Software Rigurosa.**

Siguiendo un enfoque moderno, que tiene en cuenta los tres elementos de la trilogía del software —programas, datos y propiedades—, se estudian los procesos formales que transforman dichas componentes automáticamente. Esto incluye, entre otros, los siguientes mecanismos: análisis y síntesis, transformación, evaluación parcial, depuración, certificación y verificación automática de programas. En este capítulo se pretende mostrar la potencia de la tecnología declarativa como medio tangible para acercarse a los objetivos de calidad, fiabilidad y seguridad de los productos software, en contraste con otros métodos formales más convencionales (y a la vez poco prácticos), que fomentan la formalización excesiva mediante el empleo de nociones que requieren una formación matemática poco habitual en los usuarios finales.

Se ha dedicado el Apéndice A a la introducción de un limitado número de notaciones y nociones matemáticas que es conveniente que el lector conozca. En él se introducen nociones tan fundamentales como las de conjunto, función o relación, así como estructuras abstractas como los grafos y los árboles. Este apéndice puede usarse como un manual de referencia rápido al que dirigirse solo cuando se necesite.

Para finalizar, diremos que los contenidos enumerados se adaptan a una asignatura cuatrimestral (impartida en 15 semanas, con tres horas de teoría, una de problemas y

dos de prácticas de laboratorio por semana). Dichos contenidos pueden extenderse hasta cubrir un semestre, dependiendo de la profundidad y el detalle con el que se expliquen los mismos.

AGRADECIMIENTOS

Agradecemos desde estas líneas a los compañeros con quienes hemos compartido en estos años la docencia de las asignaturas sobre programación declarativa y que, de una u otra forma, han influido en la redacción de este manuscrito. Un agradecimiento especial para María José Ramírez, que ha colaborado en la preparación de numerosos exámenes que han servido de base para la elaboración de algunos de los ejercicios que se proponen al final de cada capítulo. Así mismo, expresamos nuestro agradecimiento a Person-Educación por el interés y la confianza mostradas en la edición de nuestro libro.

Finalmente, dedicamos el presente libro a nuestras respectivas familias, que nunca han usado un lenguaje lógico pero sin cuya ayuda no habría sido posible escribir este libro.

Sagunto, Navidad de 2006

Una Panorámica de la Programación Declarativa

En este capítulo una breve introducción de la programación declarativa, incluyendo sus orígenes, sus principales aplicaciones y qué ventajas aporta con relación a otras familias de lenguajes.

1.1 COMPUTADORES Y LENGUAJES DE PROGRAMACIÓN

Aunque existen otros modelos de organización de los computadores, la mayoría de los computadores modernos presentan una organización basada en el llamado modelo de von Neumann¹. En este apartado pondremos de manifiesto que los lenguajes convencionales son una abstracción de alto nivel de la estructura de la máquina para la que se han desarrollado y que muchos de sus defectos e imperfecciones provienen de esta estrecha relación. Al hilo de esta discusión revisaremos muchas de las características de esta clase de lenguajes.

1.1.1. Organización de los computadores

Los computadores que siguen el modelo de organización de von Neumann (o máquina con *programa almacenado*) están constituidos por los siguientes componentes internos: i) una unidad central de proceso (UCP); ii) una memoria central (MC), y iii) procesadores de dispositivos periféricos, que podrán gestionar uno o varios dispositivos del mismo tipo. Todos los componentes están conectados por un *bus* único, que es un conjunto de conductores paralelos a través de los cuales se transmite información entre

¹Propuesta por el matemático americano J. von Neumann, a mediados de los años cuarenta.

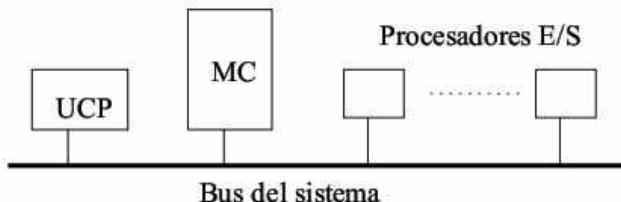


Figura 1.1 Organización de un computador genérico.

los diferentes componentes, en forma de impulsos eléctricos. La Figura 1.1 muestra un esquema simplificado de este tipo de organización. En lo que aquí respecta nos interesan las características de los dos primeros componentes y del bus del sistema.

Habitualmente, la UCP está dividida en cuatro componentes funcionales: i) Los registros; ii) la unidad de control (UC); iii) la unidad aritmético lógica (UAL); y iv) un reloj o contador de impulsos. Estos componentes están unidos por buses internos (no confundir con el bus del sistema) y líneas de control. Podemos decir que existe una correspondencia entre los dos componentes principales de la UCP —la UC y la UAL— y el tipo de informaciones que almacena la MC —instrucciones y datos—. La UC trata las instrucciones y la ALU los datos.

Los dos registros más importantes son el contador de programa (CP), que apunta a la palabra de memoria que contiene la siguiente instrucción a ejecutar, y el registro de instrucción (RI), que almacena la instrucción que se está ejecutando. En el RI deben distinguirse dos partes, el código de operación y la parte de los operandos. Existen otros registros de propósito general (i.e., acumuladores, registros de segmento, registros índice, etc.), que son utilizados para almacenar resultados intermedios. El tamaño de los registros suele ser igual al de una palabra de memoria.

Cuando un programa con sus datos está almacenado en la MC, la UC rige el comportamiento del computador mediante la interpretación y secuenciamiento de las instrucciones que lee de la memoria.

1. La interpretación consiste en:

- a) Fase de búsqueda: La instrucción contenida en la zona de memoria determinada por el CP se lleva al RI.
- b) Fase de ejecución: La UC envía las microórdenes precisas a las distintas partes del computador para que se lleve a cabo la operación descrita por la instrucción.

A menudo la fase de ejecución se subdivide. Una subdivisión típica establece las etapas de decodificación (determina el tipo de instrucción), acceso a los operandos (habitualmente se llevan desde la memoria o desde el banco de registros a la

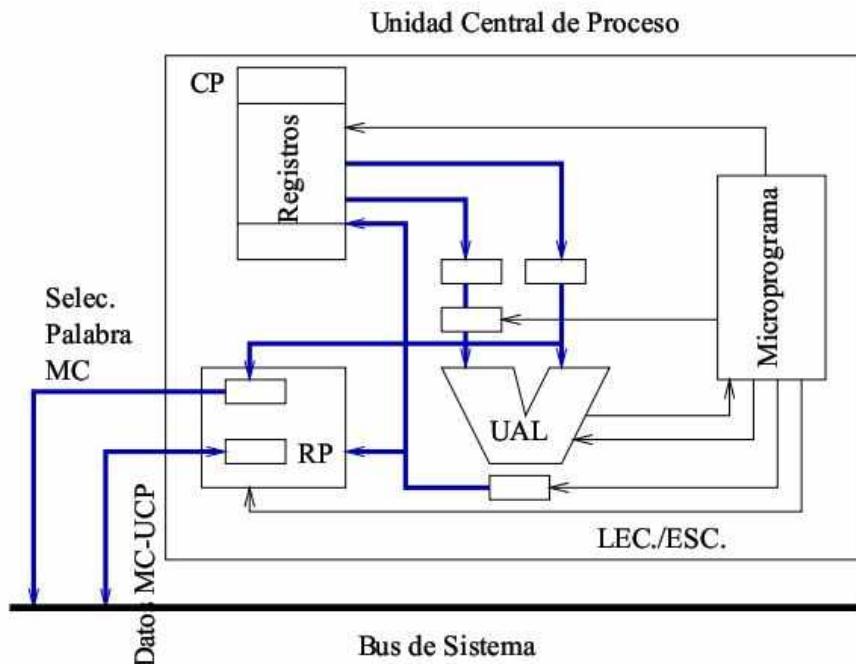


Figura 1.2 Un ejemplo de procesador y la MC.

entrada de la UAL), ejecución propiamente dicha y almacenamiento de resultados (en memoria central o en registros).

2. El secuenciamiento consiste en escribir en el registro CP la dirección a partir de la cual se encuentra la siguiente instrucción que debe procesarse.

La UAL tiene por función realizar operaciones. La mayoría de las UAL realizan un número muy limitado de operaciones a nivel de bits: lógicas, como AND, OR, y NOT; aritméticas como ADD; y desplazamientos. El resto de las operaciones e instrucciones máquina son microprogramadas (aunque en ciertas máquinas, por ejemplo las RISC, todas las instrucciones están cableadas).

La propiedad esencial de la memoria que nos interesa es su capacidad de ser *direccionalizada*. La MC puede considerarse como un conjunto de celdas (llamadas palabras), cada una de ellas con la capacidad de almacenar una información: dato o instrucción. Las celdas pueden considerarse numeradas y la UC conoce cada celda por su número, llamado dirección. Por esta razón hablamos en ocasiones de Memoria Direccionalable. La UC puede pedir LEER el contenido de una celda de una dirección determinada o ESCRIBIR una nueva información en una celda de dirección determinada. Como resultado de estas operaciones, la información se transmitirá a través del bus del sistema entre la MC y la UCP.

El bus del sistema es un conjunto de conductores paralelos a través de los cuales

transcurren una serie de señales eléctricas que se intercambian los componentes del sistema. Las líneas que forman el bus del sistema pueden clasificarse, según su función, en tres tipos: (1) dirección; (2) datos; y (3) control. Aunque la organización en torno a un bus único es la menos costosa de las organizaciones tiene, entre otras, la limitación de obligar a que la principal tarea de un programa consista en cambiar los contenidos de la MC recibiendo y enviando palabras individuales a través del bus desde y hacia la MC o la UCP. Debido a estas dificultades, John Backus ha denominado al bus del sistema *cuello de botella de von Neumann* [8]. Una dificultad adicional es que una gran parte de ese tráfico no es útil para el cómputo en sí, ya transfiere las direcciones donde se encuentran las instrucciones y los datos a procesar.

El modelo de organización y funcionamiento de los computadores que acabamos de describir conduce a un modelo de cómputo en el que los procesadores ejecutan las instrucciones de un modo puramente secuencial, una instrucción detrás de otra, siguiendo el orden en el que han sido almacenadas en la MC (rompiéndose ese orden solo cuando se ejecuta una instrucción de salto, JUMP, bien condicional o incondicional). Otra característica de este modelo de cómputo es que realiza una completa separación entre el tipo de informaciones que almacena la MC, dividiéndolas en instrucciones y datos, lo que tiene una posterior repercusión sobre el diseño de los lenguajes convencionales que se apoyan en el modelo de von Neumann. De hecho, el que tanto los datos como las instrucciones puedan almacenarse en la memoria es una de las principales características que distinguen el computador universal moderno de otras máquinas de calcular anteriores, que solo podían “programarse a mano”, por métodos mecánicos o eléctricos según la tecnología, para implementar cada función o rutina de interés. La idea de que las funciones pueden digitalizarse igual que los datos (en definitiva una función se puede describir como una secuencia de caracteres) permite su almacenamiento en la memoria y, a partir de este momento, no hay distinción esencial entre datos y programas, lo que se conoce a veces como *principio de dualidad*.

Los lenguajes convencionales han explotado la dualidad dato-programa con un marcado sesgo hacia los datos: los programas son simplemente un tipo de datos particular representado en la memoria de la máquina; esto permite ver los programas como datos de entrada de otros programas, por ejemplo un compilador. Sin embargo, podríamos decir que los lenguajes declarativos explotan la dualidad de manera mucho más amplia. En el lambda-cálculo, una lógica ecuacional de orden superior que sirve de modelo formal para muchos lenguajes lenguajes funcionales, una constante numérica se ve como una función constante que retorna dicho valor. Y en muchos lenguajes funcionales, de hecho, funciones y datos tienen el mismo aspecto: todo son listas. Las funciones son entonces *ciudadanos de primera clase* y se pueden pasar como argumento de otras funciones e incluso devolverse como resultado de la ejecución de otras funciones. Asimismo en los lenguajes lógicos, datos y programas tienen también la misma representación (todo son *estructuras*) y esto permite modificar en tiempo de ejecución el propio programa que se está ejecutando. Es decir la dualidad no solo se usa para ejecutar datos como programas sino para modificar también programas como datos.

También, el uso por parte de la UCP de un conjunto de registros para realizar sus operaciones, y la propia visión de la MC como un conjunto de celdas en las que se almacenan datos, introduce el concepto de *estado de la computación*.

1.1.2. Características de los lenguajes convencionales

Los lenguajes convencionales (o *imperativos*) están inspirados en la arquitectura de von Neumann. De hecho, los distintos recursos expresivos proporcionados por tales lenguajes pueden verse como abstracciones de los componentes de la máquina de von Neumann o como abstracciones de las operaciones elementales que toda máquina de von Neumann incorpora. Estos lenguajes utilizan las variables imitando las celdas de la MC, las instrucciones de control que generalizan las instrucciones de salto condicional o incondicional del lenguaje máquina y la instrucción de asignación que engloba las instrucciones de carga (LOAD) y almacenamiento (STORE) del lenguaje máquina y también las de movimiento (MOVE). Así pues, los lenguajes de programación convencionales, independientemente del número de niveles software existentes entre ellos y el lenguaje máquina, son en esencia una extensión del lenguaje máquina.

La instrucción de asignación resulta ser representativa del cuello de botella de von Neumann y nos obliga a pensar en términos de trasiego de información entre celdas de memoria. Más todavía, Backus [8] ha hecho notar que la instrucción de asignación separa la programación en dos mundos. El primero comprende la parte derecha de las instrucciones de asignación. Este es un mundo de *expresiones*, con propiedades algebraicas muy útiles en el que se realiza la mayor parte del cómputo. El segundo es el mundo de las instrucciones, en el que la propia instrucción de asignación puede considerarse una construcción primitiva en la que se basan las demás. Este es un mundo desordenado, con pocas propiedades matemáticas útiles.

Como ya puso de manifiesto Kowalski [81, 83] y después de él otros autores (e.g. [6] y [124]), podemos distinguir dos aspectos fundamentales en las tareas de programación:

- Aspectos *lógicos*: Esto es, *¿Qué debe computarse?*. Esta es la cuestión esencial y, de hecho, es la que motiva el uso de un ordenador como medio para resolver un determinado problema.
- Aspectos de *control*, entre los que podemos distinguir:
 - Organización de la secuencia de cómputos en pequeños pasos.
 - Gestión de la memoria durante la computación.

Dado que dichos aspectos lógicos y de control se refieren a componentes claramente distintos e independientes de las tareas de programación (*especificación* del problema e *implementación* del programa que lo resuelve, respectivamente), parece deseable que los lenguajes de programación nos permitan mantener las distancias entre ambos, sin necesidad de que las cuestiones implicadas en las tareas de especificación e implementación

interfieran entre sí [124]. Atendiendo a este criterio de independencia de los aspectos lógicos y de control, podemos elevar algunas críticas sobre el uso de los lenguajes imperativos.

1. La presencia de instrucciones de control de flujo (sentencias `if`, `while`, secuenciación, etc.) y las operaciones de gestión de memoria (distinción entre declaración y definición de una estructura o variable del programa; necesidad de reserva y liberación explícita de la memoria; uso de punteros), oscurecen el contenido lógico (puramente descriptivo) del programa.
2. En el ámbito matemático, un símbolo de variable siempre denota el mismo objeto, cualquiera que sea la ocurrencia de ese símbolo que consideremos. Sin embargo, la operación de asignación utiliza las variables de modo matemáticamente impuro.

Por ejemplo, consideremos la siguiente instrucción de asignación: `x := x + 1`. La ocurrencia de la variable `x` en la parte derecha de la asignación no denota lo mismo que la ocurrencia de la parte izquierda. La anterior instrucción debe entenderse en términos de la siguiente orden: “almacena, en la palabra a cuya dirección de memoria asociamos el identificador “`x`”, el resultado de sumar 1 al contenido de la palabra referenciada por dicho identificador `x`”. En general, la parte izquierda de una instrucción de asignación debe entenderse como una dirección de palabra mientras que la parte derecha como una expresión que se evaluará para almacenar el resultado en la palabra cuya dirección es “`x`”.

Este uso no matemático de las variables, unido al hecho de que la instrucción de asignación es sensible a la historia de la computación, dificulta el razonamiento sobre las propiedades de los programas. En general, no es suficiente con conocer el significado de los constituyentes del programa para establecer su significado sino que, en realidad, ese significado depende del contexto donde aparecen dichos constituyentes y de la historia previa de la computación, que permite la introducción de efectos laterales (es decir, el significado del programa no podrá entenderse en términos de composicionalidad).

3. Dejar las cuestiones de control al programador no parece lo más indicado. Por ejemplo, un programa escrito en Pascal para una máquina de un único procesador desaprovechará los recursos ofrecidos por una máquina con muchos procesadores. A menudo será difícil para el compilador generar un código que aproveche al máximo las características de la máquina objeto, puesto que parte del control (secuencial) estará ya especificado por el programador en el texto del programa [42, 120].

Concretamos algunos de los defectos de un lenguaje de programación imperativa mediante el estudio del siguiente ejemplo:

Ejemplo 1.1

Consideremos el conocido problema de la concatenación de dos listas. Si decidimos resolverlo mediante el uso de un lenguaje convencional como el lenguaje C es necesario dotarse de una representación explícita de las listas en la memoria del computador, además de un conjunto de operaciones que las manipulen. Una posible solución al problema viene dada por el siguiente programa:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct nodo
{
    char dato;
    struct nodo *enlace;
} LISTA;
void mostrar(LISTA *ptr);
void insertar(LISTA **ptr, char elemento);
LISTA *crear_lista();
LISTA *concatenar(LISTA *ptr1, LISTA *ptr2);
void main()
{
    LISTA *l1, *l2, *lis = NULL;
    l1 = crear_lista();
    l2 = crear_lista();
    lis = concatenar(l1, l2);
    printf("\n La nueva lista enlazada es: ");
    mostrar(lis);
}
void mostrar(LISTA *ptr)
{
    while(ptr != NULL)
    {
        printf("%c",ptr->dato);
        ptr = ptr->enlace;
    }
    printf("\n");
}
void insertar(LISTA **ptr, char elemento)
{
    LISTA *p1, *p2;
    p1 = *ptr;
    if(p1 == NULL)
    {
        p1 = malloc(sizeof(LISTA));
        if (p1 != NULL)
        {
            p1->dato = elemento;
            p1->enlace = NULL;
            *ptr = p1;
        }
    }
}
```

```
    }
else
{
    while(p1->enlace != NULL) p1 = p1->enlace;
    p2 = malloc(sizeof(LISTA));
    if(p2 != NULL)
    {
        p2->dato = elemento;
        p2->enlace = NULL;
        p1->enlace = p2;
    }
}
LISTA *crear_lista()
{
    LISTA *lis = NULL;
    char elemento;
    printf("\n Introduzca elementos: ");
    do
    {
        elemento = getchar();
        if(elemento != '\n') insertar(&lis, elemento);
    } while(elemento != '\n');
    return lis;
}
LISTA *concatenar(LISTA *ptr1, LISTA *ptr2)
{
    LISTA *p1;
    p1 = ptr1;
    while(p1->enlace != NULL) p1 = p1->enlace;
    p1->enlace = ptr2;
    return ptr1;
}
```

Este programa ilustra y concreta algunas de las propiedades de los lenguajes convencionales ya mencionadas junto a otras nuevas:

1. La secuencia de instrucciones que constituyen el programa son órdenes a la máquina que operan sobre un estado declarado no explícitamente. Ésta es la razón por la que esta clase de lenguajes se denominan *lenguajes imperativos*.
2. Para entender el programa debemos ejecutarlo mentalmente siguiendo el modelo de computación de von Neumann, estudiando cómo cambian los contenidos de las variables y otras estructuras en la MC.
3. Es necesario gestionar explícitamente la MC, lo que conduce al uso de la llamada al sistema `malloc()` para la reserva de MC, y al empleo de variables de tipo puntero, de semántica y sintaxis confusa y, por lo tanto, de difícil uso. La manipulación de punteros es una de las fuentes de errores más comunes en la programación con este tipo de lenguajes.

4. El programa acusa una falta de generalidad debido a la rigidez del lenguaje a la hora de definir nuevos tipos de datos. Por ejemplo, este programa solo puede usarse para concatenar listas de caracteres. Si quisiésemos concatenar otros tipos de listas deberíamos rehacer el programa.
5. La lógica y el control están mezclados, lo que dificulta la verificación formal del programa.

En este apartado hemos puesto de manifiesto los puntos débiles de los lenguajes convencionales, que justifican el estudio de otros paradigmas de programación. Sin embargo, el lector debe ser consciente de que reunen otras muchas ventajas que han hecho de ellos los lenguajes preferidos de los programadores en amplias áreas de aplicación (e.g., computación numérica, gestión y tratamiento de la información, programación de sistemas). Entre estas ventajas podríamos citar: eficiencia en la ejecución; modularidad; herramientas para la compilación separada; herramientas para la depuración de errores. En el Apartado 1.3 ampliaremos la información sobre las ventajas y el área de aplicación de estos lenguajes.

1.2 PROGRAMACIÓN DECLARATIVA

La *programación declarativa* (a veces llamada *programación inferencial*) puede entenderse como un estilo de programación en el que el programador especifica *qué* debe computarse más bien que *cómo* deben realizarse los cómputos. En este paradigma de programación, de acuerdo con el famoso aserto de Kowalski [82, 81, 83], un *programa* = *lógica* + *control*, y la tarea de programar consiste en centrar la atención en la *lógica* dejando de lado el *control*, que se asume automático, al sistema. El componente lógico determina el significado del programa mientras que el componente de control solamente afecta a su eficiencia. Esta distinción tiene la ventaja de que la eficiencia de un programa puede mejorarse modificando el componente de control sin tener que modificar la lógica del algoritmo (del que el programa es una representación posible en la máquina). Con más precisión, la característica fundamental de la programación declarativa es el uso de la lógica como lenguaje de programación, lo cual puede conceptualizarse como sigue:

- Un *programa* es una teoría formal en una cierta lógica, esto es, un conjunto de fórmulas lógicas que resultan ser la especificación del problema que se pretende resolver, y
- la *computación* se entiende como una forma de inferencia o deducción en dicha lógica.

Los principales requisitos que debe cumplir la lógica empleada son: i) disponer de un lenguaje que sea suficientemente expresivo para cubrir un campo de aplicación interesante; ii) disponer de una semántica operacional, esto es, un mecanismo de cómputo que

permite ejecutar los programas; iii) disponer de una semántica declarativa que permita dar un significado a los programas de forma independiente a su posible ejecución; y iv) resultados de corrección y completitud que aseguren que lo que se computa coincide con aquello que es considerado como verdadero (de acuerdo con la noción de verdad que sirve de base a la semántica declarativa). Desde el punto de vista del soporte a la declaratividad, el tercero de los requisitos es, tal vez, el más importante, ya que es el que permite especificar *qué* estamos computando. Concretamente, la semántica declarativa especifica el significado de los objetos sintácticos del lenguaje por medio de su traducción en elementos y estructuras de un dominio (generalmente matemático) conocido. Según la clase de lógica² que empleemos como fundamento del lenguaje declarativo obtenemos los diferentes estilos y características para esta clase de lenguajes (véase la Tabla 1.1).

Tabla 1.1 Estilos de Programación declarativa y características heredadas de la lógica que la fundamenta.

Clase de lógica	Estilo/Característica
Ecuacional	Funcional
Clausal	Relacional
Heterogénea	Tipos
Géneros ordenados	Herencia
Temporal	Concurrencia

La programación declarativa incluye como paradigmas más representativos la programación lógica como la funcional, cuyas principales características se resumen en los dos próximos apartados.

1.2.1. Programación Lógica

La *programación lógica* ([7, 82, 86]) se basa en fragmentos de la lógica de predicados, siendo el más popular la lógica de *cláusulas de Horn* (HCL, del inglés *Horn clause logic*), que pueden emplearse como base para un lenguaje de programación al poseer una semántica operacional susceptible de una implementación eficiente, como es el caso de la *resolución SLD* (c.f. Section 5.2). Como semántica declarativa se utiliza una semántica por *teoría de modelos* que toma como dominio de interpretación un universo de discurso puramente sintáctico: el *universo de Herbrand*. La *resolución SLD* es un método de prueba por refutación que emplea el algoritmo de *unificación* como mecanismo de base y permite la extracción de respuestas (i.e., el enlace de un valor a una *variable lógica*). La resolución SLD es un método de prueba correcto y completo para la lógica HCL.

²Algunas de estas clases son restricciones de la lógica de primer orden, como la lógica clausal o la lógica ecuacional

Algunas de las características de la programación lógica pueden ponerse de manifiesto mediante un sencillo ejemplo.

Ejemplo 1.2

Consideremos, de nuevo, el problema de la concatenación de dos listas del Ejemplo 1.1. En programación lógica el problema se resuelve definiendo una relación *app* con tres argumentos: generalmente, los dos primeros hacen referencia a las listas que se desea concatenar y el tercero a la lista que resulta de la concatenación de las dos primeras. El programa se compone de dos cláusulas de Horn³.

$$\begin{array}{l} app([], X, X) \\ app([X|X_s], Y_s, [X|Z_s]) \leftarrow app(X_s, Y_s, Z_s) \end{array}$$

Este programa tiene una lectura declarativa clara. La primera cláusula afirma que

la concatenación de la lista vacía [] y otra lista *X* es la propia lista *X*.

mientras que la segunda cláusula puede entenderse en los siguientes términos.

La concatenación de dos listas $[X|X_s]$ e Y_s es la lista que resulta de añadir el primer elemento *X* de la lista $[X|X_s]$ a la lista Z_s que se obtiene al concatenar el resto X_s de la primera lista a la segunda lista Y_s .

Uno de los primeros hechos que advertimos en el Ejemplo 1.2 es que no hay ninguna referencia explícita al tipo de representación en memoria de la estructura de datos lista. De hecho, una característica de los lenguajes declarativos es que proporcionan una *gestión automática de la memoria*, evitando una de las mayores fuentes de errores en la programación con otros lenguajes (piénsese en el manejo de punteros cuando se programa con el lenguaje C, ilustrado por el Ejemplo 1.1). Por otra parte, el programa puede responder, haciendo uso del mecanismo de resolución SLD, a diferentes cuestiones (*objetivos*) sin necesidad de efectuar ningún cambio en el programa, gracias al hecho de emplearse un mecanismo de cómputo que permite una *búsqueda indeterminista* (*don't know built-in search*) de soluciones. Esta misma característica permite computar con *datos parcialmente definidos* y hace posible que la *relación de entrada/salida* no esté fijada de antemano. Por ejemplo, el programa sirve para responder tanto preguntas como “El resultado de concatenar las listas [2, 4, 6] y [1, 3, 5, 7] ¿es la lista [2, 4, 6, 1, 3, 5, 7]?” o “¿Cuál es el resultado de concatenar las listas [2, 4, 6] y [1, 3, 5, 7]?", como “¿Qué listas dan como resultado de su concatenación la lista [2, 4, 6, 1, 3, 5, 7]?” La primera de estas preguntas se formaliza en lenguaje clausal mediante el objetivo $\leftarrow app([2, 4, 6], [1, 3, 5, 7], [2, 4, 6, 1, 3, 5, 7])$ y se responde con

³Se hace uso de la notación que emplea el lenguaje Prolog para la representación de listas, en la que las variables se escriben con mayúsculas, el símbolo “[]” representa la lista vacía y el operador “[|]” es el constructor de listas.

verdadero. La segunda pregunta se formaliza como $\leftarrow app([2, 4, 6], [1, 3, 5, 7], Z)$ y se responde devolviendo el enlace de un valor a una variable $\{Z = [2, 4, 6, 1, 3, 5, 7]\}$; aquí la variable Z se utiliza con un sentido puramente matemático, esto es, una vez establecido el enlace, el valor de la variable queda fijado, contrariamente a lo que sucede con los lenguajes imperativos que permiten el empleo de la *asignación destructiva*. Por último, la tercera de estas preguntas queda formalizada mediante el objetivo $\leftarrow app(X, Y, [2, 4, 6, 1, 3, 5, 7])$, que da lugar a una serie de posibles respuestas: $\{X = [], Y = [2, 4, 6, 1, 3, 5, 7]\}; \{X = [2], Y = [4, 6, 1, 3, 5, 7]\}; \{X = [2, 4], Y = [6, 1, 3, 5, 7]\}; \dots$ Este último ejemplo pone de manifiesto las ventajas de un mecanismo de cómputo que permite la búsqueda indeterminista de soluciones. Éstas y otras características de los lenguajes de programación lógica se resumen en la columna izquierda de la Tabla 1.2. Completaremos la explicación de la columna derecha de esta tabla en la próxima sección.

Tabla 1.2 Características de la programación lógica y de la programación funcional:
Diferencias esenciales.

Programación Lógica	Programación Funcional
Programa: Conjunto de cláusulas que definen relaciones	Programa: Conjunto de ecuaciones que definen funciones.
Semántica operacional:	Semántica operacional:
Resolución SLD (unificación)	Reducción (ajuste de patrones)
Semántica declarativa:	Semántica declarativa:
Teoría de modelos (Mod. mínimo)	Algebraica (Mod. inicial) / Denotacional
Primer orden	Orden superior
Uso múltiple de un mismo procedimiento	Uso único de cada función
Indeterminismo	Determinismo
E/S adireccional	E/S direccional y transparencia referencial
Variables lógicas	Sin variables lógicas
Sin tipos	Tipos y polimorfismo
Datos parcialmente especificados	Datos completamente definidos
No estructuras infinitas	Estructuras potencialmente infinitas
No evaluación perezosa	Evaluación perezosa

1.2.2. Programación Funcional

Los *lenguajes funcionales* están enraizados en el concepto de *función* (matemática) y su definición mediante ecuaciones (generalmente recursivas), que constituyen el programa. Desde el punto de vista computacional, la programación funcional se centra en la evaluación de expresiones (funcionales) para obtener un *valor*.

Ejemplo 1.3

Utilizando un lenguaje de programación funcional, el problema del Ejemplo 1.2 se resuelve definiendo una función *app* de dos argumentos, que representan las listas que se desea concatenar, y que devuelve como resultado la concatenación de dichas listas. El programa consiste en una declaración de tipos, una declaración de la función *app*, en la que se fija su dominio y su rango, y dos ecuaciones que la definen⁴.

$$\begin{array}{ll} \text{data } [t] = & [] \mid (t : [t]) \\ \text{app} :: & [t] \rightarrow [t] \rightarrow [t] \\ \\ \text{app } [] x = & x \\ \text{app } (x : x_s) y_s = & x : (\text{app } x_s y_s) \end{array}$$

Un primer rasgo a destacar en el programa del Ejemplo 1.3 es la necesidad de emplear recursos expresivos para fijar el perfil de la función, i.e., la naturaleza del dominio y el rango de la función. En programación funcional la descripción de dominios conduce a la idea de *tipo de datos*. Los tipos de datos se introducen en programación para evitar o detectar errores y ayudar a definir estructuras de datos. Los lenguajes funcionales modernos son lenguajes *fuertemente basados en tipos* (*strongly typed*), ya que realizan una comprobación de las expresiones que aparecen en el programa para asegurarse de que no se producirán errores durante la ejecución del mismo por incompatibilidades de tipo. No obstante, suelen incorporar también un mecanismo automático de inferencia de tipos que libera al programador de la necesidad de declarar éstos ya que son deducidos por el intérprete.

También debemos reparar en que se ha definido la estructura de datos lista, mediante una declaración de datos, donde los símbolos “[]” y “:” son los *constructores del tipo* y el símbolo *t* es una *variable de tipo*, de forma que podemos formar listas de diferentes tipos de datos. Esta facultad se denomina *polimorfismo* y es otro rasgo muy apreciado en los lenguajes funcionales modernos⁵.

Lo que caracteriza a una función (matemática), además de su perfil, es que a cada elemento de su dominio le corresponde un único elemento del rango; en otras palabras, el resultado (*salida*) de aplicar una función sobre sus argumentos viene determinado exclusivamente por el valor de éstos (su *entrada*). Esta propiedad de las funciones (matemáticas) se denomina *transparencia referencial*. Debido a que los programas en los lenguajes imperativos mantienen un *estado* interno del cómputo, que es modificado durante la ejecución de la secuencia de instrucciones, las construcciones funcionales de estos lenguajes pueden presentar *efectos laterales* (*side effects*), incumpliendo la propie-

⁴Se hace uso de la sintaxis del lenguaje funcional Haskell, en la que el símbolo “::” representa el constructor de listas.

⁵Haskell, además de poseer un sistema de tipos algebraicos polimórficos, permite la declaración de *tipos abstractos de datos* mediante la declaraciones *export / import*, que controla el ocultamiento de la información y propicia la modularidad y la seguridad.

dad de transparencia referencial y no denotando, por consiguiente, verdaderas funciones (matemáticas). La transparencia referencial permite el estilo de la programación funcional basado en el razonamiento ecuacional (la substitución de “iguales por iguales”, es decir, de una expresión por otra “equivalente”) y los cálculos deterministas. Desde el punto de vista computacional, otra propiedad de las funciones (matemáticas) es su capacidad para ser compuestas. La composición de funciones es la técnica por excelencia de la programación funcional, que permite la construcción de programas mediante el empleo de funciones primitivas o previamente definidas por el usuario. La composición de funciones refuerza la modularidad de los programas.

Ejemplo 1.4

En este ejemplo hacemos uso de la función *app* definida en el Ejemplo 1.3 y la composición de funciones para definir una versión ingenua de una función que invierte el orden de los elementos en una lista.

$$\begin{aligned} \text{rev} :: & [t] \rightarrow [t] \\ \text{rev} [] = & [] \\ \text{rev} (x : x_s) = & \text{app} (\text{rev } x_s) [x] \end{aligned}$$

Una de las características de los lenguajes funcionales que va más allá de la mera composición de funciones es el empleo de las mismas como “ciudadados de primera clase” dentro del lenguaje, de forma que las funciones puedan almacenarse en estructuras de datos, pasarse como argumento a otras funciones y devolverse como resultado. Agrupamos todas estas características bajo la denominación de *orden superior*. A veces también se dice que una función es de orden superior si algunos de sus argumentos es, a su vez, una función, esto es, si está definida sobre un dominio funcional.

Ejemplo 1.5

Un ejemplo típico de función de orden superior es la función *map*, que toma como argumento una función *f* y una lista, y forma la lista que resulta de aplicar *f* a cada uno de los elementos de la lista.

$$\begin{aligned} \text{map} :: & (t_1 \rightarrow t_2) \rightarrow [t_1] \rightarrow [t_2] \\ \text{map } f [] = & [] \\ \text{map } f (x : x_s) = & (f x) : (\text{map } f x_s) \end{aligned}$$

La ejecución de un programa consiste en la evaluación de una expresión inicial de acuerdo con las ecuaciones de definición de las funciones y algún mecanismo de *reducción*. La reducción es un proceso por el cual, mediante una secuencia de pasos,

transformamos la expresión inicial, compuesta de símbolos de función y de símbolos constructores de datos, en un *valor* (de su tipo), i.e., una expresión que solo contiene ocurrencias de símbolos constructores. El valor obtenido se considera el resultado de la evaluación. La secuencia de pasos dada en el proceso de reducción depende de la estrategia de reducción empleada. Destacan dos *estrategias de reducción o modos de evaluación*, la denominada *impaciente* y la *perezosa*⁶. Una estrategia *impaciente* no evalúa la función hasta haber evaluado completamente sus argumentos mientras que una *perezosa* evalúa los argumentos solo si su valor es necesario para el cómputo de dicha función, intentando evitar cálculos innecesarios. Por ejemplo, dada la función

$$\text{doble } x = (x + x)$$

un paso de evaluación impaciente a partir de la expresión *doble (3+4)* produciría (*doble 7*), mientras que, tras un paso de reducción perezosa, obtendríamos *(3+4) + (3+4)*.

Si bien la estrategia impaciente es más fácil de implementar en los computadores con arquitecturas convencionales⁷, puede conducir a secuencias de reducción que no terminan en situaciones en las que una evaluación perezosa es capaz de computar un valor. Por ejemplo, la evaluación de la expresión (*cero infinito*), con las funciones *cero* e *infinito* definidas como *infinito = infinito* y *cero x = x*, no termina bajo una estrategia de evaluación impaciente, mientras que el resultado es *0* usando una estrategia perezosa. De hecho, de estas dos estrategias solo la perezosa es justa (*fair*), en el sentido de garantizarla computación del valor de cada expresión de entrada, si éste existe.

Este hecho, junto con algunas facilidades de programación (e.g., la evaluación perezosa libera al programador de la preocupación por el orden de evaluación de las expresiones [68], un tema recurrente cuando el programador quiere ganar eficiencia es su preocupación por no evaluar aquello que no es absolutamente necesario) y ventajas expresivas que proporciona (e.g., la habilidad de computar con estructuras de datos infinitas), han hecho que la posibilidad de utilizar un modo evaluación perezosa sea muy apreciada en los lenguajes funcionales modernos. Muchos de los primeros lenguajes funcionales, como Lisp (puro), FP, ML, o Hope, fueron implementados empleando un modo de evaluación impaciente por las razones ya comentadas. Sin embargo, desde que existen técnicas eficientes de implementación de la *reducción en grafos* [118], los lenguajes funcionales modernos como Haskell y Miranda utilizan modos de evaluación perezosa. En este tipo de implementación, la expresión *(3+4)* del ejemplo anterior solo se evaluaría una vez, ya que sus dos apariciones dentro de la expresión *(3+4) + (3+4)* se ubican en un único nodo del grafo.

⁶En algunos contextos como el λ -cálculo, la estrategia de reducción impaciente (*eager*) se identifica con el *orden de reducción aplicativo*, mientras que la perezosa (*lazy*) con el *orden de reducción normal*. En los lenguajes imperativos, estas estrategias corresponderían básicamente al paso de parámetros por valor y por nombre, respectivamente.

⁷La estrategia de evaluación impaciente puede implementarse utilizando las técnicas de paso de parámetros por valor desarrolladas para la compilación de los lenguajes imperativos.

Como puede apreciarse los lenguajes de programación funcional son muy ricos en cuanto a las facilidades que ofrecen⁸. La Tabla 1.2 resume las características de los lenguajes funcionales en comparación con los lenguajes lógicos.

Hasta aquí se ha evitado hablar de los formalismos que sustentan esta clase de lenguajes, tal vez porque no haya un consenso unánime (como en el caso de la programación lógica) en la utilización de un determinado tipo de lógica. Pueden distinguirse tres diferentes aproximaciones: la ecuacional ([106, 107, 108]), la algebraica ([15, 46, 144]) y la funcional clásica ([9, 17, 54, 68, 118, 124]).

La *orientación clásica* de la programación funcional utiliza el λ -cálculo extendido, junto con nociones de la lógica combinatoria, como la lógica de base. En esta orientación, el λ -cálculo representa una especie de lenguaje máquina al cual pueden compilarse los programas funcionales escritos en lenguajes más sofisticados y que proporciona un mecanismo operacional básico que permite ejecutar dichos programas: la β -reducción. Para dar significado a las estructuras sintácticas se emplea una semántica denotacional [133, 140].

Las otras dos aproximaciones pueden verse como caras de la misma moneda. La *lógica ecuacional* proporciona un soporte sintáctico para la definición de funciones (mediante el uso de ecuaciones⁹) y el razonamiento ecuacional (mediante un sistema de deducción ecuacional que plasma como reglas de inferencia las conocidas propiedades reflexiva, simétrica y transitiva de la igualdad, así como las propiedades de estabilidad (también conocida como cierre por instanciación: si dos expresiones son equivalentes, lo son cualesquiera de sus instancias) y reemplazo (también llamada propiedad de cierre bajo contexto: el resultado de sustituir en cualquier contexto c una expresión por otra equivalente a ésta es equivalente a la expresión c inicial)).

Una *interpretación* del lenguaje ecuacional se define adoptando un álgebra como dominio de interpretación (el conjunto de las clases de equivalencia de términos módulo la relación de igualdad impuesta por las ecuaciones) y una función de evaluación que asocia: a los símbolos particulares del lenguaje, es decir, a los símbolos de constante y de función, elementos y funciones del dominio, respectivamente; y a las variables, elementos del dominio. De esta forma, podemos dar significado a un sistema ecuacional mediante una semántica declarativa algebraica. Por ejemplo, el siguiente programa:

⁸Algunas de las ventajas que hemos discutido en este apartado asociadas a los lenguajes funcionales han sido transferidas a lenguajes procedimentales de corte más convencional, como los lenguajes orientados a objetos. Se ha acuñado el acrónimo HOT (del inglés *High Order and strongly Typed*) para hacer referencia a los lenguajes de programación que reúnen estas características. Puede decirse que lenguajes orientados a objetos como Java son HOT, si bien el polimorfismo y el orden superior se obtienen en este lenguaje de manera oscura y un tanto imperfecta a través del mecanismo de la herencia.

⁹En el caso más general, cláusulas de Horn ecuacionales, en las que el único símbolo de predicado permitido es la igualdad.

$$\begin{aligned} \text{par } 0 &= 0 \\ \text{par } (s0) &= 1 \\ \text{par } (s(sx)) &= \text{par } x \end{aligned}$$

tiene como modelo inicial el álgebra cociente formada por dos clases de elementos: la clase del valor *0*, al que se reducen todos los naturales pares {par 0, par (s (s 0)), etc}, y la clase del valor *1*, que se corresponde con la reducción de los elementos impares {par (s 0), par (s (s (s 0))), etc}.

El teorema de Birkhoff establece la equivalencia entre la posibilidad de deducir la igualdad de dos términos (sin variables) mediante reescritura en el sistema ecuacional y la verdad de la correspondiente ecuación en el álgebra inicial cociente (modelo del sistema ecuacional)¹⁰.

El mecanismo de evaluación en esta aproximación se basa en el empleo de los sistemas de reescritura de términos. El razonamiento ecuacional se automatiza asociando a una teoría ecuacional el sistema de reescritura resultante de imponer direccionalidad a las ecuaciones que lo forman (e.g. orientándolas de izquierda a derecha) y utilizando como mecanismo de ejecución la reducción de expresiones por reescritura. De esta forma, el programa pasa a ser considerado un sistema de reglas de reescritura. En un paso de *reducción* se realiza un ajuste de patrones (*pattern matching*) que empareja un subtérmino de la expresión a evaluar con la parte izquierda de una regla y se reemplaza éste por la correspondiente instancia de la parte derecha de dicha regla.

Si el sistema de reescritura considerado es canónico¹¹, deducir la igualdad de dos términos (sin variables) a partir de un sistema ecuacional será equivalente a comprobar la igualdad (sintáctica) de las formas normales o irreducibles que se obtienen al culminar el proceso de reducción por reescritura de dichos términos.

En una orientación puramente algebraica, el interés está centrado directamente en el uso de los sistemas de reescritura como programas y el cómputo de valores a partir de expresiones funcionales, más bien que en comprobar la validez de objetivos ecuacionales. Ya hemos mencionado que el interés primordial de la programación funcional es el cómputo de valores. Si bien los sistemas de reescritura y su mecanismo de reducción están muchas veces más cerca, desde el punto de vista sintáctico y computacional, de los programas funcionales que el λ -cálculo y la β -reducción, la utilización de los sistemas de reescritura como formalización de los lenguajes de programación funcional, presenta ciertas dificultades:

¹⁰Gracias a la propiedad de “inicialidad” del álgebra cociente (informalmente, la existencia de un único homomorfismo de ese álgebra en cualquier otra de su clase), el teorema de Birkhoff puede enunciarse equivalentemente en términos de *validez* de la ecuación (verdad con respecto a todos los modelos del sistema ecuacional).

¹¹Se dice que un sistema de reescritura es canónico si es confluente y terminante (informalmente, cualquier expresión puede reducirse a una (y solo una) forma irreducible).

1. Los lenguajes funcionales emplean tipos.
2. Los programas funcionales siguen la *disciplina de constructores*, esto es, en los argumentos de las partes izquierdas de las ecuaciones que definen las funciones solamente pueden aparecer términos formados por constructores y/o variables.
3. En un programa funcional existe un orden entre las ecuaciones que forman el programa (en general, el orden en el que están escritas), lo que permite que exista un solapamiento entre las partes izquierdas de dichas ecuaciones.
4. Es habitual emplear *guardas* en las ecuaciones de definición de las funciones.

Al emplear sistemas de reescritura como programas, cada una de estas dificultades puede salvarse, respectivamente: utilizando signaturas *heterogéneas* (*many sorted*); restringiéndonos a sistemas de reescritura que también sigan la disciplina de constructores; considerando el sistema de reescritura como un conjunto ordenado de reglas de reescritura; y utilizando sistemas de reescritura condicionales o bien ampliando el sistema de reescritura con reglas que definan expresiones condicionales (del tipo *if-then-else*) cuya evaluación puede efectuarse mediante reescritura. Existen sin embargo otras dificultades (relacionadas con el empleo del orden superior y el uso de funciones currificadas) más difficilmente soslayables, que nos obligan a admitir que los programas funcionales son mucho más que sistemas de reescritura de términos. A pesar de la anterior afirmación, es de destacar que la orientación que fundamenta la programación funcional en la lógica ecuacional y en los sistemas de reescritura tiene la ventaja, respecto a la orientación clásica, de que puede establecerse una correspondencia más sencilla con el formalismo que sustenta la programación lógica¹². Esta correspondencia hace más asequible al lector desprovisto de conocimientos sobre λ -cálculo la fundamentación de la programación funcional y también facilita la discusión sobre ciertas peculiaridades de los modos de evaluación en los lenguajes funcionales.

1.3 COMPARACIÓN CON LOS LENGUAJES CONVENCIONALES Y ÁREAS DE APLICACIÓN

El objetivo de este apartado, con el que concluimos el primer capítulo, es que el lector sea consciente de que tanto la programación declarativa como la imperativa presentan ventajas que pueden ser muy útiles en las áreas de aplicación apropiadas. En este punto es conveniente remarcar que ningún lenguaje, hasta la fecha, se ha adaptado a cualquier tipo de aplicación. Así por ejemplo, el lenguaje PL/I, primer intento de lenguaje “totalitario” (auspiciado por IBM) que pretendía aunar las características de COBOL (un lenguaje para el desarrollo de aplicaciones de gestión en la empresa) y FORTRAN (un lenguaje

¹²Esta ventaja es apreciable a la hora de integrar ambos paradigmas de programación.

especialmente diseñado para el cálculo numérico) solo tuvo un éxito modesto en los años 70, ya que carecía de una visión semántica uniforme y, finalmente, perdía las ventajas específicas de los lenguajes originales. Algo comparable sucedió con el desarrollo del lenguaje Ada, auspiciado por el departamento de defensa de los EEUU.

Realizaremos la comparación entre los lenguajes declarativos y los convencionales, estudiando una serie de atributos y criterios que se han utilizado para medir la calidad de los lenguajes de programación [111, 122] y comentando en qué medida se ajusta cada clase de lenguaje a estos criterios:

1. Diseño del lenguaje y escritura de programas.

Desde esta perspectiva pueden citarse las siguientes características útiles para un lenguaje:

- a) *Sintaxis sencilla*. El programador desea que la sintaxis de un lenguaje de programación sea sencilla, ya que ésta facilita el aprendizaje y la escritura de aplicaciones. Los lenguajes declarativos poseen esta característica en alto grado, mientras que la sintaxis de los lenguajes convencionales suele ser muy compleja. Esta situación se agrava particularmente en los lenguajes orientados a objetos a los que se aúna cantidad inmanejable de librerías “estándares” que conducen a un prolongado aprendizaje.
- b) *Modularidad y compilación separada*. Estas características facilitan la estructuración de los programas, la división del trabajo y la depuración de los programas durante la fase de desarrollo. Respecto a este último punto, un diseño modular aisla y localiza los errores, que solo podrán provenir de los módulos actualmente en desarrollo y no de los ya depurados y validados. Por otra parte, la posibilidad de efectuar compilación separada es un factor de ahorro de tiempo ya que no será necesario compilar todos los módulos cada vez que se introduzcan nuevas modificaciones. La mayoría de los lenguajes imperativos proporcionan facilidades para la construcción de módulos y la compilación separada. Los lenguajes declarativos, si bien permiten un desarrollo modular de las aplicaciones, no suelen presentar herramientas para la compilación separada, máxime cuando muchos de ellos nacieron como lenguajes interpretados.
- c) *Mecanismos de reutilización del software*. Para eliminar la repetición de trabajo, un lenguaje también debe de suministrar mecanismos para reutilizar el software ya desarrollado y que se adecúa al problema a resolver. Este es un punto ligado al anterior. Los lenguajes convencionales, al poseer mecanismos más evolucionados para la modularidad y la compilación separada, han ofrecido mejores condiciones para la reutilización del software. El uso de la modularidad y la compilación separada ha permitido la creación de potentes bibliotecas que pueden considerarse extensiones del lenguaje para el que se han escrito y que facilitan la tarea de la programación. Esta potencialidad se ha desarrollado hasta el extremo en los lenguajes orientados a objetos.

Por su parte, los lenguajes declarativos suelen realizar esta función mediante, por ejemplo, la inclusión de “preámbulos de funciones”, un mecanismo más ineficiente y rudimentario. También, en la actualidad se está investigando como aumentar la reutilización del software en los lenguajes declarativos mediante el suministro de bibliotecas de plantillas genéricas y simples (cuyo significado declarativo fuese, sin duda, el esperado) que posteriormente se optimizarían utilizando una técnica de transformación automática de programas¹³ para producir un código más eficiente [74].

- d) *Facilidades de soporte al proceso de análisis.* Los lenguajes de programación declarativa pueden considerarse lenguajes de especificación. En este sentido serían muy adecuados en las fases de análisis de una aplicación y su rápido prototipado. Sin embargo, el relativo bajo nivel de los lenguajes convencionales hace que sus construcciones estén alejadas del nivel de abstracción que requiere el análisis de un problema, por lo que se hace imprescindible el uso de algún tipo de pseudocódigo para la confección de un algoritmo, siguiendo algún tipo de técnica de diseño (e.g. diseño descendente – *topdown*), que después es traducido al lenguaje elegido, en la fase de implementación. La dificultad que presentan estos lenguajes ha hecho que surjan diversas herramientas de ayuda al análisis y diseño.
- e) *Entornos de programación.* Aunque no está ligado al diseño del lenguaje en sí y aunque pueda parecer secundario éste es un factor muy importante, ya que un buen entorno de programación puede hacer que sea más fácil trabajar con un lenguaje técnicamente poco evolucionado que con uno evolucionado, pero que ofrece poco soporte externo.

Un buen entorno de programación debe ofrecer editores especiales que faciliten la escritura de programas, depuradores y, en general, utilidades de ayuda para la programación que permitan modificar y mantener grandes aplicaciones con multiples versiones. Por ejemplo, en los sistemas UNIX BSD, el lenguaje C (un lenguaje de nivel intermedio) suele acompañarse de utilidades como: *make*, que facilita la compilación de aplicaciones mediante la confección de un guión en el que se indican los módulos que deben compilarse cuando han sufrido modificación; Sistema de Control de Código Fuente (SCCS, *Source Code Control System*), para controlar el desarrollo de grandes sistemas software; y bibliotecas de rutinas de todo tipo (sobre todo rutinas matemáticas y rutinas que facilitan la confección de aplicaciones de red).

Si bien la mayoría de los lenguajes imperativos de uso en la actualidad cuentan con potentes entornos de programación (muchos de ellos visuales), los

¹³La transformación automática de programas es un método para derivar programas correctos y eficientes partiendo de una especificación ingenua y más ineficiente del problema. Esto es, dado un programa P , se trata de generar un programa P' que resuelve el mismo problema y equivale semánticamente a P , pero que goza de mejor comportamiento respecto a cierto criterio de evaluación (c.f. Capítulo 10).

lenguajes declarativos tienen aquí uno de sus puntos débiles. Una explicación a este hecho es que los lenguajes declarativos se han desarrollado y usado en ambientes universitarios y en grupos de investigación en los que estas facilidades no han sido suficientemente valoradas. Esta es una situación que cambiará conforme las técnicas que introduce la programación declarativa sean más apreciadas en sectores industriales. Por el momento, ejemplos prometedores de lenguajes declarativos con un potente entorno de programación son Visual Prolog¹⁴ y LPA Prolog¹⁵ y, con un enfoque más ambicioso¹⁶, ciaoPP y Mercury.

2. Verificación de programas.

La fiabilidad de los programas es un criterio central para medir la calidad del diseño de un lenguaje de programación. Hay diversas técnicas para verificar que un programa realiza correctamente la función para la que fue creado. Nosotros comentaremos tanto las técnicas formales como las informales y el grado de adaptación de los lenguajes a dichas técnicas:

- a) *Verificación de la corrección.* Consiste en comprobar si el programa se comporta de acuerdo a su significado esperado, i.e. a su semántica. Otra fórmula alternativa de presentar este problema sería: Dado un programa P y su especificación S ¿computan P y S la misma función? Los lenguajes declarativos, por poseer definiciones semánticas sencillas y estar basados en sólidos fundamentos que permiten aplicar los métodos formales de las matemáticas, pueden responder con relativa sencillez a este tipo de preguntas. No es el caso de los lenguajes convencionales, que poseen definiciones semánticas muy complejas y que, por consiguiente, están peor adaptados a este tipo de comprobaciones formales.
- b) *Terminación.* Un programa no solamente debe presentar el comportamiento deseado, sino que además debe terminar bajo cualquier circunstancia concebible. Por idénticas razones que en el caso anterior, los lenguajes declarativos están más adaptados para realizar este tipo de pruebas, motivo por el cual existe una amplia literatura sobre terminación de programas lógicos y funcionales (por ejemplo, véanse, como trabajos pioneros, los trabajos de Krystof Apt y Naschum Dershowitz, respectivamente).
- c) *Depuración de programas.* Si bien la comprobación del correcto funcionamiento de un programa mediante baterías de pruebas, en las que se utilizan conjuntos de datos concretos, no asegura un programa libre de errores, éste

¹⁴Información sobre este producto puede encontrarse en la dirección "<http://www.pdc.dk>".

¹⁵Información sobre este producto puede encontrarse en la dirección "<http://www.lpa.co.uk>".

¹⁶Información adicional sobre estos lenguajes y sobre sus entornos de programación puede encontrarse en las direcciones "<http://www.clip.dia.upm.es/Software/Ciao/>" y también en "<http://www.cs.mu.oz.au/research/mercury/>", respectivamente.

es un método muy socorrido por los programadores, pero que solo asegura que el programa es correcto para la batería de pruebas realizada. Así pues, si se desea tener un alto grado de certeza, esta batería debe de ser significativa, realizando un estudio exhaustivo de casos, lo que alarga el tiempo de desarrollo del programa. Si para unos ciertos datos de entrada se observa una salida inesperada o se detecta un error, el programador deberá de proceder a la depuración del programa. Para ayudar en esta tarea, el lenguaje debe suministrar mecanismos que permitan la detección rápida de errores en tiempo de compilación (que de aparecer en tiempo de ejecución serían más difíciles de depurar) y herramientas de depuración de errores en tiempo de ejecución (*debugger*). Pero también es necesario que el lenguaje sea legible para que, en caso de que los errores aparezcan en tiempo de ejecución, no sea muy complicado llegar a comprender el código fuente y así identificar rápidamente la fuente del error. Los lenguajes declarativos poseen una buena legibilidad pero no ofrecen buenas herramientas de depuración (la depuración de errores se concreta en la realización de tediosas trazas de ejecución). Sin embargo, los lenguajes convencionales poseen herramientas de depuración muy potentes, aunque su legibilidad suele ser peor. Recientemente se han desarrollado otras técnicas, conocidas como diagnóstico declarativo, que no precisan la inspección de trazas de ejecución y pueden automatizarse completamente, como veremos en el Capítulo 10.

3. Mantenimiento.

Muchos estudios han mostrado que el mayor coste en un programa, que se usa durante un periodo de varios años, es el coste de mantenimiento [122]. El mantenimiento incluye la reparación de los errores descubiertos después de que el programa se ha puesto en uso, así como y la implementación de cambios necesarios para satisfacer nuevas necesidades. Para realizar estas tareas, el lenguaje debería poseer:

- a) *Facilidad de uso y lectura.* Esto es necesario para que no sea muy complicado llegar a comprender el texto del programa y encontrar rápidamente los errores, en caso de que éstos se produzcan. También, si hay que extender el programa, un programador que no conozca la aplicación (o incluso el lenguaje) se beneficiará de estas propiedades.
- b) *Modularidad y compilación separada.* Esto es necesario para que los errores estén localizados y así facilitar las modificaciones. Nuevamente, la posibilidad de realizar compilación separada evita tener que compilar todos los módulos y ahorra gran cantidad de tiempo de desarrollo.

El grado en el que los lenguajes declarativos y los convencionales se adaptan a estas propiedades ya fue discutido en el punto (1).

4. Coste y eficiencia.

Este aspecto es uno de los más importantes a la hora de evaluar un lenguaje de programación. Existen muchas factores para los que se puede aplicar un criterio de coste, sin embargo nos centraremos en dos de ellos:

- a) *Coste de ejecución.* Históricamente se ha dado gran importancia al coste de ejecución como medida de la eficiencia de los programas. El coste de ejecución es de primaria importancia en aplicaciones que se ejecutan repetidamente. Uno de los puntos débiles de los lenguajes declarativos es que suelen ser menos eficientes que otros más convencionales. La causa última de esta ineficiencia radica en la dificultad de implementar, en máquinas de arquitecturas convencionales, las operaciones de unificación y emparejamiento, así como los mecanismos de búsqueda de soluciones que utilizan estos lenguajes. Por contra, los lenguajes convencionales son eficientes, ya que su diseño es reflejo del modelo von Neumann de computación, en el que se basa la arquitectura de los computadores actuales. No obstante, en los últimos años, se han desarrollado como ya dijimos implementaciones muy eficientes de Haskell o Prolog, que igualan y en ocasiones superan a la de lenguajes más populares como Java, cuya eficiencia no suele ser cuestionada debido a sus innegables ventajas en otros frentes.
- b) *Coste de desarrollo.* Comprende los costes asociados a las fases de análisis, programación y verificación de un programa. Es un criterio muy a tener en cuenta, ya que el coste de la hora de programador es muy alto frente al de la hora de UCP. Las estadísticas indican que, para cierto tipo de aplicaciones, los costes de desarrollo son muy bajos cuando se utilizan lenguajes declarativos y, por el contrario, los costes de utilizar lenguajes convencionales suelen ser altos. Habitualmente, en un lenguaje declarativo el número de líneas requerido para implementar un programa que resuelve un problema es una fracción del requerido cuando se usa un lenguaje convencional (e.g., la relación entre el número de líneas escritas en Prolog y el de escritas en C o Pascal es 1/10 en aplicaciones que desarrollan sistemas expertos y de 1/8 para aplicaciones que desarrollan compiladores). Teniendo en cuenta que el número de líneas producidas al año por un programador profesional es una constante, independientemente del lenguaje que utilice, la reducción en los costes de desarrollo cuando se emplean lenguajes declarativos puede ser considerable.

La discusión anterior revela que un lenguaje de programación nunca es bueno para todas las tareas. Por consiguiente, cada lenguaje tiene su dominio de aplicación (véase [122] para una discusión sobre algunos lenguajes y sus dominios de aplicación tradicionales: computación simbólica, científica, gestión, sistemas, etc.).

A pesar de ser un área de trabajo relativamente nueva, en términos del tiempo necesario para el desarrollo y la consolidación de un área de conocimiento, la programación declarativa ha encontrado una gran variedad de aplicaciones. Sin ánimo de ser exhaustivos, podemos enumerar algunas de éstas:

Procesamiento del lenguaje natural. Representación del conocimiento. Química y biología molecular. Desarrollo de Sistemas de Producción y Sistemas Expertos. Resolución de Problemas. Metaprogramación. Prototipado de aplicaciones. Bases de Datos Deductivas. Servidores y buceadores de información inteligentes. Diseño de sistemas VLSI, herramientas de soporte al desarrollo del software.

Más generalmente, la programación declarativa se ha aplicado en todos los campos de la computación simbólica (y por esto también los lenguajes declarativos se denominan a veces, *lenguajes de computación simbólica*, en contraposición a los lenguajes más tradicionales orientados a la computación numérica), la inteligencia artificial y la informática teórica (e.g., teoría de tipos). Algunos lenguajes como Prolog, λ -Prolog, Lisp (puro), ML, Haskell, Curry y Toy¹⁷ son lenguajes declarativos de propósito general, mientras que las hojas de cálculo y el lenguaje SQL (*Structured Query Language*) pueden considerarse lenguajes declarativos para dominios de aplicación específicos. Las hojas de cálculo realizan tareas de cómputo mediante la definición de expresiones matemáticas. El lenguaje SQL es un lenguaje de consulta a bases de datos [80] que utiliza las construcciones del álgebra relacional. El lenguaje SQL tiene varias partes entre ellas, el lenguaje de definición de datos (DDL, *Data Definition Language*) y el de manipulación de datos (DML, *Data Manipulation Language*), que es un lenguaje no procedimental que puede utilizarse interactivamente o inmerso en otros lenguajes (e.g. Visual Prolog y la mayoría de los lenguajes convencionales). El DML de SQL permite especificar qué datos se desean obtener sin la necesidad de indicar cómo obtenerlos.

No queremos cerrar este apartado sin mencionar algunas características que surgieron con los lenguajes declarativos y que han influido posteriormente en la concepción de otros lenguajes. Por ejemplo, la precompilación de Prolog producía un código intermedio portable comparable a la idea de los *bytecode* de Java; por otro lado, Java hereda también la gestión dinámica de la memoria *heap* con *garbage collection* de lenguajes funcionales como LISP (que no tienen otros lenguajes imperativos como Pascal o C++).

RESUMEN

En este capítulo hemos estudiado las características más notables de los lenguajes de programación convencionales y de los lenguajes de programación declarativos, con

¹⁷Curry y Toy son lenguajes, todavía en desarrollo, que integran las características de los lenguajes lógicos y funcionales.

el objetivo de entender qué son los lenguajes declarativos y qué aportan de nuevo con relación a los lenguajes convencionales, así como establecer su ámbito de aplicación.

- Hemos resumido los principios en los que se basa la organización de las computadoras actuales (modelo de von Neumann) para mostrar la íntima relación existente entre el diseño de éstas y los lenguajes convencionales. Los recursos expresivos de los lenguajes convencionales pueden considerarse como abstracciones de los componentes y operaciones elementales de la máquina de von Neumann.
- De la anterior relación proviene la eficacia de los lenguajes convencionales pero también muchas de sus deficiencias: i) mezcla de los aspectos lógicos y de control en la solución del problema ii) tratamiento matemático impuro que dificulta el estudio formal de las propiedades de los programas y la definición semántica de los mismos.
- Un programa convencional es una secuencia de instrucciones, que son órdenes a la máquina (*lenguajes imperativos*), que operan sobre un estado que, generalmente, no se declara explícitamente. Para entender el programa debemos ejecutarlo mentalmente siguiendo el modelo de computación de von Neumann.
- En el contexto de la programación declarativa, un programa es un conjunto de fórmulas lógicas, que son la especificación del problema que se pretende resolver, y la computación se entiende como una forma de inferencia o deducción en dicha lógica.
- Los principales requisitos que debe cumplir la lógica empleada como soporte de un lenguaje declarativo son:
 1. disponer de una sintaxis suficientemente expresiva;
 2. disponer de una semántica operacional que pueda utilizarse como mecanismo de cómputo y permita ejecutar de manera eficiente los programas;
 3. disponer de una semántica declarativa que permita dar un significado a los programas de forma independiente a su posible ejecución;
 4. disponer de resultados de corrección y completitud, que aseguren que lo que se computa coincide con aquello que es considerado como correcto o verdadero.
- Según la clase de lógica que empleemos como fundamento del lenguaje declarativo obtenemos los diferentes estilos de programación declarativa.
- La *programación lógica* se basa en fragmentos de la lógica de predicados, siendo el más popular la lógica de *cláusulas de Horn*, ya que posee una semántica operacional implementable de forma eficiente, como es el caso de la *resolución SLD*. Como semántica declarativa se utiliza una semántica por *teoría de modelos*.

que toma como dominio de interpretación un universo puramente sintáctico: el *universo de Herbrand*.

- La *resolución SLD* es un método de prueba por refutación, correcto y completo para la lógica de cláusulas de Horn, que emplea el algoritmo de *unificación* como mecanismo de base para la extracción de respuestas.
- El mecanismo de cómputo de los lenguajes lógicos imprime muy buenas características a estos lenguajes, entre las que cabe destacar las siguientes:
 - gestión automática de la memoria;
 - permite una *búsqueda indeterminista* (*don't-know built-in search*) de soluciones;
 - permite computar con *datos parcialmente definidos*;
 - la *relación de entrada/salida* no está fijada de antemano, por lo que un programa puede responder a diferentes cuestiones (*objetivos*) sin necesidad de efectuar ningún cambio en el programa.
- Los *lenguajes funcionales* se basan en el concepto de *función* (matemática) y su definición mediante ecuaciones (generalmente recursivas), que constituyen el programa. La programación funcional se centra en la evaluación de expresiones (funcionales) para obtener un *valor*.
- La *orientación clásica* de la programación funcional utiliza el λ -cálculo extendido, junto con nociones de la semántica denotacional y de la lógica combinatoria, como lógica de base. Sin embargo, existen otros formalismos que se han utilizado para sustentar un modelo más simple para esta clase de lenguajes: la lógica ecuacional y las álgebras libres.
- Algunas características destacables de los lenguajes funcionales son:
 - Los lenguajes funcionales modernos son lenguajes *fuertemente basados en tipos* (*strongly typed*). La necesidad de fijar el perfil de la función (dominio y el rango de la función) obliga a introducir la idea de *tipo de datos*. Los tipos se construyen a partir de los llamados *constructores del tipo* y se admite el *polimorfismo de tipos*.
 - *Transparencia referencial*: el resultado (*salida*) de aplicar una función sobre sus argumentos viene determinado exclusivamente por el valor de éstos (*su entrada*). Por consiguiente, las funciones de los programas funcionales son funciones puramente matemáticas y no presentan *efectos laterales* (*side effects*), lo que permite el estilo de la programación funcional basado en el razonamiento ecuacional (la substitución de iguales por iguales) y los cómputos deterministas.

- La transparencia referencial también da a las funciones de los programas funcionales su capacidad para ser compuestas. La composición de funciones es la técnica por excelencia de la programación funcional y permite la construcción de programas mediante el empleo de funciones primitivas o previamente definidas por el programador. La técnica de programación mediante la composición de funciones refuerza la modularidad de los programas.
- El empleo de las funciones como “ciudadanos de primera clase” dentro del lenguaje, de forma que las funciones puedan almacenarse en estructuras de datos, pasarse como argumento a otras funciones o devolverse como resultado: *Orden superior*.
- Para finalizar se han comparado las características de los lenguajes declarativos con respecto a las de los lenguajes convencionales, con el objeto de identificar sus áreas de aplicación. En la comparación se han empleado una serie de atributos y criterios comúnmente utilizados para medir la calidad de los lenguajes de programación desde diferentes perspectivas: diseño del lenguaje y escritura de programas; verificación de programas; mantenimiento y criterios de coste y eficiencia. Se concluye que un lenguaje de programación nunca es bueno para todas las tareas. Cada lenguaje tiene su dominio de aplicación.
- La programación declarativa se ha aplicado en todos los campos de la computación simbólica (*lenguajes de computación simbólica*). Más concretamente se ha aplicado en amplias áreas de la inteligencia artificial, en la metaprogramación, en el prototipado de aplicaciones, y en el área de las bases de datos deductivas, entre otras.

CUESTIONES Y EJERCICIOS

Cuestión 1.1 Indique cuál de las siguientes características no es de la programación imperativa:

- Programa = transcripción de un algoritmo.
- Instrucciones = órdenes a la máquina.
- Modelo computacional = máquina de inferencias lógicas.
- Variables del programa = referencias a memoria.

Cuestión 1.2 Enumere las principales características que se asocian con un lenguaje de programación convencional (imperativo).

Cuestión 1.3 ¿Cuál de los siguientes lenguajes podría considerarse tanto un lenguaje de especificación ejecutable como un lenguaje de programación de alto nivel?

- C++.
- Pascal.
- Prolog.
- Ada.

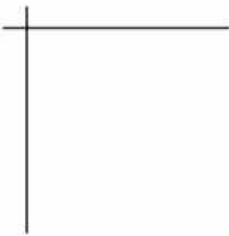
Cuestión 1.4 Señale cuál de las siguientes ideas **no** pertenece al estilo de programación declarativa:

- Programa = lógica + control.
- Programación = demostración de teoremas + técnicas de extracción de respuestas.
- Programación = lenguaje de la lógica de primer orden como notación para los programas e inferencia lógica como mecanismo de computación.
- Programación = algoritmo + estructura de datos.

Cuestión 1.5 Indique cuál de las siguientes ideas **no** se relaciona con el estilo de programación declarativa:

- Programa = teoría formal que especifica un problema.
- Semántica operacional = deducción o inferencia lógica.
- Programa = secuencia de órdenes que operan sobre los estados de la máquina.
- Programa = lógica + control.

Cuestión 1.6 Enumere las principales características que se asocian con un lenguaje de programación declarativo.



Parte I

Fundamentos

Sistemas Formales, Lógica y Lenguajes de Programación

Como se ha dicho, un lenguaje de programación declarativa descansa sobre algún tipo de lógica que se emplea como lenguaje de programación. Así pues, el conocimiento de la lógica es muy conveniente para la comprensión de este tipo de lenguajes. El objetivo de este capítulo es revisar los principales conceptos y notaciones sobre la lógica de predicados. También, entre los objetivos de este capítulo, estará el poner de manifiesto las relaciones existentes entre la lógica y los lenguajes de programación, cuando se les considera como sistemas formales, y cómo la caracterización semántica formal es una parte imprescindible de la definición de un lenguaje de programación.

2.1 SISTEMAS FORMALES

Habitualmente, la palabra *formal* se usa para referirnos a esa situación en la que se emplean símbolos cuyo comportamiento y propiedades están completamente determinados por un conjunto dado de reglas. Según Kleene [77], en un *sistema formal* los símbolos carecen de significado *a priori* y, al manejarlos, hemos de tener cuidado de no presuponer nada sobre sus propiedades, salvo lo que se especifique en el sistema. Esta visión de sistema formal, en el sentido de un mero *cálculo*, proviene, principalmente, de los trabajos de Frege, Peano y Hilbert, que han conducido a la axiomatización de la lógica. Mosterín asigna el nombre de “formalismo” al conjunto de signos y cadenas de signos que son parte de un sistema formal y hace énfasis en que un formalismo carece de significado: un *formalismo* es “un mero juego de signos y combinaciones de signos; desprovisto de toda significación” [99]. Esto se contrapone al concepto de *lenguaje*, que se define como un sistema de signos *interpretados*, i.e., signos a los que se ha atribuido un significado. También [61] y [65], por citar algunos referencias más accesibles al lec-

tor, son de la opinión de que en un sistema formal los símbolos carecen de significado. Sin embargo otros autores, siguiendo a Carnap [24], añaden a los sistemas formales un componente *semántico* que es muy conveniente desde el punto de vista del estudio de los lenguajes de programación. Según Carnap, el estudio de un sistema abarca tanto la componente sintáctica como la semántica, así como las relaciones entre ambas. Con esta perspectiva, la definición un sistema formal sería algo más que la especificación de *lenguaje formal*¹ y un *cálculo deductivo*.

Definición 2.1 (Sistema formal) *Un Sistema Formal S es un modelo de razonamiento matemático en el que se distinguen los siguientes componentes:*

1. *Sintaxis.*

- a) *Lenguaje Formal: Conjunto de símbolos y fórmulas sintácticamente correctas. Para definir un lenguaje formal se necesita:*
 - *Un vocabulario o alfabeto: Un conjunto (infinito numerable) de letras y símbolos a utilizar en las construcciones de S.*
 - *Reglas que establezcan qué cadenas de signos son fórmulas bien formadas^a en S.*
- b) *Cálculo Deductivo: conjunto de reglas que permiten obtener fórmulas nuevas atendiendo solo a aspectos sintáticos de los símbolos. Un cálculo deductivo queda definido especificando:*
 - *Un conjunto (posiblemente vacío) de fórmulas bien formadas de S que van a utilizarse como axiomas.*
 - *Un conjunto finito de reglas de inferencia.*
 - *Un concepto de deducción (demostración), formalizado como un conjunto de reglas para construir las estructuras deductivas correctas, junto con las condiciones que debe reunir una deducción para dar como resultado un teorema de S.*

2. *Semántica.*

La semántica^b estudia la adscripción de significado a los símbolos y fórmulas de los lenguajes formales. Para ello se introduce el concepto de interpretación que, habitualmente, consiste en una serie de reglas precisas que permiten asignar objetos de un dominio (no necesariamente matemático) a ciertas expresiones del lenguaje formal. Esto se acompaña de una noción de valoración y un conjunto de reglas de valoración que hace posible identificar cuándo una fórmula es verdadera en una interpretación.

^aLa expresión “fórmula bien formada” la abreviaremos mediante la notación “fbf”.

^bLa palabra “semántica” fue acuñada por Michel Bréal, en un libro sobre cómo las palabras cambian de significado, publicado en 1900.

¹Empleamos “lenguaje formal” en el mismo sentido que Mosterín emplea la palabra “formalismo”. Usaremos estos términos de manera intercambiable.

Concluimos este apartado enunciando algunas de las propiedades deseables de los sistemas formales:

- **Consistencia.**

Un Sistema Formal es consistente cuando de él no se desprenden contradicciones, esto es, es imposible demostrar una fórmula y la negación de esa fórmula.

- **Corrección.**

Un Sistema Formal es correcto si toda fórmula demostrable sin premisas, es válida: **Todo lo demostrable es cierto.**

- **Compleitud.**

Un sistema formal es completo si toda fórmula que es válida, según la noción de verdad, es demostrable sin premisas en el sistema: **Todo lo que es cierto se puede demostrar.**

- **Decidibilidad.**

Un sistema formal es decidable si existe un procedimiento finito para comprobar si una fórmula es o no válida.

2.2 LÓGICA DE PREDICADOS: REVISIÓN DE CONCEPTOS

En este apartado se resumen y revisan parte de los conceptos sobre lógica de primer orden que aparecen en [75], si bien con ligeros cambios de notación (e.g., empleamos letras mayúsculas para los símbolos de variable y minúsculas para símbolos de función y predicado). Los lectores que hayan seguido un curso de lógica de predicados pueden saltar este apartado si lo desean.

La lógica de predicados, también denominada lógica de primer orden, es susceptible de caracterizarse como un sistema formal. Como tal, podemos estudiar los componentes antes mencionadas siguiendo el esquema dado en su definición.

2.2.1. El lenguaje formal \mathcal{L}

El lenguaje formal de la lógica de predicados, \mathcal{L} , se define caracterizando las reglas que construyen, a partir de un alfabeto, las expresiones sintácticamente correctas (términos y fórmulas bien formadas).

1. **Alfabeto.**

El alfabeto está constituido por dos clases de símbolos: los comunes a cualquier lenguaje y los propios de él.

- a) Símbolos comunes a todos los formalismos.

- Un conjunto infinito numerable de símbolos de variable $\mathcal{V} = \{X, Y, \dots\}$.

- Un conjunto de conectivas² $\{\neg, \rightarrow\}$ y cuantificadores $\{\forall\}$.
- Signos de puntuación³: “(”, “)”, “;”.

b) Símbolos propios de un formalismo.

Los alfabetos de cada formalismo se diferencian en sus símbolos peculiares, distintos para cada uno de ellos. El número de estos símbolos es variable, según los formalismos. Puede haber desde uno⁴ hasta una cantidad infinita numerable.

- El conjunto de símbolos de constante $C = \{a, b, c, \dots\}$.
- El conjunto de símbolos de función de aridad n $\mathcal{F} = \{f, g, h, \dots\}$.
- El conjunto de símbolos de predicado (o conjunto de símbolos de relación de aridad n) $\mathcal{P} = \{p, q, r, \dots\}$.

Cada símbolo de función f (o predicado p) tiene asociado un número natural, denominado *aridad*, que indica el número de argumentos sobre los que se puede aplicar f (o p). Si la aridad de un símbolo f (o p) es n , solemos hacer explícito este hecho escribiendo f (o p^n). Es habitual considerar los símbolos de constante como símbolos de función de aridad cero.

2. Términos y fórmulas.

Son las cadenas de símbolos, formados de acuerdo a las reglas inductivas especificadas a continuación, que constituyen construcciones bien formadas del lenguaje. De manera genérica, nos referiremos a los términos y las fórmulas con el nombre de *expresiones de \mathcal{L}* .

- *Término.*
 - a) Si $t \in (\mathcal{V} \cup C)$ entonces t es un término.
 - b) Si $t_1, t_2, t_3, \dots, t_n$ son términos y f es un símbolo de función entonces $f(t_1, t_2, t_3, \dots, t_n)$ es un término.

Denotamos el conjunto de todos los términos mediante la letra \mathcal{T} . Un término *básico* (*ground*) es un término sin variables.

- *Fórmula atómica.*
Si $t_1, t_2, t_3, \dots, t_n$ son términos y p^n es un símbolo de relación entonces $p^n(t_1, t_2, t_3, \dots, t_n)$ es una fórmula atómica.
- *Fórmula bien formada.*

²En virtud de las leyes de equivalencia lógica que se muestran más abajo (véase 3), no es necesario introducir el resto de conectivas lógicas habituales \wedge , \vee , \leftrightarrow , así como tampoco el cuantificador existencial \exists , dado que cualquier fórmula que las contiene puede transformarse en otra equivalente que no las usa.

³Se incluyen para facilitar la legibilidad de las fórmulas. Muchos autores no los consideran parte del sistema formal.

⁴Suponemos que nuestro formalismo va a versar sobre algo, así que tendrá al menos una constante individual.

- a) Toda fórmula atómica es una fbf.
- b) Si \mathcal{A} y \mathcal{B} son fbf, también lo son: $(\neg\mathcal{A})$, $(\neg\mathcal{B})$ y $(\mathcal{A} \rightarrow \mathcal{B})$.
- c) Si \mathcal{A} es una fbf y $X \in \mathcal{V}$ entonces, $(\forall X)\mathcal{A}$ es una fbf. La variable X se denomina *variable cuantificada*.

Una fórmula *básica* (*ground*) es una fbf que no contiene variables.

3. Equivalencias.

$(\mathcal{A} \wedge \mathcal{B})$	es abreviatura de:	$(\neg(\mathcal{A} \rightarrow (\neg\mathcal{B})))$
$(\mathcal{A} \vee \mathcal{B})$	es abreviatura de:	$((\neg\mathcal{A}) \rightarrow \mathcal{B})$
$(\mathcal{A} \leftrightarrow \mathcal{B})$	es abreviatura de:	$(\neg((\mathcal{A} \rightarrow \mathcal{B}) \rightarrow (\neg(\mathcal{B} \rightarrow \mathcal{A}))))$
$(\exists X)\mathcal{A}$	es abreviatura de:	$(\neg((\forall X)(\neg\mathcal{A})))$

Observaciones 2.1

1. El lenguaje de la lógica de predicados se reduce esencialmente al de la lógica de proposiciones (o de “orden 0”) cuando eliminamos las variables y los cuantificadores. En ocasiones se simplifica la noción de fórmula atómica proposicional (e.g. *mujer(madre(lola))*) por simples variables de predicado (e.g. *p*), eliminando de esta forma la categoría de los símbolos propios. Las variables de predicado también se denominan *variables enunciativas* y se emplean para formalizar los enunciados más simples, aquéllos que no contienen conectivas. Más información sobre la lógica proposicional puede obtenerse en [75].
2. Las fórmulas atómicas son las expresiones más simples del lenguaje a las que se les puede atribuir un significado de verdadero o falso. Esto último es la razón de que se les asigne el nombre de “atómicas”.
3. En la programación lógica es habitual emplear la siguiente nomenclatura referente a las fórmulas atómicas. En general se habla de *átomo* para referirse a una fórmula atómica. También se usa el término *literal* positivo (o negativo) para hacer referencia a un átomo (o a la negación de un átomo).
4. Si \mathcal{A} es un átomo, se dice que $L_1 \equiv \mathcal{A}$ y $L_2 \equiv \neg\mathcal{A}$ es un *par de literales complementarios* el uno del otro.



En lo que sigue relajaremos las restricciones de notación de nuestro lenguaje formal permitiendo, en ocasiones, el uso de identificadores mnemotécnicos para designar símbolos de constante, de función, o de predicado.

Ejemplo 2.1

Vamos a formalizar los siguientes enunciados escritos en lenguaje natural, empleando las convenciones de nuestro lenguaje formal:

- Para cada conjunto X, hay un conjunto Y, tal que el cardinal de Y es mayor que el cardinal de X.

$$(\forall X)[conjunto(X) \rightarrow (\exists Y)[conjunto(Y) \wedge mayor(card(Y), card(X))]]$$

donde “conjunto” y “mayor” son símbolos de predicado y “card” es un símbolo de función.

- Todos los bloques que están encima de bloques que han sido movidos o están unidos a bloques que han sido movidos, también han sido movidos.

$$(\forall X)(\forall Y)[[bloque(X) \wedge bloque(Y) \wedge [encima(X, Y) \vee unido(X, Y)] \wedge movido(Y)] \rightarrow \\ \rightarrow movido(X)]$$

donde “bloque”, “encima”, “unido” y “movido” son símbolos de predicado.

Ocasionalmente, se han empleado llaves “{”, “}” y corchetes “[”, “]” en lugar de paréntesis para aumentar la claridad de las fórmulas. Para todo lo referente a la formalización del lenguaje natural véase [75] y también el Apartado 8.2.

Concluimos este subapartado con una serie de definiciones de conceptos que utilizaremos más adelante:

1. *Ocurrencia* de una variable es cualquier aparición de una variable en una fórmula.
2. *Radio de acción*, ámbito o alcance de un cuantificador:
 - En la fbf $(\forall X)\mathcal{A}$, el radio de acción de $(\forall X)$ es \mathcal{A} .
 - En la fbf $(\exists X)\mathcal{A}$, el radio de acción de $(\exists X)$ es \mathcal{A} .
3. Una ocurrencia de la variable X en una fbf se dice que es *ligada* si aparece dentro del radio de acción de un cuantificador universal $(\forall X)$ o uno existencial $(\exists X)$.
4. Una ocurrencia de la variable X en una fbf se dice que es *libre*, si su aparición no es ligada.
5. Una fórmula *cerrada* es la que no tiene ocurrencias de variables libres. Las fórmulas cerradas también se denominan *sentencias*.

También es conveniente introducir los siguientes convenios de notación referidos a metávariables que representan fbf: Dada una fbf cualquiera \mathcal{A} , escribiremos $\mathcal{A}(X_i)$ o bien $\mathcal{A}(X_1, \dots, X_n)$ cuando estemos interesados en prestar atención a ciertas variables que ocurren en ellas. Estas expresiones indicarán a menudo, aunque no siempre, que las variables mencionadas aparecen libres en la fbf. Si X_i aparece libre en $\mathcal{A}(X_i)$ entonces $\mathcal{A}(t)$ representará el resultado de sustituir por t cada ocurrencia libre de la variable X_i en \mathcal{A} .

Ejemplo 2.2

En la fbf $(\forall X_1)(r(X_1, X_2) \rightarrow (\forall X_2)p(X_2))$, podemos comprobar que:

1. X_1 aparece ligada.
2. La primera ocurrencia de X_2 aparece libre. Por consiguiente, la fórmula no es cerrada; tampoco es una fórmula básica.
3. La segunda ocurrencia de X_2 aparece ligada.
4. El radio de acción del cuantificador $(\forall X_1)$ es la fbf $r(X_1, X_2) \rightarrow (\forall X_2)p(X_2)$.
5. El radio de acción del cuantificador $(\forall X_2)$ es la fbf $p(X_2)$.

Por otro lado, la fbf $(r(a, b) \rightarrow p(b))$ es un ejemplo de fórmula básica, ya que en ella no aparecen variables. Las fórmulas cerradas y las básicas formalizan *enunciados*⁵ y son de singular importancia para nosotros en el futuro.

2.2.2. Cálculo deductivo, \mathcal{K}_L

La *teoría de la deducción*⁶ formaliza los conceptos de inferencia o prueba de teoremas, es decir, el proceso que permite obtener una conclusión a partir de ciertas premisas. La representación formal de este proceso se llama *estructura deductiva*. Las estructuras deductivas correctas se construyen haciendo uso de los axiomas, las reglas de inferencia y un concepto apropiado de deducción o de demostración.

Entre los distintos cálculos deductivos que existen para la lógica de predicados, describimos a continuación uno de los más utilizados, conocido como cálculo deductivo \mathcal{K}_L .

1. Axiomas de \mathcal{K}_L .

Cualesquiera que sean las fbf \mathcal{A} , \mathcal{B} y \mathcal{C} , las siguientes fbf son axiomas del cálculo \mathcal{K}_L

- (K1) $(\mathcal{A} \rightarrow (\mathcal{B} \rightarrow \mathcal{A}))$
- (K2) $((\mathcal{A} \rightarrow (\mathcal{B} \rightarrow \mathcal{C})) \rightarrow ((\mathcal{A} \rightarrow \mathcal{B}) \rightarrow (\mathcal{A} \rightarrow \mathcal{C})))$
- (K3) $(((\neg \mathcal{A}) \rightarrow (\neg \mathcal{B})) \rightarrow (\mathcal{B} \rightarrow \mathcal{A}))$
- (K4) $((\forall X)\mathcal{A} \rightarrow \mathcal{A})$, si X no aparece libre en \mathcal{A}
- (K5) $((\forall X)\mathcal{A}(X) \rightarrow \mathcal{A}(t))$, si t no introduce variables ligadas en \mathcal{A}
- (K6) $((\forall X)(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow (\mathcal{A} \rightarrow (\forall X)\mathcal{B}))$, si X no aparece libre en \mathcal{A}

2. Reglas de inferencia de \mathcal{K}_L .

El cálculo deductivo que estamos describiendo consta solamente de dos reglas de inferencia:

⁵Un *enunciado* es una expresión lingüística que establece un pensamiento completo, relativo a un dominio de interpretación, y sobre la cual se puede afirmar su verdad o falsedad.

⁶Con algunas variantes, se denomina también *teoría de la demostración* o *teoría de la prueba*.

- a) *Modus ponens (MP)*, que afirma que de \mathcal{A} y $(\mathcal{A} \rightarrow \mathcal{B})$ se puede inferir como consecuencia inmediata \mathcal{B} ,
- b) *Generalización (Gen)*, que afirma que de \mathcal{A} se puede inferir como consecuencia inmediata $(\forall X)\mathcal{A}$, siendo X cualquier variable.

donde \mathcal{A} y \mathcal{B} son dos fbf cualesquiera.

3. Concepto de deducción de \mathcal{K}_L .

Sea Γ un conjunto de fbf. Una sucesión finita $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ de fbf es una *deducción a partir de Γ* si, para todo $i \in \{1, 2, \dots, n\}$, se cumple alguna de las siguientes condiciones:

- a) \mathcal{A}_i es un axioma de \mathcal{K}_L ,
- b) $\mathcal{A}_i \in \Gamma$
- c) \mathcal{A}_i se infiere inmediatamente de dos elementos anteriores de la secuencia, digamos \mathcal{A}_j y \mathcal{A}_k (con $j < i$ y $k < i$), mediante la aplicación de la regla **MP** o de la regla **Gen**.

El último miembro \mathcal{A}_n se dice que es *deducible* a partir de Γ , o que es *derivable* a partir de Γ , lo que denotamos por $\Gamma \vdash \mathcal{A}_n$. Se dice que los elementos del conjunto Γ son las *premisas* y \mathcal{A}_n se llama *conclusión* de la deducción. El número n de fórmulas en la sucesión $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ se llama *longitud* de la derivación. También diremos que es una derivación de n pasos.

Una *demostración* es una deducción sin premisas. Si la fbf \mathcal{A} es el último miembro de una demostración, decimos que \mathcal{A} es un *teorema* de \mathcal{K}_L y escribimos $\vdash \mathcal{A}$.

Ejemplo 2.3

Para ilustrar cómo se hacen inferencias en el sistema \mathcal{K}_L , presentamos la demostración del teorema: $\vdash (\forall X)(p(X) \rightarrow p(X))$.

- (1) $((p(X) \rightarrow ((p(X) \rightarrow p(X)) \rightarrow p(X))) \rightarrow$
 $((p(X) \rightarrow (p(X) \rightarrow p(X))) \rightarrow (p(X) \rightarrow p(X)))$ K2
- (2) $(p(X) \rightarrow ((p(X) \rightarrow p(X)) \rightarrow p(X)))$ K1
- (3) $((p(X) \rightarrow (p(X) \rightarrow p(X))) \rightarrow (p(X) \rightarrow p(X))) \rightarrow (p(X) \rightarrow p(X))$ MP, (1), (2)
- (4) $(p(X) \rightarrow (p(X) \rightarrow p(X)))$ K1
- (5) $(p(X) \rightarrow p(X))$ MP, (3), (4)
- (6) $(\forall X)(p(X) \rightarrow p(X))$ Gen (5)

En la línea (1) se ha utilizado el axioma K2 cambiando \mathcal{B} por $(p(X) \rightarrow p(X))$ y C por $p(X)$. En la línea (2) se ha utilizado el axioma K1 cambiando \mathcal{B} por $(p(X) \rightarrow p(X))$. En la línea (4) se ha utilizado el axioma K1 cambiando \mathcal{B} por $p(X)$.

A continuación presentamos sin prueba una serie de resultados de singular importancia y que facilitan la tarea de la deducción.

Teorema 2.1 (Teorema de la deducción) Sean \mathcal{A} y \mathcal{B} fbf y Γ un conjunto de fbf.

1. Si $\Gamma \cup \{\mathcal{A}\} \vdash \mathcal{B}$ y \mathcal{A} es una fbf cerrada entonces, $\Gamma \vdash (\mathcal{A} \rightarrow \mathcal{B})$.
2. Si $\Gamma \vdash (\mathcal{A} \rightarrow \mathcal{B})$ entonces, $\Gamma \cup \{\mathcal{A}\} \vdash \mathcal{B}$.

El siguiente corolario muestra la relación existente entre deducción y demostración.

Corolario 2.1 Sea \mathcal{B} una fbf y $\Gamma = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$ un conjunto de fbf cerradas. $\Gamma \vdash \mathcal{B}$ si y solo si $\vdash (\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_n \rightarrow \mathcal{B})$.

Finalmente, presentamos el llamado teorema de intercambio pero antes necesitamos cierta terminología. Sea la fbf $\neg(\mathcal{A}_1 \rightarrow (\mathcal{A}_2 \rightarrow \mathcal{A}_1))$, donde \mathcal{A}_1 y \mathcal{A}_2 son fbf. La estructura $\neg(\dots \rightarrow (\mathcal{A}_2 \rightarrow \dots))$, se dice que es el contexto de la fbf \mathcal{A}_1 . En general, dada una fbf C que contiene una o más ocurrencias de una fbf \mathcal{A} , la denotaremos por $C[\mathcal{A}]$. Diremos que $C[\mathcal{A}]$ es el *contexto* de \mathcal{A} .

Teorema 2.2 (Teorema de intercambio) Sean \mathcal{A}, \mathcal{B} y $C[\mathcal{A}]$ dos fbf cualesquiera Si $\vdash (\mathcal{A} \leftrightarrow \mathcal{B})$ entonces $\vdash C[\mathcal{A}] \leftrightarrow C[\mathcal{B}]$.

La utilidad del teorema 2.2 radica en que da la posibilidad de plantear la deducción basándola en el intercambio de partes de las fórmulas, que se sustituyen por otras equivalentes. El teorema de intercambio puede entenderse como una regla de inferencia:

$$\frac{(\mathcal{A} \leftrightarrow \mathcal{B}), C[\mathcal{A}]}{C[\mathcal{B}]}$$

Aquí, la equivalencia que precede a la fórmula $C[\mathcal{A}]$ indica que si se da esa equivalencia y aparece en la deducción una fórmula $C[\mathcal{A}]$, podemos inferir de forma inmediata $C[\mathcal{B}]$.

2.2.3. Semántica del lenguaje de la lógica de predicados

La semántica de un sistema lógico suele abordarse desde la llamada *teoría de modelos*. En la teoría de modelos el significado se formaliza mediante la noción de *modelo* que consiste en una entidad de soporte (comúnmente matemática) junto con las propiedades y relaciones que se dan entre los elementos de esa entidad.

Interpretaciones

Una interpretación nos da una manera de asignar significado a las construcciones de \mathcal{L} . Una *interpretación* \mathcal{I} de \mathcal{L} es un par $(\mathcal{D}, \mathcal{J})$ que consiste en:

1. Un conjunto no vacío \mathcal{D} , el dominio de \mathcal{I} .

2. Una aplicación \mathcal{J} que asigna:

- A cada símbolo de constante, a , de \mathcal{L} un elemento distinguido de \mathcal{D} ; $\mathcal{J}(a) = \bar{a}_i$.
- A cada símbolo de función f n -ario de \mathcal{L} una función $\mathcal{J}(f) = \bar{f}$, tal que $\bar{f} : \mathcal{D}^n \rightarrow \mathcal{D}$.
- A cada símbolo de relación r n -ario de \mathcal{L} una relación⁷ $\mathcal{J}(r_i^n) = \bar{r}$, tal que $\bar{r} \subset \mathcal{D}^n$; esto es un conjunto de n -tuplas de \mathcal{D}^n , $\bar{r} = \{(d_1, \dots, d_n) \mid d_i \in \mathcal{D}\}$

Solo podremos hablar de verdad y falsedad en el contexto de una interpretación, pero para ello todavía falta un paso previo (si la fbf no es cerrada): asignar valores a las ocurrencias de las variables (libres) en la fórmula.

Valoración

Una valoración $\vartheta : \mathcal{T} \rightarrow \mathcal{D}$ en \mathcal{I} es una aplicación que asigna a cada término de \mathcal{L} un elemento del dominio de la interpretación \mathcal{D} y que definimos inductivamente mediante las siguientes reglas:

$$\vartheta(t) = \begin{cases} (1) & v(X) & \text{si } t \in \mathcal{V} \wedge t \equiv X; \\ (2) & \mathcal{J}(a) & \text{si } t \in C \wedge t \equiv a; \\ (3) & \mathcal{J}(f)(\vartheta(t_1) \dots \vartheta(t_n)) & \text{si } f \in \mathcal{F} \wedge (t_1 \in \mathcal{T} \wedge \dots \wedge t_n \in \mathcal{T}) \\ & & \wedge t \equiv f(t_1 \dots t_n); \end{cases}$$

donde v es una aplicación que asigna a cada variable de \mathcal{L} un elemento del dominio de interpretación \mathcal{D} .

Una valoración $\vartheta_X^{\bar{x}}$ que coincide exactamente con la valoración ϑ , salvo quizás en el valor asignado a la variable $X \in \mathcal{V}$, se denomina valoración X -equivalente de ϑ .

Satisfacibilidad

Sea una interpretación $\mathcal{I} = (\mathcal{D}, \mathcal{J})$. Sea \mathcal{A} una fbf de \mathcal{L} . Decimos que la valoración ϑ en \mathcal{I} satisface la fbf \mathcal{A} si y solo si se cumple que:

1. Si $\mathcal{A} \equiv r(t_1 \dots t_n)$ entonces $\bar{r}(\vartheta(t_1) \dots \vartheta(t_n))$ es verdadero, donde $\bar{r} = \mathcal{J}(r)$ es una relación en \mathcal{D} .
2. Si \mathcal{A} es de la forma:
 - a) $\neg \mathcal{B}$ entonces ϑ no satisface \mathcal{B} ;

⁷Muchos autores, [26, 86] entre otros, enuncian la última condición diciendo que: La interpretación asigna, por cada relator n -ario de \mathcal{L} una aplicación $\bar{r} : \mathcal{D}_{\mathcal{I}}^n \rightarrow \{V, F\}$. En cierto sentido lo que estos autores establecen es una función de verdad para las fórmulas atómicas. Esto es equivalente a nuestro tratamiento consistente en asignar una relación $\bar{r} \subset \mathcal{D}^n$, sobre la base de que si $(d_1, \dots, d_n) \in \bar{D}^n$ entonces $\bar{r}(d_1, \dots, d_n)$ se considera verdadero y, si no, falso.

- b) $(B \wedge C)$ entonces ϑ satisface B y ϑ satisface C ;
 - c) $(B \vee C)$ entonces ϑ satisface B o ϑ satisface C ;
 - d) $(B \rightarrow C)$ entonces ϑ satisface $\neg B$ o ϑ satisface C ;
 - e) $(B \leftrightarrow C)$ entonces ϑ satisface B y C , o ϑ no satisface ni B ni C ;
3. Si $\mathcal{A} \equiv (\forall X)\mathcal{B}$, para **toda** valoración ϑ_X X -equivalente de ϑ , ϑ_X satisface \mathcal{B} .
4. Si $\mathcal{A} \equiv (\exists X)\mathcal{B}$, para **alguna** valoración ϑ_X X -equivalente de ϑ , ϑ_X satisface \mathcal{B} .

Verdad

Dada una fbf \mathcal{A} de \mathcal{L} , es conveniente introducir la siguiente nomenclatura:

1. Una fbf \mathcal{A} es *verdadera* en la interpretación I si y solo si toda valoración ϑ en I satisface \mathcal{A} .
2. Una fbf \mathcal{A} es *falsa* en I si y solo si no existe valoración ϑ en I que satisfaga \mathcal{A} . Escribiremos $I \models \mathcal{A}$ para denotar que \mathcal{A} es verdadera en I .
3. \mathcal{A} es *lógicamente válida* (denotado $\models \mathcal{A}$) si y solo si para toda interpretación I , \mathcal{A} es verdadera en I .
4. \mathcal{A} es *insatisfacible* si y solo si para toda interpretación I , \mathcal{A} es falsa en I .
5. \mathcal{A} es *satisfacible* si y solo si existe una interpretación I y una valoración en ella de las variables libres tal que ϑ satisface \mathcal{A} en I .

En el contexto de la lógica de proposiciones, una fórmula que se evalúa a verdadero, cualquiera que sea la valoración de sus variables enunciativas, se denomina *tautología*, mientras que si se evalúa siempre a falso se denomina *contradicción*. En la lógica de predicados, las fórmulas que provienen de tautologías (resp. contradicciones), por sustitución de las variables enunciativas de éstas últimas por fbf de la lógica de predicados, se denominan tautologías (resp. contradicciones) del lenguaje \mathcal{L} . En ocasiones, por abuso de lenguaje, hablaremos indistintamente de fórmula tautológica o lógicamente válida (resp. de fórmula contradictoria o insatisfacible). Sin embargo, observe que si bien toda tautología de \mathcal{L} es una fórmula lógicamente válida, la implicación inversa no es cierta. Por ejemplo, piense en el axioma K4, que es una fórmula lógicamente válida de \mathcal{L} pero que no proviene de una tautología, por sustitución sus variables enunciativas. Así pues, en la lógica de predicados hay fórmulas lógicamente válidas que no coinciden con tautologías de \mathcal{L} . Lo mismo puede decirse con respecto a las contradicciones.

Un resultado interesante, que tiene repercusiones cuando se estudia la forma clausal, es el siguiente.

Proposición 2.1 Sean Y_1, \dots, Y_n variables de \mathcal{L} , sea \mathcal{A} una fbf de \mathcal{L} e \mathcal{I} una interpretación de \mathcal{L} . Entonces, $\mathcal{I} \models \mathcal{A}$ si y solo si $\mathcal{I} \models (\forall Y_1) \dots (\forall Y_n) \mathcal{A}$.

Un aspecto importante del anterior resultado es que al añadir un cuantificador universal para una variable X que aparece libre en \mathcal{A} , la nueva fórmula sigue siendo verdadera en \mathcal{I} . De este modo, cuando consideramos la verdad o falsedad de fbf con variables libres, los cuantificadores universales se sobreentienden en cierto sentido.

Modelos y consecuencia lógica

El valor de verdad de una fórmula cerrada no depende de la valoración concreta ϑ en \mathcal{I} . Si encontramos una valoración ϑ que satisface una fórmula en \mathcal{I} entonces cualquier otra valoración también la satisfará. Si una valoración no la satisface sucede lo contrario, es decir, ninguna valoración la satisfará y la fórmula será falsa en \mathcal{I} . Este resultado se formaliza en la siguiente proposición.

Proposición 2.2 Sea \mathcal{A} una fbf cerrada de \mathcal{L} e \mathcal{I} una interpretación de \mathcal{L} . Entonces, $\mathcal{I} \models \mathcal{A}$ o $\mathcal{I} \models \neg \mathcal{A}$.

Dado que para fbf cerradas los conceptos de satisfacción por una valoración en \mathcal{I} y verdad en \mathcal{I} son equivalentes, el concepto de “satisfacible” puede expresarse en términos de verdad en una interpretación \mathcal{I} :

Una fbf cerrada \mathcal{A} es *satisfacible* si y solo si existe una interpretación \mathcal{I} en la cual \mathcal{A} sea verdadera.

También decimos que \mathcal{I} *satisface* \mathcal{A} o que \mathcal{I} es *modelo* de \mathcal{A}

Definición 2.2 (Modelo) Dada una fbf cerrada \mathcal{A} de \mathcal{L} , decimos que una interpretación \mathcal{I} es *modelo* de \mathcal{A} si y solo si la fbf \mathcal{A} es verdadera en la interpretación \mathcal{I} .

El concepto de modelo se extiende fácilmente a un conjunto de fbf cerradas.

Definición 2.3 Sea Γ un conjunto de fbf cerradas de \mathcal{L} , sea \mathcal{I} una interpretación de \mathcal{L} . \mathcal{I} es *modelo* de Γ si y solo si \mathcal{I} es modelo para cada una de las fórmulas de Γ .

Nótese que la noción de modelo de la Definición 2.2 es más rica que la introducida al comienzo de este apartado. En lo que sigue reservaremos el término *modelo* para referirnos a una interpretación distinguida en la que una fórmula o conjunto de fórmulas tendrán el significado deseado, en otras palabras, se hacen verdaderas.

Ahora podemos extender los conceptos de válido, insatisfacible y satisfacible a conjuntos de fbf cerradas en los siguientes términos.

Definición 2.4 Sea Γ un conjunto de fbf cerradas de \mathcal{L} .

1. Γ es válido si y solo si para toda interpretación I de \mathcal{L} , I es modelo de Γ .
2. Γ es insatisfacible si y solo si no existe una interpretación I de \mathcal{L} que sea modelo de Γ .
3. Γ es satisfacible si y solo si existe una interpretación I de \mathcal{L} que es modelo de Γ .

Uno de los conceptos más importantes de este apartado es el de consecuencia lógica.

Definición 2.5 (Consecuencia lógica) Sea Γ un conjunto de fbf cerradas de \mathcal{L} . Sea \mathcal{A} una fbf cerrada de \mathcal{L} . \mathcal{A} es consecuencia lógica de Γ (denotado $\Gamma \models \mathcal{A}$) si y solo si toda interpretación I de \mathcal{L} que es modelo de Γ también es modelo de \mathcal{A} .

Al igual que se ha establecido una correspondencia entre deducción y demostración (c.f. Corolario 2.1), ahora estableceremos una correspondencia similar entre los conceptos de consecuencia lógica y fórmula tautológica o válida.

Teorema 2.3 (Teorema de la deducción, versión semántica) Sea \mathcal{B} una fbf cerrada y $\Gamma = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$ un conjunto de fbf cerradas de \mathcal{L} . Entonces $\Gamma \models \mathcal{B}$ si y solo si $\models (\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_n) \rightarrow \mathcal{B}$.

Por las leyes de equivalencia lógica ya estudiadas, esto es lo mismo que decir que $\Gamma \models \mathcal{B}$ si y solo si la fbf $(\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_n \wedge \neg \mathcal{B})$ es insatisfacible (en términos conjuntistas, diríamos que $\Gamma \cup \{\neg \mathcal{B}\}$ es insatisfacible).

Otro resultado interesante es el siguiente, que caracteriza la insatisfacibilidad de un conjunto de fbf en términos de la posibilidad de que una contradicción sea consecuencia lógica del mismo.

Proposición 2.3 [Caracterización de la insatisfacibilidad] Sea Γ un conjunto de fbf cerradas de \mathcal{L} . Γ es insatisfacible si y solo si existe una fbf cerrada \mathcal{A} de \mathcal{L} , tal que $\Gamma \models (\mathcal{A} \wedge \neg \mathcal{A})$.

2.2.4. Propiedades de la lógica de predicados

En este apartado revisamos las principales propiedades de la lógica de predicados como sistema forma, poniendo de manifiesto las relaciones entre sintaxis y semántica.

Consistencia

La consistencia es un concepto que se asocia a conjuntos de fórmulas y se define en los siguientes términos: un conjunto Γ de fbf es *consistente* si de él no puede deducirse cualquier fbf de \mathcal{L} . Se dice que una fórmula \mathcal{A} es consistente si el conjunto $\{\mathcal{A}\}$ lo

es. La siguiente proposición puede considerarse una forma alternativa de caracterizar la consistencia de un conjunto de fórmulas:

Proposición 2.4 [Caracterización de la consistencia] Un conjunto Γ de fbf es *consistente* si y solo si no existe una fbf \mathcal{A} tal que $\Gamma \vdash \mathcal{A}$ y $\Gamma \vdash \neg \mathcal{A}$.

Del resultado anterior se deduce que Γ es *inconsistente* si y solo si es posible derivar de ésta una contradicción; es decir, existe una fbf \mathcal{A} tal que $\Gamma \vdash (\mathcal{A} \wedge \neg \mathcal{A})$.

Como es deseable, se puede demostrar que el propio sistema \mathcal{K}_L es consistente. Esto es, dada una fbf \mathcal{A} no es posible deducir \mathcal{A} y $\neg \mathcal{A}$.

Corrección y Completitud

La lógica de predicados es correcta y completa. Es decir, dada una fbf \mathcal{A} , $\vdash \mathcal{A}$ si y solo si $\models \mathcal{A}$.

Para fbf cerradas existe una perfecta equivalencia entre el concepto sintáctico de deducción y el concepto semántico de consecuencia lógica. Esto viene expresado por la siguiente proposición.

Proposición 2.5 Sea Γ un conjunto de fbf cerradas⁸ y \mathcal{A} una fbf cerrada de \mathcal{L} . $\Gamma \models \mathcal{A}$ si y solo si $\Gamma \vdash \mathcal{A}$.

De este resultado se sigue trivialmente la equivalencia entre el concepto sintáctico de inconsistencia y el concepto semántico de insatisfacibilidad. Es decir, un conjunto de fbf cerradas Γ es insatisfacible si y solo si es inconsistente.

Decidibilidad

La lógica de predicados es *indecidable*, esto es, no existe un procedimiento finito para decidir si una fórmula es válida o no. Informalmente, el problema radica en que se pueden formar infinitos dominios de interpretación y todos han de ser considerados. Sin embargo, si la fórmula realmente es válida, hay procedimientos de prueba que pueden verificarlo. Estos procedimientos, en general, no terminan si la fórmula no es válida (son procedimientos de *semidecisión*), por este motivo, también se dice que la lógica de predicados es *semidecidible*. Uno de estos algoritmos subceptible de automatización fué definido por Herbrand en 1930. Es un algoritmo basado en criterios semánticos. La idea básica es fijar un dominio sintáctico de interpretación especial, llamado *universo de Herbrand*, que se puede construir mecánicamente y que tiene la propiedad de que

⁸La necesidad de que las fbf sean cerradas se debe a que en la prueba de esta proposición interviene el teorema de la deducción (Teorema 2.1), que en su formulación actual solo se aplica a fbf cerradas.

las interpretaciones sobre este dominio son suficientes para caracterizar la validez de la fórmula (si realmente lo es) y pueden generarse de forma automática. Estudiaremos este mecanismo en el próximo capítulo.

2.3 SEMÁNTICAS Y LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación puede considerarse que es un lenguaje formal desde el momento que su sintaxis y su semántica están definidas formalmente. La primera se ocupa de fijar las reglas que describen las estructuras sintácticas (formalmente correctas) que constituyen los programas, es decir, la forma de las instrucciones, las declaraciones, las expresiones aritméticas y otras construcciones del lenguaje. La segunda se ocupa de asignar un significado a cada una de esas construcciones sintácticas del lenguaje. La teoría de autómatas y lenguajes formales proporciona herramientas para la definición sintáctica de los lenguajes de programación (gramáticas, notaciones, etc). Respecto a la semántica, las distintas “escuelas” o familias de lenguajes se diferencian en la forma con la que enfocan la descripción de los aspectos semánticos dependiendo de su uso y de la riqueza que se pretende alcanzar en la descripción: operacional; axiomático o declarativo (con sus diversas orientaciones) son los más representativos.

En los próximos apartados justificamos la necesidad de una definición semántica (formal) y resumimos las características de algunas de las aproximaciones semánticas.

2.3.1. Necesidad de una definición semántica formal

Si bien la definición sintáctica de un lenguaje de programación mediante una gramática formal (descrita empleando algún tipo de notación para la descripción de sus reglas de producción; e.g., la notación BNF — *Backus-Naur Form*) facilita su análisis sintáctico, ésta no es suficiente para la comprensión de sus estructuras sintácticas. Por ejemplo, ¿cómo se interpretan los operadores? basta observar que el operador `**` es la exponentiación en Fortran mientras puede corresponder a la concatenación de cadenas en una variante de Lisp. A continuación se discuten algunos ejemplos que resaltan otros aspectos de este problema.

Ejemplo 2.4

Fijémonos en la definición sintáctica de programa para el lenguaje Pascal:

```

⟨Program⟩ ::= ⟨Program_head⟩⟨Block⟩‘.’
⟨Program_head⟩ ::= ‘program’⟨Identifier⟩[‘⟨Identifier_list⟩’]‘;’
⟨Identifier_list⟩ ::= ⟨Identifier⟩{‘,’⟨Identifier⟩}

```

Se ha empleado notación BNF extendida en la que los nombres encerrados entre paréntesis representan símbolos no terminales y los encerrados entre comillas simples símbolos terminales. El símbolo “`::=`”, los corchetes y las llaves son metasímbolos de la

notación BNF. Los corchetes representan opcionalidad y las llaves indican repetición de cero uno o más de los elementos encerrados entre ellas. Cada regla define un símbolo no terminal y operacionalmente debe entenderse como sigue: si en una cadena de símbolos aparece un símbolo no terminal, éste puede sustituirse por la correspondiente parte derecha de la regla que lo define. Si bien estas reglas nos permiten escribir con corrección la cabecera de un programa, cabe preguntarse por el significado de la lista de identificadores: ¿Los identificadores que siguen al nombre del programa, a qué hacen referencia?, ¿son nombres de ficheros de entrada/salida o qué son? Un desconocedor del lenguaje Pascal difícilmente responderá a esta pregunta con la sola información suministrada por la descripción sintáctica.

Si bien ciertas restricciones del lenguaje no son expresables en BNF y forman parte por tanto de la descripción semántica del lenguaje, en la mayoría de los manuales y libros que describen un lenguaje de programación, es habitual organizar la descripción del lenguaje alrededor de las diversas estructuras sintácticas que posee. En estos manuales se define la sintaxis de una construcción y se pasa a dar el significado de la misma indicando sus características de funcionamiento mediante el empleo de un lenguaje natural (español, inglés, etc.). Podríamos denominar *semántica informal* a esta forma de descripción. La definición semántica mediante el lenguaje natural (en apoyo de las reglas BNF o cualquier otra forma de expresar la sintaxis) encierra ambigüedades y en ocasiones las descripciones empleadas no suelen ser lo suficientemente precisas como para captar todos los matices del significado de una construcción. Esto puede conducir a errores de programación y a, lo que es aún peor, errores de implementación cuando se utiliza una semántica informal como base para la descripción del propio lenguaje.

Ejemplo 2.5

En el libro de Kernighan y Ritchie [76], que describe el lenguaje C, puede leerse respecto a las expresiones de asignación:

Existen varios operadores de asignación; todos se agrupan de derecha a izquierda.

expresión-asignación:

expresión-condicional

expresión-unaria operador-asignación expresión-asignación

operador-asignación:

= *= /= %= += ...

Todos requieren de un valor-l como operando izquierdo y éste debe ser modificable ... El tipo de una expresión de asignación es el de su operando izquierdo, y el valor es el almacenado en el operando izquierdo después de que ha tenido lugar la asignación.

En la asignación simple con =, el valor de la expresión reemplaza al del objeto al que se hace referencia con el valor-l. Uno de los siguientes hechos

debe ser verdadero: ambos operandos tienen tipo aritmético, en tal caso, el operando de la derecha es convertido al tipo del operando de la izquierda por la asignación; o ...

El primer comentario que se nos ocurre al leer esta descripción es el siguiente: ¿Qué quieren decir los autores cuando se refieren a que “Todos requieren de un valor-l como operando izquierdo”?⁹ Más aún, para un lector que se enfrentara por primera vez al lenguaje, ¿bastaría con esta descripción para contestar a la pregunta de en qué contextos podría emplearse una asignación¹⁰? Por otra parte, supongamos que en un programa aparece la instrucción: $x = 2,45 + 3,67$. La simple definición sintáctica de la instrucción de asignación no nos indica si la variable x ha sido declarada o, si lo fue, si se hizo con tipo real. Por consiguiente no podríamos decir a simple vista que resultado se obtendría al evaluar esa expresión. Existen las siguientes posibilidades:

- x toma el valor 6, si x se refiere a enteros;
- x toma el valor 6,12, si x se refiere a reales;

Esto muestra que es necesaria información sobre el entorno en el que se está utilizando una instrucción para comprender todo su significado. Por otro lado, es habitual distinguir dos tipos de semántica: estática y dinámica. La primera se refiere a las restricciones de sintaxis que no son expresables en BNF pero que pueden comprobarse en tiempo de compilación (precisamente durante la fase de análisis semántico). Las demás restricciones, que no se pueden comprobar en tiempo de compilación y solo durante la ejecución del programa, constituyen la semántica dinámica, e.g. la comprobación de índices dentro del rango del vector. El intento de acceder a una componente del vector en una posición del índice no válida o fuera de rango producirá un error de ejecución, como también lo haría la división de dos enteros x e y cuando el valor de y es cero al ejecutar el operador.

Para resolver problemas como el que acabamos de ilustrar, y en particular formalizar la semántica dinámica, algunos autores [122] abordan la descripción de los lenguajes de programación a partir de una organización basada en las estructuras de datos y operaciones de una *máquina virtual*, en lugar de alrededor de las estructuras sintácticas del lenguaje. En ocasiones estas estructuras de datos y operaciones están ligadas directamente a una estructura sintáctica, pero a menudo, como en el caso de la instrucción de asignación, la máquina virtual debe usar información dada por una instrucción de declaración previa e incluso estructuras de datos que no están ligadas directamente a la construcción sintáctica considerada.

Otra forma de dar significado a los lenguajes de programación ha sido la llamada *semántica por traducción* que consiste en definir el significado de un lenguaje por medio

⁹Este es un término técnico del que se da razón en [76] algunos apartados antes: Un *objeto* es una región de almacenamiento con nombre; un *valor-l* es una expresión que se refiere a un objeto. Un ejemplo obvio de una expresión valor-l es un identificador.

¹⁰Naturalmente, pueden emplearse como una instrucción; ahora bien, dado que en el lenguaje C cualquier asignación se considera una expresión y tiene un valor, esto significa que una asignación puede formar parte de otra expresión más compleja (e.g., `(c = getchar()) == 'b'`).

de la traducción de sus construcciones sintácticas a un lenguaje ya conocido. Esto puede hacerse bien por una técnica de *interpretación* o de *compilación*. Si disponemos de una máquina concreta M cuyo conjunto de instrucciones y funcionamiento es conocido, un *compilador* que transforme programas escritos en nuestro lenguaje de programación (*lenguaje fuente*) en programas del lenguaje de la máquina M (*lenguaje objeto*), sería una definición semántica del lenguaje que nos permite definir el significado de los programas del lenguaje de programación como el programa obtenido tras la compilación. Sin embargo un intérprete o un compilador no se consideran propiamente definiciones semánticas ya que este tipo de definiciones presenta varios problemas:

1. Definir las instrucciones del lenguaje en términos de las de una máquina real M aporta un grado de *complejidad* que no está presente en el propio lenguaje sino en el modelo elegido para su representación. Esto conduce a una *falta de abstracción* en la descripción semántica, que se pierde en los detalles.
2. La *falta de portabilidad* sería otro de los problemas. En este caso la definición semántica depende de la máquina M escogida. Claramente, si cambiamos a una máquina M' , el traductor deberá reescribirse en términos del lenguaje máquina de M' o la máquina M deberá simularse en términos de la M' .
3. Una de las aplicaciones de las definiciones semánticas es servir de ayuda a la construcción de traductores de lenguajes, así pues, esta orientación subvertiría uno de sus objetivos.

Los dos primeros problemas pueden suavizarse si definimos una máquina virtual, más abstracta, que sea fácil y eficientemente simulada por las máquinas reales con distintos conjuntos de instrucciones. Dicha máquina sería una abstracción de las características presentes en todas ellas. El conjunto de instrucciones de la máquina virtual permitiría definir un *código intermedio* como resultado de la compilación del lenguaje fuente¹¹. Sin embargo, dependiendo del nivel de abstracción, la definición podría no servir para algunos usos pretendidos de la misma. La discusión precedente sugiere la necesidad de dotar a los lenguajes de programación de una *semántica formal*, para asegurar que las definiciones de las construcciones del lenguaje sean claras y evitar ambigüedades en su interpretación.

Las semánticas formales se definen mediante:

- Una descripción formal de la clase de objetos que pueden manipularse por las construcciones del lenguaje. Estos objetos pueden ser: estados de una máquina (abstracta), funciones, valores, asertos u otra clase de objetos que deberán especificarse formalmente.

¹¹ Esta solución, introducida en los años 70 en los lenguajes Pascal y Prolog, ha sido adoptada en la implementación del lenguaje Java para aumentar su portabilidad.

- Un conjunto de reglas que describen las distintas formas en que las expresiones del lenguaje adquieren significado y pueden combinarse para dar la salida (asociada a cada una de las expresiones compuestas) en términos de sus constituyentes.

Al dotarnos de una semántica formal para un lenguaje de programación queremos construir un modelo matemático que nos permita comprender y razonar sobre el comportamiento de los programas [143], así como construir herramientas útiles para su manipulación automática (transformación, verificación, depuración, etc). También queremos establecer con precisión criterios para comprobar la equivalencia entre programas. Esto es de singular importancia, entre otras cosas, en el estudio de técnicas de transformación de programas, en las que se pretende establecer la equivalencia entre el programa original y el transformado como una simple consecuencia de la corrección y completitud del método de transformación.

2.3.2. Semánticas formales de los lenguajes de programación

Debido al hecho de que se requiere estudiar las características de los lenguajes de programación con distintos niveles de detalle y diferentes perspectivas, se han propuesto diversas aproximaciones a la semántica formal de los lenguajes de programación. Aquí vamos a describir sucintamente algunas de las más importantes. Una introducción asemejable a este tema puede encontrarse en [110].

Semántica operacional

Es el enfoque más antiguo de definición semántica formal de un lenguaje (con este estilo se definió la semántica de ALGOL'60). La semántica operacional proporciona una definición del lenguaje en términos de una máquina abstracta (de estados). La idea es muy sencilla, debemos de dar una noción de estado abstracto que contenga la información esencial para describir el proceso de ejecución de un programa del lenguaje en esa máquina. Entonces, la semántica de un lenguaje de programación se define indicando cuál es el efecto de las construcciones sintácticas del lenguaje sobre el estado de la máquina abstracta. Esto es, el significado de una instrucción se define en términos del cambio de estado (abstracto) que produce cuando se ejecuta. Intuitivamente, la semántica operacional equivale a la definición de un intérprete (abstracto) para el lenguaje, que lo hace ejecutable.

A menudo, la especificación operacional de un lenguaje se lleva a cabo mediante el uso de *sistemas de transición* [64] y las distintas fases por las que atraviesa la ejecución de un programa se representan mediante diferentes *configuraciones* de la máquina abstracta. Un ejemplo de este enfoque son los sistemas de transición de Plotkin [121].

En los próximos capítulos ampliaremos la información sobre la semántica operacional de los lenguajes de programación lógica.

Tabla 2.1 Axiomas para un fragmento de un lenguaje imperativo.

Regla	Antecedente	Consecuente
1. Consec_1	$\{\mathcal{A}\}I(\mathcal{B}), \{\mathcal{B}\} \vdash C$	$\{\mathcal{A}\}I(C)$
2. Consec_2	$\{C\} \vdash \mathcal{A}, \{\mathcal{A}\}I(\mathcal{B})$	$\{C\}I(\mathcal{B})$
3. Compos.	$\{\mathcal{A}\}I_1(\mathcal{B}), \{\mathcal{B}\}I_2(C)$	$\{\mathcal{A}\}I_1; I_2(C)$
4. Asign.	$X = \text{exp}$	$\{\mathcal{A}(\text{exp})\}X = \text{exp}(\{\mathcal{A}(X)\})$
5. if	$\{\mathcal{A} \wedge B\}S_1(C), \{\mathcal{A} \wedge \neg B\}S_2(C)$	$\{\mathcal{A}\}\text{if } B \text{ then } S_1 \text{ else } S_2(C)$
6. while	$\{\mathcal{A} \wedge B\}S_1(C)$	$\{\mathcal{A}\}\text{while } B \text{ do } S \{\mathcal{A} \wedge \neg B\}$

Semántica axiomática

Este es el enfoque típico de los trabajos sobre verificación formal de programas imperativos. Con este estilo se definió la semántica del lenguaje Pascal. La idea básica consiste en definir el significado de un programa como la relación existente entre las propiedades que verifica su entrada (*precondiciones*) y las propiedades que verifica su salida (*postcondiciones*). Las precondiciones y postcondiciones son fórmulas de algún sistema lógico que especifican el significado del programa. Dichas fórmulas pueden ser enunciados sobre el estado de la computación, pero no necesariamente debe ser así. Por ejemplo, la semántica axiomática de un programa que calcula el cociente C y resto R de dos enteros X e Y vendría definida por la precondición $\{Y \geq 0\}$ y la postcondición $\{X = Y * C + R\}$. Como se ha dicho, la semántica axiomática se emplea principalmente en tareas de verificación de programas y suele emplearse la notación $\{\mathcal{A}\}I(\mathcal{B})$, que indica que si el enunciado \mathcal{A} es verdadero antes de la ejecución de la instrucción I , entonces el enunciado \mathcal{B} es verdadero después de su ejecución (supuesto que la ejecución de I termina). La semántica axiomática utiliza las reglas de inferencia del cálculo de la lógica de predicados introduciendo además otras reglas de inferencia adicionales (véase la Tabla 2.1) que establecen el significado de cada estructura sintáctica del lenguaje (instrucción de asignación, condicional, iteración, etc).

Habitualmente la verificación de programas procede hacia atrás (*backward*) en el código: conocida la postcondición de una instrucción, se deriva qué precondición debe de ser cierta antes de su ejecución para asegurar lo que es cierto después (caracterizado por la postcondición); en concreto se calcula la precondición más débil que garantiza el tránsito al estado caracterizado por la postcondición. Por ejemplo, si la postcondición de una instrucción de asignación $X = Y + Z$ es $X > 0$ entonces, empleando el axioma de asignación 4, donde ahora $\mathcal{A}(X) \equiv (X > 0)$, tenemos que $\{Y + Z > 0\}X = Y + Z\{X > 0\}$. Por consiguiente, la precondición de esta instrucción de asignación debe ser $Y + Z > 0$. Empleando esta técnica se puede verificar, no sin dificultad, la corrección de pequeños programas imperativos: basta comprobar que la precondición del programa implica la precondición (más débil) que se obtiene para la primera instrucción del programa (la última en procesarse ya que el método parte de la postcondición). Sin embargo, hay que

admitir que cuando el tamaño de los programas es considerable esta técnica se torna impracticable. Esta dificultad también puede verse como un reflejo de la propia complejidad semántica de los lenguajes convencionales.

Semántica declarativa

En esta clase de semánticas el significado de cada construcción sintáctica se define en términos de elementos y estructuras de un dominio matemático conocido. Dependiendo del dominio escogido, este enfoque ha dado lugar a diferentes aproximaciones:

- 1. Semántica por teoría de modelos.**

Esta aproximación está basada en la teoría de modelos lógica, que es bien conocida por haber sido estudiada en el Apartado 2.2.3. Con este estilo se ha definido la semántica de lenguajes de programación lógica como Prolog, que estudiaremos con todo detalle en próximos capítulos.

- 2. Semántica algebraica.**

Esta aproximación está basada en la teoría de álgebras libres. Con este estilo se ha definido la semántica de lenguajes ecuacionales de primer orden, una clase especial de lenguajes funcionales donde los programas son conjuntos de ecuaciones entre términos que definen funciones. En ellos un programa se interpreta haciendo uso de nociones algebraicas como álgebra cociente y álgebra modelo de un conjunto de ecuaciones. También se ha utilizado para definir la semántica de los lenguajes de especificación de tipos abstractos de datos como OBJ.

- 3. Semántica de punto fijo.**

Esta aproximación está basada en la teoría de funciones recursivas. Generalmente se utiliza como enlace para demostrar la equivalencia entre diferentes caracterizaciones semánticas de un mismo lenguaje. Se asocia al programa un operador (generalmente continuo) que opera sobre el dominio de interpretación, de manera que el significado del programa se define como el (menor) punto fijo de dicho operador. Con este enfoque se ha definido la semántica de lenguajes lógicos, que estudiaremos en próximos capítulos.

- 4. Semántica denotacional.**

Esta aproximación está basada en la teoría de dominios introducida originalmente por Scott [134, 140]. La idea fundamental de este tipo de semánticas es interpretar las construcciones sintácticas de un lenguaje como valores de un determinado *dominio*¹² matemático. La semántica denotacional de un programa viene dada por la

¹²La palabra “dominio” se emplea con una connotación más rica que la empleada en la teoría de modelos de la lógica de predicados (y también en otras aproximaciones semánticas). En la teoría de modelos, un dominio es un conjunto simple, sin ningún tipo de estructura. En el ámbito de la semántica denotacional, un dominio es un conjunto dotado de una estructura más elaborada que se caracteriza por poseer, al menos, un orden parcial completo con un elemento mínimo.

función que éste denota. La semántica denotacional se centra en “lo esencial” de la computación, es decir en el resultado final obtenido. Así pues la semántica denotacional posee un grado de abstracción mayor que el de la semántica operacional, ya que no se fija en los estados intermedios (de una máquina abstracta) generados durante la ejecución de un programa, lo único importante son las funciones asociadas a los programas, no cómo se construyen; en cambio, en una semántica operacional los estados intermedios forman parte de la definición [110]. Informalmente, podemos decir que la semántica denotacional es más declarativa que la semántica operacional.

Técnicamente, la semántica denotacional es generalmente la más compleja de las caracterizaciones semánticas, si bien es muy rica, ya que permite dar cuenta de computaciones que no terminan y del orden superior en los lenguajes de programación. Para su definición se requiere describir: i) por cada construcción sintáctica un dominio sintáctico; en general, tendrán que definirse dominios sintácticos para los identificadores, las constantes, los operadores, las expresiones, las instrucciones (o fórmulas) y los programas; ii) los dominios semánticos; uno por cada dominio sintáctico identificado; iii) las funciones de evaluación semántica, de los dominios sintácticos a los semánticos, que asocian a cada construcción sintáctica su valor semántico; iv) un conjunto de ecuaciones semánticas. Una característica importante de las descripciones denotacionales es que las funciones de evaluación semántica vienen definidas por inducción sobre la estructura de las expresiones. Esto da lugar a un estilo de definición composicional donde el valor de una estructura sintáctica queda completamente determinado en función del valor de sus componentes.

Con este estilo se ha definido la semántica declarativa de la mayoría de los lenguajes funcionales (e.g., Haskell y ML) así como la del lenguaje imperativo ADA.

2.3.3. Lenguajes de Programación Declarativa

En el Apartado 2.1 estudiábamos como todo sistema formal se caracteriza por tener una sintaxis y semántica definidas matemáticamente (junto con la relación entre ambas). Del mismo modo, es posible definir un lenguaje de programación declarativa como un sistema constituido por tres componentes:

1. La sintaxis, en la que las construcciones del lenguaje son fórmulas de algún sistema lógico, con reglas claras que describen cómo combinar los símbolos de un alfabeto para formar las fbf del lenguaje. Los programas son conjuntos de fbf en esa lógica.
2. La semántica operacional, que se define en términos de una forma razonablemente eficiente de deducción en esa lógica. Este componente se corresponde con el cálculo deductivo de los sistemas formales tradicionales, por lo que tiene un ca-

rácter sintáctico en el sentido de no hacer asunciones sobre el posible significado de los símbolos, que se manipulan de forma automática.

3. La semántica declarativa, que se define en términos de un modelo lo más simple posible del programa, dentro de la riqueza descriptiva que se pretende.

Finalmente, deben de existir resultados de corrección y completitud que pongan en correspondencia los dos últimos componentes, de forma que las definiciones semánticas del lenguaje coincidan; es decir, que asignen al programa un significado único.

En los próximos capítulos describiremos los lenguajes de programación declarativa analizando cada uno de estos tres componentes.

RESUMEN

En este capítulo hemos estudiado las conexiones existentes entre la lógica y los lenguajes de programación, cuando se les considera como sistemas formales, y cómo la caracterización semántica es una parte imprescindible de la definición de un lenguaje de programación.

- Se ha comenzado por definir el concepto de sistema formal como un modelo de razonamiento matemático compuesto de un lenguaje formal y un cálculo deductivo. Siguiendo a Carnap [24], se añade un componente semántico que es muy conveniente después desde el punto de vista del estudio de los lenguajes de programación.
- Dado que el conocimiento de la lógica de predicados es muy conveniente para la comprensión de un lenguaje de programación declarativa, en este capítulo se han revisado los principales conceptos y notaciones sobre la lógica de predicados. Nos hemos centrado en las nociones de deducibilidad, interpretación y consecuencia lógica y en las propiedades formales: corrección, consistencia, completitud y decidibilidad.
- Mediante la discusión de diferentes ejemplos hemos constatado la necesidad de dotar a los lenguajes de programación de una semántica formal, para asegurar que las definiciones de las construcciones de los lenguajes sean claras y evitar ambigüedades en su interpretación.
- Las semánticas formales se definen mediante:
 - Una descripción formal del universo de discurso, es decir, la clase de objetos que pueden ser manipulados por los programas del lenguaje. Estos objetos pueden ser: estados de una máquina (abstracta), funciones, valores, asertos u otra clase de objetos que deberán especificarse formalmente.

- Un conjunto de reglas que describen las distintas formas en que las expresiones del lenguaje pueden combinarse para dar la salida asociada a cada una de las expresiones en términos de sus constituyentes.
- Las semánticas formales nos permiten comprender y razonar sobre el comportamiento de los programas [143], facilitando el análisis y verificación de los mismos. También permiten establecer criterios precisos para comprobar la equivalencia entre programas.
- No hay un consenso generalizado a la hora de abordar los aspectos semánticos de los lenguajes de programación. Según el nivel de detalle y la perspectiva con la que se requiere estudiar las características de los lenguajes de programación, se han propuesto diversas aproximaciones a la semántica formal de los lenguajes de éstos: operacional, axiomática, declarativa. En capítulos posteriores dedicaremos especial atención a la primera y la última de estas visiones.
- La semántica operacional proporciona una definición del lenguaje en términos de una máquina abstracta (de estados). La idea consiste en seleccionar una noción de estado abstracto y el significado de una instrucción se define en términos del cambio de estado (abstracto) que produce ésta cuando se ejecuta. Intuitivamente, la semántica operacional equivale a la definición de un intérprete (abstracto) para el lenguaje, que lo hace ejecutable. A menudo, la especificación operacional de un lenguaje se lleva a cabo mediante el uso de *sistemas de transición* [64, 121].
- La semántica declarativa se fundamenta en la elección de un dominio matemático conocido. El significado de cada construcción sintáctica se define en términos de los elementos y estructuras de ese dominio. Este enfoque ha dado lugar a diferentes aproximaciones:
 1. **Semántica teoría de modelos.** Está basada en la teoría de modelos lógica.
 2. **Semántica algebraica.** Está basada en la teoría de álgebras libres.
 3. **Semántica de punto fijo.** Está basada en la teoría de funciones recursivas.
 4. **Semántica denotacional.** Está basada en nociones de la teoría de dominios.
- Es posible entender los lenguajes de programación como sistemas formales. Los lenguajes de programación declarativa son los que mejor se adaptan a esta visión. Un lenguaje de programación declarativa, como un sistema formal, está constituido por tres componentes: la sintaxis, la semántica operacional, y la semántica declarativa. Debe de existir una correspondencia precisa entre los dos últimos componentes de forma que las diferentes caracterizaciones semánticas del lenguaje coincidan (todo resultado computado por la semántica operacional debe ser correcto, y viceversa).

CUESTIONES Y EJERCICIOS

Cuestión 2.1 ¿Cuál de las siguientes afirmaciones, relativas al cálculo deductivo de un sistema formal, es cierta?:

- Permite deducir nuevas fbf teniendo en cuenta aspectos sintácticos y semánticos.
- Tiene que ver con la definición axiomática de una serie de estructuras inductivas correctas y las reglas para obtener otras nuevas a partir de aquéllas.
- No tiene en cuenta la interpretación de los símbolos del formalismo ni el significado de las fbf.

Cuestión 2.2 Un sistema formal es correcto si:

- Toda fórmula demostrable mediante el cálculo deductivo es válida según la noción de verdad de la semántica.
- Toda fórmula semánticamente válida es demostrable mediante el cálculo deductivo.
- Existe un procedimiento finito para decidir si una fórmula es o no válida.
- Su método de cálculo es el de la lógica de predicados de primer orden.

Seleccione la respuesta correcta de entre las anteriores.

Cuestión 2.3 Indique cuál de las siguientes afirmaciones es falsa:

- En un sistema formal los símbolos carecen de significado.
- Una fórmula bien formada es cualquier cadena de símbolos que tiene significado completo.
- La teoría de la demostración estudia los formalismos con independencia de toda interpretación.

Cuestión 2.4 Indique cuál de las siguientes expresiones de \mathcal{L} no es una expresión bien formada:

- | | |
|---------------------|--|
| ▪ $f_0(X, b, Y)$. | ▪ $(\exists X)(r(X, q(b)))$. |
| ▪ $r(a, f(g(a)))$. | ▪ $(\exists X)(r(X, h(c)) \rightarrow q(X))$. |

Ejercicio 2.5 Utilizando los resultados que facilitan la deducción en el sistema \mathcal{K}_L (teorema de la deducción, teorema de intercambio, introducción de teoremas ya probados, etc.), realice las siguientes deducciones:

1. $\{(\forall X)(p(X) \rightarrow q(X)), \neg q(a)\} \vdash_{\mathcal{K}_L} \neg p(a).$
2. $\{(\forall X)(q(X) \rightarrow r(X)), (\forall X)(p(X) \rightarrow q(X))\} \vdash_{\mathcal{K}_L} (\forall X)(p(X) \rightarrow r(X)),$
3. $\vdash_{\mathcal{K}_L} (\forall X)\mathcal{A}(X) \rightarrow (\exists X)\mathcal{A}(X).$
4. $\vdash_{\mathcal{K}_L} (\neg(\forall X)\mathcal{A}(X) \rightarrow (\exists X)\neg\mathcal{A}(X)).$
5. $\vdash_{\mathcal{K}_L} (\exists X)(\forall Y)\mathcal{A}(X, Y) \rightarrow (\forall Y)(\exists X)\mathcal{A}(X, Y).$
6. $\vdash_{\mathcal{K}_L} (\forall X)(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow ((\exists X)\mathcal{A} \rightarrow (\exists X)\mathcal{B}).$

Cuestión 2.6 a) ¿Qué se entiende por interpretación I de \mathcal{L} ? b) Definase el concepto de valoración en una interpretación.

Ejercicio 2.7 Las siguientes fórmulas tienen una interpretación evidente en el “mundo de los bloques”, suponiendo que las constantes a , b y c designan bloques concretos y la constante s el suelo:

$$\begin{array}{ll} \text{sobre}(c, a), & \text{libre}(c), \\ \text{sobre}(a, s), & \text{libre}(b), \\ \text{sobre}(b, s), & (\forall X)(\text{libre}(X) \rightarrow \neg(\exists Y)\text{sobre}(Y, X)). \end{array}$$

a) Represente gráficamente la situación reflejada por las fórmulas anteriores. b) Defina una nueva interpretación que satisfaga la conjunción de las anteriores fórmulas.

Ejercicio 2.8 Sea \mathcal{L} el lenguaje de primer orden que incluye (junto con los símbolos comunes de cualquier formalismo) la constante individual a , el símbolo de función f y el símbolo de relación r . Dada la interpretación I para \mathcal{L} cuyo dominio de interpretación es \mathbb{Z} , $\mathcal{J}(a)$ es 0, $\mathcal{J}(f(X, Y))$ es $X - Y$, $\mathcal{J}(r(X, Y))$ es $X < Y$, encuéntrese, si es posible, valoraciones que satisfagan y que no satisfagan las siguientes fbf.

1. $r(X, a),$
2. $\neg r(X, f(X, Y)),$
3. $r(f(X, Y), X) \rightarrow r(a, f(X, Y)),$
4. $(\forall X)r(f(X, Y), Z).$

Cuestión 2.9 Defina los siguientes conceptos: a) Fórmula verdadera, fórmula falsa y fórmula satisfacible por una valoración en una interpretación. b) Fórmula lógicamente válida, fórmula insatisfacible y fórmula satisfacible.

Ejercicio 2.10 Diga si las siguientes expresiones son fbf de la lógica de predicados, y si no lo son de una razón de por qué no lo son. Para cada una de las expresiones que sean fbf, diga si es una fórmula lógicamente válida, insatisfacible o satisfacible (razonar las respuestas). Para las que sean satisfacibles, dé una interpretación que la haga verdadera:

1. $\text{gano}(\text{aznar}, \text{lasElecciones}) \wedge \text{gano}(\text{zapatero}, \text{lasElecciones}),$

2. $(\forall S)(\text{matriculado}(S, (\text{asig12} \vee \text{asig22}))),$
3. $(\forall X)(\text{conjunto}(X) \rightarrow \text{subconjunto}(X, X) \vee \neg \text{subconjunto}(X, X)),$
4. $(\forall X)[p(X) \vee q(X)] \rightarrow \neg(\exists X)[\neg p(X) \vee q(X)],$
5. $(\forall X)\text{esCierto}(X),$
6. $p(a) \wedge q(b) \wedge (\forall X)\neg(p(X) \vee q(X)),$
7. $\neg(\exists X) \rightarrow \text{noExiste}(X).$

Cuestión 2.11 ¿Qué se entiende por interpretación modelo de una fórmula (cerrada)?

Cuestión 2.12 Sea Δ un conjunto de fórmulas cerradas de un lenguaje de primer orden \mathcal{L} . Se dice que Δ es satisfacible si:

- Cada interpretación de \mathcal{L} es modelo de Δ .
- \mathcal{L} tiene (al menos) una interpretación que es modelo de Δ .
- Ninguna interpretación de \mathcal{L} es modelo de Δ .
- \mathcal{L} tiene una interpretación que no es modelo de Δ .

Ejercicio 2.13 Dada la interpretación $I = \langle \mathcal{D}, \mathcal{J} \rangle$, donde $\mathcal{D} = \{0, 1\}$ y \mathcal{J} es la función de interpretación que asigna: $\mathcal{J}(a) = 0$, e interpreta los símbolos f y p de acuerdo con la siguiente tabla:

$\bar{f}(0)$	$\bar{f}(1)$	$\bar{p}(0)$	$\bar{p}(1)$	$\bar{q}(0, 0)$	$\bar{q}(0, 1)$	$\bar{q}(1, 0)$	$\bar{q}(1, 1)$
1	0	F	V	V	V	F	V

Compruebe de cuál de las siguientes fórmulas es modelo I :

1. $(\exists X)(p(X) \wedge Q(X, a)).$
2. $(\exists X)(p(f(X)) \wedge q(X, f(a))).$
3. $(\forall X)(\exists Y)(p(X) \wedge q(X, Y)).$
4. $(\forall X)(\forall Y)(p(X) \rightarrow q(X, Y)).$

Cuestión 2.14 Definase el concepto de consecuencia lógica.

Ejercicio 2.15 Sea \mathcal{B} una fbf cerrada y $\Gamma = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$ un conjunto de fbf cerradas de \mathcal{L} . Demuestre que $\Gamma \models \mathcal{B}$ si y solo si $\Gamma \cup \{\neg \mathcal{B}\}$ es insatisfacible.

Cuestión 2.16 Indique cuál de las siguientes afirmaciones es falsa:

- El teorema de la deducción (versión sintáctica) permite relacionar los conceptos de deducción y de demostración.
- El teorema de la deducción (versión semántica) relaciona los conceptos de validez y consecuencia lógica.
- La relación entre fórmula válida y consecuencia lógica es comparable a la relación entre demostrable y deducible.
- En la lógica de predicados, el teorema de la completitud establece que $\vdash \mathcal{A}$ implica $\models \mathcal{A}$ siendo \mathcal{A} una fbf de la lógica.

Ejercicio 2.17 Demuestre que un conjunto de fórmulas Γ es inconsistente si y solo si existe una fbf \mathcal{A} tal que $\Gamma \vdash \mathcal{A}$ y $\Gamma \vdash \neg \mathcal{A}$.

Cuestión 2.18 La lógica de predicados, ¿es decidible? Razoné brevemente su respuesta.

Cuestión 2.19 Los lenguajes imperativos no sirven como lenguajes de especificación porque:

- No tienen una sintaxis y una semántica bien definidas.
- Introducen detalles que no son característicos del problema sino de la solución encontrada.
- No tienen una base lógica.
- No son ejecutables.

Seleccione la respuesta correcta de entre las anteriores.

Cuestión 2.20 Indique cuál de las siguientes sentencias es falsa:

- La semántica denotacional proporciona una definición del lenguaje en términos de una máquina abstracta (de estados).
- Las semánticas formales facilitan las tareas de análisis y verificación de programas.
- En la semántica por traducción el significado del programa (fuente) es el mismo que el del programa obtenido tras la compilación.

Cuestión 2.21 Describa las principales características de la semántica operacional.

Cuestión 2.22 Indique qué elección para la definición semántica de un lenguaje de programación resultaría apropiada, si quisieramos emplearla como ayuda ...

- ... a la programación: *Semántica por punto fijo*.
- ... a la implementación de un lenguaje: *Semántica operacional*.
- ... a las pruebas de equivalencia entre semánticas: *Semántica axiomática*.
- ... a la modelización con un lenguaje imperativo: *Semántica por teoría de modelos*.

Cuestión 2.23 Para la descripción de los aspectos semánticos de un lenguaje de programación con un enfoque operacional:

- Se define el significado de cada construcción del lenguaje como el objeto que dicha construcción denota.
- Se define una máquina abstracta y se expresa el significado de cada construcción del lenguaje en función de las diferentes acciones a realizar por la máquina para su ejecución.
- Se define el significado de cada construcción del lenguaje por medio de axiomas que establecen lo que se puede afirmar sobre el estado del programa tras su ejecución, en función de lo que era verdadero antes, o viceversa.
- Se define el significado de cada construcción del lenguaje como el punto fijo de una función continua y monótona definida para el programa.

Seleccione la respuesta correcta de entre las anteriores.

Cuestión 2.24 Explique con precisión que se entiende por lenguaje de programación declarativa. Indique sus principales características (entendido como sistema formal). Hable de su semántica operacional y declarativa y de la relación existente entre ambas.

Cuestión 2.25 Genéricamente, la programación declarativa puede entenderse como:

- Única y exclusivamente, programación con cláusulas de Horn definidas.
- Cualquier paradigma de programación en que los conceptos de computación en la máquina, deducción en una lógica apropiada y satisfacción en un modelo estándar del programa resultan equivalentes.
- Una metodología de programación con diagramas lógicos.
- Lo contrario de la programación natural.

Seleccione la respuesta correcta de entre las anteriores.

De la Demostración Automática a la Programación Lógica (I): introducción y métodos semánticos

La programación lógica surgió propiciada por las investigaciones realizadas en el área de la demostración automática de teoremas, especialmente, durante los primeros años de la década de 1960. En éste y en el próximo capítulo realizaremos un recorrido por algunos de los desarrollos del campo de la demostración automática de teoremas que sirven para dar luz sobre los fundamentos de la programación lógica. Este es, por lo tanto, un viaje que nos llevará desde la demostración automática a la Programación Lógica.

3.1 RAZONAMIENTO AUTOMÁTICO

El *razonamiento automático* es un campo de la informática, y más concretamente del área de la inteligencia artificial, que estudia cómo implantar en un computador programas que permitan verificar argumentos mediante el encadenamiento de pasos de inferencia, usando un sistema de leyes o reglas de inferencia no necesariamente basadas en la lógica clásica. En esta definición se hace énfasis en el razonamiento como proceso deductivo, ya que las técnicas deductivas son las mejor conocidas; así pues, otro término empleado es el de *deducción automática*. Cuando el trabajo se centra en la obtención de algoritmos que permitan encontrar pruebas de teoremas matemáticos (expresados como fórmulas lógicas) recibe el nombre de *demostración automática de teoremas*.

En la definición anterior se ha evitado identificar el razonamiento con cualquier proceso psicológico que, partiendo de unas proposiciones sobre las que tenemos la creencia

de que son verdaderas, nos permite llegar a una cierta conclusión, que también confiamos en que sea cierta en virtud del proceso desarrollado. Tampoco entraremos en la controversia de si mediante el razonamiento automático somos capaces de simular la inteligencia humana o no, aunque hay que destacar que este fue uno de los objetivos perseguidos desde sus orígenes por el razonamiento automático, en particular, y la inteligencia artificial, en general. Nosotros simplemente vamos a centrarnos en algunas técnicas desarrolladas en el área de la demostración automática de teoremas que son de especial interés para la programación lógica.

3.1.1. Aproximaciones a la demostración automática de teoremas

Han existido diversas aproximaciones a la demostración automática de teoremas que, simplificando la descripción, pueden clasificarse atendiendo a dos características esenciales:

1. La forma en la que se simula el razonamiento.

Desde el inicio han existido dos orientaciones al problema de simular el razonamiento. Históricamente, una primera orientación consistió en el desarrollo de técnicas que simulan la forma de razonamiento del ser humano. A esta orientación corresponden los trabajos pioneros de Gelernter dentro de la geometría plana o los de Newell, Shaw y Simon para la lógica de proposiciones, en los primeros años de la década de 1950. Estos demostradores de teoremas se basaban en métodos heurísticos e intentaban modelar la forma de raciocinio humana [104]. Este tipo de técnicas fueron abandonadas debido a su fracaso a la hora de afrontar problemas más complejos. Una primera razón del fracaso de esta orientación es que el cerebro humano y el modo en el que se lleva a cabo el razonamiento humano son de una estructura totalmente distinta al de los computadores. Un ejemplo de todo esto se puede observar en la regla de inferencia de instanciación. Si se pretende obtener sistemas que imiten el razonamiento humano, éstos deberían hacer uso intensivo de esta regla de inferencia (muy aplicada por las personas) que tiene un poder de deducción relativamente pequeño y conduce a la rápida aparición del problema de la explosión combinatoria.

Si bien estudiar el modo de razonamiento humano puede servir de ayuda en la confección de demostradores de teoremas, encontrando claves que permitan incrementar su eficiencia, la orientación dominante ha sido la de desarrollar técnicas que, aún alejadas del razonamiento humano, se adapten a su automatización en un computador. En esta dirección estarían los trabajos de Gilmore, Prawitz, Davis, y Putman (primeros años de la década de 1960), basados en el teorema de Herbrand, el de Wos, Robinson y Carson (1964), que implementan el primer demostrador automático basado en el *principio de resolución* de Robinson [127] (un método de prueba por contradicción para fórmulas en forma clausal debido a J.A. Robinson, también tributario de los trabajos de Herbrand) y la mayoría de los trabajos que

vinieron después. La resolución de Robinson puede verse como una regla de inferencia muy potente y bien adaptada a las características de los computadores pero relativamente alejada de las formas de razonamiento humano, ya que permite la obtención de conclusiones generales a partir de premisas generales, aplicando lo menos posible la regla de instanciación. Existen otras reglas de inferencia mucho más potentes, como *hiperresolución* [26], *modulación*, *paramodulación* [147], etc. que se han usado para mecanizar otro tipo de lógicas.

2. La cantidad de información intermedia retenida.

Una característica importante que distingue los demostradores automáticos es la cantidad de información intermedia que retienen durante el proceso de deducción. Son posibles tres alternativas: i) almacenar toda la información derivada intermedia, con el consiguiente crecimiento del *espacio de búsqueda*, i.e., del número de fórmulas (*estados*) a tratar; un ejemplo de este tipo de métodos sería la estrategia de resolución por saturación de niveles; ii) en el otro extremo se sitúan los que no almacenan o minimizan la información intermedia retenida, con la consiguiente pérdida de potencia deductiva, ya que puede ser necesario establecer nuevamente resultados que fueron obtenidos con anterioridad y son indispensables para proseguir la deducción; un ejemplo de éstos serán los demostradores basados en resolución lineal, o en alguna variación como la resolución SLD, que intentan minimizar la cantidad de información derivada retenida; la idea que se persigue con este tipo de demostradores es beneficiarse del menor coste espacial y de los mecanismos de ejecución eficientes ya mencionados, pero la imposibilidad de tratar convenientemente teorías con igualdad, la dificultad para introducir estrategias, y la ausencia de retención de información intermedia (que conduce al problema ya apuntado más arriba) hacen que este tipo de demostradores sean poco adecuados para resolver problemas de complejidad mediana y grande [146]; iii) el punto medio consistiría en retener solo información que cumpliese algunos requisitos; un ejemplo de esto sería la estrategia de resolución semántica o los métodos desarrollados por Wos y sus colaboradores para el tratamiento de la información intermedia.

Una buena parte del trabajo desarrollado sobre demostración automática en lógica clásica se basa en el principio de resolución. Otros métodos empleados han sido los basados en la deducción natural de Gentzen, los grafos de conexión [81] y las tablas semánticas (*Semantic Tableaux*)¹. Como señala M. Fitting en [55], los demostradores basados en tablas semánticas han sido, comparativamente, raros en el área. Nosotros vamos a introducir algunas de las técnicas desarrolladas para los demostradores basados en resolución, por ser las que originaron la programación lógica.

¹Este método fue desarrollado por R.M. Smullyan en la década de 1960 y también es tributario de los trabajos de Gentzen.

3.1.2. Un demostrador automático genérico

Un demostrador automático de teoremas puede verse como un programa que imita, en líneas muy generales, a los matemáticos en la forma de abordar un problema. Como los matemáticos que resuelven un difícil teorema, el demostrador tiene que llegar previamente a resultados intermedios (lemas) que, una vez almacenados como información auxiliar, constituirán los puntos de partida de una deducción lógica. Así pues es imprescindible retener la información intermedia (significativa). Sin embargo, contrariamente a lo que hacen los matemáticos, los demostradores hacen uso de una representación especial de las fórmulas denominada *forma clausal*. Con el fin de eliminar información innecesaria o redundante se representan las fórmulas intermedias en una forma canónica (*demodulation*) y después se someten a un proceso de simplificación conocido como subsumpción (*subsumtion*), que elimina expresiones que se pueden considerar redundantes al haber otra expresión más general que las incluye. En la deducción se emplean reglas de inferencia alejadas del razonamiento humano, como resolución y *paramodulación*, una regla para el tratamiento de la igualdad. Finalmente, en la mayoría de los casos, los demostradores de teoremas realizan *pruebas por contradicción*, también denominadas *pruebas por refutación* (ver más adelante el Apartado 3.2). La decisión de retener información requiere que el demostrador realice funciones que de otro modo serían innecesarias: procedimientos para la eliminación de conclusiones que están contenidas de alguna forma en otras, así como procedimientos para restringir y dirigir la búsqueda. Todo esto tiene el beneficio de un mayor poder deductivo, que es lo que permite alcanzar el éxito, al precio de un mayor coste temporal que es preciso equilibrar.

Dado que, normalmente, un demostrador realiza las pruebas por refutación y utiliza fórmulas en forma clausal, para demostrar un teorema de la forma “si \mathcal{A} entonces \mathcal{B} ”, debemos introducir la información necesaria de manera adecuada: hay que expresar en forma clausal todos los conocimientos relativos al campo del que se extrae el teorema y que formarán la teoría, esto es, las hipótesis de partida; después agregar la condición \mathcal{A} a las hipótesis (*hipótesis especial*) y, finalmente, negar la conclusión \mathcal{B} . Una vez suministrada esta información, las acciones típicas que lleva a cabo un demostrador de teoremas, son:

1. Aplicación de las reglas de inferencia (e.g., resolución y paramodulación) para obtener conclusiones. La aplicación de las reglas de inferencia es restringida por un tipo de estrategia y dirigida por otro tipo de estrategia, ambas tendentes a limitar el espacio de búsqueda.
2. Cuando se obtiene una conclusión, el programa la reescribe a una forma canónica aplicando reglas (*demoduladores*) suministradas por el experto o por el programa. El resultado de este proceso se analiza (e.g., empleando subsumpción) para ver si es un corolario trivial de información que el programa ya posee y que, por consiguiente, debe desecharse. El demostrador también puede comprobar si el resultado satisface algún criterio heurístico (e.g., *pesos*) suministrado por el investigador

para medir la relevancia del resultado de cara a facilitar una prueba efectiva; si la conclusión no supera el test, se descarta.

3. Si la conclusión obtenida entra en contradicción con alguna de las informaciones mantenidas por el demostrador, entonces la prueba concluye con éxito; si no, se repite el proceso.

3.1.3. Límites de la demostración automática de teoremas

Cuando nos ceñimos a la lógica de predicados, los límites del razonamiento automático están claros: como ya dijimos en el Apartado 2.2.4, la lógica de predicados es indecidible; es decir, no existe un procedimiento general de decisión que nos permita determinar si una fórmula \mathcal{A} es, o no, un teorema (o equivalentemente, lógicamente válida) de \mathcal{L} . De forma más general, no puede contestarse a la pregunta de si $\Gamma \vdash \mathcal{A}$ (o equivalentemente, $\Gamma \models \mathcal{A}$), donde Γ es un conjunto de fórmulas de \mathcal{L} . Por consiguiente, podemos afirmar que el objetivo de la demostración automática de teoremas es irrealizable en términos generales. Sin embargo, si la fórmula realmente es válida, hay procedimientos de prueba que pueden verificarlo. Estos procedimientos de prueba, en general, no siempre terminan si la fórmula no es válida (son procedimientos de semidecisión). En próximos apartados estudiaremos algunos de estos algoritmos. En este capítulo nos centraremos en los basados en métodos semánticos sustentados por el teorema de Herband. En el Capítulo 4 estudiaremos los procedimientos basados en métodos sintácticos, sustentados por en el principio de resolución de Robinson. Ambos métodos hacen posible la demostración automática de teoremas cuando restringimos la generalidad de nuestro dominio de aplicación.

Otro límite de la demostración automática de teoremas viene impuesto por la complejidad computacional de los problemas objeto de estudio y los algoritmos utilizados en su solución. Por un lado, recordemos el crecimiento exponencial del espacio de búsqueda, debido al problema de la explosión combinatoria, que obliga a una necesidad creciente de memoria para el almacenamiento de estructuras intermedias utilizadas en los cómputos, que sobrepasa con creces las posibilidades de las máquinas actuales. Por otro, el alto coste temporal de los algoritmos utilizados en el proceso de la demostración: unificación, subsumpción, podas en el espacio de búsqueda, etc.

A pesar de las limitaciones señaladas, la demostración automática de teoremas ha alcanzado un grado de madurez considerable en los últimos veinte años que le ha llevado a superar con éxito muchos de sus objetivos iniciales. En [146], L. Wos da una relación de más de una decena de problemas abiertos en el ámbito de las matemáticas, de considerable dificultad y resueltos por métodos automáticos. En 1997, W. McCune anunció la solución del problema de Robbins² haciendo uso del demostrador automático

²El problema de Robbins cuestiona si toda álgebra asociativa y conmutativa que satisface el llamado axioma de Robbins es también booleana. En efecto, EQP respondió afirmativamente a la cuestión de que toda álgebra de Robbins es booleana, para lo que empleó 8 días de búsqueda usando paramodulación y una estrategia de pesos (de peso máximo 70).

EQP, especializado en problemas expresables en el formalismo de la lógica ecuacional [95]. El problema de Robbins había intrigado a los matemáticos desde hacía más de 60 años.

Del análisis de los párrafos precedentes concluimos que, en el estudio de la demostración automática de teoremas, los aspectos esenciales son:

1. el estudio de los procedimientos de prueba por refutación;
2. la definición de alguna forma de representación compacta para las fórmulas;
3. la elección de reglas de inferencia adecuadas; y
4. las estrategias para dirigir la búsqueda.

En los próximos apartados y en el Capítulo 4 profundizaremos en cada uno de estos puntos.

3.2 PROCEDIMIENTO DE PRUEBA POR REFUTACIÓN

Los procedimientos de prueba basados en el teorema de Herbrand, que discutiremos en los próximos apartados, y en el principio de resolución de Robinson, que discutiremos en el Capítulo 4, son procedimientos de *prueba por refutación* (también denominada prueba por *deducción indirecta* o *reducción al absurdo*). Este tipo de procedimiento consiste en lo siguiente: i) suponemos la falsedad de la conclusión (negamos lo que queremos probar); ii) a partir de esta suposición tratamos de obtener una contradicción; iii) si llegamos a una contradicción, entonces se rechaza el supuesto en vista del resultado y iv) como consecuencia, se afirma la conclusión deseada. La siguiente discusión justifica este tipo de procedimiento de prueba y aclara algunos puntos que serán de interés en el futuro.

En el contexto de la lógica de predicados en el que nos movemos, un procedimiento de prueba intenta establecer relaciones de consecuencia lógica entre fórmulas. Esto es, trata de contestar a la pregunta

$\Gamma \models \mathcal{A}?$, donde \mathcal{A} es una fbf de \mathcal{L} y Γ un conjunto de fbf de \mathcal{L} .

en palabras, si la fbf \mathcal{A} es consecuencia lógica del conjunto Γ de fbf de \mathcal{L} (que suponemos como premisas). La corrección y completitud de la lógica de predicados nos permite expresar esta pregunta, que se ha formulado en los términos semánticos de consecuencia lógica, en los términos sintácticos, más habituales, de deducibilidad.

$$\begin{aligned}\Gamma \models \mathcal{A} &\Leftrightarrow \Gamma \cup \{\neg\mathcal{A}\} \text{ es insatisfacible} && \text{Proposición 2.3} \\ &\Leftrightarrow \Gamma \cup \{\neg\mathcal{A}\} \text{ es inconsistente} && \text{Corolario de la Prop. 2.5} \\ &\Leftrightarrow \Gamma \cup \{\neg\mathcal{A}\} \vdash \square && \text{Proposición 2.4}\end{aligned}$$

donde el símbolo \square se emplea comúnmente en programación lógica para representar una fórmula inconsistente $\mathcal{B} \wedge \neg\mathcal{B}$. Este razonamiento nos indica que si del conjunto $\Gamma \cup \{\neg\mathcal{A}\}$ se puede derivar una contradicción, entonces podemos contestar afirmativamente a nuestra pregunta inicial.

Por otro lado, suponiendo que $\Gamma = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$, la Proposición 2.3 nos permite transformar la pregunta original en la pregunta

¿La fórmula $(\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_n \wedge \neg\mathcal{A})$ es insatisfacible?
donde $\mathcal{A}_1, \dots, \mathcal{A}_n$ y \mathcal{A} son fbf de \mathcal{L} .

Así pues, hemos reducido el problema de establecer la relación de consecuencia lógica entre el conjunto de fbf Γ y la fórmula \mathcal{A} en un problema de probar la insatisfacibilidad de una fórmula. De forma equivalente, mediante un razonamiento similar al empleado más arriba, podemos transformar la pregunta original en un nuevo problema sintáctico relativo a una fórmula: probar la inconsistencia de la fbf $(\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_n \wedge \neg\mathcal{A})$.

Al transformar nuestro problema original en el de comprobar si una fórmula es insatisfacible (o alternativamente, inconsistente), cobra sentido considerar formas más simples pero equivalentes de expresar una fórmula que la hagan más fácil de manipular, con el objetivo de optimizar el procedimiento de prueba. En el próximo apartado estudiaremos las *formas estándares* en la lógica.

3.3 FORMAS NORMALES

Esencialmente, el proceso de obtener una *forma normal* a partir de una fbf dada consiste en sustituir partes de dicha fórmula por fórmulas lógicamente equivalentes. El objetivo es obtener una fórmula que emplea un número restringido de conectivas y/o cuantificadores y tiene el mismo significado que la fórmula original. En el contexto de la lógica de proposiciones, dos fórmulas son lógicamente equivalentes si tienen la misma tabla de verdad. Podemos generalizar el concepto de fórmula lógicamente equivalente de la lógica de proposiciones (adaptandolo a la lógica de predicados) mediante la siguiente definición.

Definición 3.1 (Fórmulas lógicamente equivalentes) Dos fbf cerradas \mathcal{A} y \mathcal{B} son lógicamente equivalentes, denotado $\mathcal{A} \leftrightarrow \mathcal{B}$, si y solo si $\{\mathcal{A}\} \models \mathcal{B}$ y $\{\mathcal{B}\} \models \mathcal{A}$.

Intuitivamente, dos fbf son lógicamente equivalentes si, una vez interpretadas y valoradas, toman el mismo valor de verdad bajo cualquier circunstancia. Esto es, son fórmulas idénticas desde una perspectiva semántica. Nótese también que, por la corrección y completitud de la lógica de predicados, si dos fbf \mathcal{A} y \mathcal{B} son lógicamente equivalentes se cumple que $\{\mathcal{A}\} \vdash \mathcal{B}$ y $\{\mathcal{B}\} \vdash \mathcal{A}$. Por consiguiente, $\vdash (\mathcal{A} \leftrightarrow \mathcal{B})$, esto es, en la terminología de Hamilton [65], las fbf \mathcal{A} y \mathcal{B} son *demonstrablemente equivalentes*. El concepto

Tabla 3.1 Fórmulas equivalentes de la lógica de proposiciones.

1	$(\mathcal{A} \leftrightarrow \mathcal{B}) \Leftrightarrow (\mathcal{A} \rightarrow \mathcal{B}) \wedge (\mathcal{B} \rightarrow \mathcal{A})$	
2	$(\mathcal{A} \rightarrow \mathcal{B}) \Leftrightarrow (\neg \mathcal{A} \vee \mathcal{B})$	
3	(a) $(\mathcal{A} \vee \mathcal{B}) \Leftrightarrow (\mathcal{B} \vee \mathcal{A})$	(b) $(\mathcal{A} \wedge \mathcal{B}) \Leftrightarrow (\mathcal{B} \wedge \mathcal{A})$
4	(a) $(\mathcal{A} \vee \mathcal{B}) \vee \mathcal{C} \Leftrightarrow \mathcal{A} \vee (\mathcal{B} \vee \mathcal{C})$	(b) $(\mathcal{A} \wedge \mathcal{B}) \wedge \mathcal{C} \Leftrightarrow \mathcal{A} \wedge (\mathcal{B} \wedge \mathcal{C})$
5	(a) $\mathcal{A} \vee (\mathcal{B} \wedge \mathcal{C}) \Leftrightarrow (\mathcal{A} \vee \mathcal{B}) \wedge (\mathcal{A} \vee \mathcal{C})$	(b) $\mathcal{A} \wedge (\mathcal{B} \vee \mathcal{C}) \Leftrightarrow (\mathcal{A} \wedge \mathcal{B}) \vee (\mathcal{A} \wedge \mathcal{C})$
6	(a) $(\mathcal{A} \vee \square) \Leftrightarrow \mathcal{A}$	(b) $(\mathcal{A} \wedge \diamond) \Leftrightarrow \mathcal{A}$
7	(a) $(\mathcal{A} \vee \diamond) \Leftrightarrow \diamond$	(b) $(\mathcal{A} \wedge \square) \Leftrightarrow \square$
8	(a) $(\mathcal{A} \vee \neg \mathcal{A}) \Leftrightarrow \diamond$	(b) $(\mathcal{A} \wedge \neg \mathcal{A}) \Leftrightarrow \square$
9	$\neg(\neg \mathcal{A}) \Leftrightarrow \mathcal{A}$	
10	(a) $\neg(\mathcal{A} \vee \mathcal{B}) \Leftrightarrow (\neg \mathcal{A} \wedge \neg \mathcal{B})$	(b) $\neg(\mathcal{A} \wedge \mathcal{B}) \Leftrightarrow (\neg \mathcal{A} \vee \neg \mathcal{B})$
11	(a) $(\mathcal{A} \vee \mathcal{A}) \Leftrightarrow \mathcal{A}$	(b) $(\mathcal{A} \wedge \mathcal{A}) \Leftrightarrow \mathcal{A}$

de demostrablemente equivalente es sintáctico y, unido al teorema de intercambio (Teorema 2.2), justifica la sustitución de partes de fórmulas por fórmulas equivalentes³.

La Tabla 3.1 presenta una relación de fórmulas equivalentes de la lógica de proposiciones y la Tabla 3.2 completa las equivalencias que cumplen para la lógica de predicados. Estas equivalencias se utilizarán posteriormente en la manipulación de las fórmulas.

Observaciones 3.1

1. Al emplear las metavariables \mathcal{A} , \mathcal{B} y \mathcal{C} , que representan fórmulas cualesquiera, las relaciones de equivalencia de las Tablas 3.1 y 3.2 se establecen, realmente, entre esquemas de fórmulas y no entre fórmulas.
2. Como ya se ha mencionado, el metasímbolo \square representa una fórmula insatisfacible (una *contradicción* en el caso concreto de la lógica de proposiciones, i.e. una fórmula que siempre se evalúa al valor de verdad falso). Por otra parte, el metasímbolo \diamond representa una fórmula lógicamente válida (una *tautología* en el caso concreto de la lógica de proposiciones, i.e. una fórmula que siempre se evalúa al valor de verdad verdadero).
3. Las equivalencias de la Tabla 3.1, por cumplirse esencialmente entre fórmulas de la lógica de proposiciones, pueden establecerse haciendo uso del método semántico de las tablas de verdad.
4. Las equivalencias de la Tabla 3.1 son bien conocidas en la lógica clásica, por ejemplo: (9) es la ley de la doble negación; (10.a) y (10.b) son las leyes de De

³En el futuro, hablaremos simplemente de fórmulas equivalentes como sinónimo de fórmulas lógicamente equivalentes o demostrablemente equivalentes.

Morgan; y (11.a) y (11.b) son las leyes de la idempotencia de las conectivas “ \vee ” y “ \wedge ”, respectivamente. Para más información en este sentido ver [75].



La forma clausal es una forma normal que juega un papel especial en los procedimientos de prueba basados en el teorema de Herbrand y el principio de resolución. Antes de definir qué entendemos por forma clausal debemos repasar otras formas estándares de la lógica.

3.3.1. Formas normales en la lógica proposicional

La idea detrás de las formas normales en la lógica proposicional es el deseo de representar cualquier fórmula mediante conjuntos adecuados de conectivas. Como dijimos en el Capítulo 2, puede demostrarse que cualquier fórmula que no sea una contradicción puede expresarse como una fórmula equivalente, compuesta únicamente por las conectivas “ \neg ”, “ \vee ” y “ \wedge ”. Es en este sentido en el que decimos que el conjunto $\{\neg, \vee, \wedge\}$ es un *conjunto adecuado* de conectivas. Restringiéndonos al anterior conjunto de conectivas, es posible la obtención de dos clases de formas normales, las llamadas *forma normal disyuntiva* y *forma normal conjuntiva*.

Definición 3.2 (Forma normal disyuntiva) Una fbf \mathcal{A} está en forma normal disyuntiva si y solo si $\mathcal{A} \equiv \mathcal{A}_1 \vee \mathcal{A}_2 \vee \dots \vee \mathcal{A}_n$, con $n \geq 1$, y cada \mathcal{A}_i es una conjunción de literales.

Ejemplo 3.1

La fbf $(q \wedge p) \vee (\neg r \wedge s) \vee \neg q$ es una forma normal disyuntiva.

Definición 3.3 (Forma normal conjuntiva) Una fbf \mathcal{A} está en forma normal conjuntiva si y solo si $\mathcal{A} \equiv \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \dots \wedge \mathcal{A}_n$, con $n \geq 1$, y cada \mathcal{A}_i es una disyunción de literales.

Ejemplo 3.2

La fbf $(q \vee p) \wedge (\neg r \vee s) \wedge \neg q$ es una forma normal conjuntiva. Por razones que pronto quedarán claras, la más interesante de las dos es la forma normal conjuntiva.

A continuación damos un algoritmo que nos permite obtener una forma normal conjuntiva o disyuntiva.

Algoritmo 3.1

Entrada: una fbf \mathcal{A} de \mathcal{L} .

Salida: una fórmula en forma normal conjuntiva (o disyuntiva) \mathcal{B} .

Comienzo

PASO 1. Usar las reglas (1) y (2) de la Tabla 3.1 para eliminar las conectivas “ \leftrightarrow ” y “ \rightarrow ” de la fbf \mathcal{A} .

PASO 2. Emplear repetidamente las reglas (9), (10.a) y (10.b) de la Tabla 3.1 para llevar la negación a los átomos.

PASO 3. Emplear repetidamente las reglas distributivas (5.a) y (5.b) y el resto de las reglas de la Tabla 3.1 hasta obtener una forma normal conjuntiva (o disyuntiva) \mathcal{B} .

Devolver \mathcal{B} .

Fin

Ejemplo 3.3

Obtener una forma normal conjuntiva de la fórmula $\mathcal{A} \equiv (p \vee \neg q) \rightarrow r$.

$$\begin{aligned}\mathcal{A} &\Leftrightarrow \neg(p \vee \neg q) \vee r & P1(2) \\ &\Leftrightarrow (\neg p \wedge \neg(\neg q)) \vee r & P2(10.a) \\ &\Leftrightarrow (\neg p \wedge q) \vee r & P2(9) \\ &\Leftrightarrow r \vee (\neg p \wedge q) & P3(3.a) \\ &\Leftrightarrow (r \vee \neg p) \wedge (r \vee q) & P3(5.a)\end{aligned}$$

Nótese que la fórmula $(\neg p \wedge q) \vee r$, obtenida en el tercer paso, es una forma normal disyuntiva de \mathcal{A} .

3.3.2. Formas normales en la lógica de predicados

Como hemos dicho, el objetivo de introducir formas normales es simplificar los procedimientos de prueba. En la lógica de predicados, cualquier fbf puede transformarse en una fórmula equivalente denominada *forma normal prenexa*. Lo que caracteriza a una forma normal prenexa es la disposición de los cuantificadores, que deben encontrarse todos al principio de la fórmula.

Definición 3.4 (Forma normal prenexa) Una fbf \mathcal{A} está en forma normal prenexa si y solo si $\mathcal{A} \equiv (Q_1 X_1) \dots (Q_n X_n) M$, donde X_1, \dots, X_n son variables diferentes, para cada $1 \leq i \leq n$, $Q_i \equiv \forall$ o bien $Q_i \equiv \exists$ y M es una fórmula que no contiene cuantificadores. Al componente $(Q_1 X_1) \dots (Q_n X_n)$ se le llama prefijo y a M se le denomina matriz de la fórmula \mathcal{A} .

Nótese que una fbf de \mathcal{L} sin cuantificadores se considera un caso trivial de fbf en forma normal prenexa. Una fórmula cerrada en forma prenexa $(\forall X_1) \dots (\forall X_n) M$ se denomi-

Tabla 3.2 Fórmulas equivalentes de la lógica de predicados.

12.a	$(QX)\mathcal{A}(X) \vee B \Leftrightarrow (QX)(\mathcal{A}(X) \vee B)$
12.b	$(QX)\mathcal{A}(X) \wedge B \Leftrightarrow (QX)(\mathcal{A}(X) \wedge B)$
13	(a) $\neg(\forall X)\mathcal{A}(X) \Leftrightarrow (\exists X)\neg\mathcal{A}(X)$ (b) $\neg(\exists X)\mathcal{A}(X) \Leftrightarrow (\forall X)\neg\mathcal{A}(X)$
14.a	$(\forall X)\mathcal{A}(X) \wedge (\forall X)C(X) \Leftrightarrow (\forall X)(\mathcal{A}(X) \wedge C(X))$
14.b	$(\exists X)\mathcal{A}(X) \vee (\exists X)C(X) \Leftrightarrow (\exists X)(\mathcal{A}(X) \vee C(X))$
15.a	$(Q_1 X)\mathcal{A}(X) \vee (Q_2 X)C(X) \Leftrightarrow (Q_1 X)(Q_2 Z)(\mathcal{A}(X) \vee C(Z))$
15.b	$(Q_3 X)\mathcal{A}(X) \wedge (Q_4 X)C(X) \Leftrightarrow (Q_3 X)(Q_4 Z)(\mathcal{A}(X) \wedge C(Z))$
	$(Q \equiv \forall \text{ o bien } Q \equiv \exists)$

na *fórmula universal*. Por otro lado tenemos una fórmula cerrada en la forma prenexa $(\exists X_1) \dots (\exists X_n)M$ que se denomina *fórmula existencial*.

Para transformar una fórmula a forma normal prenexa emplearemos las equivalencias de las Tablas 3.1 y 3.2 y el Algoritmo 2.

Observaciones 3.2

1. En la Tabla 3.2, la variable Z que se muestra en las equivalencias (15) es una variable que no aparece en la fórmula $\mathcal{A}(X)$.
2. En la Tabla 3.2, nuevamente en las equivalencias (15), si $Q_1 \equiv Q_2 \equiv \exists$ y $Q_3 \equiv Q_4 \equiv \forall$ entonces, no es necesario renombrar las apariciones de la variable X en $(Q_2 X)C(X)$ o en $(Q_4 X)C(X)$. En este caso se puede usar directamente las equivalencias 14.



Algoritmo 3.2

Entrada: una fbf \mathcal{A} de \mathcal{L} .

Salida: una fórmula en forma normal prenexa B .

Comienzo

- PASO 1. Usar las reglas (1) y (2) de la Tabla 3.1 para eliminar las conectivas “ \leftrightarrow ” y “ \rightarrow ” de la fbf \mathcal{A} .
- PASO 2. Emplear repetidamente las reglas (9), (10.a) y (10.b) de la Tabla 3.1 y la regla (13) de la Tabla 3.2 para llevar la negación a los átomos.
- PASO 3. Renombrar las variables ligadas, si es necesario.
- PASO 4. Emplear las reglas (12), (14) y (15) de la Tabla 3.2 para mover los cuantificadores a la izquierda y obtener una forma normal prenexa B .

Devolver B .

Fin

Ejemplo 3.4

Transformar la fórmula $\mathcal{A} \equiv (\forall X)p(X) \rightarrow (\exists X)q(X)$ a una forma normal prenexa.

$$\begin{aligned}\mathcal{A} &\Leftrightarrow \neg(\forall X)p(X) \vee (\exists X)q(X) & P1(2) \\ &\Leftrightarrow (\exists X)\neg p(X) \vee (\exists X)q(X) & P2(13.a) \\ &\Leftrightarrow (\exists X)(\neg p(X) \vee q(X)) & P4(14.b)\end{aligned}$$

Ejemplo 3.5

Transformar la fórmula $\mathcal{A} \equiv (\forall X)r(X) \rightarrow (\forall Y)s(X, Y)$ a una forma normal prenexa.

$$\begin{aligned}\mathcal{A} &\Leftrightarrow \neg(\forall X)r(X) \vee (\forall Y)s(X, Y) & P1(2) \\ &\Leftrightarrow (\exists X)\neg r(X) \vee (\forall Y)s(X, Y) & P2(13.a) \\ &\Leftrightarrow (\exists Z)\neg r(Z) \vee (\forall Y)s(X, Y) & P3 \\ &\Leftrightarrow (\exists Z)(\neg r(Z) \vee (\forall Y)s(X, Y)) & P2(12.a) \\ &\Leftrightarrow (\exists Z)(\forall Y)(\neg r(Z) \vee s(X, Y)) & P2(12.a)\end{aligned}$$

Observe que el paso *P3* es necesario para evitar que se introduzcan ligaduras que no aparecían en la fórmula original.

3.4 LA FORMA CLAUSAL

En la actualidad, la mayoría de los procedimientos de prueba operan por refutación, es decir, en lugar de probar que una fórmula \mathcal{A} es lógicamente válida, se prueba que la negación de dicha fórmula, $\neg\mathcal{A}$, es insatisfacible. Estas pruebas por refutación se aplican a fórmulas en una forma normal denominada *forma clausal*. Esta clase de forma normal fue introducida por Davis y Putnam (1960) teniendo en cuenta las siguientes ideas:

1. Una fórmula (o la negación de una fórmula) \mathcal{A} , de un lenguaje de primer orden, puede transformarse en una forma normal prenexa, por lo que toma la forma $(Q_1X_1) \dots (Q_nX_n)\mathcal{M}$, donde \mathcal{M} no contiene cuantificadores y el prefijo es una secuencia de cuantificadores universales (y/o) existenciales.
2. Dado que \mathcal{M} no contiene cuantificadores de ningún tipo, puede transformarse en una forma normal conjuntiva.
3. Los cuantificadores existenciales pueden eliminarse del prefijo utilizando funciones de Skolem, sin afectar a la inconsistencia o insatisfacibilidad de una fórmula.

3.4.1. Forma normal de Skolem

Sea \mathcal{A} una fórmula en forma prenexa, en la que la matriz se ha expresado en forma normal conjuntiva. Una *forma normal de Skolem* es una fórmula obtenida a partir de \mathcal{A}

reemplazando por *funciones de Skolem* las variables ligadas por los cuantificadores existenciales y eliminando después los cuantificadores existenciales. Siguiendo las pautas del Algoritmo 3 obtenemos una fórmula, \mathcal{B} , expresada en forma normal de Skolem, en la que ahora todos los cuantificadores son universales:

$$(\forall X_{j_1}) \dots (\forall X_{j_n}) \mathcal{M}(X_{j_1}, \dots, X_{j_n})$$

donde $\{X_{j_1}, \dots, X_{j_n}\} \subseteq \{X_1, \dots, X_n\}$. Las acciones del *Paso 3* del Algoritmo 3 reciben el nombre de *skolemización*. Las constantes c y las funciones f , empleadas en la sustitución de variables existenciales se denominan *constantes de Skolem* y *funciones de Skolem*, respectivamente. La razón intuitiva que justifica la necesidad de introducir funciones de Skolem queda clara en el siguiente ejemplo.

Ejemplo 3.6

La sentencia “todo el mundo tiene madre” puede formalizarse en la lógica de predicados mediante la fórmula: $(\forall X)(\exists Y)\text{esMadre}(Y, X)$, si suponemos que el universo de discurso es el conjunto de los seres humanos⁴. Al pasar a forma normal de Skolem la fórmula anterior, eliminando el cuantificador existencial, podríamos adoptar la posición ingenua de sustituir la variable ligada Y mediante una constante, por ejemplo, “eva”, que representase a un cierto individuo que cumpliese la propiedad mencionada. Entonces, obtendríamos: $(\forall X)\text{esMadre}(\text{eva}, X)$. Esta es una formalización que no se corresponde con el sentido de la sentencia original. Más bien lo que expresa es que existe un individuo que es la madre de todos los humanos (cuya formalización es $(\exists Y)(\forall X)\text{esMadre}(Y, X)$). Por lo tanto, si lo que se desea es eliminar el cuantificador existencial pero conservando el sentido del enunciado original, lo que necesitamos es introducir una función, “madre” que, a cada humano X , asocie su propia madre $\text{madre}(X)$. De este modo, la forma de Skolem sería:

$$(\forall X)\text{esMadre}(\text{madre}(X), X),$$

que conserva el sentido de la sentencia original.

Algoritmo 3.3

Entrada: una fbf \mathcal{A} de \mathcal{L} .

Salida: una fórmula en forma normal de Skolem \mathcal{B} .

Comienzo

- PASO 1. Usar el Algoritmo 2 para obtener una forma normal prenexa $(Q_1 X_1) \dots (Q_n X_n) \mathcal{M}$ a partir de la fbf \mathcal{A} .
- PASO 2. Usar el Algoritmo 1 para obtener la forma normal conjuntiva de \mathcal{M} .

⁴Hemos realizado este supuesto para facilitar la exposición sin enturbiarla con detalles innecesarios. Si no hacemos este supuesto, se necesita una formulación más compleja: $(\forall X)[\text{esHumano}(X) \rightarrow (\exists Y)(\text{esHumano}(Y) \wedge \text{esMadre}(Y, X))]$.

PASO 3. Eliminar los cuantificadores existenciales para obtener una forma normal de Skolem \mathcal{B} , aplicando reiteradamente los siguientes pasos: Supongamos que $Q_r \equiv \exists$ en el prefijo $(Q_1 X_1) \dots (Q_n X_n)$, con $1 \leq r \leq n$.

1. Si a Q_r no le precede ningún cuantificador universal, sustituir cada aparición de X_r por una constante c , que sea diferente de cualquier otra constante que aparezca en \mathcal{M} . Borrar $(Q_r X_r)$ del prefijo.
2. Si $Q_{s_1} \dots Q_{s_m}$ son todos los cuantificadores universales que preceden a Q_r , $1 \leq s_1 < s_2 < \dots < s_m < r$, elegir un nuevo símbolo de función m -ario, f , que no aparezca en \mathcal{M} y reemplazar todas las apariciones de X_r en \mathcal{M} por $f(X_{s_1}, X_{s_2}, \dots, X_{s_m})$. Borrar $(Q_r X_r)$ del prefijo.

Devolver \mathcal{B} .

Fin

Ilustramos la mecánica del Algoritmo 3 mediante algunos ejemplos:

Ejemplo 3.7

Sea la fórmula

$$\mathcal{A} \equiv (\exists X_1)(\forall X_2)(\forall X_3)(\exists X_4)(\forall X_5)(\exists X_6)p(X_1, X_2, X_3, X_4, X_5, X_6)$$

Dado que la fórmula se encuentra en forma prenexa y, trivialmente, su matriz está en forma normal conjuntiva, pasamos directamente al proceso de skolemización:

$$\begin{aligned} \mathcal{A} &\Leftrightarrow (\forall X_2)(\forall X_3)(\exists X_4)(\forall X_5)(\exists X_6)p(a, X_2, X_3, X_4, X_5, X_6) && P3(1) \\ &\Leftrightarrow (\forall X_2)(\forall X_3)(\forall X_5)(\exists X_6)p(a, X_2, X_3, f(X_2, X_3), X_5, X_6) && P3(2) \\ &\Leftrightarrow (\forall X_2)(\forall X_3)(\forall X_5)p(a, X_2, X_3, f(X_2, X_3), X_5, g(X_2, X_3, X_5)) && P3(2) \end{aligned}$$

Así pues, una forma normal de Skolem de \mathcal{A} sería:

$$(\forall X_2)(\forall X_3)(\forall X_5)p(a, X_2, X_3, f(X_2, X_3), X_5, g(X_2, X_3, X_5)).$$

Ejemplo 3.8

Dada la fórmula

$$\mathcal{A} \equiv (\forall X_1)(\exists X_2)(\exists X_3)[(\neg p(X_1) \wedge q(X_2)) \vee r(X_2, X_3)]$$

una forma normal de Skolem de \mathcal{A} se obtendría mediante los siguientes pasos:

$$\begin{aligned} \mathcal{A} &\Leftrightarrow (\forall X_1)(\exists X_2)(\exists X_3)[[(\neg p(X_1) \vee r(X_2, X_3)) \wedge (q(X_2) \vee r(X_2, X_3))] && P2(5.a) \\ &\Leftrightarrow (\forall X_1)(\exists X_3)[[(\neg p(X_1) \vee r(f(X_1), X_3)) \wedge (q(f(X_1)) \vee r(f(X_1), X_3))] && P3(2) \\ &\Leftrightarrow (\forall X_1)[[(\neg p(X_1) \vee r(f(X_1), g(X_1))) \wedge (q(f(X_1)) \vee r(f(X_1), g(X_1)))] && P3(2) \end{aligned}$$

A partir de ahora emplearemos simplemente el término “*forma normal*” o “*forma estándar*” para referirnos a la forma normal de Skolem.

3.4.2. Cláusulas

Definición 3.5 (Cláusula) Una cláusula es una disyunción finita de cero o más literales. Cuando la cláusula está compuesta de un solo literal diremos que es una cláusula unitaria.

Ejemplo 3.9

En el Ejemplo 3.8, las disyunciones

$$\neg p(X_1) \vee r(f(X_1), g(X_1)) \quad \text{y} \quad q(f(X_1)) \vee r(f(X_1), g(X_1))$$

son cláusulas.

Una fórmula en forma estándar, $\mathcal{A} \equiv (\forall X_1) \dots (\forall X_n) \mathcal{M}$, puede representarse como un conjunto de cláusulas, si se tiene en cuenta lo siguiente. El cuantificador universal cumple la propiedad distributiva respecto a la conectiva “ \wedge ”, de forma que podemos escribir la forma estándar, \mathcal{A} , como:

$$\begin{aligned} &(\forall X_1) \dots (\forall X_n) (\mathcal{M}_{11} \vee \mathcal{M}_{12} \vee \dots) \wedge \\ &(\forall X_1) \dots (\forall X_n) (\mathcal{M}_{21} \vee \mathcal{M}_{22} \vee \dots) \wedge \\ &\vdots \\ &(\forall X_1) \dots (\forall X_n) (\mathcal{M}_{(n-1)1} \vee \mathcal{M}_{(n-1)2} \vee \dots) \wedge \\ &(\forall X_1) \dots (\forall X_n) (\mathcal{M}_{n1} \vee \mathcal{M}_{n2} \vee \dots) \end{aligned}$$

La Definición 3.5 y el hecho anterior justifican que la fórmula \mathcal{A} pueda representarse, de una manera más simple, mediante un conjunto de cláusulas Δ .

$$\Delta = \{(\mathcal{M}_{11} \vee \mathcal{M}_{12} \vee \dots), (\mathcal{M}_{21} \vee \mathcal{M}_{22} \vee \dots), \dots, (\mathcal{M}_{n1} \vee \mathcal{M}_{n2} \vee \dots)\}$$

Una contradicción, $(L \wedge \neg L)$, se representa mediante el conjunto vacío, $\{\}$, de ahí que también se le denomine *cláusula vacía*. Ya hemos mencionado que una fórmula contradictoria también se representa haciendo uso del símbolo \square .

Ejemplo 3.10

Con esta notación la forma normal del Ejemplo 3.8 puede representarse mediante el conjunto de cláusulas:

$$\{(\neg p(X_1) \vee r(f(X_1), g(X_1))), (q(f(X_1))) \vee r(f(X_1), g(X_1)))\}$$

Por consiguiente, un conjunto Δ de cláusulas debe verse como la representación de una fórmula que es la conjunción de todas las cláusulas de Δ , donde cada variable que aparece en cada una de las cláusulas de Δ está ligada mediante un cuantificador universal.

Así pues, cuando se manipulen las cláusulas que representan una forma normal, no deberá olvidarse que cada cláusula puede considerarse precedida por una serie de cuantificadores universales que ligan todas sus variables y transforman cada una de las cláusulas en una fórmula cerrada. Esto último nos autoriza a renombrar sus variables cuando sea conveniente y a considerar que las variables de una cláusula son locales a esa cláusula.

En ocasiones puede ser conveniente representar una cláusula como un conjunto de literales. Así una cláusula $(L_1 \vee L_2 \vee \dots)$ se representaría mediante el conjunto $\{L_1, L_2, \dots\}$. Con esta notación podemos manipular las cláusulas como si se tratase de conjuntos, aplicando operaciones como la diferencia de conjuntos. Por ejemplo, podemos referirnos a la cláusula C' que resulta de eliminar el literal L de la cláusula C simplemente como diferencia de conjuntos $C' = L \setminus C$.

Ejemplo 3.11

Las cláusulas del Ejemplo 3.8 se representan mediante los conjuntos

$$\{\neg p(X_1), r(f(X_1), g(X_1))\} \text{ y } \{q(f(X_1)), r(f(X_1), g(X_1))\}.$$

3.4.3. El papel de la forma clausal en los procedimientos de prueba

Como hemos venido diciendo desde el principio de este apartado, la eliminación de los cuantificadores existenciales de una fórmula para pasar a su forma normal no afecta a su inconsistencia. Esto queda enunciado en el siguiente teorema, una demostración del cual se puede ver en [26].

Teorema 3.1 *Sea un conjunto de cláusulas Δ que representa una forma estándar de una fórmula \mathcal{A} . La fórmula \mathcal{A} es insatisfacible si y solo si Δ es insatisfacible.*

Observaciones 3.3

1. Dado un conjunto de cláusulas Δ que representa una forma normal estándar de una fórmula $(\neg\mathcal{A})$, la formula \mathcal{A} será válida si y solo si el conjunto Δ es insatisfacible.
2. La equivalencia entre una fórmula \mathcal{A} y su forma normal representada por un conjunto de cláusulas Δ , solo se mantiene cuando \mathcal{A} es insatisfacible. Sin embargo cuando \mathcal{A} no es insatisfacible, entonces \mathcal{A} no es equivalente a Δ . Por ejemplo: Sea

$\mathcal{A} \equiv (\exists X)p(X)$, su forma normal es la fórmula $\mathcal{B} \equiv p(a)$; \mathcal{A} no es equivalente a \mathcal{B} , ya que se puede pensar en una interpretación en la que \mathcal{A} es verdadera mientras \mathcal{B} es falsa. Sea la interpretación I , con el dominio de interpretación $\mathcal{D} = \{1, 2\}$, en la que se asigna a la constante a el valor 1 y al símbolo de relación p la relación \bar{p} tal que $\bar{p}(1)$ es falso y $\bar{p}(2)$ es verdadero. Claramente, \mathcal{A} es verdadera en I , pero \mathcal{B} es falsa en I , por lo que $\mathcal{A} \not\leftrightarrow \mathcal{B}$.

3. Sea $\mathcal{A} \equiv \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \dots \wedge \mathcal{A}_n$. Podemos obtener por separado un conjunto de cláusulas Δ_i , donde cada Δ_i representa una forma normal de \mathcal{A}_i , y luego formar el conjunto $\Delta = \Delta_1 \cup \Delta_2 \cup \dots \cup \Delta_n$. Puede demostrarse que \mathcal{A} es insatisfacible si y solo si Δ es insatisfacible. Esto nos indica que también es posible obtener un conjunto de cláusulas Δ equivalente a \mathcal{A} , respecto a la insatisfacibilidad, hallando los conjuntos de cláusulas asociados a cada uno de sus componentes \mathcal{A}_i por separado.

□

El Teorema 3.1 es la razón de que la forma clausal tenga un lugar de privilegio en la demostración automática de teoremas. Como ya se ha dicho, en los demostradores automáticos las pruebas se hacen por refutación y consisten en probar la insatisfacibilidad de una fórmula. El Teorema 3.1 nos dice que basta con probar la insatisfacibilidad del conjunto de cláusulas que representa la forma estándar de una fórmula. En lo que sigue utilizaremos un conjunto de cláusulas como entrada a los procedimientos de prueba, en lugar de la fórmula original.

3.5 TEOREMA DE HERBRAND

En este apartado presentamos un algoritmo de prueba por refutación basado en un método semántico. Este algoritmo permitirá establecer la insatisfacibilidad de una fórmula.

Una fórmula (o, de forma correspondiente, el conjunto de cláusulas que de ella puede extraerse) es insatisfacible si y solo si es falsa para toda interpretación, sobre cualquier dominio. Es imposible comprobar todas las interpretaciones sobre todos los posibles dominios, ya que hay infinitos de ellos. Sería una fortuna que existiese un dominio de interpretación concreto en el que bastase explorar la insatisfacibilidad de una fórmula en las interpretaciones de ese dominio concreto para probar la falsedad de la fórmula sobre cualquier interpretación. Como veremos en los próximos apartados ese dominio existe y se denomina *universo de Herbrand*.

3.5.1. Universo de Herbrand e interpretaciones de Herbrand

Un conjunto de fórmulas Γ genera un lenguaje de primer orden cuyo alfabeto está constituido por los símbolos de constante, variable, función y relación que aparecen en dichas fórmulas. Supondremos que nuestro lenguaje de primer orden dispone al menos

de un símbolo de constante, ya que deseamos que pueda verse sobre alguien (o algo). Así pues, si no hay constantes en ese alfabeto, se añadirá una constante artificial “ a ”.

Ejemplo 3.12

Sea la fbf $\mathcal{A} \equiv (\exists Y)(\forall X)p(g(X), f(Y))$. El lenguaje de primer orden generado por \mathcal{A} tiene por alfabeto:

$$\mathcal{C} = \{a\} \quad \mathcal{F} = \{f, g\} \quad \mathcal{V} = \{X, Y\} \quad \mathcal{P} = \{p\}$$

Nótese que “ a ” es la constante artificial a la que nos referimos más arriba. Cuando en el futuro se hable de un lenguaje de primer orden, \mathcal{L} , estaremos haciendo referencia, de forma implícita, al lenguaje de primer orden generado por un conjunto de fórmulas.

Definición 3.6 (Universo de Herbrand) *Dado un lenguaje de primer orden \mathcal{L} , el universo de Herbrand $\mathcal{U}_{\mathcal{L}}$ de \mathcal{L} es el conjunto de todos los términos básicos de \mathcal{L} (i.e., los términos bien formados sin variables que se pueden construir con los símbolos del alfabeto de \mathcal{L}).*

Si queremos hacer explícito en el universo de Herbrand el conjunto de fórmulas Γ que genera el lenguaje de primer orden \mathcal{L} , escribiremos $\mathcal{U}_{\mathcal{L}}(\Gamma)$. Nótese que $\mathcal{U}_{\mathcal{L}} \neq \emptyset$, ya que hemos supuesto que nuestro lenguaje de primer orden, \mathcal{L} , tiene por lo menos un símbolo de constante. El siguiente algoritmo sistematiza la obtención del universo de Herbrand mediante sucesivas aproximaciones.

Algoritmo 3.4

Entrada: Un conjunto de fórmulas Γ . Un valor límite n .

Salida: El conjunto de términos básicos de profundidad n de Γ , $\mathcal{U}_n(\Gamma)$.

Comienzo

1. Construir el conjunto $\mathcal{U}_0(\Gamma)$ compuesto por todos los símbolos de constante que aparecen en Γ . Si en Γ no aparecen símbolos de constante entonces, $\mathcal{U}_0(\Gamma) = \{a\}$.

2. Para $i = 1$ hasta n hacer

1. Construir el conjunto $\mathcal{T}_i(\Gamma)$ compuesto por todos los términos básicos, $f(t_1, \dots, t_n)$, que pueden formarse combinando todos los símbolos de función k que aparecen en Γ con los términos generados en la anterior iteración, es decir, pertenecientes a $\mathcal{U}_{i-1}(\Gamma)$;

2. $\mathcal{U}_i(\Gamma) = \mathcal{T}_i(\Gamma) \cup \mathcal{U}_{i-1}(\Gamma)$.

Devolver $\mathcal{U}_n(\Gamma)$.

Fin

Este algoritmo obtiene el conjunto de *términos básicos de profundidad n* de Γ . Naturalmente, el universo de Herbrand del conjunto Γ es $\mathcal{U}_{\mathcal{L}}(\Gamma) = \mathcal{U}_{\infty}(\Gamma)$.

Ejemplo 3.13

Para la fbf \mathcal{A} del Ejemplo 3.12 tenemos que:

$$\begin{aligned}\mathcal{U}_0(\mathcal{A}) &= \{a\} \\ \mathcal{U}_1(\mathcal{A}) &= \{f(a), g(a)\} \cup \mathcal{U}_0(\mathcal{A}) = \{a, f(a), g(a)\} \\ \mathcal{U}_2(\mathcal{A}) &= \{f(a), f(f(a)), f(g(a)), g(a), g(f(a)), g(g(a))\} \cup \mathcal{U}_1(\mathcal{A}) \\ &= \{a, f(a), f(f(a)), f(g(a)), g(a), g(f(a)), g(g(a))\} \\ &\vdots\end{aligned}$$

Ejemplo 3.14

Sea el conjunto de cláusulas $\Delta = \{p(a), \neg p(X) \vee p(f(X))\}$. El lenguaje de primer orden generado por Δ tiene como alfabeto:

$$C = \{a\} \quad F = \{f\} \quad V = \{X\} \quad P = \{p\}$$

Para el conjunto Δ tenemos que:

$$\begin{aligned}\mathcal{U}_0(\Delta) &= \{a\} \\ \mathcal{U}_1(\Delta) &= \{f(a)\} \cup \mathcal{U}_0(\Delta) = \{a, f(a)\} \\ \mathcal{U}_2(\Delta) &= \{f(a), f(f(a))\} \cup \mathcal{U}_1(\Delta) = \{a, f(a), f(f(a))\} \\ &\vdots \\ \mathcal{U}_{\mathcal{L}}(\Delta) &= \{a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a)))), \dots\}\end{aligned}$$

Se debe notar que en $\mathcal{U}_{\mathcal{L}}$ pueden aparecer términos que no estaban en las fórmulas de partida.

Definición 3.7 (Base de Herbrand) Dado un lenguaje de primer orden \mathcal{L} , la base de Herbrand $\mathcal{B}_{\mathcal{L}}$ de \mathcal{L} es el conjunto de todos los átomos básicos de \mathcal{L} (i.e., aquéllos que se pueden formar con todos los símbolos de predicado del alfabeto de \mathcal{L} y todos los términos básicos de $\mathcal{U}_{\mathcal{L}}$).

Si queremos hacer explícito en la base de Herbrand el conjunto de fórmulas Γ que genera el lenguaje de primer orden \mathcal{L} , escribiremos $\mathcal{B}_{\mathcal{L}}(\Gamma)$.

Ejemplo 3.15

Sea el conjunto de cláusulas $\Delta = \{p(a), \neg p(X) \vee q(f(X))\}$. El lenguaje de primer orden generado por Δ tiene como alfabeto:

$$C = \{a\} \quad F = \{f\} \quad V = \{X\} \quad P = \{p, q\}$$

Para el conjunto Δ , al igual que en el Ejemplo 3.14, obtenemos que:

$$\mathcal{U}_{\mathcal{L}}(\Delta) = \{a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a)))), \dots\}$$

Así pues, la base de Herbrand es:

$$\begin{aligned} \mathcal{B}_{\mathcal{L}}(\Delta) = & \{p(a), p(f(a)), p(f(f(a))), p(f(f(f(a)))), p(f(f(f(f(a))))), \dots \\ & q(a), q(f(a)), q(f(f(a))), q(f(f(f(a)))), q(f(f(f(f(a))))), \dots\} \end{aligned}$$

Se debe notar que en $\mathcal{B}_{\mathcal{L}}(\Delta)$ pueden aparecer átomos que no estaban en las fórmulas de partida.

Definición 3.8 (Interpretación de Herbrand) Una interpretación de Herbrand de \mathcal{L} (o H-interpretación) es una interpretación $\mathcal{I} = (\mathcal{D}, \mathcal{T})$ de \mathcal{L} , definida en los siguientes términos:

1. \mathcal{D} es el propio universo de Herbrand para \mathcal{L} , $\mathcal{U}_{\mathcal{L}}$.

2. \mathcal{T} es la aplicación que asigna:

- A cada símbolo de constante ai de \mathcal{L} el mismo símbolo a ; i.e., $\mathcal{T}(a) = a$.
- A cada functor n -ario f de \mathcal{L} una función $\mathcal{T}(f) = \bar{f}$, tal que $\bar{f}: \mathcal{U}_{\mathcal{L}}^n \rightarrow \mathcal{U}_{\mathcal{L}}$ asocia a cada secuencia de términos básicos t_1, \dots, t_n de $\mathcal{U}_{\mathcal{L}}$ el término básico $f(t_1, \dots, t_n)$ de $\mathcal{U}_{\mathcal{L}}$.
- Si r es un símbolo de relación n -ario de \mathcal{L} , entonces se le puede asignar cualquier subconjunto $\mathcal{T}(r) = \bar{r}$ de n -tuplas de términos básicos de $\mathcal{U}_{\mathcal{L}}^n$.

Nótese que no hay restricción respecto de las relaciones de $\mathcal{U}_{\mathcal{L}}^n$ asignadas a cada uno de los símbolos de relación n -arios de \mathcal{L} . Así pues, puede haber varias interpretaciones de Herbrand para \mathcal{L} . Dado que, para un lenguaje de primer orden \mathcal{L} , todas las H-interpretaciones comparten como puntos en común la asignación hecha para los símbolos de constante y de función de \mathcal{L} (denominada *preinterpretación* en [86] y *álgebra de Herbrand* en [6]), una H-interpretación queda únicamente determinada dando la interpretación de sus símbolos de relación. Por otra parte, hay una correspondencia uno a uno entre las distintas H-interpretaciones y los subconjuntos de la base de Herbrand, $\mathcal{B}_{\mathcal{L}}$, que se hace explícito mediante la aplicación que asigna a cada H-interpretación el conjunto de átomos básicos:

$$\{r(t_1, \dots, t_n) \mid r \text{ es un símbolo de relación } n\text{-ario de } \mathcal{L} \text{ y } (t_1, \dots, t_n) \in \bar{r}\}.$$

Esto permite identificar las H-interpretaciones de \mathcal{L} con subconjuntos (posiblemente vacíos) de la base de Herbrand $\mathcal{B}_{\mathcal{L}}$, que es lo que haremos en el futuro. Así pues, de forma alternativa podemos considerar que una interpretación de Herbrand es cualquier subconjunto de la base de Herbrand $\mathcal{B}_{\mathcal{L}}$ de \mathcal{L} , mediante el cual se representa el conjunto de los átomos básicos que se evalúan al valor de verdad V de acuerdo con dicha interpretación.

Como convenio notacional supondremos que los átomos que no están en el conjunto, se evalúan al valor de verdad F .

Ejemplo 3.16

Sea el conjunto de cláusulas $\Delta = \{p(a), \neg p(X) \vee q(X)\}$. El lenguaje de primer orden generado por Δ tiene como alfabeto:

$$\begin{array}{lll} C & = & \{a\} \\ \mathcal{V} & = & \{X\} \end{array} \quad \begin{array}{lll} \mathcal{F} & = & \emptyset \\ \mathcal{P} & = & \{p, q\} \end{array}$$

que da lugar a un universo de Herbrand finito para el conjunto Δ : $\mathcal{U}_{\mathcal{L}}(\Delta) = \{a\}$. Así pues, la base de Herbrand también es finita: $\mathcal{B}_{\mathcal{L}}(\Delta) = \{p(a), q(a)\}$. En este caso son posibles las siguientes H-interpretaciones para Δ :

$$\begin{array}{lll} \mathcal{I}_1 & = & \{p(a), q(a)\} \\ \mathcal{I}_2 & = & \{p(a)\} \quad \text{i.e., } (q(a) \text{ es falso}) \\ \mathcal{I}_3 & = & \{q(a)\} \quad \text{i.e., } (p(a) \text{ es falso}) \\ \mathcal{I}_4 & = & \emptyset \end{array}$$

Nótese que \mathcal{I}_1 es la base de Herbrand $\mathcal{B}_{\mathcal{L}}(\Delta)$.

Observaciones 3.4

1. En [26] se utiliza el siguiente convenio notacional: Dado un conjunto de cláusulas Δ y supuesto que la base de Herbrand $\mathcal{B}_{\mathcal{L}}(\Delta) = \{\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n, \dots\}$, una H-interpretación se representa mediante un conjunto de literales $\{L_1, L_2, \dots, L_n, \dots\}$, donde L_i es o bien uno de los átomos \mathcal{A}_i o bien su negación $\neg \mathcal{A}_i$, para $i = 1, 2, \dots$. Este conjunto debe entenderse en los siguientes términos: si L_i es \mathcal{A}_i entonces, \mathcal{A}_i se evalúa al valor de verdad V en esa H-interpretación; en cambio si L_i es $\neg \mathcal{A}_i$ quiere decir que \mathcal{A}_i se evalúa al valor de verdad F . Con este convenio notacional, las posibles H-interpretaciones para el conjunto de cláusulas Δ del Ejemplo 3.16 se representan mediante los conjuntos:

$$\begin{array}{lll} \mathcal{I}_1 & = & \{p(a), q(a)\} \\ \mathcal{I}_2 & = & \{p(a), \neg q(a)\} \\ \mathcal{I}_3 & = & \{\neg p(a), q(a)\} \\ \mathcal{I}_4 & = & \{\neg p(a), \neg q(a)\} \end{array}$$

Esta forma de representar las H-interpretaciones hace explícito el valor de verdad asignado a cada uno de los átomos de la base de Herbrand, lo que la hace especialmente adecuada en la discusión del teorema de Herbrand (ver [26]). Las anteriores H-interpretaciones se corresponden con la siguiente tabla de verdad:

	$p(a)$	$q(a)$
I_1	V	V
I_2	V	F
I_3	F	V
I_4	F	F

2. Con este enfoque, el conjunto de todas las H-interpretaciones de \mathcal{L} se obtiene construyendo todas las aplicaciones posibles ($\mathcal{B}_{\mathcal{L}} \rightarrow \{V, F\}$) entre la base de Herbrand y el conjunto de los valores de verdad. Si la base de Herbrand $\mathcal{B}_{\mathcal{L}}$ es de cardinalidad⁵ finita, el álgebra combinatoria nos dice que el número de estas aplicaciones viene dado por las variaciones con repetición de dos elementos, $\{V, F\}$, tomados de $|\mathcal{B}_{\mathcal{L}}|$ en $|\mathcal{B}_{\mathcal{L}}|$. Esto es, $2^{|\mathcal{B}_{\mathcal{L}}|}$.
3. El conjunto potencia o conjunto de las partes de un conjunto, junto con las operaciones de inclusión \subseteq (que impone un orden parcial entre sus elementos) forma un retículo completo⁶. Por la forma en la que se ha definido, es evidente que el conjunto de todas las H-interpretaciones de un lenguaje de primer orden \mathcal{L} coincide con el conjunto de las partes de $\mathcal{B}_{\mathcal{L}}$: $\wp(\mathcal{B}_{\mathcal{L}})$. Por consiguiente, el conjunto de las H-interpretaciones es un retículo completo cuyo máximo es el conjunto $\mathcal{B}_{\mathcal{L}}$ y cuyo mínimo es el conjunto vacío \emptyset . Algunos autores denotan el *retículo completo de las H-interpretaciones* de un lenguaje de primer orden \mathcal{L} mediante el símbolo $2^{\mathcal{B}_{\mathcal{L}}}$.
4. El orden entre los elementos de un retículo pueden ilustrarse mediante un diagrama de Hasse [129]. La Figura 3.1 muestra el diagrama de Hasse para las H-interpretaciones del conjunto de cláusulas Δ del Ejemplo 3.16. En un diagrama de Hasse los nodos representan elementos del conjunto (en nuestro caso H-interpretaciones) y las líneas entre nodos enlazan los elementos comparables por la relación de orden (en nuestro caso la relación de inclusión de conjuntos). Los elementos se disponen de manera que los del nivel inferior son menores (incluidos), según la relación de orden, que los del nivel superior. En el diagrama de la Figura 3.1, I_2 es comparable con $\mathcal{B}_{\mathcal{L}}(\Delta)$, ya que $I_2 \subseteq \mathcal{B}_{\mathcal{L}}(\Delta)$, pero no es comparable con I_3 .

□

La propiedad más importante de las H-interpretaciones queda establecida en el siguiente teorema, que dice que basta explorar las H-interpretaciones de una fórmula para probar la insatisfacibilidad de dicha fórmula. Este hecho determina el papel de privilegio

⁵El cardinal de un conjunto finito C es su número de elementos y se denota mediante el símbolo $|C|$.

⁶Un conjunto parcialmente ordenado R se dice que es un *retículo completo* si todo subconjunto S de R tiene un supremo y un infímo. El *supremo* de un conjunto $S \subseteq R$ es la menor de sus cotas superiores. El *infímo* de un conjunto $S \subseteq R$ es la mayor de sus cotas inferiores.

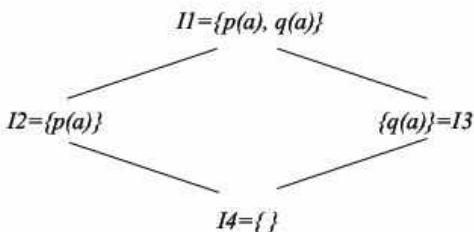


Figura 3.1 Diagrama de Hasse para las H-interpretaciones del Ejemplo 3.16.

de la forma clausal en los procedimientos de prueba por refutación. Una demostración de este teorema puede encontrarse en [26].

Teorema 3.2 *Sea Δ un conjunto de cláusulas. Δ es insatisfacible si y solo si Δ es falsa bajo todas las H-interpretaciones de Δ .*

Debido a este resultado, para probar la insatisfacibilidad de un conjunto de cláusulas no es necesario considerar otro tipo de interpretaciones que las de Herbrand, así que en el futuro, siempre que nos refiramos a interpretaciones nos estaremos refiriendo implícitamente a H-interpretaciones, a menos que se haga explícito otra cosa. Otras propiedades interesantes de las H-interpretaciones en relación con la satisfacibilidad de la cláusulas y los conjuntos de cláusulas se agrupan en la siguiente proposición. Antes de enunciar esta proposición se necesita introducir un nuevo concepto.

Definición 3.9 *Sea Δ un conjunto de cláusulas de un lenguaje de primer orden \mathcal{L} . Una instancia básica de una cláusula C de Δ es la cláusula obtenida sustituyendo cada una de las variables de C por elementos de $\mathcal{U}_{\mathcal{L}}(\Delta)$.*

Observe que, el proceso de generar instancias básicas de una cláusula se relaciona con el de valoración de una fórmula en una interpretación (véase la definición de valoración en el Apartado 2.2.3).

Proposición 3.1 *Sea Δ un conjunto de cláusulas e I una H-interpretación para Δ .*

1. *Sea C' una instancia básica de una cláusula C de Δ . Entonces C' es verdadera en I , si y solo si existe un literal positivo L'_1 de C' tal que $L'_1 \in I$ o bien un literal negativo L'_2 de C' tal que $\neg L'_2 \notin I$. En caso contrario, C' es falsa en I .*
2. *Sea C una cláusula de Δ . Entonces C es verdadera en I si y solo si para toda instancia básica C' de C se cumple que C' es verdadera en I .*
3. *Sea C una cláusula de Δ . Entonces C es falsa en I si y solo si existe al menos una instancia básica C' de C que no es verdadera en I .*

Estos resultados simplifican las acciones que habitualmente se realizan (cuando se aplican las definiciones del Apartado 2.2.3) para comprobar si una fórmula es satisfecha por una valoración en una interpretación o verdadera en una interpretación, es decir, satisfecha por todas las valoraciones en esa interpretación. Se sugiere al lector que intente demostrar los anteriores enunciados tomando como punto de partida las definiciones introducidas en el Apartado 2.2.3.

3.5.2. Modelos de Herbrand

Definición 3.10 (Modelo de Herbrand) *Un modelo de Herbrand para un conjunto de fórmulas Γ es una H-interpretación \mathcal{I} de Γ que es modelo de Γ (i.e., cada una de las fórmulas del conjunto es verdadera en la H-interpretación \mathcal{I}).*

Ejemplo 3.17

Sea el conjunto de cláusulas $\Delta = \{p(b), \neg p(X) \vee p(f(X)), \neg p(X) \vee q(b)\}$. El lenguaje de primer orden generado por Δ tiene como alfabeto:

$$\begin{array}{ll} C &= \{b\} & \mathcal{F} &= \{f\} \\ \mathcal{V} &= \{X\} & \mathcal{P} &= \{p, q\} \end{array}$$

que da lugar a un universo de Herbrand infinito para el conjunto Δ :

$$\mathcal{U}_{\mathcal{L}}(\Delta) = \{b, f(b), f(f(b)), f(f(f(b))), \dots\}$$

Así pues, la base de Herbrand también es infinita:

$$\begin{aligned} \mathcal{B}_{\mathcal{L}}(\Delta) &= \{p(b), p(f(b)), p(f(f(b))), p(f(f(f(b)))), \dots \\ &\quad q(b), q(f(b)), q(f(f(b))), q(f(f(f(b)))), \dots\} \end{aligned}$$

Algunas H-interpretaciones posibles para Δ son:

$$\begin{aligned} \mathcal{I}_1 &= \{p(b), q(b)\} \\ \mathcal{I}_2 &= \{p(b), p(f(b)), p(f(f(b))), p(f(f(f(b)))), \dots\} \\ \mathcal{I}_3 &= \{q(b), p(b), p(f(b)), p(f(f(b))), p(f(f(f(b)))), \dots\} \end{aligned}$$

A partir de la Proposición 3.1 vamos a estudiar cuáles de las anteriores H-interpretaciones son modelo del conjunto Δ .

- \mathcal{I}_1 no es modelo de Δ , ya que la cláusula $C_2 \equiv (\neg p(X) \vee p(f(X)))$ es falsa en \mathcal{I}_1 , debido a que existe al menos una instancia básica de C_2 (e.g., $\neg p(b) \vee p(f(b))$) que es falsa en \mathcal{I}_1 (ya que, $p(b) \in \mathcal{I}_1$ y $p(f(b)) \notin \mathcal{I}_1$).
- \mathcal{I}_2 no es modelo de Δ , ya que la cláusula $C_3 \equiv (\neg p(X) \vee q(b))$ es falsa en \mathcal{I}_2 , debido a que existe al menos una instancia básica de C_3 (e.g., $\neg p(b) \vee q(b)$) que es falsa en \mathcal{I}_1 (ya que, $p(b) \in \mathcal{I}_2$ y $q(b) \notin \mathcal{I}_2$).

- \mathcal{I}_3 es modelo de Δ , ya que: i) la cláusula $C_1 \equiv p(b)$ es verdadera en \mathcal{I}_3 , debido a que $p(b) \in \mathcal{I}_3$. ii) la cláusula $C_2 \equiv (\neg p(X) \vee p(f(X)))$ es verdadera en \mathcal{I}_3 , debido a que, para toda instancia básica de C_2 , $\neg p(t) \vee p(f(t))$, con $t \in \mathcal{U}_{\mathcal{L}}(\Delta)$, si $p(t) \in \mathcal{I}_3$ entonces $p(f(t)) \in \mathcal{I}_3$. iii) la cláusula $C_3 \equiv (\neg p(X) \vee q(b))$ es verdadera en \mathcal{I}_3 , puesto que cualquier instancia básica de C_3 contiene el literal $q(b)$ y $q(b) \in \mathcal{I}_3$.

Observación 3.1

Las operaciones de intersección, \cap , y unión, \cup , de conjuntos son leyes de composición interna⁷ en el conjunto de las partes de un conjunto. Las operaciones de intersección, \cap , y unión, \cup , tienen una interesante propiedad: sea un conjunto C y $S = \{S_1, S_2, \dots, S_n\} \subseteq \wp(C)$. Entonces $S_1 \cap S_2 \cap \dots \cap S_n$ (respectivamente $S_1 \cup S_2 \cup \dots \cup S_n$) es el ínfimo (supremo) del conjunto S , cuando los conjuntos se consideran ordenados según la relación de inclusión. Así pues, dado un conjunto de H-interpretaciones, la intersección de todas ellas es el ínfimo de ese conjunto. Esta propiedad nos permitirá establecer, en el Capítulo 5, el significado de un conjunto de cláusulas en función de un modelo mínimo que será la intersección de todos sus modelos.

3.5.3. El teorema de Herbrand y las dificultades para su implementación

A partir de los resultados del apartado anterior (Teorema 3.2 y Proposición 3.1) se infiere que, para conseguir probar la insatisfacibilidad de un conjunto de cláusulas $\Delta = \{C_1, \dots, C_2\}$, basta con analizar si, para toda H-interpretación \mathcal{I} , hay al menos una instancia básica de alguna cláusula de Δ que no es verdadera en \mathcal{I} . Esto nos sugiere un método semántico para probar la insatisfacibilidad de Δ , consistente en generar todas las posibles H-interpretaciones e ir comprobando si cada una de éstas hace falsa alguna instancia básica de una cláusula en Δ . Sin embargo, esto todavía puede ser inviable, ya que, como se ha advertido, el número de H-interpretaciones de un conjunto de fórmulas puede ser infinito. Herbrand demostró en 1930 que, para probar la insatisfacibilidad de un conjunto de cláusulas Δ , no es necesario generar todas las H-interpretaciones, sino que basta con generar un conjunto finito de H-interpretaciones parciales⁸ que hacen falsa alguna instancia básica C'_i de alguna $C_i \in \Delta$. Debido a que a partir del conjunto finito de H-interpretaciones parciales puede generarse el resto de las H-interpretaciones de Δ , el conjunto finito formado por las instancias básicas C'_i es a su vez un conjunto de instancias básicas de Δ insatisfacible.

Ejemplo 3.18

Sea el conjunto de cláusulas $\Delta = \{p(X), \neg p(f(a))\}$. El lenguaje de primer orden generado por Δ tiene como alfabeto:

⁷Dado un conjunto C sobre el que hay definida una operación $*$, decimos que es una ley de composición interna si y solo si, para todo elemento x e y de C , $(x * y) \in C$.

⁸Una H-interpretación parcial es una estructura en la que solo se han asignado valores de verdad a una parte de los átomos de la base de Herbrand.

$$\begin{array}{lll} C & = & \{a\} \\ V & = & \{X\} \end{array} \quad \begin{array}{lll} F & = & \{f\} \\ P & = & \{p\} \end{array}$$

Para el conjunto Δ , al igual que en el Ejemplo 3.14, obtenemos que:

$$\mathcal{U}_L(\Delta) = \{a, f(a), f(f(a)), f(f(f(a))), f(f(f(f(a)))), \dots\}$$

Así pues, la base de Herbrand es:

$$\mathcal{B}_L(\Delta) = \{p(a), p(f(a)), p(f(f(a))), p(f(f(f(a)))), p(f(f(f(f(a))))), \dots\}$$

Podemos construir las siguientes interpretaciones parciales para Δ :

- Definimos la H-interpretación parcial I'_1 como aquélla en la que $p(a) \notin I'_1$. Esto es, I'_1 caracteriza la clase de las H-interpretaciones para las que $p(a)$ es falso.
- Definimos la H-interpretación parcial I'_2 como aquélla en la que $p(a) \in I'_2$ y $p(fa) \in I'_2$. Esto es, I'_2 caracteriza la clase de las H-interpretaciones para las que tanto $p(a)$ como $p(f(a))$ son verdaderos.
- Definimos la H-interpretación parcial I'_3 como aquélla en la que $p(a) \in I'_3$ pero $p(fa) \notin I'_3$. Esto es, I'_3 caracteriza la clase de las H-interpretaciones para las que $p(a)$ es verdadero y $p(f(a))$ es falso.

Por el momento, no nos va a importar el valor de verdad que tomen el resto de los átomos de $\mathcal{B}_L(\Delta)$ en estas interpretaciones. Nótese que este conjunto de interpretaciones parciales es completo en el sentido de que cualquier posible H-interpretación está en alguna de las tres clases caracterizadas por I'_1 , I'_2 o I'_3 .

Árbol semántico Para construir un conjunto completo de H-interpretaciones parciales resulta conveniente usar el concepto de *árbol semántico*, introducido en [26]. La Figura 3.2 representa el árbol semántico (cerrado) asociado a este ejemplo: cada rama se corresponde con una interpretación parcial y el símbolo de negación que precede a ciertos átomos indica que dicho átomo es falso en esa interpretación.

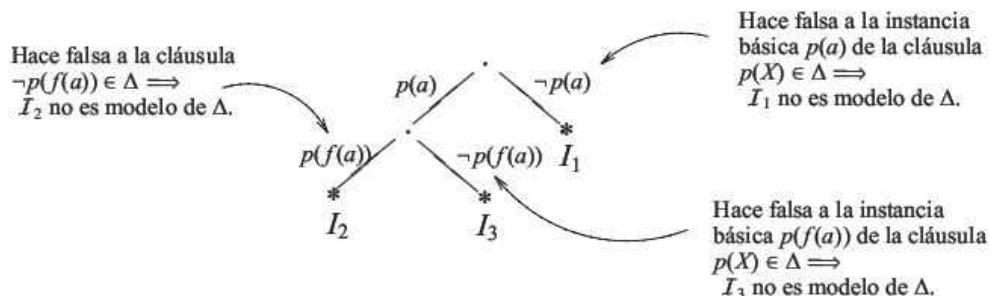


Figura 3.2 Árbol semántico.

El siguiente razonamiento justifica que el conjunto de interpretaciones parciales así construido es suficiente para probar la insatisfacibilidad de Δ :

- Las instancias básicas de $C_1 \equiv p(X)$ son: $p(a), p(f(a)), p(f(f(a))), \dots$
 - $p(a)$ es falsa en I'_1 , ya que en esta interpretación parcial $p(a) = F$;
 - $p(f(a))$ es falsa en I'_3 , ya que en esta interpretación parcial $p(f(a)) = F$.
- La única instancia básica de $C_2 \equiv \neg p(f(a))$ es: $\neg p(f(a))$.
 - $\neg p(f(a))$ es falsa en I'_2 , ya que en esta interpretación parcial $p(f(a))$ es verdadera.

Vemos que cada una de las H-interpretaciones parciales hace falsa una instancia básica de una cláusula de Δ . Por consiguiente, cualquier H-interpretación también hará falsa alguna de esas instancias básicas, así pues, Δ es insatisfacible.

Ahora, podemos agrupar las instancias básicas que son falsas en cada una de las H-interpretaciones parciales para formar el conjunto finito de instancias básicas $\Delta' = \{p(a), p(f(a)), \neg p(f(a))\}$, que es insatisfacible.

El lector puede comprobar que es posible extraer un conjunto finito de instancias básicas de cardinalidad mínima si construimos dos H-interpretaciones parciales tales que una de ellas caracteriza la clase de las H-interpretaciones para las que $p(f(a))$ es verdadero y la otra caracteriza la clase de las H-interpretaciones para las que $p(f(a))$ es falso. En este caso el conjunto insatisfacible de instancias básicas que se obtiene es: $\{p(f(a)), \neg p(f(a))\}$. En la actualidad, el teorema de Herbrand suele enunciarse en una versión que es más apta para su implementación en una máquina.

Teorema 3.3 (Teorema de Herbrand) *Un conjunto de cláusulas Δ es insatisfacible si y solo si existe un conjunto de instancias básicas de Δ que es insatisfacible.*

El teorema de Herbrand sugiere el siguiente procedimiento de prueba basado en los criterios semánticos introducidos hasta el momento.

Algoritmo 3.5

Entrada: Un conjunto de cláusulas Δ .

Salida: Un conjunto insatisfacible Δ' de cláusulas básicas (instancias de las cláusulas de Δ).

Comienzo

Inicialización: $i = 0$.

Repetir

1. Generar un conjunto de cláusulas básicas Δ'_i a partir de Δ :
 Δ'_i es el conjunto de todas las instancias básicas que pueden construirse a partir de cláusulas de Δ sustituyendo sus variables por términos del conjunto, $\mathcal{U}_i(\Delta)$, de términos básicos de profundidad i .
2. Comprobar la insatisfacibilidad de Δ'_i :
utilizando algún método de demostración de la insatisfacibilidad de la lógica de proposiciones .
3. Si Δ'_i no es insatisfacible entonces $i = i + 1$.

Hasta que Δ'_i sea insatisfacible.

Devolver $\Delta' = \Delta'_i$

Fin

Observaciones 3.5

1. Gilmore fue uno de los primeros en implementar, en 1960, el Algoritmo 5 en un computador. De él fueron las ideas de generar el conjunto Δ'_i a partir de los sucesivos $\mathcal{U}_i(\Delta)$ y de utilizar el método de la multiplicación, que se describe a continuación, para comprobar la insatisfacibilidad de Δ'_i .
2. El teorema de Herbrand garantiza que si el conjunto de cláusulas Δ es insatisfacible, el conjunto Δ' se obtendrá en un número finito de pasos. Ahora bien, si Δ no es insatisfacible, el Algoritmo 5 no terminará. Esto es lo máximo a lo que se puede aspirar respecto al problema de la insatisfacibilidad, teniendo en cuenta la indecidibilidad de la lógica de primer orden.



Ya que Δ'_i puede considerarse una conjunción de cláusulas básicas, se puede utilizar cualquier método de la lógica proposicional para comprobar su insatisfacibilidad. El *método de la multiplicación* consiste en considerar la conjunción de las cláusulas que forman Δ'_i y “multiplicarla” hasta alcanzar una forma normal disyuntiva. Entonces, cada conjunción que contiene un par de literales complementarios es eliminada. Si al seguir este proceso Δ'_i puede transformarse en la cláusula vacía \square entonces, Δ'_i es insatisfacible.

Ejemplo 3.19

Consideremos de nuevo el conjunto de cláusulas $\Delta = \{p(X), \neg p(f(a))\}$ del Ejemplo 3.18. Si aplicamos el Algoritmo 5, después de dos iteraciones se prueba que Δ es insatisfacible:

1. $\mathcal{U}_0(\Delta) = \{a\}$; $\Delta'_0 = \{p(a), \neg p(f(a))\}$. Al aplicar el método de la multiplicación se comprueba que Δ'_0 no es insatisfacible, ya que

$$\Delta'_0 = p(a) \wedge \neg p(f(a)) \neq \square.$$

2. $\mathcal{U}_1(\Delta) = \{a, f(a)\}; \Delta'_1 = \{p(a), p(f(a)), \neg p(f(a))\}$. Ahora, al aplicar el método de la multiplicación se comprueba que Δ'_1 es insatisfacible, ya que

$$\Delta'_1 = p(a) \wedge p(f(a)) \wedge \neg p(f(a)) = p(a) \wedge \square = \square.$$

Y el Algoritmo 5 termina devolviendo el conjunto $\Delta' = \{p(a), p(f(a)), \neg p(f(a))\}$.

Ejemplo 3.20

Sea el conjunto de cláusulas $\Delta = \{p(a), \neg p(X) \vee q(f(X)), \neg q(f(a))\}$. El Algoritmo 5 prueba la insatisfacibilidad de Δ en la primera iteración: $\mathcal{U}_0(\Delta) = \{a\}; \Delta'_0 = \{p(a), (\neg p(a) \vee q(f(a))), \neg q(f(a))\}$. Al aplicar el método de la multiplicación se comprueba que Δ'_0 es insatisfacible, ya que

$$\begin{aligned}\Delta'_0 &= p(a) \wedge (\neg p(a) \vee q(f(a))) \wedge \neg q(f(a)) \\ &= [(p(a) \wedge \neg p(a)) \vee (p(a) \wedge q(f(a)))] \wedge \neg q(f(a)) \\ &= (p(a) \wedge \neg p(a) \wedge \neg q(f(a))) \vee (p(a) \wedge q(f(a)) \wedge \neg q(f(a))) \\ &= (\square \wedge \neg q(f(a))) \vee (p(a) \wedge \square) \\ &= \square \vee \square \\ &= \square\end{aligned}$$

Y el Algoritmo 5 devuelve el conjunto $\Delta' = \{p(a), (\neg p(a) \vee q(f(a))), \neg q(f(a))\}$.

Los ejemplos anteriores se han escogido cuidadosamente para ilustrar y facilitar la aplicación del Algoritmo 5, lo que da lugar a una engañosa apariencia de simplicidad. La realidad es que, además de que es costoso generar el conjunto de cláusulas básicas Δ'_i cuando $\mathcal{U}_i(\Delta)$ tiene un tamaño considerable, el método de la multiplicación es muy inefficiente y conduce al conocido problema de la *explosión combinatoria*. Un conjunto con diez cláusulas básicas de dos literales cada una, da lugar a 2^{10} conjunciones de diez literales; ésta es una cantidad de información imposible de manipular en un tiempo razonable y aun de almacenar en la memoria de un computador⁹. Para paliar las inefficiencias del método de la multiplicación, Davis y Putnam introdujeron, también en 1960, un método más eficiente para comprobar la insatisfacibilidad de un conjunto de cláusulas básicas. Sin embargo, las mejoras introducidas por estos investigadores no fueron suficientes para convertir el Algoritmo 5 en un método practicable. Claramente, el problema radica en la generación de conjuntos de cláusulas básicas, obtenidos por instanciación a partir del conjunto de cláusulas original Δ , como medio para comprobar la insatisfacibilidad de Δ . En 1962, J.A. Robinson comenzó a ser consciente de que el único modo de evitar el problema de la explosión combinatoria era dotarse de un sistema de prueba en el que no fuese necesario generar de manera ciega todos los conjuntos de instancias básicas hasta encontrar (con suerte) el adecuado. En 1960, D. Prawitz había descrito

⁹Suponiendo que, en promedio, se necesite una palabra de memoria central para almacenar un literal, al menos se requeriría una memoria central de 128 Gbytes para almacenar las 2^{10} conjunciones.

un método de computar directamente el conjunto instancias básicas deseado, haciendo uso del procedimiento que hoy en día denominamos *unificación*¹⁰ que, en combinación con los procedimientos de prueba tradicionales de la lógica de predicados, orientados al modo de razonamiento humano, obtenía aún mayores mejoras que las logradas por Davis y Putnam. En el verano de 1963, J.A. Robinson concibió la idea de combinar el procedimiento de unificación con la regla del “corte” de Gentzen. Esto condujo a una regla de inferencia orientada a facilitar el cómputo en una máquina, para la que Robinson propuso el nombre de *resolución* [128].

En el siguiente capítulo nos centraremos en el estudio del principio de resolución de Robinson, que conduce a un método de prueba sintáctico.

RESUMEN

En este capítulo hemos presentado una breve introducción de un campo de la inteligencia artificial denominado demostración automática de teoremas. Hemos realizado un recorrido por algunos de los puntos que sirven de fundamentación a la programación lógica, centrándonos en los métodos de prueba semánticos basados en el teorema de Herbrand.

- La demostración automática de teoremas tiene como objetivo la obtención de algoritmos que permitan encontrar pruebas de teoremas matemáticos (expresados como fórmulas de la lógica de predicados).
- La demostración automática de teoremas tiene límites teóricos y prácticos. Por un lado la lógica de predicados es indecidible, lo que impide responder, en general, a la pregunta de si una fórmula es o no un teorema. Sin embargo, existen procedimientos de semidecisión (como los estudiados en este capítulo) que nos permiten determinar si una fórmula es un teorema cuando realmente lo es, aunque pueden no terminar si la fórmula no es en realidad un teorema. Desde un punto de vista práctico, el límite viene impuesto por la complejidad computacional de los problemas objeto de estudio y los algoritmos utilizados en su solución. Estos algoritmos consumen grandes recursos de cómputo.
- Existen diversas aproximaciones a la demostración automática de teoremas. Siguiendo a L. Wos et al. concluimos que, en el estudio de la demostración automática de teoremas, los aspectos esenciales son: i) el estudio de los procedimientos de prueba por refutación; ii) la definición de alguna forma de representación especial para las fórmulas; iii) el tipo de reglas de inferencia utilizadas y iv) las estrategias de búsqueda.

¹⁰La idea de buscar aquellas instancias que dan lugar a la demostración deseada utilizando unificación ya había sido propuesta por Herbrand en su tesis doctoral de 1930, pero pasó inadvertida hasta que Prawitz la redescubrió en 1960.

- Los procedimientos de prueba basados en el teorema de Herbrand (y en el principio de resolución de Robinson que describiremos en el próximo capítulo) realizan las demostraciones por refutación. En un procedimiento de prueba por refutación, establecer si una fórmula es consecuencia lógica de un conjunto de premisas se concreta en probar si el conjunto formado por las premisas y la negación de la conclusión a demostrar es insatisfacible (cuando el método de prueba es semántico) o, equivalentemente, es inconsistente (cuando el método de prueba es sintáctico).
- Hemos estudiado formas normales o estándares de representación de fórmulas: forma normal disyuntiva y conjuntiva, forma normal prenexa y forma normal de Skolem. La idea que nos ha guiado ha sido la de simplificar (respecto a algún criterio – por ejemplo, el tipo de conectivas y cuantificadores empleados) la representación de la fórmula original.
- La mayoría de las técnicas de demostración automática hacen uso de una representación especial para las fórmulas denominada forma clausal. La forma clausal es una forma de representación estándar en la que la fórmula original se representa por un conjunto de cláusulas. Una cláusula es una disyunción de cero o más literales. Una cláusula con cero literales se denomina cláusula vacía y se denota mediante el símbolo \square . Una cláusula vacía representa una contradicción.
- Una fórmula no es lógicamente equivalente a las cláusulas que la representan, sin embargo ambas son equivalentes respecto a la insatisfacibilidad. Esto es, una fórmula (o conjunto de fórmulas) es insatisfacible si y solo si el conjunto de cláusulas que la representa es insatisfacible. Por consiguiente, en un procedimiento de prueba por refutación basta con probar la insatisfacibilidad del conjunto de cláusulas, pudiéndonos desentender de la fórmula (o conjunto de fórmulas) original.
- Para comprobar la insatisfacibilidad de un conjunto de cláusulas basta con inspeccionar las llamadas interpretaciones de Herbrand (H-interpretación). Una H-interpretación es aquella que tiene como dominio de discurso el universo de Herbrand (el conjunto de todos los términos básicos que pueden generarse con los símbolos de constante y de función de un lenguaje de primer orden) y asigna a cada símbolo de constante él mismo, a cada símbolo de función una función en el universo de Herbrand y a cada símbolo de relación una relación en el universo de Herbrand.
- La base de Herbrand es el conjunto de todas las fórmulas atómicas básicas que pueden construirse con los símbolos de relación y los términos del universo de Herbrand de un lenguaje de primer orden. Una interpretación de Herbrand queda completamente caracterizada mediante un subconjunto de la base de Herbrand, que agrupa los hechos verdaderos en dicha interpretación.

- Herbrand demostró en 1930 que, para probar la insatisfacibilidad de un conjunto de cláusulas, no es necesario generar todas las H-interpretaciones sino que basta con generar un conjunto finito de H-interpretaciones parciales. Este resultado se conoce con el nombre de “teorema de Herbrand”. En la actualidad, el teorema de Herbrand suele enunciarse de una forma que es más apta para su implementación en una máquina: Un conjunto de cláusulas Δ es insatisfacible si y solo si existe un conjunto de instancias básicas de Δ insatisfacible.
- Los demostradores de teoremas automáticos basados en el método semántico que sugiere el teorema de Herbrand (Gillmore, 1960) son inviables, por conducir al llamado ‘problema de la explosión combinatoria’. El Principio de resolución de Robinson (1965) es un método de prueba sintáctico que intenta eliminar el problema de la explosión combinatoria evitando generar de manera ciega todos los conjuntos de instancias básicas hasta encontrar (con suerte) el adecuado. La regla de resolución es una regla de inferencia muy potente que combina el procedimiento de unificación de Prawitz con la regla del corte de Gentzen. Estudiaremos en detalle esta regla en el próximo capítulo.

CUESTIONES Y EJERCICIOS

Cuestión 3.1 a) ¿Qué es un demostrador automático de teoremas? b) ¿Cuáles son las acciones típicas que realiza un demostrador automático de teoremas?

Cuestión 3.2 ¿Cuáles son los límites prácticos de la demostración automática de teoremas?

Cuestión 3.3 a) ¿Qué es un procedimiento de prueba por refutación? b) Justifique la corrección de las acciones desarrolladas en un procedimiento de prueba por refutación.

Ejercicio 3.4 Transforme las siguientes fórmulas de la lógica de proposiciones

- | | |
|--|---|
| 1. $p \rightarrow \neg p$ | 6. $(\neg p \wedge q) \rightarrow r$ |
| 2. $\neg(p \rightarrow q)$ | 7. $\neg(p \vee \neg q) \wedge (s \rightarrow t)$ |
| 3. $p \vee (\neg p \wedge q \wedge r)$ | 8. $\neg(p \wedge q) \wedge (p \vee q)$ |
| 4. $(\neg p \wedge q) \rightarrow (p \wedge \neg q)$ | 9. $p \rightarrow (q \wedge r \rightarrow s)$ |
| 5. $\neg(p \rightarrow q) \vee (p \vee q)$ | 10. $(p \rightarrow q) \rightarrow r$ |

a) a una forma normal disyuntiva; b) a una forma normal conjuntiva.

Ejercicio 3.5 Verifique la equivalencia lógica de las siguientes fórmulas de la lógica de proposiciones:

1. $[(p \rightarrow q) \wedge (p \rightarrow r)] \Leftrightarrow [p \rightarrow (q \wedge r)],$
2. $[(p \rightarrow q) \rightarrow (p \wedge q)] \Leftrightarrow [(\neg p \rightarrow q) \wedge (q \rightarrow p)],$
3. $[p \wedge q \wedge (\neg p \vee \neg q)] \Leftrightarrow [\neg p \wedge \neg q \wedge (p \vee q)].$

Se sugiere utilizar las equivalencias de la Tabla 3.1 para tratar de transformar a la misma forma normal las fórmulas que aparecen a cada lado del signo “ \Leftrightarrow ”.

Ejercicio 3.6 Transforme a forma prenexa las siguientes fórmulas:

1. $(\forall X)[p(X) \rightarrow (\exists Y)q(X, Y)],$
2. $(\exists X)\{\neg[(\exists Y)p(X, Y)] \rightarrow ((\exists Z)q(Z) \rightarrow r(X))\},$
3. $(\forall X)(\forall Y)[(\exists Z)p(X, Y, Z) \wedge ((\exists U)q(X, U) \rightarrow (\exists V)q(Y, V))].$

Ejercicio 3.7 ([26]) Consideremos que $(\exists X)(\forall Y)M[X, Y]$ es una forma normal prenexa de una fórmula F , donde $M[X, Y]$ contiene sólamente las variables X e Y . Sea f un símbolo de función que no ocurre en $M[X, Y]$. Pruebe que F es válida si y solo si $(\exists X)M[X, fX]$ es válida.

Ejercicio 3.8 Obtenga una forma estándar para cada una de las siguientes fórmulas:

1. $\neg[(\forall X)p(X) \rightarrow (\exists Y)(\forall Z)q(Y, Z)],$
2. $\neg[(\forall X)p(X) \rightarrow (\exists)p(Y)],$
3. $[(\exists X)p(X) \vee (\exists X)q(X)] \rightarrow (\exists X)[p(X) \vee q(X)],$
4. $(\forall X)[p(X) \rightarrow (\forall Y)(q(X, Y) \rightarrow \neg(\forall Z)r(Y, X, Z))].$

Ejercicio 3.9 ([26]) Sean S_1 y S_2 la forma estándar de las fórmulas F_1 y F_2 , respectivamente. Si asumimos $S_1 \Leftrightarrow S_2$, ¿se cumple que $F_1 \Leftrightarrow F_2$? Explique la respuesta.

Ejercicio 3.10 a) ¿Qué es una cláusula? b) Exprese en forma clausal la fórmula de la lógica de proposiciones $\neg(((p \vee \neg q) \rightarrow r) \rightarrow (p \wedge r))$. c) Exprese en forma clausal las siguientes fórmulas de la lógica de predicados:

1. $(\forall X)(\forall Y)(p(X, Y) \vee \neg q(X) \vee \neg q(Y) \vee p(Y, X)),$
2. $(\forall X)(\exists Y)(p(X) \wedge q(Y) \rightarrow r(Y)),$
3. $(\forall X)p(X) \rightarrow (\exists X)[(\forall Z)q(X, Z) \vee (\forall Z)(\forall Y)r(X, Y, Z)],$

$$4. (\forall X)(\exists Y)[p(X) \rightarrow q(X, Y)] \rightarrow (\exists Y)[p(Y) \wedge (\exists Z)q(Y, Z)].$$

Ejercicio 3.11 (Celos en el Puerto) Dados los siguientes enunciados:

1. *Todas las porteñas alegres tienen un amigo marino.*
2. *Ningún porteño feliz está casado con una porteña triste.*
3. *Los porteños casados con amigas de marinos son burlados y son marinos.*
4. *Cornelio y Cornelina son un matrimonio de porteños felices.*
5. *Cete y Tita son un matrimonio de porteños felices.*
6. *Los marinos casados son todos felices.*

a) Tradúzcalos al lenguaje formal de la lógica de predicados. [Ayuda: Se recomienda una primera lectura del Apartado 8.2.4 antes de afrontar este ejercicio.] b) Exprese en forma clausal las fórmulas obtenidas en el paso anterior.

Ejercicio 3.12 (Cuerpos Celestes) Dados los siguientes enunciados:

1. *Solamente las lunas orbitan alrededor de un planeta.*
2. *La Luna orbita alrededor de la Tierra.*
3. *Los planetas del sistema solar orbitan alrededor del Sol.*
4. *Venus y la Tierra son planetas del sistema solar.*
5. *El Sol es una estrella.*
6. *Toda estrella, luna o planeta es un cuerpo celeste.*
7. *¿Cuáles son los cuerpos celestes?*

a) Tradúzcalos al lenguaje formal de la lógica de predicados. [Ayuda: Se recomienda una primera lectura del Apartado 8.2.4 antes de afrontar este ejercicio.] b) Exprese en forma clausal las fórmulas obtenidas en el paso anterior.

Cuestión 3.13 Defina los siguientes conceptos: Universo de Herbrand, Base de Herbrand e Interpretación de Herbrand.

Ejercicio 3.14 Considere el conjunto de cláusulas $\Delta = \{p(X), q(b) \vee \neg p(fY)\}$, y obtenga los conjuntos $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2$ y \mathcal{U}_3 del lenguaje de primer orden generado por Δ .

Ejercicio 3.15 Dado el conjunto de cláusulas: $\Delta = \{p(X), p(f(Y)) \vee \neg p(Y)\}$. a) Calcule los conjuntos \mathcal{U}_0 , \mathcal{U}_1 , \mathcal{U}_2 y \mathcal{U}_3 de Δ . b) Obtenga el Universo de Herbrand para el lenguaje de primer orden generado por Δ (es decir, calcule el límite de $\mathcal{U}_n(\Delta)$, con n tendiendo a infinito).

Ejercicio 3.16 Sea el conjunto de cláusulas

$$\Delta = \{p(b), \neg p(X) \vee p(f(X)), \neg p(X) \vee q(b)\}.$$

a) Indique el alfabeto del lenguaje de primer orden generado por Δ y b) Muestre el universo de Herbrand $\mathcal{U}_{\mathcal{L}}(\Delta)$ para ese lenguaje.

Cuestión 3.17 a) Enumere las condiciones que deben cumplirse para que, dado un lenguaje de primer orden \mathcal{L} , una interpretación I del lenguaje sea de Herbrand. Céntrese en aquellas características que la distinguen de una interpretación ordinaria. b) ¿Por qué un subconjunto de la base de Hebrand de un lenguaje de primer orden determina una interpretación de Hebrand?

Ejercicio 3.18 Considere la cláusula $C \equiv p(X) \vee q(X, f(X))$ y la interpretación

$$\begin{aligned} I = & \{q(a, f(a)), q(a, f(f(f(a)))), \dots, \\ & q(f(a), f(a)), q(f(a), f(f(f(a)))), \dots\}. \end{aligned}$$

¿ I satisface C ?

Ejercicio 3.19 Dado el conjunto de cláusulas $\Gamma = \{p(X), q(fY)\}$ y la interpretación

$$\begin{aligned} I = & \{p(a), p(f(a)), p(f(f(a))), \dots, \\ & q(a), q(f(f(a))), q(f(f(f(f(a))))), \dots\}. \end{aligned}$$

¿ I satisface Γ ?

Ejercicio 3.20 Para el conjunto de cláusulas del Ejercicio 3.14, ¿es posible obtener una interpretación I que satisface Δ ? Si lo es, dé una; si no, diga por qué no es posible.

Cuestión 3.21 a) ¿Cuándo se dice que una interpretación de Hebrand es modelo de una cláusula? b) ¿Qué se entiende por interpretación de Hebrand modelo de un conjunto de cláusulas?

Cuestión 3.22 Dada la cláusula $C = \{p \vee q\}$, indique cuál de los siguientes conjuntos no es modelo de C :

- | | |
|--------------|------------------|
| 1. $\{p\}$. | 3. $\{p, q\}$. |
| 2. $\{q\}$. | 4. \emptyset . |

Ejercicio 3.23 Dado el conjunto de cláusulas Δ del Ejercicio 3.16 y las siguientes H -interpretaciones:

$$\mathcal{I}_1 = \{p(b), q(b)\},$$

$$\mathcal{I}_2 = \{p(b), p(f(b)), p(f(f(b))), p(f(f(f(b))))\ldots\},$$

$$\mathcal{I}_3 = \{q(b), p(b), p(f(b)), p(f(f(b))), p(f(f(f(b))))\ldots\},$$

decida cuales son modelo del conjunto Δ .

Ejercicio 3.24 Dada la fórmula $(\exists X)(\forall Y)p(g(X), f(Y))$, calcule su universo de Herbrand asociado así como la base de Herbrand. Dé una interpretación de Herbrand que sea modelo de la anterior fórmula.

Cuestión 3.25 Dado el siguiente conjunto de cláusulas Δ :

$$\Delta = \{p(X, Y) \vee \neg q(X) \vee \neg r(Y), q(a), q(b), r(a)\},$$

compruebe si el conjunto $\{r(a), q(a), q(b), p(a, a), p(b, a), p(b, b)\}$ es un modelo de Herbrand de Δ .

Ejercicio 3.26 Considere el conjunto de cláusulas

$$\Gamma = \{p(X), \neg p(X) \vee q(X, a), \neg q(Y, a)\}.$$

a) Halle la base de Herbrand de Γ , $\mathcal{B}_{\mathcal{L}}(\Gamma)$. b) Dibuje un árbol semántico completo para Γ . c) Dibuje un árbol semántico cerrado para Γ . d) ¿es Γ un conjunto de cláusulas insatisfacible?

Ejercicio 3.27 Sea el conjunto de cláusulas

$$\Delta = \{p(X), q(X, f(X)) \vee \neg p(X), \neg q(g(Y), Z)\}.$$

Halle un conjunto de instancias básicas de Δ que sea insatisfacible. ¿Qué puede decirse respecto a la insatisfacibilidad del conjunto de cláusulas Δ ?

De la Demostración Automática a la Programación Lógica (II): el principio de resolución de Robinson

Para evitar las inefficiencias de las implementaciones directas del teorema de Herbrand, producidas por la generación sistemática de conjuntos de cláusulas básicas y la posterior comprobación de su insatisfacibilidad, introducimos el *principio de resolución de Robinson* (1965), que se aleja de los métodos basados en el teorema de Herbrand al permitir deducir consecuencias universales de enunciados universales. El principio de resolución es una regla de inferencia que se aplica a fórmulas en forma clausal y que, junto con el procedimiento de unificación, constituye un cálculo deductivo (que no precisa de axiomas u otras reglas de inferencia para ser completo). Al englobar varias reglas de inferencia conocidas (e.g., *modus ponens* e *instanciación*) es más adecuado para la mecanización que los sistemas de inferencia tradicionales que, si bien están más próximos a la forma del razonamiento humano, requieren pasos de deducción simples y elementales que conducen a demostraciones largas y tediosas [128]. La completitud (refutacional) del principio de resolución hace que éste proporcione un procedimiento sintáctico de prueba por refutación.

Básicamente, la idea esencial del método es la siguiente. Cuando se usa el principio de resolución como regla de inferencia en un sistema de demostración automática de teoremas, el conjunto de fórmulas a partir del cual deseamos probar un teorema debe expresarse en forma clausal. Después negamos la fórmula que queremos probar como teorema y la expresamos, también, en forma clausal, obteniendo de esta manera un conjunto de cláusulas. A partir de ahí se desencadena una secuencia de aplicaciones de la regla de inferencia de resolución a las cláusulas de este conjunto, obteniendo un

conjunto de cláusulas mayor. Si el teorema a demostrar es cierto, por tratarse de un método de prueba por refutación y completo, tras algún paso de resolución se inferirá la cláusula vacía, \square , señalando que hemos llegado a la buscada contradicción.

4.1 EL PRINCIPIO DE RESOLUCIÓN EN LA LÓGICA DE PROPOSICIONES

La mejor manera de obtener una idea general de la regla de inferencia de resolución es entender cómo se aplica en el caso de la lógica proposicional (lo que es equivalente a utilizar únicamente cláusulas básicas).

El Principio de resolución de Robinson para la lógica de proposiciones puede verse como una generalización de la regla de *Modus Tollens*:

$$\frac{(\neg p \vee q), \neg q}{\neg p}$$

El principio general puede enunciarse del siguiente modo:

Para todo par de cláusulas C_1 y C_2 , si existe un literal $L_1 \equiv \mathcal{A}$ en C_1 que es complementario de otro literal $L_2 \equiv \neg\mathcal{A}$ en C_2 , entonces se eliminan L_1 y L_2 de C_1 y C_2 , respectivamente, y se construye la disyunción de las cláusulas restantes. A la cláusula obtenida se le llama *resolvente* de C_1 y C_2 . Las cláusulas C_1 y C_2 se denominan *cláusulas padres*.

Más formalmente, podemos expresar el principio de resolución en términos de la siguiente regla de inferencia:

$$\frac{C_1, C_2}{(C_1 \setminus \{\mathcal{A}\}) \vee (C_2 \setminus \{\neg\mathcal{A}\})} \text{ si } \mathcal{A} \in C_1, \neg\mathcal{A} \in C_2$$

Ejemplo 4.1

Consideremos las siguientes cláusulas: $C_1 \equiv (p \vee r)$ y $C_2 \equiv (\neg p \vee q)$. La cláusula C_1 contiene el literal p , que es complementario del literal $\neg p$ de C_2 . Por lo tanto, al borrar p y $\neg p$ de C_1 y C_2 , respectivamente, y construir la disyunción de las cláusulas restantes r y q , obtenemos el resolvente ($r \vee q$).

Ejemplo 4.2

Sean las cláusulas: $C_1 \equiv (\neg p \vee q \vee r)$ y $C_2 \equiv (\neg q \vee s)$. El resolvente de C_1 y C_2 es $(\neg p \vee r \vee s)$.

Tabla 4.1 Algunos hechos notables sobre el principio de resolución.

Cláusulas padres	Resolvente	Observación
$C_1 : p \vee q$ $C_2 : \neg p$	$C : q$	Equivale a la regla de inferencia del <i>silogismo disyuntivo</i> .
$C_1 : \neg p \vee q$ $C_2 : p$	$C : q$	Ya que, $(\neg p \vee q) \Leftrightarrow (p \rightarrow q)$ lo anterior equivale a la regla de inferencia de <i>modus ponens</i> : Si $(p \rightarrow q)$ y p entonces q .
$C_1 : \neg p \vee q$ $C_2 : \neg q$	$C : \neg p$	Equivale a la regla de inferencia de <i>modus tollens</i> : Si $(p \rightarrow q)$ y $\neg q$ entonces $\neg p$.
$C_1 : \neg p \vee q$ $C_2 : \neg q \vee r$	$C : \neg p \vee r$	Equivale a la regla de inferencia del <i>silogismo hipotético</i> : Si $p \rightarrow q$ y $q \rightarrow r$ entonces $p \rightarrow r$.
$C_1 : p \vee q$ $C_2 : \neg p \vee q$	$C : q$	La cláusula $q \vee q$ es lógicamente equivalente a q . Este resolvente se denomina <i>fusión</i> .
$C_1 : p \vee q$ $C_2 : \neg p \vee \neg q$	$C : p \vee \neg p$ $C : q \vee \neg q$	Son posibles dos resolventes; ambos son tautologías.
$C_1 : p$ $C_2 : \neg p$	$C : \square$	El resolvente de dos cláusulas unitarias contrarias es la cláusula vacía.

Ejemplo 4.3

Consideremos las cláusulas: $C_1 \equiv (\neg p \vee q)$ y $C_2 \equiv (\neg p \vee r)$. Al no existir un literal en C_1 que sea complementario a otro en C_2 , no existe resolvente de C_1 y C_2 .

Es interesante observar, como pone de manifiesto la Tabla 4.1, que el principio de resolución es una regla de inferencia muy potente que incorpora otras reglas de inferencia de la lógica.

El siguiente teorema establece una propiedad importante del resolvente de dos cláusulas C_1 y C_2 .

Teorema 4.1 *Dadas dos cláusulas C_1 y C_2 , un resolvente C de C_1 y C_2 es una consecuencia lógica de C_1 y C_2 .*

Por definición, el resolvente C es consecuencia lógica de C_1 y C_2 si y solo si para cualquier valoración v , si $v(C_1) = V$ y $v(C_2) = V$ entonces $v(C) = V$. Para demostrar este resultado, sin pérdida de generalidad, puede suponerse que $C_1 \equiv \mathcal{A} \vee Q_1$, $C_2 \equiv \neg \mathcal{A} \vee Q_2$ y $C \equiv Q_1 \vee Q_2$. Entonces, $v(C_1) = V$ si y solo si $v(\mathcal{A}) = V$ o $v(Q_1) = V$. A partir de este punto procedemos por casos:

- Caso i) $v(\mathcal{A}) = V$, entonces $v(\neg\mathcal{A}) = F$ y dado que $v(C_2) = v(\neg\mathcal{A} \vee Q_2) = V$, se sigue que $v(Q_2) = V$; luego $v(Q_1 \vee Q_2) = V$;
- Caso ii) $v(Q_1) = V$, en este caso, de forma inmediata, $v(Q_1 \vee Q_2) = V$. Por consiguiente, $v(C) = V$ siempre que $v(C_1) = V$ y $v(C_2) = V$.

Del resultado anterior se infiere de forma inmediata la corrección del principio de resolución ya que, si existe una prueba por resolución de una cláusula \mathcal{A} a partir de un conjunto de cláusulas Δ , el conjunto Δ y la cláusula \mathcal{A} estarán en una relación de consecuencia lógica. Si la cláusula \mathcal{A} es la cláusula vacía, \square , entonces ésta es una consecuencia lógica de Δ . Sin embargo, \square solo puede ser una consecuencia lógica de un conjunto de cláusulas insatisfacible. Por tanto, Δ es insatisfacible. También es posible demostrar que el principio de resolución es una regla de inferencia (refutacionalmente) completa para la lógica de proposiciones. Esto es, si un conjunto de cláusulas Δ es insatisfacible entonces podemos usar el principio de resolución para generar, en un número de pasos, la cláusula vacía, \square , a partir de Δ .

Antes de seguir adelante conviene precisar la forma que adquiere una deducción en este contexto.

Definición 4.1 (Deducción) Dado un conjunto de cláusulas Δ , una deducción (resolución) de C a partir de Δ es un secuencia finita C_1, C_2, \dots, C_k de cláusulas tal que C_i es una cláusula de Δ o un resolvente de cláusulas que preceden a C_i , y $C_k = C$. Una deducción de \square a partir de Δ se denomina refutación o prueba de Δ .

En ocasiones también diremos que una cláusula C puede *deducirse* o *derivarse* de Δ si existe una deducción de C a partir de Δ .

Una vez definido el concepto de deducción, podemos agrupar los resultados anteriores y establecer el siguiente teorema que afirma la corrección y completitud del Principio de Resolución.

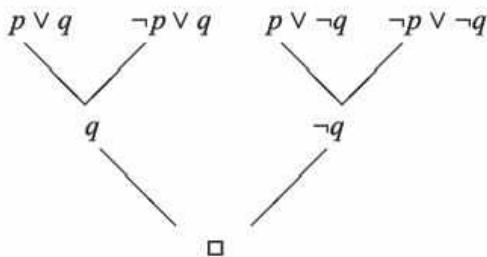
Teorema 4.2 Sea Δ un conjunto de cláusulas. Δ es insatisfacible si y solo si existe una deducción de la cláusula vacía, \square , a partir de Δ .

Los siguientes ejemplos ilustran cómo emplear el Principio de resolución para probar la insatisfacibilidad de un conjunto de cláusulas.

Ejemplo 4.4

Dado el conjunto de cláusulas $\Delta = \{(\neg p \vee q), \neg q, p\}$, podemos construir la siguiente refutación:

- (1) $\neg p \vee q$
- (2) $\neg q$
- (3) p

**Figura 4.1** Árbol de deducción para la refutación del Ejemplo 4.5.

generamos los siguientes resolventes

- (4) $\neg p$ de (1) y (2)
- (5) \square de (3) y (4)

Por tanto, Δ es insatisfacible.

Ejemplo 4.5

Para el conjunto de cláusulas $\Delta = \{(p \vee q), (\neg p \vee q), (p \vee \neg q), (\neg p \vee \neg q)\}$, existe la siguiente refutación. Partiendo de:

- (1) $p \vee q$
- (2) $\neg p \vee q$
- (3) $p \vee \neg q$
- (4) $\neg p \vee \neg q$

generamos los siguientes resolventes:

- (5) q de (1) y (2)
- (6) $\neg q$ de (3) y (4)
- (7) \square de (5) y (6)

Ya que \square es derivable, Δ es insatisfacible. La deducción anterior puede representarse en forma de árbol (denominado árbol de deducción) como se muestra en la Figura 4.1.

Para abordar el estudio del principio de resolución en el contexto de la lógica de primer orden es necesario introducir una serie conceptos previos, entre los que destaca el de unificación y unificador más general. Dedicamos los próximos apartados a esta tarea.

4.2 SUSTITUCIONES

Si queremos formalizar adecuadamente los conceptos de *unificación*, *resolución* y *respuesta computada*, entre otros, es imprescindible introducir el concepto de sustitución.

Definición 4.2 (Sustitución) Una sustitución σ es una aplicación que asigna, a cada variable X del conjunto de las variables \mathcal{V} de un lenguaje de primer orden \mathcal{L} , un término $\sigma(X)$ del conjunto de los términos \mathcal{T} de \mathcal{L} .

$$\begin{array}{rcl} \sigma : & \mathcal{V} & \longrightarrow \mathcal{T} \\ & X & \leftrightarrow \sigma(X) \end{array}$$

Es habitual representar las sustituciones como conjuntos finitos de la forma

$$\{X_1/t_1, X_2/t_2, \dots, X_n/t_n\}$$

donde, para cada i , t_i es un término diferente de X_i . Los elementos X_i/t_i de la sustitución, reciben el nombre de *enlaces*. El conjunto $\{X_1, X_2, \dots, X_n\}$ se denomina *dominio* de la sustitución y el conjunto $\{t_1, t_2, \dots, t_n\}$ se denomina *rango* de la sustitución. Nosotros denotaremos el dominio de una sustitución σ mediante el símbolo $\text{Dom}(\sigma)$ y su rango mediante el símbolo $\text{Ran}(\sigma)$. La anterior forma de representar una sustitución puede considerarse como una definición por extensión de una sustitución, en la que a cada variable X_i se le asocia su imagen t_i (mientras el resto de variables que no están en el conjunto no se modifican). Con esta notación, la *sustitución identidad*, *id*, se representa mediante el conjunto vacío de elementos: $\{\}$. Por este motivo, la sustitución identidad también recibe el nombre de *sustitución vacía*. Una sustitución donde los términos t_i son básicos (i.e., no contienen variables) se denomina *sustitución básica*.

Ejemplo 4.6

Ejemplos de sustituciones son:

$$\theta_1 = \{X/f(Z), Z/Y\}; \quad \theta_2 = \{X/a, Y/g(Y), Z/f(g(b))\}$$

El dominio de las sustituciones se puede ampliar a los términos y a las fbf's, es decir, a las expresiones de \mathcal{L} . Una definición formal del concepto de aplicación de una sustitución σ a una expresión E puede realizarse por inducción estructural sobre la forma de las expresiones de \mathcal{L} [75]. Aquí, presentamos una definición más intuitiva extraída de [26].

Definición 4.3 (Instancia) La aplicación de una sustitución $\sigma = \{X_1/t_1, X_2/t_2, \dots, X_n/t_n\}$ a una expresión E , denotado $\sigma(E)$, se obtiene reemplazando simultáneamente cada ocurrencia de X_i en la expresión E por el correspondiente término t_i . Se dice que $\sigma(E)$ es una instancia de E .

La relación “ser instancia de” induce un (pre)orden de máxima generalidad entre las expresiones del lenguaje de primer orden \mathcal{L} .

Definición 4.4 Sean E_1 y E_2 expresiones de \mathcal{L} . E_1 es más general que E_2 , denotado $E_1 \leq E_2$, si y solo si existe una sustitución σ tal que $\sigma(E_1) = E_2$.

Ejemplo 4.7

Sea la expresión $E \equiv p(X, Y, f(b))$ y la sustitución $\theta = \{Y/X, X/b\}$. La aplicación de θ a E es el término $\theta(E) = p(b, X, f(b))$. El término $p(b, X, f(b))$ es una instancia del término $p(X, Y, f(b))$. El término $p(X, Y, f(b))$ es más general que el término $p(b, X, f(b))$.

Observaciones 4.1

1. En programación lógica es habitual utilizar el convenio de representar la aplicación de una sustitución σ a una expresión E , mediante la notación $E\sigma$ en lugar de la más común $\sigma(E)$ utilizada en matemáticas. Sin embargo nosotros utilizaremos esta última con preferencia a la primera.
2. En ocasiones, cuando una sustitución se aplica a fórmulas generales de \mathcal{L} y no únicamente a expresiones de un lenguaje clausal, conviene renombrar las variables ligadas antes de aplicar la sustitución, con el fin de evitar la creación de nuevas ligaduras, que no existían antes de la aplicación de la sustitución (ver [75]).
3. La definición de instancia es compatible con la Definición 3.9 de instancia básica de una cláusula que se avanzó en el Apartado 3.5.1.



Como cualquier otra aplicación las sustituciones pueden componerse.

Definición 4.5 (Composición de Sustituciones) Dadas dos sustituciones σ y θ , la composición de σ y θ es la aplicación $\sigma \circ \theta$ tal que $(\sigma \circ \theta)(E) = \sigma(\theta(E))$.

La composición de sustituciones es asociativa (i.e., para toda sustitución ρ , σ y θ , se cumple que $(\rho \circ \sigma) \circ \theta = \rho \circ (\sigma \circ \theta)$) y la sustitución identidad es el elemento neutro (por la izquierda y por la derecha) con respecto a la composición de sustituciones (i.e., para toda sustitución θ , se cumple que $id \circ \theta = \theta \circ id = \theta$). El siguiente algoritmo muestra cómo hallar la composición de dos sustituciones, cuando se utiliza la notación conjuntista para su representación.

Algoritmo 4.1

Entrada: Dos sustituciones $\theta = \{X_1/t_1, \dots, X_n/t_n\}$ y $\sigma = \{Y_1/s_1, \dots, Y_m/s_m\}$.

Salida: La sustitución compuesta $\sigma \circ \theta$.

Comienzo**Computar:**

1. La sustitución $\theta_1 = \{X_1/\sigma(t_1), \dots, X_n/\sigma(t_n)\}$.
2. La sustitución θ_2 : eliminando de θ_1 los elementos $X_i/\sigma(t_i)$, con $X_i \in \sigma(t_i)$, que aparecen en θ_1 .
3. La sustitución σ_2 : eliminando de σ los elementos Y_i/s_i , con $Y_i \in \text{Dom}(\theta)$, que aparecen en σ .

Devolver $\sigma \circ \theta = \theta_2 \cup \sigma_2$.**Fin****Ejemplo 4.8**

Sean las sustituciones $\theta = \{X/f(Y), Y/Z\}$ y $\sigma = \{X/a, Y/b, Z/Y\}$. Aplicando el Algoritmo 1 obtenemos:

- (1) $\theta_1 = \{X/\sigma(f(Y)), Y/\sigma(Z)\} = \{X/f(b), Y/Y\}$
- (2) $\theta_2 = \{X/f(b)\}$
- (3) $\sigma_2 = \{Z/Y\}$

Así pues,

$$\sigma \circ \theta = \theta_2 \cup \sigma_2 = \{X/f(b), Z/Y\}.$$

Dado el término $t \equiv h(X, Y, Z)$, $(\sigma \circ \theta)(t) = h(f(b), Y, Y)$. Puede comprobarse que este resultado coincide con el obtenido al aplicar la Definición 4.5.

Observación 4.1

Cuando dos sustituciones σ y θ no comparten variables, i.e., $\text{Var}(\sigma) \cap \text{Var}(\theta) = \emptyset$, la composición de sustituciones se concreta en una operación de unión, i.e., $\sigma \circ \theta = \sigma \cup \theta$.

Los conceptos que se definen a continuación serán de utilidad en el futuro.

Definición 4.6 (Sustitución Idempotente) Una sustitución σ se dice idempotente si y solo si $\sigma \circ \sigma = \sigma$.

Ejemplo 4.9

Las sustituciones $\theta_1 = \{X/f(Z), Z/Y\}$ y $\theta_2 = \{X/a, Y/g(Y), Z/f(g(b))\}$ del Ejemplo 4.6 no son idempotentes. Puede comprobarse que $\theta_1 \circ \theta_1 = \{X/f(Y), Z/Y\}$ y $\theta_2 \circ \theta_2 = \{X/a, Y/g(g(Y)), Z/f(g(b))\}$. Nótese que, toda sustitución σ que satisface $\text{Dom}(\sigma) \cap \text{Var}(\text{Ran}(\sigma)) = \emptyset$ es idempotente [113]. Las sustituciones idempotentes tienen gran importancia debido a que el principio de resolución solamente computa este tipo de sustituciones.

Definición 4.7 (Renombramiento) Una sustitución ρ se denomina sustitución de renombramiento o, simplemente, renombramiento, si existe la sustitución inversa ρ^{-1} tal que $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = \text{id}$.

Definición 4.8 (Variante) Dadas dos expresiones E_1 y E_2 , decimos que son variantes si existen dos sustituciones de renombramiento σ y θ , tales que $E_1 = \sigma(E_2)$ y $E_2 = \theta(E_1)$.

Ejemplo 4.10

El término $t_1 \equiv p(f(X_1, X_2), g(X_3), a)$ es una variante de $t_2 \equiv p(f(X_2, X_1), g(X_4), a)$, pues las sustituciones $\sigma = \{X_1/X_2, X_2/X_1, X_4/X_3, X_3/X_4\}$ y $\theta = \{X_1/X_2, X_2/X_1, X_3/X_4, X_4/X_3\}$ llevan a $t_1 = \sigma(t_2)$ y $t_2 = \theta(t_1)$.

Al igual que la aplicación de sustituciones induce un (pre)orden sobre las expresiones y, en particular, sobre los términos (la relación de generalidad relativa, es decir, “ser instancia de” o “ser más general que”), la composición de sustituciones induce también un (pre)orden de generalidad relativa sobre las sustituciones.

Definición 4.9 Dadas dos sustituciones σ y θ , decimos que σ es más general que θ , denotado $\sigma \leq \theta$, si y solo si existe una sustitución λ tal que $\theta = \lambda \circ \sigma$.

Ejemplo 4.11

Sean las sustituciones $\sigma = \{X/a\}$ y $\theta = \{X/a, Y/b\}$. Se cumple que $\sigma \leq \theta$, ya que existe la sustitución $\lambda = \{Y/b\}$ tal que $\theta = \lambda \circ \sigma$.

Definición 4.10 Dadas dos sustituciones σ y θ , decimos que σ es equivalente a θ , denotado $\sigma \sim \theta$, si y solo si $\sigma \leq \theta$ y $\theta \leq \sigma$.

Nótese que las sustituciones que son variantes unas de otras son equivalentes (y viceversa).

Finalmente, introducimos las siguientes convenciones notacionales que serán de utilidad en lo que sigue: La restricción $\sigma|_V$ de una sustitución σ a un conjunto de variables $V \subseteq X$ se define como $\sigma|_V(x) = \sigma(x)$ si $x \in V$ y $\sigma|_V(x) = x$ si $x \notin V$. Dado un subconjunto de variables $W \subseteq X$, escribiremos $\sigma = \sigma'[W]$ si $\sigma|_W = \sigma'|_W$ y escribiremos $\sigma \leq \sigma'[W]$ si existe una sustitución σ'' tal que $\sigma' = (\sigma'' \circ \sigma)|_W$.

4.3 UNIFICACIÓN

La unificación de expresiones es un concepto fundamental tanto para el campo de la demostración automática como para el de la programación lógica. Informalmente, *unificar* es el proceso por el cual dos o más expresiones se convierten en idénticas mediante la aplicación de una sustitución, denominada unificadora. Como veremos, este tipo de

sustituciones sintetiza la noción del cómputo en el contexto de la programación lógica.

Definición 4.11 (Unificador)

Una sustitución θ es un unificador del conjunto de expresiones $\{E_1, E_2, \dots, E_k\}$ si y solo si $\theta(E_1) = \theta(E_2) = \dots = \theta(E_k)$. Se dice que el conjunto E_1, E_2, \dots, E_k es unificable si existe un unificador para él.

Ejemplo 4.12

La sustitución $\{X/a, Y/b\}$ es un unificador del conjunto de átomos $\{p(X, b), p(a, Y)\}$.

Definición 4.12 (Unificador Más General (u.m.g.)) Un unificador σ de un conjunto de expresiones S es un unificador más general (o de máxima generalidad) para S si y solo si cualquier unificador θ de S cumple que $\sigma \leq \theta$.

Ejemplo 4.13

Sea el conjunto de expresiones $S = \{p(f(Y), Z), p(X, a)\}$. Tanto la sustitución $\{X/f(a), Z/a, Y/a\}$ como la sustitución $\{X/f(Y), Z/a, W/a\}$ y la sustitución $\{X/f(Y), Z/a\}$ son unificadores del conjunto S , solo la última es un u.m.g. de S .

Existen distintos algoritmos para unificar un conjunto de expresiones (pueden verse algunos de ellos en [6, 26, 104] y en [126]). Aquí damos uno debido a Martelli y Montanari, que se basa en la siguiente observación: la tarea de hallar el u.m.g. de dos expresiones $E_1 \equiv F(t_1, \dots, t_n)$ y $E_2 \equiv F(s_1, \dots, s_n)$ es equivalente a solucionar el conjunto de ecuaciones $G = \{t_1 = s_1, \dots, t_n = s_n\}$, donde el símbolo “=” de las ecuaciones representa la posibilidad de obtener la igualdad sintáctica entre los argumentos de E_1 y E_2 . En el *algoritmo de unificación sintáctica* de Martelli y Montanari, la unificación de las expresiones se logra mediante una secuencia de transformaciones en la que se parte de un estado inicial $\langle G, id \rangle$, donde $G = \{t_1 = s_1, \dots, t_n = s_n\}$ es un conjunto de ecuaciones e id es la sustitución identidad, y se hace uso de la relación de unificación “ \Rightarrow ” formulada más abajo para realizar cada paso de transformación:

$$\langle G, id \rangle \Rightarrow \langle G1, \theta_1 \rangle \Rightarrow \langle G2, \theta_2 \rangle \Rightarrow \dots \Rightarrow \langle Gn, \theta_n \rangle.$$

Cuando alcanzamos un estado $\langle Gn, \theta_n \rangle \equiv \langle \emptyset, \theta \rangle$, en el que el conjunto inicial G se ha reducido al conjunto vacío, diremos que las expresiones E_1 y E_2 son unificables con u.m.g. θ , que denotamos por $mgu(E_1, E_2)$. El estado $\langle \emptyset, \theta \rangle$ simboliza el éxito en el proceso de unificación. En el caso de que las expresiones E_1 y E_2 no sean unificables, el algoritmo terminará con un estado de fallo $\langle Gn, \theta_n \rangle \equiv \langle Fallo, \theta \rangle$.

En la próxima definición damos las *reglas de Martelli y Montanari* para la unificación de un conjunto de ecuaciones.

Definición 4.13 (Relación de unificación “ \Rightarrow ”) La relación de unificación “ \Rightarrow ” es la mínima relación definida por el siguiente conjunto de reglas de transición:

1. DESCOMPOSICIÓN EN TÉRMINOS:

$$\frac{\langle \{op(t_1, \dots, t_n) = op(s_1, \dots, s_n)\} \cup E, \theta \rangle}{\langle \{t_1 = s_1, \dots, t_n = s_n\} \cup E, \theta \rangle}$$

donde op puede ser un símbolo de función o de predicado.

2. ELIMINACIÓN DE ECUACIONES TRIVIALES:

$$\frac{\langle \{X = X\} \cup E, \theta \rangle}{\langle E, \theta \rangle}$$

3. ORDENACIÓN DE PARES:

$$\frac{\langle \{t = X\} \cup E, \theta \rangle}{\langle \{X = t\} \cup E, \theta \rangle} \text{ si } t \text{ no es una variable.}$$

4. ELIMINACIÓN DE VARIABLE:

$$\frac{\langle \{X = t\} \cup E, \theta \rangle}{\langle \{X/t\}(E), \{X/t\} \circ \theta \rangle} \text{ si la variable } X \text{ no aparece en } t.$$

5. REGLA DE FALLO:

$$\frac{\langle \{op_1(t_1, \dots, t_n) = op_2(s_1, \dots, s_m)\} \cup E, \theta \rangle}{\langle Fallo, \theta \rangle} \text{ si } op_1 \neq op_2$$

donde op_1 y op_2 pueden ser símbolos de función o de predicado.

6. OCURRENCIA DE VARIABLE (occur check) :

$$\frac{\langle \{X = t\} \cup E, \theta \rangle}{\langle Fallo, \theta \rangle} \text{ si la variable } X \text{ aparece en } t \text{ y } X \not\equiv t.$$

Observación 4.2

Las reglas de la Definición 4.13 forman parte de un sistema de transición de estados. Cada una de ellas puede entenderse, y se aplica, de manera similar a las reglas de inferencia de la lógica, salvo que el estado que se muestra en la parte de la condición hace referencia al último de la secuencia (y no a cualquier otro que aparezca previamente). También pueden entenderse como reglas que, a partir de un estado, generan el estado sucesor. Nótese también que, en el caso en que $n=0$, la regla 1 incluye el caso $c=c$ para cada constante c , lo que conduce al borrado de tal ecuación. Por otro lado, la regla 2

incluye el caso de dos constantes diferentes y los casos en que se compara una constante con un término que no es una constante ni una variable.

Ejemplo 4.14

Para hallar el u.m.g. de las expresiones $E_1 \equiv p(a, X, f(g(Y)))$ y $E_2 \equiv p(Z, f(Z), f(U))$, procedemos como sigue:

1. Construimos el estado o configuración inicial:

$$\langle \{a = Z, X = f(Z), f(g(Y)) = f(U)\}, id \rangle$$

2. Aplicamos las reglas de unificación hasta alcanzar una configuración terminal (de éxito o de fallo):

$$\begin{aligned} & \langle \{a = Z, X = f(Z), f(g(Y)) = f(U)\}, id \rangle \Rightarrow_3 \\ & \langle \{Z = a, X = f(Z), f(g(Y)) = f(U)\}, id \rangle \Rightarrow_4 \\ & \langle \{X = f(a), f(g(Y)) = f(U)\}, \{Z/a\} \rangle \Rightarrow_4 \\ & \langle \{f(g(Y)) = f(U)\}, \{Z/a, X/f(a)\} \rangle \Rightarrow_1 \\ & \langle \{g(Y) = U\}, \{Z/a, X/f(a)\} \rangle \Rightarrow_3 \\ & \langle \{U = g(Y)\}, \{Z/a, X/f(a)\} \rangle \Rightarrow_4 \\ & \langle \emptyset, \{Z/a, X/f(a), U/g(Y)\} \rangle \end{aligned}$$

donde los subíndices que acompañan el símbolo de relación “ \Rightarrow ” hacen referencia a la regla empleada en cada paso.

Se ha alcanzado un estado de éxito por lo que puede afirmarse que $mgu(E_1, E_2) = \{Z/a, X/f(a), U/g(Y)\}$.

Teorema 4.3 (Teorema de Unificación) *Dadas dos expresiones E_1 y E_2 unificables, la secuencia de transformaciones a partir del estado inicial*

$$\langle G, id \rangle \Rightarrow \langle G1, \theta_1 \rangle \Rightarrow \langle G2, \theta_2 \rangle \Rightarrow \dots \Rightarrow \langle \emptyset, \theta \rangle$$

termina obteniendo el u.m.g. θ de las expresiones E_1 y E_2 , que es único salvo renombramientos de variables.

4.4 EL PRINCIPIO DE RESOLUCIÓN EN LA LÓGICA DE PREDICADOS

Una vez definidos los conceptos de unificador y unificador más general, e introducido el algoritmo de unificación, podemos formular el principio de resolución para la

lógica de predicados. Comenzamos definiendo el concepto de *factor de una cláusula*, que nos permite obtener una versión simplificada de la misma.

La siguiente operación de factorización se fundamenta en la propiedad de idempotencia del disyuntor.

Definición 4.14 (Factor de una cláusula) Si dos o más literales (con el mismo signo) de una cláusula C tienen un unificador de máxima generalidad σ , entonces se dice que $\sigma(C)$ es un factor de C . Si $\sigma(C)$ es una cláusula unitaria, entonces se dice que es un factor unitario de C .

Ejemplo 4.15

Sea $C \equiv p(X) \vee p(f(Y)) \vee \neg q(X)$. Entonces el primer y segundo literal tienen un u.m.g. $\sigma = \{X/f(Y)\}$. Por tanto, $\sigma(C) = p(f(Y)) \vee \neg q(f(Y))$ es un factor de C .

Definición 4.15 (Resolvente binario) Sean C_1 y C_2 dos cláusulas (llamadas cláusulas padres) sin variables en común. Sean L_1 y L_2 dos literales de C_1 y C_2 , respectivamente. Si L_1 y $\neg L_2$ tienen un u.m.g. σ , entonces la cláusula $(\sigma(C_1) \setminus \sigma(L_1)) \cup (\sigma(C_2) \setminus \sigma(L_2))$ se conoce como resolvente binario de C_1 y C_2 . A los literales L_1 y L_2 se les llama literales resueltos.

Ejemplo 4.16

Sea $C_1 \equiv p(X) \vee q(X)$ y $C_2 \equiv \neg p(a) \vee r(X)$. Como X aparece tanto en C_1 como en C_2 , renombramos la variable de C_2 y hacemos $C_2 \equiv \neg p(a) \vee r(Y)$. Escogemos $L_1 \equiv p(X)$ y $L_2 \equiv \neg p(a)$, de modo que el u.m.g. de L_1 y $\neg L_2$ es $\sigma = \{X/a\}$. Así, pues:

$$\begin{aligned} &(\sigma(C_1) \setminus \sigma(L_1)) \cup (\sigma(C_2) \setminus \sigma(L_2)) \\ &= (\{p(a), q(a)\} \setminus \{p(a)\}) \cup (\{\neg p(a), r(Y)\} \setminus \{\neg p(a)\}) \\ &= \{q(a)\} \cup \{r(Y)\} \\ &= \{q(a), r(Y)\} \end{aligned}$$

Por lo tanto $q(a) \vee r(Y)$ es un resolvente binario de C_1 y C_2 , mientras que $p(X)$ y $\neg p(a)$ son los literales resueltos en este caso. La Definición 4.15 puede expresarse, más formalmente, en términos de la siguiente regla de inferencia:

$$\frac{C_1, C_2}{(\sigma(C_1) \setminus \sigma(L_1)) \vee (\sigma(C_2) \setminus \sigma(L_2))} \text{ si } L_1 \in C_1, L_2 \in C_2, \sigma = \text{mgu}(L_1, \neg L_2)$$

Definición 4.16 (Resolvente) Un resolvente de las cláusulas (padres) C_1 y C_2 es uno de los siguientes resolventes binarios:

1. Un resolvente binario de C_1 y C_2 ,
2. Un resolvente binario de C_1 y un factor de C_2 ,
3. Un resolvente binario de un factor de C_1 y C_2 ,
4. Un resolvente binario de un factor de C_1 y un factor de C_2 .

Ejemplo 4.17

Sea $C_1 \equiv p(X) \vee p(f(Y)) \vee r(g(Y))$ y $C_2 \equiv \neg p(f(g(a))) \vee q(b)$, un factor de C_1 es la cláusula $C'_1 \equiv p(f(Y)) \vee r(g(Y))$. Un resolvente binario de C'_1 y C_2 es $r(g(g(a))) \vee q(b)$. Por tanto $r(g(g(a))) \vee q(b)$ es un resolvente de C_1 y C_2

Observaciones 4.2

1. A la hora de aplicar un paso de resolución, un hecho importante es la existencia de un algoritmo de unificación que permite obtener el u.m.g. de un conjunto de expresiones unificables, o de notificar de su inexistencia en caso contrario. Si no existe el u.m.g. de los dos literales, pertenecientes a las cláusulas padres y candidatos a ser resueltos, entonces no podrá darse el paso de resolución.
2. Dadas dos cláusulas, la unificación de dos de sus literales puede verse como un método que calcula la particularización más adecuada que hace posible el empleo de la regla de resolución.



Naturalmente, en el contexto de la lógica de predicados también se cumple que un resolvente C de dos cláusulas C_1 y C_2 es una consecuencia lógica de C_1 y C_2 , lo cual es indispensable para establecer la corrección del principio de resolución. Pero el resultado más importante sobre el principio de resolución es su completitud. Ambos resultados quedan reflejados en el siguiente teorema.

Teorema 4.4

Sea Δ un conjunto de cláusulas.

1. (Corrección) Si existe una deducción de la cláusula vacía, \square , a partir de Δ utilizando la regla de resolución, entonces Δ es insatisfacible.
2. (Completitud) Si Δ es insatisfacible entonces, existe una deducción de la cláusula vacía, \square , a partir de Δ utilizando la regla de resolución.

Para terminar este apartado veamos algunos ejemplos que ilustran el uso del principio de resolución.

Ejemplo 4.18

Sea el conjunto de cláusulas $\Delta = \{\neg p(f(X)) \vee s(Y), \neg s(X) \vee r(X, b), p(X) \vee s(X)\}$. Para probar por refutación que $(\exists X)(\exists Y)r(X, Y)$ es una consecuencia lógica de Δ , negamos dicha fórmula, la pasamos a forma clausal y obtenemos la siguiente prueba aplicando la regla de resolución:

(1)	$\neg p(f(X)) \vee s(Y)$	
(2)	$\neg s(X) \vee r(X, b)$	
(3)	$p(X) \vee s(X)$	
(4)	$\neg r(X, Y)$	negación de la fórmula considerada
(5)	$s(f(X)) \vee s(Y)$	resolvente de (1) y (3)
(6)	$r(f(X), b)$	resolvente de (2) y factor $s(f(X))$ de (5)
(7)	\square	resolvente de (4) y (6)

Puede comprobarse que:

- en la resolución de (1) y (3), el u.m.g. es $mgu(p(f(X)), p(X_1)) = \{X_1/f(X)\}$
- en la resolución de (2) y el factor de (5) el u.m.g. es $mgu(s(X_2), s(f(X))) = \{X_2/f(X)\}$
- en la resolución de (4) y (6), es $mgu(r(X, Y), r(f(X_3), b)) = \{X/f(X_3), Y/b\}$.

En cada paso de resolución se han renombrado sistemáticamente las variables de las cláusulas para evitar colisiones.

Ejemplo 4.19

Sea el conjunto de cláusulas $\Delta = \{\neg s(X, Y) \vee \neg p(Y) \vee r(f(X)), \neg r(Z), s(a, b), p(b)\}$. Para probar que el conjunto de cláusulas Δ es insatisfacible procedemos del siguiente modo:

(1)	$\neg s(X, Y) \vee \neg p(Y) \vee r(f(X))$	
(2)	$\neg r(Z)$	
(3)	$s(a, b)$	
(4)	$p(b)$	
(5)	$\neg s(X, Y) \vee \neg p(Y)$	resolvente de (1) y (2)
(6)	$\neg p(b)$	resolvente de (3) y (5)
(7)	\square	resolvente de (4) y (6)

Puede comprobarse que:

- en la resolución de (1) y (2), el u.m.g. es $mgu(r(f(X)), r(Z)) = \{Z/f(X)\}$

- en la resolución de (3) y (5) el u.m.g. es $mgu(s(a, b), s(X, Y)) = \{X/a, Y/b\}$
- en la resolución de (4) y (6), es la sustitución identidad, *id*.

4.5 ESTRATEGIAS DE RESOLUCIÓN

El Teorema 4.4 nos dice que si un conjunto de cláusulas es insatisfacible entonces a partir de él podrá deducirse la cláusula vacía, ahora bien, no nos dice como encontrarla. El siguiente algoritmo genérico es un esquema que permitirá después sistematizar la búsqueda de la cláusula vacía.

Algoritmo 4.2 [Algoritmo de Resolución]

Entrada: Un conjunto de cláusulas Δ .

Salida: Una condición de **éxito**.

Comienzo

Inicialización: $CLAUSULAS = \Delta$.

Repetir

1. Seleccionar dos cláusulas distintas, C_i y C_j , del conjunto $CLAUSULAS$, que sean resolubles;
2. Hallar el resolvente R_{ij} de C_i y C_j ;
3. $CLAUSULAS = CLAUSULAS \cup \{R_{ij}\}$;

Hasta que \square **aparezca** en $CLAUSULAS$.

Devolver éxito.

Fin

Este algoritmo considera el principio de resolución como una regla de inferencia que puede emplearse para saturar el conjunto inicial generando nuevas cláusulas a partir de las antiguas sin ninguna restricción. Tal y como se ha formulado, en el Algoritmo 2 no se da preferencia a una derivación particular, que en este contexto representa uno de los posibles caminos para alcanzar la cláusula vacía. En general, habrá que generar todas las secuencias de resultantes que pueden formarse considerando las formas diferentes en las que se pueden resolver las cláusulas del conjunto $CLAUSULAS$, si queremos preservar la completitud del método (es decir, si queremos asegurar que, en presencia de un conjunto de cláusulas insatisfacible, el Algoritmo 2 es capaz de encontrar la cláusula vacía \square). Como se muestra en [26], una aplicación sin restricciones del principio de resolución puede generar cláusulas que son redundantes o irrelevantes para los objetivos de la prueba, frente a otras que son útiles. Este comentario conduce a la discusión de los refinamientos del proceso de deducción.

El Algoritmo 2 posee dos grados de libertad:

1. En el punto (1), el proceso de seleccionar dos cláusulas no está especificado. Esto es, ¿qué cláusulas seleccionar para realizar la resolución?

2. En el punto (2), tampoco decimos qué literal $L_i \in C_i$ y qué literal $L_j \in C_j$ elegimos para realizar el paso de resolución que conduce al resolvente \mathcal{R}_{ij} .

Fijar estos grados de libertad da lugar a diferentes estrategias de resolución, en las que lograr eficiencia en el número de resoluciones empleadas y en la cantidad de almacenamiento empleado son los objetivos principales. En esta sección vamos a considerar una *estrategia de resolución* como una concreción o instancia del Algoritmo 2 que eventualmente reduce el número de derivaciones a generar. Es importante que una estrategia continúe siendo completa, en el sentido especificado más arriba. Esto es, que obtenga una refutación siempre que Δ sea insatisfacible.

En los próximos subapartados resumimos las características más destacadas de algunas de las principales estrategias de resolución.

4.5.1. Estrategia de resolución por saturación de niveles

La *estrategia de resolución por saturación de niveles* esencialmente es una estrategia de búsqueda a ciegas¹. Intuitivamente, para encontrar una refutación a partir de un conjunto de cláusulas inicial Δ , se generan todos los resolventes de pares de cláusulas del conjunto Δ y se añaden estos resolventes al conjunto Δ . El proceso se repite en cada iteración, generando los resolventes del nivel n a partir de los del nivel $n - 1$, hasta encontrar la cláusula vacía \square . Así pues, esta estrategia construye una colección $\Delta^0, \Delta^1, \dots$ de conjuntos de cláusulas², donde

$$\Delta^0 = \Delta$$

$$\Delta^n = \{\text{resolventes de } C_i \text{ y } C_j \mid C_i \in (\Delta^0, \dots, \Delta^{n-1}) \wedge C_j \in \Delta^{n-1} \wedge C_i < C_j\}$$

Ahora, en la i -ésima iteración del Algoritmo 2, los pasos (1) y (2) se sustituyen por la tarea de computar el conjunto Δ^i que debe incorporarse a *CLAUSULAS* en el paso 3 (i.e., $CLAUSULAS = CLAUSULAS \cup \Delta^i$).

La estrategia de resolución por saturación de niveles se denomina estrategia *en anchura* (o *en amplitud*) en [104]. La estrategia de resolución por saturación de niveles es completa pero sumamente ineficiente [26].

4.5.2. Estrategia de borrado

La *estrategia de borrado* se basa en la estrategia de resolución por saturación de niveles a la que se incorporan procedimientos para comprobar si los resolventes \mathcal{R}_{ij} de C_i y C_j son una tautología o están subsumidos por otra cláusula ya contenida en el conjunto *CLAUSULAS*.

¹Es decir, sin el empleo de información específica sobre el problema, contrariamente a lo que sucede con las estrategias de búsqueda heurísticas.

²En la práctica los conjuntos Δ^k , con $k = 1, 2, \dots$, son tratados como secuencias de cláusulas por lo que cobra sentido hablar de que una cláusula C_i precede a otra C_j , denotado $C_i < C_j$, cuando la cláusula C_i aparece en una posición i menor que la j dentro de la secuencia.

Definición 4.17 Una cláusula C subsume a una cláusula C' si y solamente si existe una sustitución σ tal que $C' \subseteq \sigma(C)$. Se dice que C' es la cláusula subsumida.

La intuición detrás del concepto de subsumpción es que si una cláusula C subsume a una cláusula C' , entonces la cláusula C' es consecuencia lógica de C . Por tanto, cualquier consecuencia que pueda derivarse de C' puede derivarse, también, a partir de C . Esto es, la cláusula C hace “innecesaria” a la cláusula C' , que puede eliminarse sin merma de poder deductivo.

Ejemplo 4.20

Sean $C \equiv p(X)$ y $C' \equiv p(a) \vee q(a)$. C' está subsumida por C porque existe una sustitución $\sigma = \{X/a\}$ tal que $\sigma(C) = p(a) \subseteq C'$. Es fácil probar que C' es consecuencia lógica de C .

En la estrategia de borrado, si $\mathcal{R}_{ij} \in \Delta^n$ es una tautología o es una cláusula subsumida por una cláusula en $CLAUSULAS$, entonces se elimina del conjunto Δ^n .

La estrategia de borrado, así definida, es completa. Cuando la estrategia de borrado no se define en combinación con la estrategia de resolución por saturación de niveles y se plantea, en términos más generales, como una estrategia que elimina cualquier tautología o cláusula subsumida que pueda aparecer en el proceso de resolución, la completitud del método no está garantizada.

4.5.3. Estrategia de resolución semántica

Esencialmente la estrategia de resolución semántica intenta limitar el número de resoluciones dividiendo en dos subconjuntos disjuntos el conjunto $CLAUSULAS$ cuya insatisfacibilidad se desea probar. La idea es que no puedan resolverse entre sí las cláusulas que pertenecen a un mismo subconjunto.

Con más precisión, la estrategia de resolución semántica se comporta de siguiente modo:

1. Divide el conjunto $CLAUSULAS$ en dos, utilizando una interpretación I :

$$\begin{aligned}\Delta_1 &= \{C \in CLAUSULAS \mid I \text{ es modelo de } C\} \\ \Delta_2 &= \{C \in CLAUSULAS \mid I \text{ no es modelo de } C\}\end{aligned}$$

2. Halla el conjunto de todos los resolventes que pueden formarse a partir de las cláusulas de Δ_1 y Δ_2 :

$$\Delta' = \{\mathcal{R}_{ij} \mid \mathcal{R}_{ij} \text{ es un resolvente de } C_i \in \Delta_1 \text{ y } C_j \in \Delta_2\}$$

3. $CLAUSULAS = CLAUSULAS \cup \Delta'$

Nótese que si el conjunto de cláusulas de partida es insatisfacible, cualquier interpretación lo divide en dos subconjuntos disjuntos, ya que habrá, al menos, una cláusula que será falsa en dicha interpretación. La estrategia de resolución semántica es completa.

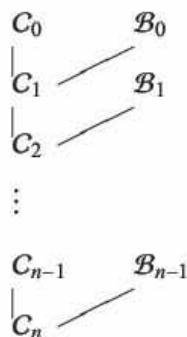
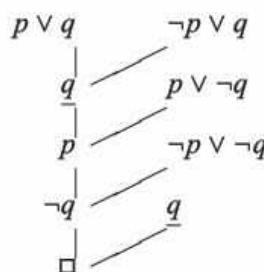
Un caso particular de la estrategia de resolución semántica es la denominada *estrategia del conjunto de soporte*, introducida por Wos, Robinson y Carson en 1965. Como hemos repetido en varias ocasiones, si queremos comprobar que una cierta cláusula C es deducible a partir un conjunto de cláusulas Δ , basta con comprobar que del conjunto $\Delta \cup \{\neg C\}$ puede derivarse la cláusula vacía, \square . Dado que en este tipo de problemas Δ se supone satisfacible, es imposible que al resolver cláusulas de Δ entre sí se deduzca la cláusula vacía. Así pues, una forma razonable de dividir el conjunto $\Delta \cup \{\neg C\}$ es considerar por un lado el conjunto Δ y por otro el conjunto $\{\neg C\}$, ya que es mejor no permitir resoluciones entre las cláusulas de Δ . De forma más general, un subconjunto $\Delta' \subseteq \Delta$ se denomina *conjunto de soporte* de Δ si el conjunto diferencia $\Delta \setminus \Delta'$ es satisfacible. La estrategia del conjunto de soporte impide el cómputo de resolventes cuyas cláusulas padres pertenezcan, ambas, al subconjunto satisfacible $\Delta \setminus \Delta'$. La estrategia del conjunto de soporte es completa.

4.5.4. Estrategia de resolución lineal

Cuando se intenta demostrar una identidad, normalmente se comienza aplicando una regla de inferencia al término izquierdo de la identidad para obtener una nueva expresión, a la que se le vuelve a aplicar una nueva regla de inferencia, y así sucesivamente hasta que se obtiene una expresión final que coincide con la expresión que aparece en el término derecho de la identidad original. La idea de resolución lineal es similar a esta forma de razonamiento encadenado. Comenzando con una cláusula, se resuelve ésta contra otra cláusula para obtener un resolvente que, a su vez, será resuelto con otra cláusula, y el proceso se repite las veces que sea necesario hasta obtener finalmente la cláusula vacía, \square . Esta simple estrategia impone una fuerte restricción al número de resoluciones propiciadas por el Algoritmo 2 y, además, todavía es completa. La resolución lineal fue inicialmente propuesta por Loveland y Luckham en 1970, y posteriormente refinada por Anderson y Bledsoe, Yates, Reiter, Kowalski, Kuehner y el propio Loveland. En este apartado, solo consideraremos la versión dada por Kowalski y Kuehner (1971) y Loveland (1972) debido a su adecuación para ser implementada en un computador digital.

Definición 4.18 (Deducción Lineal) *Dado un conjunto de cláusulas Δ y una cláusula C_0 de Δ , una deducción lineal de C_n a partir de Δ con cláusula inicial C_0 es una deducción de la forma mostrada en la Figura 4.2 donde:*

1. Para $i = 0, 1, \dots, n - 1$, C_{i+1} es un resolvente de C_i (llamada cláusula central) y B_i (llamada cláusula lateral), y
2. Cada B_i pertenece a Δ , o es una cláusula central C_j para algún j , con $j < i$.

**Figura 4.2** Árbol de deducción para una deducción lineal genérica.**Figura 4.3** Árbol de deducción para la refutación lineal del Ejemplo 4.21.**Ejemplo 4.21**

Sea $\Delta = \{(p \vee q), (\neg p \vee q), (p \vee \neg q), (\neg p \vee \neg q)\}$. La Figura 4.3 muestra una deducción lineal de \square a partir de Δ con cláusula inicial $(p \vee q)$. En la Figura 4.3 se observan cuatro cláusulas laterales. Tres de ellas pertenecen a Δ , mientras una de ellas, la cláusula “ q ”, que aparece subrayada, es un resolvente derivado antes de que \square fuese deducida.

Ejemplo 4.22

Consideremos el conjunto de cláusulas

$$\Delta = \{m(a, s(c), s(b)), p(a), m(X, X, s(X)), \\ (\neg m(X, Y, Z) \vee m(Y, X, Z)), (\neg m(X, Y, Z) \vee D(X, Z)), \\ (\neg p(X) \vee \neg m(Y, Z, U) \vee \neg d(X, U) \vee D(X, Y) \vee D(X, Z)), \\ \neg d(a, b)\}.$$

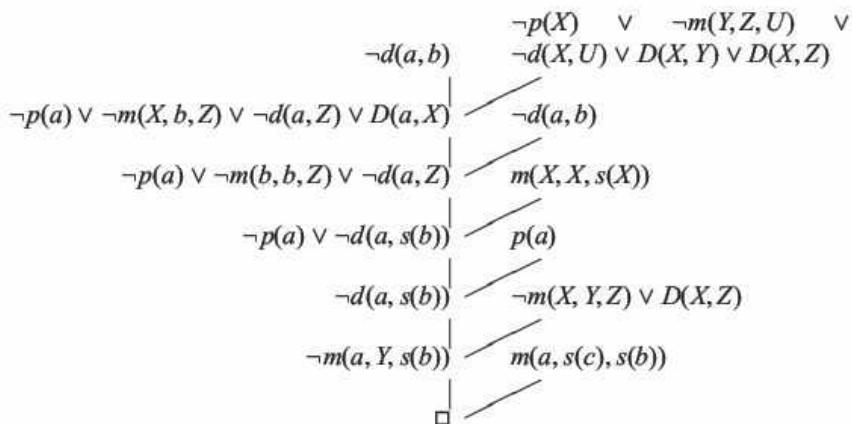


Figura 4.4 Árbol de deducción para la refutación lineal del Ejemplo 4.22.

La Figura 4.4 es una deducción o derivación lineal de \square a partir de Δ .

4.5.5. Estrategia de resolución lineal de entrada y de preferencia por cláusulas unitarias

Como hemos dicho, cuando consideramos un refinamiento del método de resolución, nos interesa que no se pierda el carácter de completitud, es decir, queremos garantizar que la cláusula vacía siempre se pueda derivar de un conjunto de cláusulas insatisfacible. Sin embargo, la eficiencia también es importante en el contexto de la demostración automática de teoremas. En ocasiones, puede interesarnos encontrar un método de resolución eficiente a pesar de que no sea completo. Esto hizo que algunos investigadores idearan métodos de resolución eficientes pero incompletos, como son la resolución lineal de entrada³ y la resolución de preferencia por cláusulas unitarias.

En la *estrategia de resolución lineal de entrada (input resolution)* una de las cláusulas padres seleccionada tiene que pertenecer al conjunto de cláusulas inicial Δ , mientras que en la *estrategia de resolución de preferencia por cláusulas unitarias*, una de las cláusulas padres tiene que ser unitaria. Estas dos estrategias son equivalentes, en el sentido de que existe una refutación unitaria para el conjunto de cláusulas Δ si y solo si existe una refutación de entrada para Δ [26].

En el siguiente capítulo estudiaremos un método que, basándose en la estrategia de resolución lineal (de entrada), intentará mejorarla para permitir implementaciones más eficientes. El precio que pagaremos por aumentar la eficiencia en la implementación,

³Sin embargo, esta estrategia sí es completa cuando nos restringimos a un tipo particular de cláusulas, las denominadas cláusulas de Horn (véase próximo capítulo).

será perder generalidad sobre el tipo de cláusulas que es capaz de manipular el método sin perder la completitud. Este método recibe el nombre de resolución lineal SLD y constituye el mecanismo operacional de la programación lógica.

RESUMEN

En este capítulo hemos continuado nuestro recorrido por algunos de los tópicos de la demostración automática de teoremas que sirven de fundamentación a la programación lógica, centrándonos en los métodos de prueba sintácticos basados en el principio de resolución de Robinson.

- Para evitar las ineficiencias de las implementaciones directas del teorema de Herbrand, producidas por la generación sistemática de conjuntos de cláusulas básicas y la posterior comprobación de su insatisfacibilidad, se introdujo el *principio de resolución de Robinson* (1965), que permite deducir consecuencias universales de enunciados universales.
- El principio de resolución es una regla de inferencia que se aplica a fórmulas en forma clausal y que, junto con el procedimiento de unificación, constituye un cálculo deductivo completo.
- Una sustitución unificadora de un conjunto de expresiones es aquélla que, aplicada sobre cada una de las expresiones del conjunto, las hace sintácticamente iguales. De entre todos los unificadores posibles nos interesa obtener el unificador más general. Existen distintos algoritmos para unificar un conjunto de expresiones y nosotros estudiamos uno de los más populares, debido a Martelli y Montanari.
- El algoritmo de unificación de Martelli y Montanari halla el unificador más general θ de dos expresiones $E_1 \equiv F(t_1, \dots, t_n)$ y $E_2 \equiv F(s_1, \dots, s_n)$ solucionando el conjunto de ecuaciones $G = \{t_1 = s_1, \dots, t_n = s_n\}$ mediante una secuencia de transformaciones que permiten pasar de un estado inicial $\langle G, id \rangle$, donde id es la sustitución identidad, a un estado final $\langle \emptyset, \theta \rangle$ de éxito, donde la sustitución θ es el unificador más general para este problema. El algoritmo de Martelli y Montanari siempre termina, con éxito o con una configuración de fallo. Cuando termina con éxito encuentra un unificador más general que es único salvo renombramiento de variables. Denotamos el unificador más general de dos expresiones E_1 y E_2 como $mgu(E_1, E_2)$.
- La regla de resolución se formaliza como sigue: Dadas dos cláusulas C_1 y C_2 (llamadas cláusulas padres) **sin variables en común**

$$\frac{C_1, C_2}{(\sigma(C_1) \setminus \sigma(L_1)) \vee (\sigma(C_2) \setminus \sigma(L_2))} \text{ si } \begin{cases} L_1 \in C_1, L_2 \in C_2, \\ \sigma = mgu(L_1, \neg L_2) \neq \text{fallo} \end{cases}$$

La cláusula $(\sigma(C_1) \setminus \sigma(L_1)) \vee (\sigma(C_2) \setminus \sigma(L_2))$ se conoce como resolvente binario de C_1 y C_2 . A los literales L_1 y L_2 se les llama literales resueltos.

- El Principio de resolución de Robinson es una regla de inferencia refutacionalmente correcta y completa. Esto es, dado un conjunto de cláusulas Δ , éste es insatisfacible si y solo si existe una deducción de la cláusula vacía, \square , a partir de Δ .
- Como se muestra en [26], una aplicación sin restricciones del principio de resolución puede generar cláusulas que son redundantes o irrelevantes para los objetivos de la prueba, dando lugar a un espacio de búsqueda muy amplio. Se han diseñado diferentes estrategias de resolución para disminuir el espacio de búsqueda. Desde nuestra perspectiva la más importante es la estrategia de resolución lineal. Esta estrategia impone una fuerte restricción al número de resoluciones propiciadas por la aplicación indiscriminada de la regla de resolución y, además, todavía es completa.

CUESTIONES Y EJERCICIOS

Cuestión 4.1 Enuncie el Principio de Resolución de Robinson para la lógica de proposiciones.

Ejercicio 4.2 Sea el conjunto de cláusulas $\Delta = \{p \vee q, \neg q \vee r, \neg p \vee q, \neg r\}$. Muestre que Δ es insatisfacible empleando resolución.

Ejercicio 4.3 Pruebe la insatisfacibilidad, empleando resolución, del conjunto de cláusulas $\{\neg p \vee r, \neg q, \neg r, p \vee q \vee r\}$. Dibuje el árbol de deducción asociado a la refutación obtenida.

Ejercicio 4.4 Pruebe que $(\neg q \rightarrow \neg p)$ es una consecuencia lógica de $(p \rightarrow q)$ empleando resolución.

Cuestión 4.5 Defina los siguientes conceptos: sustitución; relación de orden de generalidad relativa, instancia de una expresión y composición de sustituciones.

Ejercicio 4.6 Sea $\theta \equiv \{X/a, Y/b, Z/g(X, Y)\}$ una sustitución y $E \equiv p(h(X), Z)$ una fórmula atómica de \mathcal{L} . Calcule $\theta(E)$.

Cuestión 4.7 Una sustitución es idempotente si $\theta \circ \theta = \theta$. Indique cuál de las siguientes sustituciones no es idempotente:

1. $\{X/f(a)\}$.
2. $\{X/Y, Z/a\}$.
3. $\{X/Y, Y/a\}$.
4. $\{X/Y\}$.

Ejercicio 4.8 Considere las sustituciones $\sigma = \{X/h(b)\}$ y $\theta = \{Z/X, X/g(a)\}$. Calcule el resultado de componer: a) $\theta \circ \sigma$; b) $\sigma \circ \theta$.

Ejercicio 4.9 Sean $\sigma = \{X/a, Y/f(Z), Z/Y\}$ y $\theta = \{X/b, Y/Z, Z/g(X)\}$ sustituciones. Halle la composición de σ y θ .

Ejercicio 4.10 Demuestre que la composición de sustituciones es asociativa pero no conmutativa.

Cuestión 4.11 Defina con precisión los conceptos de unificador y unificador más general.

Cuestión 4.12 Describa el algoritmo de unificación de Martelli-Montanari.

Ejercicio 4.13 Determine si cada uno de los siguientes conjuntos es unificable y obtener un unificador más general.

1. $\{q(a), q(b)\}$
2. $\{q(a, X), q(a, a)\}$
3. $\{q(a, X, f(X)), q(a, Y, Y)\}$
4. $\{q(X, Y, Z), q(U, h(v, v), U)\}$
5. $\{p(X_1, g(X_1), X_2, h(X_1, X_2), X_3, k(X_1, X_2, X_3)), p(Y_1, Y_2, e(Y_2), Y_3, f(Y_2, Y_3), Y_4)\}.$
6. $\{p(f(X, X), g(Y), Z), p(f(a, V), g(b), h(W))\};$
7. $\{p(X, f(a)), p(Y, Y)\};$
8. $\{p(g(X), Y), p(g(V), h(a)), p(Z, h(Y))\};$
9. $\{p(g(b), Z), p(Y, f(Y))\}.$

Ejercicio 4.14 Comprobar si los siguientes conjuntos de expresiones son unificables. En caso de que lo sean, ¿Cuál es el unificador más general?

- a) $\{p(f(Y), W, g(Z)), p(V, U, V)\};$
- b) $\{f(a, X, g(X)), f(a, Y, Y)\};$
- c) $\{p(X, X), p(a, Y)\};$
- d) $\{h(f(X), g(a, Y)), h(f(f(Z)), g(a, X))\};$
- e) $\{p(X_1, g(X_1), X_2, h(X_1, X_2)), p(Y_1, Y_2, f(Y_2), h(g(Y_3), Y_4))\}.$

Cuestión 4.15 a) Enuncie el Principio de Resolución de Robinson para la lógica de predicados. b) Formalice su modo de operación mediante un sistema de transiciones de estado. c) ¿Por qué se dice que los sistemas de prueba que emplean resolución son sistemas de prueba por refutación?

Ejercicio 4.16 Determine los factores de las siguientes cláusulas:

1. $p(X) \vee p(a) \vee q(fX) \vee q(fa)$
2. $p(X) \vee q(Y) \vee p(fX)$
3. $p(a) \vee p(b) \vee p(X)$.

Ejercicio 4.17 Calcule los posibles resolventes de los siguientes pares de cláusulas.

1. $C_1 \equiv \neg p(X) \vee q(X, b); \quad D_1 \equiv p(a) \vee q(a, b);$
2. $C_2 \equiv \neg p(X) \vee q(X, X); \quad D_2 \equiv \neg q(a, fa);$
3. $C_3 \equiv \neg p(X, Y, U) \vee \neg p(Y, Z, V) \vee \neg p(X, V, W) \vee p(U, Z, W);$
 $D_3 \equiv p(g(X, Y), X, Y).$

Ejercicio 4.18 Pruebe que la fórmula $(\exists X)(q(X) \wedge r(X))$ es una consecuencia lógica del conjunto de fórmulas $\{(\exists X)(p(X) \wedge q(X)), (\forall X)(p(X) \rightarrow r(X))\}$, empleando resolución.

Ejercicio 4.19 Demuestre la insatisfacibilidad de los siguientes conjuntos de cláusulas empleando resolución:

- $\Delta_1 = \{\neg p(X) \vee q(f(X), X), \quad p(g(b)), \quad \neg q(Y, Z)\},$
- $\Delta_2 = \{p(X), \quad q(X, f(X)) \vee \neg p(X), \quad \neg q(g(Y), Z)\},$
- $\Delta_3 = \{q(a) \vee r(X), \quad \neg q(X) \vee r(X), \quad \neg r(X) \vee \neg s(a), \quad s(a)\},$

Dibuje los árboles de deducción asociados a cada una de las refutaciones obtenidas.

Ejercicio 4.20 ([26]) Suponga que “San Francisco es amado por todo aquél que ama a alguien”. Suponga también que “todos aman a alguna persona”. Deduzca que “todos aman a San Francisco” empleando resolución lineal. Conteste siguiendo el siguiente orden: a) formalice los enunciados en la lógica de primer orden; b) pase a forma clausal; c) obtenga una refutación.

Cuestión 4.21 ¿Cuál es la esencia de la estrategia de resolución lineal?

Ejercicio 4.22 Pruebe la insatisfacibilidad del conjunto de cláusulas

$$\{p(X, fX, e), \quad \neg s(X) \vee \neg s(Y) \vee \neg p(X, fY, Z) \vee s(Z), \quad s(a), \quad \neg s(e)\}$$

empleando resolución lineal. Utilice la última cláusula del conjunto como cláusula inicial.

Ejercicio 4.23 Pruebe la insatisfacibilidad del conjunto de cláusulas

$$\{\neg p(X) \vee \neg q(X), \neg s(a, X) \vee p(X), p(a), \neg p(X) \vee \neg r(X), \\ \neg t(X) \vee r(X) \vee s(X, f(X)), \neg t(X) \vee r(X) \vee q(f(X)), t(a)\}$$

empleando resolución lineal. Utilice la primera cláusula del conjunto como cláusula inicial.

Ejercicio 4.24 Resuelva el siguiente argumento empleando el principio de resolución o cualquier estrategia de resolución que sea completa para este problema:

Sí dos líneas cualesquiera son perpendiculares a una tercera entonces son paralelas entre sí. Si una línea es perpendicular a otra, la segunda es perpendicular a la primera. Por tanto, si dos líneas no son paralelas, no se da el caso de que haya una tercera que sea perpendicular a ambas.

Siga el mismo esquema que en la solución del Ejercicio 4.20.

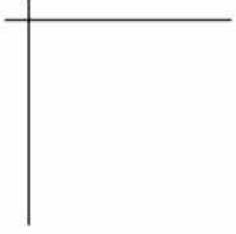
Ejercicio 4.25 Dado el conjunto de cláusulas $\{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$

1. Construya una refutación empleando cada una de las siguientes estrategias de resolución: a) conjunto soporte; b) estrategia de borrado.
2. Demuestre que no existe una refutación empleando la estrategia de resolución lineal.

Ejercicio 4.26 Pruebe la insatisfacibilidad del conjunto de cláusulas

$$\{\neg p(X, Y, U) \vee \neg p(Y, Z, V) \vee \neg p(X, V, W) \vee p(U, Z, W), \\ p(g(X, Y), X, Y), p(X, h(X, Y), Y), \neg p(f(X), X, f(X))\}$$

empleando resolución lineal de entrada. Utilice la primera cláusula del conjunto como cláusula inicial.



Parte II

Programación Lógica

Programación Lógica

Antes de comenzar el estudio de los fundamentos de la programación lógica es interesante realizar una breve discusión de las similitudes y diferencias que la unen y separan de la demostración automática de teoremas, con el fin de delimitar de forma precisa la frontera entre estos dos campos y entender por qué la programación lógica es un paradigma de programación práctico. Entre las similitudes podemos citar:

1. El empleo de un lenguaje basado en la forma clausal.
2. El empleo de pruebas por contradicción.
3. El tipo de regla de inferencia empleado: el principio de resolución. Si bien hay que decir que en el caso de la demostración automática suelen utilizarse variantes más potentes como hiperresolución (enlazada), etc.
4. Compartir ciertos aspectos del empleo de estrategias. Específicamente, ambas usan la estrategia del conjunto de soporte, si bien el tipo de resolución lineal empleado por la programación lógica debe verse como una versión muy restringida de este tipo de estrategia.

Y entre las diferencias más notables:

1. Primero y más importante, ambos campos poseen objetivos diferentes. La demostración automática persigue la construcción de sistemas de deducción que permitan atacar problemas para los que no hay un algoritmo efectivo conocido. La programación lógica se orienta a conseguir la ejecución eficiente de programas lógicos, que solucionan problemas para los que generalmente hay un algoritmo conocido (no necesariamente basado en la búsqueda en un espacio de estados).
2. Un objetivo primordial de la programación lógica es computar. Esto es, establecer para qué valores de las variables (ligadas por el cierre existencial) de una cláusula

\mathcal{G} (denominada *objetivo*) se cumple que ésta es consecuencia lógica de un conjunto de cláusulas Π (llamado *programa*). En otras palabras, y con más precisión, nos interesa computar sustituciones σ tales que $\Pi \vdash \sigma(\mathcal{G})$.

3. La programación lógica no permite el uso de cláusulas generales, restringiéndose a un tipo particular de cláusulas denominadas cláusulas de Horn¹ (cláusulas que contienen como mucho un literal positivo) o alguna extensión de éstas. En la demostración automática de teoremas no se impone ninguna restricción sobre el tipo de cláusulas a emplear en la formalización de un problema.
4. En la programación lógica no hay retención de información intermedia, mientras que para la demostración automática de teoremas esto es esencial para conseguir buenos resultados. Una de las causas que ayudan a la implementación eficiente de los lenguajes de programación lógica es precisamente que éstos no necesitan retener información intermedia.
5. La programación lógica, como ya hemos mencionado, hace un uso muy limitado de las estrategias, en comparación con el amplio uso que de ellas hace la demostración automática de teoremas.
6. La programación lógica destierra el uso del predicado de igualdad² (y, en consecuencia, la posibilidad de trabajar con ecuaciones o reglas para definir funciones), mientras que en la demostración automática de teoremas el tratamiento de la igualdad es uno de los puntos claves. La mayoría de los demostradores automáticos incorporan procedimientos internos (*built-in*) para el tratamiento de la igualdad en su más amplio sentido, esto es, como relación de equivalencia.

De todo lo dicho hasta ahora, podríamos destacar como resumen, que la programación lógica se interesa por el uso de la lógica como lenguaje de programación y por la implementación eficiente del sistema de control que lo hace posible. Siguiendo el discurso del Apartado 3.2, en este capítulo estudiaremos los distintos componentes sintáticos y semánticos que caracterizan un lenguaje de programación lógica. Como veremos, la programación lógica hace uso de un tipo especial de lógica, la denominada lógica de cláusulas de Horn, para la cual la estrategia de resolución SLD es una regla de inferencia completa que puede emplearse como principio operacional.

¹Observe que esto supone una visión restrictiva de lo que se entiende por programación lógica, que nos permite simplificar el nivel de los contenidos teóricos. En un sentido más amplio, la programación lógica abarca también el estudio de programas basados en cláusulas más generales y sus diferentes semánticas.

²Existen ciertas extensiones de la programación lógica que integra lógica y ecuaciones; ver por ejemplo [12].

5.1 SINTAXIS

5.1.1. Notación para las cláusulas

Una cláusula no vacía puede escribirse, agrupando los literales positivos y negativos, de la siguiente forma:

$$(\forall X_1), \dots, (\forall X_s)(M_1 \vee \dots \vee M_m \vee \neg N_1 \vee \dots \vee \neg N_n)$$

donde las metavariables M_i , $i = 1, \dots, m$, y N_j , $j = 1, \dots, n$, representan átomos. La cláusula anterior puede someterse a una sencilla manipulación, aplicando las equivalencias de la Tabla 3.1, de la que se obtiene la siguiente fórmula que equivale lógicamente a la original:

$$(\forall X_1), \dots, (\forall X_s)(N_1 \wedge \dots \wedge N_n \rightarrow M_1 \vee \dots \vee M_m)$$

En el ámbito de la programación lógica es habitual expresar la fórmula anterior empleando para el condicional el símbolo “ \leftarrow ” (denominado *condicional recíproco*³ y leído “si”):

$$(\forall X_1), \dots, (\forall X_s)(M_1 \vee \dots \vee M_m \leftarrow N_1 \wedge \dots \wedge N_n)$$

Esta notación se adapta mejor a una *visión operacional (procedimental)*, que es más adecuada desde el punto de vista de la resolución de problemas utilizando un lenguaje de programación. Informalmente, la última expresión puede leerse en los siguientes términos:

Para resolver el problema M_1 o el M_2 o ... o el M_m , hay que resolver los problemas N_1 y N_2 y ... y N_n .

Aquí, N_1, \dots, N_n pueden considerarse como los procedimientos o rutinas de un lenguaje de programación más convencional. Naturalmente, la *visión declarativa* todavía es posible, de forma que la última expresión también puede leerse en los siguientes términos:

si se cumple N_1 y N_2 y ... y N_n entonces debe cumplirse M_1 o M_2 o ... o M_m .

Nótese que, alternativamente, podríamos haber agrupado los literales positivos o negativos de la fórmula original de manera distinta a la que se hizo, por ejemplo:

$$(\forall X_1), \dots, (\forall X_s)(M_1 \vee \dots \vee M_k \vee \neg N_1 \vee \dots \vee \neg N_n \vee \neg(\neg M_{k+1}) \vee \dots \vee \neg(\neg M_m))$$

para obtener la siguiente fórmula:

$$(\forall X_1), \dots, (\forall X_s)(M_1 \vee \dots \vee M_k) \leftarrow (N_1 \wedge \dots \wedge N_n \wedge \neg M_{k+1} \wedge \dots \wedge \neg M_m)$$

Esta reflexión justifica la siguiente definición de cláusula general (también denominada

³También *implicación inversa (reversed implication)* en [6].

por algunos autores *cláusula de programa*) en la que, pensando es su introducción como texto de programa mediante el teclado de un ordenador, se omiten los cuantificadores en cabeza de la cláusula, teniendo siempre presente que todas las variables de la fórmula están implícitamente cuantificadas de forma universal.

Definición 5.1 (Cláusula general) Una cláusula general es una fórmula de la forma

$$\mathcal{A}_1 \vee \dots \vee \mathcal{A}_m \leftarrow L_1 \wedge \dots \wedge L_n$$

donde $\mathcal{A}_1 \vee \dots \vee \mathcal{A}_m$ es la cabeza (o conclusión o consecuente) de la cláusula general y $L_1 \wedge \dots \wedge L_n$ es el cuerpo (o condición o antecedente). Cada \mathcal{A}_i , $i = 1, \dots, m$, es un átomo y cada L_j , $j = 1, \dots, n$, es un literal (átomo positivo o negativo). Todas las variables se suponen cuantificadas universalmente. Si $m > 1$ la cláusula general se dice que es indefinida y definida si $m = 1$. Si $m = 0$ la cláusula general se dice que es un objetivo.

Nos referiremos a la notación que hemos introducido en este apartado como *notación clausal*. La Tabla 5.1 muestra una clasificación de los diferentes tipos de cláusulas.

Tabla 5.1 Clasificación de las cláusulas y objetivos.

Forma	(m, n)	Denominación
$\mathcal{A} \leftarrow$	($m = 1, n = 0$)	Cláusula unitaria; aserto; hecho.
$\mathcal{A}_1 \vee \dots \vee \mathcal{A}_m \leftarrow$	($m \geq 1, n = 0$)	Cláusula positiva; aserto indefinido.
$\mathcal{A} \leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n$	($m = 1, n \geq 0$)	Cláusula definida; cláusula de Horn con cabeza.
$\mathcal{A} \leftarrow L_1 \wedge \dots \wedge L_n$	($m = 1, n \geq 0$)	Cláusula normal.
$\mathcal{A}_1 \vee \dots \vee \mathcal{A}_m \leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n$	($m \geq 1, n \geq 0$)	Cláusula disyuntiva.
$\mathcal{A}_1 \vee \dots \vee \mathcal{A}_m \leftarrow L_1 \wedge \dots \wedge L_n$	($m \geq 1, n \geq 0$)	Cláusula general.
$\leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n$	($m = 0, n \geq 1$)	Objetivo Horn.
$\leftarrow L_1 \wedge \dots \wedge L_n$	($m = 0, n \geq 1$)	Objetivo normal.
\leftarrow	($m = 0, n = 0$)	Cláusula vacía \square .

Los \mathcal{A}_i 's y los \mathcal{B}_j 's representan átomos, mientras que los L_k 's representan literales.

5.1.2. Cláusulas de Horn y Programas Definidos

La programación lógica deja de lado las cláusulas generales para ocuparse de las cláusulas de Horn y de las cláusulas normales. Nosotros nos centraremos en las primeras dado que los resultados teóricos son más satisfactorios y permiten introducir los conceptos sobre programación lógica más relevantes.

Definición 5.2 (Cláusula de Horn) Una cláusula de Horn (o definida) es una disyunción de literales de los cuales uno, como mucho, es positivo.

Así pues, una cláusula de Horn es una cláusula de la forma

$$\mathcal{A}_1 \vee \dots \vee \mathcal{A}_k \leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n, \quad \text{con } (k = 1 \text{ ó } k = 0) \text{ y } n \geq 0,$$

donde los \mathcal{A}_i 's y los \mathcal{B}_j 's representan átomos. Las cláusulas de Horn con $k = 1$ y $n > 0$ se denominan *reglas*, si $k = 1$ y $n = 0$ se denominan *hechos* y si $k = 0$ y $n > 0$ se denominan *objetivos*. La cláusula vacía puede considerarse como un tipo especial de objetivo.

En las cláusulas de programa (que son los hechos y reglas):

$$\mathcal{A} \leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n, \quad \text{con } n \geq 0$$

el átomo A se denomina *la cabeza* y el conjunto de átomos $\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n$, que suele manipularse como una secuencia, se denomina *el cuerpo*.

Observación 5.1 *El lector puede fácilmente comprobar que, para las variables de una cláusula $\mathcal{A} \leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n$ que no aparecen a la izquierda de la conectiva \leftarrow , el cuantificador universal que está implícito en la fórmula puede escribirse a la derecha del \leftarrow como un cuantificador existencial. En particular, cuando la cláusula es un objetivo, podemos entender que las variables de éste, de forma implícita, están cuantificadas existencialmente.*

$$\forall(\leftarrow (\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n)) \Leftrightarrow \leftarrow \exists(\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n)$$

Ejemplo 5.1

Algunos ejemplos de los diferentes tipos de Cláusulas de Horn, que definen relaciones familiares, son:

<i>hermanas(X, Y) ←</i>	<i>mujer(X) ∧ mujer(Y) ∧</i>	<i>Regla</i>
	<i>padres(X, P, M) ∧ padres(Y, P, M)</i>	
<i>mujer(ana) ←</i>		<i>Hecho</i>
<i>← hermanas(X, ana)</i>		<i>Objetivo</i>

Observaciones 5.1

Algunos hechos interesantes a destacar sobre las cláusulas de Horn son los siguientes:

1. Las cláusulas de Horn tienen el mismo poder de cómputo que la máquina de Turing o el λ -cálculo (i.e., la máquina universal). Permiten por tanto programar cualquier función computable.
2. Para muchas aplicaciones de la lógica basta con restringirse a las cláusulas de Horn, ya que muchos de los problemas que pueden expresarse mediante un formalismo de primer orden pueden transformarse sin mucha dificultad en un conjunto de cláusulas de Horn.

3. La mayoría de los formalismos utilizados en la programación se parecen más a las cláusulas de Horn que a las cláusulas generales. Una cláusula “ $P \leftarrow Q_1, \dots, Q_n$ ” puede entenderse, en términos más convencionales, como la definición de un subprograma:

```
procedure P;
    call Q1
    :
    call Qn
```

cuya cabecera se corresponde con la cabeza de la cláusula y cuyo cuerpo se resuelve en base a las llamadas a otros subprogramas Q_1, \dots, Q_n que estarán definidos en otra parte del programa.

Siguiendo con la analogía, un objetivo $\leftarrow P$ puede considerarse una llamada a un procedimiento como el anterior, en el que define dicho predicado.

4. Muchos métodos de resolución de problemas desarrollados en el campo de la inteligencia artificial pueden considerarse como modelos orientados a ser expresados también mediante cláusulas de Horn [81].



En adelante denominaremos programa (lógico) definido (o simplemente programa lógico) a un conjunto de cláusulas de Horn con cabeza.

Definición 5.3 (Programa lógico definido) *Un programa definido es un conjunto de cláusulas definidas Π . La definición de un símbolo de predicado p que aparece en Π es el subconjunto de cláusulas de Π cuyas cabezas tienen como símbolo de predicado el predicado p .*

Ejemplo 5.2

El conjunto de cláusulas

$$\begin{aligned} C_1 : & p(X, Y) \leftarrow q(X, Y) \\ C_2 : & p(X, Y) \leftarrow q(X, Z) \wedge p(Z, Y) \\ C_3 : & q(a, b) \\ C_4 : & q(b, c) \end{aligned}$$

es un ejemplo de programa definido. Las cláusulas C_1 y C_2 definen el predicado p . Las cláusulas C_3 y C_4 definen el predicado q .

5.2 SEMÁNTICA OPERACIONAL

La semántica operacional de los lenguajes lógicos se basa en la estrategia de resolución denominada resolución SLD. Este es un refinamiento del procedimiento de refutación por resolución original que fue descrito, por vez primera, por Kowalski [82]. Las siglas “SLD” hacen mención a las iniciales inglesas de “resolución Lineal con función de selección para programas definidos”⁴. La estrategia resolución lineal con función de selección, denominada resolución SL, de la que la resolución SLD es descendiente directo, se debe a Kowalski y Kuehner.

En este apartado describimos con detalle la estrategia de resolución SLD e introducimos los conceptos de respuesta computada y árbol de búsqueda SLD, pero antes realizamos una reflexión sobre la forma en la que deben entenderse los objetivos dentro de los procedimientos de prueba.

5.2.1. Procedimientos de prueba y objetivos definidos

La programación lógica intenta comprobar, dado un programa Π y una fórmula $\mathcal{B} \equiv \exists(\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n)$, si se cumple que existe un valor de las variables de \mathcal{B} tal que $\Pi \vdash \mathcal{B}$. Como en los procedimientos de prueba de la demostración automática de teoremas, esta pregunta se contesta por refutación. Así pues, se niega la conclusión \mathcal{B} y se intenta probar la inconsistencia del conjunto $\Pi \cup \{\neg\mathcal{B}\}$, i.e., si $\Pi \cup \{\neg\mathcal{B}\} \vdash \square$. Nótese que

$$\begin{aligned}\neg\mathcal{B} &\Leftrightarrow \neg\exists(\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n) \\ &\Leftrightarrow \leftarrow(\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n)\end{aligned}$$

Por consiguiente, la forma de responder a la pregunta de si la fórmula $\exists(\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n)$ se sigue del programa Π , es añadir a Π la negación de dicha fórmula en forma de objetivo $\mathcal{G} \equiv \leftarrow(\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n)$ y tratar de obtener una refutación, una deducción de \square a partir de $\Pi \cup \{\mathcal{G}\}$.

Sin embargo, como ya dijimos, en la programación lógica no solo nos interesa establecer relaciones de consecuencia lógica (probar teoremas), sino que también nos interesa computar; esto es, establecer para qué valores de las variables (ligadas por el cierre existencial) se cumple el teorema a demostrar. En otras palabras, y con más precisión, nos interesa computar las sustituciones σ tales que $\Pi \vdash \sigma(\mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n)$.

5.2.2. Resolución SLD y respuesta computada

La estrategia de resolución SLD es un caso particular de resolución lineal (de entrada) en la que, dado un programa Π y un objetivo \mathcal{G} , para probar la inconsistencia de $\Pi \cup \{\mathcal{G}\}$ partimos del objetivo \mathcal{G} , que tomamos como cláusula inicial, y resolvemos ésta con alguna cláusula del programa Π . Las cláusulas laterales solamente pueden ser

⁴Esta estrategia también se ha denominado resolución LUSH, aquí las siglas hacen referencia a “Linear resolution with Unrestricted Selection function for Horn clauses”. Por otra parte, Kowalski la ha denominado inferencia analítica (*top-down inference*) en [81].

cláusulas del programa Π de forma que, en cada paso, siempre obtenemos una cláusula objetivo como cláusula central. Más aún, basta con seleccionar uno de los literales del objetivo que se está resolviendo en cada momento para mantener la completitud del procedimiento de refutación⁵.

Definición 5.4 (Regla de computación) *Llamamos regla de computación (o función de selección) φ a una función que, cuando se aplica a un objetivo \mathcal{G} , selecciona uno y solo uno de los átomos de \mathcal{G} .*

Definición 5.5 (Paso de resolución SLD) *Dado un objetivo $\mathcal{G} \equiv \leftarrow \mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_j \wedge \dots \wedge \mathcal{A}_n$ y una cláusula $C \equiv \mathcal{A} \leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_m$, entonces \mathcal{G}' es un resolvente de \mathcal{G} y C (por resolución SLD) usando la regla de computación φ si se cumple que:*

1. $\mathcal{A}_j = \varphi(\mathcal{G})$ es el átomo de \mathcal{G} seleccionado por φ ;
2. $\theta = mgu(\mathcal{A}, \mathcal{A}_j)$;
3. $\mathcal{G}' \equiv \leftarrow \theta(\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_{j-1} \wedge \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_m \wedge \mathcal{A}_{j+1} \wedge \dots \wedge \mathcal{A}_n)$

También diremos que \mathcal{G}' se deriva de \mathcal{G} y C (en un paso) y lo representaremos, en símbolos, mediante diferentes notaciones $\mathcal{G} \xrightarrow{\text{SLD}} \mathcal{G}'$, $\mathcal{G} \xrightarrow{\theta} \mathcal{G}'$ o $\mathcal{G} \xrightarrow{[C, \theta]} \mathcal{G}'$, según el grado de información que sea útil explicitar.

En esencia, las condiciones (1) a (3) indican que \mathcal{G}' se obtiene como resolvente lineal de \mathcal{G} y C cuando se explota, de entre los átomos del objetivo \mathcal{G} , exactamente \mathcal{A}_j (y ninguno más).

Definición 5.6 (Derivación SLD) *Sea Π un programa definido y \mathcal{G} un objetivo definido. Una derivación SLD para $\Pi \cup \{\mathcal{G}\}$, usando la regla de computación φ , consiste en una secuencia de pasos de resolución SLD*

$$\mathcal{G}_0 \xrightarrow{[C_1, \theta_1]} \mathcal{G}_1 \xrightarrow{[C_2, \theta_2]} \mathcal{G}_2 \xrightarrow{[C_3, \theta_3]} \dots \xrightarrow{[C_{n-1}, \theta_{n-1}]} \mathcal{G}_{n-1} \xrightarrow{[C_n, \theta_n]} \mathcal{G}_n$$

(representados en la Figura 5.1 como un árbol^a de deducción) donde:

^aNo confundir estos árboles de deducción con los árboles de búsqueda o árbol de computación de la resolución SLD.

⁵Nótese que, para no perder la completitud, cuando el procedimiento de refutación lineal se aplica a cláusulas generales, hace falta considerar todos los posibles resolventes que se obtienen seleccionando cada uno los pares de literales complementarios que pueden seleccionarse a partir de las cláusulas padres y son susceptibles de unificación.

Definición 5.6 (Derivación SLD) continuación

1. Para $i = 1, \dots, n$, \mathcal{G}_i es un resolvente de \mathcal{G}_{i-1} y C_i (por resolución SLD) usando la regla de computación φ ,
2. Cada C_i es una variante de una cláusula Π , y
3. Cada θ_i es el mgu obtenido en el correspondiente paso i -ésimo de resolución SLD.

En ocasiones utilizaremos la notación $\Pi \cup \{\mathcal{G}\} \vdash_{SLD} \mathcal{G}_n$ para representar una derivación SLD para $\Pi \cup \{\mathcal{G}\}$ que concluye en la cláusula \mathcal{G}_n .

Si \mathcal{G}_n es la última cláusula de la secuencia de resolventes, decimos que la derivación SLD tiene longitud n .

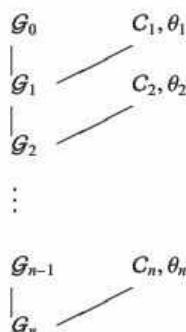


Figura 5.1 Árbol de deducción para una derivación SLD genérica.

Los nombres de las variables de la cláusula de programa C_i que se selecciona para dar un paso de resolución SLD, se eligen de forma que no haya conflicto con ninguna de las variables que hayan aparecido previamente en el curso de la computación (i.e., que no coincidan con los nombres de las variables de ninguna de las cláusulas anteriores \mathcal{G}_{j-1} y C_j , $j < i$). Para conseguirlo, podemos añadir un subíndice i a todas las variables que aparezcan en C_i ; de este modo, construimos una variante de la cláusula C_i . Este proceso de renombramiento de variables, cada vez que se usa una cláusula del programa para dar un paso de resolución SLD, se denomina estandarización de variables.

Observaciones 5.2

1. Nótese que una derivación SLD para $\Pi \cup \{\mathcal{G}\}$ es una forma de resolución lineal de entrada, ya que las cláusulas laterales son cláusulas del programa Π (el conjunto de cláusulas de entrada).

2. Contrariamente a lo que sucede en un paso de resolución lineal, en un paso de resolución SLD de la cláusula \mathcal{G}_i a la \mathcal{G}_{i+1} , no se emplea factorización (i.e., no se calcula un factor de la cláusula \mathcal{G}_i o de una cláusula de entrada para después usarlo en el paso de resolución) ni tampoco resolución con un ancestro (i.e., una cláusula central que haya sido derivada antes que \mathcal{G}_i).



Definición 5.7 (Refutación SLD) Sea Π un programa definido y \mathcal{G} un objetivo definido. Una refutación SLD para $\Pi \cup \{\mathcal{G}\}$ es una derivación SLD finita cuyo último objetivo es la cláusula vacía.

Podemos distinguir los siguientes tipos de derivaciones SLD:

- **Infinita:** cuando en cualquier objetivo \mathcal{G}_i de la secuencia, el átomo \mathcal{A}_j seleccionado por la regla de computación unifica con (una variante de) una cláusula del programa Π .
- **Finita:** la derivación termina en un número finito de pasos. Suponiendo que \mathcal{G}_n es el último objetivo de la secuencia, las derivaciones finitas pueden ser:
 - **De fallo:** ninguna cláusula de Π unifica con el átomo \mathcal{A}_j seleccionado por la regla de computación φ en el objetivo \mathcal{G}_n .
 - **De éxito:** Si $\mathcal{G}_n = \square$. Esto es, una derivación de éxito es una refutación.

Ejemplo 5.3

Dado el programa $\Pi = \{p(f(X)) \leftarrow p(X)\}$ y el objetivo $\mathcal{G} \equiv \leftarrow p(X)$, la siguiente es un ejemplo de derivación SLD infinita para $\Pi \cup \{\mathcal{G}\}$:

$$\leftarrow p(X) \xrightarrow{\{X/f(X_1)\}} \leftarrow p(X_1) \xrightarrow{\{X_1/f(X_2)\}} \leftarrow p(X_2) \xrightarrow{\{X_2/f(X_3)\}} \dots$$

Ejemplo 5.4

Dado el programa $\Pi = \{p(0) \leftarrow q(X)\}$ y el objetivo $\mathcal{G} \equiv \leftarrow p(Z)$, la siguiente es un ejemplo de derivación SLD de fallo para $\Pi \cup \{\mathcal{G}\}$:

$$\leftarrow p(Z) \xrightarrow{\{Z/0\}} \leftarrow q(X) \xrightarrow{\text{fallo}}$$

Ejemplo 5.5

Dado el programa $\Pi = \{C_1 : p(0) \leftarrow q(X), C_2 : q(1) \leftarrow\}$ y el objetivo $G \equiv \leftarrow p(Z)$, la siguiente es un ejemplo de derivación SLD de éxito para $\Pi \cup \{G\}$:

$$\leftarrow p(Z) \xrightarrow{\text{[C}_1,\{Z/0\}\text{]}}_{SLD} \leftarrow q(X) \xrightarrow{\text{[C}_2,\{X/1\}\text{]}}_{SLD} \square$$

Un objetivo primordial de la programación lógica es computar. El efecto de computar se consigue devolviendo una sustitución, que contiene los valores que deben tomar las variables de un objetivo $G \equiv \leftarrow Q$ para que la correspondiente fórmula existencial $\exists Q$ sea deducible a partir de un programa Π . El siguiente concepto expresa la noción de cálculo en programación lógica.

Definición 5.8 (Respuesta computada) *Sea Π un programa definido y G un objetivo definido. Sea $G \xrightarrow{\theta_1} SLD G_1 \xrightarrow{\theta_2} SLD \dots \xrightarrow{\theta_n} \square$ una refutación para $\Pi \cup \{G\}$. Una respuesta computada para $\Pi \cup \{G\}$ es la sustitución θ que resulta de la composición, restringida a las variables de G , de la secuencia de mgu's $\theta_1, \dots, \theta_n$. En símbolos, $\theta = (\theta_n \circ \dots \circ \theta_1) \upharpoonright_{Var(G)}$. Si $G \equiv \leftarrow Q$, se dice que $\theta(Q)$ es una instancia computada de G .*

Observación 5.2

Si $G_0 \xrightarrow{\theta_1} SLD G_1 \xrightarrow{\theta_2} SLD \dots \xrightarrow{\theta_n} SLD G_n$ es una derivación SLD, en ocasiones escribiremos $G \xrightarrow{\theta} SLD^* G_n$, donde $\theta = \theta_n \circ \dots \circ \theta_1$, como notación abreviada para la anterior derivación. Esto será útil cuando no estemos interesados en los detalles de la derivación o queramos representar una derivación que tiene un número arbitrario de pasos. Decimos que G_n se deriva (por resolución SLD) en un número indeterminado de pasos a partir de G_0 .

Ejemplo 5.6

Dado el programa $\Pi = \{C_1 : p(a, X) \leftarrow q(X), C_2 : q(W) \leftarrow\}$ y el objetivo $G \equiv \leftarrow p(Y, b)$, la respuesta computada en la refutación

$$\leftarrow p(Y, b) \xrightarrow{\text{[C}_1,\{Y/a,X/b\}\text{]}}_{SLD} \leftarrow q(b) \xrightarrow{\text{[C}_2,\{W/b\}\text{]}}_{SLD} \square$$

es $\theta = \{Y/a, X/b, W/b\} \upharpoonright_{Var(G)} = \{Y/a\}$. La instancia computada de G es $\theta(p(Y, b)) = p(a, b)$.

El principio de resolución SLD como un sistema de transición de estados

Antes de terminar este subapartado es interesante resaltar que la semántica operacional de un lenguaje lógico puede definirse, de forma alternativa y sintética, en términos de un sistema de transición de estados (ver Apartado 2.3.2). Con este enfoque, si identificamos el concepto de estado E con un par $\langle \mathcal{G}, \theta \rangle$ formado por una cláusula objetivo \mathcal{G} y una sustitución θ , definimos la *resolución SLD* (usando la regla de computación φ) como un sistema de transición cuya relación de transición $\Rightarrow_{SLD} \subseteq (E \times E)$ es la relación más pequeña que satisface:

$$\frac{(\mathcal{G} \equiv\leftarrow Q_1 \wedge \mathcal{A}' \wedge Q_2), \varphi(\mathcal{G}) = \mathcal{A}', C \equiv (\mathcal{A} \leftarrow Q) < \Pi, \sigma = mgu(\mathcal{A}, \mathcal{A}')} {\langle \mathcal{G}, \theta \rangle \Rightarrow_{SLD} \langle \leftarrow \sigma(Q_1 \wedge Q \wedge Q_2), \sigma \circ \theta \rangle}$$

donde Q, Q_1, Q_2 representan conjunciones de átomos cualesquiera y el símbolo “ $<$ ” se ha introducido para expresar que C es una (variante de una) cláusula de Π (que se toma estandarizada aparte). Ahora, una derivación SLD es una secuencia $\langle \mathcal{G}_0, id \rangle \Rightarrow_{SLD} \langle \mathcal{G}_1, \theta_1 \rangle \Rightarrow_{SLD} \dots \Rightarrow_{SLD} \langle \mathcal{G}_n, \theta_n \rangle$. Una refutación SLD es una derivación de éxito $\langle \mathcal{G}_0, id \rangle \Rightarrow_{SLD}^* \langle \square, \sigma \rangle$, donde σ es la respuesta computada en la derivación.

5.2.3. Árboles de búsqueda SLD y procedimientos de prueba

Muchos problemas de difícil solución algorítmica pueden resolverse tratándolos como un problema de búsqueda en un espacio de estados. Un espacio de búsqueda consta de un estado inicial, estados intermedios y estados finales que señalan que se ha alcanzado el éxito en la resolución del problema. El proceso de búsqueda debe ser sistemático y, partiendo de un estado inicial, producir cambios en los sucesivos estados intermedios hasta alcanzar un estado final. El espacio de estados puede representarse mediante una estructura arborescente o, más generalmente, un grafo. Un procedimiento de prueba por refutación puede entenderse como un proceso de búsqueda en un espacio de estados constituidos por los propios objetivos que se generan en cada paso de resolución. En el caso concreto de un programa definido Π y un objetivo definido \mathcal{G} , el conjunto de todas las posibles derivaciones SLD para $\Pi \cup \{\mathcal{G}\}$ se puede representar como un árbol. En lo que sigue formalizamos el concepto de árbol de búsqueda SLD dando una definición constructiva que permite generar este tipo de estructuras.

Definición 5.9 (Árbol de búsqueda SLD) *Sea un programa definido Π y un objetivo definido \mathcal{G} . Un árbol (de búsqueda) SLD τ_φ para $\Pi \cup \{\mathcal{G}\}$ (usando la regla de computación φ) es un conjunto de nodos que cumplen las siguientes restricciones:*

Definición 5.9 (Arbol de búsqueda SLD) continuación

1. El nodo raíz de τ_φ es el objetivo inicial \mathcal{G} ;
2. Si $\mathcal{G}_i \equiv \leftarrow \mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_k \wedge \dots \wedge \mathcal{A}_n$ es un nodo de τ_φ y supuesto que $\varphi(\mathcal{G}_i) = \mathcal{A}_k$ ($1 \leq k \leq n$), entonces para cada cláusula $C_j \equiv \mathcal{A} \leftarrow \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_m$ de Π (con sus variables renombradas si hace falta) cuya cabeza \mathcal{A} unifica con \mathcal{A}_k , con mgu θ , el resolvente

$$\mathcal{G}_{ij} \equiv \leftarrow \theta(\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_{k-1} \wedge \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_m \wedge \mathcal{A}_{k+1} \wedge \dots \wedge \mathcal{A}_n)$$

es un nodo de τ_φ . Se dice que \mathcal{G}_i es el *nodo padre* de \mathcal{G}_{ij} y que \mathcal{G}_{ij} es un *nodo hijo* de \mathcal{G}_i .

Nótese que cada nodo del árbol es una cláusula objetivo obtenido como resolvente SLD del nodo padre y una cláusula del programa. Los nodos que no pueden resolverse con ninguna cláusula del programa, i.e., no tienen hijos, se dice que son las *hojas* del árbol SLD. Los nodos hojas son o bien la cláusula vacía \square o *nodos de fallo*. Debido a la forma en la que se construyen, cada rama de un árbol SLD es una derivación SLD para $\Pi \cup \{\mathcal{G}\}$. Las ramas que se corresponden con derivaciones infinitas se denominan *ramas infinitas*, las que se corresponden con refutaciones se denominan *ramas de éxito* y las que lo hacen con derivaciones de fallo, *ramas de fallo*. Un árbol que contiene un número finito de nodos se dice que es un árbol finito. Dado que consideramos que nuestros programas siempre contienen un número finito de cláusulas, cada uno de los nodos de un árbol SLD tiene un número finito de hijos, i.e., los árboles SLD están ramificados finitamente. Por lo tanto, si un árbol SLD no contiene ramas infinitas, será un árbol finito⁶. Un árbol SLD en el que todas las hojas son de fallo se dice que es un *árbol SLD de fallo finito*.

En este punto conviene resaltar que cada regla de computación φ da lugar a un árbol SLD para $\Pi \cup \{\mathcal{G}\}$ distinto.

Ejemplo 5.7

[[86]] Sean el programa definido

$$\begin{aligned} \Pi = \{ & C_1 : p(X, Z) \leftarrow q(X, Y) \wedge p(Y, Z) \\ & C_2 : p(X, X) \leftarrow \\ & C_3 : q(a, b) \leftarrow \} \end{aligned}$$

y el objetivo definido $\mathcal{G} \equiv \leftarrow p(X, b)$. La Figura 5.2 muestra un árbol de búsqueda SLD para $\Pi \cup \{\mathcal{G}\}$ usando una regla de computación que selecciona el átomo más a la izquierda dentro del objetivo considerado, mientras que la Figura 5.3 muestra el árbol SLD para

⁶Esta es una consecuencia del lema de König que dice que todo árbol infinito ramificado finitamente tiene una rama infinita.

$\Pi \cup \{\mathcal{G}\}$ que se obtiene cuando el átomo seleccionado es el que está más a la derecha. Los átomos seleccionados en cada objetivo aparecen subrayados. Nótese que el árbol de la Figura 5.2 es finito mientras que el de la Figura 5.3 es infinito. Sin embargo, cada árbol contiene exactamente dos ramas de éxito que computan las respuestas $\{X/a\}$ y $\{X/b\}$. El Ejemplo 5.7 muestra que la regla de computación tiene una gran influencia

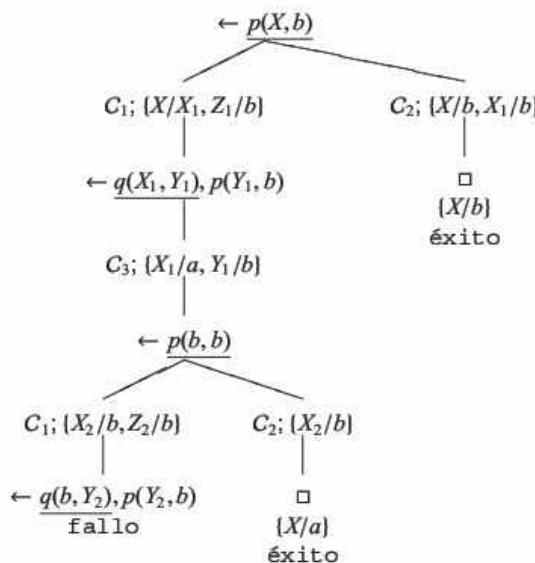


Figura 5.2 Árbol de búsqueda SLD finito del Ejemplo 5.7.

en la forma y el tamaño de los árboles SLD. Sin embargo, cualquiera que sea la regla de computación, si $\Pi \cup \{\mathcal{G}\}$ es inconsistente entonces el árbol SLD para $\Pi \cup \{\mathcal{G}\}$ contendrá una rama de éxito. Esta propiedad se denomina *independencia* de la regla de computación en [86] y puede formularse de forma alternativa para expresar que el conjunto de respuestas computadas por distintos árboles SLD (construidos con diferentes reglas de computación) para un mismo programa Π y objetivo \mathcal{G} es siempre el mismo.

Teorema 5.1 (Independencia de la Regla de Computación) *Sea Π un programa definido y $\mathcal{G} \equiv \leftarrow Q$ un objetivo definido. Si existe una refutación SLD para $\Pi \cup \{\mathcal{G}\}$ con respuesta computada θ , usando una regla de computación φ , entonces existirá también una refutación SLD para $\Pi \cup \{\mathcal{G}\}$ con respuesta computada θ' , usando cualquier otra regla de computación φ' , y tal que $\theta'(Q)$ es una variante de $\theta(Q)$.*

El resultado anterior justifica que la selección de un literal en un objetivo se pueda realizar arbitrariamente sin que afecte al resultado de la computación. La posibilidad de selección arbitraria de los literales en un objetivo se denomina indeterminismo *don't care*⁷ en [81].

⁷El término “don’t care” suele traducirse como “no importa”.

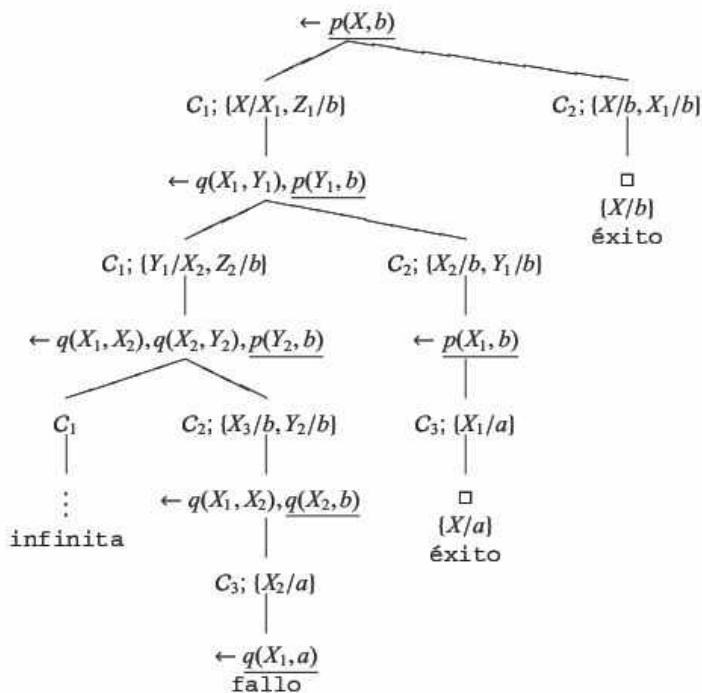


Figura 5.3 Árbol de búsqueda SLD infinito del Ejemplo 5.7.

Observación 5.3

Contrariamente a lo que sucede con la regla de computación, la selección de la cláusula del programa que se utiliza en cada paso de resolución es determinante a la hora de alcanzar el éxito en una derivación. Se dice que la selección de las cláusulas del programa debe hacerse de forma indeterminista don't know⁸ ya que hay que probar a resolver con todas las cláusulas, ya que no podemos saber cuál es la cláusula adecuada que, si se aplica en el paso de resolución actual, garantizará un progreso hacia la obtención de la cláusula vacía. Si queremos mantener la completitud del procedimiento de prueba por refutación SLD no hay más remedio que generar toda la combinatoria posible. La Definición 5.9 de árbol de búsqueda SLD tiene en cuenta el indeterminismo don't know en la selección de las cláusulas del programa de forma que, al generar los hijos de cada nodo, se intenta la resolución del nodo padre con todos las cláusulas del programa. De esta forma, el árbol de búsqueda SLD para un programa y un objetivo define todo el espacio de estados generado por la estrategia de resolución SLD (con una regla de computación dada) para ese programa y objetivo.

Desde el punto de vista de la lógica de predicados, el orden en el que se disponen las fórmulas de una teoría no se tiene en cuenta cuando se desea establecer una prueba.

⁸El término "don't know" suele traducirse como "no se sabe".

Sin embargo, es importante notar que el orden en el que se eligen las cláusulas de un programa, una vez fijada la regla de computación, influye en la forma del árbol SLD. Como vamos a ver, esto tiene repercusiones en los procedimientos de prueba por refutación SLD. Para una misma regla de computación, distintos órdenes de elección para las cláusulas producen árboles SLD diferentes en los que sus ramas aparecen permutadas. Por ejemplo el árbol de la Figura 5.2 se ha generado seleccionando las cláusulas del programa Π de arriba hacia abajo, en el orden de su aparición. Si invertimos este orden y seleccionamos las cláusulas de Π de abajo hacia arriba, el árbol que obtenemos es el representado en la Figura 5.4.

Definición 5.10 (Regla de ordenación de cláusulas) Una regla de ordenación es un criterio por el que se fija el orden en que las cláusulas del programa se intentan resolver con un nodo del árbol SLD, para generar sus correspondientes nodos hijo.

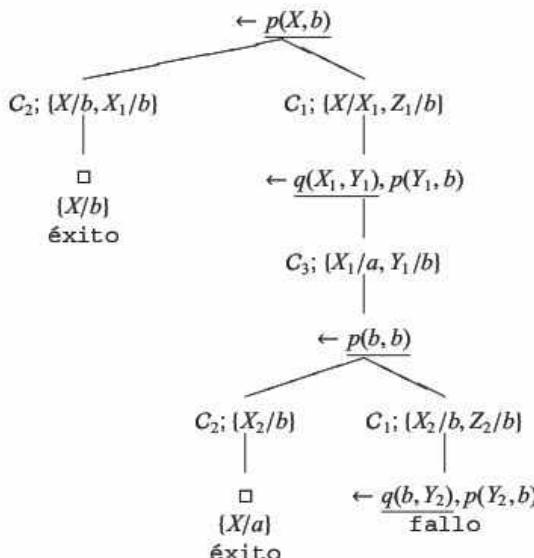


Figura 5.4 Efecto de la reordenación de cláusulas en el árbol de búsqueda SLD del Ejemplo 5.7.

Además de la regla de computación y el orden en que se ensayan las cláusulas del programa, que como se ha señalado tienen impacto sobre la forma del árbol SLD, para buscar las ramas de éxito en un árbol SLD es importante la estrategia con la que se recorre dicho árbol.

Definición 5.11 (Regla de búsqueda) Una regla de búsqueda es una estrategia para recorrer un árbol SLD en busca de una rama de éxito.

Una vez fijadas unas reglas de computación y de ordenación concretas, es evidente que, el árbol búsquedas SLD podría generarse completamente de forma explícita. Sin embargo, en la práctica, esto no se hace así. En lugar de construir explícitamente el árbol de búsquedas SLD y luego buscar en él la cláusula vacía, usando una regla de búsquedas, lo que se proporciona es una representación implícita y solamente se generan explícitamente aquellas partes por las que avanza la estrategia de búsquedas. Hay dos estrategias de búsquedas fundamentales:

1. Búsqueda en amplitud o anchura (*breadth-first*). En este caso se recorre el árbol SLD visitando primero el nodo menos profundo y a la izquierda de entre aquellos todavía no visitados, i.e. se recorre el árbol por niveles. La implementación se realiza mediante una técnica de exploración⁹: en cada iteración del procedimiento de refutación se construye el nivel siguiente del árbol de búsquedas SLD, generando todos los hijos de los estados objetivo que componen el nivel actual. Si en el nivel siguiente se encuentra la cláusula vacía, se termina con éxito; si no, se repite el proceso. Dado que se exploran todas las posibilidades de resolución de los estados objetivo de un nivel con las cláusulas del programa, esta estrategia de búsquedas mantiene la completitud del procedimiento de refutación SLD, aunque es muy costosa, ya que el crecimiento del espacio de estados es exponencial con respecto al nivel de profundidad del árbol de búsquedas SLD.
2. Búsqueda en profundidad (*depth-first*). Esta estrategia recorre el árbol SLD visitando primero el nodo más profundo y a la izquierda de entre aquellos todavía no visitados. Este tipo de búsqueda suele implementarse bien mediante una técnica de exploración o bien mediante un mecanismo de vuelta atrás (*backtracking*). El interés de esta estrategia de búsqueda radica en su eficiente implementación, ya que solo mantiene información sobre los estados de la rama que se está inspeccionando en el momento actual, con el consiguiente ahorro en el almacenamiento de memoria principal y mejora en el tiempo de ejecución, al no ser necesario generar y almacenar todo el espacio de búsquedas. Sin embargo, esta estrategia puede hacer que se pierda la completitud del procedimiento de refutación SLD. Por ejemplo, esta estrategia de búsqueda no puede encontrar la cláusula vacía para el programa y el objetivo del Ejemplo 5.7, cuando se elige como regla de computación “seleccionar el literal más a la derecha del objetivo considerado” y como regla de ordenación “seleccionar las cláusulas de arriba a abajo” (en el orden que aparecen escritas en el texto del programa), ya que primero tiene que recorrer la rama infinita situada más a la izquierda en el árbol de la Figura 5.3 y se pierde en ella.

Un *procedimiento de prueba por refutación SLD* queda completamente especificado fijando: i) una regla de computación; ii) una regla de ordenación; y iii) una regla búsquedas. Las dos primeras fijan la forma del árbol SLD y la última la manera de recorrerlo.

⁹Para una mayor información sobre estas técnicas véase [104] y los apartados 6.2.2 y 9.3 de este libro.

Los procedimientos de prueba de los sistemas Prolog tradicionales emplean una regla de computación que consiste en seleccionar el primer átomo comenzando por la izquierda del objetivo considerado, recorren las cláusulas de arriba a abajo, en el orden de aparición textual en el programa, y emplean una regla de búsqueda que realiza un recorrido de izquierda a derecha en profundidad con vuelta atrás. En el próximo capítulo volveremos sobre estos puntos.

Observación 5.4

Nótese que, aunque Prolog utiliza una regla de ordenación fija, aún se pueden simular diferentes reglas de ordenación introduciendo, explícitamente, las cláusulas de un programa en secuencias diferentes a la original.

5.3 SEMÁNTICA DECLARATIVA Y RESPUESTA CORRECTA

La semántica declarativa de los lenguajes de programación lógica se fundamenta en la semántica (teoría de modelos) de la lógica de predicados. Dado que nuestro interés consiste en probar la insatisfacibilidad de un conjunto de cláusulas, como se discutió en el Apartado 3.5, para dar significado a las construcciones de nuestro lenguaje basta con que nos restrinjamos a las interpretaciones de Herbrand. El estudio que realizamos entonces para caracterizar las interpretaciones de Herbrand nos sirve ahora para fundamentar la semántica declarativa de los lenguajes de programación lógica. Por consiguiente, respecto a ese punto, no es necesario añadir más de lo dicho en el Apartado 3.5.

Ya se ha dicho que uno de los objetivos primordiales de la programación lógica es computar, de ahí el papel central que juega el concepto de respuesta computada. La contrapartida declarativa del concepto de respuesta computada es el concepto de respuesta correcta.

Definición 5.12 *Sea Π un programa definido y sea G un objetivo definido. Una respuesta θ para $\Pi \cup \{G\}$ es cualquier sustitución para las variables de G .*

Nótese que la anterior definición no obliga a que la respuesta contenga un enlace para cada una de las variables de G . En particular, si G es una cláusula sin variables, la única respuesta posible es la sustitución *id*.

Definición 5.13 (Respuesta correcta) *Sea Π un programa definido y sea $G \equiv \leftarrow Q$ un objetivo definido. Sea θ una respuesta para $\Pi \cup \{G\}$. Entonces, θ es una respuesta correcta para $\Pi \cup \{G\}$ si y solo si $\Pi \models (\forall \theta(Q))$, i.e. $\forall \theta(Q)$ es una consecuencia lógica de Π .*

Ejemplo 5.8

Sea el programa $\Pi = \{p(X, Y) \leftarrow q(Y), \quad q(f(a)) \leftarrow\}$. Dado $G \equiv \leftarrow p(Z_1, Z_2)$, entonces $\{Z_2 / f(a)\}$ es una respuesta correcta para $\Pi \cup \{G\}$.

En virtud del teorema de la deducción, versión semántica (Teorema 2.3), podemos establecer que θ es una respuesta correcta para $\Pi \cup \{\mathcal{G}\}$ si y solo si $\Pi \cup \{\neg(\forall\theta(Q))\}$ es insatisfacible. Para comprobar que θ es una respuesta correcta, contrariamente a lo que de forma ingenua uno podría pensar, no es suficiente con comprobar que $\neg(\forall\theta(Q))$ es falsa para todo modelo de Herbrand del programa Π . Esto es, el hecho de que $\Pi \cup \{\neg(\forall\theta(Q))\}$ no tenga modelos de Herbrand no implica que deba ser insatisfacible. La razón es que, en general, $\neg(\forall\theta(Q))$ no es una cláusula y el Teorema 3.1 no es aplicable, por lo cual no podemos restringir nuestra atención solo a las interpretaciones de Herbrand, debiendo considerar, también, interpretaciones generales. Sin embargo, cuando nos limitamos a respuestas θ tales que la instancia $\theta(Q)$ es una expresión básica, puede establecerse el siguiente resultado.

Proposición 5.1 [[86]] Sea Π un programa definido y sea $\mathcal{G} \equiv \leftarrow Q$ un objetivo definido. Sea θ una respuesta para $\Pi \cup \{\mathcal{G}\}$ tal que $\theta(Q)$ es una expresión básica. Entonces, θ es una respuesta correcta para $\Pi \cup \{\mathcal{G}\}$ si y solo si $\theta(Q)$ es verdadera para todo modelo de Herbrand del programa Π .

El siguiente ejemplo hace uso del resultado anterior para justificar la corrección de algunas respuestas.

Ejemplo 5.9

Consideremos nuevamente el programa $\Pi = \{p(X, Y) \leftarrow q(Y), q(f(a)) \leftarrow\}$. El universo de Herbrand para el lenguaje de primer orden generado por Π es el conjunto $\mathcal{U}_L(\Pi) = \{a, f(a), f(f(a)), f(f(f(a))), \dots\}$ y es fácil comprobar que las H-interpretaciones \mathcal{I} que son modelo de Π contienen el conjunto (modelo mínimo) $\mathcal{M} = \{q(f(a))\} \cup \{p(t, f(a)) \mid t \in \mathcal{U}_L(\Pi)\}$. Esto es, si \mathcal{I} es modelo entonces $\mathcal{M} \subseteq \mathcal{I}$.

1. Dado el objetivo $\mathcal{G}_1 \equiv \leftarrow p(Z_1, Z_2)$.

- La respuesta $\theta_1 = \{Z_1/a, Z_2/f(a)\}$ es correcta, dado que $\theta_1(p(Z_1, Z_2)) = p(a, f(a)) \in \mathcal{M}$, luego $\theta_1(p(Z_1, Z_2))$ es verdadera en todo modelo de Herbrand de Π .
- La respuesta $\theta_2 = \{Z_1/b, Z_2/f(a)\}$ no es correcta, ya que $\theta_2(p(Z_1, Z_2)) = p(b, f(a)) \notin \mathcal{I}$, cualquiera que sea la H-interpretación \mathcal{I} (nótese que $b \notin \mathcal{U}_L(\Pi)$).

2. Dado el objetivo $\mathcal{G}_2 \equiv \leftarrow p(Z, Z)$.

- La respuesta $\sigma_1 = \{Z/f(a)\}$ es correcta, ya que $\sigma_1(p(Z, Z)) = p(f(a), f(a)) \in \mathcal{M}$, luego $\sigma_1(p(Z, Z))$ es verdadera en todo modelo de Herbrand de Π .
- La respuesta $\sigma_2 = \{Z/b\}$ no es correcta, ya que $\sigma_2(p(Z, Z)) = p(b, b) \notin \mathcal{I}$, cualquiera que sea la H-interpretación \mathcal{I} (nótese que $b \notin \mathcal{U}_L(\Pi)$).

3. Dado el objetivo $\mathcal{G}_3 \equiv \leftarrow q(f(a))$.
 - La respuesta $\gamma = id$ es correcta, ya que $\gamma(q(f(a))) = q(f(a)) \in M$.
4. Dado el objetivo $\mathcal{G}_1 \equiv \leftarrow q(f(b))$.
 - No existe respuesta correcta ya que $\Pi \cup \{\neg q(f(b))\}$ es satisfacible. Esto es debido a que, para toda H-interpretación I , $q(f(b)) \notin I$ y, por lo tanto, $\neg q(f(b))$ es verdadero en cada H-interpretación I (incluidas aquéllas que son modelo de Π).

5.4 CORRECCIÓN Y COMPLETITUD DE LA RESOLUCIÓN SLD

El siguiente resultado establece la corrección y completitud de nuestro sistema formal basado en la estrategia de resolución SLD.

Teorema 5.2 ([86]) *Sean Π un programa definido y \mathcal{G} un objetivo definido.*

- a) (Corrección) *Si existe una refutación SLD para $\Pi \cup \{\mathcal{G}\}$, i.e. $\mathcal{G} \Rightarrow_{SLD}^* \square$, entonces $\Pi \cup \{\mathcal{G}\}$ es insatisfacible.*
- b) (Completitud) *Si $\Pi \cup \{\mathcal{G}\}$ es insatisfacible entonces existe una refutación SLD para $\Pi \cup \{\mathcal{G}\}$, i.e. $\mathcal{G} \Rightarrow_{SLD}^* \square$.*

Esto es, el refinamiento conocido como resolución SLD, cuando se aplica a programas definidos, no hace perder la completitud de la regla de resolución.

Es habitual presentar los resultados de corrección y completitud estudiando la relación de las semánticas operacional y declarativa en función de la relación entre las respuestas computadas y las correctas. Es posible establecer los siguientes resultados, cuyas pruebas pueden encontrarse en [6] o bien en [86].

Teorema 5.3 (Teorema de la Corrección) *Sea Π un programa definido y sea \mathcal{G} un objetivo definido. Toda respuesta computada para $\Pi \cup \{\mathcal{G}\}$ es una respuesta correcta para $\Pi \cup \{\mathcal{G}\}$.*

No es posible probar exactamente el inverso del Teorema 5.3, esto es, que cada respuesta correcta para $\Pi \cup \{\mathcal{G}\}$ también es una respuesta computada para $\Pi \cup \{\mathcal{G}\}$. Sin embargo, puede probarse que cada respuesta correcta es una instancia de una respuesta computada.

Teorema 5.4 (Teorema de la Completitud) *Sea Π un programa definido y sea \mathcal{G} un objetivo definido. Para cada respuesta correcta θ para $\Pi \cup \{\mathcal{G}\}$, existe una respuesta computada σ para $\Pi \cup \{\mathcal{G}\}$ que es más general que θ cuando nos restringimos a las variables de \mathcal{G} (en símbolos, $\sigma \leq \theta[Var(\mathcal{G})]$).*

Dado que una respuesta correcta por definición, está restringida a las variables del objetivo \mathcal{G} , escribir $\sigma \leq \theta[\text{Var}(\mathcal{G})]$ es lo mismo que decir que existe una sustitución δ tal que $\theta = (\delta \circ \sigma)_{\mathcal{G}}$.

Ejemplo 5.10

Sea el programa definido $\Pi = \{p(X, Y) \leftarrow q(Y), \quad q(f(a)) \leftarrow\}$ y el objetivo definido $\mathcal{G} \equiv \leftarrow p(Z_1, Z_2)$. Dado el universo de Herbrand $\mathcal{U}_L(\Pi) = \{a, f(a), f(f(a)), \dots\}$, es inmediato comprobar que la sustitución $\theta = \{Z_1/a, Z_2/f(a)\}$ es una respuesta correcta. También es fácil comprobar que la sustitución $\sigma = \{Z_1/X, Z_2/f(a)\}$ es una respuesta computada y que existe una sustitución $\delta = \{X/a\}$ tal que $\theta = (\delta \circ \sigma)_{\mathcal{G}}$. Es importante notar que la relación $\sigma \leq \theta$ no se cumple si se elimina la restricción a las variables de \mathcal{G} . Es decir, no se cumple que exista una sustitución δ tal que $\theta = (\delta \circ \sigma)$.

Ejemplo 5.11

Sea el programa definido $\Pi = \{p(X, a) \leftarrow\}$ y el objetivo definido $\mathcal{G} \equiv \leftarrow p(Y, a)$. El universo de Herbrand $\mathcal{U}_L(\Pi) = \{a\}$ y es inmediato comprobar que la sustitución $\theta = \{Y/a\}$ es una respuesta correcta. La única respuesta computada es la sustitución $\sigma = \{Y/X\}$. Existe una sustitución $\delta = \{X/a\}$ tal que $\theta = (\delta \circ \sigma)_{\mathcal{G}}$; sin embargo, no se cumple que $\theta = \delta \circ \sigma$.

Para finalizar este apartado, damos dos resultados que constituyen formulaciones alternativas del teorema de la completitud para la resolución SLD (Teorema 5.2(b)). Comenzamos con un corolario inmediato de dicho teorema y de la Definición 5.9.

Proposición 5.2 Sea Π un programa definido y sea \mathcal{G} un objetivo definido. Si $\Pi \cup \{\mathcal{G}\}$ es insatisfacible entonces cada árbol SLD para $\Pi \cup \{\mathcal{G}\}$ contiene (al menos) una rama de éxito.

La siguiente reformulación es una consecuencia inmediata de la propiedad de independencia de la regla de computación y el Teorema 5.2(b).

Proposición 5.3 Sea Π un programa definido y sea \mathcal{G} un objetivo definido. Si $\Pi \cup \{\mathcal{G}\}$ es insatisfacible entonces para cada átomo $\mathcal{A} \in \mathcal{G}$ existe una refutación SLD para $\Pi \cup \{\mathcal{G}\}$ con \mathcal{A} como primer átomo seleccionado.

Esta versión del teorema de la completitud para la resolución SLD nos indica que, con independencia la regla de computación que utilicemos para seleccionar el átomo del objetivo \mathcal{G} a resolver, siempre encontraremos una refutación SLD para $\Pi \cup \{\mathcal{G}\}$. Es otra forma de poner de manifiesto el indeterminismo *don't care* en la selección de los literales de un objetivo.

5.5 SIGNIFICADO DE LOS PROGRAMAS

Las semánticas formales, además de dar cuenta del significado de las construcciones generales de un lenguaje, también deben asignar significado a los programas. Una forma de abordar este problema es asociando a cada programa ciertas propiedades que denominamos *observables*: el conjunto de éxitos básicos (i.e., átomos básicos que son derivables a partir del programa mediante resolución SLD), el número de derivaciones de éxito, las derivaciones de fallo finito, las respuestas computadas, etc.

Elegido un observable O , éste induce una relación de equivalencia, $=_O$, sobre los programas, que se define del modo siguiente: Sean Π_1 y Π_2 dos programas; $\Pi_1 =_O \Pi_2$ si y solo si Π_1 y Π_2 son indistinguibles con respecto a la propiedad observable O . Por razones históricas y de simplicidad expositiva, estamos interesados en el observable de los éxitos básicos (\mathcal{EB}). Así pues, $\Pi_1 =_{\mathcal{EB}} \Pi_2$ si y solo si Π_1 y Π_2 son indistinguibles por dar lugar a los mismos conjuntos de éxitos básicos. Un estudio del significado de los programas lógicos basado en el observable *respuestas computadas* puede encontrarse en [48, 49, 51] y una recensión en [6].

Las aproximaciones semánticas que vamos a estudiar son *correctas* y *completas* con respecto al observable \mathcal{EB} , en el sentido de que, dada una semántica cualquiera S , dos programas Π_1 y Π_2 tienen la misma semántica —i.e., $S(\Pi_1) = S(\Pi_2)$ — si y solo si $\Pi_1 =_{\mathcal{EB}} \Pi_2$.

5.5.1. Semántica operacional

La semántica operacional de un programa definido Π se caracteriza mediante el llamado conjunto de éxitos básicos de Π .

Definición 5.14 (Conjunto de éxitos básicos) *Sea Π un programa definido. El conjunto de éxitos básicos de Π , denotado $\mathcal{EB}(\Pi)$, es el conjunto de todos los átomos básicos \mathcal{A} de la base de Herbrand de Π tales que, cuando se lanzan como objetivos $\leftarrow \mathcal{A}$, existe una refutación SLD para $\Pi \cup \{\leftarrow \mathcal{A}\}$. Esto es,*

$$\mathcal{EB}(\Pi) = \{\mathcal{A} \mid (\mathcal{A} \in \mathcal{B}_L(\Pi)) \wedge (\Pi \cup \{\leftarrow \mathcal{A}\} \vdash_{SLD} \square)\}.$$

Intuitivamente, el significado del programa Π está determinado por el conjunto de los hechos básicos que pueden establecerse a partir de él.

Ejemplo 5.12

Sea el programa $\Pi = \{C_1 : p(a) \leftarrow, C_2 : p(b) \leftarrow, C_3 : r(X) \leftarrow p(X), C_4 : t(X, Y) \leftarrow p(X) \wedge r(Y), C_5 : s(c) \leftarrow\}$. Vamos a calcular la semántica operacional de Π . Para ello seguiremos el procedimiento de, primero, calcular la base de Herbrand de Π y, después, comprobar para qué elementos de ella hay una refutación SLD:

1. En este caso: $\mathcal{U}_{\mathcal{L}}(\Pi) = \{a, b, c\}$ y

$$\mathcal{B}_{\mathcal{L}}(\Pi) = \{p(a), p(b), p(c), r(a), r(b), r(c), s(a), s(b), s(c), t(a, a), \\ t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c)\}$$

2. Existen las siguientes refutaciones SLD para $\Pi \cup \{\leftarrow \mathcal{A}\}$, con $\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)$:

- $\leftarrow p(a) \xrightarrow[C_1,id]{SLD} \square$.
- $\leftarrow p(b) \xrightarrow[C_2,id]{SLD} \square$.
- $\leftarrow s(c) \xrightarrow[C_5,id]{SLD} \square$.
- $\leftarrow r(a) \xrightarrow[C_3,\{X_3/a\}]{SLD} \leftarrow p(a) \xrightarrow[C_1,id]{SLD} \square$.
- $\leftarrow r(b) \xrightarrow[C_3,\{X_3/b\}]{SLD} \leftarrow p(b) \xrightarrow[C_2,id]{SLD} \square$.
- $\leftarrow t(a, a) \xrightarrow[C_3,\{X_4/a,Y_4/a\}]{SLD} \leftarrow p(a) \wedge r(a) \xrightarrow[C_1,id]{SLD} \leftarrow r(a) \xrightarrow[C_3,\{X_3/a\}]{SLD} \leftarrow p(a) \xrightarrow[C_1,id]{SLD} \square$.
- $\leftarrow t(a, b) \xrightarrow[C_3,\{X_4/a,Y_4/b\}]{SLD} \leftarrow p(a) \wedge r(b) \xrightarrow[C_1,id]{SLD} \leftarrow r(b) \xrightarrow[C_3,\{X_3/b\}]{SLD} \leftarrow p(b) \xrightarrow[C_2,id]{SLD} \square$.
- $\leftarrow t(b, a) \xrightarrow[C_3,\{X_4/b,Y_4/a\}]{SLD} \leftarrow p(b) \wedge r(a) \xrightarrow[C_2,id]{SLD} \leftarrow r(a) \xrightarrow[C_3,\{X_3/a\}]{SLD} \leftarrow p(a) \xrightarrow[C_1,id]{SLD} \square$.
- $\leftarrow t(b, b) \xrightarrow[C_3,\{X_4/b,Y_4/b\}]{SLD} \leftarrow p(b) \wedge r(b) \xrightarrow[C_2,id]{SLD} \leftarrow r(b) \xrightarrow[C_3,\{X_3/b\}]{SLD} \leftarrow p(b) \xrightarrow[C_2,id]{SLD} \square$.

El resto de los átomos \mathcal{A} de la base de Herbrand $\mathcal{B}_{\mathcal{L}}(\Pi)$ no satisfacen la condición de que exista una refutación SLD para $\Pi \cup \{\leftarrow \mathcal{A}\}$.

Por consiguiente, $\mathcal{EB}(\Pi) = \{p(a), p(b), r(a), r(b), s(c), t(a, a), t(a, b), t(b, a), t(b, b)\}$. Como se ha podido comprobar, el procedimiento empleado para computar el conjunto $\mathcal{EB}(\Pi)$ (generar toda la base de Herbrand y seleccionar de ella los átomos que, lanzados como objetivo, conducen al éxito), es bastante tedioso de aplicar, además de ingenuo en la forma de afrontar el problema, por lo que se hace necesario un procedimiento para calcular $\mathcal{EB}(\Pi)$ que sea más constructivo. También es conveniente notar que, en la mayoría de los casos, a un programa Π puede corresponderle un conjunto de éxitos infinito.

5.5.2. Semántica declarativa

La semántica declarativa de un lenguaje lógico está asociada al concepto de interpretación de Herbrand; por lo tanto, podemos asignar significado declarativo a un programa seleccionando uno de sus modelos de Herbrand. Naturalmente, nuestra preferencia debe decantarse por el modelo *más simple* de entre los posibles. El siguiente resultado, junto con la Observación 3.4(3) y la Observación 3.1 hacen posible una caracterización precisa de ese modelo de Herbrand *más simple* de un programa.

Proposición 5.4 [Propiedad de Intersección de Modelos]

Sea Π un programa definido. Sea I un conjunto de índices y $\{\mathcal{M}_i \mid i \in I \wedge \mathcal{M}_i \models \Pi\}$ un

conjunto *no vacío* de modelos de Herbrand de Π . Entonces, el conjunto $\bigcap_{i \in I} \mathcal{M}_i$ es un modelo de Herbrand de Π .

Observación 5.5

La propiedad de intersección de modelos solo se cumple para cláusulas de Horn definidas. Por ejemplo, la cláusula $C \equiv (p \vee q)$ tiene tres modelos de Herbrand:

- $\mathcal{M}_1 = \{p\};$
- $\mathcal{M}_2 = \{q\};$
- $\mathcal{M}_3 = \{p, q\};$

cuya intersección es el conjunto vacío, que no es modelo de C .

Debido a que la base de Herbrand $\mathcal{B}_{\mathcal{L}}(\Pi)$ de un programa Π es un modelo de Herbrand de Π , el conjunto de todos los modelos de Herbrand de Π no es vacío. Como consecuencia de la anterior afirmación y de la Proposición 5.4, la intersección de todos los modelos de Herbrand de un programa Π es también un modelo de Herbrand de Π . La Observación 3.1 nos asegura que es un modelo mínimo. En adelante, denotamos este modelo como $\mathcal{M}(\Pi)$ y lo denominamos *modelo mínimo de Herbrand* de $\mathcal{B}_{\mathcal{L}}(\Pi)$.

Definimos la semántica declarativa de un programa Π como el modelo mínimo de Herbrand $\mathcal{M}(\Pi)$. El siguiente teorema muestra que $\mathcal{M}(\Pi)$ es la interpretación natural de un programa Π , ya que los átomos que la forman son, precisamente, aquellos átomos básicos que son consecuencia lógica del programa. Este resultado fue demostrado por Kowalski y van Emden.

Teorema 5.5 *Sea Π un programa definido. Entonces, $\mathcal{M}(\Pi) = \{\mathcal{A} \mid \mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi) \wedge \Pi \models \mathcal{A}\}$.*

Ejemplo 5.13

Sea el programa $\Pi = \{C_1 : q(X) \leftarrow p(X), C_2 : p(a) \leftarrow, C_3 : q(b) \leftarrow\}$. Para calcular la semántica declarativa de Π , primero calculamos la base de Herbrand de Π ; después, tenemos que comprobar qué interpretaciones de Herbrand, de entre las posibles, son modelos de Π :

1. Es fácil comprobar que $\mathcal{U}_{\mathcal{L}}(\Pi) = \{a, b\}$ y $\mathcal{B}_{\mathcal{L}}(\Pi) = \{p(a), p(b), q(a), q(b)\}$. Por consiguiente, existen ($2^4 =$)16 H-interpretaciones posibles:

- $\mathcal{I}_1 = \emptyset,$
- $\mathcal{I}_2 = \{p(a)\}, \quad \mathcal{I}_3 = \{p(b)\}, \quad \mathcal{I}_4 = \{q(a)\}, \quad \mathcal{I}_5 = \{q(b)\},$

- $\mathcal{I}_6 = \{p(a), p(b)\}$, $\mathcal{I}_7 = \{p(a), q(a)\}$, $\mathcal{I}_8 = \{p(a), q(b)\}$,
 $\mathcal{I}_9 = \{p(b), q(a)\}$, $\mathcal{I}_{10} = \{p(b), q(b)\}$, $\mathcal{I}_{11} = \{q(a), q(b)\}$,
- $\mathcal{I}_{12} = \{p(a), p(b), q(a)\}$, $\mathcal{I}_{13} = \{p(a), p(b), q(b)\}$,
 $\mathcal{I}_{14} = \{p(a), q(a), q(b)\}$, $\mathcal{I}_{15} = \{p(b), q(a), q(b)\}$,
- $\mathcal{I}_{16} = \{p(a), p(b), q(a), q(b)\}$.

Así pues, aún para este sencillo programa hay que realizar un gran esfuerzo de comprobación.

2. Tras una tediosa comprobación se puede afirmar que las únicas H-interpretaciones modelos de Π son \mathcal{I}_{14} e \mathcal{I}_{16} :

- \mathcal{I}_{14} ya que: i) solo existen dos instancias básicas de C_1 , $C'_1 \equiv q(a) \leftarrow p(a)$ y $C''_1 \equiv q(b) \leftarrow p(b)$, ambas son verdaderas en \mathcal{I}_{14} (la primera porque $p(a) \in \mathcal{I}_{14}$ implica que $q(a) \in \mathcal{I}_{14}$ y la segunda porque $p(b) \notin \mathcal{I}_{14}$) y por lo tanto $\mathcal{I}_{14} \models C_1$; ii) $p(a) \in \mathcal{I}_{14}$ y por lo tanto $\mathcal{I}_{14} \models C_2$; iii) $q(b) \in \mathcal{I}_{14}$ y por lo tanto $\mathcal{I}_{14} \models C_3$;
- \mathcal{I}_{16} coincide con la base de Herbrand de Π y por lo tanto es un modelo de Π

Por tanto, tenemos que la interpretación modelo mínimo de Herbrand es ahora $\mathcal{M}(\Pi) = \mathcal{I}_{14} \cap \mathcal{I}_{16} = \{p(a), q(a), q(b)\}$. Nuevamente, el ejemplo 5.13 pone de manifiesto la necesidad de un mecanismo menos costoso y más constructivo que nos permita hallar la interpretación estándar de un programa. El siguiente apartado nos proporciona ese mecanismo.

5.5.3. Semántica por punto fijo

En este apartado damos una visión constructiva del significado de un programa siguiendo la teoría del punto fijo. La idea es asociar, a cada programa, un operador continuo que permita, mediante sucesivas aplicaciones, construir el modelo mínimo del programa. Antes de seguir adelante conviene resumir algunos conceptos y resultados de esta teoría.

Teoría del punto fijo

Un conjunto S sobre el que se ha definido un orden parcial \leq , se dice que está *parcialmente ordenado*. Dado un conjunto S parcialmente ordenado y un subconjunto X de S , escribimos $\inf(X)$ para denotar el ínfimo del conjunto X y $\sup(X)$ para denotar el supremo del conjunto X .

Definición 5.15 Un conjunto parcialmente ordenado L es un retículo completo si y solo si para todo subconjunto X de L existe $\inf(X)$ y $\sup(X)$.

Denotamos por \top el $\sup(L)$ y el $\inf(L)$ por \perp .

Definición 5.16 *Sea L un retículo completo y $X \subseteq L$. Decimos que X es un conjunto dirigido^a si todo subconjunto finito de X tiene una cota superior en X .*

^aHemos traducido como “conjunto dirigido” lo que en los textos escritos en lengua inglesa se denomina *directed set*.

Definición 5.17 *Sea L un retículo completo y $T : L \rightarrow L$ una aplicación. Decimos que T es:*

1. Monótona si y solo si, siempre que $x \leq y$ entonces $T(x) \leq T(y)$.
2. Continua si y solo si para todo conjunto dirigido X de L entonces, $T(\sup(X)) = \sup(T(X))$.

Si bien toda aplicación continua T es monótona, lo contrario no se cumple.

Ahora podemos definir qué se entiende por punto fijo de una aplicación T y presentar algunas de sus propiedades.

Definición 5.18 *Sea L un retículo completo y $T : L \rightarrow L$ una aplicación. Decimos que $a \in L$ es un punto fijo de T si $T(a) = a$. Si $a \in L$ es un punto fijo de T , decimos que a es un menor punto fijo de T si y solo si $a \leq b$ para todo punto fijo b de T . El menor punto fijo de T lo denotamos por $mpf(T)$.*

La siguiente proposición es una simplificación de un resultado debido a Knaster y Traski.

Proposición 5.5 [Teorema del Punto Fijo] *Sea L un retículo completo y $T : L \rightarrow L$ una aplicación monótona. Entonces T tiene un menor punto fijo $mpf(T)$. Además, se cumple que $mpf(T) = \inf(\{x \mid T(x) = x\}) = \inf(\{x \mid T(x) \leq x\})$.*

A continuación introducimos el concepto de potencias ordinales de T . Los números ordinales son aquellos que utilizamos para contar. Los números ordinales finitos son los números naturales. Los números ordinales transfinitos son aquellos que suceden a los números naturales. El primer ordinal transfinito lo representamos mediante el símbolo ω . De forma muy burda podemos decir que ω es el primer infinito. Se cumple que $n < \omega$ para todo ordinal finito n , pero ω no es el sucesor de ningún ordinal finito. Por este último motivo, decimos que ω es un *ordinal límite*. Los ordinales que son los sucesores de otro ordinal se denominan *ordinales sucesores* (e.g., todo ordinal finito (salvo el cero) es el sucesor de otro ordinal finito, por lo tanto es un ordinal sucesor).

Definición 5.19 Sea L un retículo completo y $T : L \rightarrow L$ una aplicación monótona. Entonces, definimos las potencias ordinales de T inductivamente como:

$$\begin{aligned} T \uparrow 0 &= \perp \\ T \uparrow \alpha &= T(T \uparrow (\alpha - 1)), \quad \text{si } \alpha \text{ es un ordinal sucesor;} \\ T \uparrow \alpha &= \inf(\{T \uparrow \beta \mid \beta < \alpha\}), \quad \text{si } \alpha \text{ es un ordinal límite,} \end{aligned}$$

La siguiente proposición debida a Kleene muestra que, bajo la suposición de que T es continua, es posible caracterizar el $\text{mpf}(T)$ en términos de potencias ordinales de T .

Proposición 5.6 Sea L un retículo completo y $T : L \rightarrow L$ una aplicación continua. Entonces, $\text{mpf}(T) = T \uparrow \omega$. ■

Esta proposición nos dice que podemos calcular el menor punto fijo de una aplicación continua T , empleando la Definición 5.19 para computar las potencias ordinales de T hasta el primer ordinal límite ω .

El operador de consecuencias lógicas inmediatas

Dado un programa Π , es posible asociarle un retículo completo sobre el que definir un operador continuo. El objetivo es lograr una caracterización del modelo mínimo de Herbrand haciendo uso de las buenas propiedades de dicho operador. Recordemos que el conjunto $\wp(\mathcal{B}_L(\Pi))$, con el orden “ \subseteq ” de inclusión de conjuntos es un retículo completo, que denominamos el retículo completo de las interpretaciones de Herbrand (Observación 3.4(3)). Este es el retículo sobre el que definiremos el llamado operador de consecuencias lógicas inmediatas.

Definición 5.20 Sea Π un programa definido. El operador de consecuencias lógicas inmediatas T_Π es una aplicación $T_\Pi : \wp(\mathcal{B}_L(\Pi)) \rightarrow \wp(\mathcal{B}_L(\Pi))$ definida como:

$$\begin{aligned} T_\Pi(\mathcal{I}) = \{\mathcal{A} \mid \mathcal{A} \in \mathcal{B}_L(\Pi) \wedge \\ (\mathcal{A} \leftarrow \mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_n) \in \text{Basicas}(\Pi) \wedge \\ \{\mathcal{A}_1, \dots, \mathcal{A}_n\} \subseteq \mathcal{I}\} \end{aligned}$$

donde \mathcal{I} es una interpretación de Herbrand para Π y $\text{Basicas}(\Pi)$ es el conjunto de instancias básicas de las cláusulas de Π .

En palabras, la definición anterior nos dice que, para cada instancia básica de una cláusula del programa Π , si los átomos que forman su cuerpo $\{\mathcal{A}_1, \dots, \mathcal{A}_n\}$ están contenidos en la interpretación \mathcal{I} entonces la cabeza \mathcal{A} de dicha cláusula pertenecerá a la interpretación $T_\Pi(\mathcal{I})$. El siguiente argumento justifica el nombre asignado al operador T_Π : supuesto que la interpretación \mathcal{I} es modelo de Π , la referida instancia básica es verdadera en esa interpretación y si $\{\mathcal{A}_1, \dots, \mathcal{A}_n\} \subseteq \mathcal{I}$ (i.e., la fbf $\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_n$ es verdadera

en \mathcal{I}), entonces el átomo \mathcal{A} es verdadero en \mathcal{I} ; así pues, por definición, \mathcal{A} es una consecuencia lógica del programa Π ; por consiguiente, la interpretación $T_\Pi(\mathcal{I})$ está formada por todos los átomos de la base de Herbrand que son consecuencia lógica inmediata del programa Π .

Ejemplo 5.14

Consideremos de nuevo el programa del Ejemplo 5.13. Para familiarizarnos con el modo de operación de T_Π vamos a calcular el resultado de su aplicación sobre la interpretación $\mathcal{I}_9 = \{p(b), q(a)\}$. Podemos obtener $T_\Pi(\mathcal{I}_9)$ generando las instancias básicas de las cláusulas de Π y comprobando las que se ajustan a las condiciones de la Definición 5.20:

- Una instancia básica de C_1 es $q(a) \leftarrow p(a)$, pero no se cumple que $\{p(a)\} \subseteq \mathcal{I}_9$.
- Una instancia básica de C_1 es $q(b) \leftarrow p(b)$ y se cumple que $\{p(b)\} \subseteq \mathcal{I}_9$, por lo tanto $q(b) \in T_\Pi(\mathcal{I}_9)$.
- La cláusula $C_2 \equiv p(a) \leftarrow$ es básica. Es una cláusula incondicional, esto es, su cuerpo es \emptyset . Se cumple, trivialmente, que $\emptyset \subseteq \mathcal{I}_9$ y por lo tanto $p(a) \in T_\Pi(\mathcal{I}_9)$.
- Respecto a la cláusula C_3 , mediante un razonamiento similar al realizado en el último caso, podemos afirmar que $q(b) \in T_\Pi(\mathcal{I}_9)$.

Por consiguiente $T_\Pi(\mathcal{I}_9) = \{p(a), q(a), q(b)\}$.

Puede demostrarse que la aplicación T_Π es continua. Además, como establece la siguiente proposición, las H-interpretaciones que son modelo del programa Π pueden caracterizarse en términos del operador T_Π .

Proposición 5.7 Sea Π un programa definido e \mathcal{I} una H-interpretación de Π . Entonces, \mathcal{I} es modelo de Π si y solo si $T_\Pi(\mathcal{I}) \subseteq \mathcal{I}$.

Este resultado sugiere la importancia de T_Π en el cómputo de $\mathcal{M}(\Pi)$ y es la llave para la caracterización del modelo mínimo de Herbrand como el menor punto fijo de T_Π .

Teorema 5.6 (Caracterización por Punto Fijo de $\mathcal{M}(\Pi)$) *Sea Π un programa definido. Entonces, $\mathcal{M}(\Pi) = mpf(T_\Pi) = T_\Pi \uparrow \omega$.*

El anterior resultado $\mathcal{M}(\Pi) = mpf(T_\Pi)$ se cumple ya que, por la Proposición 5.5 (Teorema del Punto Fijo), $mpf(T_\Pi) = \inf\{\mathcal{I} \mid T_\Pi(\mathcal{I}) \subseteq \mathcal{I}\}$; ahora bien, por la Proposición 5.7, si $T_\Pi(\mathcal{I}) \subseteq \mathcal{I}$ entonces \mathcal{I} es modelo de Π ; esto es, $mpf(T_\Pi)$ es el menor de los modelos del programa Π y por lo tanto igual al modelo mínimo $\mathcal{M}(\Pi)$. Por otra parte, como por la Proposición 5.6 $mpf(T_\Pi) = T_\Pi \uparrow \omega$, tenemos que $\mathcal{M}(\Pi) = mpf(T_\Pi) = T_\Pi \uparrow \omega$.

Este resultado nos proporciona un procedimiento constructivo para el cómputo del modelo mínimo de Herbrand de un programa: basta con generar una secuencia de interpretaciones de Herbrand, concretamente las potencias ordinales de T_Π , hasta alcanzar el

menor punto fijo. Dado que para el retículo de las H-interpretaciones el ínfimo $\perp = \emptyset$, atendiendo a la Definición 5.19, la secuencia de cálculos comienza con $T_\Pi \uparrow 0 = \emptyset$. Informalmente, este procedimiento consiste en:

1. Añadir (todas las instancias básicas de) los hechos al conjunto de partida.
2. Aplicar las reglas a los hechos para generar nuevos hechos¹⁰, que se añaden al conjunto anterior.
3. Repetir las operaciones (1) y (2) hasta que no se obtengan hechos nuevos.

Ejemplo 5.15

Consideremos de nuevo el programa del Ejemplo 5.12.

$$\begin{aligned}
 T_\Pi \uparrow 0 &= \emptyset \\
 T_\Pi \uparrow 1 &= T_\Pi(\emptyset) \\
 &= \{p(a), p(b), s(c)\} \\
 T_\Pi \uparrow 2 &= T_\Pi(T_\Pi \uparrow 1) \\
 &= \{p(a), p(b), s(c)\} \cup \{r(a), r(b)\} \\
 &= \{p(a), p(b), s(c), r(a), r(b)\} \\
 T_\Pi \uparrow 3 &= T_\Pi(T_\Pi \uparrow 2) \\
 &= \{p(a), p(b), s(c), r(a), r(b)\} \cup \{t(a, a), t(a, b), t(b, a), t(b, b)\} \\
 &= \{p(a), p(b), s(c), r(a), r(b), t(a, a), t(a, b), t(b, a), t(b, b)\} \\
 T_\Pi \uparrow 4 &= T_\Pi(T_\Pi \uparrow 3) = T_\Pi \uparrow 3
 \end{aligned}$$

Se alcanza el punto fijo en la cuarta iteración y, por consiguiente,

$$\mathcal{M}_\Pi = \{p(a), p(b), s(c), r(a), r(b), t(a, a), t(a, b), t(b, a), t(b, b)\}.$$

Ejemplo 5.16

Consideremos de nuevo el programa del Ejemplo 5.13.

$$\begin{aligned}
 T_\Pi \uparrow 0 &= \emptyset \\
 T_\Pi \uparrow 1 &= T_\Pi(\emptyset) = \{p(a), q(b)\} \\
 T_\Pi \uparrow 2 &= T_\Pi(T_\Pi \uparrow 1) = \{p(a), q(b)\} \cup \{q(a)\} \\
 &= \{p(a), q(b), q(a)\} \\
 T_\Pi \uparrow 3 &= T_\Pi(T_\Pi \uparrow 2) = T_\Pi \uparrow 2
 \end{aligned}$$

¹⁰Desde un punto de vista práctico, esto equivale a la obtención de inferencias inmediatas aplicando la regla *modus ponens* a partir de los hechos obtenidos en la última iteración y de las instancias de las cláusulas del programa.

Se alcanza el punto fijo en la tercera iteración y, por consiguiente,

$$\mathcal{M}_\Pi = \{p(a), q(b), q(a)\}.$$

5.5.4. Equivalencia entre semánticas

La corrección de la resolución SLD asegura que un hecho (básico) que se derive de un programa Π es una consecuencia lógica de Π . El comentario anterior y la Proposición 5.5 aseguran que el conjunto de éxitos de un programa definido está contenido en el modelo mínimo de Herbrand del programa. El resultado inverso, debido a Apt y van Emden confirma la equivalencia entre la semántica operacional y la semántica declarativa por modelo mínimo de un programa Π .

Teorema 5.7 *Sea Π un programa definido. $\mathcal{EB}(\Pi) = \mathcal{M}(\Pi)$*

Una demostración de este resultado puede verse en [86], donde se aprecia el papel que juega T_Π como enlace entre la semántica operacional y la semántica declarativa de un programa Π .

Los resultados anteriores nos permiten utilizar una secuencia de potencias ordinales de T_Π para calcular también el conjunto de éxitos básicos de un programa.

5.6 SEMÁNTICAS NO ESTÁNDAR

5.6.1. Modularidad y Semánticas Composicionales

Una de las técnicas fundamentales en la ingeniería del software para tratar con la complejidad de los programas grandes es “estructurar” los programas en módulos que pueden luego componerse y reutilizarse. Los sistemas de módulos son una característica esencial de un lenguaje de programación que permite el desarrollo de bibliotecas de propósito general para dicho lenguaje.

Sin embargo, una de las mayores limitaciones que presenta la programación declarativa tiene que ver con la falta de mecanismos para llevar a cabo la estructuración y modularización de los programas. De hecho, el tratamiento de módulos se consensuó solo en la segunda parte del standard ISO Prolog (ISO/IEC 13211-2:2000) y hay que decir que, por su complejidad, el sistema de módulos de la mayoría de sistemas Prolog (con excepción de IF/Prolog) no se adapta a tal estándar sino que más bien resulta similar al modelo del viejo sistema Quintus-prolog, un estándar *de facto*. La inexistencia de un verdadero consenso ha conducido históricamente a que los sistemas de módulos sean muy dependientes del desarrollador o fabricante, dificultando esto el desarrollo de bibliotecas de código, en contraste con otros lenguajes como Java, por ejemplo.

Las dificultades de modularización en los lenguajes lógicos tienen múltiples causas, como la existencia de metapredicados y predicados sensibles al contexto. Pero la causa última del problema probablemente está en la ausencia de composicionalidad de la semántica.

La composicionalidad es una propiedad deseable que se ha reconocido como fundamental en la semántica de los lenguajes de programación al permitir explotar distintas extensiones y optimizaciones del lenguaje. Una semántica S es composicional, u homomórfica, con respecto a un operador de composición \star si el significado (la semántica) de la estructura compuesta $C_1 \star C_2$ puede obtenerse mediante la composición de los significados de los constituyentes, es decir, si existe un homomorfismo f_\star de tal forma que $S(C_1 \star C_2) = f_\star(S(C_1), S(C_2))$.

En particular, una semántica se llama *OR-composicional* si satisface la propiedad de que el significado de la unión de dos (sub-)programas puede obtenerse por combinación de la semántica de cada uno de ellos, i.e. $S(P_1 \cup P_2) = S(P_1) \sqcup S(P_2)$, usando un operador de composición \sqcup adecuado. La composicionalidad OR es importante para la programación con módulos y resulta interesante para el desarrollo de programas grandes, porque de esta forma la alteración de un módulo no supone recomputar desde cero la semántica del programa ni re-analizar el programa entero.

Pese a que parece natural pensar en programas (módulos) como conjuntos de cláusulas y en la noción de composición de programas como unión de conjuntos, no resulta tan simple determinar cómo podría definirse la semántica de los módulos si se pretende obtener una semántica OR-composicional con respecto a la unión de programas. De hecho, las semánticas estudiadas en este capítulo no son modulares, como demuestra el siguiente ejemplo.

Ejemplo 5.17

Sean los siguientes programas lógicos $\Pi_1 = \{p \leftarrow q.\}$, $\Pi_2 = \{q.\}$ y el programa que se obtiene como la unión de ambos: $\Pi_1 \cup \Pi_2 = \{p \leftarrow q. \quad q.\}$. El modelo mínimo para cada uno de estos programas es

$$\begin{aligned} M(\Pi_1) &= \emptyset \\ M(\Pi_2) &= \{q\} \\ M(\Pi_1 \cup \Pi_2) &= \{p, q\} \end{aligned}$$

Como se puede comprobar, se cumple que $M(\Pi_1 \cup \Pi_2) = \{p, q\} \neq \{q\} = M(\Pi_1) \cup M(\Pi_2)$.

Para superar este problema, en [89] se define una semántica basada en la idea de denotar cada programa lógico P por su operador de consecuencias inmediatas T_P , en vez de por el menor punto fijo de éste, i.e. $S(P) = T_P$, y demuestra la propiedad de composicionalidad de esta semántica con respecto a la unión y a la intersección de programas. Con esta denotación, se induce una relación de equivalencia muy fuerte entre programas: dos programas son equivalentes si y solo si las consecuencias inmediatas coinciden para toda interpretación de Herbrand como argumento.

5.6.2. Observable de las Respuestas Computadas

La semántica estándar de la programación lógica basada en el conjunto de los éxitos básicos sufre otro importante problema y es el ser incapaz de distinguir entre todos los programas que se comportan de forma distinta operacionalmente. Concretamente, es fácil encontrar dos programas con la misma semántica de éxitos básicos que, sin embargo, computan respuestas diferentes a un mismo objetivo y no podrían, por ejemplo, sustituirse uno por el otro en el contexto de un programa mayor sin afectar al comportamiento global del programa.

Ejemplo 5.18

Sean los siguientes programas lógicos $\Pi_1 = \{p(X).\}$, $\Pi_2 = \{p(a). p(X).\}$. Es inmediato comprobar que Π_1 y Π_2 tienen la misma semántica operacional de éxitos básicos, $\mathcal{EB}(\Pi_1) = \mathcal{EB}(\Pi_2) = \{p(a)\}$. Sin embargo, el objetivo $\leftarrow p(Z)$ computa en Π_2 dos respuestas, $\{Z/a\}$ y $\{\}$, mientras solo computa la respuesta vacía $\{\}$ en Π_1 .

Para distinguir entre programas que pueden aportar distintas respuestas computadas frente a un mismo objetivo, se ha propuesto una semántica más rica que la de éxitos básicos: la semántica de respuestas computadas, o *s*-semántica [50, 51, 19].

Definición 5.21 (Semántica de éxitos no básicos) *Sea Π un programa definido. La semántica de éxitos no básicos de Π , denotada por $\mathcal{ENB}(\Pi)$, es el conjunto de átomos $p(d_1, \dots, d_n)$ que se obtienen recogiendo las diferentes respuestas computadas $\{X_1/d_1, \dots, X_n/d_n\}$ de cada posible objetivo “plano” $\leftarrow p(X_1, \dots, X_n)$ que se puede formar con cada símbolo de predicado p del programa y tantas variables X_1, \dots, X_n distintas como indique su aridad n . Esto es,*

$$\mathcal{ENB}(\Pi) = \{p(X_1, \dots, X_n)\theta \mid \Pi \cup \{\leftarrow p(X_1, \dots, X_n)\} \vdash_{SLD}^{\theta} \square\}.$$

Ejemplo 5.19

Consideremos de nuevo los programas del Ejemplo 5.18. Podemos comprobar que la semántica de éxitos no básicos distingue perfectamente estos dos programas ya que, módulo renombramiento de las variables, $\mathcal{ENB}(\Pi_1) = \{p(Z)\}$ mientras que $\mathcal{ENB}(\Pi_2) = \{p(a), p(Z)\}$.

Observe que, a partir de la semántica de éxitos no básicos es inmediato obtener la semántica del modelo mínimo de Herbrand sin más que tomar todas las instancias básicas de los elementos del conjunto.

Una forma alternativa de definir la semántica de respuesta computadas consiste en tomar como observable las propias sustituciones que se obtienen como respuesta en vez de átomos. Esta idea conduce a la siguiente definición.

Definición 5.22 (Semántica de respuestas computadas) Sea Π un programa definido. La semántica de respuestas computadas de Π , denotada $\mathcal{RC}(\Pi)$, es el conjunto de las sustituciones que se obtienen como respuestas computadas para los objetivos “planos” que se pueden formar con los símbolos de predicado del programa. Esto es,

$$\mathcal{RC}(\Pi) = \{\theta \mid \Pi \cup \{\leftarrow p(X_1, \dots, X_n)\} \vdash_{SLD}^{\theta} \square\}.$$

Ejemplo 5.20

La semántica de respuestas computadas de los programas del ejemplo 5.18 es $\mathcal{RC}(\Pi_1) = \{\}\}$ y $\mathcal{RC}(\Pi_2) = \{\{Z/a\}, \{\}\}$, respectivamente.

Análogamente, podemos definir la semántica de un objetivo \mathcal{G} en Π como sigue

$$\mathcal{RC}_{\Pi}(\mathcal{G}) = \{\theta \mid \Pi \cup \{\mathcal{G}\} \vdash_{SLD}^{\theta} \square\}.$$

Otro tipo importante de composicionalidad en programación lógica es la composicionalidad AND. Una semántica es AND-composicional si satisface la propiedad de que la semántica de la conjunción de dos (sub-)objetivos se puede obtener por combinación de la semántica de cada uno de ellos, i.e. $S_{\Pi}(\leftarrow \mathcal{G}_1, \mathcal{G}_2) = S_{\Pi}(\leftarrow \mathcal{G}_1) \uparrow S_{\Pi}(\leftarrow \mathcal{G}_2)$, usando un operador de composición \uparrow adecuado.

Es fácil comprobar que todas las semánticas estudiadas en este apartado, incluyendo la semántica de respuestas computadas, tienen la propiedad de composicionalidad AND. En particular, para la s -semántica se utiliza como operador \uparrow la llamada *composición paralela de sustituciones* (u operador de reconciliación) que, dadas dos sustituciones θ_1 y θ_2 , consiste en calcular el u.m.g del conjunto de ecuaciones¹¹ $\widehat{\theta_1} \cup \widehat{\theta_2}$; en caso de que la unificación falle, se devuelve una sustitución distinguida τ que denota la imposibilidad de reconciliación. Esta operador se extiende a conjuntos de sustituciones de la forma obvia, calculando el conjunto que resulta de componer, dos a dos, las sustituciones de ambos conjuntos; en caso de ser conjuntos irreconciliables (es decir, si todas las unificaciones fallan) el resultado que se devuelve es el conjunto vacío. La composicionalidad AND permite hacer computaciones incrementales; es decir, si ya conocemos las respuestas a un subobjetivo, para obtener las respuestas a un objetivo mayor basta resolver la parte del objetivo todavía no resuelta y después componer paralelamente los dos conjuntos de respuestas.

Ejemplo 5.21

Para el programa Π_2 del Ejemplo 5.18 es posible calcular las respuestas del objetivo $\mathcal{G} \equiv \leftarrow p(Z), Z = c(a)$ como sigue. Como la única respuesta posible al objetivo $\leftarrow Z = c(a)$ es $\{Z/c(a)\}$, tenemos que $\mathcal{RC}_{\Pi_2}(\mathcal{G}) = \{\{Z/a\}, \{\}\} \uparrow \{Z/c(a)\} = \{Z/c(a)\}$.

¹¹ Aquí $\widehat{\theta}$ denota la *representación equational* de la sustitución θ , i.e., si $\theta = \{x_1/t_1, \dots, x_n/t_n\}$ entonces $\widehat{\theta} = \{x_1 = t_1, \dots, x_n = t_n\}$.

Para terminar, mencionaremos que en [20] se formalizó la primera semántica OR-composicional para la programación lógica que es correcta y completamente abstracta (*fully abstract*) para el observable de las respuestas computadas, la Ω -semántica. La propiedad de abstracción completa es la propiedad inversa de la corrección de una semántica y se cumple cuando programas equivalentes, i.e. con idéntica semántica, son indistinguibles desde el punto de vista de su comportamiento observable. La Ω -semántica es modular y capaz, además, de modelar el comportamiento operacional de los programas relativo al uso de variables lógicas. Su definición se basa en la idea general de la s -semántica (que, a diferencia de la semántica de éxitos básicos, sí es *fully abstract* pero, al igual que aquélla, no es OR-composicional) y se caracteriza por el hecho de incluir cláusulas (en vez de átomos) en las interpretaciones.

RESUMEN

Este capítulo se ha dedicado al estudio de los fundamentos de la programación lógica. Nos hemos centrado en la porción de la programación lógica que estudia los programas definidos, dado que los resultados teóricos para éstos son más satisfactorios y permiten introducir los conceptos más relevantes del área.

- La programación lógica, contrariamente a lo que sucede con la demostración automática de teoremas, se caracteriza por: i) deja de lado las cláusulas generales para focalizar en las cláusulas de Horn y de las cláusulas normales; ii) hace un uso muy limitado de las estrategias; iii) destierra el uso de la igualdad; y, más importante todavía, iv) se interesa por el uso de la lógica como lenguaje de programación y por la implementación eficiente del sistema de control que lo hace posible.
- Empleamos una notación especial para representar las cláusulas que denominamos notación clausal, que utiliza la implicación inversa. En la notación clausal los literales negativos se agrupan para formar el cuerpo de la cláusula y los positivos aparecen agrupados en la cabeza.
- Una cláusula de Horn (o cláusula definida) es una disyunción de literales de los cuales uno, como mucho, es positivo. Un programa definido es un conjunto de cláusulas definidas.
- La semántica operacional de los lenguajes lógicos se basa en un refinamiento del procedimiento de refutación por resolución lineal denominado resolución SLD [82]. Las siglas “SLD” hacen mención a las iniciales inglesas de “resolución Lineal con función de selección para programas definidos”.
- Llamamos regla de computación (o función de selección) φ a una función que, cuando se aplica a un objetivo G , selecciona uno y solo uno de los átomos de G .

- Formalmente, la regla de resolución SLD (usando la regla de computación φ) puede expresarse como sigue: Dado un programa definido Π y un objetivo \mathcal{G} ,

$$\begin{array}{c} (\mathcal{G} \Leftarrow Q_1 \wedge \mathcal{A}' \wedge Q_2), \varphi(\mathcal{G}) = \mathcal{A}', C \equiv (\mathcal{A} \leftarrow Q) < \Pi, \sigma = mgu(\mathcal{A}, \mathcal{A}') \\ \mathcal{G} \xrightarrow[\text{SLD}]{\sigma} \sigma(Q_1 \wedge Q \wedge Q_2) \end{array}$$

donde Q, Q_1, Q_2 representan conjunciones de átomos cualesquiera y el símbolo “ $<$ ” se ha introducido para expresar que C es una cláusula de Π que se toma estandarizada aparte. Una derivación SLD para $\Pi \cup \{\mathcal{G}\}$ es una secuencia

$$\mathcal{G} \xrightarrow[\text{SLD}]{\theta_1} \mathcal{G}_1 \xrightarrow[\text{SLD}]{\theta_2} \dots \xrightarrow[\text{SLD}]{\theta_n} \mathcal{G}_n$$

Si $\mathcal{G}_n \equiv \square$ decimos que la derivación es una refutación SLD y $\sigma = (\theta_n \circ \dots \circ \theta_1) \llcorner \text{Var}(\mathcal{G})$ es la respuesta computada en la refutación.

- Una derivación SLD para $\Pi \cup \{\mathcal{G}\}$ es una forma de resolución lineal de entrada, ya que las cláusulas laterales son cláusulas del programa Π (el conjunto de cláusulas de entrada).
- Contrariamente a lo que sucede en un paso de resolución lineal, en un paso de resolución SLD de la cláusula \mathcal{G}_i a la \mathcal{G}_{i+1} , no se emplea factorización (i.e., no se calcula un factor de la cláusula \mathcal{G}_i o de una cláusula de entrada para después usarlo en el paso de resolución) ni tampoco resolución con un ancestro (i.e., una cláusula central que haya sido derivada antes que \mathcal{G}_i).
- Un procedimiento de prueba por refutación puede entenderse como un proceso de búsqueda en un espacio de estados constituidos por los propios objetivos que se generan en cada paso de resolución. En el caso concreto de un programa definido Π y un objetivo definido \mathcal{G} , el conjunto de todas las posibles derivaciones SLD para $\Pi \cup \{\mathcal{G}\}$ se puede representar como un árbol: el árbol de búsqueda SLD.
- Cada nodo del árbol de búsqueda SLD es una cláusula objetivo que es el resolvente SLD de un nodo padre y una cláusula del programa. Los nodos que no pueden resolverse con ninguna cláusula del programa, i.e., no tienen hijos, se dice que son las hojas del árbol SLD. Los nodos hojas son o bien la cláusula vacía \square o nodos de fallo. Cada rama de un árbol SLD es una derivación SLD. Distinguimos tres tipos de ramas: infinitas, de éxito y de fallo.
- La regla de computación y la regla de ordenación (es decir, el orden en el que se seleccionan las cláusulas del programa para ser resueltas con los objetivos) determinan la forma del árbol de búsqueda SLD. Sin embargo, el conjunto de respuestas computadas por distintos árboles SLD (construidos con diferentes reglas de computación) para un mismo programa Π y objetivo \mathcal{G} es siempre el mismo. Esta propiedad se denomina independencia de la regla de computación en [86].

- La independencia de la regla de computación justifica que la selección de un literal en un objetivo se pueda realizar arbitrariamente sin que afecte al resultado de la computación. La posibilidad de selección arbitraria de los literales en un objetivo se denomina indeterminismo *don't care* en [81]. Contrariamente, la selección de la cláusula del programa que se utiliza en un paso de resolución es determinante a la hora de alcanzar el éxito en una derivación. Para asegurar la completitud, la selección de las cláusulas del programa debe hacerse de forma indeterminista *don't know*, es decir, explorando todas las alternativas posibles.
- Para buscar las ramas de éxito en un árbol SLD es importante la estrategia con la que se recorre dicho árbol. Hay dos estrategias de búsqueda fundamentales: búsqueda en anchura (*breadth-first*) y búsqueda en profundidad (*depth-first*).
- La semántica declarativa de los lenguajes de programación lógica se fundamenta en la semántica por teoría de modelos de la lógica de predicados. Dado que nuestro interés consiste en probar la insatisfacibilidad de un conjunto de cláusulas, basta restringirse a las interpretaciones de Herbrand para dar significado a las construcciones de nuestro lenguaje.
- La regla de resolución SLD es refutacionalmente correcta y completa para programas y objetivos definidos. Esto es, $\Pi \cup \{\mathcal{G}\}$ es insatisfacible si y solo si existe una refutación SLD para $\Pi \cup \{\mathcal{G}\}$.
- La contrapartida declarativa del concepto de respuesta computada es el concepto de respuesta correcta. Dado un programa definido Π y un objetivo definido $\mathcal{G} \equiv \leftarrow Q$, una respuesta θ para $\Pi \cup \{\mathcal{G}\}$ es cualquier sustitución para las variables de \mathcal{G} . Si la fórmula $\forall \theta(Q)$ es una consecuencia lógica de Π , decimos que θ es una *respuesta correcta* para $\Pi \cup \{\mathcal{G}\}$.
- El concepto de respuesta correcta permite una formulación alternativa de la corrección y completitud de la resolución SLD.
 - (Corrección) Toda respuesta computada para $\Pi \cup \{\mathcal{G}\}$ es una respuesta correcta para $\Pi \cup \{\mathcal{G}\}$.
 - (Completitud) Para toda respuesta correcta θ para $\Pi \cup \{\mathcal{G}\}$, existe una respuesta computada σ para $\Pi \cup \{\mathcal{G}\}$ que es más general que θ cuando nos restringimos a las variables de \mathcal{G} .
- Las semánticas formales, además de dar cuenta del significado de las construcciones generales del lenguaje, también deben asignar significado a los programas. Una forma de abordar este problema es asociando a cada programa cierta propiedad que denominamos observable. Por razones históricas y de simplicidad expositiva, estamos interesados en el observable de los éxitos básicos. Asignar significado formal a los programas permite establecer relaciones precisas de equivalencia entre los programas.

- La semántica operacional de un programa definido se caracteriza mediante el llamado conjunto de éxitos básicos, que es el conjunto de átomos básicos de la base de Herbrand derivables a partir del programa mediante resolución SLD.
- Asignamos significado declarativo a un programa seleccionando uno de sus modelos de Herbrand: el modelo mínimo, que es el modelo “más simple” de entre los posibles. Los programas definidos cumplen la propiedad de intersección de modelos, por la cual, dado un conjunto de modelos de Herbrand de un programa, la intersección de éstos es a su vez un modelo de Herbrand del programa. La propiedad de intersección de modelos proporciona un método de construir el modelo mínimo como la intersección de todos los modelos de Herbrand de un programa.
- Para programas definidos, la semántica operacional del conjunto de éxitos básicos y la semántica declarativa del modelo mínimo de Herbrand coinciden.
- Tanto la caracterización semántica operacional de un programa como la declarativa no son constructivas. Es posible dar una visión constructiva del significado de un programa siguiendo la teoría del punto fijo. La idea es asociar a cada programa un operador continuo, denominado operador de consecuencias inmediatas, que permita, mediante sucesivas aplicaciones, construir el modelo mínimo del programa.
- Pese a que parece natural pensar en programas (módulos) como conjuntos de cláusulas y en la noción de composición de programas como unión de conjuntos, la semántica de éxitos básicos de los programas lógicos no es modular (OR-composicional). En cambio sí goza de la propiedad de composicionalidad AND, lo cuál hace posible calcular de manera incremental las respuestas computadas de un objetivo compuesto en un programa dado.
- La semántica de los éxitos básicos de los programas lógicos puede no distinguir programas que calculan diferentes respuestas computadas. La *s*-semántica es la primera semántica correcta y completamente abstracta (aunque tampoco es modular) que modela el observable de las respuestas computadas de un programa.

CUESTIONES Y EJERCICIOS

Cuestión 5.1 Enumere y describa las principales diferencias entre la programación lógica y la demostración automática de teoremas (tal y como se caracterizó en los capítulos precedentes).

Cuestión 5.2 Una cláusula de Horn definida es:

- Una disyunción de átomos con todas las variables cuantificadas universalmente al principio de la fórmula.
- Una conjunción de literales con todas las variables cuantificadas universalmente al principio de la fórmula.
- Una disyunción de literales con todas las variables cuantificadas universalmente al principio de la fórmula y tal que, entre los literales, hay al menos un literal negativo.
- Una disyunción de literales con todas las variables cuantificadas universalmente al principio de la fórmula y tal que, entre los literales, hay al menos un literal positivo.

Cuestión 5.3 Indique cuál de las siguientes es una cláusula de Horn definida:

- | | |
|--|--|
| ▪ $\mathcal{A} \vee \neg \mathcal{B}_1 \vee \dots \vee \neg \mathcal{B}_n$. | ▪ $\mathcal{A} \wedge \neg \mathcal{B}_1 \wedge \dots \wedge \neg \mathcal{B}_n$. |
| ▪ $\neg \mathcal{A} \vee \mathcal{B}_1 \vee \dots \vee \mathcal{B}_n$. | ▪ $\neg \mathcal{A} \wedge \mathcal{B}_1 \wedge \dots \wedge \mathcal{B}_n$. |

Cuestión 5.4 Defina formalmente el concepto de paso de resolución SLD.

Cuestión 5.5 a) Explique en qué sentido la estrategia de resolución SLD es un refinamiento del principio de resolución de Robinson. b) Describa su modo de operación. c) La estrategia de resolución SLD ¿es completa?, ¿en qué contexto?

Ejercicio 5.6 Traduzca el conjunto de cláusulas del Ejercicio 4.22 a la notación habitual de la programación lógica y pruebe su insatisfacibilidad empleando resolución SLD.

Ejercicio 5.7 Halle el conjunto de respuestas computadas para los siguientes programas y objetivos:

1. $\Pi_1 \equiv \{p(X, Y) \leftarrow q(Y) \wedge r(X, Y). \quad q(a). \quad q(b). \quad r(X, X) \leftarrow q(X).\}, \quad \mathcal{G}_1 \equiv \leftarrow p(X, Y)$.
2. $\Pi_2 \equiv \{p(X, Y) \leftarrow q(X) \wedge q(Y). \quad q(a). \quad r(b).\}, \quad \mathcal{G}_2 \equiv \leftarrow p(Z_1, Z_2)$.
3. $\Pi_3 \equiv \{p(Y) \leftarrow q. \quad q \leftarrow r(Y). \quad r(0).\}, \quad \mathcal{G}_3 \equiv \leftarrow p(X)$.

Cuestión 5.8 a) Defina por inducción el concepto de árbol de búsqueda SLD para un programa Π y un objetivo \mathcal{G} . b) Clasifique los diferentes tipos de nodos y ramas que pueden aparecer en él.

Cuestión 5.9 Indique cuál de las siguientes afirmaciones referentes a un árbol SLD es cierta:

- Su geometría espacial depende de la regla de cómputo elegida pero no de la regla de ordenación.
- El coste de recorrerlo entero depende de la estrategia de búsqueda adoptada.
- Su geometría espacial depende de la regla de cómputo elegida y de la regla de ordenación.
- El número de ramas infinitas es independiente de la regla de cómputo.

Ejercicio 5.10 Considere el siguiente programa lógico

$$\Pi \equiv \{p(a) \leftarrow p(f(a)). \quad p(f(X)) \leftarrow p(X).\}$$

y el objetivo $\mathcal{G} \equiv \leftarrow p(Y)$. Construya el árbol de resolución SLD para $\Pi \cup \{\mathcal{G}\}$ y determine la naturaleza de sus ramas (es decir, ¿son de éxito, de fallo, o infinitas?).

Ejercicio 5.11 a) Represente el árbol de búsqueda para el programa

$$\{p(X, Y) \leftarrow q(X, Y). \quad p(X, Y) \leftarrow q(X, Z) \wedge p(Z, Y). \quad q(a, b). \quad q(b, c).\}$$

y el objetivo $\leftarrow p(a, Y)$. b) Indique el orden en el que se generan los nodos cuando se emplea una regla de búsqueda: i) en profundidad con vuelta atrás; ii) en anchura.

Cuestión 5.12 Dado un programa lógico Π y un objetivo \mathcal{G} , la propiedad de “independencia de la regla de computación” asegura que:

- todas las ramas del árbol de derivaciones SLD para $\Pi \cup \{\mathcal{G}\}$ son de éxito;
- el conjunto de respuestas correctas para $\Pi \cup \{\mathcal{G}\}$ no depende de la regla de computación;
- no existen ramas infinitas en el árbol de derivaciones SLD para $\Pi \cup \{\mathcal{G}\}$;
- ninguna de las anteriores.

Cuestión 5.13 Enumérense y defínanse los componentes que caracterizan un procedimiento de prueba por refutación SLD.

Ejercicio 5.14 Dado el siguiente programa lógico

$$\Pi \equiv \{\text{natural}(s(X)) \leftarrow \text{natural}(X). \quad \text{natural}(0).\}$$

y el objetivo $\mathcal{G} \equiv \leftarrow \text{natural}(X)$, determine el conjunto de respuestas correctas para $\Pi \cup \{\mathcal{G}\}$.

Cuestión 5.15 Dados un programa lógico Π y un objetivo G , “toda respuesta computada para $\Pi \cup \{G\}$ es una respuesta correcta para $\Pi \cup \{G\}$ ”, ¿es una consecuencia de la propiedad de corrección o de la de completitud del principio de resolución SLD?

Cuestión 5.16 Un intérprete para Programación Lógica con una regla de computación que selecciona los (sub)objetivos de izquierda a derecha y una estrategia de búsqueda en profundidad, ¿es correcto y completo?

Cuestión 5.17 Indique cuál de las siguientes afirmaciones es cierta para cualquier programa lógico Π :

- Cada respuesta correcta para Π es una respuesta computada para Π .
- Existe el mismo número de respuestas correctas y de respuestas computadas para Π .
- El número de respuestas correctas para Π es menor o igual que el número de respuestas computadas para Π .
- El número de respuestas correctas para Π es mayor o igual que el número de respuestas computadas para Π .

Cuestión 5.18 ¿Cuál es el significado operacional de un programa lógico?

Cuestión 5.19 La semántica operacional (conjunto de éxitos) de un programa lógico:

- Se ve afectada por la regla de computación escogida.
- Se ve afectada por el orden de las cláusulas en el programa.
- Depende del objetivo a resolver, puesto que la semántica del programa es el conjunto de las respuestas computadas para un objetivo dado.
- Es independiente de la regla de computación y del orden de las cláusulas en el programa.

Cuestión 5.20 Considere el programa lógico: $\Pi \equiv \{p(Y) \leftarrow ., p(a) \leftarrow .\}$, cuya semántica operacional (conjunto de éxitos básicos) $\mathcal{EB}(\Pi) = \{p(a)\}$. El objetivo $G \equiv \leftarrow p(X)$ tiene dos respuestas computadas con respecto a Π : $\theta_1 = \{X/Y\}$ y $\theta_2 = \{X/a\}$. Indique cuál de las siguientes afirmaciones es cierta:

- θ_1 es una respuesta computada pero no es una respuesta correcta para $\Pi \cup \{G\}$.
- Dado que $\theta_1(p(X)) \notin \mathcal{EB}(\Pi)$, θ_1 es una respuesta computada incorrecta para $\Pi \cup \{G\}$.
- θ_1 y θ_2 son ambas respuestas correctas para $\Pi \cup \{G\}$.

- θ_1 no es una respuesta computada por resolución SLD, ya que es más general que θ_2 .

Cuestión 5.21 Definase que se entiende por propiedad de intersección de modelos. Esta propiedad ¿es también válida para cláusulas generales o solo para cláusulas definidas?

Ejercicio 5.22 ¿Cuál es la semántica por teoría de modelos del programa lógico $\Pi = \{p(f(X))\}$?

Ejercicio 5.23 Dado el programa lógico $\{p(f(X)) \leftarrow q(X). \quad q(a) \leftarrow p(a).\}$, ¿cuál es el modelo mínimo de Herbrand?

Cuestión 5.24 Sean $\Pi_1 \equiv \{p(X) \leftarrow p(f(X)).\}$ y $\Pi_2 \equiv \{p(f(X)) \leftarrow p(X).\}$ dos programas lógicos. Indique cuál de las siguientes afirmaciones es cierta:

- Ya que $p(a) \in \mathcal{EB}(\Pi_1)$ y $p(a) \notin \mathcal{EB}(\Pi_2)$, tienen distinta semántica operacional.
- Ya que $p(a) \notin \mathcal{M}(\Pi_1)$ y $p(a) \in \mathcal{M}(\Pi_2)$, tienen distinta semántica declarativa.
- Estos dos programas lógicos no se pueden comparar entre sí porque su unión ($\Pi_1 \cup \Pi_2$) es un programa con dos cláusulas que definen una recursión circular.
- Π_1 y Π_2 son semánticamente equivalentes.

Ejercicio 5.25 Dado el programa lógico

$$\Pi \equiv \{p(X) \leftarrow p(f(X)). \quad p(a) \leftarrow p(X).\}$$

y la interpretación $\mathcal{I} = \{p(a), p(f(a)), p(f(f(a)))\}$, calcule $T_\Pi(\mathcal{I})$.

Ejercicio 5.26 Halle el menor punto fijo del operador de consecuencias inmediatas para los siguientes programas lógicos:

1. $\Pi_1 \equiv \{p(X) \leftarrow p(f(X)). \quad p(f(0)).\}$
2. $\Pi_2 \equiv \{p(Y) \leftarrow q. \quad q \leftarrow r(Y). \quad r(f(0)).\}$
3. $\Pi_3 \equiv \{nat(s(s(s(0))))). \quad nat(X) \leftarrow nat(s(X)).\}$
4. $\Pi_4 = \{p(X) \leftarrow p(f(X)). \quad p(f(X)).\}$
5. $\Pi_5 = \{p(X, f(a)). \quad p(X, Y) \leftarrow p(X, f(Y)).\}$

Cuestión 5.27 Sea $\mathcal{M}(\Pi)$ el modelo mínimo de Herbrand de un programa lógico Π y sea T_Π el operador de consecuencias inmediatas asociado. Indique cuál de las siguientes afirmaciones es falsa:

- $\mathcal{M}(\Pi)$ es el menor punto fijo de T_Π .
- $\mathcal{M}(\Pi) = T_\Pi \uparrow \omega$.
- $\mathcal{M}(\Pi)$ es el conjunto de éxitos básicos de Π : el conjunto de átomos sin variables $A \in \mathcal{B}_L(\Pi)$ tales que $\Pi \cup \{\leftarrow A\}$ tiene una SLD-refutación.
- $\mathcal{M}(\Pi)$ es el mínimo conjunto que contiene todas las consecuencias lógicas (básicas o no) de Π .

Cuestión 5.28 Sea el programa lógico $\Pi = \{p(X) \leftarrow q(f(X)), q(f(X)) \leftarrow .\}$, indique cuál de las siguientes afirmaciones es falsa:

- | | |
|-----------------------------------|--|
| 1. $\Pi \vdash \forall X p(X)$. | 3. $\Pi \models \forall X p(X)$. |
| 2. $p(X) \in \mathcal{EB}(\Pi)$. | 4. $p(f(a)) \in T_\Pi \uparrow \omega$. |

Ejercicio 5.29 Consideremos el siguiente fragmento de un programa lógico que contiene un único hecho $\{p(a, f(b))\}$ como definición de un predicado p . Calcule las respuestas computadas para el objetivo $\leftarrow p(X, Y), q(X, Y)$, en un programa que contiene la definición de p anterior y en el que la única respuesta computada para el objetivo $\leftarrow q(X, Y)$ sea la sustitución $\{X/f(a), Y/b\}$.

Cuestión 5.30 Sean los programas lógicos: $\Pi_1 \equiv \{p \leftarrow q.\}$, $\Pi_2 \equiv \{p \leftarrow r.\}$ y los programas que se obtienen como unión de Π_1 y Π_2 con un tercer programa $\Delta \equiv \{q \leftarrow .\}$. Esto es, $\Pi_1 \cup \Delta = \{p \leftarrow q, q \leftarrow .\}$ y $\Pi_2 \cup \Delta = \{p \leftarrow r, q \leftarrow .\}$. Razone, utilizando dichos programas, sobre la certeza de las siguientes afirmaciones:

- Para cualesquiera programas Π_1 y Π_2 , si $\mathcal{M}(\Pi_1) = \mathcal{M}(\Pi_2)$ entonces, para cualquier programa Δ , $\mathcal{M}(\Pi_1 \cup \Delta) = \mathcal{M}(\Pi_2 \cup \Delta)$.
- Existen programas Π_1 y Π_2 tales que $\mathcal{M}(\Pi_1) = \mathcal{M}(\Pi_2)$ y para algún programa Δ , $\mathcal{M}(\Pi_1 \cup \Delta) \neq \mathcal{M}(\Pi_2 \cup \Delta)$.
- Para cualesquiera programas Π_1 , Π_2 y Δ , $\mathcal{M}(\Pi_1 \cup \Delta) = \mathcal{M}(\Pi_2 \cup \Delta)$ si $\mathcal{M}(\Pi_1) = \mathcal{M}(\Pi_2)$.

El Lenguaje Prolog: Introducción

El lenguaje de programación *Prolog* (PROgramación en LOGica – Colmenauer et. al. 1973) es la realización más conocida de las ideas introducidas en el campo de la programación lógica. En éste y en el próximo capítulo abordamos algunos sus aspectos más esenciales.

Conviene mencionar que nuestro principal objetivo no es dar una descripción completa del lenguaje Prolog o lograr expertos en la programación con él, sino más bien presentar una serie de características del mismo que deben tenerse en cuenta para su correcto aprendizaje; con este fin, relizaremos en muchas ocasiones una aproximación a los conceptos dirigida por ejemplos. Para un estudio exhaustivo del lenguaje y sus aplicaciones remitimos al lector interesado al libro clásico de Clocksin y Mellish [30] y a los excelentes de Bratko [21], Sterling y Shapiro [132]. Algunas técnicas avanzadas como el uso de listas abiertas y listas diferencia, así como aspectos relativos a la terminación y verificación de programas, pueden encontrarse en [6], [56], [130], y [109]. Para conocer detalles acerca de la implementación de los intérpretes y compiladores Prolog, ver [3] y [131]. El manual de referencia sobre el ISO-Prolog estándar (parte 1, Prolog sin módulos) es [43].

6.1 PROGRAMACIÓN LÓGICA Y PROLOG

En esencia, un programa Prolog es un conjunto de cláusulas de Horn que, por motivos de eficiencia, se tratan como una secuencia (concretamente, la secuencia en que las fórmulas aparecen escritas textualmente en el programa). Dado un objetivo, el programa se ejecuta utilizando el mecanismo operacional de la resolución SLD. Para usar la programación lógica como un lenguaje de programación útil ha sido necesario mejorar sus prestaciones de manera que, en la práctica, Prolog se aleja de algunos de los supues-

tos formales de la programación lógica, tal y como han sido presentados en el capítulo precedente. El lenguaje Prolog incorpora:

1. Modificaciones en el mecanismo operacional para conseguir una mayor eficiencia en la ejecución de los programas:
 - a) eliminación de la regla de *occur check* en el procedimiento de unificación, por tratarse de una comprobación muy costosa y que, en la mayoría de los casos prácticos, no tiene ningún efecto;
 - b) la posibilidad de utilizar el predicado¹ de *corte* para reducir el espacio de búsqueda, mediante la poda de ramas que no conducen a las soluciones deseadas.
2. Aumento de la expresividad a través de una sintaxis extendida que permite el uso de:
 - a) las conectivas lógicas de disyunción y negación en el cuerpo de las cláusulas;
 - b) operadores predefinidos (*built-in*): aritméticos (e.g., “+”, “-”, “*”, “div”, y “/”), que dan acceso a las facilidades de cómputo intrínsecas del computador sobre el que está implementado el sistema Prolog; de comparación (e.g., “=:”, “<”, “>”, para comparar expresiones aritméticas) etc.;
 - c) predicados predefinidos para facilitar las operaciones de entrada y salida (e.g., “read” y “write”) y, por lo tanto, la necesaria comunicación del programa con el exterior;
 - d) predicados predefinidos para la manipulación de términos (e.g., “functor”, “=..”, “arg” y “name” que permiten el análisis y la descomposición de términos);
 - e) predicados predefinidos para la manipulación de cláusulas (e.g., el predicado “assert”, que permite insertar una nueva cláusula en el programa, en tiempo de ejecución, y el predicado “retract”, que permite eliminar una cláusula en el programa);
 - f) predicados predefinidos para la metaprogramación (e.g., “call”, predicado utilizado para invocar un determinado objetivo, y “clause”, predicado que permite analizar la estructura de las cláusulas de un programa). Estas facilidades suponen la extensión del lenguaje clausal con características que van más allá de las posibilidades de la lógica de predicados, al permitir cierto grado de programación de *orden superior*.

¹En la jerga de Prolog se da el nombre de “predicado” a cualquier tipo de relación, tanto si se trata de una verdadera relación matemática como un artificio para introducir facilidades de entrada/salida, o mecanismos de control.

En éste y en el próximo capítulo se discuten cada una de estas características y su influencia en el lenguaje. Primero se presenta un subconjunto de Prolog, denominado *Prolog puro* [132], que se ajusta a los ideales de la programación lógica tal y como fue descrita en el capítulo precedente. Después describiremos su extensión con las facilidades antes aludidas, intentando aclarar las diferencias con el ideal de la programación lógica “pura” cuando se considere necesario.

6.2 PROLOG PURO

En este apartado describimos un subconjunto de Prolog que esencialmente se corresponde con la visión de la programación lógica introducida en el Capítulo 5. Detallamos su sintaxis y las peculiaridades de su mecanismo operacional (en especial aquellas que lo alejan de la semántica operacional de la programación lógica), dejando algunas de sus principales técnicas de programación para el próximo apartado.

6.2.1. Sintaxis

La sintaxis de Prolog puro es sencilla. Las construcciones más simples del lenguaje son los términos (que se utilizan para representar los datos: `cero`, `sucesor(cero)`, `juan`, `[a, b]`, `1, 2...`) y los átomos (que se utilizan para expresar propiedades y relaciones entre los datos: `par(cero)`, `impar(sucesor(cero))`, `humano(juan)`, `esLista ([a, b])`, `doble_de(1, 2,...)`). Las instrucciones se codifican como cláusulas definidas. La única peculiaridad respecto a lo que ya conocemos hasta el momento es la notación que utiliza Prolog para designar las conectivas lógicas y el hecho de que las cláusulas deben terminar con un punto “.”. En la Tabla 6.1 se muestra la correspondencia entre la notación empleada en programación lógica y en Prolog. Por otro lado, se relaja la rigidez con la

Tabla 6.1 Conectivas lógicas y sintaxis de Prolog.

Nombre	Conectiva lógica	Símbolo	Sintaxis Prolog
Conjunción		\wedge	,
Disjunción		\vee	;
Implicación inversa (en una regla)		\leftarrow	<code>: -</code>
Negación		\neg	<code>not</code>
Implicación inversa (en una cláusula objetivo)		\leftarrow	<code>? -</code>

que la lógica de predicados designa los símbolos de constante, función, relación y los símbolos de variable, permitiéndose la utilización de identificadores. Un identificador es cualquier cadena resultante de la combinación de caracteres alfanuméricos (`a, ..., z`, `A, ..., Z`, `0, ..., 9`) y el subrayado “_”, con la única restricción de que los identificado-

res para las variables² deben comenzar por una letra mayúscula o subrayado, y los de los símbolos de constante, función, y relación por una letra minúscula³. Prolog permite constantes numéricas, en este caso se utiliza la notación habitual para representar números enteros y reales (con y sin signo). A las constantes no numéricas en Prolog se les denomina átomos⁴, si bien en este libro no usaremos esta terminología excepto si lo hacemos explícito. No hay que confundir una constante (átomo) de Prolog con un átomo o una fórmula atómica de la programación lógica, donde tiene el mismo significado que en lógica. Debido a que la mayoría de los sistemas Prolog no poseen tipos, la única forma de reconocer “la categoría” a la que pertenece un cierto objeto es por su forma sintáctica. En la Figura 6.1 se muestra una clasificación de los diferentes objetos que pueden construirse en el lenguaje Prolog.

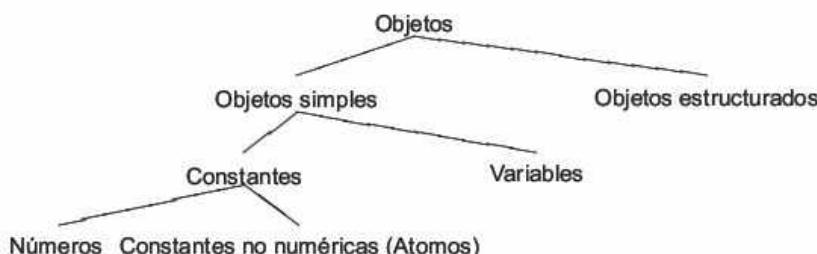


Figura 6.1 Clasificación de los objetos en Prolog.

El siguiente ejemplo ilustra las nociones introducidas hasta el momento.

Ejemplo 6.1

Si el lector conoce las reglas de construcción de términos y fbf's de la lógica de predicados (véase el Apartado 2.2.1) y sigue las peculiares convenciones notacionales de Prolog, no tendrá dificultad en escribir expresiones de Prolog sintácticamente correctas.

²Cuando se utiliza únicamente el símbolo “_” como identificador de una variable, decimos que la variable es *anónima*. Las variables anónimas reciben un tratamiento especial de los sistemas Prolog, ya que, los valores enlazados a estas variables mediante el proceso de unificación no se muestran como resultado de un cálculo. Por otra parte, las distintas ocurrencias de variables anónimas en una expresión, internamente, se tratan como variables diferentes sin conexión alguna. Desde un punto de vista lógico, se corresponden con variables cuantificadas existencialmente.

³Se aconseja al lector consultar el manual del sistema Prolog que esté usando para contrastar las posibles variaciones a esta regla. Adviértase también que muchas versiones de Prolog admiten nombres de constantes, formados únicamente por cadenas de caracteres especiales (+, -, *, /, <, >, =, :, ., &, ~ y ~) o bien identificadores encerrados entre comillas simples. Esto último es útil cuando se desea escribir una constante que comience por una letra mayúsculas, para representar, por ejemplo, el nombre de una persona (e.g., ‘Elena’).

⁴Possiblemente, el uso del nombre “átomo” para designar una constante (de la lógica de predicados) se haya tomado prestado del lenguaje Lisp (lenguaje para Procesamiento de LISTas – McCarthy et al., 1962). Los átomos son los constituyentes sintácticos más simples del lenguaje Lisp. En Lisp un átomo puede ser numérico o simbólico.

Así y todo, se enumeran una serie de ejemplos para clarificar algunos de los aspectos comentados.

1. Ejemplos de nombres de variable correctos son:

`_x, Actividad, X, Y, Numero, Nombre_predicado.`

Ejemplos de nombres de variable incorrectos son:

`[_x], Actividad*, x, &Y, Número, Nombre predicado.`

La razón de la incorrección de cada uno de estos ejemplos es la siguiente:

- en “[x]” y “Actividad*” los caracteres “[”, “]”, “*” no pueden formar parte de un identificador;
- en “x” el identificador comienza con una letra minúscula;
- “&y” no es correcto ya que el carácter “&” no puede ser parte de un identificador, además un identificador de variable debe comenzar con una letra mayúscula o el carácter de subrayado “_”;
- “Número” no es correcto ya que un identificador no puede contener una letra con tilde;
- finalmente, “Nombre predicado” no es correcto ya que un identificador no puede contener espacios en blanco.

2. Ejemplos de nombres de constante (o función o relación) correctos son:

`elegir_menu, nil, un_padre, esPrimo, es_variable,
nombre_predicado.`

Ejemplos de nombres de constante (o función o relación) incorrectos son:

`elegir/menu, [nil], 1_padre, es>Primo, var, name.`

La razón de la incorrección de cada uno de estos ejemplos es la siguiente:

- en “elegir/menu”, “[nil]” y “es>Primo” los caracteres “/”, “[”, “]” y “>” no pueden ser parte de un identificador;
- en “1_padre” el identificador comienza con un dígito;
- finalmente, “var” y “name” son palabras reservadas⁵ que corresponden a nombres de predicados predefinidos.

3. Ejemplos de nombres correctos de constantes numéricas son:

⁵Para una lista de palabras reservadas consultar el manual de la versión de Prolog que se esté usando.

2, 123, -3, 12.3 y -0.3.

Las tres primeras representan números enteros y las dos últimas números reales.

4. Ejemplos de términos sintácticamente correctos son:

```
elegir_menu(N), padreDe(antonio), suc(suc(cero)),  
fecha(5, nov, 2000), coordenada(X, Y).
```

Ejemplos de términos sintácticamente incorrectos son:

```
elegir/menu(N), es>Primo(7), var(X).
```

La razón de la incorrección de cada uno de estos ejemplos es la siguiente:

- en “elegir/menu(N)” y “es>Primo(7)” los caracteres “/” y “>” no pueden ser parte de un nombre de función;
- finalmente, en “var(X)” el identificador “var” es una palabra reservada utilizada para designar un nombre de predicado predefinido, así pues, “var(X)” es una fórmula átomica sintácticamente correcta, pero no un término.

5. Ejemplos de átomos (en el sentido de la lógica) sintácticamente correctos son:

```
esPadre(padreDe(antonio)),  
menor(coordenada(3, 6), Coordenada),  
anterior(Fecha, fecha(5, nov, 2000)),  
natural(suc(suc(cero))).
```

6. Ejemplos de reglas sintácticamente correctas son:

```
hermanos(H1, H2) :- padre(P, H1), padre(P, H2).  
hermanos(H1, H2) :- madre(M, H1), madre(M, H2).  
figura :- triangulo; circulo;  
cuadrado; rectangulo.
```

Observe que en la última regla se ha empleado la disyunción “;” para conectar los átomos que constituyen el cuerpo (lo cual es perfectamente válido en la sintaxis de Prolog). Esto puede verse como una notación abreviada del conjunto de cláusulas definidas:

```
{figura :- triangulo.      figura :- cuadrado.  
  figura :- circulo.       figura :- rectangulo.}.
```

Ciertamente, puede comprobarse que la última regla es una fórmula lógicamente equivalente a la conjunción de las cláusulas del conjunto.

Otra peculiaridad de Prolog es la jerga que emplea para referirse a las construcciones del lenguaje: es común denominar a los términos (de la lógica de predicados) *objetos*. Los objetos se dividen en *simples* o *estucurados*⁶. Los primeros son las constantes y las variables lógicas. Los últimos son los términos formados por funciones *n*-arias con $n > 0$. Sin embargo, dado que en Prolog la sintaxis de los términos y los átomos es idéntica excepto por el hecho de que el símbolo en la raíz corresponde a una función o a un predicado no existe en Prolog una diferenciación real entre ellos. De hecho, el lenguaje generaliza las nociones de función y predicado mediante el concepto de *operador*, que incluye ambas categorías de símbolos y se emplea con mucha mayor liberalidad que en las matemáticas, donde un operador es simplemente una función. Así en Prolog reciben el nombre de operador tanto los operadores aritméticos “+”, “-”, “*”, ..., que son operadores en el sentido matemático de la acepción, como los símbolos de relación y los constructores de datos introducidos por el usuario con ayuda de la directiva `: - op(Precedencia, Modo, Nombre)` —véase el Apartado 6.6—. Más aún, como veremos son también operadores algunos símbolos especiales como `!`, que designa el operador de corte y se inserta en una cláusula en pie de igualdad con los otros átomos que componen ésta, e incluso las propias conectivas lógicas lógicas de conjunción, disyunción o negación, lo cuál permite manipular como estructura cualquier construcción sintácticamente correcta del lenguaje, incluyendo las propias cláusulas que componen los programas.

Toda esta permisividad sintáctica se conoce como *sintaxis ambivalente* [6] y algunas de sus manifestaciones concretas en el lenguaje Prolog son:

- Se permiten “términos” en posiciones de “átomos”. Por ejemplo, observe la primera cláusula de la definición del operador `not` que se describe en el Apartado 7.2.3: `not(X) :- X, !, fail.`
- Se aceptan también “átomos”, e incluso cláusulas, en posiciones de “términos”. Por ejemplo, una invocación al predicado `assert` puede contener una cláusula como argumento, como se verá en el Apartado 7.4.2.
- Contrariamente a lo que sucede en la lógica de predicados, un símbolo de función o predicado puede utilizarse con diferentes aridades en un mismo programa Prolog. Se dice que el símbolo en cuestión es variádico o de aridad variable. Por ejemplo, la siguiente definición es sintácticamente correcta `p(a, q(a)) :- p(a).`
- Más aún, dos símbolos diferentes, uno de función y el otro de predicado, pueden tener idéntico nombre y la misma aridad en un mismo programa. Por ejemplo, `p(q(a)) :- q(p(a)).`

Obviamente, incluir en el lenguaje esta facilidad, que denominamos sintaxis ambivalente, tiene repercusiones en el mecanismo operacional de Prolog.

⁶Muchas veces, en lugar de hablar de “objetos estucurados”, hablaremos simplemente de “estruuras”.

Para finalizar este subapartado realizamos una reflexión acerca de la “utilidad” de los términos en el lenguaje Prolog. Es una discusión que cobra sentido cuando se interpretan los términos, dotándoles de significado, esto es, de “contenido”. Observe que los términos en Prolog no están definidos por cláusulas (como de hecho lo están las relaciones) y, por lo tanto, son “estructuras muertas” que no se evalúan. Es por este motivo que los términos suelen emplearse para representar informaciones y simular las estructuras de datos de otros lenguajes. Por ejemplo, en el lenguaje Pascal [145], el tipo registro es un tipo estructurado que resulta de la yuxtaposición de elementos de cualquier tipo (incluso estructurados) para obtener un tipo compuesto y que se declara de la forma siguiente:

```
type T = record
    s1: T1;  s2: T2; ...  sN: TN
end
```

donde cada uno de los identificadores `s1, s2, ...` son los nombres de los campos y los identificadores `T1, T2, ...` son los tipos que se asignan a los campos correspondientes. El tipo registro es útil para definir una entidad, persona u objeto, mediante una colección de sus características más representativas. Concretando, podemos declarar el tipo `persona` de la forma siguiente:

```
type persona = record
    apellidos: string;
    nombre: string;
    fecha_nacimiento: fecha;
    sexo: (varon, hembra);
    estado_civil: (soltero, casado,
                    viudo, divorciado)
end
```

donde el tipo `fecha` es a su vez un tipo estructurado declarado de la forma siguiente:

```
type fecha = record
    dia: 1..31;
    mes: 1..12;
    ano: 1..2500
end
```

En Pascal el registro `persona` está compuesto por cinco campos. En Prolog esta estructura puede representarse mediante un término formado por el símbolo de función `persona` dados los argumentos variables: `Apellidos`, `Nombre`, `Fecha_nacimiento`, `Sexo` y `Estado_civil`. Para poner énfasis en la similitud con la noción de registro del lenguaje Pascal, podemos escribir los argumentos en líneas separadas:

```
persona (Apellidos,
         Nombre,
         Fecha_nacimiento,
         Sexo,
         Estado_civil)
```

Una instancia concreta de persona sería:

```
persona (Albert,
         Luis,
         fecha(10,08,1980),
         varon,
         soltero)
```

Sin embargo, una diferencia entre la representación de estructuras mediante términos que utiliza el lenguaje Prolog, y los registros de Pascal es que el lenguaje Pascal obliga a una declaración explícita de tipos y, en particular, de los tipos de cada uno de los campos. En el lenguaje Prolog simplemente se listan los argumentos, sin especificar sus tipos, por lo que se permite asignar datos de cualquier tipo a esas variables. También, en el lenguaje Prolog, una estructura no necesita declararse previamente a su utilización. Por otra parte, el lenguaje Pascal utiliza operadores de selección para acceder a las informaciones almacenadas en un registro. Sin embargo, como se indica en el Ejercicio 6.18, el mecanismo de unificación del lenguaje Prolog permite acceder a las informaciones contenidas en un término mediante la unificación, que puede entenderse como un emparejamiento de patrones (bidireccional).

Resumiendo, los objetos estructurados (términos) se utilizan en Prolog para construir estructuras de datos y no para realizar cómputos; deben, por tanto, compararse con las estructuras de datos de otros lenguajes (convencionales). En cambio, los predicados y las relaciones se utilizan para computar; por consiguiente, son semejantes a los procedimientos de otros lenguajes.

6.2.2. Mecanismo operacional

El mecanismo operacional adoptado por los sistemas que implementan la versión pura del lenguaje Prolog es la estrategia de resolución SLD modificada con las siguientes peculiaridades:

1. La *regla de computación* de Prolog trata los objetivos como secuencias de átomos y siempre selecciona para resolver el átomo situado más a la izquierda dentro del objetivo.
2. Prolog intenta unificar el átomo seleccionado por la regla de computación con las cabezas de las cláusulas del programa considerándolas en el orden textual en el que éstas aparecen dentro del programa. Esto supone que la *regla de ordenación* toma las cláusulas de arriba hacia abajo. Contrariamente a lo que sucede en los procedimientos de prueba de la lógica de predicados, las cláusulas no se consideran como un conjunto sino como una secuencia.
3. Prolog utiliza una *regla de búsqueda* en profundidad con vuelta atrás (*backtracking*). Como se comentó en el Apartado 5.2.3, la principal característica de este

tipo de búsqueda es que se visitan los descendientes de un nodo del árbol de búsqueda antes de visitar cualquier otro nodo (hermano). Esto hace que se recorra primero la rama más a la izquierda del árbol de búsqueda antes de recorrer el resto de las ramas.

4. Prolog omite la comprobación o test de ocurrencia de variables (*occur check*) cuando realiza la unificación de dos expresiones. Esto se hace por motivos de eficiencia, dado que es escaso el número de ocasiones en las que se producen las condiciones para existir una ocurrencia de variables, pero puede conducir en ocasiones a resultados incorrectos. Es decir, la eliminación del *occur-check* hace perder la corrección general del sistema.

Para simular esta unificación sin *occur check*, basta modificar el algoritmo de unificación sintáctica de Martelli y Montanari eliminando la regla 6 de la Definición 4.13.

5. Como ya dijimos, Prolog permite sintaxis ambivalente. En particular, un mismo símbolo de función o predicado puede utilizarse con diferentes aridades en un mismo programa. La posibilidad de utilizar sintaxis ambivalente también conduce a la necesidad modificar el algoritmo de unificación. En este caso, debemos introducir la siguiente modificación en la Regla de fallo de la Definición 4.13 que impediría unificar dos expresiones encabezadas con distintos operadores o de distinta aridad:

$$\frac{\langle \{op_1(t_1, \dots, t_n) = op_2(s_1, \dots, s_m)\} \cup E, \theta \rangle}{\langle Fallo, \theta \rangle} \text{ si } op_1 \neq op_2 \vee n \neq m$$

Un *árbol de búsqueda de Prolog* es un árbol de búsqueda SLD en el que las reglas de computación, ordenación y búsqueda están fijadas en la forma especificada más arriba. Debido a que estas reglas están fijadas, para un programa Π y un objetivo G , el árbol de búsqueda de Prolog es único (salvo cambio de nombre de las variables). El árbol de la Figura 5.2 es un ejemplo de árbol de búsqueda de Prolog. En lo que sigue estudiamos con más detalle cómo explorar un árbol de búsqueda de Prolog.

Algoritmo 6.1 [Exploración en Profundidad]

Entrada: Un Programa $\Pi = \{C_1, \dots, C_k\}$ y un objetivo G .

Salida: Una respuesta computada σ y éxito o una condición de fallo.

Comienzo

Inicialización: $NuevoEstado = \langle G, id \rangle$; $PilaEstados = pilaVacia$;
 $apilar(NuevoEstado, PilaEstados)$;

Repetir

1. $\langle Q \equiv \leftarrow \mathcal{B} \wedge Q', \theta \rangle = desapilar(PilaEstados)$;
2. Si $Q \equiv \square$ entonces

Comienzo

Devolver $\sigma = \theta_{\mathcal{K}Var(G)}$ y éxito;

Si se desean más respuestas entonces continuar
 si no terminar;

Fin
3. Si no Comienzo

Para $i = k$ hasta 1 hacer

Comienzo

$C_i \equiv \mathcal{A}_i \leftarrow Q_i$;

Si $\theta_i = mgu(\mathcal{A}_i, \mathcal{B}) \neq \text{fallo}$ entonces

Comienzo

$NuevoEstado = \langle \leftarrow \theta_i(Q_i \wedge Q'), \theta_i \circ \theta \rangle$;

$apilar(NuevoEstado, PilaEstados)$;

Fin

Fin

Fin

Hasta que $PilaEstados \equiv pilaVacia$;

Devolver fallo.

Fin

En un árbol de búsqueda, cuando el átomo más a la izquierda de un nodo (objetivo) unifica con más de una regla, aparece una *ramificación* en el árbol de búsqueda y decimos que estamos en presencia de un *punto de elección*. Los algoritmos de exploración en profundidad generan en cada iteración todos los nodos hijos asociados al nodo padre que se está visitando. Después de generados los nodos hijos, éstos se exploran ordenadamente, comenzando por el que está más a la izquierda. El proceso se repite generando todos los hijos del nuevo nodo que se está explorando. En cambio, en un algoritmo de búsqueda en profundidad con vuelta atrás (*backtracking*), cuando se llega a un punto de elección, simplemente se genera el primer nodo hijo y se marca la regla siguiente del programa que se deberá emplear en un paso de resolución con el nodo padre que se está visitando. Esta marca se denomina *punto de vuelta atrás*. El algoritmo de vuelta atrás prosigue visitando el último nodo hijo generado y reproduciendo el proceso, ya comentado, que lleva a la generación de un nuevo nodo.

Tanto en los algoritmos de exploración como en los de vuelta atrás el efecto es que se recorre la rama más a la izquierda. Si en este recorrido se encuentra un nodo de fallo (i.e., un nodo sin descendientes), se retrocede para explorar el primero de sus hermanos. Si no quedan hermanos por explorar se retrocede al nivel inmediatamente anterior y se procede con los hermanos de su nodo padre. En un procedimiento de exploración el no-

do que se va a visitar ya está generado, mientras que en uno de vuelta atrás habrá que generarlo en ese momento, haciendo uso de la información contenida en el último punto de vuelta atrás almacenado. Los algoritmos de vuelta atrás son más eficientes ya que solo generan aquellos nodos que visitan, mientras que los algoritmos de exploración siempre generan algunos nodos hijos que (posiblemente) nunca se visitan y, por lo tanto, producen un gasto inútil de tiempo de cómputo y recursos. A pesar de este inconveniente, los algoritmos de exploración son relativamente simples y más fáciles de comprender. Por este motivo, es preferible la descripción del proceso de búsqueda mediante un algoritmo de exploración en lugar de usar uno de vuelta atrás.

El Algoritmo 1 realiza una exploración en profundidad de un árbol de búsqueda de Prolog y fija el procedimiento de prueba por refutación adoptado por los sistemas Prolog estándar. Si bien en los últimos tiempos han aparecido compiladores de Prolog, sin pérdida de generalidad y por simplificar consideraremos, en los ejemplos y en las discusiones que mantengamos a lo largo del texto, que el sistema Prolog utilizado es un intérprete. A este respecto, el Algoritmo 1 puede considerarse como el embrión de un intérprete para el lenguaje Prolog. En este punto es interesante hacer notar que, para que un programa Prolog pueda ejecutarse en un intérprete de Prolog, sus hechos y reglas deben de cargarse previamente en la *base de datos interna* del sistema Prolog (o *espacio de trabajo* —*workspace*—). Una vez efectuada dicha operación, el usuario puede plantear preguntas al sistema (objetivos), que éste resolverá haciendo uso del Algoritmo 1.

Observaciones 6.1

1. El Algoritmo 1 genera, parcialmente, un árbol de búsqueda en un espacio de estados. Un estado es un par $\langle \mathcal{G}, \sigma \rangle$ donde \mathcal{G} es un objetivo y σ una sustitución. La exploración del árbol comienza a partir del nodo raíz, que es el estado inicial $\langle \mathcal{G}, id \rangle$ de la computación (véase el párrafo final del Apartado 5.2.2).
2. En el Algoritmo 1, las metavariables Q , Q' y Q_i representan secuencias de átomos. Los procedimientos *apilar*, *desapilar* son los procedimientos típicos de manipulación de un pila.
3. Puesto que pueden existir ramas infinitas, este algoritmo no siempre termina. Para asegurar la terminación es imprescindible imponer una cota de profundidad máxima a la exploración del árbol. Por otro lado, aunque se encuentre una solución, el camino recorrido no tiene por qué ser de longitud mínima (véase la Figura 6.2).
4. El Algoritmo 1 se transforma en un algoritmo de exploración en amplitud sin más que realizar los cambios siguientes: i) en lugar de almacenar los estados en una pila se almacenan en una lista; ii) en lugar de generar los nodos hijos de un nodo padre en el orden indicado en el paso 3 (para $i = k$ hasta 1), se generan en orden inverso, i.e., para $i = 1$ hasta k ; iii) conforme se van generando los nuevos nodos

se añaden al final de la lista y iv) el estado a resolver se toma de la cabeza de dicha lista. De este modo, antes de resolver los objetivos de los nodos hijos se resuelven los objetivos de los hermanos del nodo padre. Así pues, se visitan todos los nodos de un nivel antes de visitar sus hijos.



Observe que el Algoritmo 1 va generando explícitamente el árbol de búsqueda de Prolog隐式 a medida que lo explora. La Figura 6.2 detalla cómo el Algoritmo 1 genera el árbol de búsqueda de Prolog para el programa II y el objetivo G del Ejemplo 5.7.

Concretamente la figura muestra la exploración del árbol de búsqueda de Prolog hasta alcanzar la primera respuesta. En cada iteración, el Algoritmo 1 visita un nodo y genera un punto de elección. En la cuarta iteración se visita el nodo $\leftarrow q(b, Y_2), p(Y_2, b)$ y se produce un **fallo** al intentar generar un nuevo punto de elección. Por consiguiente, en la quinta iteración, el algoritmo retrocede al anterior punto de elección y visita el nodo hermano, que resulta ser un nodo de **éxito**. Durante el proceso, la variable X se ha *enlazado* al valor a . Aunque a primera vista pueda parecer similar a la asignación propia de los lenguajes convencionales, el enlace de variables de la programación lógica difiere de aquélla en dos aspectos muy importantes:

1. Una vez asociado un valor a una variable mediante un enlace, ya no cambia en el curso de una derivación, i.e., la variable actúa como una variable matemática (*variable lógica*) que no puede tomar dos valores diferentes en el curso de una demostración. Piénsese en la forma de comportamiento tan diferente de las variables en los lenguajes de programación convencionales, que pueden alterarse en mitad de una computación mediante el empleo de la instrucción de asignación, que tiene carácter destructivo en general.
2. Es posible obligar al intérprete de Prolog a desenlazar una variable forzando la vuelta atrás hasta un nodo del árbol, que ocupe una posición por encima de aquélla en que se ligó. El usuario de un intérprete de Prolog puede forzar la vuelta atrás introduciendo en la línea de órdenes el operador “;”, seguido de nueva línea. En el ejemplo que ilustra la Figura 6.2, al forzar la vuelta atrás, todas las variables, incluida la variable X , se desenlanzan y se retorna al último punto de elección almacenado con el fin de proseguir la exploración. Entonces, la variable X se enlaza con el valor b .

La vuelta atrás también se produce cuando un objetivo falla⁷. En la Figura 6.2, el paso que se muestra en la quinta iteración es un ejemplo de este hecho. Nótese que, en una vuelta atrás como la anterior, no se pierden los enlaces de las variables que se obtuvieron antes del último punto de elección almacenado (al que se retorna). Por ejemplo, los

⁷El fallo se puede forzar usando el predicado predefinido `fail`, una impureza del lenguaje que se estudiará en el Apartado 7.1.2.

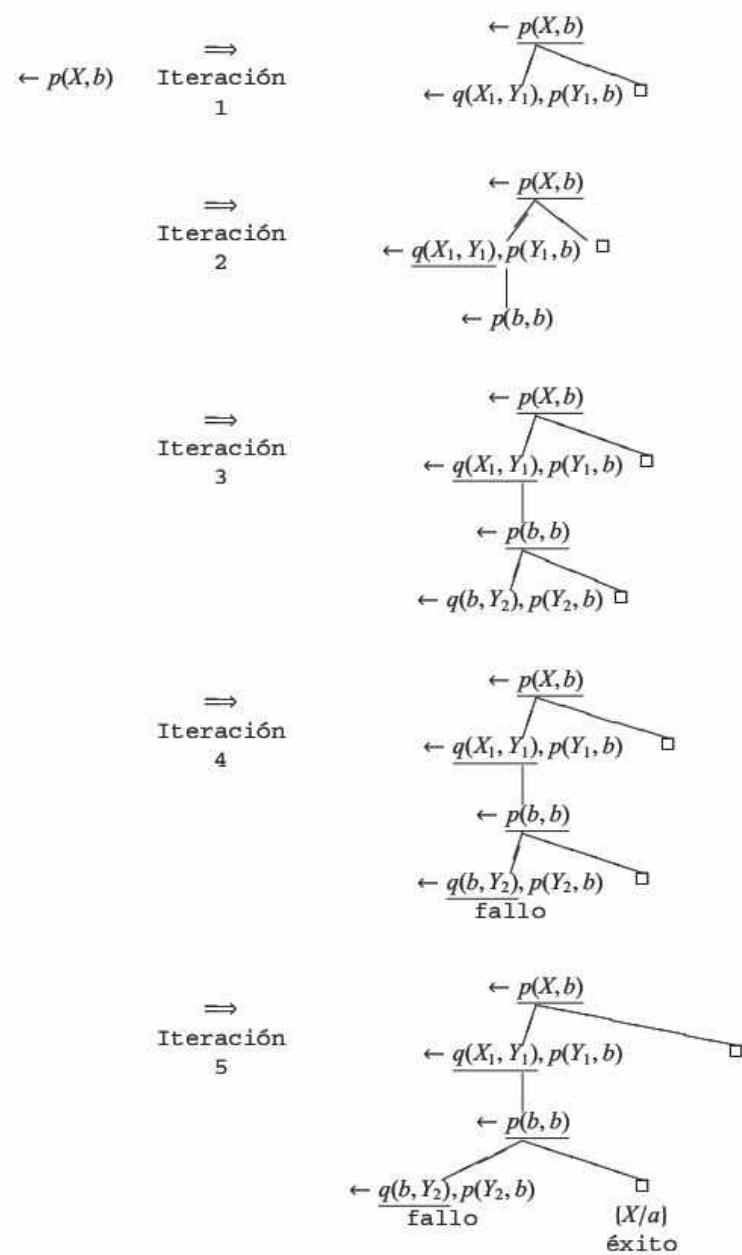


Figura 6.2 Exploración del árbol de búsqueda de Prolog de la Figura 5.2.

enlaces de las variables X, X_1, Y_1 y Z_1 no se pierden en nuestro caso, solo se pierden los enlaces de las variables X_2 y Z_2 obtenidos en el último paso de resolución (véase la

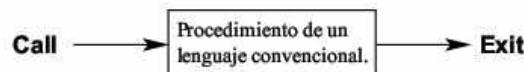
Figura 5.2). En general las variables que se desenlazan son aquéllas que se computaron en la porción de rama comprendida entre el objetivo que falla y el punto de elección al que se retorna.

6.2.3. Traza de la ejecución de un objetivo

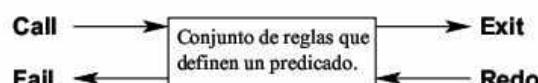
Terminamos este apartado con un breve inciso sobre el funcionamiento del predicado predefinido `trace` que utilizamos para estudiar el comportamiento operacional de los programas Prolog.

El predicado `trace` constituye una ayuda para la depuración de programas, al permitir la “traza” de la ejecución de un objetivo. En este contexto, “trazar la ejecución de un objetivo” debe de entenderse como un proceso que visualiza la información relativa a los pasos de resolución efectuados para satisfacer ese objetivo, incluyendo los valores de los argumentos y el éxito/fallo de los suobjetivos.

El procedimiento de búsqueda en profundidad con vuelta atrás, que constituye la base del mecanismo operacional del lenguaje Prolog puede verse como una generalización del mecanismo de llamada a procedimiento de los lenguajes convencionales. En la Figura 6.3(a) se muestra el diagrama de flujo resultante cuando se realiza una llamada a un procedimiento de un lenguaje convencional. Las flechas indican que el flujo de ejecución es unidireccional: el procedimiento se activa mediante una llamada (*call*) y termina mediante una operación de salida de procedimiento (*exit*). Después de que un procedimiento se ha ejecutado hasta completarse y ha terminado normalmente, no es posible su reactivación a no ser que se realice otra llamada.



a) Diagrama de flujo de una llamada a un procedimiento de un lenguaje convencional.



b) Diagrama de flujo de una llamada a un objetivo en el lenguaje Prolog.

Figura 6.3 Diagramas de flujo de ejecución de una llamada a procedimiento.

En el lenguaje Prolog hemos visto que las reglas y hechos que definen un predicado se comportan de manera similar a un procedimiento en un lenguaje convencional. Este “procedimiento” se pone en ejecución mediante la formulación de una pregunta (objetivo), que actúa a modo de llamada, y puede terminar con éxito, de forma similar a lo que sucede con un procedimiento de un lenguaje convencional. Pero el mecanismo de la vuelta atrás agrega un nuevo modo de terminación y un modo de reactivación (o resatifa-

ción): si un (sub)objetivo falla (*fail*) el control no pasa al siguiente (sub)objetivo más a la derecha; sino que termina con fallo, y acto seguido se devuelve nuevamente el control al (sub)objetivo, reactivándolo (*redo*) para intentar resatisfacerlo. La Figura 6.3(b) ilustra lo que acabamos de comentar.

Por consiguiente, una traza de un objetivo debe mostrar información relativa a las siguientes fases o acciones de ejecución:

- **Llamada (Call):** se muestra el nombre del predicado y los valores de sus argumentos en el momento en el que se invocó el (sub)objetivo.
- **Terminación con éxito (Exit):** si el (sub)objetivo se satisface y termina con éxito, entonces se muestra el valor de los argumentos que hacen el (sub)objetivo satisfactorio.
- **Terminación con fallo (Fail):** en este caso, simplemente se muestra una indicación de fallo.
- **Reactivación (Redo):** se muestra una nueva invocación del mismo (sub)objetivo, producida por el mecanismo de vuelta atrás, que de ese modo intenta resatisfacer el (sub)objetivo.

Ejemplo 6.2

Dado el programa `{p(X) :- q(X). q(uno). q(dos).}` y el objetivo “`?- p(X), q(X).`”, una sesión de depuración con un interprete de Prolog, podría tener el siguiente aspecto:

```
?- trace.
?- p(X), q(X).
<Spy>: (1:0) Call: p(_0). :
<Spy>: | (2:1) Call: q(_0). :
<Spy>: | (2:1) Exit: q(uno). :
<Spy>: (1:0) Exit: p(uno). :
<Spy>: (1:1) Call: q(uno). :
<Spy>: (1:1) Exit: q(uno). :
X = uno;
<Spy>: (1:0) Fail: q(uno). :
<Spy>: (1:0) Fail: q(uno). :
No
?- notrace.
```

El predicado `notrace` suspende el modo de traza en el que se entra cuando se ejecuta el predicado `trace`. Observe que el comando “`?- trace.`” solamente muestra información sobre el subobjetivo seleccionado en cada paso de resolución (pero no sobre el objetivo completo).

A partir de la traza anterior se puede reconstruir el árbol de búsqueda de Prolog para el programa y objetivo bajo consideración. Para ello basta tener en cuenta lo siguiente:

1. Cada mensaje de llamada (*Call*), en el que se muestra el (sub)objetivo (nombre del predicado y valores de sus argumentos) en el momento de la invocación, se corresponden con un nodo del árbol de búsqueda.
2. Los mensajes de terminación con éxito (*Exit*) dan información de los valores enlazados a las variables. Esta información se dispone en forma de etiqueta en las ramas del árbol de búsqueda.
3. Los mensajes de reactivación (*Redo*) señalan el (sub)objetivo (nodo del árbol) que se desea resatisfacer y constituyen el inicio de una nueva alternativa de cómputo.
4. Los mensajes de terminación con fallo (*Fail*), simplemente generan nodos de fallo⁸. Sin embargo, son indicativos de la necesidad de una vuelta atrás para buscar nuevas alternativas de cómputo.

Siguiendo estas pautas obtenemos el árbol de búsqueda que se muestra en la Figura 6.4.

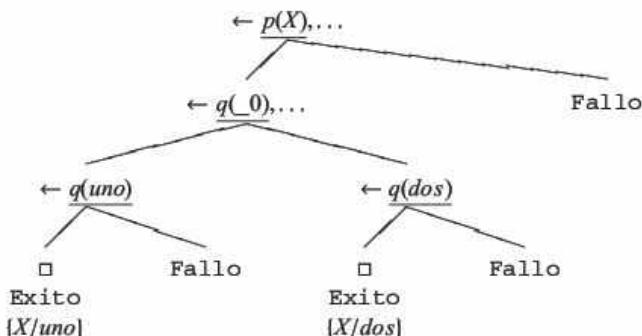


Figura 6.4 Árbol de búsqueda para el programa y objetivo del Ejemplo 6.2.

Para ejemplos realistas, una traza detallada como la que hemos mostrado en el ejemplo anterior puede carecer de valor práctico, debido a la gran cantidad de información que tiene que procesar el programador. Por este motivo, los sistemas Prolog suministran predicados capaces de producir una traza selectiva: `spy(P)` especifica el nombre de un predicado del que se quiere visualizar una traza (i.e., cuando se lanza un objetivo solamente se muestra información del predicado P); `nospy(P)` detiene el “espionaje” del predicado denominado P. Consulte el manual de su sistema Prolog para obtener más información sobre éstos y otros comandos para la depuración de programas Prolog, ya que el modo de operación de éstos es fuertemente dependiente del sistema del que se trate.

⁸Habitualmente, los nodos de fallo no se visualizan en las representaciones de los árboles de búsqueda.

6.3 PROGRAMACIÓN EN PROLOG PURO

En este apartado comentamos una serie de ejercicios de programación que ilustran algunas de las características más relevantes del estilo de programación declarativo del lenguaje Prolog.

6.3.1. Números naturales

El conjunto de los números naturales, $\mathbb{N} = \{0, 1, 2, \dots\}$, está formado por los números que usamos para contar. Los postulados de Peano componen el sistema de axiomas que caracterizan los números naturales. Estos postulados son [65]:

1. $0 \in \mathbb{N}$ (i.e., el 0 es un número natural).
2. Para todo número n , si $n \in \mathbb{N}$ entonces $s(n) \in \mathbb{N}$. El número natural $s(n)$ se denomina el *sucesor* de n .
3. Para todo número n , si $n \in \mathbb{N}$ entonces $s(n) \neq 0$.
4. Para todo par números naturales n y m , $s(n) = s(m)$ si y solo si $n = m$.
5. Para todo conjunto A de números naturales, si $0 \in A$ y $s(n) \in A$ siempre que $n \in A$, entonces todo número natural está contenido en A .

Podemos aprender bastante sobre las posibilidades de Prolog intentando expresar estos axiomas como cláusulas de un programa Prolog.

Axiomas primero y segundo

Los postulados primero y segundo definen el dominio de los números naturales como entidades que pueden construirse a partir de una constante “0” y un símbolo de función unario “*s*”, la función sucesor. Siguiendo la terminología habitual en la teoría de tipos de datos algebraicos, se dice que los símbolos “0” y “*s*” son los *constructores* del dominio de los números naturales. Esta definición puede traducirse directamente en el siguiente programa Prolog.

```
% natural(N), N es un natural
natural(cero).
natural(suc(N)) :- natural(N).
```

Hemos elegido el identificador “cero” para simbolizar la constante “0” y el identificador “*suc*” para simbolizar la función “*s*”. La primera cláusula del programa es un hecho y la segunda una regla. Este pequeño programa sirve, principalmente, para comprobar si un objeto es o no un número natural. El sistema Prolog responderá afirmativamente cuando se le planteen los siguientes objetivos “?- *natural(suc(cero))* .”, “?- *natural(suc(suc(cero)))* .”, ...; pero responderá “no” cuando se le plantea el objetivo “?- *natural(a)* .”.

Sin embargo, también es posible utilizar este programa para construir objetos del dominio de los naturales. Cuando se realiza la pregunta “?- `natural(X)`.” el sistema responde indicando todos los valores que puede tomar la variable `X`, es decir, `X = cero`, `X = suc(cero)`, `X = suc(suc(cero))`, ... Por lo tanto, el argumento del predicado `natural` actúa tanto como un parámetro de entrada como de salida. Esto último pone de manifiesto, como ya se ilustró en el Apartado 1.2.1, que en Prolog la relación de entrada/salida es *adirecional*. Esta propiedad también se denomina *propiedad de inversibilidad*. Aquí, y en casos similares, apreciamos que el algoritmo de unificación, implementado internamente en los sistemas Prolog, es una herramienta muy potente que sirve no solamente para comprobar la posibilidad de alcanzar la igualdad sintáctica de dos (o más) expresiones sino, también, para construir estructuras que son enlazadas a las variables como valores que resuelven el problema planteado.

Axiomas tercero y cuarto

Los axiomas tercero y cuarto pueden entenderse como la definición de una relación de igualdad entre los números naturales, que denominamos `eqNatural`. El tercer axioma enuncia la negación de un hecho y no puede representarse explícitamente. Como veremos, la negación en Prolog se interpreta por defecto (i.e., al no aparecer explícitamente la afirmación “`eqNatural(suc(N), cero)`” en el programa, se supondrá que la negación de este hecho es verdadera).

El axioma cuarto presenta la peculiaridad de que se formaliza en la lógica de primer orden empleando el bicondicional. Como ha señalado Kowalski, en [81], conviene desechar la parte “solamente-si” del bicondicional y quedarse con la parte “si”, cuando se representa la fórmula en notación clausal. Esto es debido a dos causas: por un lado, la parte “solamente-si” suele dar lugar a cláusulas “antinaturales” o poco prácticas a la hora de definir una relación⁹; por otro, la parte “si” es suficiente para derivar todas las instancias positivas de la relación que se está definiendo. Siguiendo esta recomendación, el axioma cuarto puede formalizarse en notación clausal como sigue:

```
eqNatural(suc(N), suc(M)) :- natural(N), natural(M),
eqNatural(N, M).
```

Para completar la definición del predicado `eqNatural` todavía es necesario introducir una nueva cláusula de programa que establezca que `cero` es igual a `cero`:

```
eqNatural(cero, cero).
```

El resultado de agrupar las dos últimas cláusulas es el siguiente programa:

```
% eqNatural(N, M), N y M son iguales
eqNatural(cero, cero).
eqNatural(suc(N), suc(M)) :- natural(N), natural(M),
eqNatural(N, M).
```

⁹Véase el Ejercicio 8.7 para constatar la veracidad de esta afirmación.

Axioma quinto

El quinto axioma es una formulación del principio de inducción matemática¹⁰. Este axioma contiene un cuantificador de segundo orden “Para todo conjunto A de números naturales”, que no es expresable en la lógica de predicados de primer orden y, por lo tanto, tampoco puede ser expresado en notación clausal.

Habitualmente, los postulados de Peano se completan con cuatro axiomas adicionales que definen la suma y el producto de números naturales. Estos axiomas son:

- Para todo número natural n , $0 + n = n$.
- Para todo número natural n y m , $s(n) + m = s(n + m)$.
- Para todo número natural n , $0 * n = 0$.
- Para todo número natural n y m , $s(n) * m = (n * m) + m$.

Las anteriores ecuaciones están definiendo la suma y el producto como funciones recursivas que devuelven un valor, la suma o el producto de dos números naturales. Debido a que una relación puede entenderse como una función booleana, que solo devuelve verdadero o falso, para devolver un resultado debemos hacerlo a través de uno de los parámetros del predicado que se está definiendo. Esto explica parte de la formalización de los anteriores axiomas mediante el siguiente programa Prolog, en el que la suma y el producto se definen como relaciones ternarias¹¹:

```
% suma(N, M, S), la suma de N y M es S
suma(cero, N, N) :- natural(N).
suma(suc(N), M, suc(S)) :- natural(N), natural(M), suma(N, M, S).

% producto(N, M, P), el producto de N y M es P
producto(cero, N, cero) :- natural(N).
producto(suc(N), M, P) :- natural(N), natural(M),
    producto(N, M, P1), suma(P1, M, P).
```

También aquí se aceptan objetivos que se contestan mediante respuestas de los tipos “si/no” (e.g., “?- `suma(suc(cero), suc(cero), suc(suc(cero)))`.”) y “?- `producto(suc(cero), suc(cero), suc(cero))`.”) y otros con *datos parcialmente definidos* (e.g. “?- `suma(N, M, suc(suc(cero)))`.”), que proporciona como respuesta los pares de números naturales N y M cuya suma es dos, cuando se fuerza la vuelta atrás introduciendo el operador “;” en la línea de órdenes del intérprete).

¹⁰Una formulación más habitual es la siguiente: dada una propiedad cualquiera P de los números naturales, si $P(0)$ y, para todo $m \in \mathbb{N}$, $P(m)$ implica $P(s(m))$ entonces, para todo $n \in \mathbb{N}$, $P(n)$. Por otra parte, nótese que una propiedad P para los números naturales define, implícitamente, el conjunto A de los número naturales que poseen esa propiedad.

¹¹En general, cualquier función n -aria puede expresarse como una relación $(n + 1)$ -aria, donde el argumento “extra” se usa para retornar el resultado devuelto por la función.

Observación 6.1

Los predicados definidos en el Apartado 6.3.1 son una muestra de definición inductiva (recursiva) basada en la estructura de los elementos del dominio de discurso, por lo que también recibe el nombre de definición por inducción estructural. En ella distinguimos un caso base, en el que se define la propiedad para el constructor cero, y un caso general, en el que se define la propiedad para un natural suc(N). Este estilo de definición de propiedades recibe el nombre de definición por ajuste de patrones en el contexto de la programación funcional. Los términos cero y suc(N) se denominan patrones. En general, un patrón es un término formado por símbolos constructores y variables.

Comprobaciones de dominio

Debido a que la mayoría de las implementaciones de Prolog no incorporan un sistema de tipos de datos, la comprobación de que los argumentos de un predicado corresponden al tipo de datos adecuado queda bajo la responsabilidad del programador.

Nótese que, debido a la naturaleza inherentemente recursiva de las definiciones en el lenguaje Prolog, en muchas ocasiones, comprobar el dominio al que pertenecen los argumentos de los predicados definidos puede llevar a un exceso de comprobaciones, lo cual afectará al rendimiento del programa. Por ejemplo, en el predicado suma se comprobará en cada llamada si N y M son naturales, lo que puede ser prohibitivo en términos de eficiencia cuando se manipulen grandes números. Aun peor, el lector puede experimentar por sí mismo que las comprobaciones de que N y M son naturales en los predicados suma y producto conducen a un funcionamiento no deseado de estos predicados en ciertas situaciones (e.g., el objetivo “?- suma(N, suc(cero), suc(suc(cero))).” entra en un bucle tras obtener la respuesta $N = suc(cero)$ – la única posible – si se fuerza la vuelta atrás mediante el operador “;”, cuando lo deseable sería que, una vez entregada dicha respuesta contestase “no” para indicar que no hay otras respuestas.).

Una solución evidente es eliminar esas comprobaciones cuando se trate de procedimientos que no forman parte del interfaz del usuario y, por lo tanto, tengamos la seguridad de que los argumentos que se van a suministrar a los predicados son los adecuados. Siguiendo esta simple receta, suministramos nuevas versiones para los predicados eqNatural, suma y producto que están libres de los problemas anteriormente comentados:

```
% eqNatural(N, M), N y M son iguales
eqNatural(cero, cero).

eqNatural(suc(N), suc(N)) :- eqNatural(N, M).

% suma(N, M, S), la suma de N y M es S
suma(cero, N, N).
suma(suc(N), M, suc(S)) :- suma(N, M, S).

% producto(N, M, P), el producto de N y M es P
producto(cero, N, cero) :- natural(N).
```

```
producto(suc(N), M, P) :- producto(N, M, P1), suma(P1, M, P).
```

Otra posible solución, para abordar el problema de la falta de eficiencia, es suministrar una definición que sirva de interfaz para el usuario y que sea la encargada de realizar las comprobaciones de tipo, para después llamar a una función auxiliar que se inhibirá de esta clase de comprobaciones. Por ejemplo, el predicado `suma` podría haberse definido como:

```
% suma(N, M, S), la suma de N y M es S
suma(N, M, S) :- natural(N), natural(M), sum(N, M, S).
sum(cero, N, N).
sum(suc(N), M, suc(S)) :- sum(N, M, S).
```

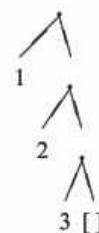
Sin embargo, esto no mejora su comportamiento operacional respecto al objetivo “?- `suma(N, suc(cero), suc(suc(cero)))`”. El problema es que las comprobaciones de tipo imponen una direccionalidad en los cómputos que rompen la *propiedad de inversibilidad* (adireccionalidad) de ciertos predicados. Nótese que esta definición de `suma` y la original tratan el primer argumento y el segundo como de entrada y el tercero como de salida, mientras que en el objetivo se está considerando el primer argumento como de salida y el resto de entrada.

6.3.2. Listas

En los próximos ejemplos que se van a comentar, es necesario el empleo de listas. Intuitivamente una lista es una secuencia de cero o más elementos. La representación interna de las listas puede tratarse como un caso especial de la representación de términos (mediante árboles binarios —véase más adelante—). En este contexto, una lista puede definirse inductivamente como sigue:

1. `[]` es una lista;
2. `.(X,L)` es una lista, si `X` es un elemento y `L` una lista; `X` se denomina la *cabeza* y `L` es la *cola* de la lista.

donde la constante “[]” representa la lista vacía de elementos y el símbolo de función “.” es un operador binario que añade un elemento al principio de una lista. Se dice que “[]” y “.” son los constructores del dominio de las listas. Es de destacar que la elección de los símbolos “[]” y “.” es completamente arbitraria y bien se podrían haber elegido otros; cualquier símbolo de constante y de función habrían sido válidos, e.g., “*nil*” y “*cons*”. Nuestra preferencia por los dos primeros radica en que algunos sistemas Prolog emplean esta notación como representación interna para las listas; e.g., SWI-Prolog [142]. Sin embargo, otros sistemas como AAIS-Prolog [1] no reconocen expresiones de la forma “`.(1,.(2,.(3,[])))`” y cualquier intento de introducir en un objetivo una expresión como la anterior dará lugar a un error, ya que AAIS-Prolog no reconoce el functor “.” como

**Figura 6.5 Una lista de enteros.**

constructor de listas. La Figura 6.5 muestra una lista representada como un árbol binario y el Ejemplo 6.3 proporciona algunos ejemplos de listas.

Ejemplo 6.3

Algunos ejemplos de listas son:

```

.(a, .(b, .(c, .(d, [ ]))),),
.(1, .(.(juan, .(pedro, [ ])), .(2, [ ]))),
.([ ], .(.(a, .(b, [ ])), .(.(1, .(2, .(3, [ ]))), [ ]))),,
.([ ], [ ])
  
```

Tal y como se ha definido la operación de concatenación permite añadir un elemento en cabeza de una lista de forma inmediata: si E es un elemento y L una lista, el resultado de la adición del elemento E a la lista L es el término $.(E, L)$. Cualquier otro tipo de operación sobre listas debe de ser definido.

Como se aprecia en el Ejemplo 6.3, la representación de las listas que utiliza el operador de concatenación “.” puede llegar a ser verdaderamente confusa. Por este motivo, el lenguaje Prolog proporciona al programador una notación alternativa para representar las listas, que no es otra cosa que azucar sintáctico con el que se endulza la poco apetitosa forma en la que se representan internamente las listas. En la notación alternativa, el símbolo de función “.” se sustituye por el operador infijo “[. | .]”. Utilizaremos en lo que sigue esta representación del constructor de listas, en vez de la más convencional “[_ | _]”, para hacer más explícito el hecho de que a la izquierda del símbolo | aparece un solo elemento (el primero de la lista) mientras a la derecha de la | hay una lista (la lista formada por todos los elementos excepto el primero).

Ejemplo 6.4

Desde el punto de vista del interfaz que presenta la nueva representación de las listas al programador, las listas del Ejemplo 6.3 pueden denotarse como:

```

[a| [b| [c| [d| [ ]]]], 
[1| [[juan| [pedro| [ ]]]| [2| [ ]]]], 
  
```

```
[[ ]|[ [a| [b| [ ]]] | [ [1| [2| [3| [ ]]]] | [ ]]]],  
[[ ]|[ ]]
```

Esta notación todavía no es lo suficientemente legible, por lo que se permiten las siguientes abreviaturas:

1. $[e_1| [e_2| [e_3| \dots [e_n| L]]]]$ se abrevia a $[e_1, e_2, e_3, \dots, e_n| L]$; y
2. $[e_1, e_2, e_3, \dots, e_n| []]$ se abrevia a $[e_1, e_2, e_3, \dots, e_n]$

Adviértase que, con esta nueva simplificación, la lista $[a| [b| [c| [d| []]]]]$ puede representarse como: $[a, b, c, d]$, $[a| [b, c, d]]$, o $[a, b, c| [d]]$. Además, una lista no tiene, necesariamente, que acabar con la lista vacía. En muchas ocasiones se utiliza una variable como patrón, que más tarde unificará con el resto de otra lista. Por ejemplo: en la lista $[a, b| L]$, la variable L es susceptible de unificar con el resto de cualquier lista cuyos dos primeros elementos sean a y b .

Ejemplo 6.5

Con las nuevas facilidades, las listas del Ejemplo 6.4 pueden denominarse como:

```
[a, b, c, d],  
[1, [juan, pedro], 2],  
[[ ], [a, b], [[1, 2, 3]]],  
[[ ], [ ]]
```

apreciándose una notable mejora en la legibilidad de estas expresiones.

Siguiendo la definición de lista dada al comienzo de este apartado y adoptando la notación recién introducida para su representación, es fácil definir un predicado `esLista` que confirme si un objeto es o no una lista.

```
% esLista(L), L es una lista  
esLista([ ]).  
esLista([_|R]) :- esLista(R).
```

Nuevamente hemos utilizado la técnica de definición por inducción estructural (con un caso base para el constructor `[]` y un caso general para el constructor `[|]`). En general, esta técnica se ajusta bien a la definición de propiedades sobre listas ya que éstas, como hemos visto, son estructuras inherentemente recursivas.

La mayoría de los sistemas Prolog proporcionan una buena colección de predicados predefinidos para la manipulación de listas. Dos de los más usuales son `member` y `append`:

- `member(E, L)`, esencialmente, sirve para comprobar si un elemento `E` está contenido en la lista `L`. La definición de este predicado es

```
% member(E, L), E esta en L
member(E, [E|_]). 
member(E, [_|L]) :- member(E, L).
```

Este predicado es inversible, de modo que puede emplearse para extraer los elementos de una lista.

- `append(L1, L2, L)`, concatena las listas `L1` y `L2` para formar la lista `L`. Una definición habitual¹² de este predicado es

```
% append(L1, L2, L), la concateacion de L1 y L2 es L
append([], L, L).
append([E|L1], L2, [E|L]) :- append(L1, L2, L).
```

Al igual que el predicado `member`, este predicado es inversible y admite múltiples usos (revise el Ejemplo 1.2).

Con la ayuda de los predicados `member` y `append` se puede definir una gran variedad de predicados, como se muestra en los siguientes subapartados.

Aplanar una lista

Vamos a definir la relación `aplanar(L, A)`, donde `L` es en general una lista de listas, tan compleja en su anidamiento como queramos imaginar, y `A` es la lista que resulta de reorganizar los elementos contenidos en las listas anidadas en un único nivel. Por ejemplo:

```
?- aplanar([[a, b], [c, [d, e]], f], L).
L = [a, b, c, d, e, f]
```

La versión que damos hace uso del predicado predefinido `atomic(X)`, que comprueba si el objeto que se enlaza a la variable `x` es o no un objeto simple “atómico” (i.e., un símbolo constante —un átomo en la jerga de Prolog—, un entero o una cadena). También se requiere utilizar el predicado predefinido `not`. Aquí y en los próximos apartados, emplearemos `not` con un sentido puramente declarativo, interpretandolo como la conectiva “`¬`” y sin entrar en el tratamiento especial que el lenguaje Prolog realiza de la negación.

¹²Sin embargo, esta definición no se comporta bien en todos los casos, ya que para un objetivo “?- `append([a], a, L)`.” se obtiene como respuesta “`L = [a|a]`”, que no es una lista, dado que “`a`” no lo es. Una forma de solucionar este problema es comprobar que, en efecto, en el segundo argumento del predicado `append` se introduce siempre una lista. Para ello basta con modificar las reglas que definen el predicado `append`:

```
append([], L, L) :- esLista(L).
append([X|L1], L2, [X|L]) :- esLista(L2), append(L1, L2, L).
```

Este hecho vuelve a poner de manifiesto que en un lenguaje sin tipos es el programador el encargado de comprobar que cada objeto que se utiliza es del tipo (dominio) adecuado.

```
% aplanar(L, A), A es el resultado de aplanar L
aplanar([], []).
aplanar([X|R], [X|P]) :- atomic(X), aplanar(R, P).
aplanar([X|R], P) :- not atomic(X), aplanar(X, P_X),
    aplanar(R, P_R), append(P_X, P_R, P).
```

La solución anterior indica que la lista vacía ya está aplanada (primera cláusula). Por otra parte, para aplanar una lista genérica $[X|R]$, se distinguen dos casos (segunda y tercera cláusula, respectivamente):

1. Si el elemento en cabeza x es atómico, ya está aplanado; por lo que basta con ponerlo en cabeza de la lista plana y seguir aplanando el resto R de la lista que se está aplanando.
2. Si el elemento en cabeza x no es atómico, habrá que aplanarlo; después se aplana el resto R de la lista original y finalmente se concatenan las listas resultantes del proceso anterior.

Último elemento de una lista

El elemento en cabeza de una lista es directamente accesible. Sin embargo, para acceder al último de una lista, es preciso introducir un predicado $\text{ultimo}(U, L)$, donde U es el último elemento de la lista L . Comúnmente este predicado se define como:

```
% ultimo(U, L), U es el ultimo elemento de la lista L
ultimo(U, [U]).
ultimo(U, [_|R]) :- ultimo(U, R).
```

La lectura declarativa de este predicado es clara: U es el último elemento de la lista $[_|R]$, si U es el último elemento de la lista remanente R ; en caso de que la lista contenga un solo elemento, es decir, la lista sea de la forma $[U]$, es un hecho que U es el último elemento. Por otra parte, el efecto procedural de este predicado consiste en recorrer toda la lista, desechariendo cada uno de los elementos, hasta alcanzar el último.

El predicado $\text{ultimo}(U, L)$, también puede definirse usando la relación append :

```
% ultimo(U, L), U es el ultimo elemento de la lista L
ultimo(U, L) :- append(_, [U], L).
```

Como es apreciable, se obtiene un resultado de una concisión sorprendente. La idea que subyace en esta definición es que la lista L es la concatenación, a una lista que contiene todos los elementos salvo el último, de la lista $[U]$ formada por el último elemento. La definición del predicado append y el algoritmo de unificación hacen el resto.

Operaciones con sublistas

El predicado `append` también puede utilizarse para definir operaciones con sublistas. Si consideramos que un prefijo de una lista es un segmento inicial de la misma y que un sufijo es un segmento final, podemos definir estas relaciones con gran facilidad haciendo uso del predicado `append`.

```
% prefijo(P, L), P es un prefijo de la lista L
prefijo(P, L) :- esLista(L), append(P, _, L).
% sufijo(P, L), P es un sufijo de la lista L
sufijo(P, L) :- esLista(L), append(_, P, L).
% sublista(S, L), S es una sublista de la lista L
sublista(S, L) :- prefijo(S, L1), sufijo(L1, L).
```

La última definición afirma que `S` es una sublista de `L` si `S` está contenida como prefijo de un sufijo de `L`.

Invertir una lista: parámetros de acumulación y recursión de cola

La relación “invertir una lista” se define de forma directa en términos del predicado `append`:

```
% invertir(L,I), I es la lista que resulta de invertir L
invertir([],[]).
invertir([H|T],L) :- invertir(T,Z), append(Z,[H],L).
```

El problema con esta versión es que es muy ineficiente debido a que destruye y reconstruye la lista original. Un examen del predicado `invertir(L,I)` revela que el número de llamadas al operador “[. | .]” es cuadrático con respecto al número de elementos de la lista que se está invirtiendo, si también contamos las llamadas al operador “[. | .]” producidas por una llamada al predicado `append`. Para eliminar las llamadas a `append` y lograr una mayor eficiencia se hace necesario el uso de un *parámetro de acumulación*. Un parámetro de acumulación es un argumento de un predicado que, como indica su nombre, se utiliza para almacenar resultados intermedios. En el caso que nos ocupa, se almacena la lista que acabará por ser la lista invertida, en sus diferentes fases de construcción.

```
% invertir(L,I), I es la lista que resulta de invertir L
% Usando un parametro de acumulacion.
invertir(L,I) :- inv(L, [], I).
% inv(Lista, Acumulador, Invertida)
inv([], I, I).
inv([X|R], A, I) :- inv(R, [X|A], I).
```

Ahora, la lista invertida se va construyendo en cada llamada al predicado `inv`, adicionando un elemento `X` a la lista acumulada en el paso anterior mediante el empleo del

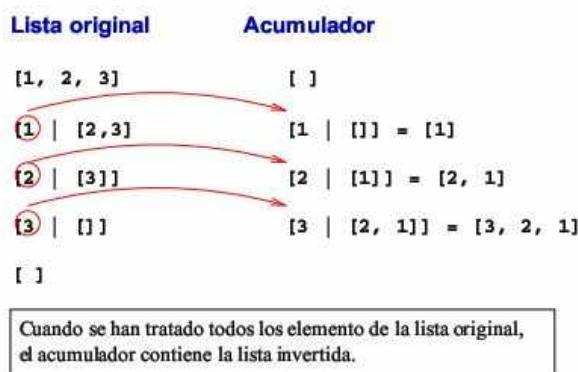


Figura 6.6 Fases en la inversión de una lista usando un parámetro de acumulación.

operador “[. | ..]”, en lugar de emplear el predicado `append`. La Figura 6.6 ilustra este proceso.

Un examen de la nueva versión revela que el número de llamadas a “[. | ..]” es lineal con respecto al número de elementos de la lista que se está invirtiendo. Por otra parte, tenemos ahora una definición recursiva que, como *última acción*, realiza una llamada a sí misma. En este caso, algunos compiladores de Prolog detectan que no es necesario almacenar el estado del predicado `inv` en la pila del sistema Prolog (el denominado `STACK`), ya que la llamada aparece en último lugar. Consiguiendo implementar una definición recursiva con la eficiencia de un procedimiento iterativo¹³. Este tipo de recursión se denomina “*recursión de cola*” (*tail-recursion*) y la optimización que hemos descrito (es decir, no almacenar en el `STACK` el entorno de la llamada) se conoce como “*optimización de última llamada*” (*last call optimization*). Para que el compilador pueda realizar este tipo de optimización es necesario que la llamada aparezca como último subobjetivo en el cuerpo de la cláusula, y que no haya puntos de *backtracking* previos pendientes. Cuando se resuelve un problema usando parámetros de acumulación es común obtener soluciones que cumplen estos requisitos.

Hemos aplicado la técnica de los parámetros de acumulación para optimizar el cálculo de una lista invertida, pero ésta es una técnica general que puede aplicarse a otros muchos problemas (e.g., el cálculo eficiente de los números de Fibonacci —Véase el Ejercicio 6.21 [opción 2]—).

Observaciones 6.2

1. Nuevamente hacemos notar que, en muchas ocasiones, comprobar los dominios de los argumentos de los predicados definidos puede afectar el rendimiento de un

¹³En realidad, este tipo de recursión se compila a código máquina exactamente igual que un bucle iterativo de un lenguaje convencional como Pascal o C.

programa. Por ejemplo, en el predicado `sublista` se comprobará en cada llamada si `L1` y `L` son listas, lo que puede ser prohibitivo en términos de eficiencia. Una primera medida en busca de la eficiencia es eliminar las comprobaciones. De este modo la definición de los predicados `prefijo` y `sufijo` quedarían así:

```
% prefijo(P, L), P es un prefijo de la lista L
prefijo(P, L) :- append(P, _, L).
% sufijo(P, L), P es un sufijo de la lista L
sufijo(P, L) :- append(_, P, L).
```

2. Por otro lado, la llamada indirecta a otros predicados en el cuerpo de una regla, también puede afectar al rendimiento. Por este motivo, puede ser aconsejable sustituir la llamada por el cuerpo del predicado al que se llama. Para fijar ideas, considérese la definición del predicado `sublista`

```
% sublista(S, L), S es una sublistas de la lista L
sublista(S, L) :- prefijo(S, L1), sufijo(L1, L).
```

En ella podemos reemplazar la llamada “`prefijo(S, L1)`” por el cuerpo de una instancia de la regla que define el predicado `prefijo`. Dicha instancia debe cumplir el requisito de que su cabeza coincida con la llamada que se pretende reemplazar. En otras palabras, la instancia de la regla sería “`prefijo(S, L1) :- append(S, _, L1)`” y se sustituiría “`prefijo(S, L1)`” por “`append(S, _, L1)`” en la definición del predicado `sublista`. Asimismo, se sustituiría “`sufijo(L1, L)`” por “`append(_, L1, L)`”. Operando de este modo obtendríamos la siguiente definición para el predicado `sublista`

```
% sublista(S, L), S es una sublistas de la lista L
sublista(S, L) :- append(S, _, L1), append(_, L1, L).
```

menos clara pero indudablemente más eficiente. El proceso de transformación que acabamos de describir es un caso particular de una transformación que se conoce como *desplegado* (unfolding). Esta es una de las técnicas básicas empleadas en el campo de la transformación automática de programás lógicos [88, 115, 117]. Volveremos sobre estas técnicas con mayor detalle en el Capítulo 10.

6.3.3. Conjuntos

Intuitivamente, un *conjunto*¹⁴ es una colección de elementos. Habitualmente, los elementos de un conjunto comparten alguna(s) propiedad(es) en común. Por ejemplo,

¹⁴En el Apéndice A se suministra más información sobre los conjuntos.

el conjunto de los números naturales pares $\{0, 2, 4, 6, \dots\}$ o el conjunto de los números naturales múltiplos de cinco $\{0, 5, 10, 15, \dots\}$. En un conjunto los elementos no aparecen repetidos y, contrariamente a lo que sucede con una secuencia de elementos, el orden de los mismos no es relevante. Por tanto, $\{a, b, c\}$ y $\{a, c, b\}$ son el mismo conjunto. De forma simple, los conjuntos pueden representarse mediante listas en las que no se repite ningún elemento. Si adoptamos esa convención, algunas operaciones típicas sobre conjuntos, implementadas usando el lenguaje Prolog, son las siguientes:

- El predicado `conjunto(C)` es verdadero cuando `C` es un conjunto y sirve para comprobar que un objeto que se pasa, en forma de lista, como argumento de una relación pertenece al dominio de los conjuntos.

```
% Tipo de datos Conjunto
repeticiones([X|R]) :- member(X, R), repeticiones(R).
% conjunto(C), la lista C es un conjunto
conjunto(C) :- not repeticiones(C).
```

El predicado `repeticiones(L)` comprueba la existencia de elementos repetidos en la lista `L`. Tiene éxito si un elemento se encuentra en la lista `L` más de una vez.

- El predicado `en(X, C)` define la relación de pertenencia de un elemento `X` a un conjunto `C`.

```
% en(X, C), X pertenece al conjunto C
en(X, C) :- conjunto(C), member(X, C).
```

- El predicado `subc(A, B)` define la relación de inclusión entre conjuntos.

```
% subc(A, B), A es subconjunto de B
subc(A, B) :- conjunto(A), conjunto(B), subc1(A, B).
subc1([], _).
subc1([X|R], C) :- member(X, C), subc1(R, C).
```

Para aumentar la eficiencia, después de comprobar que los objetos `A` y `B` que se pasan como argumentos al predicado `subc` son conjuntos, se hace uso del predicado auxiliar `subc1` que ya no realiza este tipo de comprobaciones. Como ya se ha mencionado, esta es una buena práctica de programación.

- El predicado `dif(C1, C2, D)` define la operación diferencia de conjuntos. El conjunto diferencia `D`, está formado por los miembros de `C1` que no están en `C2`.

```
% dif(C1, C2, D), D es el conjunto diferencia de C1 y C2
dif(C1,C2,D) :- conjunto(C1), conjunto(C2), dif1(C1,C2,D).
dif1(C1, [], C1).
dif1([], _, []).
dif1([X|R], C2,D) :- member(X,C2), dif1(R,C2,D).
dif1([X|R], C2, [X|D1]) :- not member(X,C2), dif1(R,C2,D1).
```

- Teniendo en cuenta que la unión de dos conjuntos C_1 y C_2 es el conjunto formado por los elementos que pertenecen a C_1 o a C_2 , se define el predicado `union(C1, C2, U)` en los siguientes términos:

```
% union(C1, C2, U), U es la union de C1 y C2
union(C1,C2,U) :- conjunto(C1),conjunto(C2),uni(C1,C2,U).
uni([],[],U).
uni([ ],U,U).
uni([X|R],C2,U) :- member(X, C2), uni(R, C2, U).
uni([X|R],C2,[X|U]) :- not member(X, C2), uni(R,C2,U).
```

- La intersección de dos conjuntos C_1 y C_2 es el conjunto formado por los elementos que pertenecen a C_1 y a C_2 , se define el predicado `intersec(C1, C2, I)` en los siguientes términos:

```
% intersec(C1, C2, I), I es la intersección de C1 y C2
intersec1(C1,C2,I) :- conjunto(C1), conjunto(C2),
                     inter1(C1, C2, I).
inter1(_, [], []).
inter1([], _, []).
inter1([X|R],C2,[X|I]) :- member(X,C2), inter1(R,C2,I).
inter1([X|R],C2,I) :- not member(X,C2), inter1(R,C2,I).
```

6.3.4. Autómata finito no determinista

Los autómatas suelen utilizarse en el reconocimiento de las sentencias que genera una gramática. En este sentido, un autómata es una máquina abstracta que recibe como entrada una cadena de símbolos y la acepta (como perteneciente al lenguaje) o la rechaza. Un autómata puede describirse mediante un grafo en el que los nodos representan los estados del autómata y los arcos etiquetados representan posibles transiciones de estado, que tienen lugar cuando se suministra como entrada el símbolo de la etiqueta. Para concretar, vamos a trabajar con el autómata de la Figura 6.7. En él distinguimos un estado

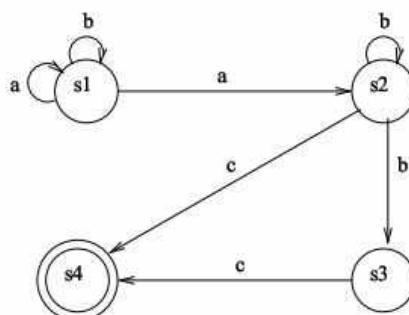


Figura 6.7 Un autómata finito no determinista.

inicial s_1 y un estado final s_4 (enmarcado en un doble círculo). Podemos apreciar que el autómata puede realizar transiciones no deterministas; por ejemplo: si partimos del estado s_1 y se recibe la letra “a”, el autómata bien podrá permanecer en el estado s_1 o cambiar al estado s_2 .

Los objetos matemáticos abstractos tienen una representación directa en Prolog, que nos permite definir predicados ejecutables que simulan su funcionamiento. En concreto, el autómata de la Figura 6.7, puede especificarse, casi de forma inmediata, mediante el empleo de las constantes s_1, s_2, s_3, s_4 , para representar los estados, las constantes a, b, c , para representar los símbolos de entrada, y empleando las relaciones:

1. `final(S)`, que caracteriza el estado final del autómata.
2. `trans(S1, X, S2)`, que define las transiciones de estado que son posibles. Su significado es: la transición desde el estado S_1 al S_2 es posible cuando el símbolo de entrada es X .

de manera que la descripción formal del autómata se materializa mediante el siguiente fragmento de programa:

```
% Descripcion del automata
% final(s4), s4 es el estado final
final(s4).
% trans(S1, X, S2), transicion de S1 a S2
% cuando la entrada es X
% Transiciones que producen cambio de estado
trans(s1, a, s2).
trans(s2, c, s4).
trans(s2, b, s3).
trans(s3, c, s4).
% Transiciones que no producen cambio de estado
trans(s1, a, s1).
trans(s1, b, s1).
trans(s2, b, s2).
```

Finalmente, para describir el comportamiento del automáta basta definir una relación `acepta(Estado, Cadena)` que sea verdadera cuando la cadena de símbolos `Cadena` es aceptada por el autómata. Una cadena se considera aceptada cuando, partiendo del estado actual `Estado`, produce una serie de transiciones que llevan el autómata al estado final. Esto es, el autómata reconoce o acepta una cadena si existe un camino en el grafo de transiciones que:

1. comienza en el estado `Estado`, que se considera inicial;
2. termina en el estado final s_4 y
3. las etiquetas de los arcos que forman el camino se corresponden con los símbolos que forman la cadena de entrada.

Si representamos las cadenas de símbolos de entrada mediante listas, la relación `acepta` (`Estado`, `Cadena`) puede definirse mediante el siguiente programa:

```
% acepta(Estado, Cadena), la Cadena es aceptada,
% cuando el automata esta en el estado Estado
acepta(Estado, []) :- final(Estado).
acepta(Estado, [C|Resto]) :-
    trans(Estado, C, Estado2), acepta(Estado2, Resto).
```

Este programa responde adecuadamente a preguntas del tipo `si/no`, como por ejemplo, “¿el autómata reconoce la cadena ‘abc’ cuando se parte del estado `s1`?”: que se formaliza mediante el objetivo “`?- acepta(s1, [a, b, c]).`”, al que el intérprete Prolog responde afirmativamente. También responde a preguntas del estilo de “¿cuáles son las cadenas de longitud 3 aceptadas por el autómata cuando se parte del estado inicial?”: “`?- acepta(s1, [X,Y,Z]).`”. El intérprete responde computando una combinación de valores para `X`, `Y`, y `Z` que da lugar a una cadena aceptada por el autómata, en este caso “`X = a, Y = b, Z = c`”, que es la primera solución que encuentra el intérprete. Para computar el resto de las combinaciones que dan lugar a cadenas aceptadas por el autómata, debemos forzar la vuelta atrás empleando el operador “`;`” (si tecleamos “`;`”, obtendremos sucesivamente las respuestas: `X = a, Y = a, Z = c; X = b, Y = a, Z = c`).

Finalmente, cuando se plantea la pregunta de “¿cuales son las cadenas aceptadas por el autómata cuando se parte del estado `s2`?”: “`?- acepta(s2, C).`”, el intérprete responde intentando computar todas las cadenas de símbolos que son aceptadas. Esto lo consigue enlazando a la variable `C` la lista construida, cada vez que se fuerza la vuelta atrás: `C = [c] ; C = [b,c] ; C = [b,b,c] ; C = [b,b,b,c] ; ...`. Naturalmente, no podemos esperar obtener todas las respuestas, ya que el árbol de búsqueda es infinito.

Observaciones 6.3

Los programas que se han estudiado hasta el momento ponen de manifiesto algunos puntos importantes:

1. Los programas escritos en Prolog pueden extenderse, simplemente, añadiendo nuevas cláusulas.
2. Las cláusulas que se añaden a los programas son de dos tipos: hechos y reglas.
3. Los hechos declaran propiedades que son (incondicionalmente) verdaderas.
4. Las reglas declaran propiedades que son verdaderas dependiendo de que se cumplan unas condiciones.
5. Las reglas se definen habitualmente por inducción (recursión). La inducción estructural es de gran ayuda cuando se definen propiedades de objetos sintácticos.

6. Las reglas se definen sin pensar en la forma en que tomarán parte en la extracción de respuestas, es decir, sin pensar en la forma en la que se ejecutará el programa (significado procedural). Las reglas se definen pensando en su significado declarativo.
7. Los objetos matemáticos abstractos (como el autómata del Apartado 6.3.4) tienen una representación directa en Prolog, que nos permite definir predicados ejecutables que simulan su funcionamiento. El lenguaje Prolog está especialmente adaptado para resolver este tipo de problemas.



6.4 PROLOG PURO NO ES PURA LÓGICA

El objetivo de este apartado es hacer notar al lector que, aun si nos restringimos a programas que usan el núcleo del lenguaje que hemos denominado Prolog puro, podemos encontrar situaciones en las que el comportamiento de los programas no es el esperado conforme a la lógica. Esto es debido a las peculiaridades del mecanismo operacional del lenguaje Prolog.

Como se ha mencionado, la mayoría de las implementaciones del algoritmo de unificación no realizan la comprobación de ocurrencia de variables (*occur check*) durante el proceso de unificación. Esto lleva a una perdida de la corrección del lenguaje Prolog ya que pueden darse situaciones en las que exista una refutación para un conjunto de cláusulas que no es insatisfacible.

Ejemplo 6.6

Dado el programa:

```
prodigo :- es_hijo(X,X).
es_hijo(Y, padre(Y)).
```

el intérprete de Prolog responde “sí” a la pregunta “?- prodigo.”.

Sin embargo, observe que el universo de Herbrand para este programa es $\mathcal{U}_L = \{a, \text{padre}(a), \text{padre}(\text{padre}(a)), \dots\}$ y que la interpretación modelo mínimo es $\mathcal{M} = \{\text{es_hijo}(t, \text{padre}(t)) \mid t \in \mathcal{U}_L\}$. Obviamente, \mathcal{M} es modelo del programa pero no lo es de “prodigo”. Por consiguiente, *prodigo* no puede ser una consecuencia lógica del programa. También, por adoptar una estrategia de búsqueda en profundidad, Prolog puede perder la completitud al no poder encontrar una refutación para un conjunto de cláusulas insatisfacible. En términos de respuestas computadas, esta falta de completitud se manifiesta en la imposibilidad de encontrar respuestas a preguntas que las tienen.

Ejemplo 6.7

Dado el programa:

```
natural(suc(X)) :- natural(X).
natural(cero).
```

el intérprete es incapaz de responder a la pregunta “?- natural(X).”, si bien existen infinitas respuestas (cero, suc(cero), suc(suc(cero)), ...). El problema es que el intérprete entra por una rama infinita antes de encontrar solución alguna.

La programación en Prolog puede encerrar muchas sutilezas. Por ejemplo, el cambio del orden de las cláusulas en un programa (como ilustra el último ejemplo) y de los subobjetivos en un objetivo, puede alterar la equivalencia entre el significado declarativo y procedimental de un programa Prolog. Vamos a estudiar estas dificultades estudiando un fragmento de programa sobre relaciones familiares.

```
% HECHOS
% padre(X, Y), X es padre de Y
padre(teraj, abraham).
padre(teraj, najor).
padre(teraj, haran).
padre(teraj, sara).
padre(haran, lot).
padre(haran, melca).
padre(haran, jesca).
padre(najor, batuel).
padre(batuel, rebeca).
padre(batuel, laban).
padre(abraham, ismael).
padre(abraham, isaac).
padre(isaac, esau).
padre(isaac, jacob).

% madre(X, Y), X es madre de Y
madre(agar, ismael).
madre(sara, isaac).
madre(melca, batuel).
madre(rebeca, esau).
madre(rebeca, jacob).

% REGLAS
% progenitor(X, Y), X es ascendiente directo de Y
progenitor(X, Y) :- (padre(X, Y); madre(X, Y)).

% ascendiente(X, Y), X es ascendiente de Y
ascendiente(X, Z) :- progenitor(X, Z).
ascendiente(X, Z) :- progenitor(X, Y), ascendiente(Y, Z).
```

Las reglas son la especificación de que un progenitor es un ascendiente directo, esto es, un parent o una madre, y un ascendiente es un progenitor o bien una persona que es un

ascendiente de un progenitor. Ahora podemos ampliar el programa añadiendo diferentes versiones de la relación ascendiente:

- Versión 2. Se intercambia el orden de las cláusulas de la versión original.

```
ascendiente2(X, Z) :- progenitor(X, Y), ascendiente2(Y, Z).  
ascendiente2(X, Z) :- progenitor(X, Z).
```

- Versión 3. Se intercambia el orden de los subobjetivos en el cuerpo de una cláusula de la versión original.

```
ascendiente3(X, Z) :- progenitor(X, Z).  
ascendiente3(X, Z) :- ascendiente3(Y, Z), progenitor(X, Y).
```

- Versión 4. Se intercambia el orden de los subobjetivos y de las cláusulas de la versión original.

```
ascendiente4(X, Z) :- ascendiente4(Y, Z), progenitor(X, Y).  
ascendiente4(X, Z) :- progenitor(X, Z).
```

Una sesión con un intérprete Prolog podría presentar el siguiente aspecto:

```
?- ascendiente(A, laban).  
    A = batuel ;  
    A = teraj ;  
    A = teraj ;  
    A = haran ;  
    A = najor ;  
    A = melca ;  
    no  
  
?- ascendiente2(A, laban).  
    A = teraj ;  
    A = teraj ;  
    A = haran ;  
    A = najor ;  
    A = melca ;  
    A = batuel ;  
    no  
  
?- ascendiente3(A, laban).  
    A = batuel ;  
    A = najor ;  
    A = melca ;  
    A = teraj ;  
    A = haran ;  
    A = teraj ;
```

```

Not enough heap space. Resetting system.
System usage: :
control: 3276:6546:3263:-239 variables: 22890:24877
<Heap space exhausted>:
(3264:3275) progenitor(_11417,teraj). : a

?- ascendiente4(A, laban).
Not enough heap space. Resetting system.
System usage: :
control: 3081:6162:3082:0 variables: 21570:26197
<Heap space exhausted>:
| | | | (3081:3080) ascendiente4(_10780,laban). : a
?-

```

La tercera versión del predicado `ascendiente` produce todas las respuestas a la pregunta, pero finalmente entra en una rama infinita. La cuarta versión no produce ninguna respuesta y entra directamente en una rama infinita. La razón en ambos casos es esencialmente la misma: la llamada recursiva al predicado `ascendiente` antes de llamar al predicado `progenitor`. Vemos también que situar el caso general antes que el caso base, empeora las propiedades de terminación del programa. Estas apreciaciones pueden verse como una regla aplicable a cualquier programa y aun a cualquier estilo de programación, y se concretan en el siguiente aforismo: “intenta hacer primero las cosas simples” [21].

Los programas que se han comentado en este apartado muestran las discrepancias entre el significado declarativo y el procedural de un programa. El significado declarativo de un programa no cambia al alterar el orden de las cláusulas en el programa o de los objetivos dentro de una cláusula. Sin embargo vemos que su significado procedural sí. No debemos interpretar este hecho como una incitación a desechar el sentido declarativo de los programas por inútil. Más bien debemos sacar la siguiente enseñanza práctica: “Hay que centrarse en los aspectos declarativos del problema. Después comprobar, mediante su ejecución, que el programa se comporta de manera eficiente y acorde con su significado esperado; si no es así, entonces intentar reordenar las cláusulas y los objetivos en un orden conveniente.” [21].

Nótese que, a pesar de lo dicho en este apartado, si se sigue un estilo “adecuado” de implementación, es posible desarrollar programas lógicamente correctos, elegantes, compactos y, a la vez, eficientes. En esto consiste “el arte de Prolog” que da título a [139].

6.5 ARITMÉTICA

Dado que el lenguaje Prolog es un lenguaje orientado a la manipulación simbólica, el repertorio de operaciones aritméticas suele ser limitado. La Tabla 6.2 muestra las operaciones básicas disponibles en la mayoría de las implementaciones y los símbolos de función asociados.

Tabla 6.2 Operaciones aritméticas

Operación	Símbolo
Suma	+
Resta	-
Multiplicación	*
División entera	div
Módulo. (resto de la división entera)	mod
División	/

En Prolog estos símbolos de función se denominan operadores y con ellos pueden construirse expresiones aritméticas que no son otra cosa que términos¹⁵.

Ejemplo 6.8

Ejemplos de expresiones aritméticas son:

- i) $3 + 2 * 5$, ii) $3 / 2 + 2 * 5$, iii) $6 \text{ div } 2$,
- iv) $3 \text{ mod } 2$, v) $6 / 2$, vi) $50 + 10 / 3 \text{ mod } 2$,
- vii) $Y + 10 / 2$.

En Prolog las expresiones no se evalúan automáticamente, contrariamente a lo que sucede en otros lenguajes¹⁶, y es preciso forzar su evaluación mediante el operador “is”, que es un operador infijo binario. Dado un objetivo “?- *Expresion1 is Expresion2.*”, tanto la parte izquierda como la parte derecha son expresiones aritméticas, que de contener variables deberán estar instanciadas en el momento de la ejecución del objetivo, cuyo efecto es forzar la evaluación de dichas expresiones y después tratar de unificarlas. La excepción tiene lugar cuando la *Expresion1* es una variable, ya que entonces queda instanciada al valor de la *Expresion2*. La evaluación concreta de cada operación aritmética se realiza mediante procedimientos especiales, predefinidos (*built-in*) en el intérprete, que acceden directamente a las facilidades de cómputo de la máquina.

Ejemplo 6.9

Evaluación de las expresiones aritméticas del Ejemplo 6.8:

```
?- X is 3 + 2 * 5.
X = 13
```

¹⁵Conviene recordar aquí que Prolog es un lenguaje sin tipos; todos los datos pertenecen a un mismo dominio *sintáctico*, el universo de Herbrand.

¹⁶Por ejemplo, el lenguaje Lisp, en el que hay que manifestar expresamente el deseo de que una expresión aritmética no se evalúe anteponiendo a la expresión el operador *quote*.

```

?- X is 3 / 2 + 2 * 5.
X = 11.500000

?- X is 6 div 2.
X = 3

?- X is 3 mod 2
X = 1

?- X is 6 / 2.
X = 3.0

?- X is 50 + 10 / 3 mod 2.
X = 60.0

?- X is Y + 10 / 2.
Illegal argument to is.
_1 <Illegal Arg>: (1:0) _0 is _1 + 10 / 2. : a

```

En el último caso, el mensaje de error se produce porque la variable *Y* no está instanciada en el momento en que se evalúa la expresión. Compare este último resultado con la salida que se obtiene cuando el objetivo es: `?- Y = 2, X is Y + 10 / 2.` Ya se ha mencionado que el operador “*is*” fuerza la evaluación de las expresiones. Por consiguiente, el operador “*is*” no puede utilizarse para producir la instanciación de una variable dentro de una expresión: si lanzamos el objetivo “`?- 40 is X + 10 / 2.`”, la respuesta es “*Illegal argument to is. _0*”. Otro error muy frecuente es usar el operador “*is*” para actualizar una variable que se usa como parámetro en llamadas recursivas. Por ejemplo, el siguiente programa erróneo, pensado para simular un contador de *1* a *N*, no se comporta de la forma esperada.

```

contar(N,N) :- write(N).
contar(X,N) :- X<N, write(X), nl, X is X + 1, contar(X,N).

```

De hecho, funciona mal tanto si la variable *x* está instanciada en la llamada inicial como si no lo está. Pongamos, por ejemplo, que la llamada inicial fuese “`contar(1,3)`”. Tras escribir en pantalla el “*1*”, seguido del salto de línea (que fuerza el predicado `nl`), el intento de unificar *1* y *2* a través de la invocación “*1 is 2*” produce como resultado un fallo (es decir, una respuesta “*no*” del intérprete). Por el contrario, si la llamada inicial se produjese con la variable *x* desinstanciada, la ejecución del predicado “`write(x)`” produce la escritura en pantalla de un valor numérico asociado a la posición en memoria de dicha variable y, después del salto de línea, se produce un mensaje de error del intérprete al invocar el operador “*is*” con una expresión aritmética a la derecha del mismo que no está convenientemente instanciada.

Sin embargo, el operador “*is*” puede servir para realizar comparaciones: si lanzamos el objetivo “`?- 36.5 is 30 + 13 / 2.`”, la respuesta es “*yes*”.

Al escribir expresiones aritméticas para su evaluación es necesario tener en cuenta que éstas se procesan de izquierda a derecha y que rigen las reglas de precedencia de la Tabla 6.3. Los operadores con menor número de precedencia son los que enlazan con

Tabla 6.3 Precedencia de las operaciones aritméticas

Precedencia	Operaciones
500	+, -
400	*, div, /
300	mod

más fuerza a sus argumentos. Consulte el Apartado 6.6 para obtener más información sobre el efecto de la precedencia de los operadores en el cálculo de las expresiones y tenga en cuenta que las reglas de precedencia pueden quedar anuladas por el empleo de los paréntesis.

Ejemplo 6.10

Al aplicar las reglas de precedencia de la Tabla 6.3 a las expresiones aritméticas del Ejemplo 6.8: i) $3 + 2 * 5$, es equivalente a la expresión $3 + (2 * 5)$; ii) $3 / 2 + 2 * 5$, es equivalente a $(3 / 2) + (2 * 5)$ iii) $50 + 10 / 3 \text{ mod } 2$, es equivalente a $50 + (10 / (3 \text{ mod } 2))$. Como puede comprobarse mediante su evaluación en un sistema Prolog.

Las expresiones aritméticas pueden compararse utilizando los operadores de comparación que se muestran en la Tabla 6.4.

Tabla 6.4 Operaciones de comparación de expresiones aritméticas.

Operación	Símbolo	Significado
Mayor que	>	E1 > E2. E1 es mayor que E2.
Menor que	<	E1 < E2. E1 es menor que E2.
Mayor o igual que	>=	E1 >= E2. E1 es mayor o igual que E2.
Menor o igual que	=<	E1 = < E2. E1 es menor o igual que E2.
Igual que	= :=	E1 = := E2. E1 es igual que E2.
Distinto que	= \=	E1 = \= E2. E1 no es igual que E2.

Los operadores de comparación fuerzan la evaluación de las expresiones aritméticas y no producen la instanciación de las variables. En el momento de la evaluación todas las variables deberán estar instanciaadas a un número.

Observación 6.2 (Orden estándar) En Prolog los términos pueden compararse utilizando el denominado orden estándar. Con ligeras variaciones, dependiendo del sistema Prolog¹⁷, el orden estándar se define como sigue:

```
variables @< constantes @< cadenas @< números @< términos
```

donde el operador “@<” se interpreta como “es menor que” o “precede a” en la relación de orden (total). Dentro de cada una de estas categorías, las expresiones se ordenan siguiendo estos criterios:

1. Las variables se ordenan por su dirección en memoria (las “viejas” se consideran menores que las “nuevas”).
2. Las constantes se ordenan alfabéticamente, al igual que las cadenas.
3. Los números se ordenan según su valor (según el orden habitual que rige en ese dominio).
4. Los términos se ordenan comparando, primero, su aridad; después, su nombre de función lexicográficamente; finalmente, se comparan recursivamente los argumentos de izquierda a derecha.

Los operadores de comparación de términos son: =@=, \=@=, @<, @=<, @> y @>=. Una expresión como “T1 =@= T2” se satisface cuando T1 tiene la misma estructura que T2. Una expresión como “T1 @=< T2” se satisface cuando T1 es sintácticamente igual a T2 (es decir, T1 == T2 —véase más adelante—) o bien T1 precede a T2. El resto de las expresiones de comparación que pueden formarse tienen un significado evidente.

Observe que cuando lo que se desea comparar son simplemente números, es preferible y más eficiente utilizar los operadores de comparación aritméticos.

6.6 OPERADORES

En el apartado anterior se han introducido diversos operadores aritméticos, que permiten la construcción de expresiones aritméticas (términos) con una sintaxis que difiere de la estándar. El lenguaje Prolog proporciona una directiva para la introducción de nuevos operadores. Ésta es una característica muy útil del lenguaje, que describiremos en lo que sigue.

Un operador se caracteriza por su nombre, su precedencia (o prioridad) y su especificador de modo (Posición/Asociatividad).

¹⁷Los criterios que se especifican en esta observación son conformes a los utilizados en la implementación de SWI-Prolog [142].

Nombre

El nombre es un átomo o una combinación de caracteres especiales (+, -, *, /, <, >, =, :, &, _, ~). Cuando se elija un nombre concreto para un operador, habrá que tener en cuenta que ciertos nombres están reservados (e.g., “is”, “=” “+”, “:-”, etc.).

Precedencia

La precedencia es un número entero (entre 1 y 1200), asignado a cada operador, que indica cómo debe intrepretarse una expresión en la que no se emplean paréntesis. En una expresión sin paréntesis, el operador con mayor número de precedencia se convierte en el operador principal de la expresión. Los operadores con menor número de precedencia son los que enlazan con más fuerza a sus argumentos. El empleo de paréntesis anula las reglas de precedencia. Por ejemplo, como ya se ha indicado, el operador “+” liga menos que el operador “*”; eso se expresa dando al operador “+” un número de precedencia mayor que al operador “*”. De esta forma, una expresión sin paréntesis, como “ $3+4*2$ ”, se interpreta como “ $3+(4*2)$ ” y no como “ $(3+4)*2$ ”.

Posición

La posición de un operador es una característica que se define empleando especificadores de modo. Los especificadores de modo “_f_”, “_f” o “f_” (donde, en este contexto, el símbolo “_” representa una “x” o una “y”—véase un poco más adelante—) nos dicen, respectivamente, si el operador es infijo, sufijo (también llamado postfijo), o prefijo.

Asociatividad

Se emplea para eliminar la ambigüedad de las expresiones en las que aparecen operadores con la misma precedencia. Por ejemplo, determina cómo interpretar una expresión de la forma: “ $16/4/2$ ”. Si se interpreta como “ $(16/4)/2$ ” el resultado es 2, pero si se interpreta como “ $16/(4/2)$ ” el resultado es 8. La asociatividad de un operador se especifica al mismo tiempo que la propiedad de posición, empleando los especificadores de modo, que son átomos (constantes no numéricas de Prolog) especiales¹⁸:

fx , fy , xf , yf , xfx , xfy , yfx .

Una “x” significa que ese argumento debe tener un número de precedencia estrechamente menor que el operador que se está especificando. Por otra parte una “y” indica que el argumento correspondiente puede tener una precedencia igual o menor. Se define

¹⁸Observe que el especificador de modo “yfy” no tiene sentido, ya que su uso conduciría a ambigüedades.

Tabla 6.5 Asociatividad de los operadores

Especificador de modo	Asociatividad
f _y	a derechas
y _f	a izquierdas
x _f _y	a derechas
y _f _x	a izquierdas

la precedencia de un argumento como sigue: si el argumento es un término su precedencia es cero, a menos que esté encabezado por un operador (al que se ha asignado una precedencia), en cuyo caso su precedencia es la de ese operador. Si el argumento está encerrado entre paréntesis su precedencia es cero.

Ejemplo 6.11

La posición y asociatividad de “/” es “yfx”. Por tanto, atendiendo a lo anterior, podemos ver que “16/4/2” se interpretará como “(16/4)/2”; esto es, el operador “/” es asociativo a izquierdas.

El operador aritmético de cambio de signo -, lo mismo que las conectivas lógicas : - , ? - suelen especificarse con asociatividad fx, lo que convierte en incorrectas expresiones como - - 4 o ? - ? - p (x). En cambio, el operador not suele tener una especificación de asociatividad fx, por lo que es posible escribir una negación doble como not not p.

El efecto sobre la asociatividad de un operador que produce cada especificador de modo se muestra en la Tabla 6.5. Los especificadores de modo legales que no aparecen en la tabla declaran operadores que no son asociativos; esto es, es obligatorio utilizar paréntesis para deshacer ambigüedades.

Declaración y definición de operadores

Se pueden declarar nuevos operadores insertando en un programa un tipo especial de cláusulas llamadas directivas. Una declaración de un operador debe aparecer antes de cualquier expresión que vaya a usarla. Con más precisión, una declaración de un operador tiene la forma:

```
: - op (Precedencia, Modo, Nombre) .
```

donde Precedencia es la precedencia, Modo es el especificador de modo y Nombre es el nombre del operador. Varios operadores pueden declararse mediante una única directiva, en la que los nombres de los operadores se suministran formando una lista.

Ejemplo 6.12

Para introducir un nuevo operador “:=” que permitiese representar las sustituciones como una lista de elementos “Variable := Término”, bastaría con declarar el operador

```
: -op(200, xfx, ':=') .
```

Ahora, la sustitución $\{X/a, Y/g(h(W)), Z/g(b)\}$ podría representarse mediante la lista $[X:=a, Y:=g(h(W)), Z:=g(b)]$. Nótese que el uso de operadores puede aumentar la legibilidad de los programas. Por otra parte, observe también que la declaración de un operador no le asigna ningún significado. Al igual que sucede en el Ejemplo 6.12, habitualmente los operadores se utilizan como functores, que combinan objetos para formar estructuras y no para invocar acciones. Si queremos que un operador tenga un significado, será preciso asociarle una definición. Esto es, será preciso asociarle un conjunto de cláusulas que determinan su significado.

Ejemplo 6.13

En apartados anteriores se han definido los predicados sobre listas: `member(Elemento, Lista)` y `append(Lista1, Lista2, Resultado)`. Supongamos que preferimos utilizar estas relaciones usando la siguiente sintáxis:

```
Elemento en Lista,  
concatenar Lista1 y Lista2 da Lista3,
```

Una forma de hacer esto es declarar “en”, “concatenar”, “y”, etc. como operadores y definir los predicados correspondientes, tal y como se muestra a continuación:

```
% OPERADORES SOBRE LISTAS.  
:- op(350, xfx, en).  
:- op(300, fx, concatenar).  
:- op(250, yfx, [y, da]).  
% Elemento en Lista  
E en [E|_].  
E en [_|Resto] :- E en Resto.  
% concatenar Lista1 y Lista2 da Lista3  
concatenar [] y L da L.  
concatenar [X|Resto] y L1 da [X|L] :- concatenar Resto y L1 da L.
```

Las declaraciones de los operadores, y en especial sus niveles de precedencia y asociatividad, contribuyen a que la expresión “concatenar Lista1 y Lista2 da Lista3” sea equivalente al término “`concatenar(da(y(Lista1, Lista2), Lista3))`”. Observe que esta formalización de las relaciones de pertenencia y concatenación nos permite plantear cuestiones al sistema en un lenguaje muy cercano al lenguaje natural.

6.7 ÁRBOLES

Los árboles son estructuras que cuentan con una amplia tradición en el campo de las matemáticas, a partir de su introducción en 1857, por parte del matemático inglés Arthur

Cayley. Aunque los árboles pueden verse como un caso particular de grafo (conexo) que no contiene ciclos¹⁹, en este apartado los presentaremos siguiendo una orientación más práctica e informal. Una definición precisa de estas estructuras y una terminología básica, puede encontrarse en el Apéndice A (Apartado A.3.3).

Los árboles son una forma útil de estructurar la información que tiene infinidad de aplicaciones en informática. Por ejemplo se han utilizado: para implementar algoritmos de búsqueda eficientes; en el análisis sintáctico de los lenguajes de programación; para construir bases de datos jerárquicas; etc. Esto justifica nuestro interés en introducir medios para su representación y técnicas de programación que faciliten su manipulación usando el lenguaje Prolog.

Los árboles son estructuras inherentemente recursivas que pueden definirse como sigue. Un *árbol n–ario* de tipo T es:

1. Bien una estructura vacía,
2. o bien un elemento de tipo T, denominado *nodo*, junto con m (siendo $0 \leq m \leq n$) subconjuntos disjuntos de elementos; estos subconjuntos son a su vez árboles n–arios de tipo T, que se denominan *subárboles* del árbol original.

A partir de esta definición, resulta evidente que una lista puede considerarse como un tipo especial de árbol en el que los nodos tienen como máximo un subárbol. En ocasiones se dice que una lista es un *árbol degenerado*.

Antes de enfrentarnos a los árboles n–arios conviene que estudiemos los árboles binarios e implementemos en Prolog algunas de sus propiedades.

6.7.1. Árboles binarios

Los árboles binarios, aunque pueden expresarse en términos de los árboles 2–arios, constituyen una estructura de datos ligeramente diferente, tanto en cuanto que cada nodo (que no sea una hoja) está obligado a tener dos hijos (es decir no están permitidos nodos con un único descendiente, como sucede con los árboles 2–arios —véase la Figura 6.8—).

Una forma habitual de representar²⁰ los árboles binarios es mediante el empleo de una constante, *nil*, que representa el árbol vacío, y símbolos de función especiales *hoja* (de aridad 1) y *arbol* (de aridad 3) para construir árboles no vacíos. Más formalmente, decimos que

- *nil*, el árbol vacío, es un árbol binario;
- si x es un elemento, entonces *hoja*(x) es un árbol binario;
- si x es un elemento y L y R son árboles binarios, entonces *arbol*(L, x, R) es un árbol binario.

¹⁹Más precisamente, circuitos simples.

²⁰Véase el libro de I.Bratko para una representación alternativa [21].

En un árbol binario, para un nodo interno, el primer (resp. el segundo) hijo de un nodo se denomina hijo de la izquierda (resp. derecha) y el subárbol que tiene como raíz el hijo de la izquierda (resp. la derecha) se denomina subárbol izquierdo (resp. derecho).

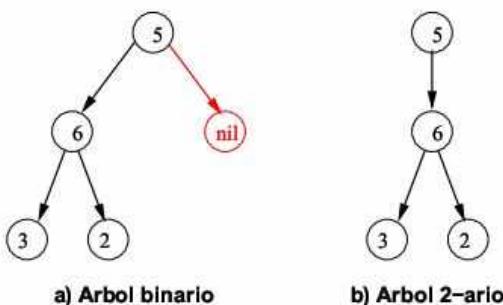


Figura 6.8 Diferencias entre un árbol binario y el correspondiente árbol 2-ario.

Ejemplo 6.14

Utilizando la anterior caracterización de árbol binario, el árbol de la Figura 6.8(a) se representa como:

```
árbol (árbol (hoja(3),
              6,
              hoja(2)
            )
      5,
      nil
    )
```

Los árboles en general, y los árboles binarios en particular, son estructuras bidimensionales (como se comprueba al representarlos gráficamente) cuyo tratamiento iterativo presenta grandes dificultades, desde el momento en que no existe una única forma de recorrerlos [114]. Sin embargo se trata de estructuras que admiten de forma natural un tratamiento basado en el empleo de inducción estructural. Mediante técnicas recursivas se pueden definir, con gran facilidad y elegancia, los siguientes predicados sobre árboles binarios:

- `esArbolBinario(O)` que permite determinar si un objeto O es o no un árbol binario.

```
% esArbolBinario(O), el objeto O es un árbol binario.
esArbolBinario(nil).
esArbolBinario(hoja(Hoja)) :- elemento(Hoja).
esArbolBinario(arbol(ArbolL, Nodo, ArbolR)) :-
```

```
esArbolBinario(ArbolL),
elemento(Nodo),
esArbolBinario(ArbolR).
```

El predicado `elemento` habría que definirlo según fuese el dominio de discurso con el que se está trabajando. En el caso de que se trate de números enteros, la definición sería:

```
% elemento(E), el elemento E pertenece al dominio de los
% enteros.
elemento(E) :- integer(E).
```

Aunque, en el resto del apartado no realizaremos comprobaciones de dominio.

- `enArbolBinario(E, A)` que permite determinar si un elemento `E` es miembro del árbol binario `A`.

```
% enArbolBinario(E, A), E es miembro del arbol binario A.
enArbolBinario(E, hoja(E)).
enArbolBinario(E, arbol(ArbolL, E, ArbolR)).
enArbolBinario(E, arbol(ArbolL, N, ArbolR)) :-
    (enArbolBinario(E, ArbolL);enArbolBinario(E, ArbolR)).
```

- `construir(L, A)` construye un árbol binario `A`, a partir de los elementos de una lista `L`. Nuestro interés es que el árbol construido sea de altura mínima, para que permanezca equilibrado²¹. Los árboles de altura mínima son útiles porque aseguran que el coste de buscar un elemento será mínimo, cuando se mantiene cierto orden entre sus elementos (véase el Apartado 6.7.2). Se puede obtener un árbol de altura mínima dividiendo la lista en dos mitades y construyendo recursivamente un árbol de altura mínima para cada una de las mitades.

```
% construir(L, A), construye un arbol binario A, a partir
% de los elementos de una lista L.
construir([], nil).
construir([X], hoja(X)).
construir(L, arbol(ArbolL, X, ArbolR)) :-
    particion(L, LL, [X|LR]),
    construir(LL, ArbolL), construir(RR, ArbolR).

% particion(L, LL, LR), divide la lista L en dos partes
% iguales LL y LR (salvo restos).
particion(L, LL, LR) :-
    length(L, N), M is (N div 2),
    length(LL, M), append(LL, LR, L).
```

²¹Véase el Ejercicio 6.34 al final de este capítulo.

En la definición del predicado `partir` se ha empleado el predicado predefinido `length`, que calcula la longitud de una lista. Esencialmente, el predicado `partir` calcula la longitud de la lista `L` y la divide por dos para hallar `M`; después encarga al predicado `append` que construya las dos mitades, `LL` y `LR`, de la lista `L`. Previamente, la llamada `length(LL, M)` crea una lista `LL` que contiene `M` variables sin instanciar. Esto facilita la tarea del predicado `append`. Observe que se está haciendo uso de la invisibilidad de los predicados `length` y `append` y de la potencia de la unificación.

6.7.2. Árboles de búsqueda

Cuando trabajamos con elementos de dominios para los que existe una relación de orden, resulta muy útil imponer ciertas restricciones a la construcción de los árboles binarios. La idea de disponer los elementos de un árbol binario en las posiciones dictadas por la relación de orden, y no en otras, da lugar a la noción de árbol de búsqueda. Un árbol binario se dice que es un *árbol de búsqueda* si en cada uno de los subárboles el nodo raíz es mayor que todos los elementos del subárbol izquierdo y menor que todos los elementos del subárbol derecho. Los árboles binarios de búsqueda se llaman “*diccionarios*” en [21] y en lo que sigue adoptaremos esa denominación con preferencia a la de “árbol de búsqueda”.

El primer ejercicio de programación que vamos a realizar con estas estructuras va a consistir en implementar un predicado, `esDiccionario(O)`, que permita determinar si un objeto `O` es o no un diccionario binario. Para llevar a cabo esta tarea resulta conveniente redefinir, con una orientación más claramente inductiva, la noción de árbol de búsqueda. Utilizando la representación para los árboles binarios introducida en el Apartado 6.7.1, podemos precisar el concepto de árbol de búsqueda. Los árboles vacíos, `nil`, y los nodos hoja, `hoja(X)`, ya están ordenados y, por lo tanto, son árboles de búsqueda. Por otra parte, un árbol binario `arbol(ArbolL, X, ArbolR)` es un árbol de búsqueda, si las tres condiciones siguientes se cumplen:

- todos los nodos en el subárbol izquierdo, `ArbolL`, son menores que `X`; para verificar esta condición basta comprobar que el máximo elemento, `MaxL`, de `ArbolL` es menor que `X`;
- todos los nodos en el subárbol derecho, `ArbolR`, son mayores que `X`; para verificar esta condición basta comprobar que el mínimo elemento, `MinR`, de `ArbolR` es mayor que `X`;
- los elementos de los subárboles `ArbolL` y `ArbolR` están ordenados de modo que ambos son árboles de búsqueda.

Esta redefinición del concepto nos conduce al siguiente programa²², en el que para cada subárbol binario se almacena el máximo y el mínimo elemento de dicho (sub)árbol

²²Con el fin de dotar de flexibilidad a los programas de este apartado, utilizamos los operadores de comparación para el orden estándar definidos en la Observación `refordenEstandar`.

con el fin de facilitar la tarea de verificar si los (sub)árboles construidos son árboles de búsqueda:

```

esDiccionario(nil).
esDiccionario(Arbol) :- esDiccionario(Arbol, _, _).

% esDiccionario(Arbol, Min, Max),
% Arbol es un diccionario con elemento mínimo Min y elemento
% máximo Max
esDiccionario(hoja(Hoja), Hoja, Hoja).
esDiccionario(arbol(nil, Nodo, ArbolR), Nodo, MaxR) :-
    esDiccionario(ArbolR, MinR, MaxR),
    Nodo @< MinR.
esDiccionario(arbol(ArbolL, Nodo, nil), MinL, Nodo) :-
    esDiccionario(ArbolL, MinL, MaxL),
    Nodo @> MaxL.
esDiccionario(arbol(ArbolL, Nodo, ArbolR), MinL, MaxR) :-
    esDiccionario(ArbolL, MinL, MaxL),
    esDiccionario(ArbolR, MinR, MaxR),
    Nodo @> MaxL,   %% Nodo es mayor que cualquier
                    %% elemento de ArbolL
    Nodo @< MinR.   %% Nodo es menor que cualquier
                    %% elemento de ArbolR

```

Se aconseja al lector que medite sobre el funcionamiento de este programa y piense cómo se obtiene el valor del máximo y el mínimo elemento de un (sub)árbol.

El empleo de diccionarios nos permite programar un algoritmo de ordenación elegante y eficiente, consistente en construir un diccionario a partir de una lista desordenada y, después, realizar un recorrido inorden (es decir, visitando primero los nodos del subárbol izquierdo, en segundo lugar la raíz y, finalmente, los nodos del subárbol derecho), lo que produce una lista ordenada. Las diferentes piezas de este programa son las siguientes:

- `construirDic(L,D)`, que construye un diccionario D a partir de la lista L.

```

% construirDic(L,D),
% Crea un diccionario D a partir de la lista L
construirDic(L, D) :- construirDic(L, nil, D).
construirDic([], D, D).
construirDic([X|R], D1, D2) :-
    insertar(D1, X, D), construirDic(R, D, D2).

% insertar(D, E, D1),
% inserta el elemento E en D para dar D1;
% La idea es insertar el nuevo elemento E como una hoja,
% conservando el orden del diccionario.
% Casos bases.
insertar(nil, E, hoja(E)).
insertar(hoja(X), E, arbol(nil, X, hoja(E))) :- X @=< E.

```

```
insertar(hoja(X), E, t(hoja(E), X, nil)) :- X @> E.
% Casos generales.
insertar(arbol(AL, X, AR), E, arbol(AL, X, NAR)) :-
    X @=< E, insertar(AR, E, NAR).
insertar(arbol(AL, X, AR), E, arbol(NAL, X, AR)) :-
    X @> E, insertar(AL, E, NAL).
```

- `inorden(A, L)`, que realiza un recorrido inorden y transforma un árbol binario A en una lista L; si el árbol binario A es un diccionario la lista resultante es una lista ordenada.

```
% inorden(A, L) transforma A en una Lista {\lt L}.
inorden(nil, []).
inorden(hoja(Hoja), [Hoja]).
inorden(arbol(AL, X, AR), Lista) :-
    inorden(AL, Ll), inorden(AR, Lr),
    append(Ll, [X|Lr], Lista).
```

- `ordenar(Desordenada, Ordenada)`, ordena la lista Desordenada dando lugar a la lista Ordenada

```
ordenar(Desordenada, Ordenada) :-
    construirDic(Desordenada, D),
    inorden(D, Ordenada).
```

Los árboles de búsqueda, como su propio nombre indica, facilitan la búsqueda eficiente de elementos en un árbol, ya que no es necesario recorrer toda la estructura del árbol para determinar si un elemento está o no en él, como muestra el siguiente programa:

```
% enDiccionario(E, D), comprueba si el elemento E esta en D.
enDiccionario(E, hoja(E)).
enDiccionario(E, arbol(_, E, _)).
enDiccionario(E, arbol(_, X, ArbolR)) :-
    X @< E, enDiccionario(E, ArbolR).
enDiccionario(E, arbol(ArbolL, X, _)) :-
    X @> E, enDiccionario(E, ArbolL).
```

Otras operaciones útiles con diccionarios se discuten en los Ejercicios 6.33 y 6.34. Una implementación alternativa de la idea de diccionario puede encontrarse en el Apartado 7.7.3 del próximo capítulo.

6.7.3. Árboles n -arios

Los árboles n -arios admiten una representación sencilla si empleamos un nodo etiquetado con un valor, junto con una secuencia de subárboles de longitud menor o igual que n . Esto es, podemos emplear un constructor binario, que denominaremos *nodo*, cuyo primer argumento será el valor del nodo y el segundo una lista de subárboles. También emplearemos un símbolo de función unario, *hoja*, para construir árboles constituidos por un solo elemento, y la constante *nil*, para representar árboles vacíos. Así pues, caracterizamos un árbol n -ario como sigue:

- *nil* es un árbol n -ario
- si *x* es un elemento, entonces *hoja(x)* es un árbol n -ario;
- si *x* es un elemento y T_1, \dots, T_k es una secuencia de árboles n -arios, con $k \leq n$, entonces *nodo(x, [T_1, \dots, T_k])* es un árbol n -ario.

Ejemplo 6.15

Utilizando la anterior caracterización de árbol n -ario, el árbol 2-ario de la Figura 6.8(b) se representa como:

```
nodo(5, [nodo(6, [hoja(3), hoja(2)])])
```

Este tipo de estructuras ha recibido diversos nombres entre los que se encuentran los de: “árboles de ramificación múltiple” y, también, “*rosadelfas*”²³. Terminamos este apartado definiendo los siguientes predicados sobre *rosadelfas*, que son adaptación de los introducidos en el Apartado 6.7.1:

- *esRosadelfa(O)* que permite determinar si un objeto *O* es o no una *rosadelfa*.

```
% esRosadelfa(O), el objeto O es una rosadelfa.
esRosadelfa(nil).
esRosadelfa(hoja(_)).
esRosadelfa(nodo(_, Rosadelfas)) :-
    esListaRosadelfas(Rosadelfas).

esListaRosadelfas([R]) :- esRosadelfa(R).
esListaRosadelfas([R|Rosadelfas]) :-
    esRosadelfa(R),
    esListaRosadelfas(Rosadelfas).
```

- *enRosadelfa(E, R)* que permite determinar si un elemento *E* es miembro de la *rosadelfa R*.

²³“*Rosadelfa*” es la traducción de Ricardo Peña para el término inglés “*rose tree*”, introducido por Lambert Meertens.

```
% enRosadelfa(E, R), E es miembro de la rosadelfa R.  
enRosadelfa(E, hoja(E)).  
enRosadelfa(E, nodo(E, _)).  
enRosadelfa(E, nodo(_, Rosadelfas)):-  
    enRosadelfas(E, Rosadelfas).  
  
% inspecciona una lista de rosadelfas  
enRosadelfas(E, [R|Rosadelfas]) :- enRosadelfa(E, R).  
enRosadelfas(E, [R|Rosadelfas]) :-  
    enRosadelfas(E, Rosadelfas).
```

- `construirRosadelfa(L, G, A)` construye una rosadelfa `R`, de grado `G` a partir de los elementos de una lista `L`.

```
% construir(L, G, A), construye una rosadelfa R, a partir  
% de los elementos de una lista L.  
construirRosadelfa([X], _, hoja(X)).  
construirRosadelfa([X|L], G, nodo(X, [R|Rosadelfas])):-  
    partir(L, G, [L1|GListas]),  
    construirRosadelfa(L1, G, R),  
    construirRosadelfas(GListas, G, Rosadelfas).  
  
% inspecciona una lista de listas y por cada lista  
% inspeccionada crea una rosadelfa  
construirRosadelfas([], _, []).  
construirRosadelfas([L|GListas], G, [R|Rosadelfas]) :-  
    construirRosadelfa(L, G, R),  
    construirRosadelfas(GListas, G, Rosadelfas).  
  
% partir(L, G, [L1|GListas]), divide la lista L en una se-  
% cuencia de listas de longitud menor o igual que G, que se  
% devuelve como una lista de listas en el segundo argumento.  
partir(L, G, [L]) :- length(L, N), N < G.  
partir(L, G, [L1|GListas]) :-  
    length(L, N), N >= G,  
    length(L1, G),  
    append(L1, LL, L),  
    partir(LL, G, GListas).
```

La definición del predicado `partir` trata dos casos: i) si la longitud de la lista `L` es menor que `G`, entonces es un resto que debe unirse directamente a la secuencia de listas; ii) en caso contrario, se fragmenta la lista `L` en un prefijo de longitud `G` y un resto de lista `LL`, que se sigue partiendo, mediante la llamada recursiva a `partir`. Observe que la llamada `length(L1, G)` puede utilizarse para crear una lista `L1` que contenga `G` variables, que posteriormente se instanciarán.

6.8 DISTINTAS CLASES DE IGUALDAD Y UNIFICACIÓN

El lenguaje Prolog suministra diferentes clases de “igualdad” útiles para comparar elementos de diferentes dominios. En este apartado se comentan algunas de sus peculiaridades.

Igualdad de expresiones aritméticas

Como hemos visto en el Apartado 6.5 anterior, podemos comprobar la igualdad de dos expresiones aritméticas mediante el operador “`=:=`”. El operador “`=:=`” evalúa sus argumentos y comprueba que son iguales.

Ejemplo 6.16

Una sesión en un intérprete del lenguaje Prolog.

```
?- 21.5 =:= (30 + 13) / 2.
yes
?- (23 + 20) / 2 =:= (30 + 13) / 2.
yes
?- (30 + 13) / 2 =:= (30 + 13) / 2.
yes
```

Igualdad sintáctica

En muchas ocasiones puede interesar comprobar la igualdad sintáctica de dos expresiones, con este fin se introduce el operador “`==`”. El operador “`==`” comprueba si sus argumentos son o no son sintácticamente iguales.

Ejemplo 6.17

Una sesión en un intérprete del lenguaje Prolog.

```
?- 21.5 == (30 + 13) / 2.
no
?- (23 + 20) / 2 == (30 + 13) / 2.
no
?- (30 + 13) / 2 == (30 + 13) / 2.
yes
?- f(g(X), b) == f(g(X), b).
yes
```

El operador “`\==`” es la negación de “`==`”.

El operador de unificación

En Prolog, el símbolo “=” representa la relación de unificación que se define mediante una única cláusula

% X = Y, X unifica con Y

Esta sencilla definición hace que, cuando se plantea el objetivo “?- *Expresion1* = *Expresion2*.”, el intérprete actúe intentando unificar la cabeza de la cláusula “*x* = *x*” con el subobjetivo “*Expresion1* = *Expresion2*”. Esto es, el intérprete utiliza el procedimiento de unificación predefinido internamente para unificar el conjunto de ecuaciones $\{x = \textit{Expresion1}, x = \textit{Expresion2}\}$. El razonamiento anterior justifica que el objetivo “?- *Expresion1* = *Expresion2*.” tenga éxito si las expresiones *Expresion1* y *Expresion2* unifican y que el intérprete responda dando el unificador más general de *Expresion1* y *Expresion2*.

Ejemplo 6.18

El objetivo “?- f(g(X), a) = f(Y, X).” tiene éxito y el intérprete responde “X = a Y = g(a)”. El unificador más general de $f(g(X), a)$ y $f(Y, X)$ es la sustitución $\{X/a, Y/g(a)\}$.

Observación 6.3 Debido a que Prolog no comprueba la ocurrencia de variables (occurrence check) en el procedimiento de unificación, el objetivo “?- $X = f(X)$. ” no falla. En cambio, proporciona la salida.

Esto ilustra las dificultades que pueden derivarse de un empleo incorrecto del algoritmo de unificación. El Capítulo 7 del libro de Apt [6] se dedica al estudio del problema del occur check.

Finalmente, observe que ninguna de estas clases de igualdad se corresponde con la igualdad semántica, que como sabemos es desterrada del ámbito de la programación lógica por el alto coste de su implementación.

RESUMEN

El lenguaje de programación *Prolog* (PROgramación en LOGica – Colmenauer et. al. 1973) es la realización más conocida de las ideas introducidas en el campo de la programación lógica. Para usar estas ideas en la construcción de un lenguaje de programación útil, ha sido necesario mejorar las prestaciones del mecanismo operacional, para conseguir una mayor eficiencia en la ejecución de los programas y aumentar la expresividad del lenguaje de las cláusulas de Horn.

En este capítulo se discuten una serie de características del lenguaje Prolog que deben tenerse en cuenta para su correcto aprendizaje.

- Hemos estudiado un subconjunto de Prolog, denominado *Prolog puro* [132], que se ajusta a los supuestos de la programación lógica.
- Exceptuando la notación que utiliza Prolog para designar las conectivas lógicas, la sintaxis de Prolog puro se corresponde con la notación clausal: las construcciones del lenguaje Prolog son términos y sus instrucciones son cláusulas definidas.
- Prolog puro utiliza como mecanismo operacional la estrategia de resolución SLD implementada con las siguientes restricciones: i) regla de computación que selecciona el átomo situado más a la izquierda dentro de la secuencia de subobjetivos; ii) regla de ordenación que inspecciona las cláusulas de arriba hacia abajo; iii) regla de búsqueda en profundidad con vuelta atrás (*backtracking*); iv) omisión de la prueba de ocurrencia de variables (*occur check*).
- A través de algunos ejemplos hemos extraído una serie de características distintivas del estilo de programación en Prolog:
 - Los programas escritos en Prolog pueden extenderse, simplemente, añadiendo nuevas cláusulas.
 - Las cláusulas que se añaden a los programas son de dos tipos: hechos y reglas.
 - Los hechos declaran propiedades que son (incondicionalmente) verdaderas.
 - Las reglas declaran propiedades que son verdaderas dependiendo de que se cumplan unas condiciones.
 - Las reglas se definen sin pensar en la forma en que tomarán parte en la extracción de respuestas, es decir, sin pensar en la forma en la que se ejecutará el programa (significado procedural). Las reglas se definen pensando en su significado declarativo.
 - En Prolog la mayoría de los predicados se definen por inducción (recursión) basada en la estructura de los elementos del dominio de discurso (inducción estructural). Este estilo de definición de propiedades recibe el nombre de *definición por ajuste de patrones* en el contexto de la programación funcional.

- La mayoría de las implementaciones de Prolog no realizan comprobaciones de tipo, siendo esa tarea responsabilidad del programador.
 - Debido a la naturaleza puramente recursiva de las definiciones en el lenguaje Prolog, en muchas ocasiones, comprobar el tipo al que pertenecen los argumentos de los predicados que se definen puede afectar al rendimiento del programa. Suministrar una definición que sirva de interfaz para el usuario y que sea la encargada de realizar las comprobaciones de tipo, para después llamar a una función auxiliar que se inhibirá de esta clase de comprobaciones, puede solucionar el problema de la eficiencia.
 - Las comprobaciones de tipo imponen una direccionalidad en los cómputos que rompen la propiedad de inversibilidad (adireccionalidad) de ciertos predicados.
 - Los objetos matemáticos abstractos tienen una representación directa en Prolog, que nos permite definir predicados ejecutables que simulan su funcionamiento. El lenguaje Prolog está especialmente adaptado para resolver este tipo de problemas.
- Prolog puro no es pura lógica: debido a las peculiaridades del mecanismo operacional del lenguaje Prolog hay situaciones en las que se puede perder la corrección y la completitud del sistema. Por otra parte, el significado procedural de un programa puede cambiar al alterar el orden de las cláusulas en el programa o de los objetivos dentro de una cláusula.
 - Finalmente, hemos introducido los operadores aritméticos básicos y diversos operadores de comparación. La directiva “`: - op(Prioridad, Modo, Nombre).`” permite introducir operadores declarados por el usuario.
 - En Prolog, la aritmética no es declarativa ya que los operadores aritméticos no son inversibles: todos sus argumentos deben estar instanciados en la llamada.
 - En el lenguaje Prolog existen diversas clases de igualdad, ninguna de las cuales es una igualdad semántica: al no permitirse escribir cláusulas cuya cabeza tenga la igualdad como símbolo de predicado, es imposible imponer que dos términos que sintácticamente son distintos deban considerarse, en una deducción, iguales.

CUESTIONES Y EJERCICIOS

Cuestión 6.1 Describa las principales características del lenguaje Prolog e indique aquéllas que lo diferencian de la programación lógica.

Cuestión 6.2 El lenguaje Prolog:

- Es una realización correcta y completa del ideal de la programación lógica.
- Implementa un algoritmo de búsqueda en profundidad con vuelta atrás en el que cada variable, una vez instanciada, no se desinstancia si no se vuelve atrás, en el árbol de búsqueda, hasta el punto de elección en que la variable se instanció.
- Considera las instrucciones como un conjunto de cláusulas.
- Trata la aritmética de forma declarativa.

Seleccione la respuesta correcta de entre las anteriores.

Ejercicio 6.3 Exprese el conjunto de cláusulas del Ejercicio 4.22 en la notación del lenguaje Prolog. Ejecute el programa resultante, en un sistema Prolog, con el fin de establecer la insatisfacibilidad del conjunto de cláusulas original.

Ejercicio 6.4 Consulte el manual de su sistema Prolog y resuma las reglas que rigen la escritura de: a) identificadores; b) términos y c) hechos y reglas.

Ejercicio 6.5 Entre los objetos Prolog que se listan a continuación, ¿cuáles son sintácticamente correctos?, ¿cuáles de estos objetos son átomos, números, variables o estructuras?

a) Diana; b) diana; c) 'Diana'; d) _diana; e) mas_tonta(diana, carlos); f) 'Diana es mas tonta que Carlos'; g) 45; h) (x, y); i) +(norte, sur); j) tres(Tristes (tigres)).

Cuestión 6.6 Indique las principales características que diferencian el mecanismo operacional del lenguaje Prolog del principio de resolución SLD.

Ejercicio 6.7 Calcule el mgu de los siguientes átomos: $\text{append}([X|X_s], Y_s, [X|Z_s])$ y $\text{append}([b], [c, d], L)$.

Cuestión 6.8 Un árbol de búsqueda de Prolog es un árbol de búsqueda SLD en el que:

- La regla de computación selecciona el átomo más a la izquierda; la regla de búsqueda es en amplitud con vuelta atrás y la regla de ordenación selecciona las cláusulas en su orden textual de aparición.
- La regla de computación selecciona el átomo más a la izquierda; la regla de búsqueda es en profundidad con vuelta atrás y la regla de ordenación selecciona las cláusulas en su orden textual de aparición.
- La regla de computación selecciona el átomo más a la derecha; la regla de búsqueda es en profundidad con vuelta atrás y la regla de ordenación selecciona las cláusulas en su orden textual de aparición.

- La regla de computación selecciona el átomo más a la derecha; la regla de búsqueda es en amplitud con vuelta atrás y la regla de ordenación selecciona las cláusulas en su orden textual de aparición.

Seleccione la respuesta correcta de entre las anteriores.

Ejercicio 6.9 (Relaciones familiares) Edite el siguiente programa sobre relaciones familiares y resuelva diferentes objetivos (preguntas). Por ejemplo: ¿Cuáles son los hijos de Abraham?; ¿Lot es una mujer? Note que no hay parámetros predeterminados de entrada o salida (inversibilidad).

```
%%% HECHOS
h1: padre(abraham,isaac).
h2: padre(haran,lot).
h3: padre(haran,melca).
h4: padre(haran,jesca).
h5: hombre(isaac).
h6: hombre(lot).
h7: mujer(melca).
h8: mujer(jesca).

%%% REGLAS
r1: hijo(X,Y) :- padre(Y,X), hombre(X).
r2: hija(X,Y) :- padre(Y,X), mujer(X).
```

a) Dibuje el árbol de búsqueda para el anterior programa y el objetivo “?- hija (X,Y).”, etiquetando cada arco con el nombre de la cláusula empleada en el paso de resolución SLD y la sustición unificadora computada. b) Ahora, utilizando el comando “?- trace.”, realice una traza de la ejecución del objetivo “?- hija (X,Y).” con el fin de estudiar como se reproduce el árbol de búsqueda obtenido en el punto anterior (tecle “;
ENTER>” para que el interprete de Prolog continúe buscando soluciones). Recuerde que el comando “?- trace.” solamente muestra información sobre el subobjetivo seleccionado en cada paso de resolución (pero no sobre el objetivo completo).

Ejercicio 6.10 Dado el programa del Apartado 6.3.4, dibuje el árbol de búsqueda para el objetivo “?- acepta(s₂, [b,c]).”. Después realice una traza para verificar que el árbol se ha construido correctamente.

Ejercicio 6.11 (relaciones familiares) Realice una ampliación de la base de conocimiento (hechos) del Ejercicio 6.9 mediante la atenta lectura de estos fragmentos del Génesis:

- “He aquí la descendencia de Teraj: Teraj engendró a Abram (posteriormente llamado ‘Abraham’, que significa ‘Padre de multitud’), Najor, y Harán. Harán engendró a Lot La mujer de Abram se llamaba Sarai (o Sara) y la de Najor Melca, hija de Harán, madre de Melca y de Jesca.”

- “*Sarai, la mujer de Abram, no le había dado hijos, pero ella tenía una esclava egipcia de nombre Agar. ... tomó a Agar y se la dio por mujer a Abram, Agar parió un hijo a Abram y a este hijo tenido de Agar, Abram le llamó Ismael.*”
- *Estando en la tierra de Guerar, Abram confeso que: “es verdad que ella (Sarai) también es mi hermana, hija de mi padre pero no de mi madre, y ahora es mi mujer.”*
- *“Sara, pues, concibió y parió un hijo en su vejez, en el tiempo predicho por Dios. Y Abraham llamó al hijo que le nació Isaac (que significa ‘el que ríe’).”*
- *Isaac casó con Rebeca, “hija de Batuel, el que Melca parió a Najor” y hermana de Labán. Isaac y Rebeca tuvieron dos hijos, Esaú y Jacob, pero ésta es otra historia.*

Solo incluya como hechos los datos aportados sobre quién es hombre o mujer; padre o madre de alguien o casado con alguien. El resto de relaciones familiares deben definirse como reglas, en función de las anteriores. Tome como modelo las definiciones de los predicados hijo e hija del Ejercicio 6.9. Observe que ahora, al disponer de mayor información, las definiciones de los predicados hijo e hija deben de precisarse mejor. a) Más concretamente, defina las relaciones: “abuelo/a de”; “hermano/a de”; “tío/a de”; “sobrino/a de”; y “primo/a de”. Defina también un predicado que nos informe sobre las relaciones que hoy consideramos incestuosas. b) Señale las relaciones que sean directamente recursivas y las que sean indirectamente recursivas. c) Plantee las preguntas (objetivos) que se le ocurran, para extraer todo el conocimiento almacenado en esta pequeña base de datos.

Ejercicio 6.12 (Cuerpos celestes) *Convierta las cláusulas obtenidas en el Ejercicio 3.12 en un programa Prolog. Utilice el programa para responder a las siguientes preguntas: ¿quién orbita alrededor del sol? ¿es el sol un planeta? ¿cuales son los cuerpos celestes?*

Ejercicio 6.13 *Defina un predicado que calcule el factorial de un número natural escrito en notación de Peano.*

Ejercicio 6.14 *Defina un predicado que describa la relación existente entre un número natural escrito en notación de Peano y su cuadrado.*

Ejercicio 6.15 *Usando el lenguaje Prolog, defínase las siguientes operaciones sobre listas:*

1. *elimina(X, L1, L2), donde L2 es el resultado de eliminar todas las apariciones del elemento X en la lista L1.*

2. *sumList(L, S)*, donde *s* es la suma de los elementos que componen la lista *L* de números. Emplee parámetros de acumulación en la solución de este apartado. ¿Se obtiene recursión de cola? ¿Por qué esta solución es más eficiente que la que no emplea parámetros de acumulación?
3. *num2list(Número Lista)* que convierte un número entero en una lista de dígitos.
4. *intercalar(L1, L2, L)* que permita entrelazar dos listas *L1* y *L2* para obtener una tercera *L*. Por ejemplo, el resultado de entrelazar las listas [a,b,c] y [1,2,3] debe dar como resultado la lista [a,1,b,2,c,3] (en otros términos, *intercalar([a,b,c], [1,2,3], [a,1,b,2,c,3])* es verdadero).
5. *eliminaRepetidos(R, L)* que, dada una lista *R*, (posiblemente) con elementos repetidos y suministrada como entrada, produzca como salida una lista *L* en la que se han eliminado los elementos que estaban repetidos en *R*. Esto es, *eliminaRepetidos([1,1,2,3,4,3], [1,2,4,3])* es verdadero.
6. *rango(N1, N2, R)* que, dados dos números enteros *N1* y *N2*, genera una lista *R* con todos los enteros entre *N1* y *N2*.

Ejercicio 6.16 Completar el siguiente programa Prolog con la llamada a un predicado *q(X,X1,X2)* adecuado para que el programa determine si una lista de números es una joroba (es decir, es una secuencia de números formada por dos tramos: uno ascendente y otro descendente; por ejemplo, es una joroba la secuencia 1 4 7 8 5 3 2):

```
enoroba(X) :- q(X,X1,X2), sube(X1), baja(X2).  
  
sube([]).  
sube([U]).  
sube([U,V|Y]) :- U < V, sube([V|Y]).  
baja([]).  
baja([U]).  
baja([U,V|Y]) :- U > V, baja([V|Y]).
```

Cuestión 6.17 Considerar el siguiente programa Prolog, en el que se usa la llamada *put(32)*, cuyo efecto es escribir un espacio en blanco en pantalla,

```
tabulador(0).  
tabulador(N) :- put(32), N is N-1, tabulador(N).
```

y el objetivo: ?- *tabulador(8)*.

- Se produce un error de ejecución porque el argumento de *put* ha de ser un carácter.

- *Se produce un error de ejecución porque el primer argumento del metapredicado is ha de ser una variable no instanciada.*
- *La ejecución tiene como efecto una tabulación de ocho espacios.*
- *El objetivo fracasa finitamente.*

Seleccione la respuesta correcta de entre las anteriores.

Ejercicio 6.18 Una base de datos puede representarse de forma natural en Prolog como un conjunto de hechos. Por ejemplo, la información sobre una familia puede estructurarse como:

```
familia(persona(antonio, foix,
                fecha(7, febrero, 1950), trabajo(renfe, 1200)),
        persona(maria, lopez,
                fecha(17, enero, 1952), trabajo(sus_labores, 0)),
        [persona(patricia, foix,
                 fecha(10, junio, 1970), trabajo(estudiante, 0)),
         persona(juan, foix,
                 fecha(30, mayo, 1972), trabajo(estudiante, 0))]).
```

Podemos ver la anterior relación como una de las filas de una tabla que almacena las informaciones de las familias. La tabla FAMILIA estaría compuesta por los campos marido, esposa, hijos. Estos campos serían a su vez estructuras.

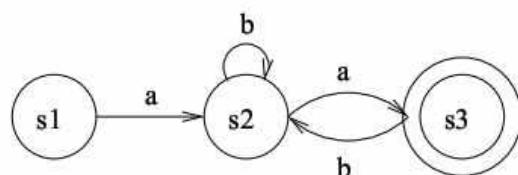
Mediante el lenguaje Prolog es muy fácil acceder a la información almacenada en una estructura, utilizando la potencia del mecanismo de unificación. Por ejemplo, podemos obtener los datos relativos al padre de una familia definiendo el predicado:

```
padre(Padre) :- familia(Padre, _, _).
```

Cuando se lanza el objetivo ?- padre(Padre) el mecanismo operacional de la resolución SLD unifica el predicado familia(Padre, _, _) (parcialmente instanciado) con los hechos de la base de datos, enlazando la variable Padre con los datos de las personas que son padres de familia. Así pues, vemos que el lenguaje Prolog está perfectamente adaptado como lenguaje de recuperación de información en una base de datos (deductiva).

a) Amplíe la base de datos con otras entradas y defina un procedimiento existe(P) que recupere los datos de una persona almacenados en la base de datos. b) Defina un predicado que obtenga el salario de una persona existente en la base de datos. c) Idem para el tipo de empleo y la fecha de nacimiento de una persona.

Ejercicio 6.19 Modele el comportamiento del autómata de la Figura 6.9, suponiendo que el estado s_3 es el estado final. Defina una relación acepta(Estado, Cadena) que sea verdadera cuando la cadena sea aceptada por el autómata. Una cadena se considera aceptada cuando, partiendo del estado actual, produce una serie de transiciones que llevan el autómata al estado final.

**Figura 6.9** Un autómata finito no determinista.

Ejercicio 6.20 (Celos en el Puerto) *Convierta las cláusulas obtenidas en el Ejercicio 3.11 en un programa Prolog. Utilice el programa para responder a la siguiente pregunta: ¿hay porteños burlados y felices?*

Este ejercicio muestra que, aunque un programa sea declarativamente correcto, puede tener un comportamiento procedural incorrecto, aun después de reordenar parte de sus cláusulas. En el caso que nos ocupa, el comportamiento procedural incorrecto se debe a que el programa entra por una rama infinita, computando siempre la misma respuesta. Este comportamiento impide que se compute el resto de respuestas correctas. Compruébese.

(Observación: Las respuestas correctas para este programa y objetivo, son `cornelio` y `cete`. Sin embargo solamente se computa la primera de las respuestas correctas. Véase el Ejercicio 7.6, dónde se sugiere una posible solución a este problema.)

Ejercicio 6.21 Defina mediante el lenguaje Prolog las siguientes operaciones:

1. `maxlist(Lista, Maximo)` que obtiene el `Maximo` de una `List`a de números.
2. `fib(N,M)` que computa los elementos de la sucesión de Fibonacci: $a_1 = 1$, $a_2 = 1$, y $a_n = a_{n-1} + a_{n-2}$ si $n > 2$ (Emplee parámetros de acumulación para aumentar la eficiencia). ¿Se obtiene recursión de cola? ¿Por qué esta solución es más eficiente que la que no emplea parámetros de acumulación?
3. `nat2int(N, I)` que convierte un natural `N` (expresado en notación de Peano) en el número entero `I` que le corresponde.
4. `between que, dados dos enteros N_1 y N_2 , genera todos los enteros X tales que $N_1 \leq X \leq N_2$.`
5. `brand` que obtiene los números de Brandreth (aquellos números cuyo cuadrado tiene cuatro cifras -por ejemplo, el 45 cuyo cuadrado es 2025- y tales que, cuando se parte su cuadrado en dos “segmentos” iguales, se obtienen dos números, cada uno de dos dígitos, que sumados coinciden con el número original (es el caso del 2025: la suma de 20 y 25 es el número 45 original). (Ayuda: los números `N` cuyo cuadrado es de cuatro cifras pueden generarse mediante una llamada al predicado `between(32, 99, N)`).

6. *brand2*, que extiende el predicado *brand* de forma que puedan hallarse todos los números *N* de Brandreth entre 1 y un cierto límite *L*. (**Ayuda:** defina un predicado capaz de contar el número de dígitos de un número).
7. *factorial(N, F)* que calcula el factorial *F* de un número entero *N* mayor o igual que cero.

Cuestión 6.22 Determine si el siguiente programa Prolog

```
{maximo(X, Y, X) :- X ≥ Y.    maximo(X, Y, Y).},
```

es o no correcto respecto a la siguiente especificación

maximo(X, Y, Max) ⇔ Max es el máximo de X e Y.

Ejercicio 6.23 Defina un predicado *partir(L, L1, L2)* que divida la lista *L* en dos partes *L1* y *L2*, tales que los elementos de *L1* son menores o iguales que un cierto elemento *N* perteneciente a *L* y los de *L2* son mayores que ese elemento *N*. El elemento *N* seleccionado no se incluye en las listas partidas *L1* y *L2*.

Ejercicio 6.24 (Ordenación rápida —quicksort—) El algoritmo de ordenación rápida aplica la estrategia de “divide y vencerás” a la tarea de ordenar una lista. La idea es particionar una lista con respecto a uno de sus elementos (en principio elegido al azar), llamado el pivote, de forma que los elementos menores o iguales que el pivote queden agrupados a su izquierda, en una de las listas, y los elementos mayores que el pivote queden agrupados a su derecha, en la otra lista. Observe que, tras la partición, lo único seguro es que el pivote está en el lugar que le corresponderá en la ordenación final. Entonces, el algoritmo se centra en la ordenación de las porciones de la lista (que no están necesariamente ordenadas), lo que nos remite al problema original. Utilizando el predicado *partir(L, L1, L2)* del Ejercicio 6.23, dé una implementación recursiva del algoritmo de ordenación rápida.

Ejercicio 6.25 (Mezcla ordenada —merge-sort—) Implemente en Prolog el algoritmo de mezcla ordenada para la ordenación de una lista de elementos. Informalmente, este algoritmo puede formularse como sigue: Dada una lista, divídase en dos mitades, ordene cada una de las mitades y, después, “mezcle” apropiadamente las dos listas ordenadas obtenidas en el paso anterior. Compare su solución con la propuesta en [114, pag. 276].

Ejercicio 6.26 (Problema de las torres de Hanoi) En este juego se utilizan tres postes y un grupo de discos. Los discos tienen diferentes diámetros y se introducen en los postes por un agujero practicado en el centro de cada disco. Los discos se encuentran inicialmente en el poste izquierdo (ordenados por tamaño, estando en la base el mayor y en la

cima el menor). El juego consiste en mover todos los discos al poste derecho, utilizando el central como poste auxiliar. Deben observarse dos restricciones: a) sólo se puede mover un disco cada vez de uno a otro poste, esto es, el disco superior de un poste; b) ningún disco puede moverse sobre otro de diámetro más pequeño.

Defina un predicado `hanoi(N, Plan)` que almacene en `Plan` la secuencia de movimientos necesaria para trasladar una torre de `N` discos desde el poste de la izquierda hasta el de la derecha. El plan puede representarse como una lista de listas, compuesta por elementos de la forma “[mover, disco, del, poste, X, al, Y]”.

Ejercicio 6.27 Implemente un programa capaz de multiplicar matrices cuadradas (esto es, matrices de orden $n \times n$).

Ejercicio 6.28 Amplíe y adapte los predicados definidos en el Ejercicio 6.27 para que sea posible multiplicar matrices de un número arbitrario de filas y columnas. Naturalmente, el programa solamente puede tener éxito si el número de columnas de la primera matriz es igual al número de filas de la segunda.

Ejercicio 6.29 El polinomio $C_nx^n + \dots + C_2x^2 + C_1x + C_0$, donde C_n, \dots, C_1, C_0 son coeficientes enteros, puede representarse en Prolog mediante el siguiente término:

`Cn * x ** n + ... + c2 * x ** 2 + c1 * x + c0.`

El operador “`**`” es un operador binario infijo. Cuando se evalúa la expresión “`x ** n`”, computa la potencia enésima de `x`. Observe que el operador “`**`” liga más que el operador binario infijo “`*`”, que a su vez liga más que el operador binario infijo “`+`” (por lo tanto no se requiere el uso de paréntesis). Observe también que, en la representación anterior, la variable `x` se trata como una constante. Defina un predicado `eval(P, V, R)` que devuelva el resultado `R` de evaluar un polinomio `P` para un cierto valor `V` de la variable `x`. A modo de ejemplo, el objetivo `?- eval(5 * x ** 2 + 1, 4, R).` debe tener éxito con respuesta `R = 81`.

Ejercicio 6.30 Defina dos operadores, para la suma y sustracción de números, que permitan escribir expresiones aritméticas simples con un lenguaje similar al natural. Esto es, expresiones como: 12 mas 3 es S, 8 menos 2 es R o 3 mas 2 es 5.

Ejercicio 6.31 Amplíe el Ejemplo 6.13 de forma que sea posible responder a preguntas del estilo de:

```
?- borrar 3 de [1,3,5,7] da [1,5,7].  
?- borrar 5 de [1,3,5,7] da L.  
?- borrar I de [1,3,5,7] da [1,3,5].
```

Ejercicio 6.32 Definanse los siguientes predicados acerca de los árboles binarios:

1. `peso(A)` que calcule el peso de un árbol binario `A`, entendiendo por “peso” el número de nodos que contiene dicho árbol.
2. `frontera(A, F)` que permite determinar la frontera `F` del árbol binario `A`. La frontera de un árbol es la lista de sus hojas.
3. `preorden(A, L)` que permita recorrer los nodos del árbol binario `A` en orden preorden, obteniendo la lista `L` de nodos visitados y en el orden que fueron visitados. Un recorrido preorden consiste en visitar primero la raíz y después los subárboles izquierdo y derecho.
4. `postorden(A, L)` que permita recorrer los nodos del árbol binario `A` en orden postorden, obteniendo la lista `L` de nodos visitados y en el orden que fueron visitados. Un recorrido postorden consiste en visitar los subárboles izquierdo y derecho antes de visitar la raíz.

Ejercicio 6.33 Defina en Prolog las siguientes relaciones para árboles binarios de búsqueda (diccionarios):

1. `maxEle(D, E)` que calcula el mayor elemento, `E`, almacenado en el diccionario `D`;
2. `minEle(D, E)` que calcula el menor elemento, `E`, almacenado en el diccionario `D`;
3. `eliminar(E, D1, D2)`, que devuelve el diccionario binario `D2` resultado de eliminar un elemento `E` en un diccionario binario inicial `D1`. (**Ayuda:** no es necesario que el árbol resultante esté equilibrado).

Ejercicio 6.34 (Arboles equilibrados) Al igual que sucede en el caso de los árboles binarios, la inserción y eliminación de elementos en un diccionario puede hacer que degeneren en una lista. Este problema puede evitarse si los árboles resultantes se mantienen equilibrados cuando se inserta o elimina un elemento. Un árbol está perfectamente equilibrado si, para cada nodo, el número de nodos del subárbol izquierdo y el número de nodos del subárbol derecho, difieren como mucho en una unidad. Existen otras nociones de “equilibrio” menos exigentes, en cuanto al coste que acarrea mantener ese criterio de “equilibrio” cuando se realizan inserciones o borrados indiscriminados. Uno de los criterios de equilibrio imperfecto más adecuados, para facilitar su posterior restauración, es el siguiente: Un árbol está equilibrado si, y sólo si, para cada uno de sus nodos, las alturas de sus dos subárboles difieren como mucho en uno. Este último tipo de árboles también se denominan árboles AVL (en honor a los formuladores del anterior criterio, Adelson-Velskii y Landis). Defina en Prolog las siguientes relaciones para diccionarios:

1. *eliminar(E, D1, D2), que devuelve el diccionario binario D2 resultado de eliminar un elemento E en un diccionario binario inicial D1;*

2. *insertar(E, D1, D2), que inserta un elemento E en el diccionario D1 para obtener el diccionario binario D2;*

pero asegúrese de que el diccionario binario D2 se mantiene equilibrado, usando alguna de las anteriores nociones de equilibrio.

Ejercicio 6.35 *Adapte los predicados definidos en el Ejercicio 6.32 para que puedan emplearse con árboles n-arios.*

El Lenguaje Prolog: Aspectos Avanzados

Con el fin de aumentar su expresividad, además de los predicados para manipulaciones aritméticas, el lenguaje Prolog posee una serie de *predicados extralógicos*. Contrariamente a los predicados de Prolog puro, que se definen mediante reglas, los predicados extralógicos realizan tareas que no pueden especificarse fácilmente mediante fórmulas lógicas. Esencialmente, existen tres clases de estos predicados:

1. Predicados que facilitan el control del modo de ejecución de un programa Prolog. El más representativo de estos predicados es el operador de corte.
2. Predicados para la gestión de la base de datos interna del sistema Prolog. Pueden añadir o eliminar cláusulas a la base de datos interna, como los predicados para la manipulación de cláusulas, o actualizar ciertos parámetros globales.
3. Predicados del sistema, que interaccionan con el sistema operativo para realizar una entrada/salida o acceder a los recursos del computador.

Mientras que los predicados de Prolog puro solamente cambian los datos que se pasan como argumentos, los predicados extralógicos pueden producir *efectos laterales*: pueden alterar datos ocultos (que definen el estado del sistema) o estructuras que no se especifican en los argumentos.

En este capítulo continuamos nuestro estudio del lenguaje Prolog discutiendo algunas de las características extralógicas mencionadas y ciertos aspectos avanzados del lenguaje.

7.1 EL CORTE, CONTROL DE LA VUELTA ATRÁS Y ESTRUCTURAS DE CONTROL

En Prolog, el control es fijo y viene dado de forma automática por las peculiares características e implementación de este lenguaje: i) recursión, como única forma de propiciar la repetición; ii) unificación, como único medio de enlazar valores a las variables; y iii) vuelta atrás automática, que permite reevaluar objetivos, en busca de nuevas alternativas, después de producirse un fallo. Hasta ahora las únicas herramientas de control que se han empleado han consistido en reordenar los objetivos dentro del cuerpo de una cláusula y el propio orden de las cláusulas dentro del programa. En este apartado estudiaremos las facilidades de control que proporciona Prolog.

7.1.1. El Corte

Prolog proporciona un predicado extralógico, denominado *el corte*, denotado como “!”, que permite podar el espacio de búsqueda impidiendo la vuelta atrás automática. Conviene utilizarlo si deseamos optimizar el tiempo de ejecución, la gestión de memoria o evitar soluciones que no interesan. Este predicado siempre tiene éxito y falla al intentar resatisfacerlo. Cuando el corte aparece en una cláusula su función es impedir la reevaluación de todos los subobjetivos que quedan a su izquierda en la cláusula de la que forma parte. Esto incluye la cabeza de la cláusula, que ya no intentará resatisfacerse. Su forma de actuar es la siguiente: cuando al retroceder por una rama del árbol de búsqueda se alcanza el corte, todas las posibles alternativas de los puntos de elección entre el *objetivo padre* (el objetivo que lanzó la cláusula que contiene el corte) y la posición en la que se ejecutó el corte son eliminadas. Por lo tanto, cuando fracasa un subobjetivo que se encuentra a la derecha del corte y, con la vuelta atrás, se retrocede hasta él, su efecto es que fracasa el objetivo padre. Dentro de una cláusula que contiene “;”, el corte elimina el resto de las alternativas. Para ilustrar el funcionamiento y como produce la poda del árbol de búsqueda estudiamos un ejemplo concreto.

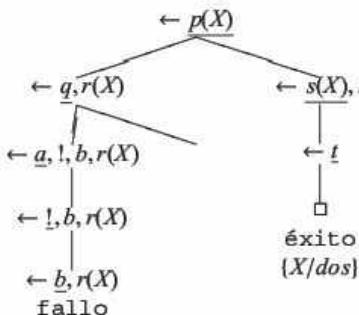
Ejemplo 7.1

Dado el programa

```
p(X) :- q, r(X).
p(X) :- s(X), t.
q :- a, !, b.
q :- c, d.
r(uno).
```

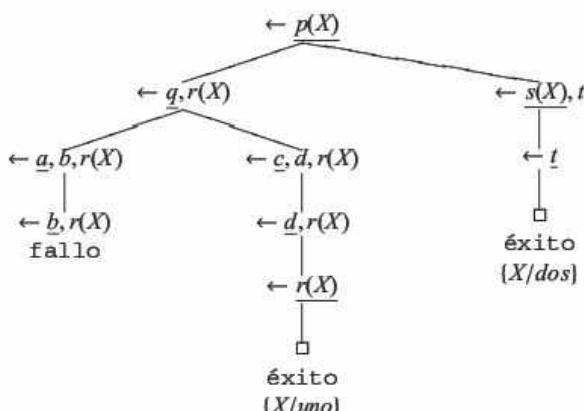
El árbol correspondiente al objetivo “?- p(X).” se muestra en la Figura 7.1. Una vez producido el *fallo*, puede apreciarse que el efecto del corte en este programa es eliminar la segunda alternativa para el predicado *q* en el punto de elección relativo al objetivo padre “*- q, r(X)*”, de forma que la vuelta atrás lleva a recorrer la rama de

éxito más a la derecha. Por consiguiente, el efecto, en este caso, ha sido la poda de una rama de éxito conducente a la respuesta $x=1$ (véase la Figura 7.2, donde se muestra el árbol de búsqueda para el mismo objetivo cuando se ha eliminado el corte del programa). En general, si hubiesen habido otras alternativas entre el punto en el que se ejecuta el corte y el objetivo padre “ $\leftarrow q, \quad r(X)$ ”, también habrían sido desechadas.



[Árbol de búsqueda para el objetivo
“ $\leftarrow p(X)$ ” y el programa del Ejemplo 7.1.]

Figura 7.1 Efecto del corte en el árbol de búsqueda (i).



[Árbol de búsqueda para el objetivo “ $\leftarrow p(X)$ ” cuando
se elimina el corte en el programa del Ejemplo 7.1.]

Figura 7.2 Efecto del corte en el árbol de búsqueda(ii).

Para comprender los efectos del corte sobre la eficiencia de los programas estudiamos el siguiente ejemplo, adaptación del que aparece en [21].

Ejemplo 7.2

Sea la función escalón definida por intención como:

$$f(x) = \begin{cases} 0 & \text{si } (x < 3); \\ 2 & \text{si } (3 \leq x \wedge x < 6); \\ 4 & \text{si } (6 \leq x). \end{cases}$$

Esta función puede expresarse en Prolog, de forma inmediata, definiendo una relación binaria $f(X, Y)$. Podemos pensar en las siguientes alternativas de programación:

```
% Alternativa 1
f1(X, 0) :- X < 3.
f1(X, 2) :- X >= 3, X < 6.
f1(X, 4) :- X >= 6.

% Alternativa 2
f2(X, 0) :- X < 3, !.
f2(X, 2) :- X >= 3, X < 6, !.
f2(X, 4) :- X >= 6.
```

Para comprender los efectos del corte sobre la eficiencia, vamos a comparar el comportamiento de ambas alternativas ante el objetivo “?- $f(1, Y), 2 < Y$.”. Apreciamos, mediante traza, el siguiente comportamiento del primer programa con respecto a este objetivo:

```
?- trace.
?- f1(1, Y), 2 < Y.
<Spy>: (1:0) Call: f1(1, _0) . :
<Spy>: | (2:1) Call: 1 < 3. :
<Spy>: | (2:1) Exit: 1 < 3. :
<Spy>: (1:0) Exit: f1(1, 0) . :
<Spy>: (1:1) Call: 2 < 0. :
<Spy>: (1:1) Fail: 2 < 0. :
<Spy>: (1:0) Redo: f1(1, _0) . :
<Spy>: | (2:1) Call: 1 >= 3. :
<Spy>: | (2:1) Fail: 1 >= 3. :
<Spy>: (1:0) Redo: f1(1, _0) . :
<Spy>: | (2:1) Call: 1 >= 6. :
<Spy>: | (2:1) Fail: 1 >= 6. :
no
```

El primer subobjetivo “ $f1(1, Y)$ ” se satisface con la regla 1 y la sustitución { $Y/0$ }. Después se intenta satisfacer el segundo de los subobjetivos “ $2 < 0$ ” pero falla. Mediante el mecanismo de vuelta atrás se intenta (re)satisfacer el subobjetivo inicial, pero nuevamente falla porque el subobjetivo $f1(1, Y)$ no tiene éxito con las reglas 2 y 3. Al no haber una respuesta, el mecanismo operacional de vuelta atrás obliga a recorrer todas

las ramas del árbol de búsqueda de Prolog (en busca de una respuesta que finalmente no encuentra). Lo que sucede es que las tres reglas son mutuamente excluyentes y cuando se tiene éxito con una se fracasa con las otras. Por lo tanto, en una definición como la anterior resulta útil no permitir la vuelta atrás haciendo uso del corte. La Figura 7.3 muestra el árbol de búsqueda asociado al objetivo “?- f1(1, Y), 2 < Y.”. Al igual que en la secuencia de pasos que constituyen la traza, en los nodos del árbol de búsqueda solo mostramos los subobjetivos seleccionados.

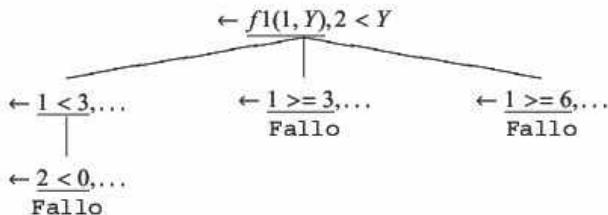


Figura 7.3 Árbol de búsqueda para el programa y objetivo del Ejemplo 7.2: Alternativa sin corte.

Si lanzamos el mismo objetivo, pero ahora haciendo uso de la definición que se ha planteado como segunda alternativa, se produce el siguiente comportamiento:

```

?- f2(1, Y), 2 < Y.
<Spy>: (1:0) Call: f2(1, _0) . :
<Spy>: | (2:1) Call: 1 < 3. :
<Spy>: | (2:1) Exit: 1 < 3. :
<Spy>: | (2:1) Call: !. :
<Spy>: | (2:1) Exit: !. :
<Spy>: (1:0) Exit: f2(1, 0) . :
<Spy>: (1:1) Call: 2 < 0. :
<Spy>: (1:1) Fail: 2 < 0. :
no
  
```

que resulta mucho más eficiente, por la poda de ramas que sabemos con seguridad que conducirán a un fallo. La Figura 7.4 muestra el árbol de búsqueda asociado al objetivo “?- f2(1, Y), 2 < Y.”.

Si el objetivo hubiese sido “f(6, Y), 8 < Y”, no se habría apreciado mejoría en el comportamiento de la alternativa 2 con respecto a la 1, ya que la definición de la función que hace uso del corte se comporta de forma semejante a la construcción *case* de los lenguajes convencionales, que comprueba todas las alternativas antes de desecharlas.

Se han identificado dos tipos de cortes: i) los denominados *cortes verdes*, que no afectan al significado declarativo del programa, de modo que el programa sigue computando las mismas respuestas que computaba cuando no había cortes; ii) los *cortes rojos*, que sí afectan al significado declarativo del programa original, de modo que el uso del corte puede hacer que se pierdan respuestas.

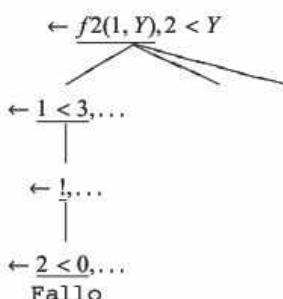


Figura 7.4 Árbol de búsqueda para el programa y objetivo del Ejemplo 7.2: Alternativa con corte.

Ejemplo 7.3

El corte introducido en el programa del Ejemplo 7.1 es rojo, ya que cambia la semántica del programa (al eliminar la rama que conduce a la respuesta $\{X/\text{uno}\}$).

El corte introducido en la alternativa 2 del Ejemplo 7.2 es verde, ya que no cambia la semántica del programa (i.e. se producen las mismas respuestas con o sin corte).

El corte, en general, y los cortes rojos, en particular, deben utilizarse con sumo cuidado ya que, en ocasiones, este predicado puede hacer que un programa no funcione o que, como hemos visto, perdamos soluciones válidas al podar ramas del árbol de búsqueda SLD.

El corte es una impureza del lenguaje ya que anula una de las ventajas de la Programación Lógica: la posibilidad de centrarse en los aspectos lógicos de la solución sin considerar los aspectos de control. El corte hace ilegible los programas, al impedir la lectura declarativa de los mismos y, en muchos sentidos, juega el mismo papel que la construcción “`goto`” en los lenguajes imperativos. Una regla de buen estilo en programación declarativa es “evitar el corte siempre que sea posible”, aunque en ocasiones puede ser imprescindible.

Observación 7.1 *Como se ha dicho en el párrafo anterior, el corte constituye una herramienta muy valiosa en la práctica para la implementación efectiva de algunas especificaciones. Véase el Ejercicio 7.4, que incide en esta cuestión y la aclara.*

7.1.2. Otros predicados de control: `false` y `true`

Prolog incorpora otras facilidades de control, como son los objetivos `true`, el objetivo que siempre tiene éxito, y `false`, el objetivo que siempre falla y fuerza la vuelta atrás. Esta particularidad de activar el mecanismo de vuelta atrás que tiene `false` le hace muy útil a la hora de forzar la obtención de todas las respuestas a un problema.

Ejemplo 7.4

Consideremos la pequeña base de datos de países

```
país('Alemania').
país('España').
país('Francia').
país('Gran Bretaña').
país('Italia').
```

El predicado países sirve para listar el contenido de la base de datos

```
paises :- país(P), write(P), nl, fail.
paises.
```

Cuando se lanza el objetivo “?- países.”, unifica en primer lugar con la cabeza de la primera regla que define el predicado países. En el objetivo que se deriva tras este paso, el subobjetivo país(P) unifica con el hecho país('Alemania'), de forma que la variable P se enlaza con el nombre del primer país almacenado, i.e. 'Alemania'. Se escribe por pantalla el nombre y finalmente se ejecuta fail produciendo un fallo. El fallo fuerza la vuelta atrás, con lo que sucesivamente se irán escribiendo el resto de los nombres almacenados (véase la Figura 7.5). El papel de fail es esencial para lograr este efecto: si eliminamos fail de la definición de países el objetivo tendrá éxito de inmediato y solo se mostrará el nombre del primer país almacenado en la base de datos.

Nótese que el listado de los nombres es un efecto lateral de la definición del predicado países y que la respuesta al objetivo “?- países.” es “yes.”. Para lograr esta respuesta ha sido necesario introducir una segunda cláusula en la definición del predicado “paises”. El lector puede comprobar el efecto que tiene la supresión de esta segunda cláusula en la respuesta obtenida.

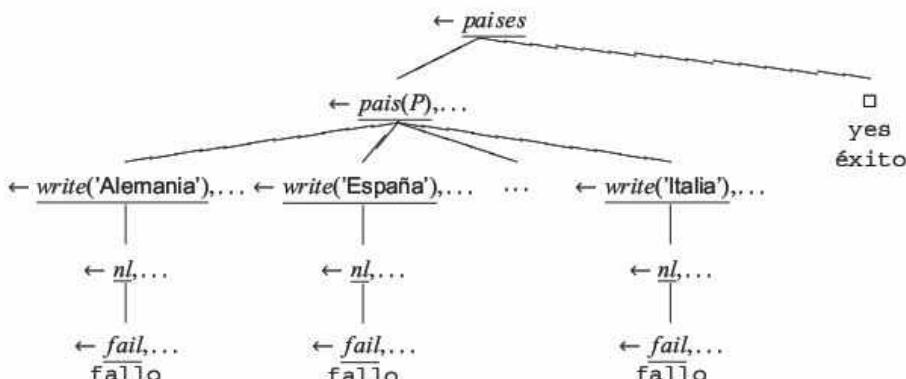


Figura 7.5 Árbol de búsqueda para el programa y el objetivo del Ejemplo 7.4.

7.1.3. Estructuras de control

La combinación del corte y los objetivos `fail` y `true` puede emplearse para definir una forma especial de negación (como se verá en el Apartado 7.2.3) y simular algunas estructuras de control de los lenguajes convencionales.

La construcción `if_then_else`

La conocida construcción `if_then_else` de los lenguajes convencionales puede simularse de varias formas haciendo uso del corte. La manera que se considera estándar es la reflejada en la siguiente definición:

```
% version 1
% ifThenElse1(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse1(Condicion, Accion1, _) :- Condicion, !, Accion1.
ifThenElse1(_, _, Accion2) :- Accion2.
```

En esta definición es conveniente notar que las variables `Condicion`, `Accion1` y `Accion2` son metavariables que pueden sustituirse por cualquier llamada a un predicado. La definición presentada en la versión 1 admite una variante totalmente equivalente pero más compacta:

```
% version 1
% ifThenElse1(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse1(Condicion, Accion1, Accion2) :- Condicion, !, Accion1;
                                         Accion2.
```

Una forma más declarativa de implementar el predicado `ifThenElse` se consigue haciendo uso del predicado `not`:

```
% version 2
% ifThenElse2(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse2(Condicion, Accion1, Accion2) :- Condicion, Accion1,
                                         not(Condicion), Accion2.
```

Sin embargo esta alternativa es menos eficiente ya que la condición se comprueba dos veces. También es conveniente notar que el comportamiento operacional es distinto cuando el corte va al final, como en la siguiente versión:

```
% version 3
% ifThenElse3(Condicion, Accion1, Accion2),
% if Condicion then Accion1 else Accion2
ifThenElse3(Condicion, Accion1, _) :- Condicion, Accion1, !.
ifThenElse3(_, _, Accion2) :- Accion2.
```

Las versiones 1 y 2 son completamente equivalentes (si Accion1 puede producir varias respuestas, las producirá cuando se cumpla la condición). La versión 3 difiere de las anteriores en que, cuando la condición se cumple, Accion1 solo produce una respuesta, ya que el corte hace que se eliminen las otras posibles alternativas.

Ejemplo 7.5

Sea el programa

```
% Un programa artificial para estudiar el comportamiento de las
% diferentes versiones del predicado ifThenElse
acc1(X) :- X = 1.
acc1(X) :- X = 2.
acc2(X) :- X = 3.
acc2(X) :- X = 4.
```

que consideramos completado con las definiciones del predicado `ifThenElse` introducidas más arriba. Una sesión en un intérprete de Prolog produce estos resultados:

```
?- ifThenElse1(true, acc1(X), acc2(Y)).
X = 1, Y = _1 ;
X = 2, Y = _1 ;
no
?- ifThenElse1(fail, acc1(X), acc2(Y)).
X = _0, Y = 3 ;
X = _0, Y = 4 ;
no
?- ifThenElse2(true, acc1(X), acc2(Y)).
X = 1, Y = _1 ;
X = 2, Y = _1 ;
no
?- ifThenElse2(fail, acc1(X), acc2(Y)).
X = _0, Y = 3 ;
X = _0, Y = 4 ;
no
?- ifThenElse3(true, acc1(X), acc2(Y)).
X = 1, Y = _1 ;
no
?- ifThenElse3(fail, acc1(X), acc2(Y)).
X = _0, Y = 3 ;
X = _0, Y = 4 ;
no
```

En la mayoría de los sistemas Prolog (e.g., SICStus Prolog, IBM Prolog o AAIIS Prolog), la construcción `if_then_else` es un operador predefinido que se denota mediante el símbolo “`->`”. La combinación “Condicion `->` Accion1; Accion2” debe leerse como “if Condicion then Accion1 else Accion2”. El operador “`->`” suele definirse conforme a la versión 1 del predicado `ifThenElse`. Por consiguiente, es equivalente a la construcción “(Condicion, !, Accion1; Accion2)”. Con más precisión, el operador “`->`” se define en algunos sistemas como:

```
Condicion -> Accion1 ; _ :- Condicion, !, Accion1.  
Condicion -> _ ; Accion2 :- !, Accion2.  
Condicion -> Accion :- Condicion, !, Accion.
```

Observe que, en dichos sistemas, las relaciones de precedencia entre los operadores “->” y “;” se establecen de manera que la expresión “Condicion -> Accion1 ; Accion2” sea tratada como un término de la forma “;(->(Condicion, Accion1), Accion2)”. Entonces, las dos primeras cláusulas son parte de la definición del operador “;”, mientras que solo la última define el operador “->”.

La construcción case

El operador “->” puede encadenarse para simular la construcción case de los lenguajes convencionales.

```
case :- ( Condicion1 -> Accion1;  
          Condicion2 -> Accion2;  
          :;  
          CondicionN -> AccionN;  
          OtroCaso).
```

que es una construcción más clara y fácil de entender que la siguiente equivalente

```
case :- ( Condicion1, !, Accion1;  
          Condicion2, !, Accion2;  
          :;  
          CondicionN, !, AccionN;  
          OtroCaso).
```

en la que se emplea el corte directamente.

Ejemplo 7.6

La función escalón definida en el Ejemplo 7.2 puede implementarse en Prolog mediante una estructura case:

```
f(X, Y) :- (X < 3 -> Y=0;  
            3 =  
            < X, X < 6 -> Y=2;  
            otherwise -> Y=4).
```

Donde “otherwise” es un predicado predefinido que siempre se evalúa a verdadero cuando se invoca.

El predicado repeat

Muchos sistemas Prolog incorporan el predicado predefinido `repeat`. Como objetivo, `repeat` siempre tiene éxito, incluso cuando Prolog intenta resatisfacerlo. Este predicado se comporta como si estuviese definido por las siguientes cláusulas:

```
repeat.
repeat :- repeat.
```

Su característica esencial es que es indeterminista. Por consiguiente, cada vez que se alcanza, con ayuda del mecanismo de vuelta atrás, genera otra rama de ejecución alternativa. Una forma típica de utilizar `repeat` se ilustra en el siguiente ejemplo.

Ejemplo 7.7

El procedimiento `hacer_tratamiento`, repetidamente lee un término del flujo de entrada estándar (CIS —Véase el Apartado 7.3.1) y lo trata, hasta que se encuentra un término `stop`.

```
hacer_tratamiento :-
repeat,
read(X),
( X = stop, !
;
tratamiento(X),
fail
).
```

donde `tratamiento(X)` es un predicado que realiza el tratamiento específico.

Aunque la sintaxis puede cambiar según la implementación del sistema Prolog de que se trate, otros predicados predefinidos para el control, además de los ya mencionados son:

1. `quit` o `abort`, que abortan cualquier objetivo activo, devolviendo el control al sistema (i.e., volvemos al prompt del sistema “?-”).
2. `exit` o `halt` nos sacan del propio intérprete, terminando la sesión en curso.

Observaciones 7.1

1. Como se ha comentado, el corte es comparable a la construcción “`goto`” de los lenguajes convencionales. Si bien es una herramienta poderosa, su proliferación puede destruir la estructura declarativa de los programas y hacer difícil su lectura y mantenimiento. Cuando es necesario usar el corte, la mejor forma de hacerlo es confinándolo en el interior de operadores que admitan una lectura más clara. Ejemplo de estos tipos de operadores serían “`not`” (que se define en el Apartado 7.2.3) y “`->`”.

2. El corte induce una visión procedural del lenguaje Prolog, como revelan los intentos de clarificar su uso mediante la definición de operadores como “ $->$ ”. En general, la visión procedural de Prolog no resulta positiva, ya que suele provocar que se utilice Prolog como un lenguaje imperativo (para lo cuál ya disponemos de una gran variedad de lenguajes bastante más adecuados).



7.2 NEGACIÓN

7.2.1. El problema de la información negativa

Para sistemas de cláusulas de Horn definidas es imposible extraer información negativa utilizando la única regla de deducción que nos está permitida, la estrategia de resolución SLD. Informalmente, esto ocurre porque ninguna cláusula del programa $L \leftarrow C$ tiene como cabeza L un literal negativo; por tanto, la resolución de cualquier premisa C de una cláusula solo permite inferir como conclusión el correspondiente átomo positivo en cabeza de ésta. Este hecho se formaliza mediante la siguiente proposición.

Proposición 7.1 Sea Π un programa definido y \mathcal{A} un átomo de la base de Herbrand de Π . Se cumplen los siguientes enunciados, que son equivalentes:

1. El literal $\neg\mathcal{A}$ no es consecuencia lógica del programa Π .
 2. El conjunto de cláusulas $\Pi \cup \{\mathcal{A}\}$ es satisfacible.
 3. No existe una refutación SLD para $\Pi \cup \{\mathcal{A}\}$.
-

Para justificar la primera de estas afirmaciones basta pensar que la base de Herbrand del programa $\mathcal{B}_L(\Pi)$ es una interpretación que necesariamente es modelo del programa pero, al no contener átomos negados, no puede ser modelo del átomo $\neg\mathcal{A}$. Así pues, por definición de consecuencia lógica, $\neg\mathcal{A}$ no puede ser consecuencia lógica del programa Π . Para justificar la segunda afirmación, piénsese que la propia base de Herbrand $\mathcal{B}_L(\Pi)$ es modelo de \mathcal{A} , ya que, por hipótesis, $\mathcal{A} \in \mathcal{B}_L(\Pi)$. Consiguientemente, la interpretación $\mathcal{B}_L(\Pi)$ es un modelo de $\Pi \cup \{\mathcal{A}\}$, i.e., $\Pi \cup \{\mathcal{A}\}$ es satisfacible. Finalmente, por el Teorema 5.2, existe una refutación SLD para $\Pi \cup \{\mathcal{A}\}$ si y solo si $\Pi \cup \{\mathcal{A}\}$ es insatisfacible. Teniendo en cuenta que $\Pi \cup \{\mathcal{A}\}$ es satisfacible como acabamos de ver, concluimos que no existe una refutación SLD para $\Pi \cup \{\mathcal{A}\}$.

Pueden alegarse dos razones a la hora de justificar la necesidad de tratar el problema de la negación:

1. Es deseable, sobre todo en el contexto de las bases de datos deductivas, interpretar como verdadera la negación de la información que no se suministra explícitamente en los programas definidos. Por ejemplo, dado el programa Π compuesto por el conjunto de hechos

```
carnívoro(gato) ←
carnívoro(perro) ←
herbívoro(cabra) ←
herbívoro(oveja) ←
herbívoro(vaca) ←
herbívoro(conejo) ←
```

Sería interesante que informaciones que no se suministran en el programa, como

herbívoro(gato) ←, herbívoro(perro) ←,

pudiesen interpretarse como una afirmación de que los hechos

$\neg \text{herbívoro(gato)} \leftarrow, \quad \neg \text{herbívoro(perro)} \leftarrow,$

son verdaderos.

Sin embargo, como afirma la Proposición 7.1 no puede extraerse información negativa de un programa definido, así que habrá que arbitrar formas para solucionar este problema.

2. Como se ha puesto de manifiesto en el Apartado 6.3.2, además de ampliar la clase de respuestas que pueden obtenerse de un programa definido, también es importante tratar la negación porque, en ocasiones, es preciso dotar a los programas de una mayor expresividad, permitiendo la presencia de literales negados en el cuerpo de las cláusulas. Esto implica el uso de cláusulas y programas normales¹.

Aunque el objetivo de partida esté compuesto por literales positivos, al tratar con programas normales, es evidente que, en un proceso de resolución SLD, se podrá plantear el problema de resolver un subobjetivo que sea un literal negativo, ya que los programas normales pueden contener en los cuerpos de las cláusulas literales que son átomos negados. A partir de un programa normal tampoco se puede extraer información negativa, debido a que las cabezas de las cláusulas que lo componen son literales positivos. Por lo tanto, también en este caso será necesario un mecanismo que solucione este problema.

¹Esto es, los programas formados por cláusulas normales, véase la Tabla 5.1.

7.2.2. La negación en la programación lógica

Para resolver el problema de la negación en la programación lógica, se han adoptado diferentes soluciones. Aquí presentamos dos de las más populares en el campo de la programación lógica.

En ambas técnicas el problema de la extracción de información negativa se trata ampliando el sistema deductivo, formado por la sola regla de inferencia de resolución SLD, con una nueva regla de inferencia. Esta nueva regla de inferencia puede utilizarse para obtener una caracterización adicional de un programa definido, la llamada *semántica de la negación* que, en conjunción con las otras semánticas estudiadas en el Apartado 5.5, completa el significado de un programa definido.

Suposición de un mundo cerrado

La suposición de un mundo cerrado (CWA – del inglés *Closed World Assumption*) es una caracterización de la negación surgida en el ámbito de las bases de datos deductivas [125]. La CWA supone que, en la especificación de un problema, se dispone de información de todos los hechos relevantes sobre el problema de manera que aquellos hechos sobre los que no se posee información se consideran falsos. Centrándonos en el contexto de la programación lógica desde un punto de vista declarativo, esta hipótesis dice que, para un programa Π y un átomo básico $\mathcal{A} \in \mathcal{B}_L(\Pi)$, podemos afirmar $\neg\mathcal{A}$ si \mathcal{A} no es consecuencia lógica del programa Π . Desde un punto de vista procedural, la CWA puede formularse como una regla de inferencia que permite la extracción de información negativa de un programa:

$$\frac{\mathcal{G} \equiv \leftarrow (Q_1 \wedge \neg\mathcal{A} \wedge Q_2), \quad \mathcal{A} \in \mathcal{B}_L(\Pi), \quad \Pi \not\models_{SLD} \mathcal{A}}{\mathcal{G} \implies_{CWA} \leftarrow Q_1 \wedge Q_2}$$

Ahora la semántica de la negación para un programa Π , empleando la CWA, puede definirse como el conjunto

$$CWA(\Pi) = \{\neg\mathcal{A} \mid (\mathcal{A} \in \mathcal{B}_L(\Pi)) \wedge (\Pi \cup \{\leftarrow \mathcal{A}\} \not\models_{SLD} \square)\}.$$

y la semántica de éxitos básicos de un programa definido Π es $EB(\Pi) \cup CWA(\Pi)$.

Aunque la CWA es una regla de inferencia sencilla y potente puede conducir a errores cuando no se cumple la hipótesis de partida, i.e., que todos los hechos positivos relevantes para el problema están en la base de datos. Por otra parte, aunque $\Pi \cup CWA(\Pi)$ siempre resulta ser un programa consistente cuando Π es un programa definido, no sucede lo mismo cuando Π es un programa normal (o general).

Ejemplo 7.8

Sea el programa normal $\Pi = \{p \leftarrow \neg q\}$. Haciendo uso solamente de la regla de resolución SLD es imposible encontrar una refutación SLD para $\Pi \cup \{\leftarrow p\}$ o para $\Pi \cup \{\leftarrow q\}$.

1. $\leftarrow p \xrightarrow{id} \neg q \Rightarrow_{SLD} \text{fallo}$, ya que $\neg q$ no unifica con la cabeza p de la cláusula.

2. $\leftarrow q \Rightarrow_{SLD} \text{fallo}$, ya que q no unifica con la cabeza p de la cláusula.

No hay otras posibilidades de resolver los objetivos, por lo tanto, aplicando la CWA podemos afirmar $\neg p$ y $\neg q$. Pero la unión de estos hechos negativos al programa para dar el conjunto $\{p \leftarrow \neg q, \neg p, \neg q\}$ es un conjunto insatisfacible.

Negación como fallo

La regla de negación como fallo (NAF – del inglés *Negation As Failure*) puede considerarse que es una restricción de la regla de inferencia CWA. Cuando se aplica la CWA en programación lógica, debemos de ser conscientes de que una refutación SLD para $\Pi \cup \{\leftarrow \mathcal{A}\}$ puede no existir debido a dos motivos:

1. $\Pi \cup \{\leftarrow \mathcal{A}\}$ tiene un árbol SLD que falla finitamente, o bien
2. $\Pi \cup \{\leftarrow \mathcal{A}\}$ tiene un árbol SLD infinito, i.e., que posee alguna rama infinita.

En el último de los casos, vemos que la CWA no es efectiva desde el punto de vista operacional. Además, en el segundo caso no se podría decir nada sobre el éxito o el fracaso de la refutación, ya que no somos capaces de distinguir entre una rama infinita o una rama de éxito (o fallo) que encuentra la cláusula vacía (el fallo) en un tiempo lo suficientemente grande como para no poder ser asumido en una computación ordinaria. Para evitar este problema, Clark [28] introdujo la regla de NAF como medio para extraer información negativa de un programa.

$$\frac{\mathcal{G} \equiv \leftarrow Q_1 \wedge \neg \mathcal{A} \wedge Q_2, \quad \mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi), \quad FF(\Pi, A)}{\mathcal{G} \Rightarrow_{NAF} \leftarrow Q_1 \wedge Q_2}$$

donde la condición $FF(\Pi, A)$ se cumple cuando existe un árbol SLD de fallo finito para $\Pi \cup \{\leftarrow \mathcal{A}\}$ (usando alguna regla de cómputo).

Ahora la semántica de la negación para un programa Π , empleando la regla de NAF, puede definirse como el conjunto

$$\mathcal{NAF}(\Pi) = \{\neg \mathcal{A} \mid (\mathcal{A} \in \mathcal{B}_{\mathcal{L}}(\Pi)) \wedge FF(\Pi, A)\}.$$

y la semántica de un programa definido Π es $\mathcal{EB}(\Pi) \cup \mathcal{NAF}(\Pi)$. Nótese que los problemas respecto a la consistencia de $\Pi \cup \mathcal{NAF}(\Pi)$ son los mismos que con respecto a la consistencia de $\Pi \cup \mathcal{CWA}(\Pi)$.

La estrategia de resolución SLD junto con la regla de NAF conforman la denominada estrategia de *resolución SLDNF*, que es la base de los sistemas actuales de programación lógica (e.g. Prolog). La estrategia de resolución SLDNF se emplea como semántica operacional de la programación lógica cuando ésta pretende manejar información negativa y programas más expresivos que los programas definidos, como son los programas normales. La estrategia de resolución SLDNF procede del siguiente modo:

1. cuando la regla de computación selecciona un literal positivo, entonces se utiliza la estrategia de resolución SLD para derivar un nuevo objetivo;
2. si el literal seleccionado es un átomo básico negativo, se desencadena un proceso recursivo para aplicar la regla de NAF;
3. si el átomo negado seleccionado no es básico, el cómputo debe detenerse ya que la regla SLDNF no soporta computaciones para subobjetivos negados que contengan variables.

Los resultados teóricos sobre los programas normales y la estrategia de resolución SLDNF no son tan confortables como los presentados para los programas definidos y la resolución SLD. Estos resultados pueden estudiarse en [7, 41, 86].

7.2.3. La negación en el Lenguaje Prolog

El lenguaje Prolog proporciona una forma de negación basada en el principio de negación como fallo. Para representar la negación utiliza el operador unario “*not*” que, como las otras conectivas de la lógica, se considera en Prolog un predicado predefinido. Su definición, haciendo uso del corte, consiste en las siguientes cláusulas:

```
not(A) :- A, !, fail.  
not(A) :- true.
```

El operador “*not*” puede considerarse una facilidad de segundo orden ya que, haciendo uso de la ambivalencia de la sintaxis, este operador se aplica sobre átomos (i.e., *A* es una metavariable). Nótese que, con esta definición, “*not (!)*” equivale a la construcción “*!, fail*”. La definición se ha construido de forma que *not (A)* falla si *A* tiene éxito y, por contra, *not (A)* tiene éxito si *A* falla.

Observe que esta implementación de la negación es un puro test que en ningún caso instanciará ninguna variable de un objetivo *not A*. Si el objetivo *not A* falla, el *A* concluirá simplemente con éxito y respuesta vacía (salida *yes* del intérprete). Por el contrario, si *A* tiene éxito (por existir al menos una respuesta computada, el objetivo original *not A* termina con fallo (salida *no* del intérprete).

Es conveniente notar que esta definición no se corresponde exactamente con la regla de NAF anteriormente introducida. Por un lado, no se controla si el átomo *A* es básico, de forma que un cómputo puede proceder aunque en el objetivo solo haya subobjetivos que son átomos negados con variables, lo que puede dar lugar a resultados erróneos y a la pérdida de respuestas. Por otro lado, la regla de computación fija de Prolog puede dar lugar a resultados erróneos aún en el caso de que el átomo negado sea básico. Concluimos entonces que la implementación de la negación en Prolog resulta incorrecta e incompleta.

Ejemplo 7.9

Dado el programa

```
q(a) :- p(b), q(b).
p(b) :- p(b).
```

y el objetivo “?- not q(a).” el intérprete entra en un cómputo infinito, ya que intenta satisfacer el objetivo “?- q(a).” para el que solo existe una derivación infinita, cuando se aplica la regla de computación “seleccionar el literal más a la izquierda” de Prolog. Sin embargo para la regla de computación “seleccionar el literal más a la derecha” existe un árbol de fallo finito para el objetivo “?- q(a).”. Dado que la regla de NAF simplemente exige que exista un árbol de fallo finito, el sistema debería ser capaz de responder “yes” al objetivo inicialmente planteado. Como hemos dicho, esta forma que tiene Prolog de tratar la negación puedeerrar a la hora de generar las respuestas de un objetivo no básico, como puede verse en los siguientes ejemplos que ilustran la pérdida de corrección y completitud del intérprete, es decir, Prolog puede perder respuestas correctas y obtener otras que no son correctas.

Ejemplo 7.10

Considere el programa

```
p(a).
q(b).
```

Para el objetivo “?- p(X), not q(X).” el intérprete encuentra la respuesta $X = a$, que es correcta. Sin embargo, el objetivo “?- not q(X), p(X).”, que es declarativamente identico al anterior, falla finitamente y no se encuentra dicha respuesta.

El ejemplo anterior sugiere que una reordenación de los subobjetivos dentro de un objetivo, y también en el cuerpo de las cláusulas del programa, puede ser muy útil con el fin de dar la oportunidad para que los literales negativos se transformen en básicos, por instanciaión de sus variables, antes de ser seleccionados.

Ejemplo 7.11

Considere el programa

```
igual(1,1).
igual(2,2).
p :- not(igual(X,2)), igual(X,1).
```

Es fácil ver que en este programa el objetivo “?- p.” es cierto, dado que existe un valor de x que satisface el cuerpo “not(igual(X,2)), igual(X,1)” (el valor $x = 1$). Sin embargo, en vez de terminar con éxito computando la respuesta vacía, el objetivo “?- p.”

falla finitamente: para resolver el objetivo “?- p.”, el intérprete lanza a resolver el objetivo derivado “?- not (igual(X,2)), igual(X,1).”. Al seleccionar el subobjetivo “?- not (igual(X,2))” se lanzará su versión positiva “?- igual(X,2)” para, cuando termine la evaluación de ésta, concluir el resultado contrario. Desafortunadamente, la evaluación de “?- igual(X,2)” concluye con éxito, con respuesta “?- X=2”. Por tanto, la evaluación de “?- not (igual(X,2))”, y consecuentemente la de “?- p”, falla.

Supongamos ahora el objetivo “?- not p.” que debería fallar, consistentemente con el hecho de que “p.” es cierto. Para resolver “?- not p.”, Prolog lanza a evaluar su versión positiva que, como acabamos de ver, termina con fallo. Por tanto, Prolog concluye que “not p.” es cierto entregando la salida incorrecta “yes”.

Para terminar, vamos a ver informalmente el motivo por el cuál en general no es adecuado utilizar el predicado `not` para resolver subobjetivos negados que contienen variables, como los de los anteriores ejemplos. Consideremos por ejemplo el objetivo “?- not p(X)” en el programa del Ejemplo 7.10. Haciendo explícita la cuantificación, esta pregunta correspondería al objetivo $(\forall X) (\leftarrow \neg p(X)) \Leftrightarrow \leftarrow (\exists X) \neg p(X)$ que es equivalente también a $\leftarrow \neg(\forall X) p(X)$. En otras palabras, solo si $p(X)$ es cierto para todo X se debe poder concluir que su negación, $\neg p(X)$, es falsa. Sin embargo, observe que Prolog interpreta incorrectamente el objetivo inicial $\leftarrow (\exists X) \neg p(X)$ como $(\leftarrow \neg (\exists X) p(X)) \Leftrightarrow \neg (\neg (\exists X) p(X)) \Leftrightarrow \neg (\leftarrow (\exists X) p(X))$ (por eso lanza el objetivo sin negar $\leftarrow (\exists X) p(X)$ y, caso de que exista al menos una respuesta para $p(X)$, concluye que $\neg p(X)$ es falso). Obviamente $\leftarrow \neg(\forall X) A \Leftrightarrow \leftarrow \neg(\exists X) A$ siendo solo equivalentes cuando el átomo A es básico dado que, en ese caso y solo en ese, $(\forall X) A \Leftrightarrow (\exists X) A \Leftrightarrow A$.

7.3 ENTRADA/SALIDA

Prolog proporciona una serie de predicados de Entrada/Salida (E/S) para incrementar el nivel de interacción entre el usuario y los programas Prolog. Sin ellos, el intercambio de información solo podría realizarse vía un proceso de unificación entre las cláusulas del programa y las preguntas (objetivos) planteados al intérprete. Los predicados predefinidos de E/S junto con el resto que sirven de interfaz con la máquina, haciendo uso de los servicios del sistema operativo, son los más dependientes de la implementación del sistema Prolog. En este apartado estudiaremos una pequeña muestra de tales predicados de E/S que están disponibles en la mayoría de las implementaciones que siguen la sintaxis de Edimburgo.

7.3.1. Flujos de E/S

En Prolog los ficheros suelen denominarse flujos, ya que la visión que suministra al usuario de los mismos es la de una secuencia de objetos (átomos, términos, o simplemente caracteres). Un programa Prolog puede leer datos de diferentes flujos de entrada y escribir datos en diferentes flujos de salida; pero durante la sesión solamente puede estar

activo (abierto) al mismo tiempo un flujo de entrada y uno de salida. Así pues, en cada instante, para Prolog existen solo dos flujos: i) el flujo de entrada estándar (CIS – *Current Input Stream*), y ii) el flujo de salida estándar (COS – *Current Output Stream*). Una de las características fundamentales de un fichero es su nombre. El programa accede a un fichero a través de su nombre. Los nombres de los ficheros pueden elegirse siguiendo las normas sintácticas prevalentes en el sistema operativo sobre el que se ejecuta el intérprete de Prolog. El teclado y la pantalla son tratados como un único flujo, que recibe el nombre de “*user*”. La secuencia de operaciones típicas cuando se trabaja con ficheros es la siguiente:

1. abrir el fichero para leer o escribir, con esta operación se asocia un flujo al fichero;
2. realizar las acciones de lectura o escritura pertinentes; y
3. cerrar el fichero, con esta operación se disocia el flujo del fichero.

Cuando se inicia una sesión con un sistema Prolog, el CIS es el teclado y el COS la pantalla, que no necesitan abrirse. Observe que no se puede tener un mismo fichero abierto para lectura y escritura (a excepción del *user*).

7.3.2. Predicados de E/S

Podemos enumerar las características generales que poseen todos los predicados de E/S:

1. cuando se evalúan siempre tienen éxito;
2. nunca se pueden resatisfacer (cualquier intento de hacerlo hace que el proceso de vuelta atrás continúe hacia la izquierda);
3. tienen como efecto lateral la entrada/salida de un carácter, un término, etc.

A continuación se muestran algunos de los predicados de E/S más usados junto a una breve explicación.

Manipulación de ficheros

- **`see(Fich)`**, asocia el CIS al fichero `Fich`, i.e., abre el fichero `Fich` para lectura.
- **`seeing(Fich)`**, devuelve en `Fich` el nombre del CIS actual.
- **`seen`** cierra el CIS actual, asignando el CIS a `user`, i.e., al teclado.
- **`tell(Fich)`**, asocia el COS al fichero `Fich`, i.e., abre el fichero `Fich` para escritura.
- **`telling(Fich)`**, devuelve en `Fich` el nombre del COS actual.

- `told`, cierra el COS actual, escribiendo en él un EOF y asignando el COS a `user`, i.e., a la pantalla.

Lectura y escritura de términos

- `read(Term)`, lee el siguiente término del CIS e instancia la variable `Term` con el término leído. Si `Term` está ya instanciada tiene éxito si el término leído es igual a `Term` y falla en caso contrario. El predicado `read` es determinista, por lo que si se produce un fallo no hay vuelta atrás para leer un término nuevo. El predicado `read` espera que los términos almacenados en el fichero sean sintácticamente correctos y estén seguidos por un carácter terminador: el punto, el carácter blanco o retorno de carro.
- `write(Term)`, si la variable `Term` está instanciada a algún término lo escribe en el COS.
- `display(Term)`, escribe en el COS, en forma prefija, la expresión infija a la que se ha instanciado la variable `Term`.

Observe que los predicados para lectura y escritura de términos pueden utilizarse, entre otras tareas, para detectar condiciones de error en un programa, como revela el siguiente ejemplo.

Ejemplo 7.12

Supongamos que queremos definir el predicado `fac(N, F)` que devuelve en su segundo argumento `F` el resultado de calcular el factorial de `N`. Una posible solución a este problema sería la siguiente:

```
fac(0, 1).
fac(N, F) :- N1 is N - 1, fac(N1, F1), F is N * F1.
```

Sin embargo, si se utiliza la anterior definición para evaluar el factorial de un número negativo, la regla recursiva entra en un “bucle infinito”, en el que el predicado `fac` continúa llamándose a sí mismo indefinidamente. Eventualmente, se producirá un error al consumirse toda la memoria de la pila del sistema Prolog, que trata de almacenar los registros de activación que se generan en cada una de las llamadas del predicado `fac`. Para evitar este problema, se puede insertar una regla extra que atrape las posibles entradas negativas, obteniéndose la siguiente definición alternativa:

```
fac(0, 1).
fac(N, F) :- N < 0,
            write('Error: primer argumento negativo').
fac(N, F) :- N1 is N - 1, fac(N1, F1), F is N * F1.
```

Ahora, en caso de que se suministre un número negativo al predicado `fac`, el intérprete no entra en un “bucle infinito” sino que se escribe un mensaje de error por pantalla.

Lectura y escritura de caracteres

- `get0(Char)`, lee el siguiente carácter del CIS e instancia la variable `Char` al valor del código ASCII del mismo. Si la variable `Char` ya está instanciada, tiene éxito cuando el carácter leído es igual a `Char` y falla en cualquier otro caso.
- `get(Char)`, se comporta igual que `get0` salvo en el hecho de que primero salta todos los caracteres no imprimibles (en particular, también los blancos) hasta encontrar el carácter imprimible que leerá. Por lo tanto `get` solo lee los caracteres imprimibles.
- `skip(X)`, tiene como efecto lateral leer todos los caracteres del CIS hasta encontrar uno que unifique con `X`.
- `put(Char)`, si `Char` está instanciada a algún entero entre 0 y 255 escribe en el COS el carácter cuyo ASCII es `Char`. Se produce un error si `Char` no es un entero o está desinstanciada.
- El predicado `tab(Num)` escribe `Num` espacios en blanco en el COS y el predicado `n1` escribe una carácter nueva línea en el COS.

7.4 OTROS PREDICADOS PREDEFINIDOS

Continuamos con el resumen de las características del lenguaje Prolog, presentando una colección de predicados predefinidos, cuya importante utilidad será demostrada en breve.

7.4.1. Predicados predefinidos para la catalogación y construcción de términos

Catalogación de términos

- `var(X)`, tiene éxito si `X` es una variable no instanciada.
- `nonvar(X)`, tiene éxito si `X` es una variable instanciada.
- `atom(X)`, tiene éxito si la variable `X` se ha instanciado a una constante (i.e., un átomo, en la jerga de Prolog) previamente al momento de su ejecución.
- `integer(X)`, tiene éxito si la variable `X` se ha instanciado a un entero.
- `atomic(X)`, tiene éxito si la variable `X` se ha instanciado a un objeto simple (atómico), i.e., un símbolo constante, un entero, o una cadena.
- `compound(X)`, tiene éxito si la variable `X` se ha instanciado a un término general, esto es, un término de aridad distinta de cero.

Análisis y construcción de términos

- `functor(T, F, N)`, tiene éxito si es posible unificar `T` a una estructura con functor principal `F` y aridad `N`. Falla si `T` y (`F` o `N`) están desinstanciadas. Si `T` está enlazada a una constante o un número, `F` unificará con `T` y `N` con 0 (término de aridad cero).
- `arg(N, T, A)`, unifica `A` con el `N`-ésimo argumento de `T`. Las variables `N` y `T` deben estar instanciadas.
- `E = .. L`, tiene éxito si se puede unificar `L` con una lista cuya cabeza es el functor principal de la expresión `E` y cuya cola son los argumentos de `E`. Al menos `E` o `L` deben estar instanciadas.
- `name(A, L)`, tiene éxito si los caracteres que forman el átomo (constante) `A` tienen como código ASCII los números que componen la lista `L`.

El siguiente ejemplo ilustra mínimamente las facilidades del lenguaje Prolog para la manipulación simbólica.

Ejemplo 7.13

Definimos un predicado `aplicaSub(S, T1, T2)`, que aplica una sustitución `S` a un término `T1` para dar un nuevo término `T2`. Este predicado hace uso de un predicado auxiliar que denominamos “`aplicaEnlace`”. Por su parte, dado un término `T1`, el predicado `aplicaEnlace(V:=T, T1, T2)` sustituye las posibles apariciones de una variable `V` en `T1` por un término `T`, obteniéndose un nuevo término `T2`.

Las sustituciones se representan como una lista de enlaces del tipo “`Variable := Término`”, conforme a la notación introducida en el Ejemplo 6.12. Además, suponemos que trabajamos con sustituciones idempotentes y que, por lo tanto, los enlaces pueden aplicarse uno después de otro sobre el término en el que se sustituyen las variables.

La solución propuesta es la siguiente:

```

:-op(300, xfx, ':=').

aplicaSub([], T, T).
aplicaSub([E|L], T, T1) :- applicaEnlace(E, T, T2),
    applicaSub(L, T2, T1).

aplicaEnlace(X:=T1, T, T1) :- var(T), T==X, !.
aplicaEnlace(X:=T1, T, T) :- var(T), T'\==X, !.
aplicaEnlace(X:=T1, T, T) :- atom(T), !.
aplicaEnlace(X:=T1, T, T2) :- compound(T),
    T=..[F|Args],
    applicaEnlaceArgs(X:=T1, Args, Args2),
    T2=..[F|Args2].
```

%% applicaEnlaceArgs(X:=T1,L1,L2), aplica el enlace X:=T1 a la

```

%% lista de argumentos L1, para obtener la lista de argumentos L2.
aplicaEnlaceArgs(X:=T1, [], []).
aplicaEnlaceArgs(X:=T1, [T|Args], [T2|Args2]) :-
    aplicaEnlace(X:=T1, T, T2),
    aplicaEnlaceArgs(X:=T1, Args, Args2).

```

Nótese que la definición del predicado `aplicaSub` se ha efectuado por recursión sobre la estructura de los términos, tal y como fueron definidos en el Apartado 2.2.1. Se recomienda al lector que revise dicha definición de término. Compruebe que las dos primeras cláusulas de la definición de la relación `aplicaEnlace` tratan el caso en el que el término `T`, sobre el que se aplica la sustitución, es una variable (i.e., `var(T)` es verdadero). La tercera cláusula trata el caso en el que el término `T` es una constante (i.e., `atom(T)` es verdadero) y la cuarta el caso en el que `T` es un término general (i.e., `compound(T)` es verdadero).

Volvemos a constatar que la definición por recursión estructural es una técnica muy útil y elegante a la hora de definir propiedades sobre estructuras sintácticas inherente-mente recursivas.

7.4.2. Predicados predefinidos para la manipulación de cláusulas y la metaprogramación

Gestión del espacio de trabajo

Como se ha mencionado con anterioridad, para que un programa Prolog pueda ejecutarse es preciso que los hechos y las reglas que lo constituyen se hayan cargado en el espacio de trabajo del sistema Prolog. Las reglas que están almacenadas en un fichero y todavía no se han cargado en el espacio de trabajo no pueden intervenir en la ejecución del programa. Un sistema Prolog solamente reconoce aquello que está en su espacio de trabajo. Por consiguiente, los efectos y acciones que se realicen durante la ejecución de un programa y un objetivo dependen de lo que allí se encuentre almacenado.

En lo que sigue, describimos brevemente una serie de predicados que permiten gestionar la base de datos interna del sistema Prolog y, por lo tanto, modificar el comportamiento de un programa.

- **Carga de programas.**
 - `consult(Fich)`, añade a la base de datos interna de Prolog todas las cláusulas que contenga el fichero `Fich`. Si ya existen cláusulas con el mismo símbolo de predicado, pueden crearse definiciones duplicadas.
 - `reconsult(Fich)`, lo mismo que `consult` pero reemplaza, por cláusulas de `Fich`, aquellas cláusulas de la base de datos interna de Prolog que tienen el mismo símbolo de predicado que otras que aparecen en `Fich`.

■ Consulta.

- `listing(Nom)`, escribe en el COS todas las cláusulas de la base de datos interna de Prolog cuyo símbolo de predicado en cabeza coincida con `Nom`. La variable `Nom` debe estar instanciada a un nombre de predicado; si no, se produce un error. Si `Nom` está enlazado a un nombre y Prolog no encuentra cláusulas con ese nombre contesta `no`.

■ Adición y supresión.

- `assert(C)`, añade la cláusula `C` al principio de la base de datos interna de Prolog.
- `assertz(C)`, añade la cláusula `C` al final de la base de datos interna de Prolog.
- `retract(C)`, elimina de la base de datos interna de Prolog la primera cláusula que unifica con `C` y falla si no hay cláusulas con las que unifique.

Observe que muchas implementaciones del lenguaje Prolog solamente permiten añadir o borrar hechos de la base de datos interna. Además, para poder hacerlo, se requiere que los predicados definidos por esos hechos se hayan declarado dinámicos, mediante la directiva “`dynamic`” (consulte el manual del sistema Prolog que esté empleando para obtener más detalles de su utilización).

La familia de predicados `assert` y `retract` aportan al lenguaje Prolog características extralógicas, que son muy discutidas por algunos autores que desean mantener la pureza y declaratividad del lenguaje [139]. Entre ellas podemos citar la capacidad de ciertos programas de automodificarse en tiempo de ejecución que, aunque indispensable para implementar algunas aplicaciones (e.g., en el campo de la lógica no monótona), puede considerarse una mala práctica de programación. Los programas que utilizan los predicados `assert` y `retract`, debido a su capacidad para automodificarse, se hacen muy difíciles de entender y depurar, por lo que deberían evitarse.

Sin embargo, un uso controlado de estos predicados puede ser útil en algunas circunstancias. Piense que el lenguaje Prolog, contrariamente a lo que sucede con los lenguajes de corte convencional, no permite el empleo de variables globales. Las variables son locales a la propia regla (esto es, el radio de acción de las variables es la propia regla). Podemos salvar este inconveniente simulando una variable global mediante la definición de un hecho que se utiliza para almacenar un valor y la utilización de los predicados `assert` y `retract` para mantener actualizado dicho valor véase el Ejercicio 7.22). Por otra parte, algunas soluciones, que definen predicados mediante técnicas recursivas ingenuas, son muy ineficientes porque tienden a recalcular un gran número de resultados intermedios. Para solucionar este problema, es posible emplear el predicado predefinido `asserta` para almacenar esos resultados intermedios en la base de datos interna del sistema Prolog, de modo

que solamente se calculen una vez. Los predicados que almacenan los resultados intermedios de una computación, para su posterior utilización en otras computaciones, se denominan *funciones-memo* en [139, Pag.221] (véase el Apartado 7.6 para obtener información más detallada sobre este punto).

Metaprogramación y Orden Superior

Se conoce como metaprogramación la escritura de programas que escriben o manipulan, como datos, otros programas (o a sí mismos). Dado que Prolog utiliza estructuras de datos similares para representar tanto programas como datos, este lenguaje resulta muy adecuado para escribir metaprogramas.

La lista de predicados predefinidos de Prolog que suelen considerarse facilidades de metaprogramación que ofrece el lenguaje incluye: `arg`, `functor`, `=...`, `assert`, `retract`, `clause` y `call`. Es interesante señalar que algunas de estas facilidades suponen la extensión del lenguaje clausal con características de *orden superior*, al permitir definir predicados donde alguno de los argumentos no es un simple dato, sino que puede ser una función u otro predicado.

- `call(Obj)`, invoca el objetivo `Obj`. Tiene éxito si `Obj` tiene éxito. Se utiliza para lanzar objetivos dentro de una regla o de un programa que solo se conocen en tiempo de ejecución. Si `Obj` es una variable deberá estar instanciada en el momento en el que vaya a ejecutarse.
- `clause(Cabeza,Cuerpo)`, unifica la variable `Cabeza` con la cabeza de las cláusulas de la base de datos interna de Prolog y la variable `Cuerpo` con el cuerpo. Si hay varias cláusulas con las que unifique `Cabeza`, se empareja con la primera y por resatisfacción con las siguientes; falla si no hay cláusulas con las que unifique. La variable `Cabeza` debe estar lo suficientemente instanciada (al menos un símbolo de predicado).

En la mayoría de los sistemas Prolog, cualquier cosa que pueda emplearse como argumento del predicado `call` puede emplearse directamente como objetivo, lo que hace innecesario este predicado (por este motivo se omitió en la definición Prolog del predicado `not` —véase el Apartado 7.2.3—). En esos sistemas, el predicado `call` se ofrece solo por razones de compatibilidad con otras versiones de Prolog.

Por otra parte, en algunos sistemas Prolog se produce un error de sintaxis cuando aparece una variable `x` como objetivo. El ejemplo más simple de este fenómeno es la cláusula `eval(x) :- x..` En estos casos, será necesario emplear `call(x)` si queremos eliminar ese error.

El siguiente fragmento de código utiliza los predicados `=...` y `call` para diseñar un predicado que puede aplicar (`map`) una función dada a cada uno de los elementos de una lista. El nombre de dicha función es un parámetro de entrada del predicado.

```
map(NombreFuncion, [H|T], [NH|NT]) :-  
  Funcion=.. [NombreFuncion, H, NH],  
  call(Funcion),  
  map(NombreFuncion, T, NT).  
map(_, [], []).
```

Por ejemplo, considerando una función de cambio de signo

```
neg(A,B) :- B is -A.
```

la siguiente llamada produce la negación de todos los elementos de una lista:

```
?- map(neg, [1,2,3], L).  
L=[-1,-2,-3]
```

Sería posible usar el mismo procedimiento para obtener otros procesadores de listas, si se añade el código de otras funciones, por ejemplo

```
inc(A,B) :- B is A+1.  
dec(A,B) :- B is A-1.
```

con el objetivo

```
?- map(inc, [1,2,3], X), map(dec, X, Y).  
X=[2,3,4], Y=[1,2,3]
```

Otro uso popular de las técnicas de metaprogramación se relaciona con la escritura de metaintérpretes. Debido a que Prolog permite acceder directamente al código del programa en tiempo de ejecución, es fácil escribir un intérprete de Prolog en Prolog mismo. Tal intérprete se llama metaintérprete. Los metaintérpretes se suelen usar para añadir funcionalidad extra a Prolog, por ejemplo para cambiar la negación predefinida o modificar el orden estándar de ejecución de los subobjetivos.

El metaintérprete más simple para Prolog se muestra en el siguiente programa.

```
solve(Goal) :- call(Goal).
```

Sin embargo, no hay ninguna ventaja en usar este metaintérprete ya que invoca inmediatamente al intérprete de Prolog. Es mucho más popular el metaintérprete “vanilla”, que utiliza la unificación predefinida de Prolog pero habilita un acceso al motor de búsqueda que permite modificarlo fácilmente. El metaintérprete “vanilla” es muy importante porque muchos sistemas expertos derivan de él.

El programa que sigue permite ejecutar solo Prolog puro pero puede extenderse para soportar la negación, la manipulación dinámica de la base de datos de cláusulas, etc.

```
solve(true) :- !.  
solve((A,B)) :-  
  solve(A), solve(B).  
solve(A) :-  
  clause(A,B), solve(B).
```

El metaintérprete anterior utiliza el predicado predefinido `clause(H, B)`, que encuentra una cláusula en el programa Prolog cuya cabeza unifica con `H` y cuyo cuerpo es `B` (si no hay tal cuerpo, entonces `B=true`).

La siguiente modificación del metaintérprete “vanilla” puede usarse para computar “pruebas” de las computaciones:

```
solve(true, fact).
solve((A, B) , (ProofA, ProofB)) :- 
    solve(A, ProofA), solve(B, ProofB) .
solve(A, A-ProofB) :- 
    clause(A, B) , solve(B, ProofB) .
```

También es posible escribir un metaintérprete que manipule listas de objetivos en lugar de las tradicionales conjunciones. En algunos casos, esto puede ser más natural ya que evita recorrer la estructura del objetivo completo cada vez que se encuentra un objetivo primitivo.

```
solve([]) .
solve([A|T]) :- 
    clause(A,B),
    add_to_list(B,T,NT),
    solve(NT) .
```

7.4.3. El predicado `setof`

Como hemos visto, el mecanismo de evaluación de Prolog produce respuestas una a una, siguiendo una estrategia en profundidad con vuelta atrás que recorre las ramas del árbol de búsqueda en preorden, i.e., primero las ramas más a la izquierda. Cada vez que se encuentra una respuesta, el intérprete la muestra y se detienen en espera de la indicación del usuario. Si el usuario desea más respuestas, el intérprete de Prolog intenta encontrarlas ayudado de la vuelta atrás. En el contexto de las bases de datos deductivas, este tipo de estrategias recibe el nombre de estrategias de “*una respuesta cada vez*” (*tuple-at-a-time*). No obstante, en muchas ocasiones puede ser de interés obtener todas las respuestas a un objetivo. Prolog proporciona para ello el predicado predefinido `setof`, que permite acumular en una lista todas las respuestas del objetivo planteado. Esto deja la puerta abierta a la posibilidad de elaborar estrategias de “*un conjunto de respuestas cada vez*” (*set-at-a-time*), como la estrategia de búsqueda en anchura (véase el Apartado 9.3.2).

La sintaxis del predicado `setof` es la siguiente: `setof(Term, Obj, Lista)`. Un subobjetivo como el anterior tiene éxito si `Lista` es la lista de instancias ordenadas y sin repeticiones del término `Term` para las que el objetivo `Obj` tiene éxito. El siguiente ejemplo puede aclarar el significado de `setof`.

Ejemplo 7.14

Volviendo al programa de las relaciones familiares del Apartado 6.4, podemos imaginar la siguiente sesión con un intérprete de Prolog:

- Buscar todos los hijos de `haran`.

```
?- setof(Hijo, padre(haran, Hijo), ListaHijos).  
Hijo = _0, ListaHijos = [jesca, lot, melca] ;  
no
```

Nótese que la pregunta no puede plantearse en estos términos.

```
?- setof(_, padre(haran, _), ListaHijos).  
ListaHijos = [_4] ;  
ListaHijos = [_4] ;  
ListaHijos = [_4] ;  
no
```

- Buscar todos los ascendientes de `isaac`.

```
?- setof(Ascend, ascendiente(Ascend, isaac), ListaAscend).  
Ascend = _0, ListaAscend = [abraham, sara, teraj] ;  
no
```

- Buscar todos los ascendientes de `lot`.

```
?- setof(Ascend, ascendiente(Ascend, lot), ListaAscend).  
Ascend = _0, ListaAscend = [haran, teraj] ;  
no
```

- Buscar todos los ascendientes de `isaac` que no son de `lot`.

```
?- setof(Ascend,  
        (ascendiente(Ascend, isaac), not ascendiente(Ascend, lot)),  
        ListaAscend).  
Ascend = _0, ListaAscend = [abraham, sara] ;  
no
```

- Buscar todos los ascendientes de `isaac` o de `lot`.

```
?- setof(Ascend,  
        (ascendiente(Ascend, isaac); ascendiente(Ascend, lot)),  
        ListaAscend).  
Ascend = _0, ListaAscend = [abraham, haran, sara, teraj] ;  
no
```

- Buscar todos los hijos de cada padre.

```
?- setof(Hijo, padre(Padre, Hijo), ListaHijos).
Hijo=_0, Padre=abraham, ListaHijos=[isaac,ismael] ;
Hijo=_0, Padre=batuel, ListaHijos=[laban,rebeca] ;
Hijo=_0, Padre=haran, ListaHijos=[jesca,lot,melca] ;
Hijo=_0, Padre=isaac, ListaHijos=[esau,jacob] ;
Hijo=_0, Padre=najor, ListaHijos=[batuel] ;
Hijo=_0, Padre=teraj, ListaHijos=[abraham,haran,najor,sara] ;
no
```

Si queremos disponer todos los hijos en una lista, independientemente del padre que los haya procreado, esta sería la forma.

```
setof(Hijo, Padre ^ padre(Padre, Hijo), ListaHijos).
Hijo = _0, Padre = _1,
ListaHijos = [abraham,batuel,esau,haran,isaac,ismael,jacob,
jesca,laban,lot,melca,najor,rebeca,sara] ;
no
```

Aquí “`^`” es un operador predefinido que sirve para formar una expresión *cuantificada existencialmente*. El objetivo anterior admite esta lectura: “ buscar todos los valores de la variable `Hijo` tales que `padre(Padre, Hijo)` es verdadero para algún `Padre`”. Así que, “`Padre ^`” puede interpretarse como “existe un `Padre`”.

El lector que esté familiarizado con las bases de datos relacionales, puede haber notado una similitud entre la instrucción `SELECT` del lenguaje SQL y el predicado `set_of`. Una consulta típica en SQL tiene la forma:

```
SELECT A1,A2,...,An
FROM r1,r2,...,rm
WHERE P
```

donde cada A_i representa un atributo (un campo), cada r_i una relación (tabla) y P es un predicado. Por ejemplo, la consulta

```
SELECT nombre-cliente
FROM depósito
WHERE nombre-sucursal = "Principal"
```

busca todos los (nombres de los) clientes que tienen un depósito en la sucursal Principal de un banco.

Los tres argumentos del predicado `set_of` se corresponden con las partes de una consulta a una base de datos:

1. Una lista de campos de la tabla, cuyos valores se están buscando.

2. Alguna condición (o combinación de condiciones) que deben cumplirse en relación con esos valores.
3. Un lugar en el que disponer el resultado.

Si representamos en Prolog la tabla `deposito` como una estructura

```
deposito(Nom-cliente, Nom-sucursal, Saldo),
```

Entonces, el objetivo

```
set_of(Nom-cliente, deposito(Nom-cliente, 'Principal', _), L),
```

sería equivalente a la anterior consulta a la base de datos: la lista `L` almacenaría los (nombres de los) clientes que tienen un depósito en la sucursal Principal de un banco.

Esta similitud no debería sorprendernos, ya que tanto los fundamentos del lenguaje Prolog, como los de la teoría de las bases de datos relationales se basan en la lógica simbólica.

Otros predicados que tienen un comportamiento similar al de `setof` son: `bagof` y `findall`. Se sugiere al lector acudir al manual de referencia de su sistema Prolog para informarse sobre su sintaxis y modo de empleo.

7.5 GENERAR Y COMPROBAR

En este apartado se describe una técnica para la búsqueda de soluciones que algunos autores encuadran dentro del estilo de programación no determinista [139] y otros dentro de las técnicas de búsqueda en un espacio de estados [126] (*véase* más adelante). Nosotros la presentamos en este punto por su sencillez y utilidad para ilustrar, al resolver problemas concretos, el uso de algunas de las facilidades introducidas hasta el momento (aritmética, el corte, negación y operaciones de E/S).

La técnica de *generar y comprobar* consiste en generar, tal vez aleatoriamente, una configuración (o estado) y comprobar que cumple las restricciones que la acreditan como una solución (o coincide con un estado objetivo, que se reconoce como solución del problema). En este método las soluciones completas deben generarse antes de verificar que cumplen las restricciones del problema.

El método de generación y prueba puede actuar de forma que genere las soluciones aleatoriamente, pero esto no garantiza que se pueda encontrar alguna vez la solución. En el siguiente apartado se utiliza el mecanismo de búsqueda de un sistema Prolog para generar sistemáticamente el espacio de estados de un problema, que después se comproban hasta detectar una solución.

7.5.1. El problema del mini-sudoku

En este apartado utilizamos, en su estado más puro, la técnica de generar y comprobar para resolver un mini-sudoku, que es una simplificación de un conocido pasatiempo que aparece en muchas revistas y periódicos. Resolver un mini-sudoku consiste en llenar con números del 1 al 4 las celdas vacías de una matriz 4×4 parcialmente llena. La matriz está dividida en submatrices 2×2 y se exige que ninguna cifra esté repetida en una fila, una columna o una submatriz.

Dado que el lenguaje Prolog no contempla el uso de matrices, las representaremos mediante el uso de listas. Cuando definamos algunos de los predicados que emplearemos en la solución del problema, en lugar de pensar en términos de listas de 16 elementos (esto es, matrices de (4×4) casillas), pensaremos en listas con un número indefinido de elementos. De este modo podremos aplicar las técnicas de definición por inducción estructural sobre listas.

Para resolver este problema, primero definimos un predicado `generar(M)`, que genere sistemáticamente, a partir de una matriz 4×4 parcialmente llena, todas las matrices 4×4 con valores de sus celdas comprendidos entre 1 y 4. Esta definición se fundamenta en el predicado `member(X, [1,2,3,4])`, que por efecto del mecanismo de vuelta atrás, enlaza la variable `X` que aparece en una casilla, sucesivamente, con los valores 1, 2, 3 y 4. El predicado `generar(M)` no se ocupa de que la matriz generada cumpla las restricciones de un mini-sudoku.

```

%% generar(M) : Genera una matriz M.
%% generar(M) es verdadero cuando M es una "matriz" con todos sus
%% elementos "rellenos". Esto es, no contiene variables.

%% Caso Base: Para la lista vacía no es preciso generar elementos
generar([]).

%% Caso Inductivo: Para una lista genérica.
%% Si X es una variable (es decir, la "casilla" no se ha
%% rellenado) seleccionar un valor para X entre [1,2,3,4] y
%% seguir generando.
generar([X|M]) :- var(X), !,
                  member(X, [1,2,3,4]),
                  generar(M).

%% Si no (es decir, la "casilla" ya estaba llena), seguir
%% generando el resto de la matriz.
generar([X|M]) :- generar(M).

```

Después definimos un predicado `comprobar(M)` que compruebe que una matriz 4×4 (completamente llena, con valores de sus celdas comprendidos entre 1 y 4), cumple las restricciones de un mini-sudoku.

```

%% comprobar(M) :
%% Es verdadero cuando M es una matriz que cumple las restriccio-

```

```
%% nes de un Mini-Sudoku.

comprobar([X11, X12, X13, X14,
          X21, X22, X23, X24,
          X31, X32, X33, X34,
          X41, X42, X43, X44]) :- fila(X11, X12, X13, X14),
                                         fila(X21, X22, X23, X24),
                                         fila(X31, X32, X33, X34),
                                         fila(X41, X42, X43, X44),
                                         columna(X11, X21, X31, X41),
                                         columna(X12, X22, X32, X42),
                                         columna(X13, X23, X33, X43),
                                         columna(X14, X24, X34, X44),
                                         cuadrado(X11, X12,
                                         X21, X22),
                                         cuadrado(X13, X14,
                                         X23, X24),
                                         cuadrado(X31, X32,
                                         X41, X42),
                                         cuadrado(X33, X34,
                                         X43, X44).

fila(X1, X2, X3, X4) :- diferentes(X1, X2, X3, X4).
columna(X1, X2, X3, X4) :- diferentes(X1, X2, X3, X4).
cuadrado(X1, X2, X3, X4) :- diferentes(X1, X2, X3, X4).

diferentes(X1, X2, X3, X4) :- X1 =\= X2, X1 =\= X3, X1 =\= X4,
                               X2 =\= X3, X2 =\= X4,
                               X3 =\= X4.
```

Para completar la solución de este problema, solamente nos queda por definir un predicado auxiliar `imprimir(M)` que imprima una matriz 4×4 , de forma que escriba cada fila de cuatro elementos en una línea diferente.

```
%% imprimir(M): imprime la matriz M.

imprimir([]).
imprimir([X|M]) :- imprimir([X|M], 4, 4).

%% imprimir(M, L, C): predicado auxiliar para imprimir la matriz
%% M. El segundo argumento L indica el numero de elemento en
%% una fila. El tercer argumento es un contador que lleva la
%% cuenta del numero de elementos impresos de una fila durante
%% las respectivas iteraciones.

imprimir([], _, _) :- nl.
imprimir([X|M], L, 0) :- nl, imprimir([X|M], L, L).
imprimir([X|M], L, C) :- NC is C-1, write(X), tab(1),
                           imprimir(M, L, NC).
```

Ahora podemos emplear los predicados anteriores para definir un procedimiento `solucionSudoku(M)` que, dada una matriz 4×4 (parcialmente llena) imprima la solución al mini-sudoku:

```
solucionSudoku(M) :- M = [X11, X12, X13, X14,
                           X21, X22, X23, X24,
                           X31, X32, X33, X34,
                           X41, X42, X43, X44],
                           generar(M), comprobar(M), imprimir(M).
```

La solución presentada en este apartado no sería adecuada para resolver un verdadero sudoku con 9×9 casillas, debido al enorme número de alternativas que tendría que explorar antes de encontrar una solución. Una forma de evitar este inconveniente es combinar la propia fase de generación de alternativas con la de comprobación, de forma que el predicado `generar` solamente pueda generar soluciones al problema o fallar. En el próximo apartado se resuelve un problema clásico, que aparece en muchos textos de informática dedicados al estudio de los lenguajes de programación, haciendo uso de esta optimización del método de generación y prueba.

7.5.2. El problema de las ocho reinas

Presentamos el conocido problema de las ocho reinas como ejemplo de problema que puede resolverse con gran elegancia desde una visión declarativa, usando la técnica de generar y comprobar. Este problema consiste en disponer 8 reinas en un tablero de ajedrez de forma que ninguna de ellas pueda atacar a otra.

La solución del problema se representa mediante un vector de 8 posiciones. Cada posición es una coordenada `c(X_i, Y_i)` donde se encuentra una reina que no es atacada por ninguna otra. Dicho vector será una lista:

```
[c(1, Y1), c(2, Y2), c(3, Y3), c(4, Y4),
 c(5, Y5), c(6, Y6), c(7, Y7), c(8, Y8)]
```

Como puede apreciarse, basta fijar las posiciones `X_i` y centrarse en la búsqueda de las posiciones `Y_i`.

```
% vector(Solucion), verdadero si al situar las reinas en las
% posiciones del vector Solucion no se atacan las unas a las otras
vector(Solucion) :-
    Solucion = [c(1, _), c(2, _), c(3, _), c(4, _),
                c(5, _), c(6, _), c(7, _), c(8, _)],
    generar(Solucion).
```

Para encontrar la solución, definimos un predicado `generar(Solucion)` que sea verdadero cuando `Solucion` represente una combinación de posiciones en la cual ninguna reina ataque a otra. Al definir el predicado `generar` es más fácil pensar en términos de

un tablero de $(n \times n)$ casillas, siendo n un número indeterminado, que pensar en un tablero de (8×8) casillas. Esto nos permite considerar que la solución puede expresarse mediante una lista, con un número indefinido de coordenadas, en vez de mediante un vector. Este cambio de punto de vista posibilita aplicar las técnicas de definición por inducción estructural sobre listas.

Contrariamente a la forma en que se diseñó la solución al problema del mini-sudoku, en lugar de generar toda una configuración del tablero y después comprobar que dicha configuración cumple las restricciones del problema de las ocho reinas (es decir, las reinas no se atacan), primero se comprueba que una reina no ataca a las otras antes de posicionarla sobre el tablero. Nuevamente, el predicado `member` se ocupa de generar sistemáticamente, con ayuda del mecanismo de vuelta atrás, todas las diferentes posiciones (filas) que puede ocupar una reina dentro de una determinada columna. El predicado `ataca` se encargará de aceptar o rechazar una de esas alternativas propuestas por `member`.

```
% generar(L), L es una lista con posiciones desde las que
% las reinas no pueden atacarse
generar([]).
generar([c(X, Y)|Resto]) :-
    generar(Restos),
    member(Y, [1, 2, 3, 4, 5, 6, 7, 8]),
    not ataca(c(X, Y), Resto).
```

Esencialmente, el predicado `generar(L)` actúa de la siguiente manera: la lista vacía es una solución ya que, trivialmente, no hay reinas en posiciones desde las que puedan atacarse unas a otras; una lista `[c(X, Y)|Resto]` es una solución al problema (con un número indeterminado de n reinas) si la reina situada en la posición `c(X, Y)` no puede atacar a las reinas situadas en el resto de posiciones y `Resto` es una solución al problema de las $n - 1$ reinas.

El predicado `ataca(c(X, Y), L)` es verdadero cuando la reina situada en la posición `c(X, Y)` puede atacar a una reina situada en alguna posición `c(X1, Y1)` de `L`. La reina situada en la posición `c(X, Y)` puede atacar a la reina situada en la posición `c(X1, Y1)` si: i) ambas están en la misma horizontal, i.e., $y = y_1$; ii) ambas están en la misma diagonal de pendiente positiva, i.e. si cumplen la ecuación de recta $y_1 = x_1 + (y - x)$; y iii) ambas están en la misma diagonal de pendiente negativa, i.e. si cumplen la ecuación de recta $y_1 = (y + x) - x_1$. Esta especificación tiene una traducción directa a reglas de Prolog.

```
% ataca(c(X, Y), L), la reina en c(X, Y) ataca a alguna de las
% situadas en las posiciones de L
ataca(c(_, Y), [c(_, Y1)|_]) :-
    Y == Y1, !.% misma horizontal
ataca(c(X, Y), [c(X1, Y1)|_]) :-
    Y1 is X1 + (Y - X), !.% misma diagonal (pendiente = 1)
ataca(c(X, Y), [c(X1, Y1)|_]) :-
    Y1 is (Y + X) - X1, !.% misma diagonal (pendiente = -1)
```

```
ataca(c(X, Y), [c(_, _) | Resto]) :-  
    ataca(c(X, Y), Resto). % en otro caso
```

Observe que cada uno de los casos en los que se ha dividido la definición del predicado `ataca` son excluyentes, por lo que cobra sentido la utilización del corte para ganar eficiencia.

Para facilitar las consultas introducimos los siguientes predicados:

```
% muestra el tablero y la solucion  
tablero(Sol) :- vector(Sol), esc_cabecera, esc_tablero(Sol).  
  
esc_cabecera :- nl, nl, nl,  
    tab(3), write('-----'), nl,  
    tab(3), write('! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 !'), nl,  
    tab(3), write('-----'), nl.  
  
esc_tablero([]).  
esc_tablero([c(N, R) | Resto]) :- esc_fila(N, R), esc_tablero(Resto).  
  
esc_fila(N, R) :- tab(2), write(N), esc_casillas(R), nl,  
    tab(3), write('-----'), nl.  
  
esc_casillas(1) :- write('! * ! ! ! ! ! ! !').  
esc_casillas(2) :- write('! ! * ! ! ! ! ! !').  
esc_casillas(3) :- write('! ! ! * ! ! ! ! ! !').  
esc_casillas(4) :- write('! ! ! ! * ! ! ! ! ! !').  
esc_casillas(5) :- write('! ! ! ! ! * ! ! ! ! !').  
esc_casillas(6) :- write('! ! ! ! ! ! * ! ! ! ! !').  
esc_casillas(7) :- write('! ! ! ! ! ! ! * ! ! !').  
esc_casillas(8) :- write('! ! ! ! ! ! ! ! * ! !').
```

Este problema pone de manifiesto que muchas veces la clave de la solución es generalizar el problema. Paradójicamente, al considerar el problema de una forma más general su solución se hace más sencilla de formular, e.g., considerar un tablero de $(n \times n)$ casillas en vez de un tablero de (8×8) casillas. Esto puede tomarse como una regla práctica de programación en Prolog.

Se aconseja al lector comparar la solución propuesta para este problema con las que aparecen en libros donde se enseña la programación con lenguajes imperativos, por ejemplo, el libro de N. Wirth [145].

Finalmente, repare que esta solución (como la de otros muchos problemas) puede mejorarse mediante un análisis más profundo de la estructura del propio problema. En este sentido, el Ejercicio 7.26 propone un método para optimizar nuestra solución del problema de las ocho reinas. La mejora propuesta abunda en la idea de seguir incorporando elementos de control a la fase de generación de soluciones.

7.6 PROGRAMACIÓN EFICIENTE EN PROLOG

Como es sabido, cada estilo de programación tiene técnicas específicas para mejorar la eficiencia del código que se escribe en los lenguajes de esa familia, más allá de principios comunes de aplicación general como son usar algoritmos y estructuras de datos adecuadas, dar preferencia a esquemas iterativos frente a recursivos, etc. Por otro lado, las técnicas específicas de un estilo o lenguaje no son transferibles a otros y, en ocasiones, dependen incluso del compilador.

En este apartado vamos a ver algunos criterios simples y generales que pueden ahorrar tiempo de ejecución y memoria empleada en los programas Prolog. Pero antes de abordar cuestiones específicas sobre la eficiencia, empecemos resumiendo algunos principios de programación generales en este tipo de lenguajes:

- Se debe tener presente siempre que el orden de las cláusulas y de los subobjetivos en los cuerpos de éstas es importante. Conviene escribir las cláusulas para resolver el caso base antes de las que representan el caso recursivo. También es conveniente escribir los subobjetivos más restrictivos delante de los otros y retrasar los cálculos hasta que sus resultados resulten necesarios, por si antes de hacerlos se detecta que no contribuyen al resultado buscado.
- Resulta más intuitivo ubicar, en cada átomo de la cabeza y el cuerpo de las cláusulas, los argumentos de entrada antes de los argumentos que acumulan cálculos intermedios y éstos antes que los argumentos de salida.
- Antes de escribir código es conveniente buscar en la biblioteca del sistema si existe alguna rutina que resuelva el problema. El código de estas rutinas y de los predicados predefinidos siempre está más optimizado que el que puede programarse a mano. Es el caso, por ejemplo, de los predicados de ordenación de listas.
- Hay que utilizar el corte con cautela y solo en los lugares adecuados: por ejemplo el lugar exacto en que se conoce que la elección escogida es la correcta, ni después ni antes.
- Conviene buscar soluciones deterministas para problemas que son deterministas.
- El efecto del corte debe tenerse lo más localizado posible. Si un predicado ha de ser determinista debe definirse como tal, sin hacer recaer en el predicado que lo invoque el impedir que se produzca una vuelta atrás no deseada.
- Se debe evitar reproducir estilos de programación que provienen de otros lenguajes.

Los apartados que siguen desarrollan criterios más específicos que deben tenerse en cuenta para tratar de maximizar la eficiencia de los programas. El predicado predefinido

`statistics`, presente en la mayoría de sistemas Prolog, puede informar del consumo de tiempo y memoria de una computación. Muchos sistemas ofrecen predicados adicionales para evaluar la eficiencia del programa.

Aprovechar la indexación de cláusulas Muchas implementaciones de Prolog construyen, para cada símbolo de predicado, una tabla *hash* que se indexa por el functor principal del primer argumento de las cabezas de las cláusulas que definen dicho predicado. Esto permite, en tiempo de ejecución, evitar los intentos de resolución con las cláusulas cuya cabeza tiene un primer argumento incompatible con el de la llamada, eliminándose incluso los puntos de elección asociados a las cláusulas no seleccionables. Esto significa que el subconjunto de cláusulas que se pueden resolver con un objetivo dado se encuentran muy rápido, prácticamente en tiempo constante, lo cuál puede resultar muy importante cuando el número de cláusulas para un predicado dado es grande. La indexación se utiliza tanto en los intérpretes como en los compiladores.

Teniendo en cuenta esto, en la definición inductiva de un predicado, es conveniente que el argumento sobre cuya estructura se basa la inducción (o, en general, aquel que discrimina cuál es la cláusula aplicable), sea el primer argumento del predicado en cuestión.

Por ejemplo, un programa como el que sigue

```
ultimo([Solouno], Solouno).
ultimo([Primero|Resto], Ultimo) :- ultimo(Resto, Ultimo).
```

se ejecutará más rápidamente que

```
ultimo(Solouno, [Solouno]).
ultimo(Ultimo, [Primero|Resto]) :- ultimo(Ultimo, Resto).
```

Por otro lado, cuando el sistema Prolog puede leer consecutivamente todas las cláusulas para un predicado hará una rutina más rápida que cuando las lee intercaladas con otras cláusulas, o si se añaden y borran por medio de `assert` y `retract`.

Delegar trabajo a la unificación El mecanismo de unificación es un método muy potente y hay que aprender a delegarle la mayor carga posible de trabajo.

Por ejemplo, si queremos programar una rutina para comprobar si una lista contiene exactamente dos números naturales iguales, una versión absurdamente larga, difícil de entender y lenta del programa sería la siguiente.

```
dosiguales(L) :- L=[H|T], T=[H1|T1],
    nat(H), H1==H,
    length(T1,N), N=0.
```

mientras una versión concisa, intuitiva y eficiente sería:

```
dosiguales([X,X]) :- nat(X).
```

Para apreciar la diferencia entre ambas versiones basta imaginar un escenario en el que el argumento está desinstanciado en la llamada a este predicado.

No transformar problemas deterministas en indeterministas Cuando un problema es determinista, no es rentable programar de forma indeterminista el cálculo de su solución. El indeterminismo es útil cuando existen distintas soluciones para un problema dado pero debe evitarse cuando resulta innecesario.

Por ejemplo, sería un desacuerdo invocar a un predicado indeterminista como `member` para comprobar si dos listas básicas contienen los mismos elementos:

```
mismos_elem(L1,L2) :- not(member(X,L1), not(member(X,L2)),  
not(member(X,L2)), not(member(X,L1))).
```

Teniendo en cuenta la existencia de algoritmos muy eficientes para la ordenación de listas, un programa alternativo con mejor comportamiento sería:

```
mismos_elem(L1,L2) :- sort(L1,ListaOrdenada),  
sort(L2,ListaOrdenada).
```

Descartar lo antes posible soluciones inválidas En el Apartado 7.5 hemos estudiado el método de generación y prueba. Cuando se aplica esta técnica, es conveniente ordenar los subobjetivos en el cuerpo de las cláusulas para detectar, lo antes posible, la eventual invalidez de una solución parcial generada (por no satisfacer alguna condición que debe cumplir la solución final).

Por ejemplo, supongamos el problema de obtener los números Brandreth (véase Ejercicio 6.21). Una versión ingenua del programa procedería del siguiente modo:

```
generar(1).  
generar(N) :- generar(M), N is M+1.  
  
brand(N) :- generar(N), C is N * N, N1 is C // 100,  
N2 is C mod 100, N is N1 + N2.
```

Teniendo en cuenta que solo los números entre 32 y 99 tienen un cuadrado de 4 cifras, un programa más refinado para resolver este mismo problema es el siguiente:

```
test_between(N,M,X) :- N=<X, X=<M.  
  
brand(N) :- generar(N), test_between(32, 99, N), C is N * N,  
N1 is C // 100, N2 is C mod 100, N is N1 + N2.
```

Y mucho mejor si la solución candidata se genera ya de forma que satisface la restricción de pertenecer al rango 32..99:

```

between(N,N,N) :- !.
between(N1,N2,N1) :- N1 < N2.
between(N1,N2,M) :- A is N1+1, between(A,N2,M).

brand(N) :- between(32, 99, N), C is N * N, N1 is C // 100,
           N2 is C mod 100, N is N1 + N2.

```

Olvidar las variables globales En programación imperativa es bastante frecuente utilizar variables globales para resolver muchos problemas. Cuando se transfiere este hábito a la programación en un lenguaje lógico, suele obtenerse programas pobres y muy inefficientes.

Por ejemplo, en los programadores Prolog no experimentados es frecuente el vicio de querer simular un bucle utilizando una variable global para contabilizar el número de veces que éste se ejecuta, como ilustra el siguiente ejemplo.

```

bucle(P,N) :- N>=1,
              assert(contador(N)),
              itera(P).

itera(P) :- retract(contador(N)),
            M is N-1, M>0,
            call(P),
            assert(contador(M)),
            itera(P).

```

Aunque depende del compilador, usualmente `assert` y `retract` son muy lentos y solo deben usarse con el propósito de almacenar información que deba sobrevivir a la vuelta atrás. Cuando meramente se pretende pasar resultados intermedios de un paso de computación al siguiente conviene utilizar argumentos.

Una versión mejor del programa anterior, que puede ser del orden de 100 veces más rápida, sería la siguiente:

```

itera(P,0).
itera(P,N) :- N>0,
              call(P),
              M is N-1,
              itera(P,M).

```

Una situación en la que sí resulta efectivo utilizar la base de datos de Prolog es para almacenar resultados de computaciones ya realizadas y que puedan ser útiles más adelante, evitando así tener luego que recomputarlos.

Por ejemplo, si deseamos definir un predicado que permita computar los elementos de la sucesión de Fibonacci: $a_1 = 1$, $a_2 = 1$, y $a_n = a_{n-1} + a_{n-2}$ si $n > 2$. La solución ingenua para este problema, que implementa la recursión directamente, es:

```

fib(0,0).

```

```
fib(1,1).
fib(N,F) :- N>1,
    N1 is N-1,
    N2 is N1-1,
    fib(N1,F1),
    fib(N2,F2),
    F is F1+F2.
```

Esta solución es muy inefficiente porque tienden a recalcular un gran número de resultados intermedios. El siguiente programa que calcula la función de Fibonacci para un número N dado, utiliza el predicado predefinido `assert` para incorporar a la base de datos, a modo de lema, los resultados que se van obteniendo, de tal forma que solamente se tengan que calcular una vez y puedan reutilizarse cuando se calcula el valor de la función para números más grandes.

```
fib_x(0,0).
fib_x(1,1).

fib(N,F) :- fib_x(N,F), !.
fib(N,F) :- N>1,
    N1 is N-1,
    N2 is N1-1,
    fib(N1,F1),
    fib(N2,F2),
    F is F1+F2,
    assert(fib_x(N,F)).
```

Para $N > 25$, el programa anterior puede ocupar del orden de 1000 veces menos memoria que la versión ingenua.

Como ya hemos dicho, los predicados que almacenan los resultados intermedios de una computación, se denominan *funciones-memo*. El predicado `fib_x` es un ejemplo concreto de este tipo de “funciones”.

Usar las listas cuando sean necesarias Las listas de Prolog son estructuras muy potentes para manipular datos pero resultan también muy costosas. Una deformación muy frecuente en los hábitos de programación en Prolog es abusar excesivamente de este tipo de estructuras, en particular cuando pueden existir estructuras más simples que resuelven el mismo problema consumiendo menos memoria y proporcionando acceso directo incluso a cada una de las componentes.

Por ejemplo, consideremos un problema en el que debemos trabajar con un predicado que se aplica a tuplas de componentes

```
(nombre, apellido1, apellido2, dni, dirección).
```

En este caso será más productivo utilizar una estructura del tipo

```
datos_personales(Nombre, Apellido1, Apellido2, DNI, Dirección)
```

que trabajar con listas de cinco elementos

```
[Nombre,Apellido1,Apellido2,DNI,Direccion|[]].
```

Otras veces es preferible utilizar árboles de búsqueda (diccionarios) en lugar de listas ordenadas. Los diccionarios facilitan las consultas y la inserción de nuevos elementos donde corresponde.

Cuando se trabaja con listas, conviene recordar que el predicado `append` es muy inefficiente. Siempre que sea posible, conviene trabajar con los elementos en la cabeza, para evitar recorrer toda la estructura. También se debe evitar recorrer varias veces una lista si es posible hacer todo el trabajo en solo una pasada. Por ejemplo, como ya se ha explicado, la versión estándar del predicado de inversión de listas definido en el Apartado 6.3.2, recorre dos veces la estructura de la primera lista mientras que, gracias al uso de un parámetro acumulador, la versión refinada, que también aparece en el Apartado 6.3.2, la recorre solo una.

En la próxima sección veremos que el uso de estructuras de datos incompletas incrementa más todavía la potencia y eficiencia de la manipulación de listas en Prolog.

7.7 ESTRUCTURAS AVANZADAS: LISTAS DIFERENCIA Y DICIONARIOS

Acabamos de discutir en el apartado anterior una optimización del programa para inversión de listas que se basa en el uso de un parámetro acumulador. Del empleo de tales acumuladores hay sólo un paso a una importante representación alternativa para las listas conocida como las *listas diferencia*, probablemente una de las técnicas de programación más ingeniosas jamás inventadas introducida por Tärnlund en 1975.

Uno de los mayores inconvenientes de la operación de concatenación de listas realizada por el predicado `append(X, Y, Z)` es que su ejecución tiene un coste temporal lineal con la longitud de la primera lista. Las listas diferencia pueden entenderse como una generalización del concepto de lista que permite la concatenación de listas en tiempo constante. El hecho de que muchos programas precisen esta operación justifica por sí mismo la importancia de este tipo de estructuras.

La idea sencilla en que se basan las listas diferencia es que las estructuras de datos en Prolog pueden contener variables. Dichas variables pueden verse como “agujeros” o, si se prefiere, como punteros a otras estructuras que serán construidas más tarde. Para extender la estructura, basta llenar dichos agujeros.

Por ejemplo, `t(Y, t(1, t(Y, Y)))` es un árbol que contiene el mismo subárbol `y` desconocido en tres lugares. Similarmente, si `X = [1, 2 | Y]` y más adelante la variable `y` se instancia a la lista `[3, 4 | Z]`, desde ese momento tendremos que `X = [1, 2, 3, 4 | Z]`. Es decir, podemos añadir datos a `x` simplemente instanciando la variable `y` que se encuentra en su cola, sin tener que recorrer previamente los prefijos de ninguna de las dos estructuras. Más aún, podremos añadir elementos más tarde instanciando también la variable `z`.

Utilizando este tipo de listas parcialmente instanciadas es posible construir una estructura de datos alternativa a las listas de Prolog, conocida como *lista diferencia* para representar secuencias de elementos. En lo que sigue usaremos por claridad y conveniencia notacional² el operador infijo de substracción “-” para construir tales listas. Su uso no guarda ninguna relación con la aritmética, aunque intuitivamente debería leerse como “diferencia”.

Para explicar las listas diferencia, vamos a partir de la idea básica de que toda secuencia de elementos puede representarse como la diferencia entre un par de listas. Por ejemplo, la secuencia 1, 2, 3 es la diferencia entre las listas:

[1, 2, 3, 4, 5]	y	[4, 5]
[1, 2, 3, 8]	y	[8]
[1, 2, 3]	y	[]

Podemos ver una lista diferencia como un par de listas, que denotamos $L_1 - L_2$, donde la lista L_2 es un sufijo de L_1 . La lista representada por esta estructura debe entenderse como la diferencia entre las listas L_1 y L_2 . La lista L_1 se llama “lista +” y la lista L_2 se llama “lista -”.

Generalizando, una lista diferencia es una construcción de la forma

[a₁, a₂, ..., a_n | X] - X

donde x es una variable. Esta estructura representa la lista [a₁, a₂, ..., a_n] (el resultado de la diferencia entre las dos listas).

Ejemplo 7.15

La lista diferencia más general mediante la que podemos representar la secuencia 1, 2, 3 es el par [1, 2, 3 | X] - X, donde [1, 2, 3 | X] es la lista + y X es la lista -.

Por supuesto, la lista y la lista diferencia son términos completamente diferentes que no pueden ser unificados. De hecho, recordemos que Prolog no evalúa términos salvo en circunstancias muy especiales.

7.7.1. Relación entre listas y listas diferencia

Para optimizar un programa que manipula listas, puede resultar conveniente utilizar las facilidades que ofrecen las listas diferencia. Para ello resultará necesario transformar primero las listas al formato de las listas diferencia para, más tarde, volver al formato de

²Observe que se podría haber empleado el símbolo “\” o cualquier otro. Por otra parte, téngase en cuenta que el empleo de un operador no es imprescindible para definir este concepto y sólo aporta claridad. Este concepto también podría concretarse suministrando los dos componentes de una lista diferencia como argumentos separados (y generalmente consecutivos) del predicado que se está definiendo y hace uso de la lista diferencia. Esto último aporta una mayor eficiencia (porque se eliminan los costes de almacenar el operador “-” cuando se forman estructuras) a costa de una pérdida de legibilidad.

las listas originales. La manera de realizar estas conversiones se basa en las siguientes reflexiones.

- Cualquier lista L puede representarse como la lista diferencia $L - []$.
- La lista vacía se representa como una lista diferencia cuyas dos listas componentes coinciden. Su representación más general como lista diferencia sería: $x - x$.

Tomando esto en cuenta, podemos ya escribir predicados de conversión entre las dos representaciones. Para empezar, la lista “plana” puede extraerse en cualquier momento de la lista diferencia simplemente instanciando la variable x a $[]$, de manera que en el primer argumento del operador $/2$ se obtiene lo que se quería extraer.

```
de_lista_diferencia_a_lista(L,L-X) :- X= [] .
```

o simplemente

```
de_lista_diferencia_a_lista(L,L-[]).
```

Obsérvese por ejemplo la respuesta a la siguiente llamada:

```
?- de_lista_diferencia_a_lista(L,[1,2,3|Y]-Y).  
L=[1,2,3], Y=[]
```

Sin embargo, cuando se pretende utilizar el predicado “`de_lista_diferencia_a_lista/2`” para obtener, a partir de una lista convencional dada, su correspondiente representación como lista diferencia, no se obtiene el resultado pretendido, ya que la lista diferencia resultante no es la más general posible. De hecho, no contiene una variable en la cola, lo que impediría ir ampliándola después incrementalmente.

```
?- de_lista_diferencia_a_lista([1,2,3],Ld).  
Ld=[1,2,3]-[]
```

Para obtener la lista diferencia pretendida $Ld = [1, 2, 3 | Y] - Y$, hay que usar el siguiente predicado que hace uso de la concatenación:

```
de_lista_a_lista_diferencia(L,M-X) :- append(L,X,M).
```

Observe ahora la respuesta al objetivo:

```
?- de_lista_a_lista_diferencia([1,2,3],Ld).  
Ld=[1,2,3|X]-X
```

Para terminar, es importante señalar que este predicado tampoco se puede invertir ya que, además de no obtenerse la respuesta esperada, el interprete Prolog puede producir respuestas que no son correctas. Cuando hacemos la pregunta simétrica

```
?- de_lista_a_lista_diferencia(L, [1, 2, 3|X] -X).
```

la ejecución entra en un “bucle” intentando producir la respuesta

```
L = []
X = [1, 2, 3, 1, 2, 3, 1, 2, 3|...]
```

Recuerde que el algoritmo de unificación del lenguaje Prolog no utiliza el mecanismo de *occur check* y, por tanto, la cabeza de la primera cláusula de la definición del predicado `append` (véase el Apartado 6.3.2) unifica con la llamada “`append(L, X, [1, 2, 3|X])`” iniciando de esta forma la computación divergente que se muestra.

7.7.2. Manipulación de listas diferencia

El uso de listas diferencia en lugar de listas conduce a programas más concisos y eficientes. Por ejemplo, la concatenación de dos listas diferencia con un coste constante puede hacerse como sigue

```
append_(X-Y, Y-Z, X-Z).
```

El predicado `append_` queda definido por un único hecho que, en términos de la concatenación simple de dos listas, puede entenderse de forma intuitiva del modo siguiente:

Si $X-Y = [A|Y] - Y$ e $Y-Z = [B|Z] - Z$ entonces $X-Z = [C|Z] - Z$; donde C es la lista que resulta de concatenar las listas A y B .

Para ilustrar esta afirmación, observe el resultado cuando ejecutamos los objetivos:

```
?- append_(([a, b, c|Y] -Y, [1, 2] -[], D).
Y=[1, 2], D=[a, b, c, 1, 2] -[])
```

```
?- append_(([a, b, c|Y] -Y, [1, 2|Z] -Z, D).
Y=[1, 2], D=[a, b, c, 1, 2|Z] -Z)
```

En los objetivos anteriores la variable D queda ligada a listas diferencia que representan la lista $[a, b, c, 1, 2]$ resultado de concatenar las listas $[a, b, c]$ y $[1, 2]$. De hecho, instanciando convenientemente el argumento de salida, podemos obtener directamente dicha lista como una instancia de una variable específica:

```
?- append_(([a, b, c|Y] -Y, [1, 2|Z] -Z, C- []).
Y=[1, 2], C=[a, b, c, 1, 2])
```

Este método de instanciación se puede utilizar como un modo alternativo para “enlazar” un programa que usa listas diferencia con un programa que espera listas convencionales como entrada. El programa resultante de sustituir las llamadas al predicado `append` por una llamada al nuevo `append_` realiza la operación de concatenación “*on-the-fly*” y recuerda mucho los programas con acumuladores. Para ver esto consideremos nuevamente el programa de inversión de listas. La versión de este programa usando listas diferencia es como sigue:

```

invertir(L,I) :- invertir_(L,I-[ ]).

invertir_([ ],X-X).
invertir_([H|T],Y-Z) :- invertir_(T,Y-[H|Z]).
```

Es decir, si la inversa de T se representa por la lista diferencia Y- [H|Z], entonces añadir H en la cabeza de T es lo mismo que eliminar H de la lista - en la lista diferencia). Note que este programa se puede obtener, a partir del programa de inversión de listas con parámetro acumulador, comutando los dos últimos argumentos y sustituyendo la "," (la coma) por el "-".

Una llamada al nuevo predicado de inversión proporciona la lista diferencia más general que representa la lista invertida.

```
?- invertir_([1,2,3,4],I).
I=[4,3,2,1|X]-X
```

Una transformación similar, aplicada al programa quicksort, nos permite sintetizar una versión del programa empleando las listas diferencia.

```

% qs(Xs, Ys), si Ys es una permutacion ordenada de la lista Xs
qs(Xs, Ys) :- qs_dl(Xs, Ys - [ ]).

% qs_dl(Xs, Y), si Y es una lista diferencia que representa la
% permutacion ordenada de la lista Xs.
qs_dl([ ], Xs - Xs).
qs_dl([X | Xs], Ys - Zs) :-
    partit(X, Xs, Menores, Mayores),
    qs_dl(Menores, Ys - [X | Y1s]),
    qs_dl(Mayores, Y1s - Zs).

partit(_, [ ], [ ], [ ]).
partit(X, [Y | Xs], [Y | Ls], Bs) :- X > Y, partit(X, Xs, Ls, Bs).
partit(X, [Y | Xs], Ls, [Y | Bs]) :- X <= Y, partit(X, Xs, Ls, Bs).
```

A continuación veremos la implementación, usando listas diferencia, de otro par de operaciones sobre listas que son típicas de las estructuras de tipo cola.

El siguiente programa sirve para añadir un elemento al final de una cola.

```
encola_(E,Lista-[E|NuevoFin],Lista-NuevoFin).
```

Observe los enlaces computados tras la siguiente llamada:

```
?- encola_(4,[1,2,3|X]-X,NL).
NL=[4|NuevoFin], NL=[1,2,3,4|NuevoFin]-NuevoFin.
```

Si queremos eliminar el primer elemento de la cola, podemos utilizar el siguiente predicado.

```
desencola_(E, [E|Lista]-Fin, Lista-Fin) .  
  
?- desencola_(E, [1,2,3,4|X]-X, NL) .  
E=1, NL=[2,3,4|X]-X.
```

La siguiente operación adicional `setup(Q)` puede utilizarse para crear una cola vacía:

```
setup(Q-Q) .
```

El siguiente objetivo ilustra el uso del programa completo para tratamiento de colas:

```
?- setup(X), encola_(a,X,Y), encola_(b,Y,Z), desencola_(A,Z,U),  
desencola_(B,U,V) .  
  
A=a,  
B=b,  
U=[B|_A]-_A,  
V=_A-_A,  
X=[a,b|_A]-[a,b|_A] .  
Y=[a,b|_A]-[b|_A] .  
Z=[a,b|_A]-_A
```

Resulta interesante observar que este programa también puede crear colas “negativas” en el sentido de que, en un cierto instante, el número de operaciones `desencola_` puede exceder el número total de operaciones `encola_`. En este caso, las variables presentes en las operaciones `desencola_` quedarán eventualmente instanciadas a los valores encolados más tarde. Por ejemplo:

```
?- setup(X), desencola_(A,X,U), desencola_(B,U,V),  
encola_(a,V,Y), encola_(b,Y,Z) ,  
  
A=a,  
B=b,  
U=[B|_A]-[a,b|_A] ,  
V=_A-[a,b|_A] ,  
X=[a,b|_A]-[a,b|_A] .  
Y=_A-[b|_A] .  
Z=_A-_A
```

Finalmente ilustramos cómo podríamos manejar una agenda usando las operaciones estudiadas en este apartado para manipulación de lista diferencia.

```
% Construir una nueva agenda vacía:  
nueva_agenda(A - A) .  
  
% Insertar un ítem al principio de la agenda, obteniendo una agenda  
% nueva  
insertar_al_inicio(Ítem,  
                    Agenda - ColaVar,                      % Vieja agenda
```

```

[Item|Agenda] - ColaVar).      % Nueva agenda

% Insertar un item al final de la agenda, obteniendo una agenda
% nueva
insertar_al_final(Item,
                  Agenda - [Item|NuevaColaVar], % Vieja agenda
                  Agenda - ColaVar).          % Nueva agenda

```

Así, dados los siguientes objetivos:

```
?- nueva_agenda(X), insertar_al_inicio(inicio, X, NuevaX).
```

la variable `NuevaX` se instanciará a una agenda de la forma: `[inicio|X]-X`.

Es inmediato observar que la implementación de agendas mediante listas diferencia ahorrando el coste de recorrer la lista: en particular, el coste de añadir un elemento en cualquiera de los dos extremos de la lista es trivial e idéntico en ambos casos.

7.7.3. Diccionarios

Otro uso de las estructuras de datos incompletas en Prolog se encuentra en la creación, uso y mantenimiento de diccionarios o conjuntos de valores indexados por una clave.

En este caso, son importantes las siguientes dos operaciones: 1) buscar el valor asociado a una clave; 2) añadir nuevas claves y sus valores correspondientes. Dichas operaciones deben realizarse preservando la consistencia de la estructura: la misma clave no puede aparecer con dos valores diferentes en un mismo diccionario. En lo que sigue, vamos a ver que ambas operaciones pueden hacerse con una única operación simple cuando se usan estructuras de datos incompletas.

Primera versión: Representación de un diccionario como una lista lineal incompleta de pares de la forma `(Clave,Valor)`.

La siguiente relación `examinar(Clave,Dic,Valor)` tiene éxito cuando existe un par en `Dic` cuyo primer elemento es `Clave` y que tiene `Valor` como segunda componente.

```

% examinar(Clave,Dic,Valor), tiene exito cuando el diccionario Dic
% contiene el Valor en la posicion indexada por la Clave.

examinar(Clave, [(Clave,Valor)|Dic], Valor).

examinar(Clave, [(Clave1,Valor1)|Dic], Valor):- 
    Clave \== Clave1, examinar(Clave,Dic,Valor).

```

Supongamos, por ejemplo, que el diccionario contiene las extensiones telefónicas de algunas personas, tomando como clave el nombre de las mismas, y que `Dic` está inicialmente instanciada a `[(alpuente,3512),(julian,3716)|X]`. Entonces, es posible formular los siguientes objetivos para:

- Buscar un número de teléfono.

```
?-Dic=[(alpuente,3512), (julian,3716)|X],  
          examinar(alpuente,Dic,N).  
N = 3512,  
Dic = [(alpuente,3512), (julian,3716)|X]  
yes
```

- Comprobar el número de teléfono de una persona.

```
?-Dic=[(alpuente,3512), (julian,3716)|X],  
          examinar(julian,Dic,3716).  
Dic = [(alpuente,3512), (julian,3716)|X]  
yes
```

- Introducir nuevos valores.

```
?-Dic=[(alpuente,3512), (julian,3716)|X],  
          examinar(lucas,Dic,3522).  
X = [(lucas,3522)|_A],  
Dic = [(alpuente,3512), (julian,3716), (lucas,3522)|_A]  
yes
```

- Comprobar la consistencia.

```
?-Dic=[(alpuente,3512), (julian,3716)|X],  
          examinar(alpuente,Dic,3511).  
no
```

Este último objetivo, en lugar de introducir una nueva entrada para `alpuente`, responde “no” porque la primera cláusula que define el predicado `examinar` detecta una entrada ya existente con la misma Clave pero diferente Valor.

Segunda versión: Representación de un diccionario como un árbol binario ordenado.

En este caso, representamos el diccionario como una estructura con 4 componentes `dic(Clave, Valor, Izq,Der)`, donde `Izq` y `Der` son, respectivamente, los subdicionarios izquierdo y derecho, y `Clave` y `Valor` representan la clave y el valor asociado.

De acuerdo con la nueva representación, la relación `examinar(Clave,Dic,Valor)` puede codificarse como sigue:

```
examinar(Clave,dic(Clave,Valor,Izq,Der),Valor):- !.  
examinar(Clave,dic(Clave1,_,Izq,Der),Valor):-  
          Clave@<Clave1,examinar(Clave,Izq,Valor).  
examinar(Clave,dic(Clave1,_,Izq,Der),Valor):-  
          Clave@>Clave1,examinar(Clave,Der,Valor).
```

Observe que es necesario emplear el corte porque la naturaleza extralógica de los operadores de comparación puede conducir a comportamientos anómalos cuando las claves no están instanciadas.

El lector puede fácilmente comprobar que esta representación permite también búsquedas e inserciones muy eficientes. Por ejemplo, observe la siguiente secuencia de llamadas (donde la primera llamada utiliza una variable desinstanciada BT):

```
?- examinar(lucia,BT,38), examinar(diego,BT,46),
   examinar(lucia,BT,Edad) .  
  
BT=dic(lucia, 38, _C, dic(diego, 46, _B, _A)), Edad=38.
```

Las dos primeras llamadas inicializan el diccionario para contener las dos entradas consideradas, mientras que la última llamada consulta la edad de lucia en el diccionario.

Finalmente, obsérvese que para encontrar la clave (o claves) asociada a un valor, se debe implementar el recorrido del árbol binario aplicando las técnicas estudiadas en la Sección 6.7.2 sobre árboles de búsqueda.

RESUMEN

En este capítulo hemos continuado nuestro estudio del lenguaje Prolog discutiendo algunas características extralógicas y ciertos aspectos avanzados.

- En Prolog, el control es fijo y viene dado de forma automática. Ahora bien, el lenguaje Prolog proporciona una serie de predicados extralógicos, como el corte (!), que permite la poda del espacio de búsqueda, o fail, que permite el control de la vuelta atrás.
- Cuando el corte aparece en una cláusula, su función es impedir la reevaluación de todos los subobjetivos que quedan a su izquierda en la cláusula de la que forma parte. Esto incluye la cabeza de la cláusula, que ya no intentará resatisfacerse. Para ilustrar el funcionamiento del corte y cómo produce la poda del árbol de búsqueda hemos estudiado algunos ejemplos concretos.
- Para sistemas de cláusulas de Horn definidas es imposible extraer información negativa utilizando la estrategia de resolución SLD. Para resolver el problema de la negación se han adoptado diferentes soluciones. Hemos presentado dos de las más populares en el campo de la programación lógica: la suposición de un mundo cerrado y la negación como fallo (un caso particular de la anterior).
- La suposición de un mundo cerrado es una caracterización de la negación, surgida en el ámbito de las bases de datos deductivas [125], en la que todo hecho que no se afirma explícitamente se considera falso. Más formalmente, dado un programa

Π y un átomo \mathcal{A} de la base de Herbrand del programa, podemos afirmar $\neg\mathcal{A}$ si \mathcal{A} no es consecuencia lógica del programa Π .

- La negación como fallo (Clark [28]) puede considerarse que es una restricción de la suposición de un mundo cerrado, en la que para afirmar la negación de un átomo \mathcal{A} de la base de Herbrand del programa, es suficiente que exista un árbol SLD de fallo finito para $\Pi \cup \{\leftarrow \mathcal{A}\}$.
- El lenguaje Prolog proporciona una forma de negación basada en el principio de negación como fallo, aunque no se corresponde exactamente con él. El predicado predefinido `not` se implementa haciendo uso del corte. La definición de `not` se ha construido de forma que `not(A)` falla si `A` tiene éxito y, por contra, `not(A)` tiene éxito si `A` falla.
- Dado que el lenguaje Prolog es un lenguaje orientado a la manipulación simbólica el repertorio de operaciones aritméticas suele ser limitado. Sin embargo, Prolog dispone de una amplia gama de predicados predefinidos para la catalogación y manipulación de términos así como para la metaprogramación.
- La técnica de *generar y comprobar* consiste en generar una configuración (o estado) y comprobar que cumple las restricciones que la acreditan como una solución (o coincide con un estado objetivo). Muchos problemas se pueden ajustar a este esquema. Aunque esta técnica no es efectiva cuando se intentan resolver problemas complejos, su eficiencia puede mejorar mediante la incorporación de métodos heurísticos (véase más adelante en el Capítulo 9).
- Existen algunos principios básicos a seguir para desarrollar código más claro y eficiente. Como norma general, se debe evitar reproducir estilos de programación que provienen de otros lenguajes, como puede ser el uso de variables globales. Conviene buscar siempre soluciones deterministas para problemas que son deterministas. El corte puede ayudar a garantizar el determinismo pero debe usarse con cautela y solo en los lugares adecuados. Finalmente, es aconsejable realizar cuanto antes las comprobaciones para desechar las soluciones inválidas mientras que, por otro lado, conviene retrasar los cálculos hasta necesitar estrictamente sus resultados. Como último consejo: la unificación es muy potente: ¡úsela!
- El empleo de estructuras de datos incompletas posibilita el desarrollo de técnicas de programación muy potentes, como se muestra al tratar con listas diferencia y diccionarios.
- Uno de los mayores inconvenientes de la operación de concatenación de listas realizada por el predicado `append/3` es que su ejecución tiene un coste temporal lineal con la longitud de la primera lista. Las listas diferencia pueden entenderse como una generalización del concepto de lista que permite la concatenación en

tiempo constante. El hecho de que muchos programas precisen esta operación justifica por sí mismo la importancia de este tipo de estructuras.

- Las estructuras de datos incompletas, como es el caso de los diccionarios, pueden generarse incrementalmente en el curso de una computación.

CUESTIONES Y EJERCICIOS

Cuestión 7.1 Definanse los conceptos de corte verde y corte rojo.

Ejercicio 7.2 Estudie el comportamiento de los programas:

Programa 1	Programa 2
p(X,Y) :- q(Y), r(X). %1	p(X,Y) :- q(Y), r(X). %1
p(X,Y) :- s(X), r(Y). %2	p(X,Y) :- s(X), r(Y). %2
q(X) :- s(X), b. %3	q(X) :- s(X), !, b. %33
q(X) :- r(X). %4	q(X) :- r(X). %4
r(uno). %5	r(uno). %5
r(dos). %6	r(dos). %6
s(tres). %7	s(tres). %7
a. %8	a. %8
b:-fail. %9	b:-fail. %9

1. Dibuje y compare el árbol de búsqueda de Prolog para estos dos programas y el objetivo “?- p(X, Y)”. Indíquese la cláusula y la sustitución computada en cada paso, así como la respuesta computada en cada derivación. ¿Qué ramas han sido podadas?
2. El corte introducido en el programa ¿es rojo o verde? Razonar la respuesta.

Ejercicio 7.3 Sean los siguientes programas escritos en Prolog:

```
% member(E, L), E está en L
member(E, [E|_]). 
member(E, [_|L]) :- member(E, L).


---


% member2(E, L), E está en L
member2(E, [E|_]) :- !.
member2(E, [_|L]) :- member2(E, L).
```

1. Dibuje y compare el árbol de búsqueda de Prolog para el programa `member` y el objetivo “?- `member(X, [1, 2, 3])`” con el correspondiente árbol para el programa `member2` y el objetivo “?- `member2(X, [1, 2, 3])`”. Indíquese la cláusula y la sustitución computada en cada paso, así como la respuesta computada en cada derivación.

2. *El corte introducido en la definición de member2, ¿es un corte verde o rojo?*
3. *¿Cuál sería la utilidad del predicado member2?*

Ejercicio 7.4 Defina un predicado add(X, L, L1) que añada un elemento X, a la lista L para dar L1, sin incurrir en repeticiones de elementos. Esto puede hacerse mediante la siguiente regla:

Si X es miembro de la lista L entonces L1 = L; si no, L1 es igual a la lista [X | L];

para cuya implementación es necesario el empleo del corte. Compruebe que omitiendo el corte o alguna construcción derivada del corte (e. g. el predicado not) sería posible la adición de elementos repetidos. Por ejemplo:

```
?- add(a, [a, b, c], L).  
L = [a, b, c];  
L = [a, a, b, c]
```

Así pues, en la solución de este ejercicio es vital el empleo del corte para obtener el significado esperado de la especificación y no como un mero instrumento para aumentar la eficiencia.

Ejercicio 7.5 Defina la relación elimina(X, L1, L2), donde L2 es el resultado de eliminar la primera aparición del elemento X en la lista L1 (**Ayuda:** utilice el corte).

Ejercicio 7.6 (Celos en el Puerto) En el Ejercicio 3.11 vimos que, aunque el programa obtenido era declarativamente correcto, tenía un comportamiento procedural incorrecto. Esto era debido a que el programa, cuando se ejecutaba la pregunta deseada, entraba por una rama infinita, computando siempre la misma respuesta. Emplea el corte para podar la rama infinita que conduce a respuestas redundantes e impide que se computen el resto de respuestas correctas.

Ejercicio 7.7 Queremos definir un predicado diferente(X, Y) que sea verdadero cuando X e Y no unifiquen. Esto puede implementarse en Prolog teniendo en cuenta que:

*Si X e Y unifican
entonces diferente(X, Y) falla; si no, diferente(X, Y) tiene éxito.*

Defina este predicado empleando: a) el corte, fail y true; b) empleando únicamente el predicado predefinido not. (**Observación:** Está prohibido utilizar el predicado predefinido "\=", que tiene éxito cuando dos expresiones, suministradas como argumentos, no unifican.)

Ejercicio 7.8 El predicado predefinido `repeat` no es imprescindible en la programación con el lenguaje Prolog. a) Para confirmar la afirmación anterior, escriba un pequeño programa llamado `cubo` que solicite un número por teclado y calcule su cubo. El programa debe operar como a continuación se indica:

```
?- cubo.
Siguiente numero: 5.
El cubo de 5 es 125
Siguiente numero: 3.
El cubo de 5 es 27
Siguiente numero: stop.
yes
```

b) Dé una versión del mismo enunciado empleando `repeat`.

Cuestión 7.9 Dé una definición operacional de la regla de negación como fallo.

Cuestión 7.10 La regla de negación como fallo:

- Permite dar un paso de inferencia si es posible establecer una refutación para el programa y un átomo básico negado.
- Permite dar un paso de inferencia si es posible establecer una refutación para el programa y un átomo básico.
- Permite dar un paso de inferencia si es posible obtener un árbol de fallo finito para el programa y un átomo básico (tomado como objetivo).
- Permite dar un paso de inferencia si es posible obtener un árbol de fallo finito para el programa y un átomo (tomado como objetivo).

Seleccione la respuesta correcta de entre las anteriores.

Ejercicio 7.11 Sea el programa lógico

$$\Pi \equiv \{q(a) \leftarrow p(b) \wedge q(b). \quad p(b) \leftarrow q(b).\}.$$

Calcule la semántica de la negación para el programa Π , empleando la regla de NAF.

Ejercicio 7.12 Dado el programa definido

$$\{p(X) \leftarrow q(Y, X) \wedge r(Y). \quad q(s(X), Y) \leftarrow q(Y, X). \quad r(0)\}.$$

Muestre que existe una regla de cómputo para la que puede construirse un árbol de búsqueda SLD de fallo finito.

Ejercicio 7.13 Dado el programa normal

```
{vuela(X) ← pájaro(X) ∧ ¬anormal(X).  
    pájaro(águeda).    pájaro(perdiz).  
    pájaro(pinguino). anormal(pinguino).  
    pájaro(avestruz).  anormal(avestruz).}.
```

Construya un árbol de búsqueda SLDNF para el objetivo “ $\leftarrow \text{vuela}(X)$ ”.

Ejercicio 7.14 (Problema de las torres de Hanoi) Modifique el programa de las torres de Hanoi, propuesto en el Ejercicio 6.26, para que el plan resultante se muestre directamente por pantalla.

Ejercicio 7.15 Escriba un programa que cuente los caracteres, las palabras y las líneas contenidas en un fichero. Procure dar un formato agradable a la salida por pantalla.

Ejercicio 7.16 Definase el predicado leerPalabra(P), que lee una palabra desde el teclado, introducida como una secuencia de caracteres ASCII, e instancia P con el atomo formado por la palabra. Nótese que este predicado trata de simular el predicado estándar read(P), que no debe usarse en la implementación de leerPalabra (**Ayuda:** Utilice el predicado de E/S get0(Char) y el predicado name(Atomo, Secuencia), que transforma una Secuencia en un Atomo —y viceversa—).

Ejercicio 7.17 Ejecute objetivos en un sistema Prolog que sean capaces de construir los términos: “f(X, g(a))” y “fecha(9, marzo, 1998)”. Use los predicados predefinidos functor y arg.

Cuestión 7.18 Cuando se plantea el objetivo “?- f(g(X, h(Y)), Z) =.. W” en un intérprete de Prolog, el resultado es:

- X = X', Y = Y', Z = Z' W = [f, g(X', h(Y')), Z'].
- X = X', Y = Y', Z = Z' W = [f, g, X', h, Y', Z'].
- W = [f, g(X, h(Y)), Z].
- W = [f, g, X, h, Y, Z].

Seleccione la respuesta correcta de entre las anteriores.

Ejercicio 7.19 Las reglas de producción:

```
<termino>      ::= <funcionID>(<argumentos>) |  
                    <constanteID> | <variableID>  
<argumentos>  ::= <termino>, <argumentos> | <termino>  
<funcionID>    ::= <minuscula><identificador>  
<constanteID>  ::= <minuscula><identificador>  
<variableID>   ::= <mayuscula><identificador>
```

generan un el lenguaje de términos. Ejemplos de términos generados por esta gramática son:

Constantes	a, cons_1.
Variables	X2, Y, Param_1.
Términos	par_com(X, b), f(a, g(X, Y)), funl(cons1, fun2(cons2, Var2), c).

Suponiendo que se suministra el término a analizar como un string (esto es, una lista de códigos ASCII), el siguiente programa Prolog es un (fragmento de un) analizador sintáctico del lenguaje de términos generado por la gramática anterior.

```
argumentos(Args) :- termino(Args), !.
argumentos(Args) :- append(AA, RestoArgumentos, Args),
                 append(Termino, [44], AA),
                 termino(Termino),
                 argumentos(RestoArgumentos).

termino(Term) :- constanteID(Term), !.
termino(Term) :- variableID(Term), !.
termino(Term) :- append(FFF, [41], Term),
                 append(FF, Argumentos, FFF),
                 append(FuncionID, [40], FF),
                 funcionID(FuncionID),
                 argumentos(Argumentos).
```

- a) Se desea modificar el analizador para que, además de analizar una expresión la transforme en otra en la que los términos, constantes, y variables se etiqueten anteponiendo los átomos TERM, CONS y VAR (respectivamente). Ejemplo: La expresión $f(a, g(x, y))$ debería, además de reconocerse como término, transformarse en la expresión:

TERM(f(CONS(a), TERM(g(VAR(x), VAR(y))))).

- b) Defina un predicado parse que tome una expresión (una secuencia de caracteres ASCII) y escriba la expresión etiquetada mencionada arriba en un flujo de salida distinto del estándar.

(Ayuda: Use el predicado write para escribir la expresión etiquetada en un flujo de salida asociado a un fichero denominado outputfile).

Ejercicio 7.20 a) Defina un predicado unifica(X, Y) que tenga éxito cuando las expresiones X e Y puedan ser unificadas. Esto es, queremos que se comporte como el predicado predefinido en Prolog para la unificación de expresiones. b) Amplie el programa anterior para que el predicado unifica verifique la ocurrencia de variables.

Ejercicio 7.21 Defina un predicado, `ficha(H)` que, actuando sobre un hecho almacenado en una base de datos, lo visualice en forma de ficha. Por ejemplo, aplicado a una entrada de una base de datos bibliográfica, el objetivo:

```
?- ficha(libro(tol85,  
           autor('Tolkien', 'J.R.R.' ),  
           titulo('El Se\~nor de los Anillos'),  
           editorial(minotauro),  
           prestado(jul101, fecha(28, noviembre, 2004)))).
```

se debería visualizar por pantalla como:

```
libro:  
    tol85,  
    autor:  
        Tolkien,  
        J.R.R.,  
    titulo:  
        El Se\~nor de los Anillos,  
    editorial:  
        minotauro,  
    prestado:  
        jul101,  
        fecha:  
            28,  
            noviembre,  
            2004
```

Ejercicio 7.22 (Variables globales) Como se ha comentado, en el lenguaje Prolog no existe la posibilidad de utilizar variables globales. Sin embargo, éstas pueden simularse definiendo un predicado `var_gobal(Variable, Valor)` que, una vez cargado en el espacio de trabajo, sea actualizado mediante la utilización de los predicados `assert` y `retract`. Defina un operador binario “`::=`” que se comporte como una instrucción de asignación con respecto a las variables globales “almacenadas” por el predicado `var_gobal`. Suponga que en la base de datos interna se almacena el predicado `var_gobal(contador, 0)`; entonces el efecto de ejecutar la llamada `contador ::= 2` debería ser equivalente al de ejecutar la siguiente secuencia de llamadas:

```
retract(var_gobal(contador, _)).  
asserta(var_gobal(contador, 2)).
```

La primera llamada elimina el hecho `var_gobal(contador, 0)` y la segunda llamada incorpora el hecho `var_gobal(contador, 2)` al espacio de trabajo, produciendo un efecto similar al de una asignación del valor 2 a la variable global `contador`.

Ejercicio 7.23 Dada la pequeña base de datos de países del Ejemplo 7.4, defina las siguientes relaciones:

1. Un predicado alta que permita ampliar la base de datos de países. Cuando alta sea ejecutado deberá aparecer por pantalla:

```
=====
===== ALTA DE PAISES =====
=====
Introducir pais: 'China'.
```

.

```
=====
Introducir pais:
```

Cuando se le suministre el término no, concluirá la introducción de países. El predicado alta debe controlar si un país ya existe en la base de datos antes de introducirlo. (Ayuda: utilice el predicado assert)

2. Un predicado baja que permita borrar países de la base de datos. Cuando baja sea ejecutado deberá aparecer por pantalla:

```
=====
===== BAJA DE PAISES =====
=====
Eliminar pais: 'China'.
```

.

```
=====
Eliminar pais:
```

Cuando se le suministre el término no, concluirá la sesión de actualización. El predicado baja debe controlar si un país existe en la base de datos para poder borrarlo. (Ayuda: utilice el predicado retract)

3. Un predicado print_paises que imprima por pantalla la lista de países contenidos en la base de datos (Ayuda: utilice el predicado fail para forzar la vuelta atrás).
4. Un predicado salvar_en(Fichero) que salva la base de datos en un fichero especificado.
5. Un predicado fin para finalizar la sesión de trabajo y sacar del sistema al usuario. Antes de finalizar se guardan los hechos en el fichero db_paises, si la base de datos se ha modificado (Ayuda: utilice el predicado salvar_en y halt).
6. Un predicado menu que muestre el siguiente menú de ayuda:

```
=====
===== MENU DE AYUDA =====
=====
alta ..... Alta de nuevos paises.
baja ..... Baja de paises.
print_paises ..... Imprime la lista de los paises.
fin ..... Acabar la sesion.
```

Este predicado también puede entenderse como el “procedimiento” principal de esta pequeña aplicación.

Ejercicio 7.24 Utilizando el programa del Apartado 7.5, resolver el mini-sudoku:

```
[1, X12, X13, 3,
X21, 3, X23, 1,
X31, 1, X33, X34,
3, X42, X43, 2 ]
```

Estudie el comportamiento de los predicados generar y comprobar y del propio programa mediante diferentes trazas.

Ejercicio 7.25 (Sudoku) Partiendo de la solución presentada en el Apartado 7.5, confeccione un programa que permita resolver verdaderos sudokus, para matrices con 9×9 casillas. Dos puntos que deberán tratarse al construir el programa son: i) Extender la representación de la matriz con información sobre los índices de fila y columna; ii) Definir un predicado que filtre, para cada casilla, las alternativas que no son susceptibles de conducir a una solución.

Ejercicio 7.26 (Problema de las ocho reinas) La solución suministrada en el Apartado 7.5.2 todavía puede optimizarse si tenemos en cuenta la siguiente consideración:

Cuando se sitúa una reina en un tablero, seleccionando una fila determinada, ninguna otra reina podrá emplazarse en esa misma fila. Así por ejemplo, si inicialmente una reina puede situarse en cualquiera de las filas [1, 2, 3, ..., 8] y elegimos posicionarla en la fila 2 entonces, la siguiente reina solamente podrá posicionarse en una de las filas [1, 3, ..., 8].

Por lo tanto, supone un sobreesfuerzo utilizar la llamada `member(Y, [1, 2, 3, ..., 8])` para generar las diferentes posiciones alternativas sobre las que puede situarse una reina, pues sabemos que cada vez que se sitúa una reina sobre el tablero el número de alternativas decrece.

a) Defina un predicado `selecciona(A, Alternativas, Resto)` que seleccione una alternativa `A` de una lista de `Alternativas` y deje las restantes alternativas en la lista `Resto`. Por ejemplo, el objetivo `?- selecciona(3, [2, 3, 4], Resto)` enlaza

la variable Resto con la lista [2, 4]. Queremos que el predicado selecciona tenga una versatilidad similar al predicado member, de modo que una llamada a selecciona (A, [1, 2, 3, ..., 8], Resto) sea capaz de generar todas las alternativas útiles para nuestro problema.

b) Utilice el predicado selecciona para optimizar la solución al problema de las ocho reinas propuesta en el Apartado 7.5.2.

Ejercicio 7.27 (Cripto-aritmética) En el problema cripto-aritmético SEND + MORE = MONEY, cada letra representa uno y un solo dígito del cero al nueve (es decir, a cada letra se le asigna el mismo dígito y, a su vez, no puede asignarse el mismo dígito a letras diferentes) y a las letras iniciales S y M se les debe asignar un dígito mayor que cero. Con estas restricciones, el problema consiste en encontrar una asignación de dígitos para las letras capaz de satisfacer la anterior ecuación. Utilice el predicado selecciona, definido en el Ejercicio 7.26, para resolver este problema cripto-aritmético mediante la técnica de generar y comprobar.

Ejercicio 7.28 El siguiente predicado para descomponer un número entero en sus sumandos puede optimizarse convirtiendo en determinista el cálculo del segundo sumando, una vez fijado el primero. Hágalo.

```
descomponer(N,S1,S2) :- entre(0,N,S1),
entre(0,N,S2),
N ==:= S1+S2.
```

Ejercicio 7.29 Optimice el siguiente programa. Evalúe la ganancia en eficiencia lograda.

```
sort(Xs,Ys) :- append(As,[X,Y|Bs],Xs),
X>Y,
append(As,[Y,X|Bs],Xs1),
sort(Xs1,Ys).

sort(Xs,Xs) :- ordenada(Xs).

ordenada([]).
ordenada([X,Y|Ys]) :- X<=Y, ordenada([Y|Ys]).
```

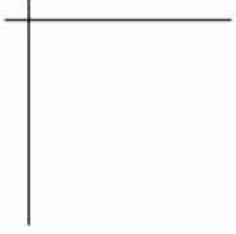
Cuestión 7.30 Conteste a la siguiente cuestión: ¿Es posible invertir el predicado de concatenación de listas diferencia append_ para partir una lista en dos? Razone la respuesta.

Ejercicio 7.31 Los siguientes predicados (el primero para duplicar una lista y el segundo para encontrar los prefijos comunes de dos listas), no funcionan cuando se utilizan con listas diferencia.

```
twice(L, LL) :- append_(L, L, LL).

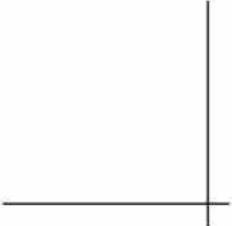
common_prefix(P, L1, L2) :- append_(P, _, L1), append_(P, _, L2).
```

En el primer caso esto se debe a que, en ausencia del occur-check, Prolog tiene éxito en el intento de concatenar una lista diferencia consigo misma, produciendo así un término infinito. Razone cuál es el problema que se presenta en el segundo predicado.



Parte III

Aplicaciones de la Programación Lógica



Representación del conocimiento

El objetivo de este capítulo es presentar una de las aplicaciones de la programación lógica: la representación del conocimiento. Se ha elegido esta aplicación porque a través de ella podemos introducir nuevas técnicas de programación. Ciertamente, en los capítulos previos, ya hemos utilizado técnicas de representación del conocimiento, sin embargo, en este capítulo abordamos su estudio de manera sistemática.

Antes de pasar al estudio detallado del papel de los lenguajes de programación lógica como herramientas de representación del conocimiento conviene iniciar la discusión con algunos de los problemas teóricos relacionados con la representación del conocimiento.

8.1 EL PROBLEMA DE LA REPRESENTACIÓN DEL CONOCIMIENTO

La representación del conocimiento ha sido tema de estudio desde los primeros tiempos en el área de la inteligencia artificial. Desde un punto de vista práctico, la *Inteligencia Artificial* (IA) es una parte de la informática que investiga cómo construir programas que realicen tareas inteligentes, entendiendo por tales aquellas acciones o comportamientos que asociamos con la inteligencia en los seres humanos (comprensión del lenguaje natural, resolución de problemas, razonamiento, aprendizaje, etc.). Para que un programa pueda simular un comportamiento inteligente es preciso que almacene “conocimiento” sobre una determinada área de aplicación.

Si nos preguntamos sobre qué es *conocimiento* y seguimos con nuestro enfoque pragmático, evitando discutir el sentido filosófico del término, podríamos decir que es el cúmulo de informaciones referentes a un área que permiten a un agente resolver problemas (no triviales) relativos a ese área. Un sistema que resuelva un problema en un determinado dominio diremos que tiene conocimiento en ese dominio. Principalmente estamos interesados en las formas de representar el conocimiento (de un agente humano) en un

computador. En general, una representación del conocimiento es una descripción formal que puede emplearse para la computación simbólica, independientemente de que esa descripción pueda traducirse, finalmente, en una combinación de estructuras de datos y procedimientos que las usan para producir un comportamiento inteligente.

Una característica en todas las representaciones del conocimiento es que manejamos dos tipos de entidades:

- i) los hechos, verdades de un cierto mundo que queremos representar;
- ii) la representación de los hechos en un determinado formalismo.

Estas entidades inducen la necesidad de distinguir entre dos niveles [103]:

- i) el *nivel del conocimiento*, donde se describen los hechos y el comportamiento;
- ii) el *nivel simbólico*, donde se describen los objetos del nivel del conocimiento mediante símbolos manipulables por programas.

Siguiendo la orientación de [126] nos centraremos en los hechos, en las representaciones y en la correspondencia que debe existir entre ellos. Ahora el razonamiento real es simulado por el funcionamiento de los programas, que manipulan las representaciones internas de los hechos. Se parte de los hechos iniciales, se representan en un determinado formalismo y el programa modifica dicha representación interna de los hechos iniciales hasta alcanzar una representación interna de los hechos finales. Así pues, son necesarias *funciones de correspondencia* (Figura 8.1) para, dado un hecho inicial, obtener una representación interna y, dada una representación interna final, establecer el hecho final correspondiente. Las funciones de correspondencia no suelen ser biunívocas, lo que



Figura 8.1 Razonamiento y funciones de correspondencia.

puede constituir un problema (e.g., un hecho podría tener más de una representación interna, y viceversa, lo que podría dificultar o impedir la obtención de una respuesta que se correspondiese con la respuesta real al problema).

Algunos autores [104] consideran que el conocimiento que necesita un sistema de IA puede dividirse en: i) *conocimiento declarativo*, que tiene que ver con los hechos y relaciones conocidas sobre un problema dado; ii) *conocimiento procedimental*, que se representa en forma de reglas que sirven para manipular el conocimiento declarativo y iii) *conocimiento de control*, que incluye el conocimiento acerca de diversos procesos y estrategias que se utilizan para guiar o limitar la aplicación de las reglas que establecen el conocimiento procedimental. El conocimiento de control suele basarse en algún tipo de *heurística*, i.e., conocimiento extraído de la experiencia en la resolución de problemas dentro del dominio que se desea representar. Esta distinción es aplicable tanto al nivel del conocimiento como a nivel simbólico. La primera característica tiene que ver con los hechos y las dos últimas con el comportamiento.

Un buen sistema de representación del conocimiento, en un dominio particular, debe poseer las siguientes propiedades:

1. Suficiencia de la representación: la capacidad de representar todos los tipos de conocimiento necesarios en el dominio. No basta con que el formalismo sirva para representar informaciones concretas, tal y como suele hacerse cuando se almacena información en una base de datos (e.g., “un león es un carnívoro”) sino, también, debe ser capaz de dar cuenta de conocimientos genéricos que requieren cierto grado de cuantificación en el que las variables toman valores en uno o varios dominios de discurso (e.g., “un animal que come otros animales es un carnívoro”). Ciertamente el conocimiento que expresa el último enunciado sería difícil de caracterizar si eligiésemos un lenguaje de programación convencional como vehículo para la representación del conocimiento.
2. Suficiencia deductiva: la capacidad de manipular las representaciones internas de los hechos con el fin de obtener otros nuevos.
3. Eficiencia deductiva: la capacidad de incorporar información (de control) adicional en las estructuras de conocimiento con el fin de guiar el proceso deductivo en la dirección más adecuada. Esta información se corresponde con el conocimiento de control. Habitualmente, para realizar estas tareas se requiere incorporar un conocimiento experto sobre el área tratada que permita distinguir, de entre todos los conocimientos almacenados, aquéllos que son relevantes a la hora de resolver un problema, impidiendo los procesos deductivos que (posiblemente) no conducirán a una solución del problema.
4. Eficiencia en la adquisición: la capacidad de adquirir nueva información con facilidad. Esta capacidad requiere de una sencillez en la notación empleada por el lenguaje de representación, para facilitar la introducción manual de nuevos conocimientos por parte del usuario del sistema de IA (o automática por parte del propio sistema de IA). Las expresiones resultantes deben ser fáciles de escribir y de leer, siendo posible entender su significado sin la necesidad de conocer como

las ejecutará el computador. Esto es, para conseguir esta capacidad es conveniente una representación declarativa.

No existe una técnica de representación del conocimiento que cumpla todos estos objetivos. Si bien se han ensayado diversas técnicas, unas son mejores que otras según para qué cosas. Los lenguajes para la representación deben poseer en mayor o menor grado algunas de estas propiedades. Dado que los lenguajes declarativos disponen de una sintaxis formal y una semántica clara, así como capacidades deductivas, están muy bien dotados como lenguajes de representación (simbólica) y pueden utilizarse para implementar las distintas técnicas de representación del conocimiento.

En lo que sigue estudiamos cómo podemos emplear la lógica de predicados como vehículo para la representación del conocimiento. Haremos énfasis en cómo podemos extraer, a partir de una descripción de un problema en lenguaje natural, una representación en la notación clausal de la lógica de predicados, que es directamente traducible a una implementación del problema en el lenguaje Prolog.

8.2 REPRESENTACIÓN MEDIANTE LA LÓGICA DE PREDICADOS

Utilizar la lógica de predicados es la forma clásica de aproximarse a la representación del conocimiento, que consiste en transformar el conocimiento sobre un dominio concreto en fórmulas bien formadas. Algunas consideraciones que la hacen atractiva como herramienta para la representación del conocimiento son:

1. Es una forma natural de expresar relaciones entre los elementos de un dominio de discurso.
2. Es precisa, ya que existen métodos formales para determinar el significado de una expresión y la validez de una fórmula de un formalismo lógico: la noción de interpretación y los métodos de la teoría de modelos.
3. Posee diferentes cálculos deductivos que permiten la construcción de derivaciones, y existen resultados de corrección y completitud de los cálculos deductivos que ligan la sintaxis con la semántica. La capacidad de realizar deducciones es el punto fuerte de la lógica de predicados como formalismo para la representación del conocimiento.
4. Puede automatizarse el proceso deductivo de una forma flexible, lo que facilita el uso de la lógica de predicados como lenguaje de programación.
5. Contrariamente a lo que sucede con las lógicas no monótonas, es modular. Esto es, puede aumentarse la teoría (base de conocimiento) sin afectar la validez de los resultados anteriormente deducidos, si bien esto puede convertirse en una desventaja cuando es preciso trabajar con conocimiento revisable o imperfecto.

La mayoría de las veces, el conocimiento sobre un dominio está expresado en un lenguaje natural como el castellano, el inglés, etc. Por consiguiente el problema de representar el conocimiento utilizando el lenguaje de la lógica de predicados se transforma en un problema de traducción. En principio podemos considerar que la traducción del lenguaje natural al lenguaje formal es una tarea inversa a la de interpretar. Dado que los enunciados del lenguaje formal se hacen verdaderos o falsos dependiendo de la interpretación empleada, es claro que para traducir un enunciado particular del lenguaje natural (que tiene una interpretación estándar) hay que partir de esa interpretación. Para ello se identifica el dominio de discurso, en el que tomarán valores las variables, y se establece una correspondencia entre las funciones y las relaciones y los símbolos que las representarán. Después se sustituye cada parte del enunciado del lenguaje natural por el símbolo que lo representa. Sin embargo, la tarea de hallar una traducción del lenguaje natural al lenguaje formal es, en buena medida, un arte que necesita de mucha práctica y para el cual no hay reglas precisas. El lenguaje natural, a diferencia del lenguaje formal, no tiene una gramática simple y regular. Además, el significado de sus expresiones depende del contexto y encierra una gran cantidad de ambigüedades, mientras que, sea cual sea la circunstancia, toda expresión del lenguaje formal denota (una vez fijada una interpretación) la misma cosa y las fórmulas (cerradas) el mismo valor de verdad. Consiguientemente, no existe una función de correspondencia que, aplicada a un enunciado del lenguaje natural, conduzca a una única fórmula del lenguaje formal i.e. para un enunciado del lenguaje natural pueden haber varias representaciones. Dadas las dificultades a las que nos enfrentamos, en la práctica, una buena táctica (la única posible en muchos casos) es preguntarse qué significa el enunciado del lenguaje natural y buscar una fórmula que tenga un significado, con respecto a la interpretación estándar, tan próximo al del enunciado del lenguaje natural como sea posible.

En lo que sigue realizamos algunas aclaraciones y damos algunos consejos prácticos que permiten convertir una sentencia escrita en castellano en una fórmula de la lógica de predicados. Para un estudio más profundo y sistemático del problema de la traducción del lenguaje natural a un lenguaje formal recomendamos la lectura de [75, Apartado 5.4.2] y del capítulo 5 del libro de B. Mates [94].

8.2.1. Nombres, functores y relatores

En las oraciones gramaticales del lenguaje ordinario se distinguen, principalmente, dos partes: el sujeto y el predicado. El sujeto suele referirse a objetos o individuos acerca de los cuales se está afirmando algo, mientras el predicado se refiere a una propiedad que posee el sujeto.

- **Nombres y variables:** El sujeto de una oración está constituido por lo que llamaremos un *designador*: una o varias palabras que hacen referencia a objetos o individuos. Hay muchas clases de designadores, siendo los *nombres* los más usuales. Los nombres son sucesiones de signos o símbolos que designan algo (un número, una ciudad, una persona, ...). Ejemplos de nombres son: "dos", "Enrique",

“Madrid”. En el lenguaje formal de la lógica, en lugar de nombres del lenguaje ordinario, suelen emplearse las primeras letras minúsculas del alfabeto: a , b y c (si es necesario con subíndices: a_0, a_1, a_2, \dots). Estos símbolos se denominan, en el lenguaje formal, *constantes*. Nosotros hemos relajado esta convención mediante el uso de identificadores que escribiremos siguiendo las convenciones del lenguaje Prolog.

Es frecuente que los matemáticos utilicen variables, sobre todo cuando quieren decir algo bastante general, como que la ecuación

$$X + Y = Y + X$$

se satisface cualesquiera que sean los números reales que pongamos en el lugar de las variables. En el lenguaje ordinario, los pronombres juegan el papel de las variables en las fórmulas matemáticas. Cuando decimos “Él ha sido el asesino” el lector no puede saber a quién se refiere ese “él” porque desconoce el contexto. Es tanto como decir “ X ha sido el asesino”. Igualmente, cuando decimos “Yo he ido al cine”, el pronombre “yo” puede estar designando a Pascual, a Germán o a cualquier otra persona, porque depende del contexto; así pues es tanto como si dijésemos “ X ha ido al cine”. También la conjunción de un artículo y un nombre común puede sustituirse, en ocasiones, por una variable, así por ejemplo en lugar de “el niño corre” podríamos escribir “ X corre”. Las variables no designan a ningún objeto o individuo en particular.

Observación 8.1

El hecho de que, al representar los pronombres por variables, las variables no designen a ningún objeto o individuo en particular puede hacer que se pierda información. Es una tarea de quien está realizando la representación interna decidir si el pronombre personal debe ser sustituido por una variable o por una constante que designe una persona concreta (e.g., “pascual” en vez de “ X ”, si es Pascual quien dijo “Yo he ido al cine”).

- **Functores.** Los nombres son designadores simples, en el sentido de que ninguna parte propia de ellos es a su vez un nombre. Pero no todos los designadores son así. Por ejemplo: “El río que atraviesa la capital de España” es un designador compuesto que se refiere a un objeto: el río Manzanares. “Manzanares” es su nombre, pero no la única expresión que lo designa. “La capital de España” es, a su vez, un designador que también es compuesto; designa la ciudad de Madrid.

Hay expresiones que, seguidas de un número determinado de designadores, forman a su vez un designador. Estas expresiones se llaman *functores*. Por ejemplo: El designador “El obispo de ...” seguido del nombre “Roma” designa a Benedicto XVI. Representaremos los functores mediante símbolos de función y los designadores compuestos mediante términos del lenguaje formal. Como se ha hecho con

los símbolos de constante, la mayoría de las veces relajaremos las convenciones del lenguaje formal permitiendo el uso de identificadores, que utilizaremos como símbolos de función.

Si simbolizamos “El obispo de ...” por “ g ”, podremos escribir “ $g(X)$ ”. Advierta que, cuando se interpreta, “ $g(X)$ ” no designa ningún obispo en particular. En general, cuando se interprete un término que contenga variables, no designará a ningún objeto o individuo; es decir, no será un designador propiamente dicho. El resultado es lo que llamamos un *término abierto*.

- **Relatores.** Hay expresiones lingüísticas que, junto a un número determinado de designadores forman un enunciado. Llamaremos relatores a dichas expresiones. Los relatores designan relaciones entre los individuos del dominio de discurso y se representan mediante símbolos de relación. Así ocurre, por ejemplo, con el relator “... es más alto que ...” que, unido a los nombres “Juan” y “Pascual”, da lugar al enunciado “Juan es más alto que Pascual”. En general, en la lógica de predicados, los relatores se representan mediante símbolos de relación y las relaciones entre individuos mediante el empleo de fórmulas atómicas. Nuevamente, permitiremos el uso de identificadores como símbolos de relación.

Habitualmente, por convención, el símbolo de relación se emplea en forma prefija y el orden de los argumentos consiste en tomar como primer argumento al sujeto de la oración. Así pues, el enunciado anterior podría representarse mediante la fórmula atómica: *masAlto(juan, pascual)*. En este contexto, los relatores monarios tradicionalmente se denominan *predicados* (y coinciden con la parte de la oración gramatical de ese nombre).

Observe que los predicados pueden usarse para definir, por extensión, conjuntos (clases) de objetos o individuos, cuando empleamos variables junto con dichos predicados. Por ejemplo: si representamos el predicado “... es un número par” mediante el predicado $par(X)$, el conjunto de los números pares $\{2, 4, 6, \dots\}$ puede definirse como $\{X \mid par(X)\}$.

8.2.2. Conjunciones y conectivas

Los enunciados formados por relatores y designadores se denominan enunciados atómicos ya que son las partes más simples de una oración con significado completo, y pueden combinarse con otros enunciados (posiblemente atómicos) para formar *enunciados compuestos* o *moleculares*. En el lenguaje natural empleamos ciertas partículas que sirven de nexo de unión entre diferentes partes de una oración. Estas partículas se denominan conjunciones y se representan en el lenguaje de la lógica de predicados mediante los símbolos que hemos denominado conectivas. A continuación se presentan las principales equivalencias entre conjunciones y conectivas, que pueden emplearse para formalizar enunciados moleculares:

1. Negación (\neg). La fórmula $\neg A$ permite simbolizar un enunciado, a partir de otro cualquiera A , del tipo: *no A; ni A; no es cierto que A; es falso que A*.
2. Conjunction (\wedge). La fórmula $A \wedge B$, simboliza enunciados del lenguaje natural de la forma: *A y B; A y también B; A e igualmente B; tanto A como B; A pero B; A no obstante B; A sin embargo B*. Habitualmente la partícula “*ni*” se presenta en términos de una conjunción de negaciones, como en el enunciado “*ni A ni B*” que puede representarse mediante la fórmula “ $\neg A \wedge \neg B$ ”.
3. Disyunción (\vee). La fórmula $A \vee B$ simboliza conexiones de la forma: *A o B; al menos A o B*. La disyunción, tal y como se emplea en la lógica, tiene el mismo significado que la partícula “o” del lenguaje ordinario en su sentido *inclusivo*: La disyunción $A \vee B$ solamente es falsa cuando tanto el enunciado A como el B son falsos, y es verdadera en el resto de los casos. Sin embargo, el disyuntor no refleja el segundo de los sentidos ordinarios de la partícula “o”. Cuando decimos “o blanco o negro” estamos indicando que debe darse una cosa o la otra, pero no ambas a la vez. Este es el sentido del llamado “o” *exclusivo*. Podemos expresar la disyunción exclusiva en términos de la negación, la conjunción y la disyunción de la forma siguiente: $(A \vee B) \wedge \neg(A \wedge B)$ que traduce el sentido literal de la disyunción exclusiva, es decir, “*p o q, pero no ambos*”.
4. Condicional (\rightarrow). La fórmula $A \rightarrow B$ simboliza enunciados de la forma: *si A entonces B; si A, B; A implica B; A solo si B; A es suficiente para B; B si A; B es necesario para A; no A a menos que B; B cuandoquiera que A; B siempre que A*.
5. Bicondicional (\leftrightarrow). $A \leftrightarrow B$ denota enunciados de la forma: *A si y solo si B; A necesario y suficiente para B*.

Observación 8.2 *Puesto que las conectivas se emplean en lógica para determinar el valor de verdad de los enunciados compuestos, no nos interesan de ellos las connotaciones de otro orden que puedieran tener. Aunque la conjunción “pero” se emplea cuando hay cierta oposición entre los dos enunciados que une, su contribución al valor de verdad del enunciado compuesto es el mismo que el de la conjunción “y”. Así que el enunciado “Juan es inteligente pero María es más astuta” es verdadero en la medida que lo sea el enunciado “Juan es inteligente y María es más astuta”. Obligados por la necesidad de crear un lenguaje formal simple y preciso, simbolizamos varias conjunciones del lenguaje natural mediante una única conectiva. Si bien puede alegarse una perdida del sentido original del enunciado al adoptar esta posición reduccionista, la diferencia entre ambos enunciados es retórica, no lógica.*

Ejemplo 8.1

Consideremos la interpretación \mathcal{I} en la que el dominio de discurso son los números naturales, e.i., $\mathcal{D}_{\mathcal{I}} = \mathbb{N}$, y la función de interpretación \mathcal{J} asigna:

$par(X) :$	X es par;	$X = Y :$	X es idéntico a Y ,
$impar(X) :$	X es impar;	$X < Y :$	X es menor que Y ;
$primo(X) :$	X es primo;	$X > Y :$	X es mayor que Y ;
$suma(X, Y, S) :$	$X + Y = S$;		
$mult(X, Y, M) :$	$X \times Y = M$;		

y, por simplicidad, cada número natural es representado por sí mismo. Ahora podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

- “2 es par”: se traduce a la fórmula “ $par(2)$ ”.
- “2 no es impar”: se traduce a la fórmula “ $\neg impar(2)$ ”.
- “es un número impar y mayor que tres”: se traduce a “ $impar(X) \wedge X > 3$ ”.
- “no es el caso de que 5 sea el producto de 2 por 3”: se traduce a la fórmula “ $\neg mult(2, 3, 5)$ ”.
- “ser primo un número es suficiente para ser un número impar”: se traduce a la fórmula “ $primo(X) \rightarrow impar(X)$ ”.
- “2 o es par o es impar”: se traduce a la fórmula

$$(par(2) \vee impar(2)) \wedge \neg(par(2) \wedge impar(2)).$$

Ejemplo 8.2

Dada la interpretación \mathcal{I} en la que el dominio de discurso, $\mathcal{D}_{\mathcal{I}}$, son los seres humanos y la función de interpretación \mathcal{J} asigna:

$padre(X, Y) :$	X es el padre de Y ;	$m :$	María;
$madre(X, Y) :$	X es la madre de Y ;	$e :$	Enrique;
$marido(X, Y) :$	X es el marido de Y ;	$g :$	Guillermo;
$hermano(X, Y) :$	X es hermano de Y ;	$a :$	Arturo;
$hermana(X, Y) :$	X es hermana de Y ;		

Ahora podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

- “Enrique es padre”: se traduce a la fórmula “ $padre(e, X)$ ”.

- “María es abuela”: se traduce a la fórmula

$$\text{madre}(m, X) \wedge (\text{padre}(X, Y) \vee \text{madre}(X, Y)).$$

- “Guillermo es el cuñado de María”: se traduce a la fórmula

$$(\text{marido}(g, X) \wedge \text{hermana}(X, m)).$$

- “María es la tía de Arturo”: se traduce a la fórmula

$$[(\text{marido}(Z, m) \wedge \text{hermano}(Z, W)) \vee \text{hermana}(m, W)] \wedge \\ (\text{padre}(W, a) \vee \text{madre}(W, a)).$$

Observación 8.3

Tal vez, una traducción más precisa del enunciado “Enrique es padre”, en el primer caso del Ejemplo 8.2, debería utilizar el cuantificador existencial. Lo mismo puede decirse del resto de los casos (ver el Ejemplo 8.4).

8.2.3. Granularidad y representación canónica

La traducción de un enunciado depende del vocabulario introducido en la interpretación de la que se parte. Se podría haber ampliado la interpretación I del Ejemplo 8.2 con el símbolo de relación “ $\text{esPadre}(X)$ ” para representar el predicado “ X es padre”, de modo que la traducción del enunciado habría sido completamente inmediata: $\text{esPadre}(e)$. Este tipo de decisiones queda en manos del que ha de realizar la traducción y afecta a la granularidad de la representación. Entendemos por *granularidad* el conjunto de primitivas utilizadas para representar el conocimiento. Como vemos, no siempre está claro qué primitivas deben utilizarse en una representación. Vemos pues que una interpretación I proporciona un conjunto de primitivas con el que representar el conocimiento de una forma canónica.

El razonamiento anterior se aplica a todos los casos en los que no es posible una traducción directa.

8.2.4. Cuantificadores

El enunciado “Todos los hombres son mortales” corresponde a un tipo distinto de enunciado compuesto o molecular, de los estudiados hasta ahora. Necesitamos algo más que un análisis del sujeto y predicado de la oración, ya que el sentido del enunciado depende fuertemente de la palabra “Todos”, que nos sirve para afirmar algo sobre todos los individuos u objetos de una clase. También en las oraciones del lenguaje ordinario empleamos la palabra “Algunos”, como en “Algunos hombres son griegos”, que nos sirve para asignar una propiedad a unos cuantos individuos de una clase. En general las partículas que juegan en una oración el mismo papel que las palabras “todos” o “algunos”

se formalizan mediante el uso de los *cuantificadores*. El uso de cuantificadores lleva asociado la introducción de variables (cuantificadas) en las fórmulas que, cuando sean interpretadas, tomarán valores en el dominio de la interpretación.

Los enunciados del lenguaje natural que se basan en el uso de la partícula “todos” se representan empleando el *cuantificador universal* “ \forall ”. Otras partículas que conllevan el uso del cuantificador universal son: “cada”, “cualquier”, “cualquiera”, “todo”, “toda”, “nada”, “ningún”, “ninguno”, “ninguna”. Los enunciados del lenguaje natural que se basan en el empleo de “algunos” se representan empleando el *cuantificador existencial* “ \exists ”. Otras partículas que conllevan el uso del cuantificador existencial son: “existe”, “hay”, “algún”, “alguno”, “alguna”, “algunas”.

Observaciones 8.1

- Nótese que, en ocasiones, los artículos determinados “el”, “los”, etc. expresan cuantificación universal, como por ejemplo en los enunciados “el hombre es un mamífero” o “Los gatos persiguen a los ratones”. Por el contrario, los artículos indeterminados “un”, “unos”, etc. pueden expresar cuantificación existencial, como por ejemplo en “unos hombres son mejores que otros”.
- También las palabras “tiene” y “tienen” expresan existencia como, por ejemplo, en “Juan tiene un progenitor que le ama”.



A la hora de traducir enunciados del lenguaje natural al lenguaje formal de la lógica de predicados, es bueno tener en mente las equivalencias de la Tabla 8.1, en la que se muestra la traducción a nuestro lenguaje formal de los cuatro tipos de enunciados (categóricos) de los que se ocupa la Silogística Aristotélica¹. En los enunciados categóricos, las metavariables \mathcal{A} y \mathcal{B} representan predicados.

Tabla 8.1 Enunciados categóricos y su formalización.

Enunciado	Formalización
Todo \mathcal{A} es \mathcal{B}	$(\forall X)(\mathcal{A}(X) \rightarrow \mathcal{B}(X))$
Algún \mathcal{A} es \mathcal{B}	$(\exists X)(\mathcal{A}(X) \wedge \mathcal{B}(X))$
Ningún \mathcal{A} es \mathcal{B}	$(\forall X)(\mathcal{A}(X) \rightarrow \neg \mathcal{B}(X))$
Algún \mathcal{A} no es \mathcal{B}	$(\exists X)(\mathcal{A}(X) \wedge \neg \mathcal{B}(X))$

¹ Esta teoría lógica puede considerarse un precedente de la lógica de predicados (si bien no es completamente reducible a ella).

Ejemplo 8.3

Si consideramos nuevamente la interpretación \mathcal{I} del Ejemplo 8.1, podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

- “Existe un número natural mayor que (cualquier) otro dado”: se traduce a la fórmula “ $(\forall X)(\exists Y)(Y > X)$ ”.
- “0 es un número natural menor que (cualquier) otro”: se traduce a la fórmula “ $(\forall X)(\neg(X = 0) \rightarrow 0 < X)$ ”.
- “Si el producto de dos números es par, entonces al menos uno de ellos es par”: se traduce a la fórmula

$$(\forall X)(\forall Y)[(\exists Z)((\text{mult}(X, Y, Z) \wedge \text{par}(Z)) \rightarrow (\text{par}(X) \vee \text{par}(Y))],$$

o también

$$(\forall X)(\forall Y)(\forall Z)[(\text{mult}(X, Y, Z) \wedge \text{par}(Z)) \rightarrow (\text{par}(X) \vee \text{par}(Y))].$$

- “Todo entero par mayor que 4 es la suma de dos primos”: se traduce a la fórmula “ $(\forall Z)[(\text{par}(Z) \wedge Z > 4) \rightarrow (\exists X)(\exists Y)(\text{suma}(X, Y, Z) \wedge \text{primo}(X) \wedge \text{primo}(Y))]$ ”.
- “Todo entero mayor que 1 es divisible por algún primo”: se traduce a la fórmula “ $(\forall X)[X > 1 \rightarrow (\exists Y)(\exists Z)(\text{primo}(Y) \wedge \text{mult}(Y, Z, X))]$ ”.

Ejemplo 8.4

Empleando el cuantificador existencial podemos traducir con mayor riqueza o precisión los enunciados del Ejemplo 8.2 al lenguaje formal:

- “Enrique es padre”: se traduce a la fórmula “ $(\exists X)\text{padre}(e, X)$ ”.
- “María es abuela”: se traduce a la fórmula
 $(\exists X)(\exists Y)(\text{madre}(m, X) \wedge (\text{padre}(X, Y) \vee \text{madre}(X, Y))).$
- “Guillermo es el cuñado de María”: se traduce a la fórmula
 $(\exists X)(\text{marido}(g, X) \wedge \text{hermana}(X, m)).$
- “María es la tía de Arturo”: se traduce a la fórmula
 $(\exists Z)(\exists W)[[(\text{marido}(Z, m) \wedge \text{hermano}(Z, W)) \vee \text{hermana}(m, W)]$
 $\wedge (\text{padre}(W, a) \vee \text{madre}(W, a))].$

Ejemplo 8.5

Si consideramos nuevamente la interpretación \mathcal{I} del Ejemplo 8.2, podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

- “Todo el mundo tiene padre”: se traduce como “ $(\forall X)(\exists Y)(padre(Y, X))$ ”.
- “Ningún padre de alguien es madre de nadie”: se traduce a la fórmula
 $(\forall X)[(\exists Y)padre(X, Y) \rightarrow \neg((\exists Z)madre(X, Z))]$,
o bien, de forma equivalente,
 $(\forall X)(\forall Y)(\forall Z)[padre(X, Y) \rightarrow \neg madre(X, Z)]$.
- “Todos los abuelos son padres”: se traduce a la fórmula
 $(\forall X)[(\exists Y)(\exists Z)(padre(X, Y) \wedge padre(Y, Z)) \rightarrow (\exists Y)padre(X, Y)]$,
o bien, de forma equivalente,
 $(\forall X)(\forall Y)(\forall Z)(\exists W)[(padre(X, Y) \wedge padre(Y, Z)) \rightarrow padre(X, W)]$.

Observación 8.4

Recuérdese que las variables cuantificadas de una fórmula pueden renombrarse sistemáticamente sin alterar el sentido de la fórmula, simplemente reemplazando todas las ocurrencias de una variable cuantificada por otra variable que no haya ocurrido antes en ella.

8.2.5. Consideraciones y recomendaciones finales

Antes de concluir este apartado es interesante realizar algunas consideraciones y recomendaciones.

No siempre es posible encontrar una correspondencia simple entre los enunciados del lenguaje natural y las fórmulas lógicas. Algunas razones que justifican esta afirmación son:

1. Una conectiva o un cuantificador pueden aparecer en una formalización a pesar de que las partículas que habitualmente lo representan en el lenguaje natural no aparezcan. Por ejemplo, el condicional y el cuantificador universal no aparecen explícitamente en el enunciado “Los españoles son europeos” que, sin embargo, se formaliza como: $(\forall X)(español(X) \rightarrow europeo(X))$.
2. El lenguaje natural contiene conectivas que no son funciones veritativas o que son casos límitrofes (e.g., “porque” o “puesto que”).
3. A una fórmula puede corresponderle varios enunciados del lenguaje natural, según la interpretación elegida. Sin embargo, lo que resulta más grave es que, debido a la ambigüedad del lenguaje natural, a un enunciado castellano pueda corresponderle diferentes representaciones formales no equivalentes.
4. Cuando los enunciados a formalizar hacen referencia (implícita) a tiempos relacionados o expresan una relación causal surgen problemas adicionales a la hora de

su formalización. Por ejemplo, nadie duda en formalizar el enunciado “Si llueve entonces Pascual abre el paraguas” como

$$\text{llueve} \rightarrow \text{abre}(\text{pascual}, \text{paraguas}).$$

Esta formalización no hace justicia al enunciado original, ya que, una consecuencia lógica de esta fórmula es

$$\neg\text{abre}(\text{pascual}, \text{paraguas}) \rightarrow \neg\text{llueve}$$

Ahora bien, nadie puede pensar que el enunciado castellano “Si Pascual no abre el paraguas entonces no llueve” se siga del enunciado original. El problema es que el enunciado original expresa una relación de causa efecto en la que hay tiempos relacionados. El sentido del enunciado original es que si llueve en un tiempo t entonces Pascual abrirá el paraguas en un tiempo t' inmediatamente posterior. Por consiguiente, si la interpretación no permite introducir las debidas referencias temporales la traducción perderá parte del sentido original. Dado que la lógica de predicados clásica no contempla el uso del tiempo, es difícil expresar este tipo de relaciones de una forma adecuada y compacta haciendo uso de ella. Sería necesario recurrir a una lógica temporal para tratar satisfactoriamente los aspectos temporales del conocimiento.

Por sencillez, en muchas ocasiones no tendremos en cuenta esta consideración, realizando la traducción de la mejor manera posible con lo medios disponibles. Después, si las cosas no funcionan habrá que refinar la representación.

5. Es posible obtener enunciados formales que sean consecuencia lógica el uno del otro y que, pese a ello, no sean representaciones igualmente satisfactorias de un enunciado del lenguaje natural. Por ejemplo, el enunciado “3 es un número primo” se representa de forma adecuada como “ $\text{primo}(3)$ ”, pero no como “ $\neg\neg\text{primo}(3)$ ” aunque sean fórmulas lógicamente equivalentes.
6. Finalmente, en las tareas de formalización, se puede partir de una interpretación con un vocabulario que proporciona un conjunto de símbolos primitivos cuya grano de arena no es adecuada y dificulta la representación. Por ejemplo, el lector puede intentar formalizar el enunciado “Ningún tío es una tía”, partiendo de la interpretación del Ejemplo 8.2, para tomar plena conciencia de este problema.

Todas estas dificultades ponen de manifiesto que es prácticamente imposible dar reglas precisas y sistemáticas para la traducción de enunciados del lenguaje natural al lenguaje formalizado de la lógica de predicados. Las únicas recomendaciones posibles son: i) meditar detenidamente sobre el sentido del enunciado del lenguaje natural; ii) seleccionar una interpretación de partida \mathcal{I} que sea adecuada y facilite la posterior representación del enunciado; iii) derivar una fórmula que, cuando se interprete, tenga un significado lo

más próximo posible al enunciado original. Por consiguiente, como ilustra la Figura 8.2 para un ejemplo concreto, el proceso de traducción es un proceso de “ida y vuelta” en el que, una vez realizada la formalización de un enunciado (apoyándose en una interpretación estándar \mathcal{I}), tenemos que comprobar que el significado de la fórmula obtenida (cuando se interpreta mediante la interpretación \mathcal{I}) se ajusta al enunciado original.

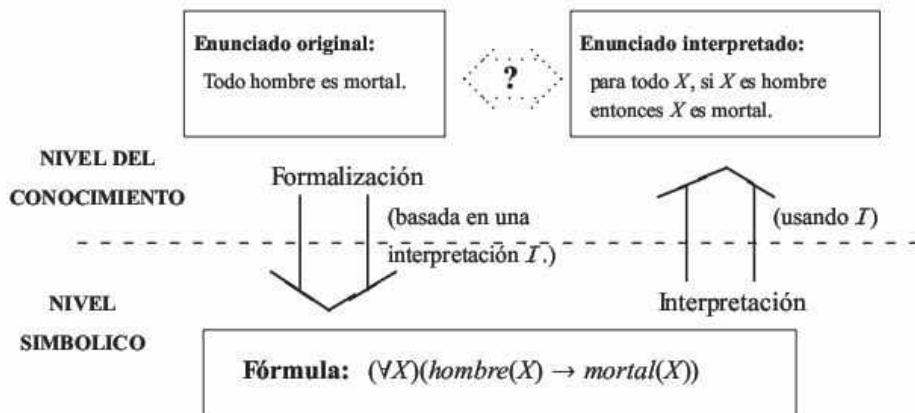


Figura 8.2 Traducción de un enunciado al lenguaje formal de la lógica.

8.3 REPRESENTACIÓN EN FORMA CLAUSAL

Dado el papel tan importante que juega la forma clausal en la programación lógica, vamos a estudiar algunas peculiaridades de la representación del conocimiento mediante cláusulas.

Para traducir cualquier enunciado a la forma clausal el procedimiento general consiste en:

1. traducir el enunciado a una fórmula de la lógica de predicados, siguiendo las recomendaciones del apartado anterior;
2. aplicar el Algoritmo 3 para obtener una forma normal de Skolem;
3. extraer las cláusulas de la forma normal de Skolem y representarlas en notación clausal.

Debido al paso (1), el proceso completo no puede formalizarse como un algoritmo. A continuación ilustramos el proceso completo mediante el siguiente ejemplo adaptado de [126].

Ejemplo 8.6

Consideremos el conjunto de enunciados castellanos

1. Marco Bruto era un hombre.
2. Marco Bruto era partidario de Pompeyo.
3. Todos los partidarios de Pompeyo eran romanos.
4. Julio César fue un gobernante.
5. Todos los romanos que eran enemigos de Julio César, le odiaban.
6. Todo el mundo es enemigo de alguien.
7. Los romanos solo intentan asesinar a los gobernantes de los que son enemigos.
8. Marco Bruto asesinó a Julio César.

Vamos a representarlos en forma clausal; para ello seguimos el procedimiento general indicado más arriba:

Paso 1

Seleccionamos la interpretación de partida \mathcal{I} , en la que el dominio de discurso, $\mathcal{D}_{\mathcal{I}}$, son los seres humanos y la función de interpretación \mathcal{J} asigna:

$hombre(X)$:	X es un hombre;	$bruto$:	Marco Bruto;
$romano(X)$:	X es un romano;	$pompeyo$:	Pompeyo;
$gobernante(X)$:	X es un gobernante;	$cesar$:	Julio César;
$partidario(X, Y)$:	X es partidario de Y ;		
$enemigo(X, Y)$:	X es enemigo de Y ;		
$odia(X, Y)$:	X odia a Y ;		
$asesina(X, Y)$:	X asesina a Y ;		

Ahora podemos traducir los siguientes enunciados del lenguaje natural al lenguaje formal:

1. “Marco Bruto era un hombre” se traduce a la fórmula “ $hombre(bruto)$ ”.
2. “Marco Bruto era partidario de Pompeyo”: se traduce como “ $partidario(bruto, pompeyo)$ ”.
3. “Todos los partidarios de Pompeyo eran romanos”: se traduce a la fórmula “ $(\forall X)(partidario(X, pompeyo) \rightarrow romano(X))$ ”.
4. “Julio César fue un gobernante”: se traduce como “ $gobernante(cesar)$ ”.

Nótese que en la formalización de estos enunciados castellanos estamos perdiendo parte del significado, ya que no se puede describir el concepto de tiempo pasado.

5. “Todos los romanos que eran enemigos de Julio César, le odiaban.”: se traduce a la fórmula “ $(\forall X)(romano(X) \wedge enemigo(X, cesar) \rightarrow odia(X, cesar))$ ”.

Nótese que el pronombre relativo “que” (y de igual modo “quien” y “donde”) se refiere a individuos ya mencionados en el mismo enunciado.

6. “Todo el mundo es enemigo de alguien”: se traduce como “ $(\forall X)(\exists Y)enemigo(X, Y)$ ”.

Si el enunciado se hubiese interpretado “hay alguien de quien todo el mundo es enemigo” entonces se hubiese formalizado como: “ $(\exists Y)(\forall X)enemigo(X, Y)$ ”.

7. “Los romanos solo intentan asesinar a los gobernantes de los que son enemigos”: se traduce como “ $(\forall X)(\forall Y)(romano(X) \wedge gobernante(Y) \wedge asesina(X, Y) \rightarrow enemigo(X, Y))$ ”.

Si el enunciado se hubiese interpretado en el sentido de que “lo único que los romanos intentan hacer es asesinar a los gobernantes de los que son enemigos” se hubiese formalizado como:

$$(\forall X)(\forall Y)(romano(X) \wedge gobernante(Y) \wedge enemigo(X, Y) \rightarrow asesina(X, Y)).$$

Otro problema a la hora de formalizar es que la elección de este vocabulario, más concretamente la relación *asesina*(*X, Y*) no permite distinguir entre el asesinato y el intento de asesinato. Tampoco permite establecer conexiones entre intentar un asesinato o intentar cualquier otra cosa.

8. “Marco Bruto asesinó a Julio César”: se traduce a la fórmula “*asesina(bruto, cesar)*”.

Paso 2

Una vez que hemos expresado los enunciados castellanos como fórmulas de la lógica de predicados, aplicamos el algoritmo de transformación a forma normal:

1. *hombre(bruto).*

2. *partidario(bruto, pompeyo).*

3. $(\forall X)(partidario(X, pompeyo) \rightarrow romano(X))$

$$\Leftrightarrow (\forall X)(\neg partidario(X, pompeyo) \vee romano(X))$$

\Rightarrow (paso a forma clausal)

$\neg partidario(X, pompeyo) \vee romano(X).$

4. *gobernante(cesar).*

5. $(\forall X)(romano(X) \wedge enemigo(X, cesar) \rightarrow odia(X, cesar))$

$$\begin{aligned} &\Leftrightarrow (\forall X) \neg(romano(X) \wedge enemigo(X, cesar) \vee odia(X, cesar)) \\ &\Leftrightarrow (\forall X) (\neg romano(X) \vee \neg enemigo(X, cesar) \vee odia(X, cesar)) \\ &\Rightarrow (\text{paso a forma clausal}) \end{aligned}$$

$\neg romano(X) \vee \neg enemigo(X, cesar) \vee odia(X, cesar).$

6. $(\forall X)(\exists Y)enemigo(X, Y)$

\Rightarrow (skolemización)

$$(\forall X)enemigo(X, elEnemigoDe(X))$$

\Rightarrow (paso a forma clausal)

$enemigo(X, elEnemigoDe(X)).$

7. $(\forall X)(\forall Y)(romano(X) \wedge gobernante(Y) \wedge asesina(X, Y) \rightarrow enemigo(X, Y))$

$$\begin{aligned} &\Leftrightarrow (\forall X)(\forall Y) (\neg(romano(X) \wedge gobernante(Y) \wedge asesina(X, Y)) \\ &\quad \vee enemigo(X, Y)) \end{aligned}$$

$$\begin{aligned} &\Leftrightarrow (\forall X)(\forall Y) (\neg romano(X) \vee \neg gobernante(Y) \vee \neg asesina(X, Y) \\ &\quad \vee enemigo(X, Y)) \end{aligned}$$

\Rightarrow (paso a forma clausal)

$\neg romano(X) \vee \neg gobernante(Y) \vee \neg asesina(X, Y) \vee enemigo(X, Y).$

8. $asesina(bruto, cesar).$

Paso 3

Finalmente, el conjunto de cláusulas expresadas en notación clausal es:

1. $hombre(bruto) \leftarrow$
2. $partidario(bruto, pompeyo) \leftarrow$
3. $romano(X) \leftarrow partidario(X, pompeyo)$
4. $gobernante(cesar) \leftarrow$
5. $odia(X, cesar) \leftarrow romano(X) \wedge enemigo(X, cesar)$
6. $enemigo(X, elEnemigoDe(X)) \leftarrow$
7. $enemigo(X, Y) \leftarrow romano(X) \wedge gobernante(Y) \wedge asesina(X, Y)$
8. $asesina(bruto, cesar) \leftarrow$

Esta representación se puede implementar directamente en Prolog y nos permite preguntar, por ejemplo, quién es enemigo de César.

Observaciones 8.2

1. Este ejemplo ilustra nuevamente que muchos enunciados del lenguaje natural son ambiguos (e.g., el 6 y el 7) y que, normalmente, hay más de una forma de representar el conocimiento. No obstante, las representaciones elegidas en primera opción son las más adecuadas para este ejemplo.

2. La interpretación de partida elegida dependerá del uso que se desee hacer. En este ejemplo, la elección podría haber sido distinguir entre “intenta asesinar” y “asesinar”. Si hubiésemos procedido de esa manera el séptimo enunciado se habría formalizado así:

$$\text{enemigo}(X, Y) \leftarrow \text{romano}(X) \wedge \text{gobernante}(Y) \wedge \text{intenta_asesinar}(X, Y).$$

Pero ahora es imposible probar que Bruto es enemigo de César. La razón es que falta conocimiento sobre el problema (en la terminología del lenguaje Prolog, se dice que la relación “intenta_asesinar” no está definida). Debe añadirse un nuevo enunciado que establezca que, para asesinar a alguien, es condición necesaria haber intentado asesinarlo. La representación de este enunciado en notación clausal es:

$$\text{intenta_asesinar}(X, Y) \leftarrow \text{asesina}(X, Y).$$

En general, no es probable que en un conjunto de enunciados esté todo el conocimiento necesario para razonar sobre el tema.



Si analizamos los pasos dados en la traducción de los enunciados del Ejemplo 8.6, podemos extraer algunas enseñanzas prácticas y “atajos” para su representación en notación clausal:

1. Los verbos de los enunciados (e.g., “ser enemigo”, “asesinar”, “odiár”, etc.) se traducen como predicados y relaciones (esto es, “ $\text{enemigo}(X, Y)$ ”, “ $\text{asesina}(X, Y)$ ”, “ $\text{odia}(X, Y)$ ”, etc.). Observe además que, el verbo principal de un enunciado suele emplazarse como conclusión y (i.e., la cabeza de una cláusula) y el resto de los predicados en la condición (i.e., el cuerpo de la cláusula). Por ejemplo, el enunciado “Todos los romanos que eran enemigos de Julio Cesar, le odiaban”, se formaliza mediante cláusula:

$$\text{odia}(X, cesar) \leftarrow \text{romano}(X) \wedge \text{enemigo}(X, cesar).$$

2. Sin embargo, el verbo “ser”, en sus diferentes conjugaciones, normalmente no se traduce en un predicado. Es el nombre o adjetivo que le sigue el que se convierte en el predicado principal. Por ejemplo, el enunciado “Marco Bruto era partidario de Pompeyo.”, se traduce en “ $\text{partidario(bruto, pompeyo)}$ ”, ya que el verbo “ser” está predicando la condición de partidario de alguien que posee Marco Bruto. También observe que, en general, los enunciados del tipo “... es un ...” se simbolizan mediante predicados que representan subconjuntos de individuos del universo de discurso. Más adelante volveremos sobre este tema.
3. En la forma clausal, la cuantificación existencial se representa mediante constantes y funciones de Skolem. En nuestro ejemplo, el uso de funciones de Skolem ha

sido inmediato pero en otras ocasiones encierra ciertas sutilezas. Para ilustrar este punto, volvamos al Ejemplo 3.6 y representamos el enunciado “todo el mundo tiene madre” mediante la fórmula “ $(\forall X)[esHumano(X) \rightarrow (\exists Y)(esHumano(Y) \wedge esMadre(Y, X))]$ ”. Si ahora damos los pasos necesarios para obtener su representación en notación clausal, obtenemos:

$$\begin{aligned}
 & (\forall X)[esHumano(X) \rightarrow (\exists Y)(esHumano(Y) \wedge esMadre(Y, X))] \\
 \Leftrightarrow & (\forall X)[\neg esHumano(X) \vee (\exists Y)(esHumano(Y) \wedge esMadre(Y, X))] \\
 \Leftrightarrow & (\forall X)(\exists Y)[\neg esHumano(X) \vee (esHumano(Y) \wedge esMadre(Y, X))] \\
 \Leftrightarrow & (\forall X)(\exists Y)[(\neg esHumano(X) \vee esHumano(Y)) \\
 & \quad \wedge (\neg esHumano(X) \vee esMadre(Y, X))] \\
 \Rightarrow & (\text{skolemización}) \\
 (\forall X)[& (\neg esHumano(X) \vee esHumano(madre(X))) \\
 & \quad \wedge (\neg esHumano(X) \vee esMadre(madre(X), X))] \\
 \Rightarrow & (\text{paso a forma clausal}) \\
 \neg esHumano(X) \vee & esHumano(madre(X)) \\
 \neg esHumano(X) \vee & esMadre(madre(X), X) \\
 \Rightarrow & (\text{paso a notación clausal}) \\
 esHumano(madre(X)) \leftarrow & esHumano(X) \\
 esMadre(madre(X), X) \leftarrow & esHumano(X)
 \end{aligned}$$

Aquí, el hecho destacable es que debemos usar la función de Skolem “*madre*” y que, partiendo de un enunciado, obtenemos dos cláusulas, la primera de las cuales declara que el individuo “*madre(X)*” es humano si “*X*” lo es.

También un enunciado del lenguaje natural como “algunos hombres son animales”, y otros semejantes en su forma, presentan esta característica. El lector puede comprobar que su paso a notación clausal conduce a la obtención de dos cláusulas:

$$\begin{aligned}
 hombre(a) \leftarrow \\
 animal(a) \leftarrow
 \end{aligned}$$

Es conveniente notar que estas cláusulas también sirven como representación del enunciado “algunos animales son hombres”.

4. Cuando llegamos a una formalización en la lógica de predicados del tipo

$$(\forall X)(\forall Y)(romano(X) \wedge gobernante(Y) \wedge asesina(X, Y) \rightarrow enemigo(X, Y)),$$

para pasar a la notación clausal, los pasos dados en el Ejemplo 8.6 sugieren como regla práctica que basta con eliminar los cuantificadores e introducir la implicación inversa. Sin más preámbulos puede escribirse:

$$enemigo(X, Y) \leftarrow romano(X) \wedge gobernante(Y) \wedge asesina(X, Y).$$

En la Tabla 8.2 se muestran las correspondientes representaciones en notación

clausal para una serie de fórmulas típicas de la lógica de predicados. Las representaciones de la tabla pueden utilizarse como esquemas que sirven de atajos en el proceso de traducción a la forma clausal. Como puede apreciarse, muchas veces, en el momento de la traducción es preferible no eliminar el condicional por la disyunción y aplicar alguno de estos esquemas. Esta forma de proceder puede ahorrar numerosos pasos en el proceso de traducción.

Para profundizar en los diferentes aspectos del problema de la representación en forma clausal y adquirir cierta habilidad en el uso de atajos, se recomienda la lectura del Capítulo 2 y el 10 del libro de Kowalski [81].

Tabla 8.2 Esquemas para facilitar la representación en forma clausal de una fórmula de la lógica de predicados.

Fórmula	Notación clausal
$(\forall X)\mathcal{B}(X)$	$\mathcal{B}(X) \leftarrow$
$(\forall X)(\mathcal{A}(X) \rightarrow \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow \mathcal{A}(X)$
$(\forall X)(Q(X) \rightarrow \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow Q(X)$
	$\mathcal{B}_1(X) \leftarrow Q(X)$
$(\forall X)(Q(X) \rightarrow \mathcal{B}_1(X) \wedge \dots \wedge \mathcal{B}_n(X))$	\vdots
	$\mathcal{B}_n(X) \leftarrow Q(X)$
	$\mathcal{B}(X) \leftarrow Q_1(X)$
$(\forall X)(Q_1(X) \vee \dots \vee Q_n(X) \rightarrow \mathcal{B}(X))$	\vdots
	$\mathcal{B}(X) \leftarrow Q_n(X)$
$(\forall X)(Q(X) \rightarrow \mathcal{B}_1(X) \vee \dots \vee \mathcal{B}_n(X))$	$\mathcal{B}_1(X) \leftarrow Q(X) \wedge \neg \mathcal{B}_2(X) \wedge \dots \wedge \neg \mathcal{B}_n(X)$
$(\forall X)\neg Q(X)$	$\leftarrow Q(X)$
$(\forall X)(\mathcal{A}(X) \rightarrow \neg \mathcal{B}(X))$	$\leftarrow \mathcal{A}(X) \wedge \mathcal{B}(X)$
$(\forall X)(Q_1(X) \rightarrow \neg Q_2(X))$	$\leftarrow Q_1(X) \wedge Q_2(X)$
$(\forall X)(Q_1(X) \rightarrow \neg Q_2(X) \vee \mathcal{B}(X))$	$\mathcal{B}(X) \leftarrow Q_1(X) \wedge Q_2(X)$
$(\forall X)(Q_1(X) \rightarrow (Q_2(X) \rightarrow \mathcal{B}(X)))$	$\mathcal{B}(X) \leftarrow Q_1(X) \wedge Q_2(X)$
$(\exists X)\mathcal{A}(X)$	$\mathcal{A}(a) \leftarrow$
	$\mathcal{A}_1(a) \leftarrow$
$(\exists X)(\mathcal{A}_1(X) \wedge \dots \wedge \mathcal{A}_n(X))$	\vdots
	$\mathcal{A}_n(a) \leftarrow$
$(\exists X)(\mathcal{A}_1(X) \vee \dots \vee \mathcal{A}_n(X))$	$\mathcal{A}_1(a) \leftarrow \neg \mathcal{A}_2(a) \wedge \dots \wedge \neg \mathcal{A}_n(a)$
$(\exists X)(\mathcal{A}(X) \wedge \neg \mathcal{B}(X))$	$\mathcal{A}(a) \leftarrow$
	$\leftarrow \mathcal{B}(a)$
$(\exists X)\neg Q(X)$	$\leftarrow Q(a)$
$(\forall X)(\exists Y)Q(X, Y) \rightarrow \mathcal{B}(X)$	$\mathcal{B}(X) \leftarrow Q(X, Y)$
$(\forall X)(Q(X) \rightarrow (\exists Y)\mathcal{B}(X, Y))$	$\mathcal{B}(X, f(X)) \leftarrow Q(X)$

Los \mathcal{A} 's y los \mathcal{B} 's representan átomos, mientras que los Q 's representan conjunciones de literales. Los símbolos a y f son, respectivamente, una constante y una función de Skolem.

8.3.1. Cuantificación existencial, variables extra, negación y respuesta a preguntas

Como ya sabemos, la existencia de un elemento se representa en la forma clausal mediante el uso de constantes y funciones de Skolem. En este apartado intentaremos poner en relación la cuantificación existencial con la aparición de variables extra en las cláusulas, la negación y la respuesta a preguntas. Para centrar el discurso, trabajaremos sobre una serie de ejemplos.

Ejemplo 8.7

Consideremos el enunciado castellano “Una persona es abuelo de otra si tiene algún hijo que sea progenitor de esa otra”. Utilizando la interpretación del Ejemplo 8.2 ampliada con la relación

$\text{progenitor}(X, Y) : X \text{ es un padre o una madre de } Y,$

podemos formalizar el enunciado en la lógica de predicados como:

$$(\forall X)(\forall Y)[(\exists Z)(\text{padre}(X, Z) \wedge \text{progenitor}(Z, Y)) \rightarrow \text{abuelo}(X, Y)].$$

Si aplicamos los pasos necesarios para la traducción de esta fórmula a la forma clausal, obtenemos:

$$\begin{aligned} & (\forall X)(\forall Y)[(\exists Z)(\text{padre}(X, Z) \wedge \text{progenitor}(Z, Y)) \rightarrow \text{abuelo}(X, Y)] \\ & \Leftrightarrow (\forall X)(\forall Y)[\neg(\exists Z)(\text{padre}(X, Z) \wedge \text{progenitor}(Z, Y)) \vee \text{abuelo}(X, Y)] \\ & \Leftrightarrow (\forall X)(\forall Y)[(\forall Z)(\neg\text{padre}(X, Z) \vee \neg\text{progenitor}(Z, Y)) \vee \text{abuelo}(X, Y)] \\ & \Leftrightarrow (\forall X)(\forall Y)(\forall Z)(\neg\text{padre}(X, Z) \vee \neg\text{progenitor}(Z, Y) \vee \text{abuelo}(X, Y)) \\ & \Rightarrow (\text{paso a forma clausal}) \\ & (\neg\text{padre}(X, Z) \vee \neg\text{progenitor}(Z, Y) \vee \text{abuelo}(X, Y)) \end{aligned}$$

Ahora el enunciado original tiene la siguiente representación en notación clausal:

$$\text{abuelo}(X, Y) \leftarrow \text{padre}(X, Z) \wedge \text{progenitor}(Z, Y).$$

Al representar en notación clausal el enunciado del Ejemplo 8.7 apreciamos que la variable Z aparece en el cuerpo de la cláusula pero no es su cabeza. Este tipo de variables se denominan *variables extra*. Como muestra el Ejemplo 8.7 estas variables están asociadas a un cuantificador existencial en la fórmula de la lógica de predicados. Por consiguiente, las variables extra que aparecen en las cláusulas expresan existencia. En general, para una cláusula del tipo “ $B(X) \leftarrow Q(X, Y)$ ” son posibles las siguientes lecturas:

1. para todo X e Y , se cumple $B(X)$ si se cumple $Q(X, Y)$;
2. para todo X , se cumple $B(X)$ si existe un Y tal que se cumple $Q(X, Y)$.

La negación puede expresarse de forma directa en la lógica de predicados. Sin embargo, en la notación clausal debe expresarse indirectamente, ya que la cabeza de una cláusula y, por tanto, los hechos no pueden aparecer negados. La única posibilidad es representar la negación en forma de un objetivo.

Ejemplo 8.8

Considerando que Bat es el perro de mi sobrina, el enunciado castellano “Bat no es un hombre” se representa en la lógica de predicados mediante la fórmula “ $\neg \text{hombre}(\text{bat})$ ” que se corresponde con el objetivo “ $\leftarrow \text{hombre}(\text{bat})$ ”.

Ejemplo 8.9

El enunciado castellano “Nada es malo y bueno a la vez” se representa en la lógica de predicados mediante la fórmula

$$(\forall X) \neg (\text{malo}(X) \wedge \text{bueno}(X))$$

que se corresponde con la cláusula objetivo

$$\leftarrow (\text{malo}(X) \wedge \text{bueno}(X)).$$

Si hablamos en términos coloquiales y un tanto informales, podemos decir que en la forma clausal, el símbolo “ \leftarrow ” de los objetivos puede entenderse como una negación.

Dado un conjunto de enunciados que constituyen una teoría, podemos aplicar las técnicas de la programación lógica para responder a preguntas de diversos tipos. Las más habituales son:

1. Preguntas del tipo “si/no” (e.g. ¿Bruto es partidario de Pompeyo?, que se formaliza en la lógica de predicados como: $\text{partidario}(\text{bruto}, \text{pompeyo})$).
2. Preguntas del tipo “llenar espacios en blanco” (e.g. ¿Bruto odia a algún gobernante?, que se formaliza como: $(\exists X)[\text{odia}(\text{bruto}, X) \wedge \text{gobernante}(X)]$).

En general las preguntas del segundo tipo vienen representadas por fórmulas de la forma

$$(\exists X) \dots (\exists Z) Q(X, \dots, Z),$$

con una o más variables cuantificadas existencialmente y se responden hallando los valores (más generales) de las variables que permiten probar dicha fórmula a partir de la teoría. Como las técnicas deductivas de la programación lógica se basan en la aplicación de un método de refutación por resolución, que solo es válido para fórmulas expresadas en forma clausal, tendremos que:

1. Transformar los enunciados de la teoría a forma clausal.

2. Negar lo que queremos probar:

$$\neg(\exists X) \dots (\exists Z) Q(X, \dots, Z) \Leftrightarrow (\forall X) \dots (\forall Z) \neg Q(X, \dots, Z),$$

de manera que la pregunta original se transforma en la cláusula objetivo:

$$\leftarrow Q(X, \dots, Z).$$

3. Aplicar la estrategia de resolución SLD².

Ejemplo 8.10

Dado el conjunto de cláusulas del Ejemplo 8.6, expresadas en notación clausal:

1. $hombre(bruto) \leftarrow$
2. $partidario(bruto, pompeyo) \leftarrow$
3. $romano(X) \leftarrow partidario(X, pompeyo)$
4. $gobernante(cesar) \leftarrow$
5. $odia(X, cesar) \leftarrow romano(X) \wedge enemigo(X, cesar)$
6. $enemigo(X, elEnemigoDe(X)) \leftarrow$
7. $enemigo(X, Y) \leftarrow romano(X) \wedge gobernante(Y) \wedge asesina(X, Y)$
8. $asesina(bruto, cesar) \leftarrow$

y el objetivo “ $\leftarrow odia(bruto, X) \wedge gobernante(X)$ ”, se obtiene como respuesta computada la sustitución $\theta = \{X/cesar\}$ y “ $odia(bruto, cesar) \wedge gobernante(cesar)$ ” como instancia computada del objetivo inicial.

Nótese que, al igual que sucede con las variables extra, las variables de un objetivo están asociadas con un cuantificador existencial. Desde el punto de vista de lo que se prueba, un objetivo como “ $\leftarrow odia(bruto, X) \wedge gobernante(X)$ ” admite la siguiente lectura: existe un X tal que $odia(bruto, X)$ y $gobernante(X)$.

En muchas ocasiones se quiere establecer como consecuencia de un conjunto de cláusulas, una fórmula que es una implicación, como cuando se desea probar por ejemplo que “todo primo es impar”. En casos como éste y similares, la traducción del enunciado correspondiente conduce a una fórmula de la lógica de predicados del tipo:

$$(\forall X)(Q_1(X) \rightarrow Q_2(X)).$$

²Si el conjunto de cláusulas obtenido no es un conjunto de cláusulas de Horn, habrá que aplicar una estrategia completa de resolución para obtener la respuesta a la pregunta planteada.

Para emplear el principio de resolución, nuevamente, debemos negar la fórmula que se persigue como conclusión y transformarla a forma clausal. Obtenemos:

$$\begin{aligned} & \neg(\forall X)(Q_1(X) \rightarrow Q_2(X)) \\ & \Leftrightarrow \neg(\forall X)(\neg Q_1(X) \vee Q_2(X)) \\ & \Leftrightarrow (\exists X)\neg(\neg Q_1(X) \vee Q_2(X)) \\ & \Leftrightarrow (\exists X)(Q_1(X) \wedge \neg Q_2(X)) \\ & \Rightarrow (\text{paso a forma clausal}) \\ & Q_1(a) \wedge \neg Q_2(a) \end{aligned}$$

que, en notación clausal, se expresa como:

$$\begin{aligned} & Q_1(a) \leftarrow, \\ & \leftarrow Q_2(a). \end{aligned}$$

Como resultado de este proceso podemos extraer la siguiente regla práctica: para transformar a forma clausal la negación de una implicación, basta con afirmar la existencia de un individuo a que satisfaga todas las condiciones del antecedente, i.e., hay que introducir el hecho “ $Q_1(a) \leftarrow$ ”, y negar que satisfaga las conclusiones del consecuente i.e., hay que introducir el objetivo “ $\leftarrow Q_2(a)$ ”.

8.3.2. Conjuntos, tipos de datos y las relaciones “instancia” y “es_un”

En castellano se emplean palabras que hacen referencia a conjuntos de individuos, e.g., “hombres”, “animales”, “números naturales”, “números primos”. De una forma muy simple podemos identificar los conjuntos con los tipos de datos de los lenguajes de programación. En la lógica de predicados se emplea un símbolo de predicado, que hace referencia a una propiedad o propiedades que caracterizan a los individuos del conjunto, para representar un conjunto o tipo. Asimismo, se emplea un predicado aplicado sobre una variable para expresar la pertenencia de un individuo indeterminado a ese conjunto o tipo (e.g., “ $hombre(X)$ ”, “ $animal(X)$ ”, “ $natural(X)$ ”, “ $primo(X)$ ”, etc.). A lo largo de todo el Capítulo 6 y el 7 hicimos uso de esta forma de representar los tipos de datos. Sin embargo es posible una forma alternativa de tratar los conjuntos o tipos de datos.

Para tratar los tipos de datos como individuos en sí y no como propiedades de individuos suelen emplearse símbolos de constante para designar los conjuntos con un nombre y las relaciones “*instancia*” y “*es_un*”³ para identificar elementos del conjunto y declarar la inclusión de ciertos conjuntos en otros:

$$\begin{aligned} & \textit{instancia}(X, A) : \quad X \text{ pertenece al conjunto } A; \\ & \textit{es_un}(A, B) : \quad A \text{ es subconjunto de } B; \end{aligned}$$

³Las relaciones “*instancia*” y “*es_un*” también han recibido los nombres de “elem” y “subc”, respectivamente. Como veremos, estas relaciones se emplean para modelar la herencia de propiedades.

Cuando se representan los conjuntos utilizando las relaciones “*instancia*” y “*es_un*”, se necesita introducir un nuevo axioma en la teoría que afirme que un elemento que pertenece a un subconjunto A de B, también pertenece al conjunto B:

$$(\forall X)(\forall A)(\forall B)(\text{instancia}(X, A) \wedge \text{es_un}(A, B) \rightarrow \text{instancia}(X, B)),$$

Representada en notación clausal, esta fórmula se escribe como:

$$\text{instancia}(X, B) \leftarrow \text{instancia}(X, A) \wedge \text{es_un}(A, B).$$

Ejemplo 8.11

El conjunto de cláusulas del Ejemplo 8.6 se puede escribir, empleando esta nueva forma de representar los conjuntos, como:

1. $\text{instancia}(\text{bruto}, \text{hombre}) \leftarrow$
2. $\text{partidario}(\text{bruto}, \text{pompeyo}) \leftarrow$
3. $\text{instancia}(X, \text{romano}) \leftarrow \text{partidario}(X, \text{pompeyo})$
4. $\text{instancia}(\text{cesar}, \text{gobernante}) \leftarrow$
5. $\text{odia}(X, \text{cesar}) \leftarrow \text{romano}(X) \wedge \text{enemigo}(X, \text{cesar})$
6. $\text{enemigo}(X, \text{elEnemigoDe}(X)) \leftarrow$
7. $\text{enemigo}(X, Y) \leftarrow \text{romano}(X) \wedge \text{gobernante}(Y) \wedge \text{asesina}(X, Y)$
8. $\text{asesina}(\text{bruto}, \text{cesar}) \leftarrow$

donde ahora las constantes “*hombre*”, “*romano*” y “*gobernante*” son constantes que representan conjuntos.

Ejemplo 8.12

Si representamos los enunciados

1. Los hombres son mortales.
2. Todo romano es un hombre.
3. César fue un gobernante romano.

en notación clausal, obtenemos:

1. $\text{mortal}(X) \leftarrow \text{hombre}(X)$
2. $\text{hombre}(X) \leftarrow \text{romano}(X)$
3. $\text{gobernante}(\text{cesar}) \leftarrow$
4. $\text{romano}(\text{cesar}) \leftarrow$

Estas fórmulas se pueden escribir empleando la nueva forma de representar los conjuntos como:

1. $\text{es_un}(\text{hombre}, \text{mortal}) \leftarrow$
2. $\text{es_un}(\text{romano}, \text{hombre}) \leftarrow$
3. $\text{instancia}(\text{cesar}, \text{gobernante}) \leftarrow$
4. $\text{instancia}(\text{cesar}, \text{romano}) \leftarrow$
5. $\text{instancia}(X, B) \leftarrow \text{instancia}(X, A) \wedge \text{es_un}(A, B)$

donde ahora las constantes “*hombre*”, “*romano*”, “*gobernante*”, y “*mortal*” son constantes que representan conjuntos.

Observaciones 8.3 En el último ejemplo es conveniente notar que:

1. Los enunciados del tipo “Los hombres son mortales” definen una inclusión de conjuntos.
2. Ha sido necesario introducir la cláusula 5 para poner en correspondencia las relaciones “*instancia*” y “*es_un*”; de otro modo, ahora, no se podría contestar afirmativamente a la pregunta de si César es mortal.



Dado que son posibles varias representaciones para la noción de conjunto, hay que ser consistente y, una vez elegida una forma de representación, no debe mezclarse con otras.

8.3.3. Representación mediante relaciones binarias

En la lógica de predicados, los argumentos de una relación n -aria tienen una misión, fijada arbitrariamente, según el orden que ocupan en la fórmula (ya indicamos que nosotros elegíamos como sujeto de la relación al primero de los argumentos). Alternativamente, toda relación n -aria puede expresarse como la unión de $n+1$ relaciones binarias que ponen de manifiesto de forma explícita la función de cada uno de sus argumentos. Para sustituir una relación n -aria por $n + 1$ relaciones binarias se recomienda seguir los siguientes pasos:

1. Tratar la relación n -aria como una individualidad, asignandole un nombre (e.g., “*d1*”) y, también, asignar un nombre a la acción que simboliza *d1* (e.g., “*dar*”).
2. Introducir una relación binaria que indique la pertenencia de la relación n -aria a una determinada acción (e.g., “*d1* es un acto de *dar*”, que puede simbolizarse como:
instancia(d1, dar)).
3. Introducir una relación binaria por cada argumento de la relación n -aria, para fijar el papel de cada argumento en la acción “*d1*”.

Ejemplo 8.13

La relación ternaria “Juan dio un libro a María” puede representarse en notación clausal mediante el hecho: *dar(juan, libro, maria) ←*. Esta relación puede expresarse en los

siguientes términos:

1. Hay una acción que denominamos d1
2. que es una instancia de un acto de dar
3. por un dador, juan
4. de un objeto, libro
5. a un receptor, maria

Este análisis permite transcribir, de manera inmediata, la relación ternaria inicial en términos de cuatro relaciones binarias:

1. *instancia(d1, dar) ←*
2. *dador(d1, juan) ←*
3. *objeto(d1, libro) ←*
4. *receptor(d1, maria) ←*

Observe que se ha prescindido del hecho que indica que “*dar*” es del tipo “*acción*” (i.e., *es_un(dar, acción)*). En los sistemas de IA que utilizan esta técnica para representar el conocimiento, las relaciones binarias como “*dador(d1, juan)*”, que caracterizan la relación *d1*, reciben el nombre de *slots*. El nombre de la relación “*dador*” es el nombre del *slot* y el nombre del segundo argumento “*juan*” es el valor del *slot*. Esta nomenclatura proviene de los sistemas de representación mediante *marcos* (véase el Ejercicio 8.21 para obtener más información).

Entre las ventajas de la representación binaria podemos citar las siguientes:

1. Es más fácil de leer (y entender) que la representación *n*-aria, cuando el número *n* de argumentos es grande.
2. Es más expresiva que la representación *n*-aria. Dado que en la representación binaria las relaciones *n*-arias reciben un nombre y, por lo tanto, se tratan como un individuo, es posible emplearlas como argumentos de otra relación. Esto no puede hacerse con propiedad para otro tipo de relación ya que conduciría a expresiones que no son fórmulas bien formadas de la lógica de predicados.
3. Facilita la inserción de hechos nuevos, reduciendo la necesidad de modificar los hechos existentes, lo que es particularmente útil cuando se trabaja en el contexto de las bases de datos deductivas. Si en el ejemplo anterior quisieramos detallar que el libro regalado era “La ciudad y las estrellas” escrito por Arthur C. Clarke, bastaría con realizar una ligera modificación en el tercer hecho para escribirlo como “*objeto(d1, l1) ←*”, e introducir cuatro nuevas relaciones binarias:

5. *instancia(l1, libro) ←*
6. *titulo(l1, ‘La ciudad y las estrellas’) ←*
7. *autor(l1, ‘Arthur C. Clarke’) ←*

Con otra técnica de representación de la información (e.g., tablas de un base de datos relacional), habría sido necesario modificar por completo la estructura de la base de datos.

4. También en el contexto de las bases de datos deductivas, facilita la formulación de preguntas, ya que no es necesario recordar la estructura de la información almacenada. Observe que, si se representa la acción de dar un libro a María mediante un objeto estructurado

```
dar( juan,
    libro( autor('Arthur C. Clarke'),
          titulo('La ciudad y las estrellas')),
    maria) ←
```

para extraer información debemos conocer qué representa cada argumento dentro de esa estructura y, todavía con más precisión, qué posición ocupa la información dentro de una estructura fuertemente anidada.

Por ejemplo, para obtener el título del libro tendríamos que plantear el siguiente objetivo: “*dar(, libro(, titulo(X)),) ←*” que enlazaría la variable *X* con el título deseado.

5. En sintonía con el punto anterior, dado que la relación se almacena como un conjunto de hechos en la base de datos interna del sistema Prolog, es posible acceder a las informaciones a través del mecanismo de búsqueda del propio sistema Prolog.
6. Desde el punto de vista computacional, permite la especificación de reglas generales y restricciones de integridad que no podrían expresarse con tanta facilidad mediante la representación *n*-aria.

Por contra, algunas desventajas de la representación mediante relaciones binarias son:

1. Hay muchas relaciones que se expresan más fácilmente como relaciones *n*-arias.
2. El acceso a todas las informaciones relacionadas con una entidad u objeto es difícil y poco eficiente. La razón es que no hay una estructura comparable al registro de una base de datos relacional, a cuyas informaciones se accede a través de un campo clave.
3. Dado que la relación se almacena como un conjunto de hechos en la base de datos interna del sistema Prolog, cualquier modificación de su estructura requiere el uso de los predicados *assert* y *retract* para añadir ciertos hechos y eliminar otros, con la consiguiente pérdida de declaratividad. Adicionalmente, cambiar la estructura de una relación puede requerir muchas operaciones de copia, lo que lleva aparejado un coste computacional elevado y la consiguiente pérdida de eficiencia.

Por consiguiente, los argumentos a favor de la representación mediante relaciones binarias no son definitivos y el diseñador de un sistema deberá sopesar, cuidadosamente, cuál es la opción más adecuada conforme a sus intereses.

8.4 ELECCIÓN DE UNA ESTRUCTURA DE DATOS: VENTAJAS E INCONVENIENTES

Una estructura de datos por sí misma no puede verse como una técnica de representación del conocimiento, debemos suplementarla con capacidades deductivas. Sin embargo, la elección de una buena estructura de datos puede ser determinante a la hora de representar el conocimiento. En este apartado damos una serie de consejos que pueden ayudar a establecer una buena elección. Para concretar ideas nos centraremos directamente en las construcciones del lenguaje Prolog.

Básicamente, una estructura de datos puede representarse en Prolog de dos maneras diferentes:

1. como un término (objeto simple o estructurado, si usamos la propia terminología del lenguaje Prolog), que puede pasarse como argumento de una relación;
2. como un conjunto de hechos almacenados en la base de datos interna del sistema Prolog.

Así pues, la elección fundamental consiste en decidir si usar términos o relaciones (definidas, en parte, por un conjunto de hechos) para representar estructuras de datos.

Ciertos problemas admiten de forma natural una representación de sus estructuras de datos mediante términos: constantes, listas y objetos estructurados.

- Las constantes, como ya se ha dicho, se emplean para representar nombres de personas (e.g. haran, isaac, etc., nombres empleados en el programa sobre relaciones familiares del Apartado 6.4), valores y, en general, objetos simples (e.g. los estados del autómata finito no determinista del Apartado 6.3.4).
- Las listas deben emplearse cuando el número de elementos a tratar es indeterminado o se desconoce su posición dentro de una estructura (término) o esa posición no se considera importante. En el Apartado 9.2.3 del próximo capítulo, plantearemos el problema de la generación de planes en el mundo de los bloques: un mundo ideal compuesto por un conjunto de bloques que hay que apilar en un orden concreto sobre unas posiciones concretas encima de una mesa (por ejemplo, las posiciones que denotamos mediante las constantes “p1”, “p2” y “p3”). Para este problema, una posible representación del estado (o situación) en que se encuentra el sistema de bloques es mediante una lista de pilas de bloques. Las pilas de bloques son, a su vez, listas. Por ejemplo, si el sistema se encuentra en un estado en el que el bloque “a” está sobre el “b” y éste sobre la posición “p1”, mientras

que la posición “ p_2 ” está vacía y el bloque “c” está sobre la posición “ p_3 ” (como muestra la Figura 8.3), su representación será:

$\begin{array}{ccc} [[a,b], & [], & [c]] \\ \hline \cdots & \cdots & \cdots \\ p_1 & p_2 & p_3 \end{array}$ estamos suponiendo que la lista 1
está sobre la posición “ p_1 ” etc.

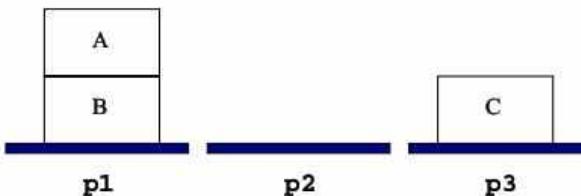


Figura 8.3 Situación en el mundo de los bloques.

Por otra parte, si admitimos cierta flexibilidad a la hora de representar la configuración sobre las posiciones p_1 , p_2 o p_3 , esto es, no distinguimos entre las diferentes posiciones, los siguientes estados son equivalentes al anterior:

$\begin{array}{l} [[a,b], [c], []] \text{ o } [[], [a,b], [c]] \text{ o } [[], [c], [a,b]] \text{ o } \\ [[c], [], [a,b]] \text{ o } [[c], [a,b], []]. \end{array}$

Observe que estamos ignorando el orden en el que se presentan las pilas dentro de una configuración, es decir, la lista principal se está tratando como un conjunto. Esta circunstancia, en la que todas las posibles permutaciones se consideran equivalentes, obliga a un procesamiento extra para comparar estados o buscar un elemento particular en una secuencia de estados (ya que el mecanismo de unificación del lenguaje Prolog no es capaz de realizar esta tarea convenientemente, al contrario de lo que sucede cuando representamos las estructuras de datos mediante términos). A este respecto, la definición del siguiente predicado puede ser útil:

```
%%% equiv(Xs, Ys) : La lista Xs es equivalente a la lista Ys,
%%% si posee los mismos elementos independientemente de
%%% su orden.
equiv([], []).
equiv([X|Xs], Ys) :- selecciona(X, Ys, R), equiv(Xs, R).
```

Este predicado tiene una gran versatilidad⁴ y puede aplicarse a cualquier estructura de datos que sea factible representar mediante listas, desde conjuntos elementales a configuraciones de bloques. Cuando x_s e y_s sean configuraciones, serán equivalentes si están compuestas por las mismas pilas, cualquiera que sean las posiciones

⁴En concreto, puede generar todas las permutaciones de una lista de elementos. Véase el Ejercicio 8.14.

p1, p2, y p3 sobre las que están dispuestas. El predicado `selecciona(X, L, R)`, definido en el Ejercicio 7.26, selecciona un elemento X de una lista de L y deja los restantes elementos en la lista remanente R.

- Si, debido la naturaleza del problema, el número de elementos o características definitorias de un problema no cambian, es conveniente utilizar un término de estructura fija. Este es el caso del problema de los contenedores de agua, que se estudia en el Apartado 9.2.2, en el que se elige un término `est(C7, C5)` (un vector de dos posiciones) para representar la noción de estado; la variable C7 representa los litros de agua almacenados en el contenedor de siete litros y la variable C5 representa los litros de agua almacenados en el contenedor de cinco litros. También en el caso del problema del mono y el plátano, que se resuelve en el Apartado 9.2.1, se elige un término para representar el estado del sistema. En concreto, se utiliza el término `s(Pm, Sm, Pc, Tm)`, con cuatro argumentos que sirven para almacenar información sobre las características más representativas del problema. Como ya se ha comentado en otras partes de este libro, es fácil acceder a las informaciones almacenadas en las estructuras de tamaño fijo utilizando el mecanismo de unificación del lenguaje Prolog. Por ejemplo, el hecho

```
possible(s(puerta, suelo, ventana, no tiene)).
```

muestra la posibilidad de que el sistema esté en un cierto estado. Ahora supongamos que queremos conocer la posición del mono en ese estado. La definición del predicado

```
posicion_mono(P) :- possible(s(P, _, _, _)).
```

facilita el acceso a esa información en un solo paso de unificación. Una desventaja de representar las estructuras de datos mediante términos es que, conforme las estructuras se hacen más complejas, la tarea de acceder a la información que almacenan se hace más tediosa. Para términos grandes, se hace imposible definir un predicado de acceso a la información en un solo paso de unificación. El programador deberá escribir (sub)procedimientos para seleccionar e inspeccionar las correspondientes subestructuras de esos términos. Sin embargo, una ventaja de este tipo de representación de la información es que las variables de un término pueden verse como “punteros” a otras estructuras. El programador no tienen que preocuparse de si un elemento es una variable que se ha enlazado a una estructura o es la estructura propiamente dicha. El mecanismo de unificación del lenguaje Prolog hace esa tarea de “desenlace” (acceso automático usando los enlaces) automáticamente para él.

La otra gran alternativa para representar información consiste en usar la base de datos interna del sistema Prolog para almacenar dicha información como un conjunto

de hechos. Su principal ventaja es la posibilidad de utilizar el propio mecanismo de búsqueda del sistema Prolog.

En un caso simple, como el que ilustra el Ejemplo 7.4, se puede almacenar el nombre de un conjunto de entidades, en este caso países, como un conjunto de hechos. En el caso más general, cualquier estructura puede representarse como un conjunto de hechos, empleando una técnica más elaborada como la introducida en el Apartado 8.3.3 para la representación mediante relaciones binarias. En este último caso, las estructuras de datos se representan utilizando constantes para referenciar los hechos que son parte de la misma estructura. Por ejemplo, dado el conjunto de hechos:

```
%% Acto de dar d1           %% Libro 11
instancia(d1, dar).      instancia(11, libro).
dador(d1, juan).         titulo(11, 'La ciudad y las estrellas').
objeto(d1, 11).           autor(11, 'Arthur C. Clarke').
receptor(d1, maria).
```

la constante `d1` puede verse como un índice o clave (única) capaz de agrupar todas las informaciones acerca de la acción de dar un libro a María y la constante `11` juega el mismo papel con relación a las informaciones que caracterizan una entidad, el libro “*La ciudad y las estrellas*” escrito por Arthur C. Clarke. Observe también como los identificadores de relación se usan solamente para relacionar constantes, algunas de las cuales pueden entenderse como “punteros” a otras entidades. Naturalmente, al adoptar este tipo de representación, todas las ventajas e inconvenientes enumerados en el Apartado 8.3.3 tienen plena vigencia. En lo que sigue, y para finalizar este apartado, profundizaremos en uno de esos aspectos.

Cuando las estructuras de datos se representan mediante una colección de hechos, el mecanismo de búsqueda del propio sistema Prolog puede ayudarnos en la recuperación de información. Basta una simple pregunta, cuando se desea acceder a la información directamente almacenada en el espacio de trabajo. El objetivo “?- `titulo(11, T)`.” nos permite obtener el título del libro, que se enlaza a la variable `T`. Sin embargo, no es fácil acceder a informaciones almacenadas de forma indirecta. Con respecto al ejemplo anterior, ahora no es inmediato responder a la pregunta “¿De qué clase es el objeto que se dio a María?” En principio, solamente podemos acceder a la información que está directamente almacenada, es decir, el hecho “`objeto(d1, 11)`.”. Para ello planteamos el objetivo “?- `objeto(d1, X)`.”, que enlaza la variable `X` a `11`. Ahora bien, si queremos contestar a la pregunta inicial, a través de la clave `11` podemos acceder a las informaciones relativas a la entidad `11`. Entonces, se puede interrogar al sistema sobre qué tipo de objeto es `11` mediante el objetivo “?- `instancia(11, Y)`.”. Finalmente, el sistema proporciona la respuesta enlazando la constante `libro` a la variable `Y`, ya que el hecho “`instancia(11, libro)`” es directamente accesible por unificación.

En general, en este contexto, un “puntero” a una entidad es simplemente una constante que se emplea como un índice o referencia a esa entidad. Aunque la recuperación directa de ciertos datos es sencilla (cuando están almacenados en la base de datos interna

de Prolog como simples hechos), el programador deberá “cortocircuitar” esos “punteros” para acceder a informaciones sobre entidades a las que solamente se puede llegar indirectamente. En el próximo apartado estudiamos la herencia de propiedades, que puede entenderse como un mecanismo de “desenlace” para un tipo particular de representación del conocimiento.

Para terminar este apartado, queremos señalar la relación existente entre las dos aproximaciones para representación del conocimiento presentadas aquí (basadas en términos y en átomos) y las aproximaciones más populares en el área de la modelización conceptual de bases de datos y sistemas de información: la aproximación deductiva y la operacional, respectivamente. En la primera, la historia del sistema queda registrada en el modelo que se construye como un término (por ejemplo `despide (microsoft, juan, contrata (microsoft, juan, solicita (juan, ofrece_plazas (microsoft, 1, bd_vacia))))`). Dicho término anida las funciones que representan todos los eventos ocurridos en la vida del sistema, es decir, su historia. Esta representación, propia de los sistemas de “razonamiento hacia atrás” o “por deducción”, facilita las actualizaciones mientras penaliza las consultas, ya que la respuesta a cada pregunta se deriva cada vez en tiempo de ejecución (por ejemplo, `vacantes(microsoft, despide (microsoft, juan, contrata (microsoft, juan, solicita (juan, ofrece_plazas (microsoft, 1, bd_vacia))))`). Por el contrario, la aproximación operacional o dinámica se basa en una representación que usa hechos que se asertan a la base de datos, cada vez que ocurre un evento, junto con todas las informaciones que se derivan de la ocurrencia de dicha acción. Este tipo de representación, propia de los sistemas de “razonamiento hacia adelante” o “por saturación”, favorece las consultas ya que toda la información derivada aparece explícitamente registrada en la base de datos y accesible directamente sin tener que recomputarla cada vez. Por el contrario, se penalizan las actualizaciones justo por el hecho de que, cada vez que ocurre un evento, no solo se registra la ocurrencia del evento sino que se satura la base de datos almacenando también todas las informaciones que se derivan de él. El primer tipo de representación es común en los sistemas de modelización que se basan en tecnología algebraica o funcional, donde no existe la posibilidad de simular una memoria global, mientras el segundo saca ventaja para simular la representación en memoria de la historia del sistema usando los predicados Prolog `assert` y `retract`.

8.5 REDES SEMÁNTICAS Y REPRESENTACIÓN CLAUSAL

Las redes semánticas⁵, desarrolladas por Quillian (1968) como modelo psicológico de la memoria asociativa humana, han alcanzado una considerable popularidad en el campo de la IA como herramientas para la representación del conocimiento ya que permiten implementar una de las formas más útiles de inferencia: la herencia de pro-

⁵También denominadas *redes asociativas*.

piedades. En este apartado estamos interesados, principalmente, en mostrar las relaciones existentes entre las redes semánticas y la lógica, por lo que nuestro tratamiento es muy limitado en otros aspectos. Un tratamiento más amplio pero asequible de las redes semánticas, y estructuras similares, se presenta en [27] y [126] (entre otros libros introductorios de IA).

Una *red semántica* es un grafo dirigido constituido por nodos y por arcos. Los *nodos* representan entidades que pueden ser individuos, objetos, clases (conjuntos) de individuos u objetos, conceptos, acciones genéricas (eventos) o valores. Los *arcos* representan relaciones entre nodos o bien atributos (i.e., características) de un nodo. Así pues, una red semántica consiste en un conjunto de entidades relacionadas. En la Figura 8.4 se muestra una red semántica (adaptación de la que aparece en [21]).

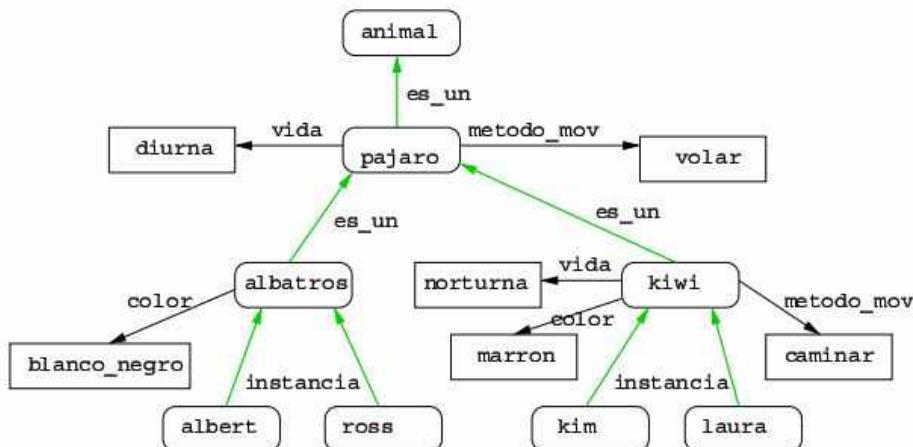


Figura 8.4 Un ejemplo de red semántica.

Ejemplo 8.14

En la Figura 8.4, “pájaro” es una entidad (una clase —de animal—) y “vida” es un atributo de esa entidad mientras que “diurna” es un valor de ese atributo. “Kiwi” es otra entidad (una clase —de pájaro—) y “color” es un atributo de dicha entidad cuyo valor es “marrón”. “Kim” también es una entidad (en este caso, un objeto —instancia de la clase Kiwi—).

Las redes semánticas, a diferencia de la representación del conocimiento mediante la lógica de predicados, están orientadas a la representación jerárquica de grandes conjuntos de hechos estructurados. Esto les da innegables ventajas. Las dos más importantes son [27]:

1. *Indexación de la información:* la capacidad de acceder a todas las informaciones relevantes sobre una entidad. Una vez estamos posicionados sobre un nodo pode-

mos acceder a todos sus atributos y relaciones mediante el uso de punteros (que es un mecanismo menos costoso que el de la unificación).

2. *Herencia de propiedades*: en la Figura 8.4 todos los atributos de los nodos representan conocimientos que pertenecen al dominio específico del problema. Sin embargo los atributos “es_un”, que indica que una clase está contenida en otra, e “instancia”, que indica que un elemento pertenece a una clase, son la base para la herencia de propiedades⁶. Los valores de los atributos de una entidad pueden ser heredados por otras entidades que son instancia de esa entidad o están en la relación “es_un” con esa entidad. Por ejemplo, el atributo “modo_desplazamiento” cuyo valor es “volar”, es heredado por la entidad “albatros”. El Algoritmo 1 muestra, de forma simplificada, cómo pueden inferirse hechos sobre un dominio, que no han sido almacenados explícitamente, mediante el mecanismo de la herencia.

Algoritmo 8.1 [Herencia de Propiedades]

Entrada: Una red semántica S y una petición de acceso a un valor V de un (supuesto) atributo A de una entidad E .

Salida: El valor V o una condición de **fallo**.

Comienzo

Inicialización: $NuevoE = E; Resultado = \text{falso}; SalirBucle} = \text{falso}$.

Repetir

En caso de que $NuevoE$ **posea** el atributo

1. A : $Resultado = V$, donde V es el valor del atributo A ;
 $SalirBucle = \text{verdadero}$;

2. *instancia*: $NuevoE = E'$, donde E' es el valor del atributo *instancia*;

3. *es_un*: $NuevoE = E'$, donde E' es el valor del atributo *es_un*;

4. **En otro caso:** $SalirBucle = \text{verdadero}$.

Fin caso

Hasta que $SalirBucle$.

Devolver $Resultado$.

Fin

De forma simple, el Algoritmo 1 procede de la siguiente forma: se sitúa en el nodo que representa la entidad E y busca si posee el atributo A . Si lo posee, accede a su valor mediante el correspondiente apuntador; si no, asciende en la jerarquía de la red

⁶Otro atributo de utilidad general cuyos valores son heredables es el atributo “tiene_parte” (*has_part*). Por simplicidad, en esta breve exposición no se considera este tipo de atributos. Véase sin embargo el Ejercicio 8.22.

semántica haciendo uso de las relaciones *instancia* o *es_un* y busca una entidad que posea el atributo *A*. Si la encuentra, devuelve su valor; en caso contrario, devuelve *falso*.

Para no complicar el Algoritmo 1 con detalles innecesarios, no se ha considerado la *herencia múltiple*, es decir, la posibilidad de que una entidad pueda heredar propiedades pertenecientes a varias clases padres.

A pesar de las diferencias entre las redes semánticas y la lógica de predicados, como se muestra en [104], éstas son técnicas equivalentes de representación del conocimiento en el sentido de que lo que puede expresarse con una es posible expresarlo con otra. La equivalencia entre la lógica de predicados y las redes semánticas también se estudia en [81] y más formalmente en [104]. En lo que sigue ponemos de manifiesto esta equivalencia haciendo ver, mediante un ejemplo, que es posible modelar el comportamiento de la red semántica de la Figura 8.4 utilizando un lenguaje de programación lógica como Prolog.

Ejemplo 8.15

Desde el punto de vista de la lógica de predicados, un nodo de una red semántica es un término mientras que un arco es una relación entre términos. Realizado este inciso, es inmediato representar en forma clausal el conocimiento que encierra la red semántica de la Figura 8.4.

```
% Entidad pajaro
es_un(pajaro,animal).
metodo_movimiento(pajaro,volar).
vida(pajaro,diurna).

% Entidad albatros
es_un(albatros,pajaro).
color(albatros,blanco_negro).

% Entidad albert
instancia(albert,albatros).

% Entidad ross
instancia(ross,albatros).

% Entidad kiwi
es_un(kiwi,pajaro).
color(kiwi,marron).
metodo_movimiento(kiwi,caminar).
vida(kiwi,nocturna).

% Entidad kim
instancia(kim,kiwi).
```

Como ya se hizo ver en el apartado dedicado a la representación de conjuntos, cuando se hace uso de las relaciones *instancia* y *es_un*, es necesario un axioma adicional

que describe cómo se puede combinar una relación de `instancia` con la relación `es_un` para dar lugar a una nueva relación de `instancia`.

```
% conocimiento implicito
instancia(X, B) :- es_un(A, B), instancia(X, A).
```

Este axioma nos permite responder a la pregunta de si `kim` es un pajaro, que en forma clausal se expresa como: `?- instancia(kim, pajaro).`

Para modelar el mecanismo de inferencia de la herencia de propiedades, debemos introducir un conjunto de reglas capaces de implementar el Algoritmo 1.

```
% pos(Hecho), el Hecho es posible
% Reglas que modelan la herencia de propiedades
% Un Hecho es posible si es deducible directamente de la red
pos(Hecho) :- Hecho,!.
% Un Hecho es posible si puede ser inferido por herencia
pos(Hecho) :- Hecho =.. [Rel,Arg1,Arg2],
              (es_un(Arg1,SuperArg); instancia(Arg1,SuperArg)),
              SuperHecho =.. [Rel,SuperArg,Arg2],
              pos(SuperHecho).
```

La primera regla trata el caso 1 del Algoritmo 1, en el que la entidad posee el atributo considerado. Dado que representamos la red semántica como un conjunto de cláusulas, una entidad posee un atributo si la relación que lo representa aparece explícitamente como un hecho del conjunto de cláusulas. Por consiguiente, la primera regla, que comprueba si un hecho es posible, simplemente lanza el hecho que desea contrastar como objetivo y, si unifica con alguno de los hechos presentes en la base de datos, concluye que la entidad posee el atributo, con lo que termina (devolviendo el valor del atributo correspondiente). La segunda regla implementa los casos 2 y 3 del Algoritmo 1, en el que la entidad no posee el atributo y es necesario ascender en la jerarquía de la red semántica para buscarlo. Con este este fin, la regla analiza el hecho considerado usando el predicado predefinido “`=..`”. La primera vez que lo usa es para descomponer el hecho en una lista de sus componentes: el nombre del atributo considerado, i.e. el nombre de la relación que se enlaza a la variable `Rel`; y sus argumentos. Dado que todo el conocimiento se ha expresado mediante relaciones binarias, sabemos que solo hay dos argumentos. El argumento `Arg1` debe contener información sobre la entidad y el argumento `Arg2` sobre el valor del atributo. En el segundo subobjetivo se comprueba si la entidad `Arg1` es instancia o está incluida en una superclase, cuyo nombre se enlaza a la variable `SuperArg`. Si es así, se forma un hecho que denominamos “`SuperHecho`” para comprobar si la superclase `SuperArg` posee el atributo `Rel`. Estas reglas nos permiten contestar a preguntas como: ¿Qué tipo de vida tiene Albert? (i.e., “`?- pos(vida(albert, v)).`”).

Observaciones 8.4

1. Nótese que la variable “Hecho” es una metavariable. Así pues, para implementar la herencia de propiedades debemos hacer uso de las habilidades del lenguaje Prolog para la metaprogramación. Aunque las reglas que definen la relación `pos` forman parte del lenguaje clausal, los objetivos como “?- `pos(vida(albert, V))`. ” no, ya que su argumento es una fórmula y no un término.
2. Tanto la red semántica como su representación en forma clausal tiene serias limitaciones. Por ejemplo, es imposible responder a preguntas como: ¿Qué entidades tienen vida diurna? (i.e., “?- `pos(vida(X, diurna))`. ”). La razón es que no hay arcos descendentes en la jerarquía y tampoco relaciones que los representen.



Concluimos el apartado haciendo notar que el Ejemplo 8.15 ilustra un tipo de programación denominada *orientada a objetos*. Este estilo de programación estructura los programas centrándose en las entidades y no en las acciones que realizan las entidades, como sucede con otros estilos de programación. En una primera aproximación, podemos decir que cuando se analiza el conocimiento (información) concerniente a un problema, la guía para su estructuración son los nombres y no los verbos.

RESUMEN

En este capítulo hemos presentado algunas de las aplicaciones de la programación lógica en dos de los campos más significativos de la Inteligencia Artificial (IA): la representación del conocimiento y la resolución de problemas. Hemos elegido estos campos porque, a través de ellos, podemos introducir nuevas técnicas de programación (declarativa).

- Una representación del conocimiento es una descripción formal que puede emplearse para la computación simbólica con la intención de que esa descripción pueda reproducir un comportamiento inteligente.
- Una característica común a todas las representaciones del conocimiento es que manejamos dos tipos de entidades: i) los hechos, verdades de un cierto mundo que queremos representar; ii) la representación de los hechos en un determinado formalismo. Estas entidades inducen la necesidad de distinguir entre dos niveles de representación [103]:
 1. el *nivel del conocimiento*, donde se describen los hechos y el comportamiento;
 2. el *nivel simbólico*, donde se describen los objetos del nivel del conocimiento mediante símbolos manipulables por programas.

- Siguiendo la orientación de Rich y Knight, 1994, hemos focalizado en los hechos, en las representaciones simbólicas y en la correspondencia que debe existir entre ellos.
- El razonamiento real es simulado por el funcionamiento de los programas, que manipulan las representaciones internas de los hechos. Son necesarias funciones de correspondencia para: i) dado un hecho inicial dotarlo de una representación interna y ii) dado una representación interna final establecer el hecho final correspondiente. Uno de los objetivos de la representación del conocimiento es la definición de las funciones de correspondencia y uno de sus problemas principales radica en el hecho de que las funciones de correspondencia no suelen ser biunívocas.
- Un buen sistema de representación del conocimiento, en un dominio particular, debe poseer las siguientes propiedades: i) suficiencia de la representación; ii) suficiencia deductiva; iii) eficiencia deductiva; y iv) eficiencia en la adquisición.
- Si bien no existe una técnica de representación del conocimiento que cumpla todos estos requisitos, la lógica posee una serie de características que la hacen atractiva como técnica de representación del conocimiento: i) es una forma natural de expresar relaciones; ii) existen métodos formales para determinar el significado y la validez de una fórmula; iii) posee diferentes cálculos deductivos que permiten la construcción de derivaciones (siendo éste uno de sus puntos fuertes); iv) existen resultados de corrección y completitud; v) como hemos visto, puede automatizarse el proceso deductivo; y vi) es modular (en el sentido de que la lógica clásica es monótona).
- La mayoría de las veces, el conocimiento sobre un dominio particular está expreso en un lenguaje natural. Por consiguiente, el problema de la representación del conocimiento, en nuestro contexto, se concreta en el de la traducción al lenguaje de la lógica de predicados.
- Hemos estudiado cómo la selección de una interpretación de partida puede ayudarnos en la traducción de enunciados del lenguaje natural a fórmulas lógicas. También se ha discutido el problema de la elección de la granularidad de la representación. Concluimos que, debido a la ambigüedad del lenguaje natural, no hay reglas precisas para su traducción. Lo mejor que podemos hacer es dar una serie de consejos prácticos para la traducción: i) meditar detenidamente sobre el sentido del enunciado del lenguaje natural; ii) seleccionar una interpretación de partida \mathcal{I} que sea adecuada y facilite la posterior representación del enunciado; y iii) derivar una fórmula que, cuando se interprete usando \mathcal{I} , tenga un significado lo más próximo posible al enunciado original.

- Dado que las cláusulas son, directamente, instrucciones de un lenguaje de programación lógico, nos hemos centrado en la representación en forma clausal, poniendo nuevamente en práctica algunas de las técnicas para la obtención de formas normales introducidas en el Capítulo 3. También hemos estudiado casos concretos de representación a forma clausal: cuantificación existencial, variables extra, negación y respuesta a preguntas; conjuntos y las relaciones “instancia” y “es_un”; y, finalmente, representación mediante relaciones binarias.
- Hemos visto que (en el lenguaje Prolog) hay dos formas de representar la información: mediante términos o mediante conjuntos de hechos almacenados en la base de datos interna de Prolog. Hemos discutido las ventajas e inconvenientes de elegir una u otra representación. Ambas poseen puntos a favor y puntos en contra, por lo que la decisión de elegir una u otra dependerá del problema concreto que se desee resolver.
- Finalmente, se ha mostrado que es inmediato representar en forma clausal el conocimiento que encierra una red semántica.

CUESTIONES Y EJERCICIOS

Cuestión 8.1 Señale cuál de las siguientes nociones relativas a un sistema de representación del conocimiento es falsa:

- Eficiencia en la adquisición es la capacidad de incorporar información (de control) adicional en las estructuras de conocimiento, con el fin de guiar el proceso deductivo en la dirección más adecuada.
- Eficiencia deductiva es la capacidad de incorporar información (de control) adicional en las estructuras de conocimiento, con el fin de guiar el proceso deductivo en la dirección más adecuada.
- Suficiencia deductiva es la capacidad de manipular las representaciones internas de los hechos con el fin de obtener otros nuevos.

Cuestión 8.2 a) Explique qué propiedades hacen de la lógica clásica una herramienta adecuada para la representación del conocimiento. b) Enumere alguno de sus inconvenientes.

Cuestión 8.3 Estudie las similitudes existentes entre las funciones de correspondencia, que ligan los hechos y las representaciones en una representación del conocimiento, y el concepto de interpretación de la lógica de predicados.

Ejercicio 8.4 (Mundo de los robots [105]) Los robots de un mundo bidimensional con obstáculos, como el representado en la Figura 8.5, pueden desplazarse de una celda a otra si está disponible, es decir, libre de obstáculos, o en ella no está situado otro robot. Un robot puede trazar un camino recorriendo una secuencia de celdas. Los robots están constreñidos por las cuatro paredes que rodean la habitación. Teniendo en cuenta los anteriores comentarios, utilizar la lógica de predicados para representar el conocimiento almacenado en ese pequeño mundo (introduzca las constantes, las funciones y los predicados que sean necesarios).

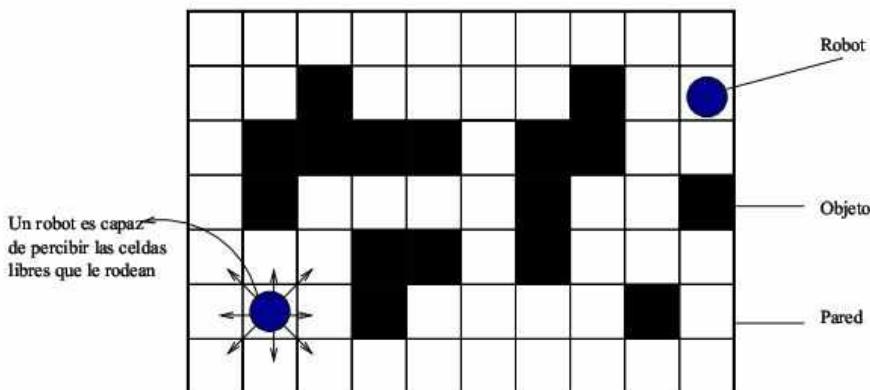


Figura 8.5 Robots en un mundo bidimensional con obstáculos.

Ejercicio 8.5 Dado el argumento:

Todos los estudiantes son ciudadanos.

Δ *Los votos de los estudiantes son votos de ciudadanos.*

- a) Expréselo formalmente en la lógica de predicados, mediante los símbolos de relación $s(X)$, $c(X)$, y $v(X, Y)$ que se interpretan como: “ X es un estudiante”, “ X es un ciudadano” y “ X es un voto de Y ”, respectivamente. b) Empleando las reglas que transforman una fórmula de la lógica de primer orden en una forma clausal, encuentre el conjunto de cláusulas asociadas con este argumento. c) Pruebe la corrección del argumento usando el principio de resolución.

Ejercicio 8.6 Dado el conjunto de cláusulas del Ejemplo 8.6, expresadas en notación clausal:

1. $\text{hombre(bruto)} \leftarrow$
2. $\text{partidario(bruto, pompeyo)} \leftarrow$
3. $\text{romano}(X) \leftarrow \text{partidario}(X, \text{pompeyo})$
4. $\text{gobernante(cesar)} \leftarrow$
5. $\text{odia}(X, \text{cesar}) \leftarrow \text{romano}(X) \wedge \text{enemigo}(X, \text{cesar})$
6. $\text{enemigo}(X, \text{elEnemigoDe}(X)) \leftarrow$
7. $\text{enemigo}(X, Y) \leftarrow \text{romano}(X) \wedge \text{gobernante}(Y) \wedge \text{asesina}(X, Y)$
8. $\text{asesina(bruto, cesar)} \leftarrow$

a) Represente el árbol de búsqueda para el problema de determinar quién es enemigo de César y muestre las respuestas computadas. b) Halle el universo de Herbrand, la base de Herbrand y el modelo mínimo del programa.

Ejercicio 8.7 Cuando se define un concepto se emplea la expresión “si, y solamente si” para poner en relación lo que se define con la definición. Por ejemplo, podemos definir el predicado “abuelo” del siguiente modo:

Para todo X e Y , X es abuelo de Y si, y solamente si, existe un Z tal que, X es padre de Z y Z es progenitor de Y .

En la lógica de predicados la expresión “si, y solamente si” tiene una formalización directa empleando la conectiva bicondicional. Sin embargo, cuando se emplea la forma clausal debe formalizarse en dos partes independientes: una correspondiente a la parte “si” y otra correspondiente a la parte “solamente si”. Es más, la parte “solamente si” conduce con frecuencia a representaciones antinaturales y poco intuitivas. Por este motivo, y porque la parte “si” es suficiente para derivar todos los hechos positivos sobre la relación que se está definiendo, Kowalski aconseja utilizar únicamente la parte “si” de las definiciones a la hora de representar el conocimiento en forma clausal⁷. Formalice y represente en notación clausal la definición del predicado “abuelo”, con el fin de constatar las dificultades recién comentadas. Compare el resultado con la representación obtenida en el Ejemplo 8.7. (Ayuda: se necesitan tres cláusulas y la introducción de una función de Skolem.)

Ejercicio 8.8 [105] El circuito lógico de la Figura 8.6 tiene cuatro cables, c_1 , c_2 , c_3 y c_4 , una puerta “and” y una puerta “not”. Las entradas c_1 y c_2 pueden estar en estado “on” u “off”. Si la puerta and funciona correctamente (ok), el cable c_3 está en on si y solo si las entradas c_1 y c_2 están ambas en on. Si la puerta not funciona correctamente (ok), el cable c_4 está en on si y solo si el cable c_3 está en off.

⁷Sin embargo, observe que la parte “solamente si” de las definiciones es necesaria para demostrar propiedades de los programas. También se necesita, en el contexto de las bases deductivas, para contestar preguntas que incluyen cuantificadores universales y negación [81].

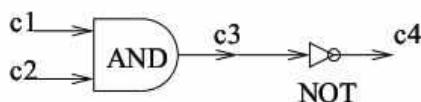


Figura 8.6 Un circuito lógico simple.

1. Utilice expresiones tales como `ok` (`and`), `on` (`c1`) y otras, para describir el funcionamiento del circuito.
2. Usando las fórmulas que describen el circuito y suponiendo que que todos los componentes funcionan correctamente y que `c1` y `c2` están en `on`, utilice el principio de resolución para demostrar que la salida `c4` está en `off`.
3. Finalmente, suponga que las entradas `c1` y `c2` están en `on` y que la salida `c4` está en `off`. Usando las fórmulas que describen el funcionamiento del circuito, utilice el principio de resolución para demostrar que la puerta `and` o la `not` están funcionando incorrectamente.

Ejercicio 8.9 Formalice los siguientes enunciados y preguntas en forma clausal:

- Todo hombre tiene algún amigo.
- Juan ¿tiene amigos?
- Todo individuo de un dominio de discurso está incluido en algún conjunto.
- Dos conjuntos son diferentes si no comparten los mismos elementos.
- ¿Existe un conjunto que incluya a todos los demás?

Allí donde proceda, señale las variables extra y compruebe que su aparición se corresponde con la presencia de cuantificación existencial en las fórmulas de la lógica de predicados de las que proceden.

Ejercicio 8.10 ¿Qué hay de erróneo en el siguiente argumento presentado por Henle en 1965?

Los hombres están dispersos por toda la tierra. Sócrates es un hombre. Por consiguiente, Sócrates está disperso por toda la tierra.

Elija un modo representación en la lógica que ponga de manifiesto el problema y resuelva esta aparente paradoja.

Ejercicio 8.11 Consideremos la base de datos del Ejercicio 6.18, que estructuraba como un hecho la información sobre una familia:

```

familia(persona(antonio, foix,
                fecha(7, febrero, 1950), trabajo(renfe, 1200)),
         persona(maria, lopez,
                fecha(17, enero, 1952), trabajo(sus_labores, 0)),
         [persona(patricia, foix,
                fecha(10, junio, 1970), trabajo(estudiante, 0)),
          persona(juan, foix,
                fecha(30, mayo, 1972), trabajo(estudiante, 0))]).

```

1. Complete la base de datos con nuevas entradas y represente el conocimiento almacenado, utilizando ahora relaciones binarias. (**Ayuda:** Emplear el concepto de red semántica puede facilitar esta tarea.)
2. Responda a las siguientes preguntas, formulando los objetivos correspondientes:
 - a) Encontrar los nombres y apellidos de las mujeres casadas que tienen tres o más hijos.
 - b) Encontrar los nombres de las familias que no tienen hijos.
 - c) Encontrar los nombres de las familias en las que la mujer trabaja pero el marido no.
 - d) Encontrar los nombres de todas las personas en la base de datos.
 - e) Encontrar los hijos nacidos después de 1980.
 - f) Encontrar todas las esposas empleadas.
 - g) Encontrar los nombres de las personas desempleadas nacidas antes del año 1960.
 - h) Encontrar las personas nacidas después de 1950 cuyo salario esté comprendido entre 600 y 1000 euros.

Ejercicio 8.12 ([81]) Suponiendo que se dan los datos de las tablas de proveedor, pieza y suministro:

proveedor	num_proveedor	nombre	provincia	ciudad

pieza	num_pieza	nombre	color	peso

suministro	num_suministro	num_pieza	cantidad

1. Complete la base de datos con un número relevante de entradas y represente el conocimiento almacenado, utilizando: a) relaciones n-arias; b) relaciones binarias.

2. Responda a las siguientes preguntas, formulando los objetivos correspondientes, primero en la representación con relaciones n-arias y después en la representación con relaciones binarias:

- a) ¿Cuáles son los proveedores de tuercas?
- b) ¿Cuáles son las ciudades de los proveedores de tuercas?
- c) ¿Cuáles son los nombres de las piezas que proporciona cada proveedor?
- d) ¿Cuáles son los nombres de los proveedores instalados en Madrid que suministran tuercas que pesan más de 200 gramos?
- e) ¿Cuáles son los nombres de los proveedores de tuercas y pernos?
- f) ¿Cuáles son los nombres de los proveedores de tuercas o pernos?

Ejercicio 8.13 (coloreado de mapas) Considere el problema de colorear un mapa de modo que ninguno de los países vecinos tenga el mismo color. Seleccione las estructuras de datos apropiadas y dé una solución para este problema.

Ejercicio 8.14 Estudie el comportamiento del predicado `equiv` definido en el Apartado 8.4 y enumere sus posibles usos. Plantee una pregunta al sistema Prolog capaz de generar todas las permutaciones de una lista de elementos.

Ejercicio 8.15 (Tablero de ajedrez recortado) Sea un tablero de ajedrez, donde se ha suprimido dos casillas en los vértices opuestos. Se pretende cubrir todas las casillas del tablero utilizando fichas de dominó. Con cada ficha se cubrirán dos casillas. Está prohibido que las fichas se solapen o salgan del tablero. Seleccione una representación que facilite la solución de este problema.

Cuestión 8.16 ¿Qué es una red semántica?

Cuestión 8.17 Complete las siguientes afirmaciones relativas a una red semántica:

- Representan un modelo psicológico de ...
- Permiten implementar la ...
- Una red semántica es un grafo dirigido constituido por nodos y por arcos. Los arcos representan ...
- Facilitan el acceso a todas las informaciones relevantes sobre ...

Cuestión 8.18 Indique cuál de las siguientes afirmaciones relativas a una red semántica es falsa:

- Los atributos “instancia” indican que un objeto pertenece a una clase.

- Es un mecanismo de acceso a las informaciones menos costoso que el de la unificación.
 - Es un grafo dirigido constituido por nodos y por arcos. Los nodos representan acciones genéricas (eventos) o valores.
 - Facilitan el acceso a todas las informaciones relevantes debido a su estructuración en una jerarquía arbórea.

Ejercicio 8.19 Dado el siguiente conjunto de enunciados:

Juan le dio un libro a María. El libro, cuyo autor era "J.R.R. Tolkien" y estaba editado por Minotauro, tenía por título "El Señor de los Anillos". Juan había pagado 30 euros por este libro.

a) Represente el conocimiento expresado en estos enunciados mediante una red semántica. b) Escriba (un fragmento de) un programa Prolog que exprese, mediante relaciones binarias, el conocimiento almacenado en la red semántica anterior.

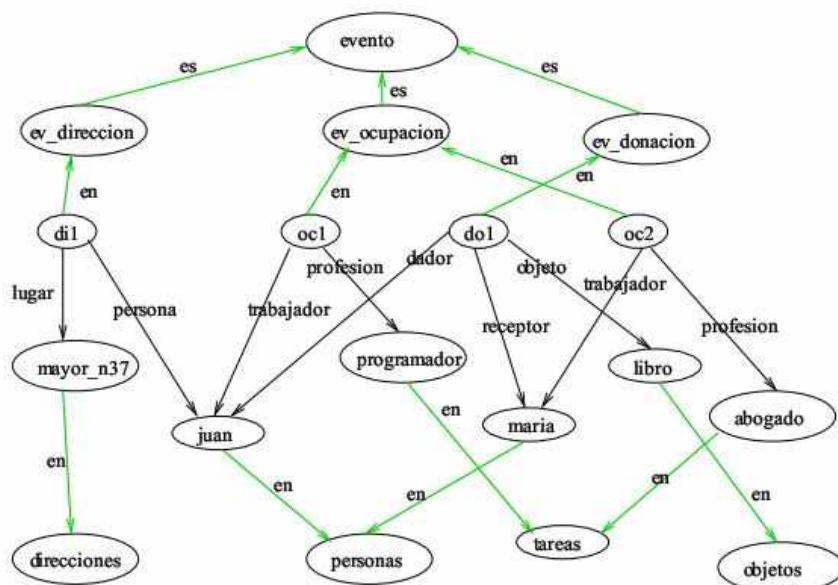


Figura 8.7 Un fragmento de red semántica con informaciones personales.

Ejercicio 8.20 Escriba (un fragmento de) un programa Prolog que exprese, mediante relaciones binarias, el conocimiento (explicito) almacenado en la red semántica de la Figura 8.7 (que aparece en [104]).

Ejercicio 8.21 (Marcos) Un marco es una estructura de datos cuyos componentes se denominan slots y almacenan información sobre objetos o entidades abstractas. Un slot posee un nombre y tiene un valor, que constituyen sus atributos. Los valores pueden ser de varios tipos: valores simples, referencias a otros marcos, o procedimientos que disparan acciones o computan el valor del slot a partir de otras informaciones. Un slot puede que no contenga valor alguno; decimos que está vacío. Los slot vacíos pueden “llenarse” mediante un mecanismo de inferencia basado en la herencia de propiedades, semejante al de las redes semánticas.

Parte del conocimiento sobre pájaros contenido en la red semántica de la Figura 8.4 puede representarse, utilizando marcos, de la forma siguiente:

MARCO: pájaro	MARCO: albatros	MARCO: albert
una_clase_de: animal	una_clase_de: pájaro	instancia_de: albatros
vida: diurna	color: blanco_negro	
metodo_mov: volar		

1. Complete la representación de la Figura 8.4 especificando los marcos asociados al resto de las entidades.
2. Ahora, exprese los marcos mediante hechos del lenguaje Prolog. Más concretamente, utilice el siguiente formato para los hechos:

Nombre_del_marco(Nombre_del_slot, Valor_del_slot).

3. Observe que el slot color está vacío para el marco albert (y, por consiguiente, no aparece en su representación). Sin embargo, su valor puede inferirse, por herencia, del color típico de los albatros. Defina un predicado que implemente la herencia de propiedades en una representación del conocimiento mediante marcos.

Ejercicio 8.22 Considere la red semántica que aparece en la Figura 8.8. a) Amplíe el programa Prolog que implementa la herencia de propiedades para que sea capaz de manejar correctamente el atributo genérico “tiene”, que pone en conexión a una entidad con una de sus partes constituyentes. b) Responda a las preguntas: “¿quién tiene colmillos y caza palomas?”; “¿quién tiene pelo?”

Ejercicio 8.23 Dado el conjunto de hechos:

%% Acto de dar d1	%% Libro l1
instancia(d1, dar).	instancia(l1, libro).
dador(d1, juan).	titulo(l1, 'La ciudad y las estrellas').
objeto(d1, l1).	autor(l1, 'Arthur C. Clarke').
receptor(d1, maria).	

- a) Realice una representación (gráfica) en términos de una red semántica. b) Escriba un programa capaz de responder a la pregunta: ¿Qué tipo de objeto se dio a María en el acto de dar d1?

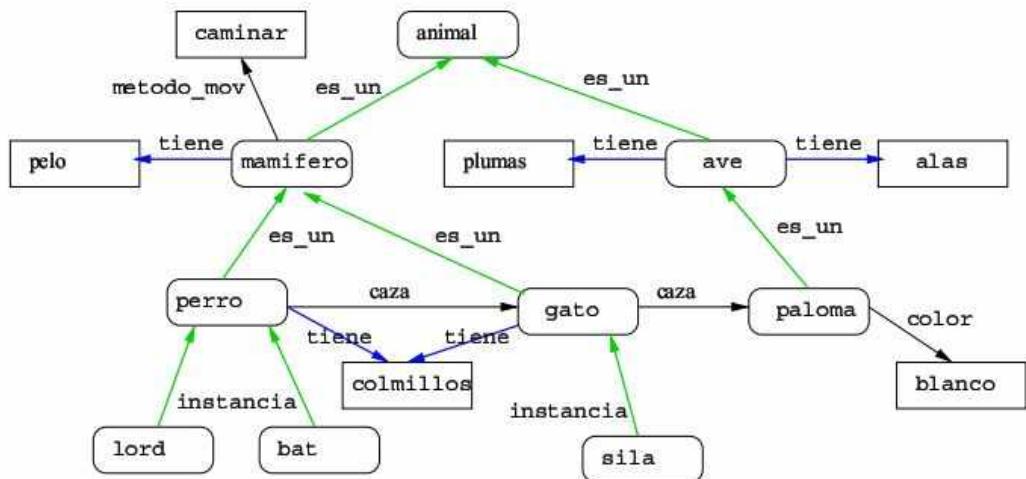


Figura 8.8 Red semántica y a atributo genérico "tiene_parte".

Resolución de problemas

El objetivo de este capítulo es presentar algunas de las aplicaciones de la programación lógica que resultan útiles para la resolución de problemas, lo que nos permitirá profundizar en nuevas técnicas de programación. Al igual que la representación del conocimiento, la resolución de problemas ha sido tema de estudio desde los inicios del área de la inteligencia artificial (IA), ya que una de las características por las que se distingue a un ser inteligente es su capacidad para resolver problemas. En general, para construir un sistema que resuelva un problema específico en un dominio determinado es necesario realizar las siguientes acciones:

1. Definir el problema con precisión. Esta actividad conlleva, entre otras, la tarea de especificar las situaciones iniciales así como las finales, que se identifican como una solución aceptable del problema.
2. Analizar el problema, para identificar características que permitan determinar las técnicas más adecuadas para su resolución.
3. Aislar y representar el conocimiento necesario para la resolución del problema.
4. Elegir las mejores técnicas que resuelvan el problema y aplicarlas.

En el Capítulo 8 nos ocupamos de la tercera de estas acciones. En este capítulo nos vamos a centrar, primordialmente, en las dos primeras y en la cuarta.

La mayoría de los problemas a los que se enfrenta el área de la IA son sumamente complejos y no existe para ellos una solución algorítmica directa. Debido a este hecho, la única forma posible de solución es la aplicación de una técnica genérica que denominamos *búsqueda en un espacio de estados* y a la que se ajustan muchas clases de problemas; por ejemplo, la especificación de un autómata finito implementada en el Apartado 6.3.4. Incluso el propio mecanismo operacional de la programación lógica

puede pensarse que sigue este esquema, donde el cómputo de respuestas se realiza mediante una estrategia de búsqueda a ciegas (es decir, sin emplear información específica sobre la naturaleza del problema que se está resolviendo).

Aunque la búsqueda es un proceso de gran importancia en la resolución de problemas difíciles, para los que no se dispone de técnicas más directas, cuando se dispone de otras técnicas de resolución es mejor evitarla, pues suele ser muy costosa. Por otra parte, las técnicas de búsqueda son un mecanismo muy general que puede proporcionar un marco donde acoplar otras técnicas más directas.

En el próximo apartado estudiamos las principales características de la técnica de búsqueda en un espacio de estados. En apartados sucesivos discutiremos varios ejemplos de cómo se aplica.

9.1 RESOLUCIÓN DEL PROBLEMA MEDIANTE BÚSQUEDA EN UN ESPACIO DE ESTADOS

La representación de un problema como un *espacio de estados* y su solución mediante una búsqueda en ese espacio de estados es una idea fundamental que proviene de los primeros trabajos de investigación en el área de la IA y constituye la base para la mayoría de los métodos de resolución de problemas en la IA. El esquema general de solución de este tipo de problemas responde a la siguiente estructura:

1. Definir formalmente el espacio de estados del sistema: consiste en identificar una noción de estado que recoja las características esenciales del problema (e.g. una configuración de los objetos o actores más relevantes). Identificar uno o varios estados iniciales y estados objetivo. Los estados iniciales son los estados en los que se encuentra el sistema en el momento inicial, son el punto de partida y describen las situaciones en las que comienza el proceso de resolución del problema. Los estados objetivo son estados finales que se consideran soluciones aceptables del problema y proporcionan una condición de parada; una vez alcanzados, el problema se considera resuelto. Existen técnicas que permiten definir el espacio de estados sin tener que enumerar todos sus estados. En los próximos apartados mostraremos algunos ejemplos.
2. Definir las reglas de producción, operación o movimiento: un conjunto de reglas que permiten transformar el estado actual y pasar de un estado a otro cuando se cumplen ciertas condiciones.
3. Especificar la estrategia de control: es importante saber si una regla es aplicable a un estado o no lo es. Dado que, en general, se puede aplicar más de una regla a un estado (aunque en ocasiones ninguna) también es importante decidir qué reglas aplicar y el orden en el que aplicarlas. Estas decisiones tienen un gran impacto en el rendimiento del sistema y son asumidas por la estrategia de control, que dirige

la búsqueda. Además de decidir sobre la aplicabilidad de las reglas, el sistema de control debe comprobar si se cumplen las condiciones de terminación y registrar las reglas que se han ido aplicando. Una estrategia de control debe cumplir los siguientes requisitos: i) que cause algún cambio de estado (necesidad de *cambio local*); ii) que sea sistemática y evite la obtención de secuencias de estados redundantes (necesidad de *cambio global*).

En el esquema general que acabamos de definir, los dos primeros componentes especifican formalmente el espacio de estados. Podemos pensar en este espacio de estados (conceptual) como un árbol (o, más generalmente, como un grafo dirigido) en el que los nodos son estados del problema y los arcos están asociados a las reglas u operaciones (Figura 9.1). Partiendo de un estado inicial, cada operación (aplicable) genera un nuevo

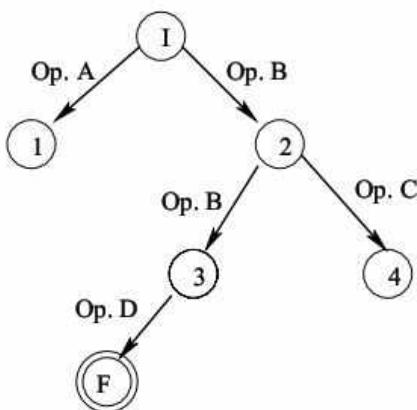


Figura 9.1 Un árbol representando el espacio de estados de un problema.

estado hijo. Ahora la *solución* de un problema consiste en encontrar un camino, en el espacio de estados, que nos lleve desde el estado inicial a un estado objetivo. El camino puede representarse mediante una secuencia de estados que componen dicho camino o la secuencia de operaciones o acciones que permiten pasar desde el estado inicial al estado final. Por consiguiente, la búsqueda en un espacio de estados tiene un papel primordial en la resolución de problemas de IA. El componente de control determinará, entre otros cometidos, el orden en el que se realiza esa búsqueda. Finalmente, también es importante notar que, en algunos problemas, la solución puede consistir en encontrar un estado objetivo (final) sin que sea relevante el camino por el que se ha alcanzado.

Observación 9.1 *Ciertas generalizaciones de formalismos de computación, conocidas como sistemas de producción, establecen una clara separación entre los componentes de una computación (datos, operaciones y control) y parecen encerrar la esencia del funcionamiento de muchos sistemas de IA. Los elementos principales de un sistema de*

producción de IA son una base de datos global, un conjunto de reglas de producción y un sistema de control [104].

9.2 RESOLUCIÓN DE PROBLEMAS Y PROGRAMACIÓN LÓGICA

Como ya hemos adelantado, y siguiendo el enfoque más popular, la especificación completa de un problema mediante la técnica de la búsqueda en un espacio de estados se compone de:

- i) la definición del espacio de estados, indicando los estados iniciales y finales;
- ii) la descripción de las reglas movimiento u operación.

El control, aunque parte importante en la resolución del problema, no formaría parte de esta especificación. Como bien sabemos ya, esta separación entre lógica y control es una de las máximas de la programación declarativa. Por consiguiente, los lenguajes de programación lógica, como Prolog, están bien adaptados para la resolución de problemas, permitiendo al experto centrarse en la especificación del problema y dejando (en parte) la estrategia de control al propio mecanismo operacional del lenguaje, lo que hace innecesario (en el mejor de los casos) programar la búsqueda.

En este apartado aprenderemos a especificar la parte declarativa de un problema que puede considerarse de búsqueda en un espacio de estados, mediante un conjunto de cláusulas de Horn. Para lograr este objetivo estudiaremos diversos ejemplos.

9.2.1. Problema del mono y el plátano

El problema del mono y el plátano es un ejemplo sencillo de resolución de problemas. Presentamos a continuación una variante de este problema que consiste en lo siguiente:

Hay un mono situado en la puerta de entrada de una habitación. En medio de la habitación cuelga un plátano del techo. El plátano está lo bastante alto como para que no pueda alcanzarse desde el suelo (por mucho que el mono salte). Al lado de la ventana, en el lado opuesto de la habitación, hay una caja que puede utilizar el mono. El mono puede realizar las siguientes acciones: andar por la habitación, empujar la caja (y trasladarla de un sitio a otro de la habitación), subirse a la caja y coger el plátano si la caja esta directamente debajo del plátano. ¿Puede el mono coger el plátano?

Damos una solución a este problema siguiendo fielmente los pasos marcados por la técnica de resolución de problemas mediante búsqueda en un espacio de estados: primero, identificamos la noción de estado que utilizamos para representar el espacio

de búsqueda, junto con la descripción de los estados iniciales y los estados objetivo (o finales); después, especificamos el conjunto de reglas de producción o movimientos; finalmente, especificamos el axioma de estados, que activa la búsqueda de soluciones dejando el control al sistema Prolog.

El concepto de estado para este problema se puede representar mediante un término $s(P_m, S_m, P_c, T_m)$, donde los argumentos indican:

- P_m : el mono está en la posición P_m ;
- S_m : el mono está sobre el suelo (constante `suelo`) o sobre la caja (constante `encima`);
- P_c : la caja está en la posición P_c ;
- T_m : el mono tiene o no tiene el plátano;

Abstraemos los posibles valores de las posiciones mediante un conjunto de constantes: estar al lado de la puerta, representado por la constante `puerta`; estar al lado de la ventana, representado por la constante `ventana` y estar en el centro de la habitación, representado por la constante `centro`. Una variable indicará cualquier otra posición.

Las acciones o movimientos posibles se especifican mediante los siguientes términos y constantes:

- `andar(P1, P2)`: el mono anda de la posición P_1 a la P_2 .
- `empujar(P1, P2)`: el mono empuja la caja de la posición P_1 a la P_2 .
- `subir`: el mono sube encima de la caja.
- `coger`: el mono coge el plátano.

Observe que `andar`, `empujar`, ... son meros símbolos constructores, no definidos por reglas, de manera que los términos que puede formarse con ellos juegan el papel de estructuras de datos que especifican informaciones sobre situaciones relativas al problema.

En la solución propuesta se han definido dos predicados, que se describen a continuación:

- `possible(s)`: Indica que es posible el estado s .
- `move(S1, M, S2)`: Permite pasar de un estado a otro, cuando realizamos una acción o movimiento; el significado de este predicado es que el movimiento M hace que el estado S_1 cambie al estado S_2 . Este predicado, junto con la noción de estado introducida anteriormente, define implícitamente el espacio de estados del problema.

Agrupando todos estos elementos, construimos el programa Prolog capaz de resolver este problema:

```
% El estado inicial es posible
possible(s(puerta, suelo, ventana, no tiene)).  
  
% Acciones y movimientos.
move(s(P1, suelo, P, T), andar(P1, P2), s(P2, suelo, P, T)).
move(s(P1, suelo, P1, T), empujar(P1, P2), s(P2, suelo, P2, T)).
move(s(P, suelo, P, T), subir, s(P, sobre, P, T)).
move(s(centro, sobre, centro, no tiene),
     coger, s(centro, sobre, centro, tiene)).  
  
% Axioma del espacio de estados.
possible(S2) :- possible(S1), move(S1, M, S2).
```

Conviene notar que las acciones y movimientos reflejan las restricciones del problema: el mono puede *andar* de una posición *P1* a una posición *P2* (primer hecho) o bien cambiar de posición la caja empujándola (segundo hecho); el mono puede *subir* a la caja si está en el *suelo* y al lado de la caja —es decir, ambos están en la misma posición *P* de la habitación— (tercer hecho); finalmente, el mono puede *coger* el plátano, pasando a un estado en el que *tiene* el plátano, si está en el *centro* de la habitación y *sobre* la caja (cuarto hecho).

El axioma del espacio de estados afirma que el estado *S2* es posible si el estado *S1* es posible y existe una acción o movimiento *M* que permite pasar del estado *S1* al *S2*.

El estado final puede representarse mediante el objetivo “?- *possible*(_, _, _, tiene).”, es decir, cualquier estado en el que el mono tiene el plátano. También se puede preguntar por todos los estados posibles planteando el objetivo “?- *possible*(S).” y por todas las acciones posibles mediante el objetivo “?- *move*(_, A, _).”.

9.2.2. Problema de los contenedores de agua

El problema de los contenedores de agua suele enunciarse en los términos siguientes:

Supongamos que tenemos un depósito con agua suficiente y dos contenedores de agua de 7 y 5 litros, inicialmente vacíos. Encontrar una secuencia de acciones que dejen 4 litros de agua en el contenedor de 7 litros. Las acciones legales son: llenar un contenedor, tomando agua del depósito; vaciar un contenedor en el depósito; trasvasar agua de un contenedor a otro (hasta que el contenedor desde el que se trasvaza esté vacío); trasvasar agua de un contenedor a otro (hasta que el contenedor al que se trasvaza esté lleno).

Nuevamente, se resuelve el problema mediante una búsqueda en un espacio de estados. La noción de estado se puede representar como un término *est(C7, C5)* (un vector de dos posiciones), en el que la variable *C7* representa los litros de agua almacenados en el contenedor de siete litros y la variable *C5* representa los litros de agua almacenados en el contenedor de cinco litros.

El problema de los contenedores de agua tiene una formulación sencilla usando el lenguaje Prolog:

```

% Se introduce el operador ">=" para indicar el sentido del
% trasvase de agua
:- op(800, xfx, =>).

% ACCIONES.
% Llenar un contenedor.
mov(est(X, Y), llenar(c7, litros(Z)), est(7, Y))
    :- X < 7, Z is 7-X.
mov(est(X, Y), llenar(c5, litros(Z)), est(X, 5))
    :- Y < 5, Z is 5-Y.

% Vaciar un contenedor.
mov(est(X, Y), vaciar(c7, litros(X)), est(0, Y)) :- X > 0.
mov(est(X, Y), vaciar(c5, litros(Y)), est(X, 0)) :- Y > 0.

% Trasvase de agua de un contenedor a otro
% (hasta que el contenedor desde el que se trasvaza este vacio):
% Del contenedor de 7 litros al de 5.
mov(est(X, Y), trasvasar(vaciar_c7=>en_c5, litros(X)), est(0, V))
    :- 0 =< X, V is (X + Y), V =< 5.
% Del contenedor de 5 litros al de 7.
mov(est(X, Y), trasvasar(vaciar_c5=>en_c7, litros(Y)), est(U, 0))
    :- 0 =< Y, U is (X + Y), U =< 7.

% Trasvase de agua de un contenedor a otro
% (hasta que el contenedor al que se trasvaza este lleno):
% Del contenedor de 7 litros al de 5.
mov(est(X, Y), trasvasar(c7=>llenar_c5, litros(C)), est(U, 5))
    :- C is (5 - Y), C =< X, U is (X - C).
% Del contenedor de 5 litros al de 7.
mov(est(X, Y), trasvasar(c5=>llenar_c7, litros(C)), est(7, V))
    :- C is (7 - X), C =< Y, V is (Y - C).

% AXIOMA DEL ESPACIO DE ESTADOS y formacion del plan.
plan(est(0, 0), []). % ESTADO INICIAL.
plan(E2, [A|Resto_plan]) :- plan(E1, Resto_plan), mov(E1, A, E2).

% OBJETIVO. (el plan se forma en orden inverso)
objetivo(P) :- plan(est(4, _), P).

```

Los términos que caracterizan las acciones posibles (`llenar`, `vaciar` y `trasvasar`) se han construido usando constructores binarios, de forma que los valores de sus argumentos describan lo más claramente posible las acciones que se realizan. El segundo argumento de esos términos muestra los litros con que se llena un contenedor, los litros que se vacían y los litros que se trasvasan, respectivamente.

A diferencia del problema del Apartado 9.2.1, en este problema la solución buscada es un camino (representado como una secuencia de acciones –movimientos–).

En [81] se presenta una solución al problema de los contenedores que, cuando se traduce literalmente al lenguaje Prolog, no es capaz de encontrar una solución. Deben

realizarse modificaciones para poder obtener respuestas a los objetivos planteados. Dos diferencias destacables, respecto a la aproximación seguida en este apartado, es que el espacio de estados se representa mediante átomos (en vez de términos) y la búsqueda de una solución se formula como el problema de hallar un camino.

9.2.3. Generación de planes en el mundo de los bloques

El problema de la generación de planes en el mundo de los bloques consiste en encontrar una secuencia de acciones (plan) que permita reordenar una pila de bloques, para pasar de una situación inicial a otra final, en la cual los bloques están ordenados de la forma deseada (en nuestro caso particular, como se muestra en la figura 9.2). Deben cumplirse las siguientes restricciones: solo puede moverse un bloque cada vez; un bloque puede cogerse si sobre él no hay otro bloque; un bloque puede situarse sobre otro bloque (pero no sobre sí mismo) o sobre ciertas posiciones de la mesa. Por simplicidad, solamente admitimos la posibilidad de colocar los bloques sobre tres posiciones distintas, que denotaremos más adelante como: p_1 , p_2 , y p_3 .

A la hora de resolver este problema conviene tener en cuenta las siguientes consideraciones:

1. *Representación del espacio de estados y estados objetivo.*

Representaremos un estado (situación) como una lista de pilas de bloques¹. Las pilas de bloques serán a su vez listas. Por ejemplo, si el estado inicial consiste en que el bloque “c” está sobre el “a” y éste sobre el “b” y éste sobre la posición “ p_1 ”, mientras que las posiciones “ p_2 ” y “ p_3 ” están vacías (como muestra la Figura 9.2), la representación será:

$[[c, a, b],$	$[],$	$[]]$	
-----	--	--	estamos suponiendo que la lista de bloques 1
p_1	p_2	p_3	está sobre la posición “ p_1 ” etc.

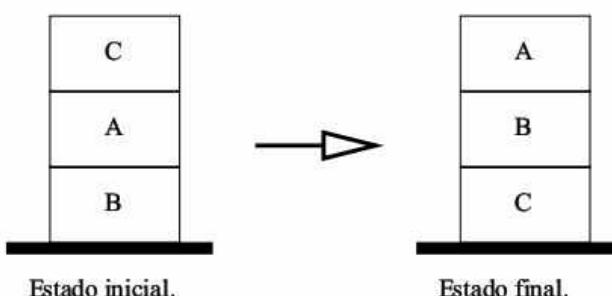


Figura 9.2 Reordenación en el mundo de los bloques.

¹Para este problema, se han propuesto otras formas (posiblemente más expresivas) de representar un estado. Por ejemplo, véase [105].

Por otra parte, consideramos que, cualquiera de los siguientes, son estados objetivo:

```
[[a,b,c],[],[]] o [[], [a,b,c], []] o [[],[],[a,b,c]]
```

es decir, admitimos cierta flexibilidad a la hora de disponer la configuración final (y también la inicial) sobre las posiciones p1, p2 o p3.

2. Búsqueda en el espacio de estados.

Queremos calcular un plan, esto es, buscamos un camino solución desde un estado inicial a un estado objetivo; por tanto, una solución natural a este problema se puede representar como una lista de estados². Partiendo de un estado inicial se generarán (mediante una relación “sucesor” actuando sobre estados) una secuencia de estados:

$$E_0 \rightarrow E_1 \rightarrow E_2 \rightarrow \dots \rightarrow E_n.$$

La generación de estados sucesores terminará cuando alcancemos un estado objetivo, lo que viene caracterizado por la relación:

```
objetivo(E) :- member([a,b,c], E).
```

La búsqueda será propiciada por una relación, “resolver” que, actuando sobre un estado inicial que se le pasará como parámetro, devolverá una lista solución en su segundo parámetro. La relación `resolver(Estado, Solucion)` se especifica en los siguientes términos:

- Si Estado es un objetivo entonces Solucion = [Estado], 0
- Si existe un estado, Estado1, sucesor de Estado y hay un camino solución, Solucion1, desde Estado1 a un estado objetivo, entonces Solucion = [Estado | Solucion1].

Resolveremos un problema planteando una pregunta al sistema

```
?- resolver(Inicial, Solucion).
```

donde Inicial será una configuración que defina un estado inicial³, por ejemplo: `[] , [c,b,a], []`. De forma más general, una variable representando un estado inicial desde el que puede alcanzarse un estado objetivo.

Agrupando lo dicho hasta el momento, la solución propuesta para el problema de la generación de planes en el mundo de los bloques es la siguiente:

²Observe que el plan también podría estar constituido por una secuencia de acciones, como en el ejemplo de los contenedores de agua.

³Esto difiere del problema del mono y el plátano o de los contenedores de agua, en los que el predicado definido por el axioma del espacio de estados toma como argumento un estado final.

```
% Acciones o Movimientos.  
%  
% sucesor(Estado2, Estado1): el Estado2 es un sucesor del Estado1  
% si, por ejemplo, en el Estado2 hay una pila, [Top1|pila2], cuya  
% cabeza, Top1 era la cima de una pila [Top1|pila1] del Estado1 y  
% que ha pasado a ser la cima de una pila2 del Estado1.  
%  
% Los siguientes constituyen movimientos legales.  
sucesor([Pila1, [Top1|Pila2], Pila3], [[Top1|Pila1], Pila2, Pila3]).  
sucesor([Pila1, Pila2, [Top1|Pila3]], [[Top1|Pila1], Pila2, Pila3]).  
  
sucesor([[Top2|Pila1], Pila2, Pila3], [Pila1, [Top2|Pila2], Pila3]).  
sucesor([Pila1, Pila2, [Top2|Pila3]], [Pila1, [Top2|Pila2], Pila3]).  
  
sucesor([[Top3|Pila1], Pila2, Pila3], [Pila1, Pila2, [Top3|Pila3]]).  
sucesor([Pila1, [Top3|Pila2], Pila3], [Pila1, Pila2, [Top3|Pila3]]).  
  
% Estado objetivo.  
objetivo(Estado) :- member([a,b,c], Estado).  
  
% Axioma del espacio de estados:  
resolver(E, [E]) :- objetivo(E).  
resolver(E, [E|Sol1]) :- sucesor(E1, E), resolver(E1, Sol1).
```

9.3 ESTRATEGIAS DE BÚSQUEDA Y CONTROL

La estrategia de control dicta el orden en el que se aplican las reglas de producción o movimientos para generar nuevos estados. Esta circunstancia determina el orden en el que se generan dichos estados y, por lo tanto, la geometría del espacio de estados. Una parte muy importante del control es la estrategia de búsqueda, que especifica el orden en el que se visitan los estados generados en busca de un estado objetivo. Otras tareas adicionales, aunque no menos importantes, de la estrategia de control incluyen: comprobar la aplicabilidad de las reglas de producción, la comprobación de que se cumplen ciertas condiciones de terminación y registrar las reglas o acciones que han sido aplicadas.

Es evidente que el árbol (o, más generalmente, el grafo) que representa el espacio de estados podría generarse completamente aplicando las reglas de producción, tal y como hemos sugerido más arriba. Sin embargo, en la práctica, esto no suele hacerse así. En lugar de construir explícitamente el árbol de búsqueda y luego buscar en él, la mayoría de los sistemas de control representan implícitamente el espacio de estados mediante las reglas de producción o movimientos y solamente se generan explícitamente aquellas partes por las que avanza la búsqueda. Por consiguiente, puede hablarse de que es la propia estrategia de búsqueda (incorporando en ella el resto de los componentes de control) la que genera (parcialmente) el árbol de búsqueda. Durante el estudio que se realiza de los métodos de búsqueda, es conveniente que el lector tenga en mente la

distinción entre árboles de búsqueda implícitos y (parcialmente) los explícitos, que son en realidad los que utiliza la estrategia de búsqueda.

Por otra parte, es importante advertir que muchas de las discusiones y comentarios que realizamos sobre estrategias de búsqueda se limitan a algoritmos de exploración para árboles, ya que los espacios de estados de algunos de los ejemplos que estamos considerando se ajustan a una estructura arborescente. Sin embargo, los programas de búsqueda que implementaremos en los próximos apartados también pueden funcionar cuando la estructura del espacio de estados es un grafo (si bien puede duplicarse el trabajo cuando se accede a un mismo estado por diferentes caminos). Observe que un procedimiento de búsqueda sobre un árbol que mantenga información sobre los nodos ya generados puede transformarse en un procedimiento de búsqueda sobre un grafo sin más que modificar la acción que se realiza cada vez que se genera un nodo. Por ejemplo, si uno de los sucesores s de un nodo n ha sido generado con anterioridad (es decir, s aparece en la lista de nodos ya generados), en lugar de insertar el nuevo nodo, se establece un enlace (“hacia atrás”) desde n a la primera aparición del nodo s (véase la Figura 9.3).

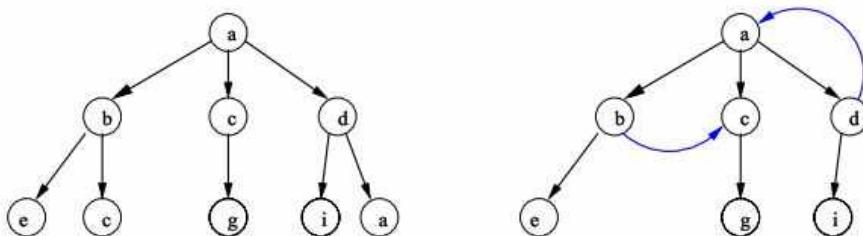


Figura 9.3 Conversión de un árbol de búsqueda en un grafo.

Otro aspecto importante relativo a las estrategias de control es la dirección en la que se realiza la búsqueda. Alternativamente, se puede buscar partiendo del estado inicial hasta llegar al estado objetivo y entonces se habla de *razonamiento hacia adelante* (la solución del problema de planificación en el mundo de los bloques es un ejemplo de este tipo de razonamiento). Se puede buscar partiendo del estado objetivo hasta llegar a un(os) estado(s) inicial(es) y entonces se habla de *razonamiento hacia atrás* (las soluciones dadas para los problemas del mono y el plátano y los contenedores de agua son ejemplos de este tipo de razonamiento).

En lo que resta de este apartado nos centraremos en el estudio de algunas estrategias de búsqueda básicas, principalmente desde perspectiva de su implementación mediante el lenguaje de programación Prolog. Estas estrategias (como la mayoría de las estudiadas en este capítulo), caen dentro del ámbito de las técnicas de control tentativo, en las que es posible reconsiderar, con posterioridad, la aplicación de una regla que se seleccionó y aplicó en un punto y un momento anterior. Para ello se toman medidas precisas, de forma que se pueda volver a ese punto del proceso y aplicar entonces otra regla en la misma forma.

9.3.1. Estrategia de búsqueda en profundidad

Como se ha explicado en los Apartados 5.2.3 y 6.2.2, la principal característica de la estrategia de búsqueda en profundidad es que el espacio de estados se explora de forma que se visitan los descendientes de un estado antes de visitar cualquier otro nodo (hermano). Esto hace que, si el espacio de búsqueda es un árbol, se recorra primero la rama más a la izquierda⁴ antes de recorrer el resto de las ramas.

El mecanismo operacional del lenguaje Prolog utiliza este tipo de búsqueda, por lo que estamos adoptando implícitamente una estrategia de búsqueda en profundidad cuando dejamos control al propio mecanismo operacional del lenguaje para evitar programar explícitamente la búsqueda, como en los problemas que hemos resuelto el Apartado 9.2.

Por ejemplo, las reglas correspondientes al axioma del espacio de estados y al estado objetivo del problema de la generación de planes en el mundo de los bloques, descrito en el Apartado 9.2.3:

```
resolver(E, [E]) :- objetivo(E).
resolver(E, [E|Sol1]) :- sucesor(E1, E), resolver(E1, Sol1).
```

pueden verse como una implementación en Prolog de la estrategia de búsqueda en profundidad.

La estrategia de búsqueda en profundidad suele funcionar bien para ciertos problemas, como sucede con el del mono y el plátano (Apartado 9.2.1) o con el de los contenedores de agua (Apartado 9.2.2) que proporcionan una solución. Sin embargo, puede conducir a graves dificultades dependiendo de la geometría del espacio de búsqueda que se recorre. Por ejemplo, la solución presentada para el problema de la generación de planes en el mundo de los bloques, en el Apartado 9.2.3, no es efectiva. El lector puede constatar que se produce un desbordamiento de la pila local del sistema Prolog, cuando se formula la pregunta: “?- resolver([[c,a,b],[],[]], Plan).”, con el fin de obtener un plan (una secuencia de configuraciones, en este caso) que nos lleve desde la configuración inicial [[c,a,b],[],[]] a una configuración objetivo. La razón de este mal comportamiento es que durante la búsqueda de la solución se generan muchos estados (configuraciones) redundantes que obligan a realizar nuevas llamadas recursivas y, por lo tanto, a un crecimiento incontrolado de la pila local que almacena los registros de activación (entornos) de esas llamadas.

Una forma simple de resolver el problema consiste en introducir un medio para detectar la generación de estados redundantes.

```
% Busqueda en profundidad con deteccion de ciclos.
% resolver(E,Sol):
% Resolver es verdadero si existe un plan que, partiendo del
% estado inicial E, nos lleva a un estado objetivo.
% Por simplicidad, el parametro Sol almacena el plan
```

⁴Supuesto un régimen en el que seleccionan de izquierda a derecha los descendientes del estado.

```
% en orden inverso.
resolver(E,Sol) :- resolver([], E, Sol).

% resolver(Camino_ant, E, Sol)
% El parametro Camino_ant almacena la secuencia de estados
% visitados hasta el momento: E es el estado que estamos
% visitando; Sol devuelve la solucion en forma de camino
% [E|Camino_ant], cuando E es un estado objetivo.
resolver(Camino_ant, E,[E|Camino_ant]) :- objetivo(E).
resolver(Camino_ant, E, Sol1) :-
    sucesor(E1, E), not(member(E1, Camino_ant)),
    resolver([E|Camino_ant], E1, Sol1).
```

En un espacio de estados con estructura arborescente, la aparición de estados redundantes suele estar ligada a la existencia de ramas infinitas. Si el espacio de estados es un grafo, los estados redundantes están asociados a la existencia de ciclos dentro del grafo. La solución anterior permite la detección de ciclos en un grafo cuando se realiza una búsqueda en profundidad.

Observación 9.2 *Cuando el espacio de búsqueda es infinito, el mecanismo de búsqueda en profundidad con detección de ciclos todavía puede perderse en caminos infinitos, lo que podría impedir que alcancase otros estados objetivo existentes.*

Para evitar caminos acíclicos infinitos, una solución más bien grosera consiste en imponer un límite a la profundidad máxima que puede alcanzarse al recorrer una rama o camino (véase el Problema 9.13). Otro método más sofisticado para asegurar la terminación de la búsqueda de soluciones consiste en asociar al espacio de estados un orden bien fundado (o un buen preorden) que permita distinguir cuando un estado E es "mayor o igual" que un cierto estado E' , lo que denotamos por $E \geq E'$. Por ejemplo, podemos ordenar los estados en función de la talla o número de símbolos de alguno de sus argumentos cuando dicho argumento decrezca en cada llamada recursiva. La característica esencial de este tipo de relaciones matemáticas es que no permite la existencia de secuencias infinitas E_1, \dots, E_n, \dots tales que $E_i < E_{i+1}$ para todo $i \geq 1$. Por consiguiente, puede diseñarse un procedimiento de búsqueda que, dado un nodo E del espacio de estados, detenga la búsqueda por ese camino siempre que uno de sus estados sucesores E' , originado en la última expansión, cumpla que $E' \geq E$. La propiedad de ser un orden bien fundado nos asegura que esta condición siempre se alcanzará. Observe, sin embargo, que no siempre es posible asociar un orden bien fundado a un espacio de estados.

La estrategia de búsqueda en profundidad es muy eficiente desde el punto de vista del consumo de memoria, debido a que solo almacena información de la rama o camino que se está recorriendo. Sin embargo, no es óptima, si lo que importa es encontrar los caminos de longitud mínima. La estrategia de búsqueda en amplitud, que estudiamos en el próximo apartado, garantiza que primero se hallarán los caminos solución más cortos.

9.3.2. Estrategia de búsqueda en anchura

La principal característica de la estrategia de búsqueda en anchura es que el espacio de estados se explora de forma que se visitan los hermanos de un estado antes de visitar a sus descendientes. Esto hace que, si el espacio de búsqueda es un árbol, se recorra éste por niveles. En los apartados 5.2.3 y 6.2.2 se ha introducido esta estrategia para el caso particular de la exploración de un árbol SLD en busca de una rama de éxito. En este apartado presentaremos un planteamiento más general.

Para describir con mayor detalle el funcionamiento de la estrategia de búsqueda en anchura, supondremos un espacio de estados genérico y la existencia de un procedimiento capaz de expandir todos los sucesores de un nodo estado; es decir, de hallar todos los estados que pueden obtenerse a partir de uno dado aplicando las reglas de producción o movimiento para el problema que se está considerando.

Algoritmo 9.1 [Exploración en anchura]

Entrada: El estado inicial *I*.

Salida: Una condición de éxito o de fallo.

Comienzo

Inicialización: ListaCandidatos = [*I*];

Repetir

1. EstadoActual = cabeza(ListaCandidatos);
2. ListaCandidatos = cola(ListaCandidatos);
3. Si EstadoActual es un objetivo entonces

Comienzo

Devolver éxito;

 Si se desean más respuestas entonces continuar

 si no terminar;

Fin

4. Si no Comienzo

 NuevosCandidatos = expandir(EstadoActual);

 ListaCandidatos = ListaCandidatos ++ NuevosCandidatos;

Fin

Hasta que ListaCandidatos = [];

Devolver fallo.

Fin

Un algoritmo de exploración en anchura debe de mantener una lista de nodos candidatos que están por inspeccionar. Se selecciona uno de ellos⁵ para comprobar si es o no un estado objetivo. Si no es un estado objetivo se expande y sus sucesores se sitúan

⁵Por simplicidad, en el Algoritmo 1, se ha elegido el primero de la lista de candidatos, pero se podría haber aplicado cualquier otro criterio más apropiado, basado en el conocimiento específico que se posee sobre el problema que se está resolviendo. Este modo de proceder da lugar a un tipo de búsqueda que se denomina *heurística* en [105] (véase el Apartado 9.4.3 más adelante).

al final de la lista de candidatos para tratarse con posterioridad. En el Algoritmo 1 las funciones “cabeza” y “cola” operan sobre listas, devolviendo la cabeza y la cola de una lista respectivamente. El operador “++” concatena dos listas.

Entre las ventajas de la estrategia de búsqueda en anchura destacan las siguientes:

1. Si el espacio de estados posee un camino entre un estado inicial y un estado objetivo, está garantizado que estrategia terminará encontrándolo y no quedará atrapada explorando ramas infinitas en un árbol o caminos cílicos en un grafo.
2. Garantiza que la solución, de existir, se encontrará en un número mínimo de pasos. Esto es debido a que no se exploran las rutas de $n + 1$ pasos antes de haber explorado todas las de n pasos; es decir, no puede explorarse una ruta “larga”, que conduzca a una solución, antes de haber examinado todas las de menor longitud, que pueden conducir a una solución.

En cambio, entre las desventajas más graves de esta estrategia podemos citar que requiere almacenar todos los estados del nivel que se está explorando o, aún peor, si se desea proporcionar un plan como solución del problema, debe almacenar todos los (fragmentos iniciales de los) caminos explorados hasta el momento. Por lo tanto, necesita consumir más memoria que la estrategia de búsqueda en profundidad, que solamente almacena información sobre el camino que se está explorando en cada momento.

La implementación en Prolog del Algoritmo 1 no es difícil pero tiene cierta dificultad técnica, no solamente por la necesaria codificación recursiva, sino también por la necesidad de estar familiarizado con el uso del predicado predefinido `setof` (o `bagof`) que se emplea en la implementación de `expandir`. En lo que sigue mostramos una implementación del anterior algoritmo. Para concretar, consideraremos el caso de la búsqueda de un camino en un grafo como el representado en la Figura 9.4.

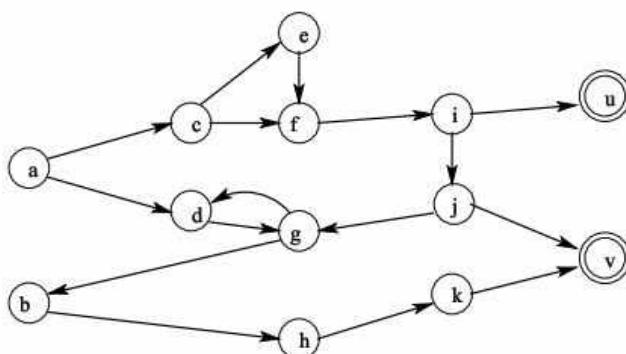


Figura 9.4 Búsqueda de un camino en un grafo dirigido.

```
% REPRESENTACION DEL ESPACIO DE ESTADOS
% Representamos los estados mediante un conjunto de
% constantes: a, b, c, ..., k, u, v. Consideraremos
% que a y b son los estados iniciales y u y v los
% estados objetivo
%
inicial(a).      objetivo(u).
inicial(b).      objetivo(v).

%
% Se introduce el operador "==>" para representar de forma
% intuitiva la relaci\'on sucesor
%
:- op(800, xfx, ==>).

%
% MOVIMIENTOS
%
% X ==> Y:
% Y es un estado sucesor (descendiente) del estado X
%
a ==> c.      d ==> g.      h ==> k.      k ==> v.
a ==> d.      e ==> f.      i ==> j.
b ==> h.      f ==> i.      i ==> u.
c ==> e.      g ==> d.      j ==> v.
c ==> f.      g ==> b.      j ==> g.

%
% expandir(E, Sucesores):
% Verdadero si "Sucesores" es la lista de sucesores del
% estado E
%
expandir(E, Sucesores) :- setof(S, E ==> S, Sucesores), !.
expandir(_, []).  %% setof falla porque E no tiene sucesores

%
% BUSQUEDA EN ANCHURA
%
anchura(E):
% verdadero si existe un camino entre el estado inicial E y
% un estado objetivo
%
anchura(E) :- inicial(E), bucle([E]). 

bucle([E|_]) :- objetivo(E).
bucle([E|Candidatos]) :-
    expandir(E, NuevosCandidatos),
    append(Candidatos, NuevosCandidatos, ListaCandidatos),
    bucle(ListaCandidatos).
```

Si existe un camino entre un estado inicial y un estado objetivo, el programa anterior lo encuentra siempre. Sin embargo, esto no asegura una respuesta cuando el grafo contenga ciclos y no haya un camino entre un estado inicial y un estado objetivo. El lector puede comprobar que el programa anterior no termina cuando marcamos el estado *a* como inicial (introduciendo el hecho “*inicial(a)*.”), eliminamos *v* de los estados ob-

jetivo (borrando el hecho “objetivo(v).”) y lanzamos la pregunta “?- anchura(d).” (véase el Problema 9.15).

Para evitar problemas como el mencionado con anterioridad, podemos modificar nuestro programa introduciendo un medio para detectar la generación de estados redundantes producidos por la existencia de ciclos en el grafo que representa el espacio de búsqueda. Para ello es preciso mantener información de los estados generados previamente, lo que puede ser muy costoso en problemas donde la noción de estado es más compleja. En lo que sigue se muestran las adaptaciones de los predicados `expandir` y `anchura` necesarias para alcanzar estos propósitos:

```
% expandir(E, Generados, Sucesores):
% Verdadero si "Sucesores" es la lista de sucesores del
% estado E. El segundo argumento, que contiene una lista de
% estados ya generados, se utiliza para detener el crecimiento
% de ramas con estados redundantes.

expandir(E, Generados, Sucesores) :-
    setof( S, (E ==> S, not(member(S, Generados))), Sucesores), !.
expandir(_, _, []) . %% setof falla porque E no tiene sucesores

% BUSQUEDA EN ANCHURA
%
% anchura(E):
% Verdadero si existe un camino entre el estado inicial E y un
% estado objetivo

anchura(E) :- inicial(E), bucle([E], [E]).

% bucle(Candidatos, Generados):
% Verdadero si la lista de candidatos contiene un estado
% objetivo. El segundo argumento es auxiliar y contiene los
% estados generados hasta el momento.

bucle([E|_], _) :- objetivo(E).
bucle([E|Candidatos], Generados) :-
    expandir(E, Generados, NuevosCandidatos),
    append(Candidatos, NuevosCandidatos, ListaCandidatos),
    append(Generados, NuevosCandidatos, ListaGenerados),
    bucle(ListaCandidatos, ListaGenerados).
```

Observe que la primera solución del problema de la búsqueda de un camino en un grafo despliega dicho grafo convirtiéndolo en un árbol en el que algunos nodos y subramas están repetidos en otras partes del árbol. Sin embargo, en la segunda solución del problema, las acciones de los predicados `expandir` y `anchura` se combinan para impedir la aparición de estados redundantes (ya visitados) en el árbol de búsqueda. Podemos afirmar, de una manera un tanto burda, que el efecto de estos dos predicados es transformar el árbol de búsqueda implícito, para cada pregunta, recreando (parte de) el grafo original (véase el Problema 9.15). Otra manera de entender el efecto de estos predicados

es apreciando que, si partiendo de un estado existen varios caminos que conduzcan a un estado objetivo, solamente se consideran los que no comparten nodos intermedios (los otros posibles caminos son desechados).

Hasta el momento hemos estado interesados en detectar la existencia de un camino que uniese un estado inicial con un estado objetivo. Ahora bien, si nuestro interés no se centra en comprobar su existencia sino en mostrar dicho camino como resultado de la búsqueda, la solución se complica porque debemos de mantener un conjunto de caminos candidatos (en lugar de simplemente estados candidatos). En este caso, el proceso de búsqueda en anchura puede resumirse en los pasos siguientes:

1. Si la cabeza del primer camino es un estado objetivo, devolver ese camino como la solución al problema;
2. si no, eliminar el primer camino del conjunto de candidatos y expandir el estado que está en cabeza para generar todas las extensiones (de un paso) del primer camino y añadirlas al final de la lista de caminos candidatos.

A continuación se muestra un programa que implementa el proceso anterior. La peculiaridad de la nueva solución, respecto a las anteriores es que la lista de candidatos contiene todos los caminos que se están recorriendo, lo que impide la generación de ciclos mediante la detección de ancestros en un camino candidato.

```
% anchura(E, Plan):  
% Verdadero si Plan es un camino entre el estado inicial E y un  
% estado objetivo. El plan se muestra en orden inverso.  
  
anchura(E, Plan) :- inicial(E), bucle([[E]], Plan).  
  
% bucle(Candidatos, Plan):  
% Verdadero si Candidatos (la lista de caminos candidatos) con-  
% tiene un camino con un estado objetivo, que se devuelve como  
% el Plan.  
  
bucle([[E|RestoPlan] | _], [E|RestoPlan]) :- objetivo(E).  
bucle([Camino | Candidatos], Plan) :-  
    expandir(Camino, NuevosCandidatos),  
    append(Candidatos, NuevosCandidatos, ListaCandidatos),  
    bucle(ListaCandidatos, Plan).
```

El predicado auxiliar `expandir` extiende un camino candidato `[E | Camino]` con todas los estados sucesores `s1, s2, ...` para los que existe un movimiento legal `E ==> s1, E ==> s2, ...`; en otras palabras, genera una lista de nuevos caminos candidatos `[[s1, E | Camino], [s2, E | Camino], ...]`. También detiene el crecimiento de los caminos con estados redundantes, impidiendo que un camino `[s, E | Camino]` forme parte de la lista de nuevos caminos candidatos, si el estado sucesor `s` aparece como ancestro en el camino `[E | Camino]` que se está expandiendo.

```
% expandir([E|Camino], NuevosCaminos):
% NuevosCaminos es la lista de nuevos caminos candidatos, que
% resultan de expandir el camino [E|Camino] con los estados
% sucesores del estado E.

expandir([E|Camino], NuevosCaminos) :-
    setof( [S,E|Camino],
           (E ==> S, not(member(S, [E|Camino]))), NuevosCaminos), !.
expandir(_, []). %% setof falla porque E no tiene sucesores
```

Una deficiencia del programa anterior es la ineficiencia de la operación `append` de concatenación de listas. Este problema puede eliminarse mediante el uso de listas diferencia, tal y como se explicó en el Apartado 7.7.

Observación 9.3 Una técnica que combina los beneficios de la búsqueda en profundidad (economía de memoria) con los de la búsqueda en anchura (obtención del camino más corto a un nodo objetivo), es el descenso iterativo [79]. Esta estrategia realiza sucesivas búsquedas en profundidad, aumentando en cada paso la profundidad límite, hasta encontrar un estado objetivo.

Aunque los caminos recorridos se olvidan y deben recomputarse en cada iteración, en la práctica este esfuerzo computacional no afecta muy negativamente la eficiencia de la estrategia, en particular cuando se aumenta la profundidad límite. Sorprendentemente, el número de nodos expandidos durante el proceso de búsqueda no es mucho mayor que para el caso en que se usa una estrategia de búsqueda en anchura.

Un texto que analiza las propiedades formales de esta técnica es [105] y una implementación, utilizando el lenguaje Prolog, puede encontrarse en [21].

9.4 ESTRATEGIAS DE BÚSQUEDA HEURÍSTICA

Las estrategias de búsqueda que hemos estudiado hasta el momento no usan información específica sobre la naturaleza del problema que se está intentando resolver (si exceptuamos algunas restricciones impuestas en las propias reglas de movimiento, que no constituyen parte esencial de esas estrategias pero que inciden sobre la aplicabilidad de las reglas). Esto es, son estrategias de búsqueda *sin información* (también denominadas estrategias de búsqueda *a ciegas*). Como hemos mostrado, estas estrategias se caracterizan por una aplicación sistemática de las reglas de movimiento, lo cual es suficiente para resolver problemas sencillos como el problema de los contenedores de agua y la mayoría de los presentados en los apartados anteriores. Sin embargo, esas estrategias son inadecuadas para problemas grandes o más complejos.

Las estrategias de búsqueda sin información expanden demasiados nodos antes de encontrar un camino que lleve a un estado objetivo. Es posible reducir el coste de la exploración con el auxilio de información específica sobre el problema, que suele denominarse *información heurística*. Los procedimientos de búsqueda que utilizan este tipo

de información con el fin de reducir el espacio de búsqueda se denominan *estrategias de búsqueda heurística*.

Este apartado se dedica al estudio de algunas de estas estrategias de búsqueda. Pero antes discutiremos razones para su empleo y algunas de sus limitaciones.

9.4.1. Búsqueda heurística: necesidad y limitaciones

La palabra “heurística” proviene del griego “*heurikein*”⁶, que significa “descubrir”. Por tanto, en una primera aproximación, podría decirse que una *heurística* es una técnica que permite descubrir soluciones a un problema de forma eficiente, en situaciones en que otro tipo de técnicas fracasan. El conocimiento heurístico es de naturaleza práctica y surge de un estudio detallado del problema. Las técnicas heurísticas tratan de aumentar la eficiencia del proceso de búsqueda aun a costa de sacrificar la completitud del proceso. En general, no debe de esperarse que una heurística obtenga la respuesta óptima para un problema, si bien proporcionará una buena respuesta en un tiempo razonable (inferior al coste teórico exponencial que pueda tener ese problema).

Existen dos formas fundamentales de incorporar el conocimiento heurístico sobre un dominio a un proceso de búsqueda [126]: i) en las propias reglas (conocimiento específico del mundo); ii) como una función heurística que evalúa los estados individuales del problema y determina su grado de “deseabilidad” (conocimiento general del mundo). Una *función heurística* es una aplicación entre las descripciones de los estados del problema y una medida o estimación de lo prometedor que es un estado en el camino para alcanzar un estado objetivo. De forma simple, la idea es continuar el proceso de búsqueda a partir del estado más prometedor de entre los de un conjunto de estados (expandidos por la aplicación de las reglas de movimiento que caracterizan el problema).

Ejemplo 9.1

Algunas funciones heurísticas sencillas son:

- Para el juego del ajedrez, se puede asignar un valor a cada pieza y juzgar una posición en función del número de piezas de ventaja sobre el oponente.
- Para el problema del viajante de comercio (véase más adelante), se puede adoptar como función heurística el número de kilómetros recorridos hasta el momento.
- Para el juego de las tres en raya, se puede utilizar una función f tal que:
 - i) Si p no es una posición en la que gane el jugador 1 o el 2, $f(p) = o(p, 1) - o(p, 2)$; donde $o(P, J)$ es un operador que calcula el número de filas, columnas y diagonales que están libres para el jugador J en la posición P .
 - ii) Si p es una posición ganadora para el jugador 1, $f(p) = \text{maxEntero}$.

⁶La exclamación *eureka* también tiene su origen en esta palabra griega [126].

- iii) Si p es una posición ganadora para el jugador 2, $f(p) = -\max_{\text{Entero}}$.

Observe que un valor alto de la función heurística señala a veces un estado prometedor (como ocurre con el juego del ajedrez y las tres en raya) mientras que otras veces es un valor bajo el que indica la aparición de una situación ventajosa (como en el problema del viajante de comercio).

Ya hemos comentado que el argumento fundamental para emplear estrategias de búsqueda heurística es la necesidad de reducir los espacios de búsqueda de los problemas, para no caer en la temida explosión combinatoria que aqueja a estos métodos. Sin embargo, existen otras razones que podemos alegar para usar búsqueda heurística:

- Las soluciones encontradas por una heurística constituyen, en promedio, una buena aproximación aunque pueden no ser adecuadas en los casos peores. Sin embargo, estos casos se dan raras veces.
- En ocasiones no se necesita una solución óptima y basta con una buena aproximación. En otros casos se busca una solución que cumpla un número de requisitos y, una vez encontrada, cesa la búsqueda.
- La definición de heurísticas y el estudio de su comportamiento para ver cuál es más eficiente, nos ayuda a entender mejor los problemas bajo consideración.

Es importante notar que las estrategias de búsqueda heurística, aunque suponen un avance con respecto a las estrategias de búsqueda a ciegas, poseen diversas limitaciones. Por un lado, la eficacia de las estrategias de búsqueda heurística depende de la forma en que explotan el conocimiento sobre el dominio particular de ese problema. Así que, por sí solas, no son capaces de superar el problema de la explosión combinatoria. Por otra parte, los costes de computación de una técnica de búsqueda pueden separarse en dos clases principales: i) costes de la aplicación de las reglas; ii) costes de control. Las estrategias de búsqueda a ciegas tienen un coste de control mínimo, pues la selección arbitraria de las reglas no requiere grandes cálculos, pero conducen a costes muy elevados de aplicación de dichas reglas, ya que éstas se aplican indiscriminadamente sin valorar su contribución real para la obtención de una solución. En el lado opuesto, informar completamente al sistema de control acerca del dominio del problema bajo consideración, lleva consigo aparejado un alto coste de control, tanto por las necesidades de almacenamiento en memoria como por la cantidad de cálculos que requiere la selección de las reglas a aplicar. Por el contrario, las estrategias completamente informadas dan lugar a costes mínimos en la aplicación de las reglas y guían al sistema directamente a la solución (i.e., se comportan de forma determinista). Como muestra la Figura 9.5 (adaptada de [104]), el coste total del proceso de búsqueda es el coste de la aplicación de las reglas sumado al de la estrategia de control. Observe que estos gastos deben permanecer equilibrados y, en muchos problemas, la eficiencia óptima se obtiene con estrategias de control que utilizan información que no llega a ser completa.

Naturalmente, existen muchas heurísticas de carácter general. Por ejemplo:

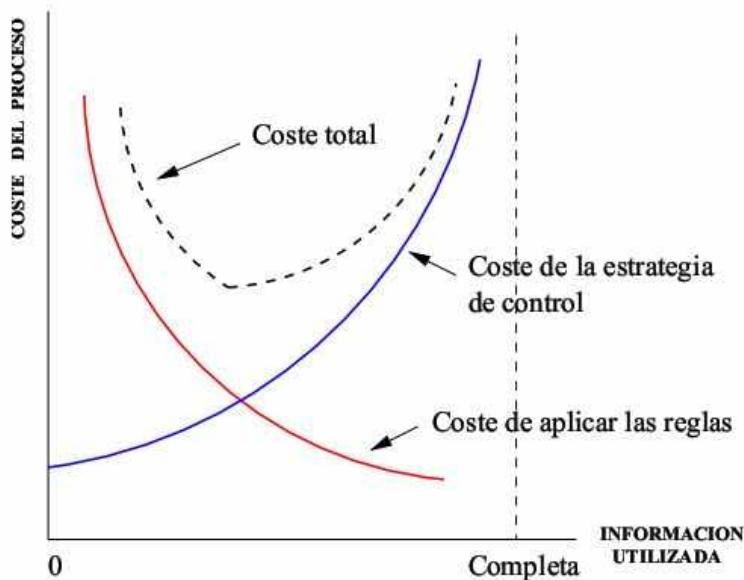


Figura 9.5 Costes del proceso de búsqueda.

Dado un problema a resolver, buscar un problema similar que ya esté resuelto y estudiar si las técnicas aplicadas a ese problema pueden ser útiles para resolver el nuestro.

Sin embargo, aunque estamos interesados en heurísticas aplicables a clases amplias de problemas más que en heurísticas de propósito especial (que exploten el conocimiento específico de un dominio para resolver problemas particulares), nos interesan aquéllas que pueden ser implementadas mediante un programa. En lo que sigue, presentamos algunas técnicas de control que utilizan heurísticas de propósito general y un método de búsqueda que combina las ventajas de las estrategias de control sistemáticas y las heurísticas.

9.4.2. Heurísticas locales

Las heurísticas *locales* deciden cuál será su próximo movimiento, atendiendo a las consecuencias inmediatas que va a tener su elección, en lugar de explorar exhaustivamente todas las consecuencias. En este apartado presentamos dos de ellas: la heurística del vecino más próximo y el método de la escalada.

Heurística del vecino más próximo

La heurística del vecino más próximo es un ejemplo de heurística de propósito general que consiste en seleccionar, en cada paso, la alternativa localmente superior,

desechando el resto de alternativas. Más concretamente, podríamos especificar esta heurística como sigue

Algoritmo 9.2 [Heurística del vecino más próximo]

Entrada: El estado inicial I .

Salida: Una condición de éxito o de fallo.

Comienzo

Inicialización: EstadoActual = I ; Visitados = [I];

Repetir

1. **Si** EstadoActual es un objetivo **entonces**

Comienzo

Devolver éxito; terminar;

Fin

2. **Si no** Comienzo

Candidatos = expandir(EstadoActual);
 CandidatosFinales = Candidatos - Visitados;
 EstadoActual = mejor(CandidatosFinales);
 Visitados = [Visitados] ++ [EstadoActual];

Fin

Hasta que se cumpla una condición de terminación;

Devolver fallo.

Fin

Por simplicidad y claridad, el procedimiento de expansión y selección del mejor candidato se ha separado en distintas fases que podrían aparecer entrelazadas en una implementación eficiente (*véase* más adelante). Por otra parte, se ha dejado sin especificar la función *mejor*, que selecciona el “mejor” de los estados candidatos. Para que el algoritmo funcione se necesitará una definición precisa de ese término, así como de la condición de terminación. Observe que este algoritmo solamente será efectivo si todos y cada uno de los nodos del espacio de estados están conectados directamente unos con otros (i.e., es un grafo fuertemente conectado). Esta es la condición necesaria que garantiza que, tarde o temprano, todos los nodos serán visitados y se alcanzará un nodo objetivo, en caso de existir una solución al problema. En este punto, también es conveniente notar que esta clase de heurística está relacionada con las técnicas de *control irrevocable*, en las que cada regla aplicable que se selecciona es aplicada irrevocablemente, sin que sea posible reconsiderar su aplicación en un tiempo posterior. Por el contrario, las técnicas utilizadas en el resto de los apartados de este capítulo caen dentro de los regímenes de *control tentativo* que sí permiten reconsiderar la aplicación de una regla determinada.

A continuación ilustramos la utilidad de la heurística del vecino más próximo resolviendo con ella el *problema del viajante de comercio*:

Un viajante de comercio tiene una lista de ciudades que debe de visitar.

Cada una de estas ciudades está unida directamente por una carretera a las

otras ciudades y se conoce la distancia que las separa. El problema consiste en encontrar la ruta más corta que, empezando y terminando en una ciudad (arbitraria) pase por el resto de las ciudades visitándolas exactamente una vez.

Informalmente, cuando el Algoritmo 2 se concreta para este problema, el proceso de encontrar una solución puede resumirse como sigue:

1. Seleccionar arbitrariamente una ciudad de comienzo, que se convierte en la ciudad actual.
 2. **Repetir.**
 - a) Seleccionar la ciudad más cercana a la actual de entre las no visitadas.
 - b) Ir a esa ciudad, que se convierte en la ciudad actual.
- hasta que** todas las ciudades hayan sido visitadas.

El siguiente programa Prolog utiliza estas directrices (y gran parte de las técnicas aprendidas en apartados anteriores) para dar solución a un problema del viajante de comercio como el representado en el grafo de la Figura 9.6.

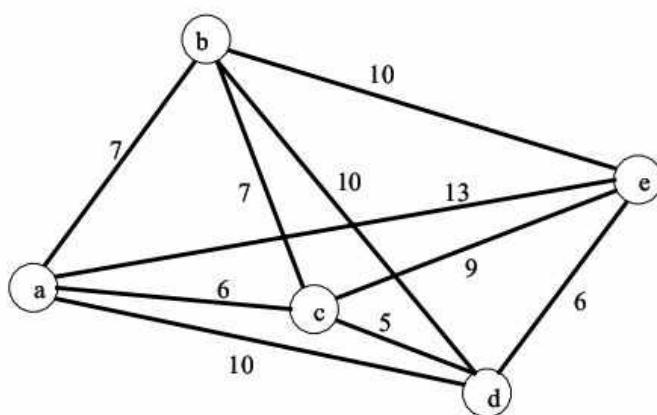


Figura 9.6 Grafo para un problema del viajante de comercio.

```

% REPRESENTACION DEL ESPACIO DE ESTADOS
%
% Se introduce el operador "<==>" para representar de forma
% intuitiva los arcos del grafo
:- op(100, xfx, '<==>').

% Mediante este operador se asocia una distancia a cada arco

```

```
% del grafo
:- op(200, xfx, ':').

% REPRESENTACION DEL GRAFO (REGLAS DE MOVIMIENTO)
%
% X<==>Y : D
% X e Y estan unidos por una carretera y distan D kilometros
% (Esta relacion juega el papel de las reglas de movimiento de
% otros problemas)
%
a<==>b : 7.      b<==>c : 7.      c<==>d : 5.      d<==>e : 6.
a<==>c : 6.      b<==>d : 10.     c<==>e : 9.
a<==>d : 10.     b<==>e : 10.
a<==>e : 13.

% RUTA OBJETIVO
%
% En este problema la solucion es un camino (una ruta).
%
% objetivo(Ruta):
% verdadero si Ruta es una lista que contiene todas las ciudades
% sin repetirlas (excepto la ultima).
objetivo([_|RestoRuta]) :- equiv(RestoRuta, [a,b,c,d,e]).
```

El predicado `equiv` se definió en el Apartado 8.4 y, esencialmente, comprueba que ambas listas poseen los mismos elementos independientemente de su orden.

```
% REGLAS DEL ESPACIO DE ESTADOS Y HEURISTICA DEL VECINO
% MAS PROXIMO
%
% viajante(Inicio, Ruta:Distancia)
% verdadero si Ruta es una lista que comienza en Inicio y termina
% en Inicio, pasando por todas las ciudades sin repetirlas.
% Distancia contienen los kilometros recorridos.
viajante(Inicio, Ruta:Distancia) :-
    viajante(Inicio, Inicio, []:0, Ruta:Distancia).

% viajante(Inicio,Final,RutaAcumulada:D_Acumulada,Ruta:D_Total)
% verdadero si Ruta es una lista que comienza en Inicio y
% termina en Final, sin repetir las ciudades intermedias por
% las que pasa.
viajante(_, _, Ruta:D, Ruta:D) :- objetivo(Ruta), !.

viajante(I, F, RutaAc:D_Ac, Ruta:D_Total) :-
    vecinoProximo(I, RutaAc, Vecino:D) ->
        (N_D_Ac is D_Ac + D,
         viajante(Vecino, F, [I|RutaAc]:N_D_Ac, Ruta:D_Total))
    ;
    (((I<==>F : D_F) ; (F<==>I : D_F)),
     N_D_Ac is D_Ac + D_F,
     viajante(I, F, [F,I|RutaAc]:N_D_Ac, Ruta:D_Total))
    ).
```

Dado que trabajamos con grafos fuertemente conectados, la llamada `vecinoProximo(I, RutaAc, Vecino:D)` siempre tiene éxito (y se ejecutan las acciones correspondientes a la parte `if_then` del condicional), a menos que desde `I` no pueda accederse a ciudades todavía sin visitar. Si la anterior llamada falla, se ejecutan las acciones correspondientes a la parte `else` del condicional. Entonces, estamos en una situación en la que basta recorrer la distancia `D_F` que separa la ciudad actual `I` de la ciudad `F` que inicia y cierra la ruta.

```
% HEURISTICA DEL VECINO MAS PROXIMO
%
% vecinoProximo(Inicio, RutaAcumulada, Vecino:Distancia)
%
vecinoProximo(E, RutaAcumulada, Vecino:Distancia) :-
    expandir(E, RutaAcumulada, Sucesores),
    minimo(Sucesores, Vecino:Distancia).

% expandir(E, Generados, Sucesores):
% verdadero si "Sucesores" es la lista de sucesores del
% estado E (es decir, la lista de ciudades accesibles desde
% la ciudad E y las distancias que las separan).

expandir(E, Generados, Sucesores) :-
    setof(S:D,
          ((E<==>S:D) ; (S<==>E:D)), not(member(S,Generados))),
    Sucesores, !.
expandir(_, _, []). % si setof falla porque E no tiene sucesores

% minimo(Sucesores, M) selecciona la ciudad accesible que esta
% a menor distancia. Esto es, el vecino mas proximo.
minimo([(S:D)|Resto], M) :- minimo(Resto, S:D, M).
minimo([], M, M).
minimo([(S:D)|Resto], N:A, M) :-
    (D < A) -> minimo(Resto, S:D, M); minimo(Resto, N:A, M).
```

Cuando planteamos la pregunta “?- `viajante(a,L)`.” al sistema Prolog, éste responde con la solución: `L = [a, b, e, d, c, a]`:34. Ciertamente, esta es la ruta más corta y la solución al problema. Sin embargo, la única forma de asegurar que esto es así es calcular cada una de las otras posibles rutas y comprobar que no existe otra más corta. Este punto ilustra la diferencia entre los problemas en los que se busca *cualquier camino* que nos lleve a una solución y aquéllos en los que se busca el *mejor camino*. Estos últimos son mucho más complejos de resolver y conducen con frecuencia a la temida explosión combinatoria. La heurística del vecino más próximo se caracteriza por rechazar la mayoría de las alternativas, centrándose en una de ellas con el fin de obtener un procedimiento eficiente. Sin embargo, para los problemas en los que se debe encontrar el *mejor camino* no pueden usarse heurísticas que puedan olvidarse del camino que conduce a la mejor solución. Se requiere una búsqueda más exhaustiva. En el Apartado 9.4.3 se presenta una estrategia de búsqueda heurística que recuerda todas las alternativas ex-

ploradas, con el fin de volver a una de ellas tan pronto como ésta se transforma en más prometedora que la que se inspecciona en la actualidad.

Observación 9.4 *El problema del viajante de comercio es muy conocido en la bibliografía sobre análisis de la complejidad de los algoritmos. La única forma de resolver este problema es diseñar un algoritmo que explore todas las posibles rutas y se quede con la de menor longitud. Si existen n ciudades, el número de rutas diferentes es de $(n - 1)!$ Dado que el tiempo necesario para examinar una ruta es proporcional a n , el tiempo total empleado en la búsqueda será proporcional a $n!$ Para valores de n pequeños, una búsqueda sistemática permite resolver el problema. Ahora bien, en cuanto el número de ciudades crece, el problema se transforma en intratable. Por ejemplo, si suponemos que hay que visitar 20 ciudades, es necesario explorar del orden de $1,21 \times 10^{17}$ rutas y el tiempo empleado será proporcional a $2,43 \times 10^{18}$. Por pequeño que sea el tiempo necesario para ejecutar una instrucción, estaríamos hablando de millones de años de cómputo.*

El problema del viajante de comercio cae dentro de la clase de problemas NP (problemas de complejidad No Polinomial), que engloba a aquéllos problemas cuya complejidad es superior a $O(n^k)$, donde k es una constante, y se consideran intratables.

El procedimiento que se ha implementado en este apartado, sirve para resolver una clase particular de problemas del viajante de comercio (cuyo espacio de estados puede representarse mediante un grafo fuertemente conectado). Este procedimiento se ejecuta en un tiempo proporcional a n^2 , lo cual es una mejora significativa frente a un tiempo proporcional a $n!$

El método de escalada

El método de escalada, para hallar un máximo de una función, pertenece a una clase de estrategias que se caracterizan por utilizar conocimiento local para alcanzar un conocimiento global, como la heurística del vecino más próximo. En el proceso de escalada, situados en cualquier punto, nos movemos en la dirección de la pendiente máxima (conocimiento local) para, eventualmente, alcanzar un máximo de la función (conocimiento global). Para funciones que no presentan máximos locales⁷, el conocimiento de la dirección de máxima pendiente es suficiente para encontrar una solución.

Podemos adaptar el proceso de escalada, utilizado por los matemáticos, asociando al espacio de estados una *función de evaluación* que asigne a cada estado un valor real. Entonces, la estrategia de control puede emplear dicha función para seleccionar un operador de movimiento: siempre que la aplicación de un operador de movimiento a un estado produzca un nuevo estado que dé lugar a un incremento en el valor de la función de evaluación, se elegirá ese operador de movimiento. Observe que, para que el método sea efectivo, la función elegida debe de ser tal que alcance su máximo para un estado

⁷Un máximo local es un estado “mejor” que todos sus estados vecinos, pero que no es mejor que otros estados del espacio de búsqueda.

objetivo. El proceso que acabamos de describir constituye la forma más sencilla de escalada y recibe el nombre de *escalada simple*. Una forma de implementar este método es la siguiente:

Algoritmo 9.3 [Escalada simple]

Entrada: El estado inicial I y una función de evaluación h .

Salida: Una condición de éxito o de fallo.

Comienzo

Inicialización: EstadoActual = I ;

Repetir

1. Evaluar el estado actual: Valoración = $h(\text{EstadoActual})$;
2. Si EstadoActual es un objetivo entonces

Comienzo

Devolver éxito; terminar;

Fin

3. Si no Comienzo

 Aplicar un operador de movimiento para obtener un NuevoEstado, a partir de EstadoActual;

 Si ($h(\text{NuevoEstado}) > \text{Valoración}$), EstadoActual = NuevoEstado;

Fin

 Hasta que no existan operadores por aplicar a EstadoActual;

Devolver fallo.

Fin

Observe que el empleo de una función de evaluación h , como una forma de introducir conocimiento específico sobre el problema, es lo que convierte a este método (y a otros estudiados a lo largo de este apartado) en una estrategia de búsqueda heurística.

Una variación útil del método de escalada simple consiste en considerar todos los posibles movimientos a partir del estado actual y elegir el mejor de ellos como nuevo estado⁸. Este método se denomina método de *escalada por la máxima pendiente* o *búsqueda del gradiente* y, como el lector puede apreciar, tiene una estrecha relación con la heurística del vecino más próximo.

Los métodos de escalada comparten con otros métodos que utilizan heurísticas locales la desventaja de que no ofrecen garantías de que se encontrará la solución deseada. Tanto la escalada simple como la de máxima pendiente pueden acabar sin encontrar un estado objetivo, porque encuentran un estado s a partir del cual no es posible generar un nuevo estado e tal que $h(e) > h(s)$ (es decir, un estado nuevo que sea “mejor” que el viejo). Esto sucede cuando el proceso de escalada se encuentra con un máximo local (o una meseta⁹).

⁸Observe la diferencia con el método de escalada simple, en el que se selecciona el primer movimiento que produce un estado que “mejora” el estado actual.

⁹Un área plana del espacio de búsqueda en la que un conjunto de estados vecinos poseen el mismo valor cuando se les aplica la función de evaluación.

9.4.3. Búsqueda *primero el mejor*

La estrategia de *búsqueda primero el mejor* se parece al método de la escalada en que usa una función de evaluación para determinar cuan “prometedor” es un nodo. Por contra, en lugar de seguir un único camino de forma irrevocable, mantiene información de los diferentes caminos generados hasta el momento, pudiendo volver a considerar alguno de ellos si, en un tiempo posterior, se transforma en más prometedor que el actual. De esta forma, el método no queda estancado en un máximo local.

La estrategia basada en buscar primero el mejor intenta combinar las ventajas de la búsqueda en profundidad con las de la búsqueda en anchura. Si existe un camino hasta un nodo objetivo, por una parte lo encontrará sin tener que explorar todas las ramas (como la búsqueda en profundidad) y, por otra, el camino que encuentre será óptimo (de forma similar a como la búsqueda en anchura encuentra un camino de longitud mínima), si se cumplen ciertas condiciones de admisibilidad [105].

Una aproximación informal

Desde el punto de vista de su implementación es semejante al algoritmo de búsqueda en anchura, si bien su comportamiento puede recordar a un proceso de búsqueda en profundidad. Antes de presentar formalmente el algoritmo que implementa la búsqueda *primero el mejor*, vamos a discutir de manera intuitiva su modo de operación.

Como en el caso de la búsqueda en anchura, el proceso de búsqueda *primero el mejor* parte de un estado inicial que se incluye en una lista de nodos a inspeccionar. En cada paso, el algoritmo selecciona el nodo más prometedor que se haya generado hasta el momento y no haya sido inspeccionado. Habitualmente se usa una función de evaluación con una componente heurística para determinar lo prometedor que es un nodo. El nodo seleccionado se expande, aplicando las reglas de movimiento que le son aplicables, para generar sus sucesores. Si alguno de los nodos sucesores es un nodo objetivo, la inspección termina. En caso contrario, los nodos sucesores se añaden a una lista de nodos por inspeccionar. Nuevamente, se selecciona el más prometedor de entre los de la lista y el proceso continúa de la misma forma.

La Figura 9.7 ilustra el comentario anterior. Por simplicidad y para no introducir complicaciones innecesarias en este momento, consideramos que el espacio de búsqueda es un árbol. En dicha figura, los nodos seleccionados en cada paso son los remarcados. Los números que encierran cada uno de los nodos son los valores asignados por la función de evaluación. Asociamos los números más pequeños con los nodos más prometedores. Como habitualmente venimos haciendo, representamos el nodo objetivo con un doble círculo. En cada nodo disponemos un enlace “hacia atrás” que permite reconstruir el camino recorrido hasta alcanzar el nodo. Observe que, en nuestro ejemplo, durante los primeros pasos se despliega una rama (como en un proceso de búsqueda en profundidad), porque sus nodos son más prometedores. Ahora bien, tan pronto como ésta se torna menos prometedora comienzan a expandirse nodos de niveles menos profundos (como en

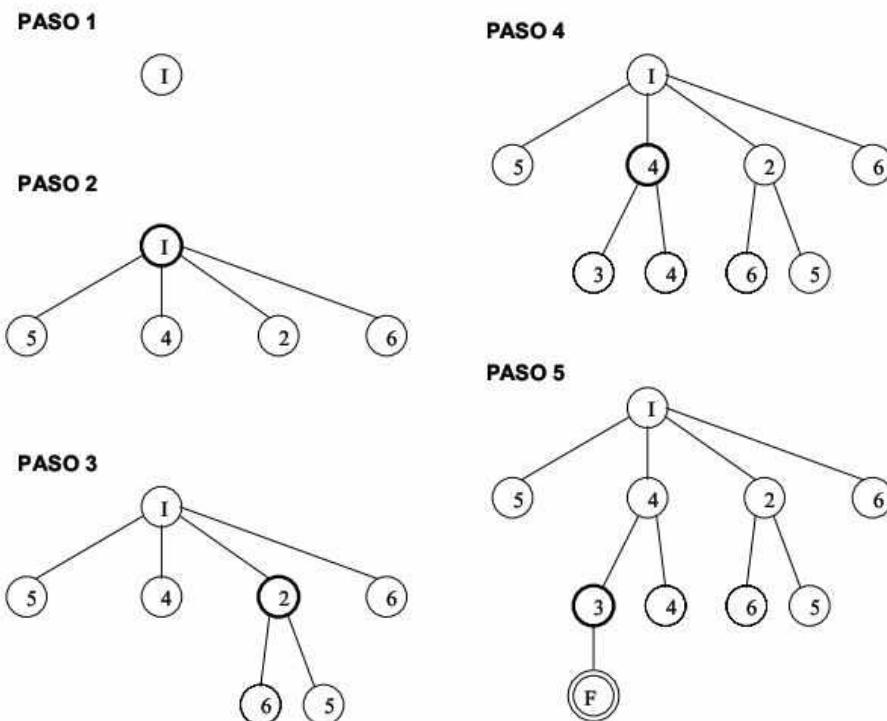


Figura 9.7 Pasos en un proceso de búsqueda *primero el mejor*.

un proceso de búsqueda en anchura). Sin embargo, si las cosas van bien, no será necesario expandir completamente los niveles del árbol de búsqueda. Finalmente, nótese que la estrategia de buscar primero el mejor selecciona el mejor estado disponible, aun si éste tiene un valor mayor (y por tanto es menos prometedor) que el que se estaba explorando en el último paso. Por ejemplo, en la Figura 9.7, el paso 4 es ilustrativo de este hecho: se selecciona un nodo con valor 4, mayor que el valor 2 asociado al nodo explorado en el paso precedente. Este comportamiento contrasta con el del proceso de escalada, que se detiene en cuanto no se encuentra un estado sucesor mejor que el estado actual.

Algoritmo A*

En lo que sigue, presentamos el procedimiento de búsqueda *primero el mejor*, que también recibe el nombre de *algoritmo A** [104]. Éste puede verse como un algoritmo genérico de búsqueda en grafos que sintetiza todas las estrategias de búsqueda tentativa estudiadas en los apartados precedentes.

El algoritmo A* genera un grafo explícito, G , llamado *grafo de exploración*, y un árbol de exploración T . El árbol T , incluido en G , queda definido por los enlaces “hacia atrás” que unen cada nodo con un único antecesor y que se generan en el paso 5b del

algoritmo. Los caminos explorados quedan preservados en el grafo G pero solamente un camino distinguido (el menos costoso que lleva a un nodo) queda definido en T . Esto es debido a que los enlaces que constituyen el árbol T se reconfiguran a medida que avanza el proceso de búsqueda (véase el paso 5b). Supondremos que existe una función `expandir` que, aplicada a un nodo (estado), devuelve sus sucesores. La función `expandir` controla que los sucesores de un nodo n no sean a su vez ascendientes de n . En el caso más simple, en el que la función `expandir` siempre produzca nodos sucesores que no se hayan generado antes (i.e., siempre se generan estados nuevos), el grafo G y el árbol T coincidirán.

Para implementar el algoritmo A* se emplean las siguientes estructuras de datos:

1. Una lista denominada `ListaCandidatos` que contiene los nodos que todavía no se han expandido y que, por lo tanto, no tienen sucesores. Los nodos de esta lista se caracterizan por ser las hojas del árbol T .
2. Una lista, denominada `ListaVisitados`, compuesta por los nodos que ya se han expandido. Los nodos de esta lista se reconocen inmediatamente porque son nodos internos del árbol T (que tienen sucesores) o por ser hojas ya consideradas pero que no generaron sucesores.

Ambas listas son necesarias para propiciar una búsqueda sobre un grafo e impedir caminos cílicos en el mismo (cada vez que se genera un nodo, se controla si éste ya se había generado con anterioridad). Sin embargo, `ListaCandidatos` se emplea, principalmente, para ordenar los candidatos conforme a algún criterio heurístico. Con tal fin se emplea una función de evaluación heurística f , que consta de dos componentes:

1. La función $g(n)$ es una medida del coste de ir desde el estado inicial i hasta el nodo actual n por un camino de coste mínimo.
2. La función $h(n)$ representa el coste de un camino de coste mínimo que enlace el nodo actual n con un nodo objetivo (para todo nodo objetivo y todo camino que une n con cada uno de ellos).

Por consiguiente, la función $f(n) = g(n) + h(n)$ proporciona el valor del camino de mínimo coste para ir desde el estado inicial i a un estado objetivo, pasando por el nodo n . La función de evaluación heurística f no puede evaluarse con precisión desde las primeras fases de la resolución de un problema, siendo únicamente posible una estimación. Denominaremos g^* y h^* a las estimaciones de g y h , respectivamente, siendo $f^*(n) = g^*(n) + h^*(n)$ la estimación de f . Es habitual definir la función $g^*(n)$ como el coste real de ir desde el estado inicial i al estado n por el mejor camino encontrado hasta el momento. Esto es $g^*(n) = \sum_{k=1}^n \text{coste}(K, L)$, donde i, \dots, k, l, \dots, n es la secuencia de nodos que constituyen el mejor camino encontrado entre i y n , y $\text{coste}(K, L)$ es una función de coste asociada al arco que une los nodos k y l (e.g., el coste de aplicar un operador de movimiento para pasar del estado k al l). Por consiguiente, la función $g^*(n)$

puede computarse con precisión. La función $h^*(n)$ es una estimación del coste adicional necesario para alcanzar un nodo objetivo a partir del estado actual n . Esta es una verdadera función heurística porque para definirla suele ser imprescindible utilizar conocimiento acerca del dominio del problema. La función $h^*(n)$ constituye una verdadera estimación.

Como se verá más adelante, es importante que, cuando se defina, h^* no sobreestime a h ; esto es, que $h^*(n) \leq h(n)$, para todo nodo n . Suele ser fácil encontrar una función h^* que no sobreestime a h . Por ejemplo, en el problema del cálculo de rutas en un grafo cuyos nodos representan ciudades (véase el Apartado 9.4.3), la distancia en línea recta entre una ciudad origen y una ciudad objetivo es una cota inferior de la distancia (óptima) existente entre esas dos ciudades.

En el problema del puzzle-8 (véase el Problema 9.26), el número de piezas descolocadas respecto a la configuración objetivo es una cota inferior del número de pasos que hay que dar para alcanzar dicha configuración [105].

El funcionamiento del algoritmo A* es simple, a pesar de su aparente complejidad. Selecciona el candidato mejor posicionado en la lista `ListaCandidatos`, aquél que tiene el mínimo valor de f^* , y comprueba si es un estado objetivo. Si es un estado objetivo el algoritmo termina (dando la posibilidad de realizar más consultas); si no, se expande el nodo y se obtiene una lista `NuevosCandidatos` que se procesan para ampliar el grafo de búsqueda con nuevos nodos, actualizando los enlaces que sea necesario actualizar para conseguir los mejores caminos encontrados hasta el momento. Al mismo tiempo, el algoritmo inserta los `NuevosCandidatos` en la `ListaCandidatos` y el ciclo comienza nuevamente.

Algoritmo 9.4 [Algoritmo A*]

Entrada: El estado inicial I . Una función de evaluación f^* tal que, para un estado N , $f^*(n) = g^*(n) + h^*(n)$, donde $g^*(n)$ es el coste de ir de I a N , y $h^*(n)$ es una estimación del coste de ir de N a un estado objetivo por el mejor camino.

Salida: Una condición de éxito y un camino de coste mínimo desde el nodo inicial a un nodo objetivo, o una condición de fallo.

Comienzo

Inicialización: Crear un grafo G que solo contiene el estado inicial I ;
`ListaCandidatos = [(I,g*(I),h*(I))]; ListaVisitados = [];`

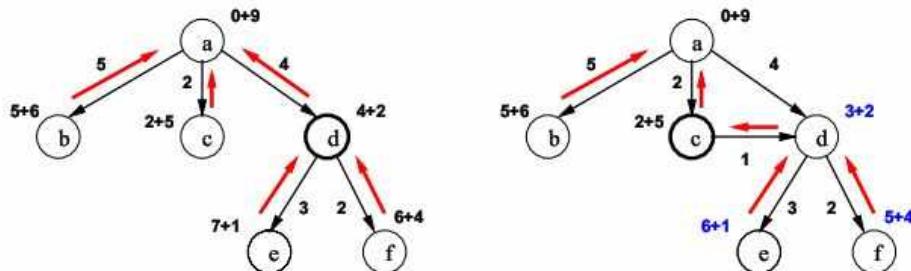
Repetir

1. `(EstadoActual,Cg*,Ch*) = cabeza(ListaCandidatos);`
2. `ListaCandidatos = cola(ListaCandidatos);`
3. `ListaVisitados = [(EstadoActual,Cg*)] | ListaVisitados;`

4. Si EstadoActual es un objetivo entonces
Comienzo
Devolver éxito y un camino;
Si se desean más respuestas entonces **continuar** si no **terminar**;
Fin
5. **Si no Comienzo**
- NuevosCandidatos = expandir(EstadoActual);
 Incorporar NuevosCandidatos como sucesores de EstadoActual en el grafo G (i.e., establecer un enlace “hacia delante” desde el EstadoActual a cada uno de sus sucesores);
 - Para cada sucesor s en la lista de NuevosCandidatos hacer procesar(s) (i.e., establecer un enlace “hacia atrás” desde s a EstadoActual y, cuando sea necesario, actualizar los enlaces de forma que se preserven los mejores caminos);
 - Reordenar la ListaCandidatos según el orden creciente de los valores de la función de evaluación f^* (es decir, en cabeza de ListaCandidatos se encuentra el estado $(E, g^*(E), h^*(E))$ con $f^*(E) = g^*(E) + h^*(E)$ mínimo, que se considera el más prometedor);
- Fin**
- Hasta que** ListaCandidatos = [];
- Devolver** fallo. **Fin**

El Procedimiento procesar(s) requiere algunas explicaciones adicionales. En él se distinguen tres casos excluyentes unos de otros, según que el estado s sucesor del estado actual pertenezca a alguna de las listas ListaCandidatos o ListaVisitados, o a ninguna de ellas. Esto es, según s sea un estado que ya se ha generado con anterioridad y aparece representado en el grafo explícito de búsqueda o no. Si s no ha sido generado con anterioridad (caso i), se establece un enlace “hacia atrás” a EstadoActual y se añade s a ListaCandidatos. Cuando s ha sido generado con anterioridad (casos ii y iii), es porque se ha encontrado un camino distinto (y posiblemente mejor), desde el estado inicial τ hasta el propio s, en el árbol de exploración τ . Queremos que el árbol de exploración almacene el camino menos costoso de los encontrados hasta el momento. Cuando el coste del camino nuevo, $g^*(s)$, es menor que el del encontrado antes, c_{viejo} , entonces es necesario modificar el árbol de exploración para que el enlace “hacia atrás” del nodo que se ha vuelto a generar apunte al nodo EstadoActual, que pasa a ser su antecesor.

Observe que, en el (caso iii), este proceso puede conllevar aparejado la alteración de otros enlaces y la propagación de los costes del nuevo camino a los sucesores de s en el grafo G (véase la Figura 9.8, donde para cada nodo N se refleja el valor de $f^*(N) + h^*(N)$ en cada iteración). En cualquier otro caso, no se hace nada y el estado s es desecharo.



A) Grafo de estados: Iter. 2

B) Grafo de estados: Iter. 3

Iteración	EstadoActual	NuevosCandidatos	ListaCandidatos	ListaVisitados
0		[]	[a]	[]
1	a	[b, c, d] $\xrightarrow{\text{(caso i)}}$	[b, c, d]	[a]
2	d	[e, f] $\xrightarrow{\text{(caso i)}}$	[e, f, b, c]	[d, a]
3	c	[d] $\xrightarrow{\text{(caso iii)}}$	[e, f, b]	[c, d, a]

Figura 9.8 Modificación del árbol de exploración.

```

procesar(S);
    Sea  $g^*(S) = \text{coste}(\text{EstadoActual}, S) + C_{g^*}$  en
    Comienzo
        i) Si S no pertenece a ListaCandidatos o ListaVisitados
            entonces Comienzo
                Establecer un enlace "hacia atrás" desde S a EstadoActual;
                ListaCandidatos = [(S,  $g^*(S)$ ,  $h^*(S)$ ) | ListaCandidatos];
            Fin
        ii) Si  $(S, C_{\text{viejo}}, _)$  pertenece a ListaCandidatos y  $g^*(S) < C_{\text{viejo}}$ 
            (i.e., el coste del camino que pasa por EstadoActual y lleva a S es menor
            que el anterior)
            entonces Establecer un enlace "hacia atrás" desde S a EstadoActual;
        iii) Si  $(S, C_{\text{viejo}})$  pertenece a ListaVisitados y  $g^*(S) < C_{\text{viejo}}$ 
            (i.e., el coste del camino que pasa por EstadoActual y lleva a S es menor
            que el anterior)
            entonces Comienzo
                Establecer un enlace "hacia atrás" desde S a EstadoActual;
                Para cada descendiente D reordenar enlaces ("hacia atrás") y
                reevaluar  $g^*(D)$  y  $h^*(D)$ ;
            Fin
    Fin

```

Observaciones 9.1

- Los algoritmos estudiados para búsqueda sin información son casos particulares del algoritmo primero el mejor. Basta con hacer la función heurística $h^*(N) = 0$ y

elegir la función $g^*(n)$ de forma adecuada: si $g^*(n)$ es la profundidad del nodo n , obtenemos un algoritmo de búsqueda en anchura; si $g^*(n)$ es el valor negativo de la profundidad del nodo n , obtenemos un algoritmo de búsqueda en profundidad.

2. La función heurística $h^*(n)$ indica la bondad del nodo n en sí mismo, medida por su “distancia” al objetivo. La función $g^*(n)$ indica la bondad del camino que conduce al nodo n , medida por el “coste” de ir desde el estado inicial al nodo n .
3. Si para alcanzar el objetivo no es relevante el camino seguido, se puede hacer que $g^*(n) = 0$, de modo que siempre se elija el nodo que parece más cercano al objetivo. Este comportamiento recuerda a la heurística del vecino más próximo, salvo por su régimen de control tentativo que le impediría caer estancado en máximos locales.



Para finalizar presentamos un resultado, demostrado por Hart, Nilson y Raphael en 1968, que garantiza que el algoritmo A* encuentra un camino de coste mínimo a un objetivo si éste existe.

Teorema 9.1 *Sean un grafo G y una función heurística h^* , que cumplen las siguientes condiciones:*

- *Para todo nodo n del grafo G , n tiene un número finito de sucesores;*
- *El coste de cada arco del grafo G es mayor que una cantidad positiva ϵ ;*
- $h^*(n) \leq h(n)$, para todo nodo n del grafo G .

Si existe un camino de coste finito entre el nodo inicial i y un nodo objetivo f , el algoritmo A termina encontrando el camino de coste mínimo entre i y f .*

Una nueva versión de la demostración de este teorema se puede encontrar en [105].

El algoritmo A* aplicado a la búsqueda de la ruta más corta entre dos ciudades

En este apartado, como aplicación del algoritmo A*, presentamos una solución al problema de obtener la ruta más corta entre dos ciudades. Para concretar, nos referiremos al conjunto de ciudades del mapa de carreteras dibujado en la Figura 9.9. Por claridad, damos una solución que se ciñe, en lo esencial, a lo expuesto en el apartado anterior y utiliza las mismas estructuras de datos. Se deja al lector como ejercicio la tarea de introducir las mejoras que crea necesarias con el fin de aumentar la eficiencia y disminuir los requisitos de memoria de esta solución. Observe que es posible abordar la solución de este problema mediante estructuras de datos alternativas a las que aquí utilizamos. Por ejemplo, en [21] se usa un término para representar el grafo de búsqueda.

En el grafo de la Figura 9.9 los nodos son ciudades y los arcos son carreteras (de doble sentido). El coste asociado a cada arco representa la distancia entre dos ciudades. Se conoce la posición de cada ciudad en un sistema de coordenadas, de forma que es posible calcular la distancia que las separa en línea recta. Representaremos los nodos del grafo mediante un término $n(\text{Ciudad}, x, y)$, donde Ciudad es un identificador del nodo mientras que X e Y son sus coordenadas, que reflejan su posición en el mapa.

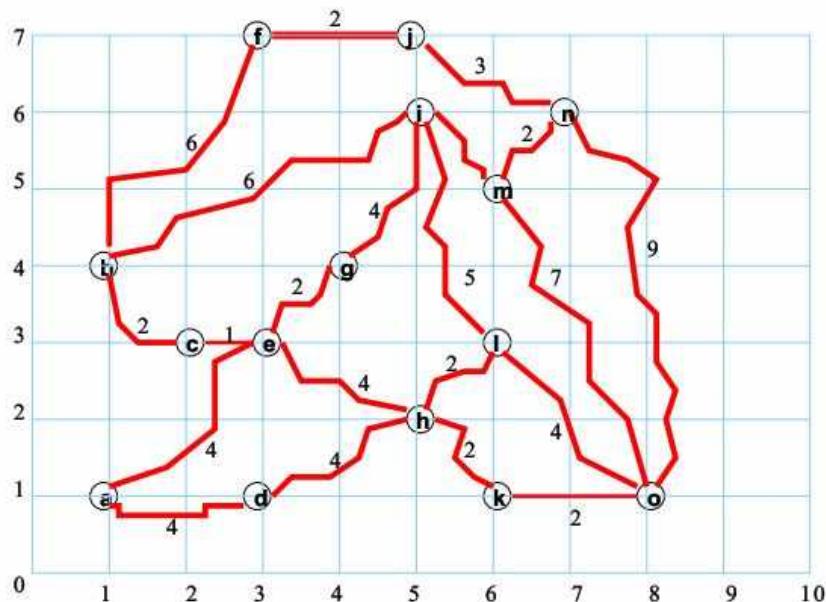


Figura 9.9 Mapa de Carreteras.

En este problema, el usuario especifica el estado inicial (la ciudad de partida) y el estado objetivo (la ciudad de llegada) cuando realiza una consulta. La solución es el mejor camino que permite acceder a la ciudad objetivo desde la ciudad de partida. Representamos la solución mediante una lista de las ciudades por las que se pasa hasta llegar a la ciudad objetivo, es decir, una ruta. También suministramos la longitud de la ruta.

En nuestra aplicación, la lista `ListaCandidatos` contiene, en lugar de simples estados, estructuras del tipo `[E | Camino] : G*(E) + H*(E)` donde `E` es un estado candidato (la siguiente ciudad a visitar para llegar a la ciudad objetivo por un camino de coste mínimo) y `Camino` es la mejor ruta (recorrida hasta el momento) para llegar a `E` desde el estado (ciudad) inicial. La lista `ListaCandidatos` solamente almacena el mejor camino encontrado para llegar a un estado candidato `E`. Si en algún momento se encuentra otro mejor, se elimina la ruta antigua y se sustituye por la nueva. Observe que asociamos a cada ruta de `ListaCandidatos` un término constituido por los sumandos `G=g*(E)` y `H=h*(E)`, lo que nos permite seleccionar en cada instante el camino más prometedor de una serie de

candidatos (aquél para el cuál la expresión $G + h$ se evalúa a un valor mínimo) La lista visitados contiene las ciudades visitadas hasta el momento e información de coste. Con más precisión, almacena estructuras del tipo $E:G$ donde E es un estado visitado y $G=g^*(E)$ es el coste de ir desde el estado inicial al estado E por el mejor camino encontrado hasta el momento. Si con posterioridad se encuentra un camino mejor que conduce a E , desde el estado inicial, se actualiza el valor de G .

La función de evaluación $f^*(N) = g^*(N) + h^*(N)$ se elige de forma que: $g^*(N)$ es el coste real de ir desde la ciudad de partida a la ciudad N por el mejor camino encontrado hasta el momento; $h^*(N)$ es la distancia en línea recta entre la ciudad N y la ciudad de llegada. Observe que, tal como se ha definido, h^* no sobreestima a h .

Con estas precisiones, estamos en disposición de presentar el programa Prolog que resuelve el problema de la búsqueda de la ruta más corta entre dos ciudades.

```
% REPRESENTACION DEL ESPACIO DE ESTADOS
% Representaremos los nodos del grafo como ternas
% n(Nodo, X, Y), donde Nodo es un identificador del nodo mientras
% que X e Y son sus coordenadas, que reflejan su posicion en
% el mapa. Conocer las coordenadas de los nodos permite estimar
% la distancia existente entre dos de ellos.

% Se introduce el operador "<==>" para representar de forma
% intuitiva los arcos del grafo
:- op(100, xfx, '<==>' ).

% Mediante este operador se asocia un coste a cada arco
% del grafo
:- op(950, xfx, ':') .

% REPRESENTACION DEL GRAFO (REGLAS DE MOVIMIENTO)
% X<==>Y : c(X,Y)
% X e Y estan unidos por una conexion directa. c(X,Y) es el
% coste de ir del nodo X al Y
% (Esta relacion juega el papel de las reglas de movimiento de
% otros problemas)
%
n(a,1,1)<==>n(d,3,1):3.      n(g,4,1)<==>n(i,5,6):4.
n(a,1,1)<==>n(e,3,3):4.      n(h,5,2)<==>n(k,6,1):2.
n(b,1,4)<==>n(c,2,3):2.      n(h,5,2)<==>n(l,6,3):2.
n(b,1,4)<==>n(f,3,7):6.      n(i,5,6)<==>n(l,6,3):5.
n(b,1,4)<==>n(i,5,6):6.      n(i,5,6)<==>n(m,6,5):2.
n(c,2,3)<==>n(e,3,3):1.      n(j,5,7)<==>n(n,7,6):3.
n(d,3,1)<==>n(h,5,2):4.      n(k,6,1)<==>n(o,8,1):2.
n(e,3,3)<==>n(h,5,2):3.      n(l,6,3)<==>n(o,8,1):4.
n(e,3,3)<==>n(g,4,4):2.      n(m,6,5)<==>n(o,8,1):7.
n(f,3,7)<==>n(j,5,7):2.      n(m,6,5)<==>n(n,7,6):3.
                                         n(n,7,6)<==>n(o,8,1):9.
```

```
% ECUACIONES DEL ESPACIO DE ESTADOS: BUSQUEDA PRIMERO EL MEJOR
% primeroElmejor(I,F,Plan):
% Verdadero si existe un camino minimo entre el estado inicial I
% y un estado final (objetivo) F. En Plan se devuelve dicho camino.
% El plan se muestra en orden inverso.

primeroElmejor(I,F,Plan) :- bucle(F, [[I]:0+_], [], Plan).

% bucle(Final,Candidatos,Visitados,Plan):
% Verdadero si Candidatos (la lista de caminos candidatos) contiene
% un camino con el estado objetivo F; dicho camino se devuelve
% como el Plan. Ademas, el Plan recoge el coste del camino.

bucle(F, [[F|RestoPlan]:G+_], _, [F|RestoPlan]:G):-!.
bucle(F, [[E|Camino]:G+H|RestoCandidatos], Visitados, Plan) :-
    expandir([E|Camino]:G+H, F, NuevosCandidatos),
    procesar(NuevosCandidatos, RestoCandidatos, [E:G|Visitados],
             ListaCandidatos, ListaVisitados),
    bucle(F, ListaCandidatos, ListaVisitados, Plan).

% PREDICADOS AUXILIARES
% expandir([E|Camino]:G+H, NuevosCaminos, F):
% NuevosCaminos es la lista de nuevos caminos candidatos, que
% resultan de expandir el camino [E|Camino] con los estados
% sucesores del estado E (es decir, los nodos accesibles desde
% E y las distancias que los separan).
% expandir elimina los estados sucesores S que sean ascendientes
% de E, calcula el coste g(S) del nodo S y una estimacion de h(S).

expandir([E|Camino]:G+_ , F, NuevosCaminos) :-
    setof([S,E|Camino]:Ng+Nh,
          sucesorLegal(S, [E|Camino]:G+_ , Ng, Nh),
          NuevosCaminos), !.
expandir(_, _, []). %% setof falla porque E no tiene sucesores

sucesorLegal(S, [E|Camino]:G+_ , Ng, Nh) :-
    ((n(E,X1,Y1)<==>n(S,X2,Y2)):C) ;
    ((n(S,X2,Y2)<==>n(E,X1,Y1)):C) ,
    not(member(S, [E|Camino])),
    Ng is G+C, estimarNh(X2,Y2,F,Nh).

% procesar(NuevosCandidatos, RestoCandidatos, Visitados,
%           ListaCandidatos, ListaVisitados),
% Implementa los pasos 5b y 5c del algoritmo A*. Cuando el
% tratamiento termina se devuelve la ListaCandidatos que hay que
% continuar procesando. La ListaCandidatos se suministra ordenada
% segun el orden creciente de los valores de f. Tambien se
% devuelve una ListaVisitados actualizada
%
```

```

% nada que procesar
procesar([], RestoCandidatos, Visitados, RestoCandidatos,
         Visitados):-!.

% El nuevo candidato E no esta en RestoCandidatos o Visitados
procesar([[E|Camino]:G+H|Candidatos], RestoCandidatos,
         Visitados, ListaCandidatos, ListaVisitados):-
    (not(member([E|_]:_, RestoCandidatos)),
     not(member(E:_, Visitados))) ->
    (insertar([[E|Camino]:G+H], RestoCandidatos, N_RestoCandidatos),
     procesar(Candidatos, N_RestoCandidatos,
              Visitados, ListaCandidatos, ListaVisitados)).

% El nuevo candidato E esta en RestoCandidatos
procesar([[E|Camino]:G+H|Candidatos], RestoCandidatos,
         Visitados, ListaCandidatos, ListaVisitados):-
    (member([E|Camino_viejo]:C_viejo+H_viejo, RestoCandidatos),
     G < C_viejo) ->
    (eliminar([E|Camino_viejo]:C_viejo+H_viejo, RestoCandidatos,
              N_RestoCandidatos),
     insertar([[E|Camino]:G+H], N_RestoCandidatos,
              NN_RestoCandidatos),
     procesar(Candidatos, NN_RestoCandidatos,
              Visitados, ListaCandidatos, ListaVisitados));
    (procesar(Candidatos, RestoCandidatos,
              Visitados, ListaCandidatos, ListaVisitados)).

% El nuevo candidato E esta en Visitados
procesar([[E|Camino]:G+H|Candidatos], RestoCandidatos,
         Visitados, ListaCandidatos, ListaVisitados):-
    (member(E:C_viejo, Visitados),
     G < C_viejo) ->
    (buscarEliminar(E, RestoCandidatos, PrefijosParaE,
                   N_RestoCandidatos),
     actualizarVisitados(E:G, Visitados, N_Visitados),
     Diferencia is C_viejo - G,
     expandirCamino([E|Camino]:G+H, PrefijosParaE, Diferencia,
                   OtrosCandidatos),
     insertar(OtrosCandidatos, N_RestoCandidatos,
              NN_RestoCandidatos),
     procesar(Candidatos, NN_RestoCandidatos,
              N_Visitados, ListaCandidatos, ListaVisitados));
    (procesar(Candidatos, RestoCandidatos,
              Visitados, ListaCandidatos, ListaVisitados)).

```

El Programa se completa con la siguiente colección de predicados auxiliares, cuyos significados se explican por sí mismos.

```

% Estimacion de la funcion heuristica h*
% estimarNh(X,Y,F,Nh)

```

```
% Nh es una estimacion de la distancia entre el punto (X,Y) y el
% objetivo F.
estimarNh(X,Y,F,Nh) :- (n(_,_,_)<==>n(F,X2,Y2):_ ;
                           n(F,X2,Y2)<==>n(_,_,_):_),_
                           !, estimarNh(X,Y,X2,Y2,Nh).

% estimarNh(X1,Y1,X2,Y2,Nh)
% Nh es la distancia en linea recta entre los puntos (X1,Y1) y
% (X2,Y2). Este tipo de estimacion asegura que h* =< h (i.e.,
% h* no sobreestima a h).
estimarNh(X,Y,X,Y,0).
estimarNh(X1,Y1,X2,Y2,Nh) :- X is abs(X2-X1), Y is abs(Y2-Y1),
                               Nh is sqrt(X**2 + Y**2).

% insertar(Caminos, Candidatos, N_Candidatos)
% Inserta una lista de Caminos en la lista de Candidatos
% guardando el orden: primero el mejor camino.
insertar([], Candidatos, Candidatos).
insertar([Camino1:G1+H1|Caminos], Candidatos, N_Candidatos) :-
    insertarCamino(Camino1:G1+H1, Candidatos, NN_Candidatos),
    insertar(Caminos, NN_Candidatos, N_Candidatos).

insertarCamino(Cam1:G1+H1, [], [Cam1:G1+H1]).
insertarCamino(Cam1:G1+H1, [Cam2:G2+H2|Candidatos],
                           N_Candidatos) :-
    F1 is G1+H1, F2 is G2+H2,
    F1<F2 -> (N_Candidatos = [Cam1:G1+H1, Cam2:G2+H2|Candidatos]);
    (insertarCamino(Cam1:G1+H1, Candidatos, N2_Candidatos),
     N_Candidatos = [Cam2:G2+H2|N2_Candidatos])
   .

% eliminar(Camino, Candidatos, N_Candidatos)
% Elimina el Camino de la lista de Candidatos.
eliminar(_, [], []).
eliminar(Camino, [Camino|RestoCandidatos], RestoCandidatos) :- !.
eliminar(Camino1, [Camino2|RestoCandidatos],
          [Camino2|N_Candidatos]) :-
    eliminar(Camino1, RestoCandidatos, N_Candidatos).

% buscarEliminar(E, Candidatos, PrefijosParaE, N_Candidatos)
% Busca caminos candidatos que contengan el nodo E y los elimina,
% pero antes extrae una lista de prefijos para expandir el nuevo
% camino hasta E.
buscarEliminar(E, Candidatos, PrefijosParaE, N_Candidatos) :-_
    buscarEliminar(E, Candidatos, [], PrefijosParaE, []),
                           N_Candidatos).

% buscarEliminar(E, Candid, PrefAc, PrefijosParaE, CandAc,
%
```

```

N_Candid) .  

buscarEliminar(_, [], PrefAc, PrefAc, CandAc, CandAc).  

% Si Camino contiene E, extraer prefijo, acumularlo en PrefAc y  

% no acumular Camino. Si Camino no contiene E, ignorar prefijo,  

% no acumularlo en PrefAc y acumular Camino preservando el orden  

buscarEliminar(E, [Camino|Candidatos], PrefAc, PrefijosParaE,  

                CandAc, N_Candidatos):-  

    extraerPrefijo(E, Camino, PrefijoE) ->  

    buscarEliminar(E, Candidatos, [PrefijoE|PrefAc],  

                    PrefijosParaE, CandAc, N_Candidatos);  

    (append([Camino], CandAc, N_CandAc)),  

    buscarEliminar(E, Candidatos, PrefAc, PrefijosParaE,  

                    N_CandAc, N_Candidatos)).  

% extraerPrefijo(E, Camino, PrefijoE)  

% Si E es miembro de Camino, devuelve en PrefijoE el prefijo de  

% Camino hasta alcanzar E; si no, falla.  

extraerPrefijo(E, Camino:C+H, PrefijoE:C+H) :-  

    member(E, Camino), append(PrefijoE, [E|_], Camino).  

% actualizarVisitados(E:G, Visitados, N_Visitados),  

% Actualiza la lista de Visitados, asignando al nodo E su nuevo  

% coste G.  

actualizarVisitados(_:_ ,[],[]).  

actualizarVisitados(E:G, [E:_|Visitados], [E:G|Visitados]) :-!.  

actualizarVisitados(E1:C1, [E2:C2|Visitados],  

                    [E2:C2|N_Visitados]) :-  

    actualizarVisitados(E1:C1, Visitados, N_Visitados).  

% expandirCamino(Camino, Prefijos, Diferencia, OtrosCandidatos)  

% Expande el Camino, usando una lista de prefijos, para obtener  

% OtrosCandidatos. Equivale a la reordenación de enlaces que se  

% realiza en el punto 5b.iii del algoritmo A*.  

expandirCamino(_, [], _, []).  

expandirCamino(Cam, [Pref:C+H|Prefs], D,  

                [N_Cam:NC+H|R_OtrosCand]) :-  

    NC is C - D,  

    append(Pref, Cam, N_Cam),  

    expandirCamino(Cam, Prefs, D, R_OtrosCand).

```

RESUMEN

En este capítulo hemos presentado algunas de las aplicaciones de la programación lógica en uno de los campos más significativos de la Inteligencia Artificial (IA): la resolución de problemas. Hemos elegido este campo porque, a través de él, podemos introducir nuevas técnicas de programación (declarativa).

- La mayoría de los problemas a los que se enfrenta la IA no tienen una solución

algorítmica directa. La única forma posible de solución es la búsqueda en un espacio de estados. El esquema general de solución de este tipo de problemas responde a la siguiente estructura:

- Definir formalmente el espacio de estados del sistema: identificar una noción de estado y uno o varios estados iniciales y estados objetivo.
- Definir las reglas de producción u operaciones de movimiento: un conjunto de reglas que permiten cambiar el estado actual y pasar de un estado a otro cuando se cumplen ciertas condiciones.
- Podemos pensar en este espacio de estados (conceptual) como un grafo dirigido: los nodos son estados del problema, mientras los arcos están asociados a las reglas u operaciones. Ahora la solución de un problema consiste en encontrar un camino, en el espacio de estados, que nos lleve desde el estado inicial a un estado objetivo.
- Muchos autores separan la especificación del espacio de estados de la descripción del control. La separación entre lógica y control es una de las máximas de la programación declarativa. Por consiguiente, los lenguajes de programación lógica (Prolog) están bien adaptados para la resolución de problemas: i) permitiendo que nos centremos en la especificación del problema; ii) haciendo innecesario (en el mejor de los casos) programar la búsqueda.
- Mediante algunos ejemplos clásicos, hemos aprendido a especificar la parte declarativa de un problema, que se ajusta a la técnica de búsqueda en un espacio de estados, el problema del mono y el plátano; el problema de los contenedores de agua y el problema de la generación de planes en el mundo de los bloques.
- La estrategia de control dicta el orden en el que se aplican las reglas de producción o movimientos para generar nuevos estados. Esa circunstancia determina el orden en el que se generan dichos estados y, por lo tanto, la geometría del espacio de estados. Una parte muy importante del control es la estrategia de búsqueda, que especifica el orden en el que se visitan los estados generados en busca de un estado objetivo. Otras tareas adicionales, aunque no menos importantes, de la estrategia de control incluyen: comprobar la aplicabilidad de las reglas de producción, la comprobación de que se cumplen ciertas condiciones de terminación y registrar las reglas o acciones que han sido aplicadas.
- Un aspecto importante de las estrategias de control es la dirección en la que se realiza la búsqueda. Se distinguen dos opciones fundamentales: razonamiento hacia adelante y razonamiento hacia atrás.
- Hemos estudiado dos estrategias de exploración sistemáticas: búsqueda en profundidad y en anchura. Estas estrategias no utilizan información específica sobre el problema que se desea resolver (estrategias de búsqueda a ciegas).

- La principal característica de la estrategia de búsqueda en profundidad es que el espacio de estados se explora de forma que se visitan los descendientes de un estado antes de visitar cualquier otro nodo (hermano). Entre las ventajas de esta estrategia se pueden citar su economía en el consumo de memoria (solamente almacena información sobre el camino que se está explorando en cada momento). Su principal inconveniente es que puede quedar atrapada explorando ramas infinitas en un árbol o caminos cílicos en un grafo, por lo que pueden perderse respuestas (generándose un problema de incompletitud).
- La estrategia de búsqueda en anchura explora el espacio de estados de forma que se visitan los hermanos de un estado antes de visitar a sus descendientes. Entre las ventajas de la estrategia de búsqueda en anchura destacan las siguientes: i) Si el espacio de estados posee un camino entre un estado inicial y un estado objetivo, esta estrategia garantiza que se terminará encontrándolo. ii) La solución, de existir, se encontrará en un número mínimo de pasos. En cambio, entre las desventajas más graves podemos citar la inefficiencia en el consumo de memoria (requiere almacenar todos los estados del nivel que se está explorando).
- Las estrategias de búsqueda sin información expanden demasiados nodos antes de encontrar un camino que lleve a un estado objetivo. Es posible reducir el coste de la exploración con el auxilio de información específica sobre el problema, que suele denominarse *información heurística*. Los procedimientos de búsqueda que utilizan este tipo de información con el fin de reducir el espacio de búsqueda se denominan *estrategias de búsqueda heurística*.
- Las técnicas heurísticas tratan de aumentar la eficiencia del proceso de búsqueda aun a costa de sacrificar la completitud del proceso. En general, no debe de esperarse que una heurística obtenga la respuesta óptima para un problema, si bien proporcionará una buena respuesta en un tiempo razonable (inferior al coste teórico exponencial que pueda tener ese problema).
- Una *función heurística* es una aplicación entre las descripciones de los estados del problema y una medida o estimación de lo prometedor que es un estado en el camino para alcanzar un estado objetivo. De forma simple, la idea es continuar el proceso de búsqueda a partir del estado más prometedor de entre los del conjunto de estados expandidos por la aplicación de las reglas de movimiento que caracterizan el problema.
- En este capítulo se han estudiado diversas técnicas heurísticas: i) heurística del vecino más próximo y método de escalada (heurística local y control irrevocable); ii) búsqueda *primero el mejor* (heurística global y control tentativo).
- La estrategia de búsqueda *primero el mejor* es una estrategia sistemática que permite incorporar en su seno diversas funciones heurísticas dependiendo de la natu-

raleza del problema que se desea resolver. Esta característica le dota de una gran flexibilidad como herramienta de resolución de problemas.

- La estrategia de búsqueda *primero el mejor* intenta combinar las ventajas de la búsqueda en profundidad con las de la búsqueda en anchura. Si existe un camino hasta un nodo objetivo, por una parte lo encontrará sin tener que explorar todas las ramas (como la búsqueda en profundidad) y, por otra, el camino que encuentre será óptimo (de forma similar a como la búsqueda en anchura encuentra un camino de longitud mínima), si se cumplen ciertas condiciones de admisibilidad.
- El proceso de búsqueda primero el mejor parte de un estado inicial que se incluye en una lista de nodos a inspeccionar. En cada paso, el algoritmo selecciona el nodo más prometedor que se haya generado hasta el momento y no haya sido inspeccionado. Habitualmente se usa una función de evaluación con una componente heurística para determinar lo prometedor que es un nodo. El nodo seleccionado se expande, aplicando las reglas de movimiento que le son aplicables, para generar sus sucesores. Si alguno de los nodos sucesores es un nodo objetivo, la inspección termina. En caso contrario, los nodos sucesores se añaden a una lista de nodos por inspeccionar. Nuevamente, se selecciona el más prometedor de entre los de la lista y el proceso continua de la misma forma.
- En algún sentido, este procedimiento es, similar al método de escalada (por la máxima pendiente), excepto en dos aspectos: i) el método de escalada, al seleccionar un estado (producido por una regla de movimiento) abandona el resto de las alternativas, que nunca se vuelven a considerar; ii) la búsqueda *primero el mejor* selecciona el mejor estado disponible, aun si éste tiene un valor mayor (y, por tanto, es menos prometedor) que el que se estaba explorando en el último paso, mientras que el proceso de escalada se detiene en cuanto no se encuentra un estado sucesor mejor que el estado actual.

CUESTIONES Y EJERCICIOS

Cuestión 9.1 *La técnica de resolución de problemas mediante búsqueda en un espacio de estados:*

- *Necesita de una definición exhaustiva de las reglas de producción o movimiento, aunque no es imprescindible que éstas produzcan un cambio local.*
- *Es preferible aun cuando existan soluciones algorítmicas.*
- *Utiliza términos para representar los estados, cuando se utiliza la lógica de predicados.*

- *Es comparable a los sistemas de producción.*

Seleccione la respuesta correcta de entre las anteriores.

Cuestión 9.2 *¿Cuándo conviene aplicar la técnica de resolución de problemas mediante búsqueda en un espacio de estados?*

Cuestión 9.3 *Enumere los pasos que deben darse cuando se resuelve un problema mediante la técnica de búsqueda en un espacio de estados.*

Cuestión 9.4 *Referente a la técnica de resolución de problemas mediante búsqueda en un espacio de estados, indique qué afirmación es falsa:*

- *Necesita de una definición exhaustiva de las reglas de producción.*
- *Es preferible aún cuando existan soluciones algorítmicas.*
- *Representa el espacio de estados (conceptual) como un grafo.*
- *La estrategia de control debe cumplir los requisitos de necesidad de cambio local y de cambio global.*

Ejercicio 9.5 *Resuelva el problema de las ocho reinas estructurándolo como un problema de búsqueda en un espacio de estados. ¿Es más eficiente la especificación propuesta que las anteriores soluciones propuestas para este problema? Diseñe un método cuantitativo efectivo para medir qué solución es más eficiente.*

Ejercicio 9.6 *Modifique el programa del mono y el plátano para que el plan (i.e., la sucesión de acciones que llevan a la solución) se almacene en una lista que se devuelve como solución al problema.*

Ejercicio 9.7 *Defina un autómata capaz de reconocer el lenguaje a^*bc^* (es decir, secuencias de caracteres formadas por cero o más a s seguidas de una b y cero o más c s) e impleméntelo usando el lenguaje Prolog. Utilice las técnicas habituales para la solución de problemas mediante búsqueda en un espacio de estados.*

Cuestión 9.8 *Describa las funciones de una estrategia de control y enumere los puntos del proceso de búsqueda en los que se toman las decisiones de control.*

Cuestión 9.9 *¿Qué se entiende por grafo de estados implícito en un problema de búsqueda?, ¿y por grafo de estados explícito?*

Cuestión 9.10 *Compare las estrategias de búsqueda en profundidad y en anchura, resumiendo sus ventajas e inconvenientes.*

Ejercicio 9.11 Modifique el programa de las contenedores de agua para que contemple la detección de posibles ciclos en el plan que conduce a la solución.

Ejercicio 9.12 Un granjero tiene un lobo, una cabra y una col (muy grande). Quiere llevarlos consigo y atravesar un río, pero su barco solamente tiene cabida para dos ocupantes (un elemento más el granjero mismo). Si deja solos al lobo y a la cabra, el lobo se comerá a la cabra. Sin embargo, si deja a la cabra sola con la col, la cabra se comerá a la col. ¿Cómo puede atravesar el río sin dejar el lobo solo con la cabra o la cabra sola con la col?

Ejercicio 9.13 Modifique el procedimiento de búsqueda en profundidad del Apartado 9.3.1 de forma que evite el recorrido de caminos acíclicos infinitos en un espacio de estados. Más concretamente, defina un predicado `resolver2(E,Sol,Profundidad)` que, dado un estado inicial `E`, devuelva una solución (plan) en `Sol` de longitud menor o igual a `Profundidad`.

Ejercicio 9.14 Modifique el Algoritmo 1 del Apartado 9.3.2 de forma que tenga constancia de los estados previamente generados y se pueda evitar el recorrido de caminos cílicos en un grafo.

Ejercicio 9.15 Modifique la solución al problema de la búsqueda de un camino en el grafo representado en la Figura 9.4 del Apartado 9.3.2 marcando el estado `a` como inicial (introduciendo el hecho “`inicial(d).`”) y eliminando `v` de entre los estados objetivo (borrando el hecho “`objetivo(v).`”).

- a) Compruebe que la primera versión del predicado `anchura` no termina, cuando se lanza la pregunta “`?- anchura(d).`”, mientras la segunda sí que termina.
- b) Use el comando `trace` y represente los espacios de búsqueda generados por ambas versiones para las preguntas “`?- anchura(a).`” y “`?- anchura(d).`”. Compruebe que, cuando se usa la segunda versión del predicado `anchura`, al realizar la primera pregunta se visitan exactamente los nodos del grafo original, mientras que, al realizar la pregunta “`?- anchura(d).`”, se visita un fragmento del mismo.

Ejercicio 9.16 (Misioneros y caníbales) Tres misioneros y tres caníbales se encuentran en una orilla de un río. A todos ellos les gustaría pasar a la otra orilla. Los misioneros no se fían de los caníbales. Por ello, los misioneros han planificado el viaje de forma que el número de misioneros en cada orilla del río nunca sea menor que el número de caníbales en esa misma orilla. Solo disponen de una lancha de dos plazas. ¿Cómo podrían atravesar el río sin que los misioneros corran peligro de ser devorados por los caníbales? Resuelva el problema usando una estrategia de búsqueda: a) en profundidad; b) `anchura`.

Ejercicio 9.17 (Búsqueda de coste uniforme (Dijkstra, 1959)) La estrategia de búsqueda de coste uniforme es una extensión de la búsqueda en anchura para resolver problemas en los que se desea encontrar un camino de coste mínimo. La idea es seleccionar, en cada paso, el nodo que tienen asociado un menor coste, medido desde el nodo inicial. Así se sigue la alternativa menos costosa en cada momento. Si un camino se transforma en más costoso, es abandonado para seguir el camino que, en ese momento, sea menos costoso. El proceso continua hasta encontrar un nodo objetivo (o terminar con fallo). a) Modifique el Algoritmo 1 de búsqueda en anchura para que se adapte a los requisitos de una búsqueda de coste uniforme. (Ayuda: Asocie a cada arco que une dos nodos k y l , una función de coste $c(k, l)$. El coste de un camino, desde el nodo inicial hasta un nodo n , es la suma del coste de cada uno de los arcos que lo componen y puede denotarse por $g(n)$. Expanda los nodos n con $g(n)$ mínimo.) b) Para el grafo de la Figura 9.4, implemente un programa Prolog que materialice el algoritmo de búsqueda de coste uniforme y halle un camino de coste mínimo entre el nodo a y un nodo objetivo.

Cuestión 9.18 Relativo a la estrategia de búsqueda de coste uniforme del problema anterior, indique qué afirmación es falsa:

- En cada paso, expande los nodos de menor coste.
- En cada paso, expande los nodos que están a la misma profundidad.
- Puede verse como un caso especial de la búsqueda heurística.
- Es un algoritmo de búsqueda sistemático.

Ejercicio 9.19 Liste el orden en el que se visitan los nodos del árbol de la Figura 9.10, para cada una de las siguientes estrategias de búsqueda: a) en profundidad; b) en anchura; c) de coste uniforme. Supóngase que, cuando se carezca de un criterio mejor, se selecciona el nodo más a la izquierda. Los valores que etiquetan los arcos de las ramas del árbol expresan el coste necesario para recorrerlos.

Cuestión 9.20 Describa las analogías y diferencias existentes entre el método de escalada y la estrategia de búsqueda primero el mejor.

Ejercicio 9.21 Dados cuatro dados alineados en fila y una situación de partida arbitraria, se pretende manipularlos de forma que, al final, todos los dados muestren la misma cara. Resuelva el problema aplicando el método de la escalada. (Ayuda: defina una función de evaluación cuyo valor máximo se alcance con la configuración objetivo).

Cuestión 9.22 Compare los algoritmos que definen las estrategias de búsqueda en anchura y primero el mejor. Analice sus estructuras de datos.

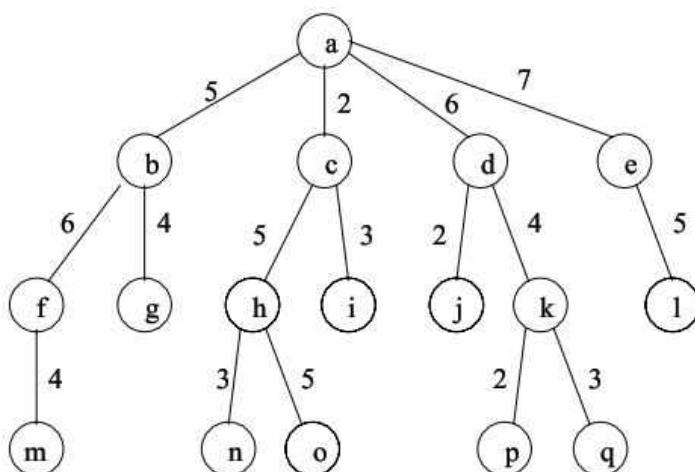


Figura 9.10 Un ejemplo de árbol de búsqueda.

Ejercicio 9.23 Suponga que utilizamos el programa de búsqueda del mejor camino entre dos puntos en un grafo para trazar rutas entre ciudades. Amplie el programa para que contemple opciones que permitan: a) Hallar el camino más rápido. (**Ayuda:** Añada información de coste temporal. Tenga en cuenta que atravesar ciudades tiene un alto coste temporal y modifique la función heurística para tener en cuenta esta circunstancia.) b) Obtener o bien la ruta más corta o la más rápida entre dos puntos pasando por una secuencia de puntos de interés.

Cuestión 9.24 Indique qué elección de las funciones g^* y h^* se necesita para que el algoritmo primero el mejor se comporte como un algoritmo de búsqueda de coste uniforme.

Ejercicio 9.25 (Mundo de los robots) Considere de nuevo el Problema 8.4, de los robots en un mundo bidimensional con obstáculos. Diseñe un programa que permita a un robot desplazarse entre dos puntos sin chocar con ningún obstáculo o la pared, minimizando la longitud del camino. (**Ayuda:** Recuerde que el camino más corto entre dos puntos es la línea recta.)

Ejercicio 9.26 (puzzle-8) El problema del puzzle-8 (Figura 9.11) consiste en ordenar ocho fichas numeradas que pueden moverse sobre un tablero de 3×3 . Uno de los cuadros queda siempre vacío, lo que permite mover hacia él una de las fichas adyacentes (lo que podría interpretarse también como un movimiento del cuadro vacío). La Figura 9.11 muestra una configuración inicial y otra final para este problema. Determine un plan que permita pasar desde la configuración inicial a la final.

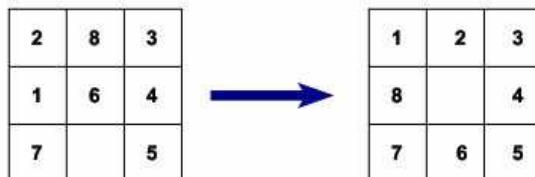


Figura 9.11 Estados inicial y final para el problema del puzzle-8.

Ejercicio 9.27 ([104]) Considerere un puzzle consistente en un vector de 7 posiciones, con la siguiente configuración inicial:

N	N	N	B	B	B	V
---	---	---	---	---	---	---

Hay tres fichas negras (N), tres blancas (B) y una casilla vacía (V). El puzzle permite los siguientes movimientos:

1. Una ficha puede moverse a una casilla vacía adyacente, con coste unidad
2. Una ficha puede saltar sobre otras dos (como máximo) hasta una celda vacía, con coste igual al número de fichas sobre las que ha saltado.

El objetivo del puzzle es tener todas las fichas blancas a la izquierda de todas las negras (sin importar la posición de la celda vacía). a) Defina una función heurística, h , para este problema; b) Obtenga un plan que resuelva este problema.

Ejercicio 9.28 Implemente un programa Prolog que permita jugar a las tres en raya. Utilice una función heurística como la sugerida en el Ejemplo 9.1. (**Ayuda:** Consulte el Apartado 3.4 de [104] o el Capítulo 12 de [105], que aborda el estudio de estrategias de búsqueda para juegos con dos jugadores.)

Programación Lógica y Tecnología Software Rigurosa

A diferencia del hardware, generalmente los sistemas software todavía carecen de garantías industriales de buen funcionamiento. Con el crecimiento explosivo de las “tecnologías de la información”, la distinción entre “sistemas críticos” e “informática de consumo” se ha reducido drásticamente, por lo que la ausencia de garantías de calidad del software resulta inadmisible hoy en día en todos los ámbitos y no solo ya en aquellos dominios de aplicación donde las consecuencias de un fallo han resultado tradicionalmente críticas.

Para hacer posible la deseable transferencia tecnológica a la industria de los resultados de la investigación sobre técnicas de especificación y verificación formal de programas, es necesario reducir drásticamente los costes de la especificación y automatizar completamente las tareas. En este capítulo vamos a explorar la estrecha relación que existe entre los métodos formales, con base en la lógica, y la concepción de técnicas y herramientas automáticas para dar soporte sistemático y racional al desarrollo del software. En contraste con otros métodos formales más convencionales (y a la vez poco prácticos), que fomentan una formalización excesiva mediante el empleo de nociones que requieren una formación matemática poco habitual en los usuarios finales, la tecnología declarativa se presenta como una aproximación muy efectiva en este área para acercarse a los objetivos de calidad, fiabilidad y seguridad de los productos software, que entronca con la corriente actual conocida como aproximación ágil o ligera (*lightweight*), basada en una aplicación selectiva y focalizada de los métodos formales, que resulta más efectiva y rentable en la práctica que otros métodos más tradicionales.

Nuestra hipótesis de partida en este capítulo es que las tecnologías rigurosas y declarativas, en apoyo de una orientación ágil, pueden servir de base sólida para el desarrollo de herramientas, métodos y lenguajes de soporte para el desarrollo de software fiable y de alta calidad. Siguiendo un enfoque moderno, que tiene en cuenta los tres elementos de la trilogía del software —programas, datos y propiedades—, en lo que sigue vamos

a describir algunos procesos formales que transforman dichas componentes automáticamente. Esto incluye, entre otros, los siguientes mecanismos: análisis, transformación, síntesis y depuración automática de programas.

10.1 MOTIVACIÓN

Los sistemas informáticos desempeñan un papel esencial en la Sociedad de la Información. A medida que la sociedad se implica más en tales sistemas (a través de Internet, las redes de telecomunicaciones, etc) y crece su dependencia hacia ellos, se hace patente la necesidad de asegurar la corrección de su comportamiento. La creciente avalancha de nuevas posibilidades que facilita Internet para el desarrollo diario de las empresas, las instituciones y los hogares soslaya, y a veces oculta, el hecho bien conocido (o sospechado) de que el *software* al que actualmente confiamos nuestras necesidades informáticas es poco seguro. Los usuarios de las nuevas tecnologías, trabajando en un contexto global donde cada vez más gente y organizaciones entran en contacto, cooperan y comercian, encuentran cada vez más frustrantes las consecuencias derivadas de los fallos del software (como el tener que reiniciar el equipo para recuperar un comportamiento estable del sistema), por no hablar de las repercusiones que pueden tener estos fallos en las áreas donde la seguridad es crítica.

Numerosas referencias bibliográficas se hacen eco de errores catastróficos que redundan en pérdidas irrecuperables, como el fallido lanzamiento del cohete Ariane 5, con un coste cercano a los 3.000 millones de euros, o la deficiente gestión informática de las olimpiadas de Atlanta en 1996. Otros errores de concepción del *software*, como el responsable del famoso Y2K (*efecto 2000*), han sido difundidos ampliamente por los medios de comunicación. La fragilidad de la infraestructura software y el coste de los errores es un problema con gran impacto social y económico (hay estudios que lo cifran en un 80 % del coste total de un proyecto software). Más que nunca, lo que el mercado informático demanda es calidad del software, entendido como una suma de atributos: fiabilidad, seguridad, robustez, corrección y eficiencia del software, entre otros. De hecho, la necesidad de garantizar todos estos atributos se hace patente no solo en áreas tradicionales de seguridad crítica donde en coste y repercusión de un fallo son inaceptables, sino también en el “software de consumo”, que es vital para el éxito de otros servicios, productos o negocios.

Es habitual hoy en día, por ejemplo, actualizar el núcleo de un SO en nuestros PC con componentes que provienen de fuentes remotas, como controladores de dispositivos que uno descarga por la red. Este código sospechoso tiene acceso al espacio de direcciones que maneja el SO, y la seguridad se convierte en un requerimiento igualmente crítico en este contexto. De hecho, es cada vez más frecuente que los sistemas software se actualicen de forma dinámica, bajando componentes de código que provienen de fuentes probablemente inseguras (pensemos cuántas veces actualizamos a través de la

red la versión de nuestro navegador, o descargamos una aplicación o un parche para el sistema operativo) Muchas veces sucede sin nuestro consentimiento o incluso sin conocimiento. En estas circunstancias resulta necesario reforzar las técnicas convencionales (por ejemplo criptográficas, en el caso de la seguridad), con mecanismos para detectar y prevenir un comportamiento anómalo de los programas.

Para alcanzar el elevado nivel de prestaciones que deben satisfacer los complejos sistemas informáticos modernos, su desarrollo, explotación y mantenimiento requieren la utilización, durante las distintas etapas de su vida útil, de técnicas y herramientas de asistencia (preferentemente automática). Por naturaleza, las técnicas y métodos automáticos deben descansar en sólidos fundamentos formales que permitan al usuario de los mismos (ingeniero de requisitos, programador, implementador o gestor) confiar en la aplicación bajo cualquier circunstancia. Ésta es la perspectiva que guía el uso de la lógica con fines computacionales. Un control total de la calidad solo puede obtenerse por un proceso riguroso que cubra todas las fases del desarrollo de software y que llamaremos genéricamente *tecnología rigurosa*.

Con el término *tecnología rigurosa* nos referimos a toda una serie de métodos, modelos, lenguajes de especificación y programación, y herramientas que permiten un desarrollo del software basado en principios semánticamente bien fundados y que aseguren su adecuación a los requisitos. En general podemos también aplicar el nombre de Tecnologías Declarativas a todos estos recursos dado que la lógica proporciona a la ingeniería del *software* el marco científico y tecnológico adecuado para el devenir de una verdadera ingeniería, creadora y sustentadora de la tecnología del *software* que la sociedad de la información necesita. La tecnología resultante descansa en bases formales sólidas que permiten garantizar la corrección y efectividad de las técnicas desarrolladas. A diferencia de otros métodos formales más convencionales (y a la vez poco prácticos), que fomentan la formalización excesiva mediante el empleo de lenguajes demasiado expresivos que requieren una formación matemática poco habitual en los usuarios finales del producto, la lógica proporciona una vía para la aproximación *lightweight* que integra armónicamente una variedad de lenguajes, herramientas y técnicas.

El término *lightweight* se hizo popular a finales de los años 90 para referirse a métodos de desarrollo alternativos a los habitualmente propuestos en el campo de la Ingeniería del Software, aunque ya fue acuñado a mediados de los 90 para referirse al uso parcial de los métodos formales [69]. La idea fundamental de estos métodos, que comenzaron a denominarse *ágiles*, es la eliminación del exceso de “burocracia” tradicionalmente implícita en la aplicación de las metodologías, promoviendo su agilización.

Uno de los inconvenientes que ha presentado tradicionalmente la aplicación de los métodos formales en la Informática, y particularmente al *software*, ha sido la dificultad para realizar una transferencia tecnológica efectiva de resultados a la industria. Esta dificultad se debe, en gran parte, al alto coste que supone una verificación formal completa y que, a excepción de los entornos de seguridad crítica, en los demás resulta prohibitivo. En consecuencia, el coste de la especificación debe reducirse drásticamente y el análisis debe automatizarse pues, de otro modo, la industria no encontrará razones para

la adopción de los métodos formales. Con algunos métodos formales es difícil alcanzar estos objetivos. Llegamos así al nacimiento de un campo conocido como *Ingeniería del Software Automática* (ISA), entendiendo por la misma el subcampo de la Ingeniería del Software que se orienta a garantizar automáticamente la calidad del código, incluyendo la seguridad, corrección, robustez y, en general, todos los atributos por los que se mide la calidad. Nótese que se trata de un salto cualitativo respecto al estado actual de la ciencia, y no simples mejoras de la tecnología existente o pequeños avances en metodologías, plataformas o modelos propios de la investigación en Ingeniería del Software. Debemos pensar en métodos formales que no requieran una formación matemática inasumible en los usuarios finales o una potencia de cálculo y unos recursos inexistentes en informática de consumo. Un método formal “ligero” (*lightweight*), que enfatice la parcialidad y la aplicación focalizada, puede traer grandes beneficios a un coste reducido en este contexto. Este es el caso de los métodos que se sustentan en la tecnología declarativa.

10.1.1. Una aproximación *lightweight*

Para evitar los problemas de las aproximaciones formales más tradicionales, que generalmente resultan demasiado rígidas y difícilmente rentables en la práctica, esta aproximación se basa en integrar diferentes lenguajes y técnicas buscando automatizar ciertas fases que no se realizan de forma automática en otras aproximaciones más clásicas, como la inferencia de especificaciones a partir de ejemplos, la verificación automática del código, la depuración racional de los programas o la estimación de las mejoras obtenidas mediante transformaciones formales del código. Este tipo de aproximación permite un uso rápido y a bajo coste de los métodos formales, al adaptarse fácilmente a las necesidades particulares e inmediatas de cada proyecto.

Los criterios que guían este enfoque son [69]:

- Parcialidad (por ejemplo, en el lenguaje, en el enfoque o en el modelo).
- Selectividad (de notaciones, componentes, herramientas, ..., dependiendo de cada dominio, fase, versión, nivel o problema particular).
- Integración (de métodos, estilos, lenguajes; por ejemplo, en un mismo sistema cabe usar un lenguaje lógico o funcional para especificar o programar parte de la aplicación y combinar el desarrollo con técnicas de inducción como soporte para la corrección de errores en el código).
- Transparencia al usuario final (usuarios no expertos).

En algún sentido, esta aproximación a los métodos formales aprovecha algunas de las buenas ideas de la *extreme programming* [11] que, frente a las metáforas propias de otras metodologías más rígidas (llamadas a veces, metodologías *monumentales*), es “adaptativa” (a los conocimientos del equipo de desarrollo, a los cambios en los requerimientos, etc.).

10.2 LA TRILOGÍA DEL SOFTWARE

Los métodos formales de la ingeniería del software se suelen entender como el uso de técnicas basadas en distintas teorías matemáticas (generalmente, la lógica y el álgebra) para la *especificación* y *verificación* de programas. Esta concepción de los métodos formales es parcial e incompleta: son muchos más los distintos procesos formales que pueden realizarse mecánicamente durante el desarrollo del software. En este apartado se introduce una visión considerablemente más amplia, basada en la denominada *trilogía del software*.

En el proceso de desarrollo del software se manejan habitualmente tres tipos de componentes [97]: los *programas* (con su documentación asociada), las *propiedades* (incluyendo aquí especificaciones, tipos, restricciones temporales, requerimientos de corrección, prestaciones, etc) y los *datos* (bien sean una selección de pares de entrada/salida o trazas obtenidas al observar los cálculos intermedios).

Cuando se disponen los programas, propiedades y datos formando un triángulo, los arcos que unen los distintos vértices del triángulo representan los procesos que permiten producir un tipo de elementos a partir de otros (véase la Figura 10.1). Esta visión del desarrollo de un sistema software no impone ningún orden particular ni dependencia funcional entre los distintos procesos. Esto permite escoger el orden más adecuado en función de las destrezas y conocimientos del personal involucrado en el desarrollo y de las peculiaridades del problema (por ejemplo, en una aplicación cabe escoger entre derivar los juegos de datos a partir de los requerimientos o, por el contrario, inferir –i.e., “aprender”– las especificaciones a partir de los ejemplos).

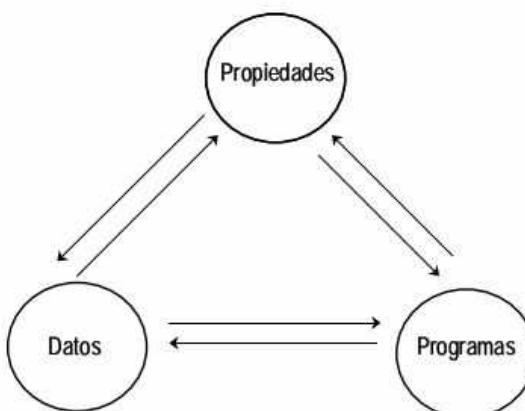


Figura 10.1 La Trilogía del Software

10.2.1. Datos, propiedades y programas

En lo que sigue caracterizamos, con mayor detalle, los distintos componentes de la trilogía del software.

Programas

Los programas son documentos formales en tanto que deben someterse a un procesamiento automático, como el que se realiza con herramientas como compiladores e intérpretes. La sintaxis y semántica del lenguaje en que están escritos los programas determina sus propiedades formales. Aunque la sintaxis suele estar mejor formalizada que la semántica (al menos en los lenguajes utilizados más comúnmente), la existencia de marcos formales potentes como la semántica denotacional, las técnicas de punto fijo, etc., permiten dar soporte formal a herramientas automáticas como las que se aplican en el análisis, verificación y optimización mecánica de programas.

Propiedades

En un modelo ideal de desarrollo del software como es el ciclo de vida en cascada, los requisitos del sistema se determinan antes de comenzar el diseño de los programas. Una aproximación interesante pero menos explorada consiste en explotar la dirección inversa de este lado del triángulo: derivar la especificación, propiedades, requisitos, etc. a partir de los propios programas. Esto se puede hacer mediante herramientas automáticas de inferencia de tipos o de análisis de programas (basadas, por ejemplo, en técnicas de interpretación abstracta), entre otras posibilidades.

En un modelo de desarrollo basado en el *prototipado automático*, las especificaciones pueden ser ejecutables; en este caso, se comportan a la vez como requisitos y como programas. Por este motivo, resultan interesantes los lenguajes de especificación ejecutables que, además de proporcionar un vehículo expresivo claro y conciso, enemigo de la ambigüedad propia de las especificaciones de requerimientos presentadas en lenguaje natural, permiten considerar la propia especificación como un programa y, en consecuencia, ejecutar de forma inmediata la especificación definida. Esto proporciona una forma simple de prototipado, que permite al ingeniero de requisitos obtener una primera aproximación de los resultados que su especificación produciría en caso de ser (correctamente) implementada. De esta forma pueden desenmascararse deficiencias de planteamiento o comportamiento que podría resultar muy costoso detectar y corregir en fases más avanzadas del proceso de producción de *software*. La utilidad del prototipado se ve reforzada por la disponibilidad de herramientas de generación (y optimización) automática, que aplican transformaciones basadas en la semántica, cuya corrección y efectividad están garantizadas formalmente. Esta forma de prototipado *asistido* contribuye a facilitar la generación de la especificación definitiva.

En otros casos, las especificaciones pueden utilizarse como oráculos que permiten comprobar la corrección de los programas. Este es el caso de algunas técnicas empleadas en la depuración racional, declarativa o algorítmica de programas [138].

La idea de la *depuración declarativa* se basa en la idea de que, en un lenguaje declarativo, la noción de error es declarativa. Es decir, es posible identificar pequeñas piezas de código como erróneas de forma independiente de cualquier semántica operacional. Esta idea fue inicialmente propuesta por Shapiro [136] para el caso de programas lógicos. La búsqueda de errores en un programa se puede enfocar desde el punto de vista del *conocimiento procedural*, entendido como la secuencia de operaciones aplicadas durante la ejecución, o del *conocimiento declarativo* del programa, que se usa aquí para contrastar si los resultados devueltos por el programa son correctos o no [102].

En el caso de la programación lógica pura, Levi et al. definieron en [35, 32] un marco de depuración declarativa que extiende al diagnóstico respecto a las respuestas computadas la metodología de [52, 136]. El marco no requiere determinar los síntomas con antelación y es independiente del objetivo. El esquema se basa en utilizar el operador de consecuencias inmediatas T_P para identificar los errores y tiene la ventaja de aportar un método de diagnosis independiente del síntoma.

Datos

Los datos pueden ser tanto de entrada/salida como datos internos que el programa puede utilizar o producir. Durante el desarrollo de un programa, muchas veces se utilizan *trazas*, que son secuencias de datos obtenidas en una ejecución del programa (usando un intérprete o monitor “instruido” específicamente para este propósito). Las trazas pueden existir también antes de los programas, como una forma de especificar su comportamiento y reciben el nombre de *escenarios*. Cuando los datos se usan como entrada de un proceso de aprendizaje, suelen denominarse *ejemplos* (positivos o negativos).

Otros procesos, como la verificación algorítmica, producen como salida *contraejemplos* que atestiguan que la propiedad a verificar no se cumple por el sistema y pueden ser útiles para localizar la causa de la disfunción. En general, la corrección de las salidas respecto a las entradas se comprueba mediante un oráculo (que puede ser el propio usuario si se hace manualmente o puede estar implementado mediante una especificación ejecutable).

10.2.2. Procesos formales

Los métodos formales han explorado este triángulo de forma exhaustiva y en todas direcciones. Si representamos los diferentes tipos de procesamiento posible para los distintos tipos de componentes, obtenemos un grafo como el que mostramos en la Figura 10.2. Sin ánimo de ser exhaustivos, a continuación resumimos brevemente los distintos procesos formales, o mecánicos, que pueden realizarse.



Figura 10.2 Perspectiva de los Métodos Formales de la Ingeniería del Software Automática

De propiedades a propiedades Como hemos mencionado anteriormente, en un proceso de desarrollo del software ideal, las especificaciones anteceden a los programas. En el proceso de desarrollo de programas, es habitual considerar un escenario realista representado por la siguiente espiral: (especificación informal → especificación formal incompleta → programa insatisfactorio → mejor especificación → programa más satisfactorio, etc). Este escenario muestra el proceso de desarrollo como un ciclo de vida dentro del cuál la síntesis de programas habilita una transición entre especificaciones y programas.

Desde el momento en que las especificaciones pueden ser ejecutadas, no existe una frontera nítida entre las especificaciones y los programas. Algunos marcos formales proporcionan un modelo para el *refinamiento mecánico* de especificaciones, que procede de forma iterativa e incremental –evolutiva– hasta alcanzar una versión que se considera el programa (véase la Figura 10.3). Si cada refinamiento preserva la semántica, entonces el proceso total produce un programa correcto (acorde con las especificaciones iniciales). Generalmente, estos métodos de *síntesis* están automatizados solo parcialmente, como en el caso del B-method [2].

De propiedades a programas Siguiendo el principio conocido como *derivación formal* o *síntesis de programas*, el código de la aplicación (o parte de él) puede ser obtenido (de forma asistida y sistemática) a partir de la especificación del sistema. Sin embargo, pese a que los primeros trabajos sobre derivación formal de programas se remontan a los

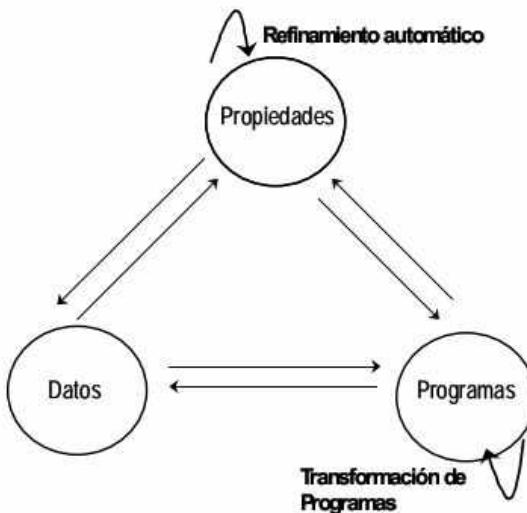


Figura 10.3 Perspectiva de los Métodos Formales de la Ingeniería del Software Automática – Ciclos

mismos orígenes de la programación como disciplina científica [44], estas técnicas han alcanzado una aceptación muy limitada, y los escasos ejemplos de aplicación práctica de este principio pasan por una automatización bastante parcial del proceso, como es el caso del B-method mencionado anteriormente.

La programación lógica proporciona un marco de trabajo inigualable para la síntesis de programas ya que las especificaciones, el proceso de síntesis y los programas resultantes pueden todos expresarse en lógica. Por este motivo, la síntesis de programas lógicos está considerada un importante paso adelante hacia una aproximación basada en la lógica para el desarrollo formal del software en cualquier paradigma. Por ello no es difícil adivinar que el tema de la síntesis automática haya recibido una gran atención desde los primeros tiempos de la programación lógica.

Los métodos de síntesis de programas lógicos pueden clasificarse como sigue:

- Síntesis constructiva, conocida también como *pruebas como programas*. A partir de la especificación de un programa P , se demuestra la conjeta de que, para toda entrada de P , existe una salida que satisface la relación entrada-salida que P computa. Entonces el programa P se extrae de dicha prueba.
- Síntesis deductiva. La especificación es un conjunto de sentencias "si y solo si" (junto con sus axiomas relevantes) que definen los predicados de P . La especificación inicial se asume completa. Las cláusulas para P se deducen directamente de

su especificación, por ejemplo reescribiendo ésta mediante transformaciones de plegado/desplegado.

Una subclase importante de la síntesis deductiva se denomina síntesis transformacional. Aquí las definiciones son cláusulas de programa y se transforman en cláusulas equivalentes mediante transformaciones de plegado/desplegado o de otro tipo, como la evaluación parcial. Este tipo de síntesis ejemplifica la estrecha y compleja relación que existe entre síntesis y transformación. La distinción entre ambas es bastante subjetiva y a menudo depende del contexto de trabajo. Una distinción posible es que la síntesis suele comenzar en una especificación no ejecutable del problema, lo cuál suele significar una descripción no recursiva del mismo, mientras la transformación suele arrancar de una especificación ya ejecutable.

- Síntesis inductiva. En este caso se parte de una especificación incompleta. La síntesis inductiva deriva P de una generalización o compleción de la especificación. Este área es un campo de trabajo en sí mismo, que recibe el nombre de Programación Lógica Inductiva.

Dadas las limitaciones de espacio, en esta presentación nos centraremos solo en proporcionar una breve introducción a la síntesis deductiva de programas.

De programas a propiedades Las técnicas más populares para extraer propiedades, especificaciones, etc, a partir de los programas incluyen el *análisis estático* [38], la *inferencia de tipos* [98] y la *verificación formal* de programas [66]. El análisis estático suele formalizarse en el marco de la teoría de la Interpretación Abstracta [38, 39, 40], que puede entenderse como una teoría de aproximación semántica que se usa para proporcionar estáticamente (en tiempo finito) las respuestas correctas a cierto tipo de cuestiones relevantes sobre el comportamiento de los programas en tiempo de ejecución. Los datos y los operadores semánticos “concretos” se reemplazan por sus respectivas versiones “abstractas”. En este contexto, un análisis se ve como una computación aproximada, definida sobre descripciones de los datos en lugar de sobre los datos mismos. Diferentes estilos de definición semántica conducen a diferentes aproximaciones al análisis de programas.

En el caso de los programas lógicos, existen dos aproximaciones principales: el análisis descendente (*top down*) y el análisis ascendente (*bottom up*) [40, 90]. La aproximación descendente, que propaga la información en el mismo sentido que la regla de resolución, es quizás la más popular. La aproximación ascendente propaga la información como en la computación del menor punto fijo del operador de consecuencias en un paso. La principal diferencia entre ambas aproximaciones está en la independencia o no del análisis respecto al objetivo, siendo independiente en el caso ascendente y dependiente en el descendente.

Podemos preguntarnos por qué es necesaria la interpretación abstracta y la respuesta es bastante directa. Deseamos poder inferir, analizar o probar propiedades dinámicas

(de la ejecución) del programa generalmente en tiempo de compilación (es decir, estáticamente). La mayoría de problemas relacionados con este análisis son problemas indecidibles o demasiado complejos. Estas dificultades provocan la necesidad de disponer de métodos, que aun siendo formales, sean parciales en el sentido de que puedan dar una respuesta correcta en algunos casos. Con la interpretación abstracta podemos dar respuestas parciales a preguntas relacionadas, por ejemplo, con el análisis del flujo de datos, como ocurre en el problema de analizar la terminación de los programas, por poner un ejemplo. Un método de análisis basado en la interpretación abstracta será capaz de dar como respuesta un *sí*, un *no* o un *no se*, en aquellos casos en que no se pueda proporcionar una respuesta fiable (por la limitación de los recursos de cómputo disponibles o porque la propia indecidibilidad del problema implica que solo es posible decidir una propiedad más fuerte que la de interés), lo que indica la parcialidad de la aproximación.

De programas a datos Los métodos de *prueba estructural* o de “caja blanca” se basan en la siguiente estrategia: primero, se eligen determinados criterios de prueba, tales como el recorrido de un conjunto de caminos que exploran todos los usos de las definiciones; a continuación, se construyen baterías de datos y se alimenta con ellos el programa en un entorno en el que se controla el cubrimiento de casos de acuerdo con el criterio considerado. El proceso se repite hasta que se alcanza un cubrimiento adecuado, que se mide como un porcentaje de las instrucciones, caminos posibles, etc. La tarea que representa el mayor desafío es la generación automática de los juegos de datos. Una aproximación muy interesante consiste en seleccionar y especificar caminos que satisfacen un criterio de prueba particular, y acumular las condiciones que definen dichos caminos mediante *restricciones*. A continuación, se resuelven dichas restricciones obteniendo los juegos de datos como soluciones de los sistemas de restricciones. Por ejemplo, dado el siguiente fragmento de código imperativo:

```

while x >do
  if x <= 10
    then %(Camino-de-Prueba-1)
    ...
    else %(Camino-de-Prueba-2)
    ...
  endif
endwhile

```

Cualquier solución a la restricción $\{x>5 \ \& \ x\leq 10\}$ acumulada en el primer camino de prueba podría usarse para validar dicho camino.

Este proceso requiere disponer de resolutores de restricciones lo suficientemente potentes como para encontrar las soluciones, si existen, o detectar su inexistencia en caso contrario, lo cual no es posible si el lenguaje es demasiado rico. En esta aproximación,

la confianza en el programa crece cuanto más amplio sea el cubrimiento considerado, aunque nunca permite garantizar la corrección total (ni parcial) del programa.

De propiedades a datos Las pruebas de programa son necesarias incluso si el desarrollo comienza con una especificación formal, y el desarrollo posterior es también completamente formal, puesto que la especificación podría contener errores y, en tal caso, la única forma de descubrirlos es comparar la especificación con otro tipo de propiedad o requisito formal. Puesto que no es conveniente esperar a validar el sistema final para encontrar el error, en ocasiones puede resultar útil validar la propia especificación. Esto se denomina *pruebas funcionales* o de “caja negra”. Similarmente a las pruebas estructurales, se trabaja con una noción de cubrimiento que, en este caso, usa *hipótesis* (u objetivos del test), escritas en el lenguaje de especificación considerado y que determinan de forma precisa los límites o alcance de las pruebas realizadas.

De datos a programas (o a propiedades) La *síntesis de programas a partir de ejemplos* [16], también conocida como *aprendizaje de programas*, representa uno de los intentos más tempranos de mecanizar el desarrollo de programas. El marco más común para este tipo de métodos es la *inferencia inductiva*. Se debe notar que no existe ninguna diferencia *a priori* entre el aprendizaje de especificaciones y el aprendizaje de programas. En ambos casos, para hacer viable y efectivo el proceso, las funciones que se inducen pertenecen a fragmentos restringidos del lenguaje de especificación o de programación considerado.

De programas a programas La compilación es la técnica más popular para producir programas a partir de otros programas. Normalmente, su salida se escribe en un lenguaje de menor nivel que el del programa de entrada y, en el caso extremo, solo resulta legible para la máquina. La compilación puede ser un proceso totalmente sintáctico durante el cuál se traduce un programa en otro, pero también puede incorporar optimizaciones que exploten la semántica del lenguaje. También es posible *transformar* programas, sin cambiar de lenguaje, para extender su funcionalidad o para derivar versiones de mayor calidad (típicamente más eficientes). Esto último puede lograrse mediante técnicas de especialización y evaluación parcial de programas [37, 72], que potencian el diseño de programas genéricos, que pueden usarse en varios contextos y después especializarse de forma mecánica para mejorar su eficiencia.

10.2.3. Hiper–arcos y otros procesos formales

Existe otra forma de ir de datos (y programas) a otros programas: mediante las técnicas de *depuración racional*. Cuando un programa no supera las pruebas, es posible usar los datos que demuestran que el comportamiento es incorrecto como un *síntoma*, que ayudará a localizar y corregir el error. Esto puede hacerse automáticamente, usando la especificación (o una abstracción finita de su semántica) a modo de un oráculo que

permite encontrar las partes del programa que son incorrectas (por ejemplo, mediante técnicas descendentes (*top-down*), basadas en la exploración de árboles de computación, o usando técnicas ascendentes (*bottom-up*), que se basan en el operador de consecuencias inmediatas), como en los depuradores declarativos de los programas lógicos [138].

Puesto que estas técnicas usan como entrada tanto los casos de prueba (datos) fallidos como un oráculo (especificación del problema), y producen como salida un programa (el programa ya corregido), no pueden entenderse estrictamente como arcos que van de datos a programas sino más bien como hiper-arcos que relacionan los tres vértices del triángulo: programas, propiedades y datos. Como fase final del proceso, el programa puede corregirse mecánicamente, usando ejemplos (que pueden generarse automáticamente a partir de la especificación) para inducir el código correcto.

A modo de ejemplo, considere el siguiente programa lógico, que es incorrecto respecto a la semántica habitual del predicado *par*:

```
par(0)
par(s(X)) ← par(X)
```

Consideremos los siguientes ejemplos positivos *par(0)*, *par(s(s(0)))* y el siguiente ejemplo negativo *par(s(0))* y veamos de qué forma es posible “aprender” el código correcto. Aplicando un paso de transformación de desplegado a la segunda cláusula del programa (véase 10.3.1), obtenemos el siguiente programa transformado:

```
par(0)
par(s(0))
par(s(s(X))) ← par(X)
```

en el cuál la segunda de las cláusulas puede eliminarse ya que cubre solo un ejemplo negativo y ninguno positivo. Una vez eliminada, se obtiene el programa correcto esperado:

```
par(0)
par(s(s(X))) ← par(X)
```

Las técnicas de transformación de programas también desempeñan un papel primordial en la moderna tecnología de *model-checking* [29, 112], que permite verificar formalmente algunas propiedades de interés de los programas (en particular, de los programas reactivos: equidad, alcanzabilidad, vivacidad, etc.). Para ello, primero se especifica formalmente la propiedad a analizar (por ejemplo, usando algún tipo de lógica temporal, como la lógica LTL o la CTL). A continuación, se transforma el programa en un grafo (estructura de Kripke) que representa todos los estados por los que puede pasar el programa en ejecución; seguidamente, se compacta la estructura de Kripke usando generalmente técnicas de digitalización que permiten obtener una representación binaria conocida como árbol de decisión binario (BDD). La solución (el punto fijo del operador asociado a la relación de transición del grafo), si existe, representa el conjunto de estados que satisfacen la propiedad temporal verificada. En caso de que la propiedad no se veri-

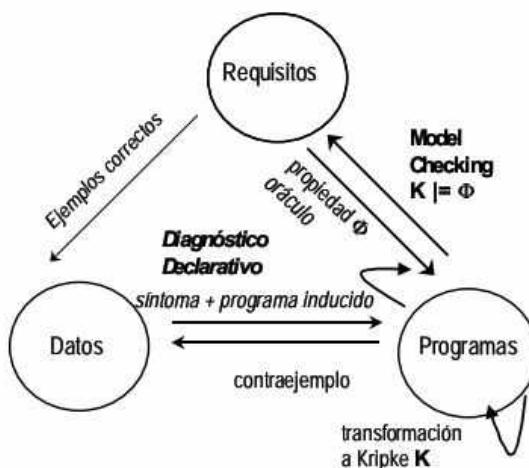


Figura 10.4 Perspectiva de los Métodos Formales de la Ingeniería del Software Automática - Hiper-arcos

fique, el método entrega un contraejemplo. Un *model-checker* explora exhaustivamente un espacio de estados astronómico (10^{120} estados/seg.) y permite encontrar muchos más errores que las técnicas de prueba convencionales. En comparación con las técnicas de demostración de teoremas, que se aplican también a la verificación formal de programas, la ventaja del model-checking es que se aplica de forma totalmente automática y sin requerir una habilidad especial en lógica matemática por parte del usuario.

Las técnicas de verificación algorítmica o *model checking* [29, 112] tienen el triple atractivo de que:

- Son completamente automáticas.
- Su aplicación no requiere supervisión por parte del usuario ni experiencia en disciplinas matemáticas (contrariamente a las técnicas de verificación declarativa más clásicas, que se basaban en el uso experimentado, por parte del usuario, de sistemas de axiomas y reglas de inferencia).
- En el caso de no verificar la propiedad analizada, la herramienta produce un contraejemplo que ayuda a identificar la fuente del error.

Estas ventajas y el uso de técnicas simbólicas, que permiten la enumeración explícita de un número astronómico de estados, han revolucionado el campo de la verificación formal transformándolo de disciplina académica en tecnología de amplio uso industrial.

De forma análoga a las técnicas de depuración racional, la verificación algorítmica de programas toma varios tipos de entradas (en particular, el programa y la propiedad a verificar) y produce varias salidas (la especificación de los estados iniciales que verifican la propiedad o el contraejemplo que demuestra que ésta no es cierta). En el triángulo del desarrollo del software, consecuentemente, el *model-checking* se representa también como un hiper-arco que incluye relaciones entre los tres vértices (véase la Figura 10.4).

Por otro lado, en un contexto abierto de programación, a través de Internet, resulta conveniente interponer barreras de acceso seguras, basadas en técnicas formales de certificación de código que es posible reinterpretar en términos de aproximación de propiedades y verificación de dichas aproximaciones. El proceso de certificación puede estar basado en un análisis estático del código para determinar si éste satisface una determinada política de accesos y flujos de información, especificada por las clases de información que pueden albergar las variables del programa y las relaciones legales entre diferentes clases. La certificación automática combina las técnicas de análisis estático con la verificación algorítmica y la depuración racional del código.

Sin embargo, cuando se plantea clásicamente la verificación de ciertas propiedades de un sistema software, no se tiene en cuenta la forma de garantizar a los posibles clientes/usuarios de dicho sistema que esa “certificación” se ha realizado. Desde el punto de vista de la efectividad, no es admisible que el cliente tenga que revalidar la propiedad ya verificada por un proveedor de código. Para evitar esto, las técnicas de *proof-carrying code* (código con demostración asociada) permiten adjuntar al código (sea fuente o binario) la “demostración de la verificación” realizada. Esto permite al consumidor de código verificar que el código proporcionado por un proveedor sospechoso se adhiere a una serie de reglas (*política de seguridad*) que garantizan un buen comportamiento de los programas en algún sentido deseado como, por ejemplo, la limitación en el uso de recursos: gestión segura y acotada de memoria, gestión segura de tipos, acceso controlado a ficheros, manejo de recursos con candados, limitación de ancho de banda, consumo acotado de tiempo de procesamiento, terminación, determinismo, etc.

La seguridad del código móvil es un problema actual que ha recibido notable atención desde el punto de vista de los mecanismos de monitorización (*firewalls*) y certificación por parte de terceros autorizados, si bien la certificación automática con un enfoque preventivo como el expuesto en este apartado está mucho menos desarrollada. El código móvil puede ser el código de una página Web en algún lenguaje de marcado como XML o el contenido activo o dinámico de páginas Web en algún lenguaje de programación (Java, JavaScript, Perl,...). El contenido activo de la página es código ejecutable que puede contener errores cometidos de buena fe que, a pesar de ello, causarían daños en la integridad de los datos o del software del sistema llegando a comprometer incluso su disponibilidad. En otras ocasiones, el contenido activo de las páginas puede contener código malicioso, desarrollado con intención de obtener información confidencial del sistema local o atentar contra su integridad o su disponibilidad. También se consideran código móvil los “parches” y actualizaciones o los *plug-in* que son invocados por las aplicaciones o por los navegadores y se descargan vía Web. Y, por supuesto, lo es tam-

bién el software como tal, escrito en un lenguaje de programación de alto nivel como C o Java y compilado a código ejecutable para plataformas específicas generalmente antes de descargarse por la red, a menudo sin consentimiento o incluso sin el conocimiento del usuario del sistema local. Si se cuenta con un cortafuegos configurado adecuadamente, éste puede advertir al usuario de la descarga de programas sospechosos de contener errores o ser de naturaleza maliciosa pero sigue siendo éste en definitiva quien, sin conocer con certeza la fiabilidad del código, debe decidir si permite o no la descarga. Las técnicas de certificación automática tienen varios escenarios de aplicación: proveedores de potencia computacional distribuida, sistemas operativos extensibles, actualizaciones de aplicaciones software, seguridad del código móvil, sistemas con restricciones críticas de seguridad, dispositivos *handheld*, etc. El proceso de obtención de los certificados puede ser complejo y costoso pero su comprobación por parte del consumidor del código.

Una de las implementaciones más populares del esquema general de certificación automática se conoce como *compilación certificante*. Dicha técnica enriquece un compilador con maquinaria que puede provenir del campo de la demostración automática de teoremas, de la teoría de tipos o de la programación lógica para generar *código con demostración asociada*; esto es, además del código compilado, como salida de la compilación se genera también la correspondiente prueba. De esta forma, es el productor quien realiza el trabajo pesado de generar una prueba de que el código respeta la política, mientras el consumidor simplemente realiza la validación de dicha prueba. En el esquema de la trilogía, la compilación certificante se representaría también con un hiper-arco, constituido por un arco principal de programas a programas y otro arco secundario para generar y manipular la prueba (propiedades).

En resumen, los métodos y técnicas de la lógica pueden aplicarse prácticamente en todas las etapas del desarrollo de proyectos. Es bien conocida la dificultad de realizar una especificación completa inicial del problema a resolver sin obviar detalles relevantes para el cliente o para el personal implicado en el desarrollo. Las limitaciones humanas, tanto de previsión, expresión y comunicación, como de comprensión, se alían para hacer del desarrollo de proyectos un proceso iterativo donde cada vuelta atrás resulta costosa, repercutiendo cada nuevo cambio introducido en un encarecimiento económico y temporal muy acentuado. Este problema está revelando su importancia también en los aspectos de gestión hasta el punto que el mismo tipo de recursos lógicos que pueden usarse durante el proceso de desarrollo se han propuesto también para las actividades de gestión (por ejemplo, para la especificación formal de los contratos software [119] y también para la ayuda integrada a la toma de decisiones). Una perspectiva completa de los métodos lógicos comentados a lo largo de esta sección se muestra en la Figura 10.5.

10.2.4. Herramientas avanzadas para el desarrollo del software: métodos ligeros

Como hemos expuesto, la variedad de recursos expresivos ofrecida por la lógica la hace muy interesante para su empleo en procesos total o parcialmente automatizados de

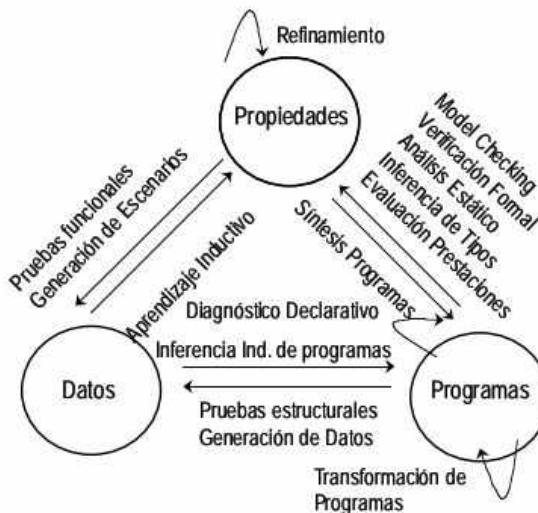


Figura 10.5 Perspectiva completa de la Ingeniería del Software Automática

creación y mantenimiento de sistemas computacionales, tanto si son utilizados, como si no, como lenguajes de implementación del producto final. Por ejemplo, es un hecho poco conocido que parte de la biblioteca estándar de C++ consiste en código escrito en un estilo funcional. Es posible plantear una integración completa de un lenguaje funcional puro simple en C++ utilizando las capacidades de extensibilidad del lenguaje y sin modificar el entorno estándar de compilación y ejecución [96]. Partiendo de la perspectiva general sobre la Ingeniería del Software Automática desarrollada en los apartados anteriores, en lo que resta de capítulo detallamos las principales ideas de algunas de las técnicas y herramientas de desarrollo avanzadas – entendiendo como tales aquéllas que automatizan los métodos y técnicas de la lógica – en diferentes escenarios que pueden presentarse durante el desarrollo de un proyecto: transformación de programas y evaluación parcial, síntesis, depuración y diagnóstico automático.

10.3 TRANSFORMACIÓN DE PROGRAMAS Y EVALUACIÓN PARCIAL

La *transformación de programas* es un método para derivar programas correctos y eficientes partiendo de una especificación ingenua y más ineficiente del problema. Esto es, dado un programa P , se trata de generar un programa P' que resuelve el mismo pro-

blema y equivale semánticamente a P , pero que goza de mejor comportamiento respecto a cierto criterio de evaluación.

La transformación de programas también puede verse como una metodología para el desarrollo del software [115], de ahí su gran importancia. La idea básica consiste en dividir el proceso de desarrollo (comenzando con una especificación —posiblemente ingenua— escrita en un lenguaje de programación) en una secuencia de pequeños pasos de transformación.

En la literatura se han propuesto una gran variedad de técnicas de transformación para mejorar el código de los programas. Dos de las más conocidas son: las transformaciones de *plegado/desplegado* [23, 115] y la *evaluación parcial* de programas [70, 72].

Aunque la evaluación parcial ha desarrollado sus propias técnicas, también puede considerarse como un caso particular de las transformaciones de plegado/desplegado. Este será, por simplicidad expositiva, el punto de vista que adoptaremos para su introducción en este apartado.

10.3.1. Transformaciones de plegado/desplegado

Las transformaciones de *plegado/desplegado* fueron introducidas por Brustall y Darlington en [23], para programas funcionales constituidos por conjuntos de ecuaciones (recursivas). Dichas ecuaciones pueden entenderse como fórmulas de una *lógica ecuacional*, aquélla en la que no se contempla ninguna conectiva lógica y donde el único símbolo de predicado considerado es la igualdad. Esta aproximación a la transformación de programas se basa en un conjunto de reglas de transformación, que se aplican a los programas en un orden determinado por una *estrategia*:

1. *Definición*: permite la introducción de funciones nuevas o la extensión de las existentes.
2. *Instanciación*: permite concretar (o especializar) una función asignando datos de entrada conocidos a sus argumentos.
3. *Desplegado (unfolding)*: permite el reemplazamiento de una llamada a función por su respectiva definición.
4. *Plegado (folding)*: como su nombre indica, es la transformación inversa del desplegado, es decir, el reemplazamiento de cierto fragmento del código por la correspondiente llamada a función.
5. *Abstracción*: se introduce una cláusula “*where*” (i.e., una expresión cualificada) a partir de una ecuación que define una función.
6. *Leyes*: se transforma una ecuación, que define una función, utilizando en su parte derecha cualquier ley referente a las operaciones primitivas que contenga —por ejemplo, la asociatividad de la multiplicación: $x \times (y \times z) = (x \times y) \times z$ —.

La Figura 10.6 concreta estas técnicas de transformación para el caso en el que las funciones están definidas mediante ecuaciones que se interpretan como reglas de reescritura¹. Las reglas de inferencia de la Figura 10.6 deben utilizarse de modo semejante al empleado en la lógica. Así pues, al aplicar la regla de definición tendremos la posibilidad de introducir una regla nueva $l_2 \rightarrow r_2$ en un programa P , si no existe ninguna regla $l_1 \rightarrow r_1 \in P$ tal que l_1 solape con l_2 . La aplicación de cada una de estas reglas da lugar a una secuencia de transformación de la que, finalmente, se extraerá el programa transformado. Ilustramos el uso de estas reglas y los beneficios que pueden reportar mediante el siguiente ejemplo.

Ejemplo 10.1

Suponga que queremos confeccionar un programa que compute la media de una lista de números. Podríamos partir de un programa P claro, desde el punto de vista declarativo, y obviamente correcto:

```
media(L) → suma(L)/longitud(L)
suma([]) → 0
suma([X|R]) → X + suma(R)
longitud([]) → 0
longitud([X|R]) → 1 + longitud(R)
```

Este programa es ineficiente porque recorre la lista L dos veces antes de calcular la media. Se puede derivar una nueva versión del programa, que realice las dos operaciones juntas mientras se recorre la lista una sola vez, aplicando un proceso de transformación de plegado/desplegado:

```
% Definición
med(L) → (suma(L), longitud(L))
% Instanciación y despliegado
med([]) → (suma([]), longitud([]))
med([]) → (0, 0)
% Instanciación, despliegado, abstracción y plegado
med([X|R]) → (suma([X|R]), longitud([X|R]))
med([X|R]) → (X + suma(R), 1 + longitud(R))
med([X|R]) → (X + u, 1 + v) where (u, v) = (suma(R), longitud(R))
med([X|R]) → (X + u, 1 + v) where (u, v) = med(R)
% Abstracción y plegado
media(L) → u/v where (u, v) = (suma(L), longitud(L))
media(L) → u/v where (u, v) = med(L)
```

que conduce al programa transformado P' :

¹Las reglas de reescritura pueden verse como ecuaciones orientadas en las que la parte izquierda l es la función definida por la parte derecha r . En este contexto, un programa es un conjunto de reglas de reescritura.

$$\text{media}(L) \rightarrow u/v \text{ where } (u, v) = \text{med}(L)$$

$$\text{med}([]) \rightarrow (0, 0)$$

$$\text{med}([X|R]) \rightarrow (X + u, 1 + v) \text{ where } (u, v) = \text{med}(R)$$

que computa la media de la lista de números L recorriendo dicha lista una sola vez, acumulando la suma y la longitud de la lista conforme avanza la evaluación. El éxito alcanzado en el ejemplo anterior se debe al orden preciso en el que se han aplicado las reglas de transformación. Por este motivo es imprescindible que un proceso de transformación sea guiado por una *estrategia* de transformación adecuada. En el ejemplo anterior se ha utilizado una estrategia denominada *tupling*. El paso crucial dentro de la anterior secuencia de transformación (que, como se ha argumentado, mejora la eficiencia del programa original), es el paso de definición empleado. Debido a que este paso suele requerir bastante ingenio, en general, se ha denominado “paso de *eureka*” en la literatura de transformación de programas. Los paso de *eureka* requieren una cierta intervención del usuario del sistema de transformación, por lo que éstos suelen ser semiautomáticos, en contraste con la técnica de evaluación parcial, que sí puede automatizarse de manera efectiva.

1. DEFINICIÓN:

$$\frac{(\forall R_i)(R_i \equiv (l_1 \rightarrow r_1) \in P \wedge \neg \text{unifica}(l_1, l))}{(l \rightarrow r) \in P}$$

2. INSTANCIACIÓN:

$$\frac{(l \rightarrow r) \in P}{(\theta(l) \rightarrow \theta(r)) \in P}$$

3. DESPLEGADO (*unfolding*):

$$\frac{(l_1 \rightarrow r_1) \in P \wedge (l_2 \rightarrow C[\theta(l_1)]) \in P}{(l_2 \rightarrow C[\theta(r_1)]) \in P}$$

4. PLEGADO (*folding*):

$$\frac{(l_1 \rightarrow r_1) \in P \wedge (l_2 \rightarrow C[\theta(r_1)]) \in P}{(l_2 \rightarrow C[\theta(l_1)]) \in P}$$

5. ABSTRACCIÓN:

$$\frac{(l \rightarrow C[t_1, \dots, t_n]) \in P}{(l \rightarrow C[x_1, \dots, x_n] \text{ where } \langle x_1, \dots, x_n \rangle = \langle t_1, \dots, t_n \rangle) \in P}$$

donde: θ es una sustitución; $C[\square]$ es un contexto y $C[t]$ es el resultado de substituir en el contexto $C[\square]$ cada una de las apariciones del símbolo “ \square ” por el término t .

Figura 10.6 Reglas de transformación de Burstall y Darlington.

10.3.2. Evaluación parcial

La *evaluación parcial* (EP) de programas (también conocida como *especialización*) [72], es una técnica de transformación de programas que consiste en la especialización de programas con respecto a ciertos datos de entrada, conocidos en tiempo de compilación. Por otra parte, como estudiaremos más adelante, ofrece un marco unificado para el campo de los procesadores de lenguajes, en particular, compiladores e intérpretes [58]. En general, las técnicas de EP incluyen algún criterio de parada para garantizar la terminación del proceso de la transformación [84, 92]. La EP es, por tanto, una técnica de transformación automática, lo cual la distingue de otras técnicas de transformación de programas tradicionales [23, 115]. La EP ha sido aplicada a los lenguajes imperativos tanto como a los declarativos y a una gran variedad de problemas concretos. Una panorámica general sobre este campo y su área de aplicación se presenta en [72] y en [73]. Un breve pero excelente tutorial sobre EP (si bien centrado en la especialización de programas imperativos y funcionales) es [37]. Otros trabajos que examinan aspectos concretos del área son [5, 59].

La idea de especializar funciones con respecto a uno o varios de sus argumentos es vieja en el campo de la teoría de funciones recursivas, donde recibe el nombre de *proyección*. Sin embargo la evaluación parcial trabaja con programas (textos), más bien que con funciones matemáticas, y va más allá de la simple proyección de funciones matemáticas.

La EP establece cómo ejecutar un programa cuando solo conocemos parte de sus datos de entrada. De forma más precisa, dado un programa P y parte de sus datos de entrada in_1 , el objetivo de la EP es construir un nuevo programa P_{in_1} que, cuando recibe el resto de los datos de entrada in_2 , computa el mismo resultado que produce P al procesar toda su entrada ($in_1 + in_2$). Esto último asegura la corrección de la transformación efectuada. El programa que realiza el proceso de EP recibe el nombre de *evaluador parcial* y el resultado de la EP, el programa P_{in_1} , se denomina *programa especializado*, *programa evaluado parcialmente* o también *programa residual*. La idea que se esconde detrás del proceso de EP consiste en: i) realizar tantos cálculos como sea posible en tiempo de EP, haciendo uso de los datos de entrada conocidos in_1 , también denominados *datos estáticos* (por contraposición con los datos de entrada desconocidos in_2 , que son denominados *datos dinámicos*, y que solo se conocen en tiempo de ejecución del programa residual); ii) generar código relacionado con aquellos cálculos que no puedan realizarse por depender de los datos de entrada desconocidos. Así pues, un evaluador parcial realiza una mezcla de acciones de cómputo y de generación de código; ésta es la razón por la que Ershov denominó a la EP *computación mixta (mixed computation)*. Cuando se realizan cálculos en tiempo de EP, haciendo uso de los datos de entrada conocidos, decimos que hay *propagación de la información*. El objetivo de la EP es obtener la mejor de las especializaciones posibles maximizando la propagación de la información. Se espera que el programa resultante pueda ejecutarse de forma más eficiente ya que, usando el conjunto de datos (parcialmente) conocidos, es posible evitar algunas computaciones (en tiempo

de ejecución) que se realizarán (una única vez) durante el proceso de EP. Para cumplir estos fines, la EP utiliza, además de la computación simbólica, algunas técnicas bien conocidas provenientes de la transformación de programas [23] (véase el Apartado 10.3.1), procurando su automatización.

De entre las reglas que se muestran en la Figura 10.6 solamente las cuatro primeras presentan un interés inmediato para las técnicas de EP. Por otro lado, existe un alto grado de indeterminismo en la aplicación de estas reglas. Una estrategia, que reduce dicho grado de indeterminismo, es la presentada en el siguiente esquema básico de transformación, que puede considerarse como el embrión de un *evaluador parcial* basado en transformaciones de plegado/desplegado²:

1. Repetir hasta que convenga

- buscar una **definición**, e
- **instanciar** la definición para permitir,
- pasos de **desplegado** en diversos puntos, seguidos por,
- un **plegado** de las reglas resultantes haciendo uso de alguna de las reglas obtenidas.

2. Extraer el programa transformado.

Observe que, en este contexto los pasos de definición están guiados por datos estáticos o un conjunto de llamadas a especializar y, por lo tanto, pueden automatizarse de forma completa.

De entre todas estas técnicas, el desplegado es la herramienta de transformación fundamental de la EP. Para programas funcionales, los pasos de plegado y desplegado solo involucran ajuste de patrones (*pattern matching*). En el caso de los programas lógicos, el ajuste de patrones se sustituye por la unificación, obteniéndose así una mayor potencia de propagación de la información. Una técnica específica empleada en la EP es la denominada *especialización de puntos de control* del programa, que combina las reglas de definición, desplegado y plegado. Esta técnica puede entenderse como un proceso consistente en una definición al que le sigue una sucesión de pasos de desplegado (tantos como sea posible) que se detienen en cuanto se reconoce una configuración “ya vista” anteriormente, momento en el que se genera código para llamar a esa configuración “ya vista” (i.e., se realiza un paso de plegado). En un lenguaje imperativo, un *punto de control* es una etiqueta del programa; en un lenguaje declarativo puede considerarse que es la definición de una función o predicado. La idea es que una etiqueta o una función del programa P pueda aparecer en el programa especializado P_{in1} en varias versiones especializadas, cada una correspondiente a datos determinados conocidos en

²En [72, pag.356] se presenta un algoritmo detallado para este tipo de evaluadores parciales. Sin embargo, observe que son posibles otras aproximaciones a la EP. Por ejemplo, consulte [59] para obtener información sobre un algoritmo de EP basado en una orientación más estándar.

tiempo de EP. Es conveniente notar que esta técnica es un reflejo del esquema básico de transformación anteriormente esbozado. Otra de las técnicas empleadas por la EP es la *abstracción*³, consistente en generalizar una expresión cuando no es posible su especialización; en cierto sentido, puede verse como la transformación inversa de la instanciación y puede caracterizarse en términos de un proceso de definición al que le sigue uno de plegado [4].

A continuación ilustramos el proceso de EP y aclaramos algunos de los conceptos y técnicas introducidos, mediante una serie de ejemplos.

Ejemplo 10.2

Sea el fragmento de un programa P que computa la función x^n :

$$\begin{aligned} \textit{pow}(0, X) &\rightarrow 1 \\ \textit{pow}(N, X) &\rightarrow X * \textit{pow}(N - 1, X) \end{aligned}$$

Si queremos especializar el programa para el dato de entrada conocido $N = 3$, los pasos que podría realizar un hipotético evaluador parcial serían:

```
% Definición
pow3(X) → pow(3, X)
% Instanciación, desplegado y computación simbólica
pow(3, X) → X * pow(3 - 1, X)
pow(3, X) → X * pow(2, X)
pow(3, X) → X * (X * pow(2 - 1, X))
pow(3, X) → X * (X * pow(1, X))
pow(3, X) → X * (X * (X * pow(1 - 1, X)))
pow(3, X) → X * (X * (X * pow(0, X)))
pow(3, X) → X * (X * (X * 1))
pow(3, X) → X * (X * X)
% Desplegado final
pow3(X) → X * (X * X)
```

conduciendo al programa especializado P_3 :

$$\textit{pow3}(X) \rightarrow X * (X * X)$$

que computa la función x^3 . De manera simple, podemos entender que $\textit{pow3}(X)$ constituye una especialización del punto de control $\textit{pow}(N, X)$ en la que no ha sido necesario realizar pasos de plegado, debido a que el programa especializado no contiene llamadas a función. Tampoco se han requerido pasos de abstracción para lograr la especialización.

Ejemplo 10.3

Consideremos de nuevo el programa del Ejemplo 10.2. Si queremos especializar el programa con respecto a la expresión $(\textit{pow}(M, B) * \textit{pow}(N, B))$, los pasos que podría realizar

³La técnica de abstracción utilizada en el contexto de la EP es más compleja que la descrita en apartados precedentes y su caracterización precisa está fuera del alcance pretendido en esta introducción.

un hipotético evaluador parcial serían:

```
% Definición  
ppow(M,N,B) → pow(M,B) * pow(N,B)  
% Instanciación, desplegado y computación simbólica  
ppow(0,N,B) → pow(0,B) * pow(N,B)  
ppow(0,N,B) → 1 * pow(N,B)  
ppow(0,N,B) → pow(N,B)  
% Instanciación, desplegado y computación simbólica  
ppow(M,0,B) → pow(M,B) * pow(0,B)  
ppow(M,0,B) → pow(M,B) * 1  
ppow(M,0,B) → pow(M,B)  
% Desplegado y computación simbólica  
ppow(M,N,B) → B * pow(M-1,B) * pow(N,B)  
ppow(M,N,B) → B * pow(M-1,B) * B * pow(N-1,B)  
ppow(M,N,B) → B * B * pow(M-1,B) * pow(N-1,B)  
% Plegado  
ppow(M,N,B) → B * B * ppow(M-1,N-1,B)
```

Pudiendo extraerse el programa especializado:

```
ppow(0,N,B) → pow(N,B)  
ppow(M,0,B) → pow(M,B)  
ppow(M,N,B) → B * B * ppow(M-1,N-1,B)  
pow(0,X) → 1  
pow(N,X) → X * pow(N-1,X)
```

que permite el cómputo de la expresión ($\text{pow}(M,B) * \text{pow}(N,B)$) de manera más eficiente que el programa original. La definición de la función $\text{ppow}(M,N,B)$ constituye una especialización del punto de control $\text{pow}(N,X)$ que facilita el cómputo de la expresión ($\text{pow}(M,B) * \text{pow}(N,B)$). En este ejemplo se ha podido realizar un paso de plegado, debido a que en el proceso de especialización hemos reconocido la aparición de una regularidad: la expresión “ya vista” $\text{pow}(M-1,B) * \text{pow}(N-1,B)$, que puede sustituirse por su definición $\text{ppow}(M-1,N-1,B)$, dando lugar a un paso de desplegado. El programa especializado, si bien no obtiene una mejora en cuanto al número de multiplicaciones a realizar, consigue fundir las secuencias de llamadas recursivas iniciadas por ($\text{pow}(M,B)$ y $\text{pow}(N,B)$), que deberían realizarse por separado, en una única secuencia en la que se simplifica el control. Si se hubiese partido de un programa iterativo, el efecto hubiese sido la combinación de dos bucles en uno solo.

Aunque al comentar los Ejemplos 10.2 y 10.3 no se ha hecho énfasis en el control del proceso de EP, estos ejemplos ilustran una forma de EP, denominada *online*⁴, en la que

⁴En contraposición a la EP *online*, existe otro tipo de aproximación a la EP, que se denomina *offline*, que se caracteriza por ser un proceso consistente en varias fases: *binding-time analysis*, generación de anotaciones y EP propiamente dicha. Los evaluadores parciales *online* permiten alcanzar una mayor precisión en la especialización de los programas que los evaluadores parciales *offline*; sin embargo, estos últimos reducen el coste del proceso de EP. Por limitaciones de espacio dejaremos de lado la aproximación *offline* de la EP (véase [37] para una breve introducción).

todas las decisiones de control (e.g. ¿qué evaluar?, ¿cuándo parar?) se toman en tiempo de EP.

10.3.3. Corrección de la Evaluación Parcial

En cuanto a la fundamentación teórica de la EP, el tema más relevante es el de la *corrección* de la misma. A este respecto hay dos puntos a tratar: la corrección semántica (i.e., la preservación del comportamiento observable) y el control de la terminación.

Comenzaremos discutiendo la corrección semántica de la evaluación parcial. Con el fin de formalizar este concepto en un marco general, se introducen las siguientes convenciones y notaciones estándares [72], que serán útiles en este subapartado y en el próximo. Vamos a tratar con diversos lenguajes; emplearemos las letras L , S y T para referirnos, respectivamente, a un lenguaje de implementación, a un lenguaje fuente y a un lenguaje objeto.

Denotaremos por D el conjunto de los datos que pueden pasarse como valores a un programa, incluyendo también los textos que forman los programas. Supondremos que suministramos listas de datos como entrada para los programas; denotaremos por D^* el conjunto de todas las listas que pueden formarse a partir de los elementos de D . Suponemos que la semántica de los lenguajes que empleamos está definida en un estilo operacional no especificado. Si el lenguaje es imperativo, suponemos que los cómputos se realizan mediante una secuencia de instrucciones que producen cambios de estado.

Más genéricamente, podemos asumir que los cómputos se realizan mediante deducción aplicando ciertas reglas de inferencia. Si P es un programa en un lenguaje L , entonces $\llbracket P \rrbracket_L$ denota el significado del programa P escrito en el lenguaje L . El significado del programa se establece como una función $\llbracket P \rrbracket_L : D^* \rightarrow D \cup \{\perp\}$ tal que si $in \in D^*$ entonces

$$out = \llbracket P \rrbracket_L(in)$$

siendo $out \in D$ el resultado de ejecutar P para la entrada in ; cuando la ejecución del programa P no termina el valor de out es \perp (indefinido).

Supongamos un programa fuente P al que se suministra la entrada de datos estática e , conocida previamente, y la entrada de datos dinámica d , conocida con posterioridad. Entonces definimos la EP de P , utilizando un evaluador parcial $Spec$ mediante la ecuación:

$$P_e = \llbracket Spec \rrbracket_L([P, e]).$$

Definición 10.1

Supuesto que el proceso de EP termina, decimos que un evaluador parcial es:

1. **Correcto** si y solo si para toda entrada dinámica $d \in D$, $\llbracket P_e \rrbracket_T(d) = \text{out}$ implica que $\llbracket P \rrbracket_S([e, d]) = \text{out}$ (i.e., si out es un valor computado por P_e entonces P lo computa también.)
2. **Completo** si y solo si para toda entrada dinámica $d \in D$, $\llbracket P \rrbracket_S([e, d]) = \text{out}$ implica que $\llbracket P_e \rrbracket_T(d) = \text{out}$ (i.e., si out es un valor computado por P entonces P_e lo computa también.)

Los conceptos de corrección y completitud se han definido en un sentido *fuerte* de forma que, si un evaluador parcial es correcto y completo, podemos establecer la identidad semántica entre el programa especializado y el original. Esto es, se cumple que

$$\llbracket P_e \rrbracket_T(d) = \llbracket P \rrbracket_S([e, d]),$$

para cada entrada dinámica $d \in D$. En palabras, P y P_e computan los mismos valores (salidas).

El control de la terminación es un problema crucial en el ámbito de la EP. La no terminación es un comportamiento indeseable para una herramienta que pretende optimizar programas automáticamente. La no terminación del proceso de EP puede producirse por una de las siguientes razones:

1. Un intento de construir una instrucción, una expresión, o en general una estructura infinita.
2. Un intento de construir un programa residual que contenga infinitos puntos de control (etiquetas, procedimientos, funciones definidas o predicados).

La causa última de este comportamiento es la misma: un fallo en la estrategia de desplegado que emplea el evaluador parcial [71]. En cualquier caso, la no terminación hace que un evaluador parcial sea poco aceptable para su uso por parte de un usuario inexperto, y completamente inaceptable para su uso como herramienta de generación automática de software. Para asegurar la terminación de la evaluación parcial, durante el proceso de especialización la mayoría de los evaluadores parciales mantienen una historia de la computación, que permite su consulta a la hora de tomar decisiones sobre si desplegar una expresión o no; y, en caso de que la decisión sea “no”, determinar cómo realizar el plegado para especializar la expresión o si hay que abstraer, para convertirla en otra más general. Existen varios compromisos que deben tenerse en consideración cuando realizamos la EP: desplegar con suma liberalidad puede llevar al problema (1), dando lugar a un tiempo de especialización infinito y a la no generación del programa residual; generar expresiones residuales demasiado especializadas (i.e., conteniendo demasiados datos estáticos) puede conducir a un programa residual infinito y, por lo tanto, a la aparición del problema (2); en el otro extremo, generar expresiones residuales demasiado generales

puede hacer perder toda la especialización, obteniéndose programas residuales que son (esencialmente) el programa original sin especializar. Los aspectos que acabamos de comentar han permitido distinguir dos niveles de control en el problema de la terminación de la EP [93] que, en buena medida, pueden ser abordados de forma independiente: el llamado *control local*, asociado con el punto (1); y el llamado *control global*, asociado con el punto (2).

Un aspecto relacionado con el problema de la terminación es el de la *precisión*, entendiendo por tal la obtención del máximo potencial especializador. Aquí, también podemos distinguir dos niveles: i) el nivel de *precisión local*, asociado con el desplegado de una expresión y el hecho de que puede perderse potencial para la especialización si se detiene el desplegado de la expresión demasiado pronto (o demasiado tarde); y ii) un nivel de *precisión global* que está asociado al número de puntos de control especializados; en general disponer de un programa residual con el mayor número de puntos de control especializados con respecto a una variedad de datos estáticos conducirá a una mejor especialización.

Como puede apreciarse, el control de la EP ofrece aspectos que pueden entrar en conflicto, como son el de la terminación y el de la precisión. Un buen algoritmo de EP debe asegurar la corrección y la terminación mientras minimiza las pérdidas de precisión.

Por último, se dice que un especializador es *parcialmente* correcto si, en el supuesto de que termina, genera un programa residual semánticamente equivalente al original. Si además de generar un programa residual semánticamente equivalente al original, termina cualquiera que sea la circunstancia, se habla de *corrección total*.

10.3.4. Generación automática de programas

En este apartado formalizamos el concepto de evaluador parcial, lo que nos permite establecer relaciones interesantes entre la evaluación parcial y la generación automática de programas.

Para establecer la propiedad esencial que caracteriza a un evaluador parcial de una manera formal, supongamos un programa fuente P al que se suministra la entrada de datos estática in_1 y la entrada de datos dinámica in_2 . Entonces podemos describir el cómputo (del resultado) en un paso mediante la ecuación,

$$out = \llbracket P \rrbracket_S([in_1, in_2])$$

y el cómputo (del resultado) en dos pasos, utilizando un evaluador parcial $Spec$ mediante las ecuaciones

$$\begin{aligned} P_{in_1} &= \llbracket Spec \rrbracket_L([P, in_1]) \\ out &= \llbracket P_{in_1} \rrbracket_T(in_2). \end{aligned}$$

Combinando estas ecuaciones se obtiene una *definición ecuacional* del evaluador parcial $Spec$:

$$\llbracket P \rrbracket_S([in_1, in_2]) = \llbracket \llbracket Spec \rrbracket_L([P, in_1]) \rrbracket_T(in_2)$$

donde, si una parte de la ecuación está definida, también lo estará la otra y con el mismo valor. De estas ecuaciones también se desprende que el evaluador parcial puede estar escrito en un lenguaje de implementación L , tener como entrada un programa P escrito en un lenguaje fuente S , y producir como salida un programa especializado escrito en un lenguaje objeto⁵ T . Cuando solamente se emplea un lenguaje en la discusión, los subíndices pueden eliminarse obteniéndose la siguiente ecuación simplificada:

$$\llbracket P \rrbracket([in_1, in_2]) = \llbracket \llbracket Spec \rrbracket([P, in_1]) \rrbracket(in_2)$$

Si el evaluador parcial, escrito en un lenguaje L , admite como entrada programas también escritos en L , se dice que es *autoaplicable*. La capacidad de autoaplicación posibilita la escritura de evaluadores parciales que puedan especializarse a sí mismos; éste es un tema de gran interés dentro del campo de la EP.

La definición ecuacional del evaluador parcial nos permite poner en relación los conceptos de interpretación, compilación y EP. Primeramente, nótese que un intérprete es un programa Int , escrito en un lenguaje L , que puede ejecutar un programa $Fuente$ en un lenguaje S junto con sus datos de entrada in . En símbolos,

$$\llbracket Fuente \rrbracket_S(in) = \llbracket Int \rrbracket_L([Fuente, in]),$$

que es la ecuación de definición de un *intérprete* Int para S escrito en L . Un compilador es un programa $Comp$, escrito en un lenguaje L , que genera un programa $Objeto$ en un lenguaje T . En símbolos,

$$Objeto = \llbracket Comp \rrbracket_L(Fuente),$$

que es la ecuación que define un programa $Objeto$. El efecto de ejecutar el programa $Fuente$ sobre los datos de entrada in se consigue, una vez realizada la compilación, ejecutando el programa $Objeto$ con los datos de entrada in ,

$$\llbracket Fuente \rrbracket_S(in) = \llbracket Objeto \rrbracket_T(in).$$

Combinando estas ecuaciones obtenemos la definición de un *compilador* $Comp$ de S a T escrito en L ,

$$\llbracket Fuente \rrbracket_S(in) = \llbracket \llbracket Comp \rrbracket_L(Fuente) \rrbracket_T(in)$$

Ahora pueden resumirse las capacidades de la EP para la *generación automática* de programas mediante la siguiente proposición.

Proposición 10.1 [Proyecciones de Futamura] Sean $Spec$ un especializador de L a T escrito en L , e Int un intérprete para S escrito en L .

⁵Posiblemente un lenguaje máquina, si el interés primordial fuese el incremento de la eficiencia del programa especializado con respecto al original.

- 1^aproy. : $Objeto = \llbracket Spec \rrbracket_L([Int, Fuente])$
- 2^aproy. : $Comp = \llbracket Spec \rrbracket_L([Spec, Int])$
- 3^aproy. : $Cogen = \llbracket Spec \rrbracket_L([Spec, Spec])$

Estas ecuaciones, aunque sencillas de probar (*véase* [72]), no son fáciles de entender intuitivamente. La primera de ellas indica que se puede compilar un programa *Fuente* especializando su intérprete *Int* con respecto a dicho programa *Fuente*. La segunda dice que se puede obtener un compilador mediante autoaplicación del evaluador parcial, i.e., especializando el propio *Spec* con respecto a un intérprete *Int*. La tercera establece que *Cogen* es un generador de compiladores, que transforma un intérprete en un compilador, i.e., $Comp = \llbracket Cogen \rrbracket_T(Int)$. Así pues, la EP permite la compilación (primera proyección), la generación de compiladores (segunda proyección) y la generación de generadores de compiladores (tercera proyección).

10.3.5. Ventajas de la Evaluación Parcial

En este apartado se discuten varias de las principales ventajas que puede aportar la EP.

Aumento en la Eficiencia de los Programas Una de las principales motivaciones de la EP es el aumento en la *eficiencia (speedup)* de los programas. Debido a que parte de los cómputos se han realizado previamente, en tiempo de EP, esperamos que el programa especializado sea más rápido que el programa original. Es común realizar una medida del aumento de la eficiencia, obteniendo la razón entre el tiempo de ejecución del programa original y del especializado [72, 71]. A la hora de medir la eficiencia de la EP, debe considerarse también el coste del tiempo de especialización del programa original. La EP es claramente ventajosa cuando un (procedimiento de un) programa debe ejecutarse reiteradamente para una porción de su entrada, ya que entonces el coste que pueda suponer la especialización del programa será ampliamente amortizado por las sucesivas ejecuciones del programa especializado, que será más “rápido” que el original. Neil D. Jones argumenta en [72] que la EP puede ser ventajosa incluso para una única ejecución, ya que muchas veces sucede que el tiempo invertido en la especialización del programa original sumado al tiempo de ejecución del programa especializado es inferior al tiempo que resultaría si se ejecutase el programa original con toda su entrada.

Productividad, Reusabilidad y Modularidad Otro objetivo de la EP es propiciar la *productividad* en el desarrollo de programas mediante el aumento de la *reusabilidad* y la *modularidad* del código. De todos es sabido que es más fácil establecer el significado declarativo (correcto) para una especificación sencilla de un problema; por contra, la ejecución de esta especificación como programa puede resultar ineficiente. Un evaluador parcial puede facilitar y hacer más ágil el desarrollo de los programas al permitir

explotar una biblioteca de plantillas genéricas y simples (cuyo significado declarativo fuese, sin duda, el esperado) que posteriormente se especializan de forma automática para producir un código más eficiente. La corrección del proceso de EP asegura la corrección semántica del programa especializado. Disponer de una biblioteca de plantillas genéricas, para las tareas más comunes, también mejoraría la reusabilidad del código.

De hecho, muchas veces cuando programamos nos encontramos con un conjunto de tareas similares para resolver y que corresponden a diferentes aspectos de un mismo problema. Una forma de afrontar esta situación es escribir un procedimiento específico y eficiente para cada una de estas tareas. Podemos enumerar dos desventajas en esta forma de proceder:

1. Debe de realizarse un sobreexceso de programación, lo que aumenta el coste de creación del programa y de su verificación.
2. El mantenimiento del programa se hace más difícil, ya que un cambio en las especificaciones puede requerir el cambio de cada uno de los procedimientos.

Con ser grave la primera de las deficiencias apuntadas, la segunda es la que puede producir mayores costes a largo plazo. Muchos estudios indican que el mayor coste en el *ciclo de vida* de un programa no es el coste inicial de diseño, codificación y verificación, sino el coste posterior asociado al mantenimiento del programa mientras está en producción y uso. Una solución alternativa, que elimina las deficiencias comentadas anteriormente, consiste en escribir un procedimiento altamente parametrizado capaz de solucionar cada uno de los aspectos del problema. Nuevamente, podemos apuntar dos deficiencias:

1. La dificultad propia de programar un procedimiento genérico que cubra todas las alternativas de forma eficiente.
2. La ineficiencia inherente a este tipo de procedimientos, ya que un procedimiento altamente parametrizado gastará mucho de su tiempo de ejecución en la comprobación e interpretación de los parámetros y (relativamente) poco tiempo en los cálculos que debe realizar.

La EP puede ayudarnos a vencer esta disyuntiva. Podemos escribir un procedimiento genérico altamente parametrizado (posiblemente ineficiente) y utilizar un evaluador parcial para especializarlo, suministrando los valores de los parámetros adecuados para cada una de las tareas específicas a resolver. Esto permite obtener automáticamente un conjunto de procedimientos específicos y eficientes, tal y como deseábamos.

Autoaplicación y Generación Automática de Programas

Uno de los objetivos más perseguidos en el campo de la evaluación parcial es lograr evaluadores parciales autoaplicables. La autoaplicación permite llevar a la práctica los resultados teóricos formulados por Futamura [58] y Ershov [47] (agrupados en la Proposición 10.1), que propician la generación automática de programas.

Dentro de la generación automática de programas una de las aplicaciones más notables es la *generación de compiladores dirigida por la semántica* [141]; por ello entendemos lo siguiente: dada una especificación de un lenguaje de programación, basada en una semántica formal⁶, transformarla automáticamente en un compilador. La motivación para la generación automática de compiladores es clara: el ahorro en esfuerzos de programación que supone la construcción de un compilador, que en ocasiones no es correcto con respecto a la semántica propuesta para el lenguaje que compila. La corrección de la EP permite que la transformación automática de una especificación semántica de un lenguaje en un compilador haga desaparecer estos errores. Las tareas de diseñar la especificación de un lenguaje, escribir el compilador y mostrar la corrección del compilador, se reducen a una sola tarea: escribir la especificación del lenguaje en una forma adecuada para ser la entrada de un generador de compiladores.

Como puede apreciarse, la EP y la autoaplicación tiene unas posibilidades muy prometedoras. Aunque todavía se necesita algún esfuerzo de investigación para entender perfectamente su teoría y sus técnicas prácticas, la EP ya ha tenido sus primeras aplicaciones industriales en diversos campos tecnológicos. También se ha aplicado a la resolución de problemas dentro del ámbito de la inteligencia artificial.

10.3.6. Evaluación parcial de programas lógicos

Komorowski introdujo la evaluación parcial en el campo de la Programación Lógica [78]. Después de varios años de olvido, la evaluación parcial ha atraído un interés considerable en este campo [13, 25, 60, 59, 62, 87, 88, 93], donde se le conoce generalmente como *deducción parcial (Partial Deduction)*

En [14] se presenta un algoritmo para la evaluación parcial de programas lógicos, mientras que en [88] se presentan los fundamentos de la deducción parcial en un marco formal. Dentro del marco teórico establecido por Lloyd y Shepherdson en [88] para la deducción parcial, la evaluación parcial de programas lógicos puede describirse como sigue. Dado un programa P y un conjunto de átomos S , el objetivo de la deducción parcial es obtener un nuevo programa P' que compute las mismas respuestas que P para cualquier *objetivo (input goal)* que sea una instancia de un átomo perteneciente a un conjunto S de átomos evaluados parcialmente. El programa P' se obtiene agrupando en un conjunto las *resultantes*, que se obtienen de la siguiente manera: para cada átomo A de S , primero construir un árbol-SLD finito (y posiblemente incompleto), $\tau(A)$, para $P \cup \{ \leftarrow A \}$; entonces considerar las hojas de las ramas de $\tau(A)$ que no son de fallo, digamos G_1, \dots, G_r , y las sustituciones computadas a lo largo de estas ramas, digamos $\theta_1, \dots, \theta_r$; y, finalmente, construir el conjunto de cláusulas: $\{\theta_1(A) \leftarrow G_1, \dots, \theta_r(A) \leftarrow G_r\}$; que son las resultantes que constituyen el programa residual P' . La restricción a

⁶Nótese que la semántica operacional de un lenguaje puede considerarse que es un intérprete (abstracto) del lenguaje

especializar únicamente átomos $A \in S$ (y no conjunciones de átomos, por ejemplo) está motivada por la necesidad de que las resultantes obtenidas sean cláusulas de Horn.

Para programas y objetivos definidos, Lloyd y Shepherdson han demostrado que la deducción parcial es siempre correcta, pero no necesariamente completa. La completitud se restablece exigiendo que $P' \cup \{G\}$ sea *S-cerrado*, i.e., cada átomo A' en $P' \cup \{G\}$ es una instancia de un átomo B en S (también se dice que el átomo A' está *cubierto* por el átomo B). El requisito anterior recibe el nombre de *condición de cierre*. Por otra parte, el resultado de corrección es *débil*, en el sentido de que si para un objetivo y el programa especializado se computa una respuesta, entonces el programa original computará una más general. Para garantizar que el programa residual P' no produce respuestas adicionales se necesita una condición adicional de *independencia*, que se cumple cuando dos átomos en S no tienen una instancia en común. De esta forma se consigue la corrección en una forma *fuerte* que supone la igualdad de las respuestas obtenidas por el programa original P' y el transformado P . La condición de independencia se logra mediante técnicas de renombramiento. Para programas y objetivos normales los resultados son menos satisfactorios. Muestran que la evaluación parcial es, en general, no solamente incompleta, sino, también incorrecta. Sólo cuando se añade la condición de independencia a la condición de cierre se restablecen las propiedades de corrección y completitud fuertes de la evaluación parcial.

Como se ha comentado, los problemas de la terminación del proceso de la evaluación parcial son de primordial importancia y han sido tratados en [22, 91, 92, 93]. Martens y Gallagher abordan el problema de la terminación estructurando el procedimiento en dos niveles: local y global. En el *nivel local*, una *regla de despliegado (unfolding rule)* controla la generación de los árboles de resolución de los que se extraen las resultantes, de forma que éstos se mantengan finitos, gracias a la aplicación de un test de parada apropiado (usando habitualmente un orden bien fundado). En el *nivel global* se controla la aplicación recursiva de la regla de despliegado mediante la selección de los átomos que se van a considerar en el paso siguiente. El nivel global debe garantizar una especialización suficiente (mediante la adición de nuevos átomos), a la vez que se asegura la condición de cierre y la terminación del proceso.

Gracias al mecanismo de unificación, la deducción parcial es capaz de propagar información sintáctica sobre los datos de entrada, tal como la estructura de los términos, y no solo valores constantes, haciendo así la evaluación parcial de los programas lógicos más potente y simple que la evaluación parcial de los programas funcionales.

La relación entre la deducción parcial y la transformación de programas basada en técnicas de plegado/despliegado también ha sido objeto de estudio en años recientes [18, 85, 115, 123, 135]. Como se indica en [115], la evaluación parcial es esencialmente un subconjunto de las transformaciones de plegado y despliegado en la que se hace uso casi exclusivo de la regla de despliegado como herramienta de transformación básica (solamente se obtiene una forma limitada de plegado, al exigir que los programas transformados cumplan la condición de cierre). Por lo tanto, las estrategias basadas en las técnicas de plegado/despliegado consiguen una serie de optimizaciones de los programas

(eliminación de estructuras de datos innecesarias, eliminación de argumentos redundantes, fusión de computaciones recursivas, etc) que la deducción parcial no puede obtener. Como contrapartida, la deducción parcial es menos compleja computacionalmente y se puede automatizar más fácilmente. Se han realizado esfuerzos por aunar lo mejor de ambas técnicas de transformación. Por ejemplo, [116] presenta un algoritmo de evaluación parcial expresado en términos de las técnicas de transformación de plegado/desplegado. Siguiendo una orientación distinta, [62, 85] consideran la extensión del marco clásico de la deducción parcial [88] para que se puedan especializar conjunciones de átomos, lo que permite conseguir algunos de los beneficios adicionales que obtienen las técnicas de plegado y desplegado. Esta nueva técnica ha recibido el nombre de *deducción parcial conjuntiva (Conjunctive Partial Deduction)*.

10.4 SÍNTESIS DE PROGRAMAS

La síntesis de programas ha sido un campo de trabajo muy activo al margen de la programación lógica; sin embargo, desde los primeros tiempos de la PL, la síntesis fue una de las áreas que ha suscitado mayor interés.

La *síntesis de programas* hace referencia al desarrollo sistemático de programas a partir de especificaciones (posiblemente no ejecutables). Cuando tal desarrollo sistemático consiste en la transformación de la especificación dada en un programa, el proceso se conoce como síntesis deductiva [67] (llamada también síntesis *transformacional*) y puede verse ésta como una simple forma de optimización automática del código. En particular, en la programación lógica el proceso de optimización automática se denomina síntesis cuando la especificación original no es ejecutable. A modo de ejemplo, la siguiente especificación del predicado *subset*

$$\text{subset}(L_1, L_2) \Leftrightarrow \forall X \text{ member}(X, L_1) \Rightarrow \text{member}(X, L_2)$$

junto con la definición estándar para *member*/2

$$\text{member}(X, L) \Leftrightarrow L = [H|T] \wedge (X = H \vee \text{member}(X, T))$$

el proceso de síntesis consistiría en transformar la especificación de *subset* en el siguiente programa lógico

$$\begin{aligned} &\{ \text{subset}([], L_2). \\ &\quad \text{subset}([H|T], L_2) \leftarrow \text{member}(H, L_2), \text{subset}(T, L_2) \}. \end{aligned}$$

que se puede obtener aplicando las técnicas de transformación estudiadas en la sección anterior.

La síntesis de programas lógicos recursivos a partir de información *incompleta*, como podrían ser ejemplos concretos de entrada/salida, es otra subárea diferente de este campo de trabajo, conocida como síntesis inductiva y relacionada muy estrechamente con la *Programación Lógica Inductiva (ILP)*, que responde a un problema más amplio.

10.4.1. Programación Lógica Inductiva

En su forma más general, la tarea de la Programación Lógica Inductiva es inferir una hipótesis H a partir de cierta información (incompleta) E , que se conoce como la “evidencia” y que se asume como incompleta, y de una base de conocimiento B , de forma que $B \wedge H \models E$, donde E , H y B son conjuntos de cláusulas. En la evidencia E se suele distinguir entre la evidencia positiva $E+$ y la evidencia negativa $E-$. En la práctica, $E+$ está formado por un conjunto de literales básicos positivos (ejemplos positivos), mientras $E-$ se restringe a literales básicos negativos (ejemplos negativos). Esto da lugar a una descripción extensional, mientras que las hipótesis constituyen una descripción intensional. En la terminología propia del área del aprendizaje computacional (*machine learning*), se dice que la descripción de conceptos H se debe *aprender* de instancias y contraejemplos de conceptos E , representados éstos mediante símbolos de predicado. El caso más interesante es aquél en el que las hipótesis a aprender son *recursivas*, es decir, al menos un átomo en el cuerpo de una de las cláusulas de H tiene como símbolo de predicado el mismo símbolo que el del átomo en la cabeza de H . Los programas recursivos *computan algo*, lo que no siempre ocurre con aquéllos no recursivos, que pueden tratarse de meros *clasificadores* de datos como pertenecientes a uno u otro concepto.

Por ejemplo, dados los ejemplos positivos (en la columna de la izquierda) y los ejemplos negativos (columna de la derecha)

$\text{subset}([], []).$	$\neg\text{subset}([k], []).$
$\text{subset}([], [a, b]).$	$\neg\text{subset}([n, m, m], [m, n]).$
$\text{subset}([d, c], [c, d, e]).$	
$\text{subset}([h, f, g], [f, i, g, h, j]).$	

y adoptando como base de conocimiento el siguiente programa lógico

```

 $\text{select}(X, [X|Xs], Xs).$ 
 $\text{select}(X, [H|Ys], [H|Zs]) \leftarrow \text{select}(X, Ys, Zs).$ 

```

El siguiente programa lógico sería una hipótesis

```

 $\text{subset}([], Xs).$ 
 $\text{subset}([X|Xs], Ys) \leftarrow \text{select}(X, Ys, Zs), \text{subset}(Xs, Zs).$ 

```

Recordemos aquí que hablamos de síntesis de programas solo cuando se parte de una especificación original *completa*, como la axiomatización $\text{subset}(L_1, L_2) \leftrightarrow \forall X \text{member}(X, L_1) \Rightarrow \text{member}(X, L_2)$.

10.4.2. Aproximaciones y extensiones de ILP (y síntesis induktiva)

El agente que proporciona las entradas a una técnica de ILP se suele denominar el instructor (*teacher*), mientras la técnica en sí se denomina técnica de inducción o

aprendizaje (y el agente que la realiza se denomina aprendiz *learner*). La evidencia de que se dispone inicialmente, aunque *incompleta*, se asume *correcta*; es decir, describe un subconjunto finito de la relación esperada. En estas circunstancias se suele decir que no hay *ruido*.

La inducción puede entenderse como un proceso de búsqueda en un grafo (o espacio de estados) donde los nodos corresponden a hipótesis y los arcos corresponden a operadores que transforman dichas hipótesis. Como es habitual, el desafío consiste en navegar en dicho espacio de estados de forma eficiente, aplicando alguna técnica de control inteligente (por ejemplo, organizando el espacio de estados de acuerdo a un orden parcial y utilizando técnicas de poda).

La inducción puede ser *interactiva* o *pasiva*, dependiendo de si la técnica hace preguntas (*queries*) a algún *oráculo* (*informant*) o no. Las cuestiones planteadas pueden ser muy variadas, incluyendo la petición de clasificar en positivos o negativos algunos ejemplos inventados por el algoritmo de síntesis.

La inducción puede ser *incremental* o *no incremental*, dependiendo de si las evidencias se proporcionan de una en una (con salida ocasional de algunas hipótesis intermedias) o todas al principio del proceso (generándose como salida en ese caso una única hipótesis final).

La inducción puede ser *ascendente* (*bottom-up*) o *descendente* (*top-down*), dependiendo de si las hipótesis evolucionan monótonamente partiendo de lo más específico (que sería el programa lógico vacío) o desde lo más general (que sería aquel programa lógico que tuviese éxito para todos los objetivos iniciales posibles). De otra forma, la síntesis descendente es una forma de especialización), mientras la síntesis ascendente se corresponde con un proceso de generalización.

Una vez que se acepta una hipótesis (por las razones que sea), uno puede querer validarla. Ya que no se dispone de una descripción completa de la relación esperada, solo resulta posible *validar* la hipótesis pero no verificarla matemáticamente. Idealmente, una hipótesis cubre todas las evidencias (positivas) disponibles. Resulta por tanto razonable validar la hipótesis midiendo su precisión (expresada en porcentajes) en cubrir de forma correcta otras evidencias. Por este motivo, la evidencia proporcionada inicialmente se denomina el conjunto de entrenamiento (*training set*), mientras la evidencia adicional que se aporta posteriormente se llama conjunto de test (*test set*).

Un *criterio de identificación* define el momento en que una técnica de inducción consigue identificar la relación esperada.

Del mismo modo que las técnicas de transformación pueden introducir nuevas funciones (o predicados), también la síntesis inductiva *inventa* predicados. La invención de predicados se define como sigue: (1) introducir en las hipótesis algunos predicados que no están en la evidencia ni en el conocimiento de fondo (*background*); (2) inducir programas para definir estos nuevos predicados. Esto requiere el uso de reglas *constructivas* de inferencia inductiva, donde el consecuente inductivo puede involucrar símbolos que no están en el antecedente, en contraposición con el caso de las reglas puramente *selectivas*.

Una generalización de la técnica de ILP se conoce como *revisión de teorías* o también *depuración declarativa*. La idea clave aquí es que se proporciona una entrada adicional, conocida como hipótesis inicial (o teoría) H_i , con la restricción de que la hipótesis final H será una variante de ésta tan próxima como sea posible, en el sentido de que solo los posibles errores (*bugs*) de H_i con respecto a E serán encontrados y corregidos (*debugged*) incrementalmente para producir H . Esta generalización se reduce al caso normal en sus dos casos extremos; es decir, cuando se parte de la teoría H_i más específica o más general, dependiendo de si el proceso de inducción procede de manera ascendente o descendente. En otra terminología, se habla a veces de aprendizaje *conducido por modelos* o *conducido por los datos*, en cuyo caso no hay teoría inicial. Uno de los primeros sistemas para sintetizar programas lógicos a partir de ejemplos es el Sistema de Inferencia de Modelos (MIS) de Shapiro [137], que puede verse alternativamente como un caso especial de depurador de programas. MIS está basado en modelos y es incremental, en el sentido de que los ejemplos se van introduciendo uno a uno a partir del programa vacío. Para cada nuevo ejemplo introducido, el programa sintetizado a partir de los ejemplos previos se actualiza para cubrir también el nuevo ejemplo. La estrategia general de MIS es la siguiente. Para cada nuevo ejemplo, si se trata de un ejemplo positivo que no está cubierto por el programa, entonces se añade al programa una nueva cláusula que lo cubra. Si se trata de un ejemplo negativo que está cubierto por el programa, entonces se elimina de éste la cláusula que lo cubre. Si el programa resultante es inconsistente respecto a los ejemplos previos, el programa se modifica siguiendo la estrategia anterior. De esta forma, los programas generados son siempre correctos con respecto a los ejemplos introducidos.

La depuración de programas es otra técnica formal orientada a la detección automática de errores. Pero además pueden incluirse una fase posterior, también automática. En la sección 10.5.1 profundizaremos más en la fecunda relación que existe entre la síntesis inductiva la y depuración de programas, comparable por otro lado a la natural conexión ya estudiada entre síntesis deductiva y transformación.

10.4.3. Corrección de la síntesis

El problema de verificar la corrección del programa sintetizado es complementario al de la síntesis. Específicamente, hemos de verificar que el programa sintetizado satisface su especificación o, más generalmente, que el método de síntesis es correcto (*sound*); es decir, que siempre produce programas correctos.

Los criterios para la corrección relacionan la relación esperada (o más bien su especificación lógica) y el programa lógico sintetizado. La especificación y el programa denotan alguna relación, de acuerdo con cierto tipo de semántica. En lo que sigue introducimos criterios de corrección paramétricos con respecto a la semántica considerada.

Definición 10.2 Sea P un programa lógico e I la especificación de la semántica deseada de P . Entonces,

1. P es parcialmente correcto con respecto a I , si $\llbracket P \rrbracket \subseteq \llbracket I \rrbracket$.
2. P es completo con respecto a I , si $\llbracket P \rrbracket \supseteq \llbracket I \rrbracket$.
3. P es totalmente con respecto a I , si $\llbracket P \rrbracket = \llbracket I \rrbracket$.

La corrección parcial requiere que la semántica de P esté incluida en la semántica de I . La completitud es la propiedad inversa de la corrección parcial y requiere que el significado de la especificación esté incluido en la semántica del programa. La combinación de corrección parcial y completitud se denomina corrección total.

¿No sería bonito poder inducir programas correctos a partir de unos cuantos ejemplos correctos de su funcionamiento? Este viejo ideal de la Programación Automática ha constituido una intensa área de investigación desde los años 60 que, desde la perspectiva de la síntesis, se conoce como *programming-by-examples*. El cuello de botella de esta técnica está en el uso del conocimiento de *background*. En cualquier escenario de programación realista, la información de *background* consta de cláusulas para numerosos predicados, como sucede en el proceso de síntesis que realizan los humanos. Sin embargo, nosotros los humanos tenemos tendencia a organizar dinámicamente este conocimiento de acuerdo con criterios de oportunidad o relevancia, de manera que no se nos ocurriría utilizar la definición de la relación “abuela” para construir un programa de ordenación. El problema de encontrar la forma de definir criterios semejantes para organizar o seleccionar la información de *background* no resulta en absoluto trivial. Una aproximación natural surge de la idea de simular mecánicamente la creatividad (lo que se conoce como *knowledge discovery*). La extracción de conocimiento (y también el campo cercano a ésta conocido como minería de datos) se relacionan con la obtención y transformación de información escondida en conocimiento valioso a través del descubrimiento de relaciones y patrones entre dichos datos. Si bien esto puede sonar como una vaga reformulación de las tareas de la ILP, debe más bien considerarse como una aplicación de ésta al caso en que se trabaja con datos muy voluminosos. Para una revisión en profundidad de estos conceptos, se sugiere al lector consultar [57].

10.4.4. Síntesis constructiva

La síntesis constructiva es una aproximación que se originó en el paradigma de programación funcional. También es conocida esta aproximación con el nombre de *proofs-as-programs* [10]. Con los años, se ha ido convirtiendo también en un importante campo de trabajo dentro de la programación lógica.

En el campo de la programación funcional, la síntesis constructiva se basa en el isomorfismo de *Curry-Howard* de la teoría de tipos constructiva, que establece que hay una correspondencia uno a uno entre la prueba constructiva de la existencia de un teorema y

un programa (es decir, una función) que computa valores *testigo* de las variables cuantificadas existencialmente de dicho teorema. Es decir de la prueba (constructiva) de una fórmula de la forma

$$\forall i. \exists o. r(i, o) \quad (10.1)$$

es posible extraer un programa tal que, para todas las entradas i , compute una salida o que satisfaga la relación considerada r . De esta forma, el proceso de síntesis constructiva consta de dos pasos:

1. construir la fórmula 10.1 y demostrarla en una lógica constructiva.
2. extraer de la prueba un programa para computar r .

Es interesante hacer notar que el programa extraído es una función y que la teoría de tipos es usualmente una lógica (tipificada) de orden superior, por lo que la descripción de esta técnica cae fuera del alcance de esta obra, en la que no consideramos programas lógicos tipados o de orden superior.

10.5 DIAGNÓSTICO Y DEPURACIÓN AUTOMÁTICA DE PROGRAMAS

La *depuración* es el proceso mediante el cual se corrige un programa, buscando eliminar las discrepancias entre lo que calcula realmente y lo que había sido especificado. Aunque el *diagnóstico* se refiere a la identificación de un error en un programa que se comporta incorrectamente y la *depuración* al proceso más general de identificación, localización y corrección de dicho error [136, 53], en la literatura relacionada [33] se utilizan en general ambos términos indistintamente.

La idea principal de la depuración de programas es la de, a partir de una especificación *esperada* del programa, comprobar si los resultados obtenidos por el programa coinciden con los de dicha especificación. La especificación puede darse en formas diferentes, por ejemplo mediante otro programa, mediante una interpretación, mediante un conjunto de respuestas, etc.

No debe confundirse un depurador con un *tracer*, que simplemente permite al programador seguir la ejecución *paso a paso* o introduciendo *breakpoints*. Se debe notar que la depuración permite detectar dos tipos de errores: errores de *incorrectitud* y errores de *insuficiencia*. Una incorrectitud ocurre cuando el programa calcula algo que no coincide con lo esperado por el programador, mientras que una insuficiencia ocurre cuando el programa deja de calcular un resultado esperado por el programador, es decir, cuando no consigue obtener una respuesta dada. El depurador no debe solo diagnosticar si se ha producido un error sino que debe ser capaz de encontrar el punto desde el que se deriva dicho error. En caso de que se trate de una incorrectitud, debe encontrar el punto en el que se produce la discrepancia entre lo esperado y lo que se obtiene. En relación

a los errores derivados de la insuficiencia, el depurador intentará reconstruir el proceso hasta el momento de la ejecución en el que se produce un fallo, detectando así dónde se ha quedado bloqueado el cálculo del resultado.

10.5.1. Diagnóstico Declarativo

El *diagnóstico (depuración) declarativo* se define como sigue, en concordancia con los enfoques dados en este campo por [33, 136, 52]. Dado un programa P , la semántica $\llbracket P \rrbracket$ y la especificación M de la semántica deseada de P , el diagnóstico declarativo es un método para probar la corrección y completitud de P con respecto a M , o para determinar los errores y los componentes del programa que son fuente de error, en el caso en que $\llbracket P \rrbracket \neq M$. Para efectuar la depuración declarativa es, por tanto, necesario especificar la semántica real del programa y su semántica deseada, buscando establecer las posibles discrepancias entre ambas. Para expresar el significado esperado existen dos enfoques fundamentales:

- Uso de *oráculos* [136]: típicamente, el depurador establece un diálogo con el usuario, que identifica síntomas de error (respuestas incorrectas, respuestas perdidas), para que el sistema localice la causa del fallo y emita un diagnóstico [102]. En otros depuradores, el oráculo se implementa mediante una especificación cuya semántica se corresponde con la que se esperaría del programa.
- Método de *aserciones* [45]: el oráculo se reemplaza por anotaciones en el programa (aserciones) que reflejen adecuadamente el significado pretendido del programa [31].

En cuanto a la semántica, y dado que la depuración declarativa se relaciona con las propiedades de la teoría de modelos de la programación declarativa, las semánticas declarativas exploradas más comúnmente son: el modelo mínimo de Herbrand [136], el conjunto de modelos de la compleción del programa [86] y el conjunto de consecuencias lógicas atómicas del programa [52, 53]. Es posible enfocar los métodos de diagnóstico de acuerdo con una propiedad que se quiera observar en una computación, mediante la especificación de una semántica que la modele adecuadamente. La definición del observable depende del paradigma de programación y de la propiedad a observar. Algunos ejemplos de observables de la programación lógica son: patrones de llamada, respuestas computadas, respuestas parciales, resultantes, terminación, éxitos y fallos (definidos en [34]).

La siguiente definición formaliza el diagnóstico con respecto al observable α [34] y es una extensión para el diagnóstico de respuestas computadas de las definiciones dadas en [136, 86, 52].

Sea P un programa, α una propiedad observable, M^α la especificación de la semántica deseada de P que modela el observable α y $\llbracket P \rrbracket^\alpha$ la semántica operacional de P con respecto a α .

1. P es parcialmente correcto con respecto a M^α , si $\llbracket P \rrbracket^\alpha \subseteq M^\alpha$
2. P es completo con respecto a M^α , si $M^\alpha \subseteq \llbracket P \rrbracket^\alpha$
3. P es totalmente correcto con respecto a M^α , si $M^\alpha = \llbracket P \rrbracket^\alpha$

Un síntoma es la aparición de una anomalía durante la ejecución de un programa [53], que resulta de la comparación de la semántica del programa con la semántica deseada por el programador, como lo precisa la siguiente definición [35]:

Sea P un programa, $\llbracket P \rrbracket$ la semántica de P e I la semántica deseada de P .

1. Un síntoma de incorrección es un átomo p tal que $p \in \llbracket P \rrbracket$ y $p \notin I$
2. Un síntoma de incompletitud es un átomo tal que $p \in I$ y $p \notin \llbracket P \rrbracket$

Desde el punto de vista de la ejecución del programa, un síntoma de incorrección es un resultado que no está en las expectativas del usuario y un síntoma de incompletitud se presenta cuando el programa es incapaz de computar algún resultado esperado por el usuario. Los algoritmos de depuración declarativa clásicos exigen que la semántica deseada M sea declarada extensionalmente. Sin embargo, en general M es infinita. Los algoritmos pueden manejar este tipo de semánticas trabajando con síntomas que se obtienen usando técnicas de generación de tests. Estos algoritmos son los llamados algoritmos dirigidos por los síntomas.

Otra forma de manejar semánticas infinitas es utilizar aproximaciones de la semántica, que generalmente se basan en la teoría de Interpretación Abstracta. El diagnóstico abstracto es una generalización de la técnica de diagnóstico declarativo en la que se consideran propiedades relacionadas con la semántica operacional y observables, en lugar de las propiedades basadas en la semántica declarativa. Las técnicas de diagnóstico abstracto aparecen, por tanto, como una combinación de tres técnicas bien conocidas: la depuración declarativa [136, 86], la aproximación conocida como *s*-semántica (o semántica de respuestas computadas) para la definición de programas que modelen varios comportamientos observables y, por último, la teoría de la interpretación abstracta.

10.5.2. Depuración Declarativa en Programas Lógicos Puros

Las técnicas de depuración declarativa estándar de los programas lógicos puros se formulan generalmente en el marco de la teoría de modelos lógica. La formulación más clásica se basa en la semántica del menor modelo de Herbrand del programa, la cual es adecuada para modelar las características de los programas lógicos en el caso ground. Las *s*-semánticas [51, 49] constituyen una caracterización más cercana al comportamiento del programa y, consecuentemente, constituyen una herramienta más potente de ayuda a las técnicas de depuración declarativa. Como vimos en la Sección 5.6, en ellas se utiliza una noción de interpretación no estándar que básicamente consiste en un subconjunto de la base de Herbrand extendida, formada por todos los átomos, posiblemente

con variables, que pueden formarse con los símbolos del alfabeto del programa P (módulo la varianza inducida por el cambio de nombre). Gracias a la equivalencia entre la semántica operacional del conjunto de éxitos no básicos (cf. 5.21) $\mathcal{ENB}(P)$ con respecto a un programa P y el menor punto fijo (de una extensión natural) del operador de consecuencias inmediatas para el observable de respuestas computadas T_P^s con respecto a un programa P , es posible generar fácilmente algoritmos cuya estrategia de búsqueda puede ser descendente o ascendente.

En [35], se propone un algoritmo de diagnóstico cuya estrategia es descendente y se diferencia de las que se basan en árboles de computación [136] o demostración [100, 86] en que la búsqueda se realiza con objetivos atómicos más generales, que no precisa síntomas como entradas y que se introduce la posibilidad de simular el oráculo. La diferencia radica en que, dado que en el proceso de diagnóstico se conoce la s -semántica deseada I , al comparar ésta con el resultado de un paso de aplicación del operador $T_P^s(I)$, es posible determinar las cláusulas incorrectas del programa, tal y como se formaliza en las siguientes definiciones. Para simplificar la notación, escribiremos simplemente T_P en vez de T_P^s .

Definición 10.3 Sea I la especificación la semántica deseada de P . Si existe un atomo $p \in T_{\{c\}}(I)$ tal que $p \notin I$, entonces la cláusula $p \in P$ es incorrecta con respecto a p . Decimos también que p está cubierta incorrectamente por c .

Por tanto, la incorrección de la cláusula c puede ser detectada por una simple transformación de la semántica deseada I .

Definición 10.4 Sea I la especificación la semántica deseada de P . Un atomo p no está cubierto por P , si $p \in I$ and $p \notin T_P(I)$.

Es decir, se dice que p no está cubierto si no puede derivarse mediante una cláusula de programa, usando el operador de consecuencias inmediatas del programa.

Proposición 10.2 Si no hay en P cláusulas incorrectas con respecto a la especificación la semántica deseada I , entonces P es parcialmente correcto respecto a la semántica del conjunto de éxitos.

La Proposición 10.2 sugiere por tanto una metodología muy simple para comprobar la corrección parcial. La completitud es un problema más difícil y pueden darse algunos casos de incompletitud que no puedan detectarse comparando la especificación I de la semántica deseada de P y $T_P(I)$. Es decir, la ausencia de ecuaciones no cubiertas no permite inferir que el programa sea completo. Este problema se relaciona con la posible existencia de varios puntos fijos para el operador T_P operator (para los detalles técnicos, ver [35]).

Proposición 10.3 Si T_P tiene un único punto fijo y no hay átomos no cubiertos, entonces P es completo con respecto a \mathcal{I} .

Alternativamente, la generación de respuestas computadas, en programas lógicos puros, también puede ser descrita usando árboles de computación o demostración, que como sabemos difieren de los árboles de búsqueda o ejecución convencionales. Cada nodo en el árbol de computación contiene un átomo “probado” en la ejecución del programa; sus hijos son el cuerpo de la instancia de la cláusula que se utilizó para derivar el nodo, existiendo un hijo por cada átomo del cuerpo; las hojas son instancias de hechos del programa o predicados del sistema. Es importante tener en cuenta que, una vez obtenido el árbol, que se supone finito, la información contenida es independiente de la estrategia de búsqueda y de la regla de computación. El árbol de demostración se puede construir por un meta intérprete [132], por una transformación del programa [101] o se puede usar el átomo para representar de manera implícita su propio árbol de demostración [100].

La depuración declarativa para respuestas computadas incorrectas consiste en la búsqueda de un nodo equívoco (nodo no válido con hijos válidos). Para ello, el árbol de computación se recorre preguntando al oráculo por la validez de cada nodo. En [100], se prueba que el proceso de búsqueda para árboles de computación finitos, termina con éxito y, si tiene nodos equívocos más altos, los encuentra todos. Este resultado determina la corrección y completitud del esquema general para algoritmos de diagnóstico declarativo de respuestas incorrectas, que se aplica al caso de árboles de computación finitos que tengan nodos equívocos más altos. En resumen, para el diagnóstico de respuestas computadas incorrectas se busca una instancia de cláusula cuya cabeza sea incorrecta y el cuerpo válido en la interpretación deseada. Las diferentes aproximaciones varían según el tipo de programa sobre el cual se aplica, la semántica utilizada para interpretar el programa, la estrategia de búsqueda, la clase de preguntas, la simulación y el uso de afirmaciones en la concepción del oráculo.

De manera similar al caso de respuestas computadas incorrectas, en los diferentes algoritmos propuestos para la depuración de soluciones perdidas, la concepción en general es la misma y las diferencias se establecen en cuanto a las estrategias de búsqueda, el uso o no de síntomas y las preguntas realizadas al oráculo. En los enfoques clásicos, el manejo del oráculo presenta grandes dificultades debido a la cantidad de información, los supuestos y las restricciones que se deben imponer. Con el uso de las *s*-semánticas, el horizonte de aplicación es más amplio y es posible probar que la ausencia de átomos que no están cubiertos implica la completitud para los programas cuyo operador de consecuencias inmediatas asociado tenga un único punto fijo. La implicación anterior no es cierta si la unicidad del operador no se cumple, dado que, existen programas que no tienen átomos que no están cubiertos y no son completos [35, 36].

RESUMEN

En este capítulo hemos abordado el estudio de las relaciones entre los métodos formales ágiles, basados en la lógica, y el software automático de calidad.

- Se introduce la aproximación ágil (*lightweight*) para el desarrollo del software, que propugna el uso parcial de los métodos formales. Esto contrasta con la aplicación tradicional de los métodos formales, que fomenta una formalización excesiva (y a la vez poco práctica) mediante el empleo de nociones que requieren una formación matemática poco habitual en los usuarios finales.
- Siguiendo un enfoque moderno, se describen los tres elementos de la trilogía del software —programas, datos y propiedades— y se estudian los procesos formales que transforman dichas componentes automáticamente. Esto incluye, entre otros, los siguientes mecanismos: síntesis de programas, transformación de programas y evaluación parcial, depuración y verificación automática.

CUESTIONES Y EJERCICIOS

Cuestión 10.1 *¿Cuál de las siguientes parejas de procesos formales se aplican en sentidos opuestos sobre el esquema de la trilogía del software (es decir, uno de los procesos lleva de componentes de tipo A a B, mientras que el otro transforma componentes de tipo B a A)?:*

1. *inferencia de tipos e inferencia de especificaciones;*
2. *inferencia de especificaciones y aprendizaje de programas;*
3. *aprendizaje de programas y síntesis de programas;*
4. *síntesis de programas a partir de ejemplos y generación de juegos de datos.*

Cuestión 10.2 *¿Cómo se representaría la la síntesis o derivación formal de programas dentro de la trilogía del software?:*

1. *como un proceso de datos a requisitos;*
2. *como un proceso de datos a programas;*
3. *como un proceso de programas a requisitos;*
4. *como un proceso de requisitos a programas .*

Cuestión 10.3 *Indicar cuál de los siguientes características no pertenece a la corriente conocida como lightweight formal methods:*

1. parcialidad (*en el lenguaje, análisis, modelo, automatización, etc*);
2. adaptación a las necesidades inmediatas y particulares del proyecto;
3. conocimiento, por parte de los usuarios, de los detalles de diseño de la herramienta (*lenguaje de implementación, formalismo subyacente, etc*);
4. uso puntual de formalismos en diversas etapas del ciclo de vida.

Cuestión 10.4 La evaluación parcial se entiende en la trilogía del software como:

1. un proceso de datos a programas;
2. un proceso de programas a programas;
3. un proceso de programas a requisitos;
4. un proceso de requisitos a programas.

Cuestión 10.5 La evaluación parcial es una técnica de:

1. síntesis de programas;
2. aprendizaje de programas;
3. transformación de programas;
4. verificación de programas.

Cuestión 10.6 ¿Cuál de las siguientes afirmaciones es cierta para la deducción parcial?:

1. se basa en la construcción de árboles de computación parciales (*hasta que el conjunto de resultantes asociados cubre todas las computaciones previstas dentro el objetivo especializado*);
2. dado un programa y un objetivo, elimina todo cómputo relacionado con el objetivo;
3. dado un programa y un objetivo, ejecuta el objetivo hasta obtener una solución;
4. dado un programa y un objetivo, elimina todo cómputo no relacionado con el objetivo.

Cuestión 10.7 ¿Cuál de las siguientes afirmaciones NO es cierta para la especialización de programas?:

1. construye una traza de la ejecución del objetivo a evaluar y elimina toda la información estática;

2. *elimina los cómputos que dependen de la información estática presente en el programa;*
3. *dado un programa y un objetivo, mejora el coste temporal de la ejecución de ese objetivo en el programa;*
4. *elimina todo cálculo relacionado con el objetivo a especializar.*

Cuestión 10.8 Una estrategia de despliegado de llamadas debe:

1. *evitar el despliegado infinito;*
2. *asegurar la repetición de llamadas;*
3. *potenciar la creación de funciones residuales;*
4. *ninguna de las anteriores.*

Cuestión 10.9 Dado un intérprete `int` y un evaluador parcial `mix`, indicar cuál de las siguientes ecuaciones define un generador de compiladores `cogen`:

1. `cogen=[[int]] [int,int];`
2. `cogen=[[mix]] [mix,int];`
3. `cogen=[[int]] [mix,mix];`
4. `cogen=[[mix]] [mix,mix].`

Cuestión 10.10 Dado un intérprete `int` y un evaluador parcial `mix`, indicar cuál de las siguientes ecuaciones define un compilador:

1. `comp=[[mix]] [int,int];`
2. `comp=[[mix]] [mix,int];`
3. `comp=[[int]] [mix,mix];`
4. `comp=[[mix]] [mix,mix].`

Fundamentos y Notaciones Matemáticas

En este apéndice nuestra intención es resumir brevemente las nociones matemáticas básicas que emplearemos en este libro, así como fijar cierta terminología. Una introducción asequible a estos temas puede encontrarse en [63] y en [129].

A.1 CONJUNTOS

Un *conjunto* es una colección de objetos. Los objetos que forman un conjunto se denominan elementos o miembros del conjunto. Designaremos los conjuntos mediante letras mayúsculas: A , B , C , etc. Si es preciso, también utilizaremos subíndices: A_1 , B_1 , C_1 , etc. Escribimos $e \in A$ para indicar la pertenencia de un elemento e al conjunto A , y escribimos $e \notin A$ para indicar el hecho contrario. Conviene aceptar como conjunto a la colección vacía de objetos, que no contiene ningún elemento; a dicho conjunto se le denomina *conjunto vacío*, y se le denota mediante el símbolo ' \emptyset '. Es habitual describir los conjuntos listando sus elementos entre llaves '{' y '}'.

Ejemplo A.1

El conjunto de las letras minúsculas del alfabeto se representa como: $\{a, b, c, \dots, z\}$.

Otra forma de expresar los conjuntos es asociándoles alguna propiedad característica mediante el *operador de construcción* de conjuntos '|'

Ejemplo A.2

El conjunto de las letras minúsculas del alfabeto también puede representarse como: $\{x \mid x \text{ es una letra minúscula del alfabeto}\}$. Sin necesidad de enumerar todos y cada uno

de sus elementos. La expresión anterior tiene la siguiente lectura: “el conjunto de elementos, x , tal que x es una letra minúscula del alfabeto”.

Desde esta perspectiva, el conjunto vacío está caracterizado por una propiedad P que no cumple ningún elemento.

Por convención, cualquier objeto puede ocurrir como mucho una vez en un conjunto, de forma que un conjunto no contiene elementos duplicados. El orden en el que aparecen los objetos tampoco cuenta.

Un conjunto de singular importancia es el de los números naturales $\{0, 1, 2, \dots\}$, del que describiremos algunas propiedades en el Apartado 6.3.1. Por el momento introducimos una serie de notaciones. Denotamos el conjunto de los números naturales mediante el símbolo \mathbb{N} . El segmento inicial $\{1, 2, \dots, k\}$ del conjunto de los números naturales positivos \mathbb{N}^+ se denota \mathbb{N}_k^+ . Cuando $k = 0$, entonces $\mathbb{N}_0^+ = \emptyset$. El conjunto de los números enteros $\{\dots, -2, -1, 0, 1, 2, \dots\}$ se denota por \mathbb{Z} . El conjunto de los números reales se denota por \mathbb{R} .

Decimos que S es un *subconjunto* de un conjunto C o que S está *incluido* en C , y escribimos $S \subseteq C$, si todo elemento de S es un elemento de C . Empleamos la notación $S \not\subseteq C$ para indicar que S no está incluido en C . Decimos que un conjunto A es *igual* a otro B , y escribimos $A = B$, si $A \subseteq B$ y $B \subseteq A$; esto es, A y B contienen los mismos elementos. Escribimos $A \neq B$ si no se cumple que $A = B$. Finalmente, escribimos $S \subset C$ para indicar que $S \subseteq C$, pero $S \neq C$; en este caso decimos que S es un *subconjunto propio* de C .

Emplearemos los símbolos usuales ‘ \cup ’, ‘ \cap ’, ‘ \setminus ’ para denotar la unión, intersección y diferencia (o diferencia relativa) de conjuntos, respectivamente. Si $A \cap B = \emptyset$, decimos que A y B son *conjuntos disjuntos*. Dado un conjunto A , $\wp(A)$ denota el conjunto de todos los subconjuntos de A . El conjunto $\wp(A)$ se denomina el *conjunto potencia* de A o el *conjunto de las partes* de A .

Ejemplo A.3

Dado el conjunto $B = \{0, 1\}$ de valores booleanos. El conjunto conjunto de las partes de B , $\wp(B)$, está constituido por: $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.

Un medio para la construcción de conjuntos es el producto cartesiano de conjuntos. La idea de producto cartesiano descansa sobre el concepto de n -tupla (ordenada). Una n -tupla es una colección ordenada de elementos, que denotamos por $\langle a_1, a_2, \dots, a_n \rangle$. Decimos que a_1 es el primer componente, a_2 es el segundo componente, ..., y a_n es el n -ésimo componente de la n -tupla. En una n -tupla el orden sí importa, esto quiere decir que $\langle a_1, a_2, \dots, a_n \rangle = \langle b_1, b_2, \dots, b_n \rangle$ si y solo si $a_i = b_i$, para todo $i = 1, 2, \dots, n$. El *producto cartesiano* de los conjuntos A_1, A_2, \dots, A_n , que denotamos por $A_1 \times A_2 \times \dots \times A_n$, es el conjunto de n -tuplas $\{\langle a_1, a_2, \dots, a_n \rangle \mid a_i \in A_i, \text{ para todo } i = 1, 2, \dots, n\}$. Denotamos también por A^n el producto cartesiano formado por las n -tuplas de elementos pertenecientes a un mismo conjunto A .

Ejemplo A.4

Dado el conjunto $B = \{0, 1\}$ de valores booleanos. El producto cartesiano B^2 está constituido por los pares: $\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle\}$.

A.2 RELACIONES Y FUNCIONES**A.2.1. Relaciones**

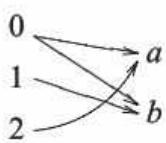
Una *relación* sobre los conjuntos A_1, A_2, \dots, A_n es un subconjunto del producto cartesiano $A_1 \times A_2 \times \dots \times A_n$. También se habla de relación *n-aria* (o *n-ádica*). Dada una relación *n-aria*, es habitual usar la notación $R(a_1, \dots, a_n)$ para indicar que $\langle a_1, a_2, \dots, a_n \rangle \in R$. Dada una relación 2-aria, R , decimos que R es una *relación binaria* entre A_1 y A_2 . Al conjunto A_1 se le denomina el *dominio* de la relación y al conjunto A_2 su *rango* o *codominio*.

Ejemplo A.5

Dados los conjuntos $A_1 = \{0, 1, 2\}$ y $A_2 = \{a, b\}$. El conjunto

$$R = \{\langle 0, a \rangle, \langle 0, b \rangle, \langle 1, b \rangle, \langle 2, a \rangle\} \subseteq A_1 \times A_2$$

es una relación binaria de A_1 a A_2 . Este conjunto expresa, por ejemplo, que 1 está relacionado con b mediante la relación R , ya que $\langle 1, b \rangle \in R$; pero 1 no está relacionado con a mediante la relación R , ya que $\langle 1, a \rangle \notin R$. Las relaciones pueden representarse, como se muestra a continuación, utilizando grafos dirigidos (un conjunto de nodos y arcos) y tablas:



R	a	b
0	V	V
1	F	V
2	V	F

Cuando se trata con relaciones binarias, en ocasiones, es conveniente utilizar la notación $a R b$ en lugar de $\langle a, b \rangle \in R$ o $R(a, b)$ para indicar que a está relacionado con b y $a \not R b$ en lugar de $\langle a, b \rangle \notin R$. La inversa de una relación binaria R es la relación $R^{-1} = \{\langle b, a \rangle \mid a R b\}$.

A.2.2. Funciones

Una clase de relaciones de interés especial son las funciones. Dados los conjuntos A y B , una *función* $f : A \rightarrow B$ es una relación $f \subseteq A \times B$ que verifica las siguientes condiciones:

ciones¹: (1) para todo $a \in A$, existe un $b \in B$ tal que $\langle a, b \rangle \in f$ y (2) si $\langle a, b \rangle, \langle a, b' \rangle \in f$, entonces $b = b'$.

Ejemplo A.6

La relación binaria R del Ejemplo A.5 no es una función ya que incumple la condición (2): $0Ra$ y $0Rb$, pero $a \neq b$.

Si no se cumple la condición (1), se dice que f es una función *parcial*. Cuando $\langle a, b \rangle \in f$, escribimos $f(a)$ para referirnos al elemento b y decimos que b es la *imagen* de a . Dada una función $f : A \rightarrow B$ y el subconjunto $C \subseteq A$, $f(C) = \{f(a) \mid a \in C\} \subseteq B$ se denomina conjunto *imagen* del subconjunto C . El *rango* de una función $f : A \rightarrow B$ es el conjunto *imagen* $f(A)$.

Ejemplo A.7

Sea la función $f : \mathbb{Z} \rightarrow \mathbb{N}$, tal que a cada número entero asigna su cuadrado. Esto es, $f(x) = x^2$. El dominio de f es \mathbb{Z} , su codominio es \mathbb{N} . Observe que f es una función porque a cada elemento del dominio le corresponde una sola imagen. El rango de f es subconjunto de los números naturales que son cuadrados perfectos, esto es, el conjunto $\{0, 1, 4, 9, \dots\}$.

Si $A = A_1 \times A_2 \times \dots \times A_n$, esto es, A es un producto cartesiano de n conjuntos, decimos que la f es una función *n-aria* (*n-ádica*, o de n variables).

Dadas dos funciones $g : A \rightarrow B$ y $f : B \rightarrow C$, la *composición* de ambas funciones se denota como $f \circ g : A \rightarrow C$ y se define de manera que $(f \circ g)(x) = f(g(x))$. La composición de funciones es asociativa. Esto es, dadas las funciones $g : A \rightarrow B$, $f : B \rightarrow C$ y $h : C \rightarrow D$, se cumple que $h \circ (f \circ g) = (h \circ f) \circ g$. Sin embargo, la composición de funciones no cumple la ley commutativa.

Ejemplo A.8

Sean las funciones $f : \mathbb{Z} \rightarrow \mathbb{Z}$ y $g : \mathbb{Z} \rightarrow \mathbb{Z}$ definidas de forma que $f(x) = 2x + 2$ y $g(x) = 3x - 2$. Tenemos que:

- $(f \circ g)(x) = f(g(x)) = f(3x - 2) = 2(3x - 2) + 2 = 6x - 2$,
- $(g \circ f)(x) = g(f(x)) = g(2x + 2) = 3(2x + 2) - 2 = 6x - 4$.

Por consiguiente, $f \circ g \neq g \circ f$.

Una función $f : A \rightarrow B$ es *inyectiva* (o *uno a uno*) si para todo $a, a' \in A$, $f(a) = f(a')$ implica que $a = a'$. Una función f es *sobreyectiva* si para todo $b \in B$ existe un $a \in A$ tal que $f(a) = b$. Una función *biyectiva* (o correspondencia *biunívoca*) es una función inyectiva y sobreyectiva. La Figura A.1 ilustra las cuatro posibles clases de funciones. Si $f : A \rightarrow B$ es biyectiva, la función $f^{-1} : B \rightarrow A$ definida por $f^{-1}(b) = a$ si y solo si

¹El concepto de función que manejamos aquí se denomina a veces *aplicación* o *función total* en la literatura.

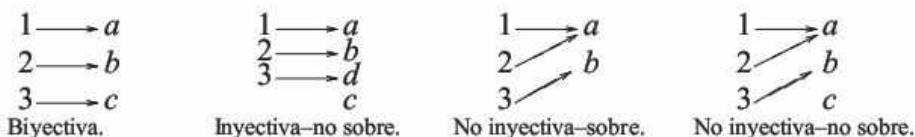


Figura A.1 Diferentes clases de funciones.

$f(a) = b$ se denomina función *inversa* de f .

Una función $f : A \rightarrow A$ es *idempotente* si para todo $a \in A$, $f(f(a)) = f(a)$. Un elemento $a \in A$ se denomina un *punto fijo* de f si $f(a) = a$. La función $id_A : A \rightarrow A$ definida por $id_A(a) = a$ para todo $a \in A$ se denomina la función *identidad* para el conjunto A .

Dadas las funciones $f : A \rightarrow B$ y $f_1 : A_1 \rightarrow B_1$, donde $A \subseteq A_1$ y $B \subseteq B_1$, decimos que f_1 es una *extensión* de la función f si para todo $a \in A$, $f_1(a) = f(a)$. A veces emplearemos el mismo símbolo ‘ f ’ para denotar tanto la función original como su extensión a un conjunto A_1 que incluye el dominio original A .

A.2.3. Numerabilidad y secuencias

Un conjunto A se dice *finito* si existe una biyección entre un segmento inicial \mathbb{N}_n^+ de los números naturales positivos y el conjunto A . Escribimos $|A|$ para denotar la *cardinalidad* del conjunto finito A , esto es, el número de elementos n que contiene el conjunto A .

Un conjunto A se dice que es *infinito numerable* si existe una biyección entre \mathbb{N}^+ y A . En cambio, diremos que es *infinito no numerable*, si existe una aplicación inyectiva, no biyectiva, de \mathbb{N}^+ en dicho conjunto. El concepto de *cardinalidad* puede extenderse a conjuntos infinitos con el fin de comparar el tamaño de dichos conjuntos. En general, dos conjuntos A y B se dice que tienen la misma cardinalidad si existe una biyección entre ambos.

Ejemplo A.9

El conjunto de los números pares es infinito numerable, ya que es posible construir la siguiente biyección:

$$\begin{array}{ccccccc} 0, & 1, & 2, & \dots & n, & \dots \\ \downarrow & \downarrow & \downarrow & \dots & \downarrow & \dots \\ 0, & 2, & 4, & \dots & 2n, & \dots \end{array}$$

Por consiguiente, el conjunto de los números pares y el de los números naturales tienen la misma cardinalidad.

Una *secuencia finita* s de elementos tomados de un conjunto A es una función $s : \mathbb{N}_n^+ \rightarrow A$, y una *secuencia infinita* s es una función cuyo dominio en el conjunto de los naturales positivos; esto es, $s : \mathbb{N}^+ \rightarrow A$. Emplearemos el término ‘secuencia’ para referirnos indistintamente tanto a las del primer tipo como a las del segundo. Nos

referiremos al n -ésimo elemento de la secuencia escribiendo s_n en lugar de $s(n)$, por consiguiente, escribiremos s_1, \dots, s_n para denotar una secuencia finita y s_1, s_2, \dots para denotar una secuencia infinita. Las secuencias también reciben el nombre de ‘sucesiones’.

Ejemplo A.10

Una *sucesión aritmética* es una secuencia $a_1, a_2, \dots, a_n, \dots$ de números, tal que la diferencia entre dos elementos consecutivos es una constante d . Esto es, $a_n = a_{n-1} + d$. Algunos ejemplos de sucesiones aritméticas son:

$$\begin{aligned} &1, 2, 3, 4, 5, 6, \dots \\ &3, 8, 13, 18, 23, 28, \dots \\ &0, 2, 4, 6, 8, 10, \dots \end{aligned}$$

A.2.4. Relaciones binarias

Una clase de relaciones muy importante es la que se establece entre los elementos de un mismo conjunto, por lo que cobra especial interés estudiar sus propiedades. Sea A un conjunto y $R \subseteq A \times A$ una relación binaria sobre A . Decimos que R es: *reflexiva*, si para todo $a \in A$, $a R a$; *irreflexiva*, si para todo $a \in A$, $a \not R a$; *transitiva*, si para todo $a, b, c \in A$, $a R b$ y $b R c$ implica $a R c$; *antisimétrica*, cuando $a R b$ y $b R a$ implica $a = b$; *simétrica*, cuando $a R b$ implica $b R a$.

Dado un conjunto A incluido en un conjunto U , y una propiedad P sobre los elementos de U , el *cierre* de A respecto a P es el menor conjunto incluido en U que contiene A y satisface P . Denotamos como $R^=$ el *cierre reflexivo* de la relación R : $R^= = R \cup D$, donde $D = \{(a, a) \mid a \in A\}$ es la relación diagonal (o identidad) sobre el conjunto A . El *cierre simétrico* de R es la relación $R \cup R^{-1}$. El *cierre transitivo* de R es la relación R^+ tal que $a R^+ b$ si existe una secuencia $a \equiv s_1, \dots, s_n \equiv b$, con $n \geq 1$ y $s_i R s_{i+1}$ para todo $i = 1, \dots, n - 1$. El *cierre reflexivo y transitivo* de R es la relación $R^* = R^= \cup R^+$.

Si la relación $R \subseteq A \times A$ tiene las propiedades reflexiva, transitiva y simétrica, se dice que es una *relación de equivalencia* sobre A . El subconjunto $[a]_R = \{b \mid b \in A \wedge a R b\}$ es la *clase de equivalencia* de a . El conjunto formado por las clases de equivalencia, denotado por A/R , se denomina *conjunto cociente* de la relación de equivalencia R sobre A . El conjunto cociente A/R determina una partición del conjunto A en subconjuntos no vacíos.

Ejemplo A.11

Dado el conjunto de los seres humanos, la relación definida como:

xRy , si x e y tienen los mismos padres,

es una relación de equivalencia que partitiona la humanidad en clases de equivalencia: los conjuntos formados por los individuos que son hermanos.

A.3 GRAFOS Y ARBOLES

Un *grafo* es una estructura compuesta de un conjunto de nodos (o vértices) y un conjunto de arcos² (o aristas) que conectan unos nodos con otros. Hay diversos tipos de grafos dependiendo de la naturaleza de los arcos y del número de ellos que conectan un par de nodos. La Figura A.2 proporciona una representación de dos clases distintas de grafos, en la que los nodos se muestran enmarcados por círculos y los arcos por segmentos no orientados o bien dirigidos.

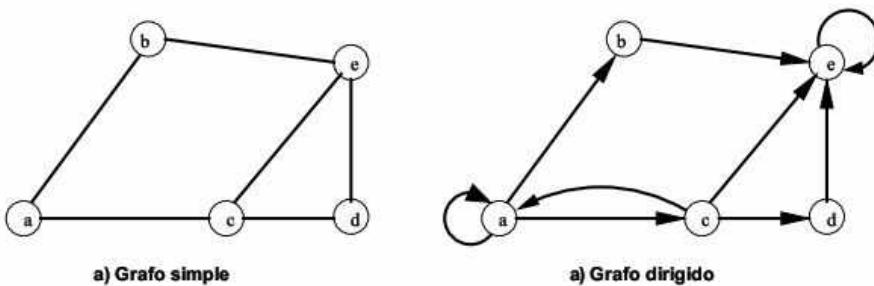


Figura A.2 Distintas clases de grafos.

A.3.1. Grafos no dirigidos

En un *grafo simple* los arcos se corresponden con pares no ordenados de nodos (i.e., los arcos no imponen una direccionalidad y es posible recorrerlos en ambos sentidos). Por otra parte, un arco solamente puede conectar un único par de nodos. Así pues, existe una función f que a cada arco a le asocia un par de nodos. Se dice que los nodos u y v son *adyacentes* o *vecinos* si existe un arco a del grafo, tal que $f(a) = \{u, v\}$. También se dice que el arco a *conecta* los nodos u y v , que se denominan *puntos terminales* del arco. El *grado* de un nodo es el número de arcos que inciden sobre él.

Si dos nodos están conectados por más de un arco (esto es, existen arcos a_i y a_j tales que $f(a_i) = f(a_j)$), hablamos de que el grafo es un *multigrafo*. Un arco que conecta un nodo con él mismo, se denomina *bucle*. Un multigrafo que contiene bucles se denomina *pseudografo* en [129]. Por tanto, un pseudografo es el tipo más general de grafo.

Un *camino* entre un nodo u y un nodo v es una secuencia de arcos a_1, a_2, \dots, a_n del grafo tales que $f(a_1) = \{u, x_1\}$, $f(a_2) = \{x_1, x_2\}, \dots, f(a_n) = \{x_{n-1}, v\}$. La *longitud* de un

²Algunos autores reservan la palabra “arco” para las aristas de los grafos dirigidos.

camino es el número de arcos que lo componen. Cuando el grafo es simple, un camino suele denominarse mediante la secuencia de nodos que atraviesa, $u, x_1, x_2, \dots, x_{n-1}, v$, ya que el listado de esos nodos determina el camino de forma única. Un camino que comienza y termina en el mismo nodo se denomina *círculo*. Un *círculo simple* es aquél que no contiene el mismo arco del grafo más de una vez.

Un grafo se denomina *conexo* si existe un camino entre cualquier par de nodos distintos del grafo. Por ejemplo, el grafo de la Figura A.2(a) es un grafo conexo. Los grafos que no poseen la propiedad de ser conexos están constituidos por la unión de dos o más subgrafos conexos que no comparten nodos. Estos subgrafos disjuntos se denominan las *componentes conexas* del grafo.

A.3.2. Grafos dirigidos

Un *grafo dirigido* (o *grafo orientado* o *digrapho*) se caracteriza por que los arcos se corresponden con pares ordenados de nodos (i.e., los arcos imponen una direccionalidad). Dado un arco a del grafo, si $f(a) = \langle u, v \rangle$, donde u y v son nodos, u se denomina nodo *inicial* y v nodo *terminal*. También se dice que v es *accesible desde u* , que v es un *sucesor* de u o bien que u es un *predecesor* de v . Los *multigrafos dirigidos* son grafos dirigidos que pueden contener arcos múltiples.

El concepto de camino definido para grafos no orientados se concreta de forma evidente para (multi)grafos dirigidos. Un *camino* de un nodo u a un nodo v es una secuencia de arcos a_1, a_2, \dots, a_n tales que $f(a_1) = \langle u, x_1 \rangle$, $f(a_2) = \langle x_1, x_2 \rangle, \dots, f(a_n) = \langle x_{n-1}, v \rangle$. Nuevamente, cuando el grafo no contiene arcos múltiples el camino queda únicamente determinado por la secuencia de nodos que atraviesa. En un (multi)grafo dirigido, un circuito recibe el nombre de *ciclo*.

Un grafo dirigido se denomina *fuertemente conexo* si existe un camino de u a v y de v a u para cualquier par de nodos u y v del grafo. Un grafo dirigido, aunque no sea fuertemente conexo todavía puede estar “hecho de una pieza”, si es débilmente conexo. Un grafo dirigido es *débilmente conexo* si el grafo no dirigido que resulta de eliminar el sentido de los arcos de un grafo dirigido es conexo.

Ejemplo A.12

Dado el grafo de la Figura A.2(b):

- El grado del nodo a es 5. Los nodos a y b están conectados. El nodo b es accesible desde a , por lo que a es un predecesor de b .
- El camino a, a, b, e, e es un camino de a a b de longitud 4. Este camino no es simple. El camino a, c, a es un ciclo.
- Es un grafo débilmente conexo, pero no es fuertemente conexo.

Aunque en este breve resumen hemos dejado de lado las propiedades de las distintas clases de grafos, todos los resultados obtenidos para los grafos simples son válidos para

grados dirigidos (sin bucles); solamente cambia el vocabulario empleado. La Tabla A.1 resume las características de los diferentes tipos de grafos.

Tabla A.1 Tipos de grafos.

Tipo	arcos	¿se permiten arcos múltiples?	¿se permiten bucles?
grafo simple	no dirigido	no	no
multigrafo	no dirigido	sí	no
pseudografo	no dirigido	sí	sí
grafo dirigido	dirigido	no	sí
multigrafo dirigido	dirigido	sí	sí

Finalmente, es importante decir que en muchas aplicaciones resulta conveniente etiquetar los arcos de los nodos con *pesos* (valores). Dichos pesos están relacionados con alguna noción de coste, por lo que ese tipo de grafos se utiliza para solucionar problemas de hallar caminos de coste mínimo.

A.3.3. Árboles

Un árbol es un caso particular de grafo conexo que no contiene circuitos simples. En un grafo conexo acíclico, puede convenir designar uno de los nodos como raíz. Una vez seleccionado un nodo como raíz, cobra sentido asignar una dirección a los arcos del grafo. Así, el grafo se convierte en un grafo dirigido que se denomina *árbol con raíz* (*rooted tree*, en la literatura inglesa).

La Figura A.3 proporciona una representación gráfica de un árbol, en la que los arcos ayudan a mostrar las relaciones de jerarquía existentes entre los elementos que componen el árbol (no obstante, en muchas ocasiones omitiremos la orientación de los arcos del árbol, que se dará por sobre entendida). Este tipo de representación recuerda a

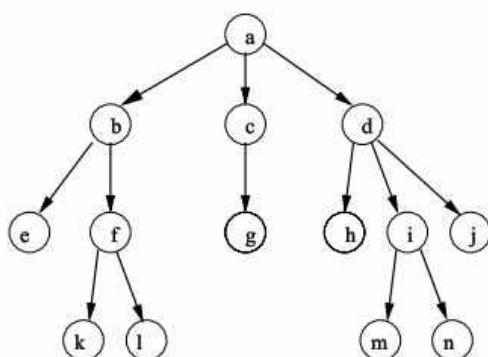


Figura A.3 Representación gráfica de un árbol 3-ario.

un “árbol” visto del revés (con la raíz en la parte superior y las hojas en la parte inferior), lo que justifica el nombre asignado a esta estructura de datos. No es de extrañar que la terminología sobre árboles tenga su origen en la botánica y en la genealogía. Todo árbol tiene un nodo distinguido que se denomina *nodo raíz*. Si v es un nodo, distinto del nodo raíz, el único nodo u para el que existe un arco directo entre u y v , se dice que es el *padre* de v . También se dice que v es un *nodo hijo* de u . Al número de hijos de un nodo se le llama *grado*. El *grado de un árbol* es el grado máximo de todos los nodos que lo componen. En un árbol se distinguen dos tipos de nodos los *nodos internos* y los *nodos hojas*. Los nodos hoja se caracterizan por no tener hijos mientras que los internos por tenerlos (observe, que el nodo raíz es comúnmente un nodo interno, pero si el árbol contiene un único nodo, el nodo raíz es un nodo hoja). Los hijos de un nodo forman el conjunto de sus *sucesores directos*. Los nodos con el mismo padre se dice que son *hermanos*.

La secuencia de nodos x_1, \dots, x_n , tal que, para todo $i \in \{1, \dots, n-1\}$, x_i es padre de x_{i+1} , se dice que es el (único) *camino* de x_1 a x_n . Cuando x_1 es el nodo raíz y x_n un nodo hoja, el camino se dice que es una *rama* del árbol. La *longitud* de un camino viene dada por el número de arcos que unen la secuencia de nodos que lo conforman (en otras palabras, el número de elementos de la secuencia menos uno). La *profundidad* de una nodo u es la longitud del (único) camino desde el nodo raíz al nodo u . El nodo raíz tiene profundidad cero. El conjunto de nodos situados a profundidad i , constituyen el *nivel i* del árbol. El máximo nivel de un árbol se dice que es su *profundidad o altura*. Dado un camino x_1, \dots, x_{i-1}, x_i , se dice que $\{x_1, \dots, x_{i-1}\}$ es el conjunto de los *ancestros* (o *antecesores*) de x_i . Los *descendientes* (o *sucesores*) de x_i son el conjunto de nodos que tienen a x_i por ancestro. Si u es un nodo de un árbol, el *subárbol* con raíz u , consiste en la porción del árbol original formada por el propio nodo u , todos sus descendientes y los arcos que los unen.

Ejemplo A.13

Dado el árbol de la Figura A.3:

- El nodo a es el nodo raíz, los nodos b, c, d, f e i son nodos internos y el resto nodos hoja.
- Los nodos h, i y j son hijos del nodo d . Todos ellos son hermanos. El grado del nodo d es tres, que es el grado del propio árbol.
- Los nodos b y a son los ancestros de e . Los nodos h, i, j, m y n son los descendientes de d .
- La secuencia d, i, n constituye un camino de longitud 2 entre d y n . El camino formado por los nodos a, d, i y n constituyen una rama del árbol. La profundidad del nodo n es 3.

- Los nodos k, l, m y n , forman el nivel 3 del árbol. La altura del propio árbol es 3.
- Los nodos b, e, f, k y l y los arcos que los unen, constituyen el subárbol con raíz b .

Bibliografía

- [1] Inc. AAIS Systems. AAIS-Prolog Reference Manual. Technical report: vesion M-2.0, Advanced A.I. Systems, Inc., 1988.
- [2] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, London, UK, 1996.
- [3] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, MA, 1991.
- [4] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A transformation system for lazy functional logic programs. In A. Middeldorp and T. Sato, editors, *Proc. of 4th Int'l Symp. on Functional and Logic Programming, FLOPS'99*, pages 147–162. Springer LNCS 1722, 1999.
- [5] M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.
- [6] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, Englewood Cliffs, NJ, 1997.
- [7] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass., 1990.
- [8] John Backus. Can logic programming be liberated from the von neumann's style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.
- [9] H. Barendregt. λ -calculus and Functional Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 321–363. Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990.
- [10] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.
- [11] K. Beck. *Extreme Programming Explained: Embrace Changes*. Addison–Wesley, 1999.

- [12] Marco Bellia and Giorgio Levi. The relation between logic and functional languages: A survey. *Journal of Logic Programming*, 3(3):217–236, 1986.
- [13] K. Benkerimi and J.W. Lloyd. A Procedure for the Partial Evaluation of Logic Programs. Technical Report TR-89-04, Department of Computer Science, University of Bristol, Bristol, England, May 1989.
- [14] K. Benkerimi and J.W. Lloyd. A Partial Evaluation Procedure for Logic Programs. In S. Debray and M. Hermenegildo, editors, *Proc. of the 1990 North American Conf. on Logic Programming*, pages 343–358. The MIT Press, Cambridge, MA, 1990.
- [15] M. Bidoit, H. J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella. Algebraic System Specification and Development. In *Lecture Notes in Computer Science*, volume 501. Springer-Verlag, Berlin, 1991.
- [16] A. Biermann. The Inference of Regular LISP Programs from Examples. *IEEE Transactions on Systems, Man and Cybernetics*, 8:585–600, 1978.
- [17] R. Bird. *Introducción a la Programación Funcional con Haskell*. Prentice-Hall, Madrid, 2000.
- [18] A. Bossi, N. Cocco, and S. Dulli. A Method for Specializing Logic Programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.
- [19] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19-20:149–197, 1994.
- [20] A. Bossi, M. Gabbrielli, G. Levi, and M.C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.
- [21] I. Bratko. *PROLOG: Programming for Artificial Intelligence*. Addison-Wesley, 1990.
- [22] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.
- [23] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.
- [24] R. Carnap. *Introduction to Semantics*. Harvard Uni. Press, 1942.
- [25] D. Chan and M. Wallace. A Treatment of Negation during Partial Evaluation. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 299–318. The MIT Press, Cambridge, MA, 1989.

- [26] Chin-Liang Chang and R. Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., 1973.
- [27] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, 1985.
- [28] K.L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and data bases*, pages 293–322. Plenum Press, New York, 1978.
- [29] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
- [30] W.F. Cloksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1987.
- [31] M. Comini, R. Gori, and G. Levi. Logic programs as specifications in the inductive verification of logic programs. In *Proceeding of Appia-Gulp-Prode'00, Joint Conference on Declarative Programming*, 2000. URL: <http://nutella.di.unipi.it/~agp00/AccptList.html>.
- [32] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
- [33] M. Comini, G. Levi, M.C. Meo, and G. Vitiello. Proving Properties of Logic Programs by Abstract Diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, pages 22–50. Springer LNCS 1192, 1996.
- [34] M. Comini, G. Levi, and G. Vitiello. Abstract Debugging of Logic Programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450. Springer-Verlag, Berlin, 1994.
- [35] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In J.W. Lloyd, editor, *Proc. of the 1995 Int'l Symp. on Logic Programming*, pages 275–287. The MIT Press, Cambridge, MA, 1995.
- [36] M. Comini, G. Levi, and G. Vitiello. Efficient detection of incompleteness errors in the abstract debugging of logic programs. In M. Ducassé, editor, *Proc. 2nd International Workshop on Automated and Algorithmic Debugging, AADEBUG'95*, 1995.
- [37] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

- [38] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
- [39] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. of Sixth ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.
- [40] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.
- [41] S.K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.
- [42] A.J.T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.
- [43] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, Berlin, 1996.
- [44] E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [45] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Meta-programming in logic programming. In Harvey Abramson and M. H. Rogers, editors, *Algorithmic debugging with assertions*, pages 501–522. The MIT Press, 1989.
- [46] H. Ehrg and B. Mahr. *Fundamentals of Algebraic Specification: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.
- [47] A.P. Ershov. On the Partial Computation Principle. *Information Processing Letters*, 6(2):38–41, 1977.
- [48] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A new Declarative Semantics for Logic Languages. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of Fifth Int'l Conf. on Logic Programming*, pages 993–1005. The MIT Press, Cambridge, MA, 1988.
- [49] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
- [50] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

- [51] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.
- [52] G. Ferrand. Error Diagnosis in Logic Programming, and Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.
- [53] G. Ferrand and A. Tessier. Declarative Debugging. *Computational Logic: The Newsletter of the European Network in Computational Logic*, 3(1):71–76, 1996.
- [54] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, Reading, MA, 1987.
- [55] M. Fitting. *First-Order Logic and Automated Theorem Proving (2nd ed.)*. Springer-Verlag, 1996.
- [56] P. Flach. *Simply Logical: Intelligent Reasoning by Example*. John Wiley & Sons, 1994.
- [57] P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects.
- [58] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
- [59] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.
- [60] J.P. Gallagher. Transforming Logic Programs by Specializing Interpreters. In *Proc. of 7th European Conf. on Artificial Intelligence ECAI'86*, pages 109–122, 1986.
- [61] M. Garrido. *Lógica simbólica*. Tecnos, 1997.
- [62] R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In *Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'96*, pages 152–166. Springer LNCS 1140, 1996.
- [63] W. Grassmann and J.P. Tremblay. *Matemática Discreta y Lógica*. Prentice Hall, 1998.
- [64] C.A. Gunter. *Semantics of Programming Languages*. The MIT Press, Cambridge, MA, 1992.
- [65] A. G. Hamilton. *Lógica para Matemáticos*. Paraninfo, 1981.

- [66] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [67] C.J. Hogger. Derivation of Logic Programs. *ACM*, 28(2):372–392, 1981.
- [68] P. Hudak. Conception, Evolution and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, Sept. 1989.
- [69] D. Jackson and J. Wing. Lightweight Formal Methods. *IEEE Computer*, 29(4):22–23, 1996.
- [70] N.D. Jones. Automatic Program Specialization: A Re-Examination from Basic Principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, Amsterdam, 1988.
- [71] N.D. Jones. What *not* to Do When Writing an Interpreter for Specialisation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 216–237. Springer LNCS 1110, 1996.
- [72] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
- [73] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, Sept. 1996.
- [74] P. Julián-Iranzo. *Especialización de Programas Lógico-Funcionales Perezosos*. PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Mayo 2000.
- [75] P. Julián-Iranzo. *Lógica Simbólica para Informáticos*. Editorial RA-MA, 2004.
- [76] B.W. Kernighan and D.M. Ritchie. *El Lenguaje de Programación C*. Prentice-Hall Hispanoamericana, 1991.
- [77] S.C. Kleene. *Introduction to Metamathematics*. D. van Nostrand, Princeton, NJ, 1952.
- [78] H.J. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proc. of 9th ACM Symp. on Principles of Programming Languages*, pages 255–267, 1982.
- [79] R. Korf. Depth-first iterative deepening: An optimal admissible tree search algorithm. *Artificial Intelligence*, 27(1):97–109, 1985.

- [80] H.F. Korth and A. Silberschatz. *Fundamentos de Bases de Datos*. McGraw-Hill, 1993.
- [81] R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979. Traducción al español: Lógica, programación e Inteligencia Artificial. Diaz de Santos, Madrid, 1986.
- [82] R.A. Kowalski. Predicate Logic as a Programming Language. In *Information Processing 74*, pages 569–574. North-Holland, 1974.
- [83] R.A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
- [84] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In G. Levi, editor, *Proc. of the Static Analysis Symposium, SAS'98*, pages 230–245. Springer LNCS 1503, 1998.
- [85] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming, JICSLP '96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.
- [86] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
- [87] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. Technical Report CS-87-09, Computer Science Department, University of Bristol, 1987. Revised version 1989, in [88].
- [88] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
- [89] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of Fifth Int'l Conf. on Logic Programming*, pages 1006–1023. The MIT Press, Cambridge, MA, 1988.
- [90] K. Marriott and H. Søndergaard. Semantics-based Dataflow Analysis of Logic Programs. In G. Ritter, editor, *Information Processing 89*. North-Holland, 1989.
- [91] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122:97–117, 1994.
- [92] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. Technical Report CSTR-94-16, Computer Science Department, University of Bristol, December 1994.

- [93] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.
- [94] B. Mates. *Lógica Matemática Elemental*. Tecnos, S.A., 1974.
- [95] W. McCune. 33 basic test problems: A practical evaluation of some paramodulation strategies. In R. Veroff, editor, *Automated Reasoning and Its Applications, Essays in Honor of Larry Wos*, pages 71–114. The MIT Press, Cambridge, MA, 1997.
- [96] M. McNamara and Y. Smaragdakis. Functional programming in c++. In J.W. Lloyd, editor, *Proc. of 5th International Conference on Functional Programming, ICFP'2000*, pages 118–129. ACM Press, 2000.
- [97] D. Le Métayer, Valérie-Anne Nicolas, and Oliver Ridoux. Exploring the software development trilogy. *IEEE Software*, 6:75–81, 1998.
- [98] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [99] Jesús Mosterín. *Lógica de Primer Orden*. Ariel, 1983.
- [100] L. Naish. A Declarative Debugging Scheme. Technical report 95/1, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1995.
- [101] L. Naish and T. Barbour. Declarative Debugging of a Logical-Functional Language. Technical report 94/30, Department of Computer Science, University of Melbourne, Melbourne, Australia, December 1994.
- [102] Lee Naish. Declarative Debugging of Lazy Functional Programs. *Australian Computer Science Communications*, 15(1):287–294, 1993.
- [103] A. Newell. The Knowledge Level. *ai*, 18(1):87–127, 1982.
- [104] Nils J. Nilsson. *Principios de Inteligencia Artificial*. Ediciones Díaz de Santos, 1987.
- [105] Nils J. Nilsson. *Inteligencia Artificial: Una Nueva Síntesis*. McGraw-Hill Interamericana de España, 2001.
- [106] M. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.
- [107] M. J. O'Donnell. *Equational Logic as a Programming Language*. The MIT Press, Cambridge, MA, 1985.

- [108] M. J. O'Donnell. Equational Logic Programming. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5 on Logic Programming, Chapter 2. Oxford Science Publications, 1994.
- [109] R. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.
- [110] F. Orejas. Definición Semántica en Lenguajes de Programación. *Novática*, VII(38/39):28–34, 1981.
- [111] F. Orejas. Lenguajes de Programación para los Computadores de la 5 Generación. *Mundo Electrónico*, (158):55–66, 1986.
- [112] Coordinateur P. Schnoebelen. *Verification de logiciels. Techniques et outils du model-checking*. Vuibert Informatique, Paris, 1999.
- [113] C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In M.S. Paterson, editor, *Proc. of 17th Int'l Colloquium on Automata, Languages and Programming*, pages 386–399. Springer LNCS 443, 1990.
- [114] R. Peña-Marí. *Diseño de Programas: Formalismo y Abstracción*. Prentice Hall, Englewood Cliffs, NJ, 2005.
- [115] A. Petrorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
- [116] A. Petrorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 355–385. Springer LNCS 1110, 1996.
- [117] A. Petrorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
- [118] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, Englewood Cliffs, N.J., 1987.
- [119] S.J. Peyton-Jones, J-M. Eber, and J. Sewar. Composing contracts: and adventure in financial engineering. In *Proc. of Fifth Int'l Conf. on Functional Programming, ICFP'2000*, pages 280–292. ACM Press, 2000.
- [120] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, MA,, 1993.
- [121] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

- [122] T.W. Pratt and M.V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice Hall, Englewood Cliffs, N.J., 1996.
- [123] M. Proietti and A. Pettorossi. The Loop Absorption and the Generalization Strategies for the Development of Logic Programs and Partial Deduction. *Journal of Logic Programming*, 16(1&2):123–161, 1993.
- [124] C. Reade. *Elements of Functional Programming*. Addison-Wesley, Reading, MA,, 1993.
- [125] R. Reiter. On Closed World Databases. In H. Gallaire and J. Minker, editors, *Logic and data bases*, pages 55–76. Plenum Press, New York, 1978.
- [126] E. Rich and K. Knight. *Inteligencia Artificial*. McGraw-Hill, 1994.
- [127] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.
- [128] J.A. Robinson. Logic and Logic Programming. *Communications of the ACM*, 35(3):41–65, 1992.
- [129] K.R. Rosen. *Discrete Mathematics and its Applications*. MacGraw-Hill, 1991.
- [130] P. Ross. *Advanced Prolog: Techniques and Examples*. Addison–Wesley, Reading, MA, 1989.
- [131] Peter Van Roy. 1983–1993: The wonder years of sequential prolog implementation. *Journal of Logic Programming*, 19,20:385–441, 1994.
- [132] S. Safra and E. Shapiro. Meta Interpreters for Real. In H.-J. Kugler, editor, *Information Processing 86, Dublin, Ireland*, pages 271–278. North-Holland, Amsterdam, 1986.
- [133] D. Scott. *Outline of a Mathematical Theory of Computation*. Technical Monograph PRG-2. Oxford University Computing Laboratory, November 1970.
- [134] D. Scott. Domains for Denotational Semantics. In *Proc. of ICALP*. Springer LNCS 140, 1982.
- [135] H. Seki. Unfold/fold Transformation of General Logic Programs for the Well-Founded Semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.
- [136] E.Y. Shapiro. Algorithmic Program Debugging. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM, New York, 1982.
- [137] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, MA, 1983.

- [138] E.Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, Tokyo, 1983.
- [139] L. Sterling and E. Shapiro. *The Art of Prolog (Second Edition)*. The MIT Press, Cambridge, MA, 1994.
- [140] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, MA, 1977.
- [141] M. Tofte. *Compiler Generators: What They Can Do, What They Might Do, and What They Will Probably Never Do*, volume 19 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.
- [142] J. Wielemaker. SWI-Prolog 4.0 Reference Manual. Technical report: version 4.0.11 Dec. 2001, University of Amsterdam, 2001.
- [143] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, Cambridge, MA, 1996.
- [144] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 674–788. Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990.
- [145] N. Wirth. *Algoritmos + Estructuras de Datos = Programas*. Ediciones del Castillo, 1980.
- [146] L. Wos and W. McCune. Automated Theorem Proving and Logic Programming: A Natural Symbiosis. *Journal of Logic Programming*, 11:1–53, 1991.
- [147] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introducción and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1984.

Índice alfabético

- s*-semántica, 157
 - éxitos básicos, 146
- lógica ecuacional, 412
- A* (algoritmo), 374
 - ajuste de patrones, 17
 - alcance de un cuantificador, 36
 - alfabeto, 33
 - álgebra de Herbrand, 80
 - algoritmo A*, 374
 - algoritmo de unificación, 106
 - análisis estático, 404
 - aprendizaje de programas, 406
 - aproximación ágil (*lightweight*), 395, 397, 398
- árbol, 210
 - altura, 450
 - antecesor, 450
 - búsqueda (de), 214
 - camino, 450
 - equilibrado, 213, 231
 - frontera, 231
 - grado, 450
 - n*-ario, 211
 - nivel, 450
 - nodo hermano, 450
 - nodo hijo, 450
 - nodo hoja, 450
 - nodo interno, 450
 - nodo padre, 450
 - nodo raíz, 450
 - rama, 450
 - subárbol, 450
 - sucesor, 450
 - árbol de búsqueda de Prolog, 176
 - árbol de búsqueda SLD
 - de fallo finito, 137
 - nodo de fallo, 137
 - nodo hijo, 137
 - nodo hoja, 137
- nodo padre, 137
 - rama de éxito, 137
 - rama de fallo, 137
 - rama infinita, 137
- árbol de exploración, 374
 - árbol semántico, 86
 - árbol SLD, 136
 - aridad, 34
 - átomo, 35
 - axiomas de \mathcal{K}_L , 37
- Backus-Naur Form*, 45
 - base de datos interna, 178, 255
 - búsqueda indeterminista, 11
- código con demostración asociada, 410
 - cálculo deductivo, 32
 - cálculo deductivo \mathcal{K}_L , 37
 - cierra
 - reflexivo, 446
 - reflexivo y transitivo, 446
 - simétrico, 446
 - transitivo, 446
 - dáusula
 - factor (de), 109
 - factor unitario (de), 109
 - clase de equivalencia, 446
 - dáusula, 75
 - antecedente (de una), 128
 - cabeza (de una), 128
 - central, 115
 - conclusión (de una), 128
 - condición (de una), 128
 - consecuente (de una), 128
 - cuerpo (de una), 128
 - de Horn, 128
 - definida, 128
 - falsa en \mathcal{I} , 83
 - general, 128
 - indefinida, 128

- inicial, 115
- instancia básica, 83
- lateral, 115
- objetivo, 128, 129
- subsumida, 114
- unitaria, 75
- vacía, 75
- verdadera en \mathcal{I} , 83
- visión declarativa, 127
- visión operacional, 127
- visión procedimental, 127
- cláusula de programa, 128
- cláusulas padres, 98
- compilación certificante, 410
- compilador, 48
- completitud, 44, 200
- Composicionalidad-OR, AND, 155
- comprobaciones de dominio, 187
- computador, 1
 - bus del sistema, 3
 - memoria central (MC), 3
 - organización, 1
 - unidad central de proceso (UCP), 2
- conclusión, 38
- condicional recíproco, 127
- conjunto
 - cardinalidad de un, 445
 - ociente, 446
 - definición, 441
 - finito, 445
 - imagen de un, 444
 - infinito no numerable, 445
 - infinito numerable, 445
 - partes de un, 442
 - pertenencia a, 441
 - potencia de un, 442
 - producto cartesiano, 442
 - subconjunto de un, 442
 - subconjunto propio de un, 442
 - vacio, 441
- conjunto adecuado (de conectivas), 69
- conjunto de soporte, 115
- conjunto unificable, 106
- conjuntos, 195
 - diferencia de, 442
 - disjuntos, 442
 - igualdad de, 442
- intersección de, 442
- unión de, 442
- conocimiento, 295
 - de control, 297
 - declarativo, 297
 - global, 371
 - local, 371
 - procedimental, 297
- conocimiento y lógica, 298
- consecuencia lógica, 43
- consistencia, 43
- consistente, 43
- constante de Skolem, 73
- constructores, 184
- contexto, 39
- contradicción, 68
- contradicción, 41
- control
 - irrevocable, 367
 - tentativo, 355, 367
- Corrección, 44
- corrección, 200
- corte, 168, 234
 - rojo, 237
 - verde, 237
- CPD, véase deducción parcial conjuntiva
- cripto-aritmética, 291
- cuantificador, 305
 - existencial, 305
 - universal, 305
- cuello de botella de von Neumann, 4
- datos parcialmente definidos, 11, 186
- decidibilidad, 44
- deducción, 38, 100
 - lineal, 115
- deducción automática, 61
- deducción parcial, 425
 - completitud, 426
 - condición de cierre en la, 426
 - condición de independencia en la, 426
 - corrección, 426
 - resultante, 425
 - terminación, 426
- deducción parcial conjuntiva, 427
- deducible, 38
- demostrablemente equivalente, 67

- demostración automática de teoremas, 61
aproximaciones, 62
- demostración, 38
- depuración de programas Prolog, 181
- derivable, 38
- derivación SLD, 132
longitud, 133
- descenso iterativo, 363
- desplegado, 195, 412
- dominio, 51
- don't know built-in search*, véase búsqueda indeterminista
- efecto lateral, 13, 233
- enlace, 102, 179
- entrada/salida adireccional, 185
- enunciado, 37
- espacio de búsqueda, 63
- espacio de estados, 346
búsqueda (en un), 345
- espacio de trabajo, 178, 255
- especialización, 415
- estandarización de las variables, 133
- estrategia de búsqueda
anchura (en), 358
coste uniforme (de), 391
descenso iterativo, 363
primero el mejor, 373
profundidad (en), 356
tentativa, 374
- estrategia de control, 346
- estrategia de reducción, véase modo de evaluación
- estrategia de resolución, 113
amplitud (en), 113
anchura (en), 113
borrado, 113
conjunto de soporte, 115
entrada, 117
lineal, 115
preferencia por cláusulas unitarias, 117
saturación de niveles (por), 113
semántica, 114
- estrategias de búsqueda, 354
a ciegas, 363
heurística, 364
sin información, 363
- estructura deductiva, 37
- estructura, 173
- eureka, 414
- evaluación parcial, 415
en la prog. lógica, véase deducción parcial
- evaluador parcial, 415, 416, 421
offline, 418
online, 418
autoaplicable, 422
- explosión combinatoria, 89, 365, 370
- factor de una cláusula, véase cláusula
- forma clausal, 62, 64, 72, 76
existencia, 316
negación, 317
representación en, 309
- forma estándar, 75
- forma normal, 67, 75
conjuntiva, 69
disyuntiva, 69
prenexa, 70
Skolem (de), 72
- formal, 31
- formalismo, 31
- formalización, 298
- fórmula
atómica, 34, 35
básica, 35
bien formada, 34
cerrada, 36
existencial, 71
universal, 71
- fuertemente basados en tipos, 13
- función, 443
n-ádica, 444
n-aria, 444
biyectiva (biyección), 444
evaluación (de), 371
heurística, 364
idempotente, 445
identidad, 445
imagen de un elemento, 444
inversa, 445
inyectiva, 444
parcial, 444
punto fijo de una, 445
rango de una, 444

- selección (de), 132
- Skolem (de), 73
- sobreyectiva, 444
- funciones de correspondencia, 296
- funciones-memo, 257, 272
- generación automática de programas, 421
- generar y comprobar, 262
- gestión automática de la memoria, 11
- grafo, 447
 - camino, 447, 448
 - ciclo, 448
 - circuito, 448
 - circuito simple, 448
 - conexo, 448
 - débilmente conexo, 448
 - dirigido, 448
 - exploración (de), 374
 - fuertemente conexo, 448
 - multigrafo, 447
 - pseudografo, 447
 - simple, 447
- granularidad, 304
- hecho, 129
- Herbrand
 - base (de), 79
 - interpretación (de), 80
 - modelo (de), 84
 - universo (de), 78
- herencia, 330
 - múltiple, 331
- heurística, 297, 364
 - búsqueda del gradiente, 372
 - escalada por la máxima pendiente, 372
 - escalada simple, 372
 - local, 366, 371
 - método de escalada, 371
 - vecino más próximo (del), 366
- implicación inversa, 127
- inconsistencia, 44
- indecidibilidad, 44
- indecidible, 44
- indeterminismo
 - don't care*, 138
- inducción estructural, 187
- ínfimo, 82
- información heurística, 363
- ingeniería del software automática, 395, 398
- insatisfacible, 41, 43
- instancia, 102
 - computada, 135
- inteligencia artificial, 295
- interpretación, 16, 32
- inversibilidad, 185, 188
- lenguaje, 31
 - formal, 32, 33
- lenguajes convencionales, véase lenguajes imperativos
- lenguajes imperativos, 5, 8
 - desventajas, 5
 - ventajas, 9
- lista, 188
 - cabeza (de), 188
 - cola (de), 188
- listas diferencia, 273
- literal, 35
- literales complementarios, 35
- lógica de predicados, 33
- lógica ecuacional, 16
- lógica y control, 5, 9
- lógicamente equivalente, 67
- máquina virtual, 47, 48
- marcos, 322, 342
- matriz, 70
- merge-sort, 229
- método de escalada, 371
- método de la multiplicación, 88
- mezcla ordenada, 229
- modelo, 39
- modelo mínimo de Herbrand, 148
- modelo von Newmann, 1
- modo de evaluación
 - impaciente, 15
 - perezoso, 15
- movimientos, 346
- naturales, 184
- negación, 244
- negación como fallo, 247
- nivel
 - del conocimiento, 296
 - simbólico, 296

- nodo, 211
notación clausal, 128
n-tupla, 442
objetivo, 11, 128, 129, 131
objeto, 173
 estucurado, 173
observable, 146
occur check, 107
occur check, 168, 176
 problema (del), 220
ocurrencia, 36
 libre, 36
 ligada, 36
operador, 173, 207
 ;, 179
 aritmético, 203
 de comparación, 206
 de consecuencias lógicas inmediatas, 151
 predefinido, 168
orden de reducción
 aplicativo, 15
 normal, 15
orden estándar, 207
orden superior, 14, 168, 257
ordenación rápida, 229
parámetro de acumulación, 193
paramodulación, 64
patrón, 187
pattern matching, véase ajuste de patrones
Peano
 notación (de), 184
 postulados (de), 184
plegado, 412
polimorfismo, 13
predicados extralógicos, 233
prefijo, 70
preinterpretación, 80
premisas, 38
problema
 búsqueda de un camino en un grafo (de la), 359
 celos en el Puerto (de los), 94, 228, 284
 coloreado de mapas (del), 340
 contenedores de agua (de los), 350
 cuerpos celestes (de los), 94, 225
 mini-sudoku (del), 263, 290
 misioneros y caníbales (de los), 390
 mono y el plátano (del), 348
 mundo de los robots (del), 336, 392
 ocho reinas (de las), 265, 290, 389
 planificación en el mundo de los bloques, 352
 puzzle-8 (del), 392
 relaciones familiares (de las), 201, 224
 sudoku (del), 290
 tablero de ajedrez recortado (del), 340
 torres de Hanoi (de las), 229, 286
 viajante de comercio (del), 367
procedimiento de prueba por refutación SLD, 141
procesador, 2
producto cartesiano, 442
programa definido, 130
programación lógica inductiva, 427
programación
 declarativa, 9
 declarativa (aplicaciones), 24
 funcional, 12
 algebraica, 16
 clásica, 16
 ecuacional, 16
 imperativa, 5
 lógica, 10
 orientada a objetos, 333
Prolog, 167, 233
 con aritmética, 203
 puro, 169
propagación de la información, 426
propiedad de intersección de modelos, 147
prototipado automático, 400
prueba, 100
 estructural, 405
 por deducción indirecta, 66
 por reducción al absurdo, 66
 por refutación, 64, 66
pruebas funcionales, 406
punto de elección, 177
punto de vuelta atrás, 177
punto fijo, 445
quicksort, 229
radio de acción de un cuantificador, 36
razonamiento